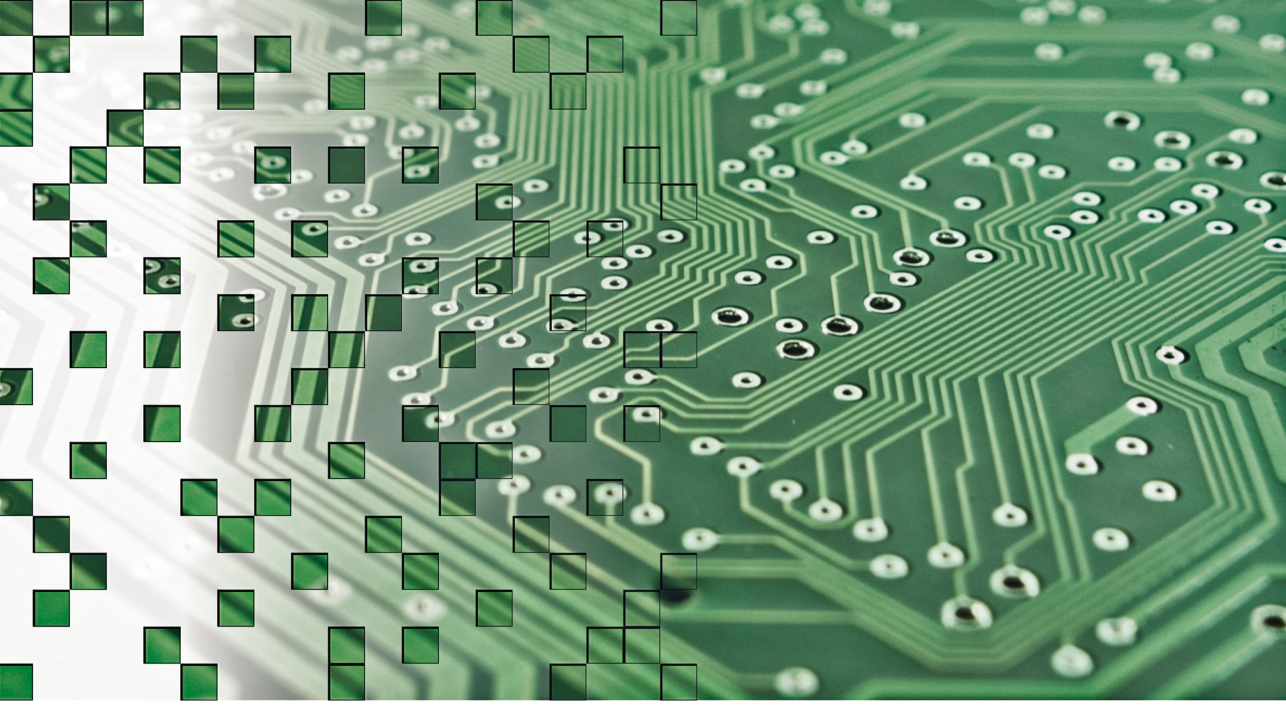


FAU Forschungen, Reihe B, Medizin, Naturwissenschaft, Technik 21

Moritz Kreutzer

Performance Engineering for Exascale-Enabled Sparse Linear Algebra Building Blocks

FAU UNIVERSITY PRESS

Moritz Kreutzer

Performance Engineering for Exascale-Enabled Sparse Linear Algebra Building Blocks

FAU Forschungen, Reihe B
Medizin, Naturwissenschaft, Technik
Band 21

Herausgeber der Reihe:
Wissenschaftlicher Beirat der FAU University Press

Moritz Kreutzer

Performance Engineering for Exascale-Enabled Sparse Linear Algebra Building Blocks

Erlangen
FAU University Press
2018

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Der vollständige Inhalt des Buchs ist als PDF über den OPUS Server der Friedrich-Alexander-Universität Erlangen-Nürnberg abrufbar:
https://opus4.kobv.de/opus4-fau/home

Verlag und Auslieferung:

FAU University Press, Universitätsstraße 4, 91054 Erlangen

Druck: docupoint GmbH

ISBN: 978-3-96147-103-4 (Druckausgabe)
eISBN: 978-3-96147-104-1 (Online-Ausgabe)
ISSN: 2198-8102
DOI: 10.25593/978-3-96147-104-1

Performance Engineering for Exascale-Enabled Sparse Linear Algebra Building Blocks

---

Performance-Engineering für exascalefähige Grundbausteine linearer Algebra mit dünn besetzten Matrizen

Der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur
Erlangung des Doktorgrades Dr.-Ing.
vorgelegt von

Moritz Kreutzer

aus Sonneberg

Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung:	18. Dezember 2017
Vorsitzender des Promotionsorgans:	Prof. Dr.-Ing. Reinhard Lerch
Gutachter:	Prof. Dr. Gerhard Wellein Prof. Dr. rer. nat. Dominik Göddeke

In memory of Philipp.

Abstract

The increasing demand for solving larger and more complex problems in computational science and engineering is a major driving factor to deploy computer systems with ever-advancing performance capabilities. To increase the available performance, modern High-Performance Computing (HPC) platforms come with multiple levels of parallelism, complex memory hierarchies, heterogeneous architectures, and extreme scales. To match the need for sustainable and efficient software under these premises, special value has to be attached to the inherent challenges like efficiency on all scales and performance portability across heterogeneous architectures. This work addresses the development of high-performance scientific software for sparse linear algebra, which is an important field of research and forms the foundation of many applications of computational science and engineering, with a special focus on sparse eigenvalue solvers on current and future supercomputers.

One of the most prominent building blocks of sparse linear algebra is Sparse Matrix-Vector Multiplication (SpMV). The irregular nature of this kernel, the fact that its execution is usually strongly limited by main memory bandwidth, and the strong dependence of the optimal implementation on the underlying hardware architecture open a wide field of opportunities for analysis and tuning. In this work, a platform-agnostic storage format for high-performance general SpMV is developed: SELL-C-σ. Based on existing, device-specific formats, its design is justified and best practices for the selection of tuning parameters are provided. Benchmark experiments demonstrate the high efficiency and performance portability of this unified format for SpMV on all relevant HPC hardware architectures, including three generations of Intel multi-core CPUs, two generations of Intel many-core CPUs, and two generations of NVIDIA GPUs. Based on a set of more than 20 test matrices from manifold application domains, it is demonstrated that SpMV using a unified SELL-C-σ matrix usually not only yields competitive performance but even surpasses device-specific formats and implementations for many test cases.

Sparse linear algebra algorithms can often be formulated using blocks of vectors instead of single vectors. This technique is in some cases motivated by numerical benefits, but it also contains performance optimization potential by mitigating the strong limitation imposed by main memory bandwidth. Vector blocking requires the development of further software components like Sparse Matrix-Multiple Vectors Multiplication (SpMMV) and multiplication operations of block vectors. In this work, the shift of hardware bottlenecks is analyzed and highly efficient implementation of block vector kernels are presented. Significant performance deficiencies of established software libraries for block vector multiplications are revealed and average speedups in the range of $4\times$ with a maximum factor of $27\times$ for such operations are demonstrated.

Kernel fusion is employed in the present work on top of optimized basic building blocks, leading to custom compute kernels for sparse linear algebra algorithms. Several of those kernels are analyzed in this work, and it is demonstrated that speedups in the range of $2\times$ to $5\times$ for a Kernel Polynomial Method (KPM) implementation can be achieved by a combination of vector blocking and kernel fusion on all considered hardware architectures.

The performance engineering process consistently implemented for all software development efforts in this work – including code analysis, benchmarking, the evaluation of hardware metrics, and code optimization – is closely guided by performance models. Performance modeling is an indispensable ingredient for the development of high-performance software components, as it helps in revealing hardware bottlenecks, identifying suitable optimization techniques, and assessing the efficiency of a present implementation on a given hardware platform. To that end, the Roofline performance model is identified as a suitable tool, refined, and applied in this work.

To be accessible by a broader community, all software development efforts conducted within this work are combined into the scalable open-source software library GHOST. Being based on MPI+X parallelism with truly heterogeneous data-parallel execution and a holistic view of applications, algorithms, and implementations, it delivers a unique feature set for highly performant sparse linear algebra on current and future supercomputers. Several eigenvalue solvers for sparse matrices are implemented based on GHOST, including a simple Lanczos method, polynomial filtering techniques like Chebyshev Filter Diagonalization (ChebFD) and KPM, and a Block Jacobi-Davidson QR (BJDQR) method. Those solvers represent different classes of methods, each addressing a different class of target eigenvalues.

To demonstrate the applicability of the developed software components, full-application performance of such solvers for real-world problems on some of the world's largest supercomputers with completely different hardware architectures – including homogeneous multi-core CPU clusters, GPU-accelerated clusters, and self-hosted many-core CPU clusters – is presented. Combining all of the developed building blocks and techniques, solutions to the standard eigenvalue problem for relevant quantum physics applications, addressing novel materials like topological insulators, are acquired. At the largest scale, matrices with up to 26 billion rows and 7 terabytes of raw data are investigated on thousands of homogeneous and heterogeneous compute nodes with hundreds of TFLOP/s sustained performance and verifiable high efficiency from the single node to the extreme scale.

Zusammenfassung

Der steigende Bedarf zur Lösung größerer und komplexerer Probleme im Bereich des wissenschaftlichen Rechnens ist ein treibender Faktor für die Entwicklung und den Einsatz von Hochleistungsrechnern mit stetig wachsenden Hardware-Fähigkeiten. Wichtige Maßnahmen zur Steigerung der verfügbaren Leistung in modernen Systemen des Hochleistungsrechnens (High-Performance Computing, HPC) umfassen die Einführung mehrerer Parallelitätsebenen, komplexer Speicherhierarchien und heterogener Architekturen, sowie eine stetige Vergrößerung der HPC-Systeme. Um dem Bedarf nach nachhaltiger und effizienter Software unter diesen Voraussetzungen gerecht zu werden, muss den inhärenten Herausforderungen wie Effizienz auf allen Skalen und Performance-Portabilität für heterogene Architekturen Beachtung geschenkt werden. Diese Dissertation widmet sich der Entwicklung hoch-performanter wissenschaftlicher Software für lineare Algebra mit dünn besetzten Matrizen, welche ein bedeutendes Forschungsgebiet und die Grundlage für viele Anwendungen wissenschaftlichen Rechnens darstellt. Im Speziellen steht die Entwicklung von Eigenwert-Lösern für dünn besetzte Matrizen auf aktuellen und zukünftigen Hochleistungsrechnern im Fokus.

Eine der bedeutendsten Grundoperationen auf dem Gebiet der dünn besetzten linearen Algebra ist die Multiplikation einer dünn besetzten Matrix mit einem Vektor (Sparse Matrix-Vector Multiplication, SpMV). Der irreguläre Charakter dieser Operation, die üblicherweise starke Limitierung ihrer Ausführungsgeschwindigkeit durch die Hauptspeicher-Bandbreite und der starke Einfluss der zugrundeliegenden Hardware-Architektur auf die optimale Implementierung eröffnen ein breites Spektrum an Möglichkeiten zu Analyse und Performance-Verbesserung. Ein Teil dieser Arbeit befasst sich mit der Entwicklung eines plattformunabhängigen Datenformats für hocheffiziente allgemeine SpMV: SELL-C-σ. Das Design dieses Datenformats wird auf Basis existierender Datenformate begründet und Vorschläge zur Wahl von Optimierungsparametern werden unterbreitet. Benchmark-

Experimente beweisen die hohe Effizienz und Performance-Portabilität dieses vereinheitlichten Formats auf allen relevanten HPC-Architekturen, darunter drei Generationen von Intel Mehrkernprozessoren, zwei Generationen von Intel Vielkernprozessoren und zwei Generationen von NVIDIA Grafikprozessoren. Basierend auf einer Auswahl von über 20 Test-Matrizen aus unterschiedlichsten Anwendungsfeldern wird gezeigt, dass SELL-C-σ nicht nur konkurrenzfähig, sondern oftmals sogar effizienter als plattformspezifische Datenformate und Implementierungen in Bezug auf SpMV-Performance ist.

Algorithmen dünn besetzter linearer Algebra können häufig mit Blöcken von Vektoren anstatt einzelner Vektoren formuliert werden. Dieses Vorgehen ist in manchen Fällen durch numerische Vorteile motiviert, birgt jedoch auch Potential zur Steigerung der Performance durch Abschwächung der starken Limitierung von der Hauptspeicher-Bandbreite. Block-Vektoren erfordern die Entwicklung weiterer Software-Komponenten, darunter die Multiplikation einer dünn besetzten Matrix mit einem solchen Block-Vektor (Sparse Matrix-Multiple Vectors Multiplication, SpMMV) und Multiplikationen von Block-Vektoren. Diese Arbeit analysiert die durch Block-Formulierungen hervorgerufene Veränderung der limitierenden Hardware-Ressourcen und präsentiert hocheffiziente Implementierungen von Block-Vektor-Operationen. Erhebliche Performance-Defizite etablierter Software-Bibliotheken für Block-Vektor-Multiplikationen werden aufgezeigt, auf deren Basis durchschnittliche Performance-Verbesserungen im Bereich von $4\times$ mit maximalen Faktoren von $27\times$ demonstriert werden können.

Neben optimierten Grundbausteinen wird in der vorliegenden Arbeit von der Fusion elementarer Berechnungs-Routinen Gebrauch gemacht, welche zu spezifischen Routinen für Algorithmen dünn besetzter linearer Algebra führt. Einige dieser Routinen werden in dieser Arbeit analysiert, darunter eine Implementierung der “Kernel Polynomial Method” (KPM), für welche Performance-Verbesserungen im Bereich von $2\times$ bis hin zu $5\times$ auf allen betrachteten Hardware-Architekturen erreicht werden können. Diese Beschleunigungsfaktoren sind das Resultat einer Kombination von Block-Vektoren und fusionierten Routinen.

Der Performance-Engineering-Prozess – einschließlich Code-Analyse, Benchmarking, der Evaluierung von Hardware-Metriken und Code-Optimierung – orientiert sich an Performance-Modellen und findet konsequente Anwendung im Rahmen der Software-Entwicklung in dieser Arbeit. Performance-Modellierung ist ein unverzichtbarer Bestandteil in der

Entwicklung hocheffizienter Software-Komponenten, denn sie erlaubt es, limitierende Hardware-Ressourcen aufzudecken, geeignete Optimierungsstrategien zu bestimmen und die Effizienz einer Implementierung auf einer gegebenen Hardware-Plattform zu beurteilen. Zum Zweck der Performance-Modellierung wird das Roofline-Modell in dieser Arbeit als ein geeignetes Instrument identifiziert, auf Basis der relevanten Operationen angepasst und angewandt.

Um einer breiteren Öffentlichkeit Zugang zu gewähren, werden alle im Rahmen dieser Arbeit entwickelten Software-Komponenten in der skalierbaren und quelloffenen Software-Bibliothek GHOST zur Verfügung gestellt. Ein heterogener, auf MPI+X basierender datenparalleler Ansatz und eine holistische Sicht auf Anwendungen, Algorithmen und Implementierungen ermöglichen die Bereitstellung einzigartiger Funktionalität zur hocheffizienten Ausführung von Algorithmen dünn besetzter linearer Algebra auf aktuellen und zukünftigen Hochleistungsrechnern. GHOST-basierte Implementierungen mehrerer Eigenwert-Löser für dünn besetzte Matrizen stehen zum Download zur Verfügung, darunter eine einfache Lanczos-Methode, polynomiale Filter-Methoden wie die KPM und Chebyshev-Filterdiagonalisierung (ChebFD) sowie Block Jacobi-Davidson QR (BJDQR). Diese Löser repräsentieren verschiedene Klassen von Methoden für verschiedene Klassen von Eigenwertproblemen.

Die Anwendbarkeit der entwickelten Software-Komponenten wird durch Experimente auf Basis realer Probleme aus der Praxis demonstriert. Hierfür werden einige der größten Hochleistungsrechner der Welt mit gänzlich unterschiedlichen Hardware-Architekturen verwendet, darunter homogene Systeme mit Mehrkernprozessoren sowie gesockelten Vielkernprozessoren, als auch heterogene, mit Grafikprozessoren beschleunigte, Systeme. Basierend auf den entwickelten Software-Bausteinen und Methoden werden Eigenwertprobleme für aktuelle Anwendungen der Quantenphysik gelöst, darunter die Untersuchung neuartiger Materialien wie topologischer Isolatoren. Die größten Experimente beinhalten dünn besetzte Matrizen mit bis zu 26 Milliarden Zeilen und 7 Terabyte Rohdaten auf tausenden von homogenen und heterogenen Rechenknoten, wobei hunderte TFLOP/s Performance mit verifizierbar hoher Effizienz vom einzelnen Rechenknoten bis hin zu extremem Skalen erreicht werden können.

Acknowledgments

First and foremost, I would like to express my gratitude towards my supervisor Prof. Gerhard Wellein for giving me the opportunity to work in his group and providing exceptional support, supervision, and guidance during the past six years.

Special thanks go to Georg Hager for always having an open door, answering countless questions and delivering valuable input. Also, I would like to thank all my colleagues of the HPC group at RRZE for providing a fruitful, supportive, and friendly atmosphere (in alphabetical order): Christie, Faisal, Jan, Julian, Markus, Michael, Rasa, Thomas, and Thomas.

I would like to thank all my colleagues involved in the ESSEX project from Cologne, Greifswald, Tokyo, Tsukuba, and Wuppertal for the lively scientific exchange during the past years.

Thanks to Hartwig Anzt for making my visit in Knoxville possible and his assistance during this visit and beyond.

My deepest thanks go to my parents, my brother Daniel, and my girlfriend Jule for their everlasting support, encouragement, and company. Without them, it would have been a lot harder for me to pursue my goals. Last, but not least, I send thanks to all my friends for bringing joy into my life.

This work was supported by the German Research Foundation (DFG) through the Priority Program 1648 "Software for Exascale Computing" (SPPEXA) under the project ESSEX and the Competence Network for Scientific High Performance Computing in Bavaria (KONWIHR).

Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

List of Abbreviations

API	Application Programming Interface
AVX	Advanced Vector Extensions
AVX2	Advanced Vector Extensions 2
BJDQR	Block Jacobi-Davidson QR
BLAS	Basic Linear Algebra Subprograms
BMC	Block Multi-Coloring
CA-GMRES	Communication-Avoiding Generalized Minimum Residual
CAQR	Communication-Avoiding QR
ccNUMA	cache-coherent Non-Uniform Memory Access
ChebFD	Chebyshev Filter Diagonalization
CM	Cuthill-McKee
CoD	Cluster on Die
COO	Coordinate
CPU	Central Processing Unit
CRS	Compressed Row Storage
CSCS	Swiss National Supercomputing Centre
CSX	Compressed Sparse eXtended
DDR4	Double Data Rate Fourth-Generation
DFG	German Research Foundation
DOS	Density Of States
DRAM	Dynamic Random Access Memory
DUNE	Distributed and Unified Numerics Environment
ECC	Error-Correcting Code
ECM	Execution Cache Memory
ESSEX	Equipping Sparse Solvers for Exascale

FLOP	Floating Point Operation
FMA	Fused Multiply-Add
GEMM	General Dense Matrix-Matrix Multiplication
GHOST	General, Hybrid, and Optimized Sparse Toolkit
GMRES	Generalized Minimum Residual
GPU	Graphics Processing Unit
HBM2	High Bandwidth Memory 2
HPC	High-Performance Computing
HPCG	High Performance Conjugate Gradient
HPL	High Performance Linpack
HPM	Hardware Performance Monitoring
HSW	Intel Xeon E5-2697v3 "Haswell"
HYB	Hybrid
ICC	Intel C Compiler
IDR	Induced Dimension Reduction
ILU	Incomplete LU
IVB	Intel Xeon E5-2660v2 "Ivy Bridge" with DDR3-1866 SDRAM
IVB*	Intel Xeon E5-2660v2 "Ivy Bridge" with DDR3-1600 SDRAM
JCAHPC	Joint Center for Advanced High Performance Computing
JD	Jacobi-Davidson
JDQMR	Jacobi-Davidson Quasi-Minimal Residual
JDQR	Jacobi-Davidson QR
JDS	Jagged Diagonal Storage
K20	NVIDIA Tesla K20X "Kepler"
K20m	NVIDIA Tesla K20m "Kepler"
KNC	Intel Xeon Phi 5110P "Knight's Corner"
KNL	Intel Xeon Phi 7250 "Knight's Landing"
KPM	Kernel Polynomial Method
LLC	Last Level Cache
LRZ	Leibniz Supercomputing Center
MCDRAM	Multi-Channel Dynamic Random Access Memory
MEM	Main Memory

MIC	Many Integrated Core
MKL	Math Kernel Library
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
P100	NVIDIA Tesla P100 "Pascal"
PCG	Preconditioned Conjugate Gradient
PCIe	Peripheral Component Interconnect Express
PETSc	Portable, Extensible Toolkit for Scientific Computation
PHIST	Pipelined Hybrid Iterative Solver Toolkit
pJDS	padded Jagged Diagonal Storage
PRIMME	Preconditioned Iterative Multimethod Eigensolver
PU	Processing Unit
QPI	QuickPath Interconnect
RCM	Reverse Cuthill-McKee
RODC	Read-Only Data Cache
RRZE	Erlangen Regional Computing Center
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SM	Streaming Pascal Multiprocessor
SMT	Simultaneous Multithreading
SMX	Streaming Kepler Multiprocessor
SNB	Intel Xeon E5-2670 "Sandy Bridge"
SpMMV	Sparse Matrix-Multiple Vectors Multiplication
SpMP	Sparse Matrix Pre-Processing
SpMV	Sparse Matrix-Vector Multiplication
SSE	Streaming SIMD Extensions
SVQB	Singular Value QB
T&S-GEMM	Tall & Skinny General Dense Matrix-Matrix Multiplication
TBB	Threading Building Blocks

TCP	Transmission Control Protocol
TSMM	Tall & Skinny Matrix-Small Matrix Multiplication
TSMTTSM	Tall & Skinny Matrix Transposed-Tall & Skinny Matrix Multiplication
TSQR	Tall Skinny QR

List of Symbols

Symbol	Unit	Description
α		right hand side vector data traffic overhead factor in SpMV
b_s	B/s	maximum attainable main memory bandwidth
β		chunk occupancy of a SELL-C-σ matrix
C		chunk height of a SELL-C-σ matrix
c_v		coefficient of non-zero count variation of a sparse matrix
E	J	energy to solution
F	FLOP	number of executed double precision floating point operations
f_{ADD}		number of double precision floating point operations for an addition
f_{MUL}		number of double precision floating point operations for a multiplication
I	FLOP/B	arithmetic intensity of a kernel
I_{M}	FLOP/B	inverse machine balance
L_C	B	cache line size
L_C^{D}		number of real double precision data items in a cache line
n_b		number of vectors in a block
n_{nz}		total number of non-zero entries of a sparse matrix
n_{nzc}		average number of non-zero entries per sparse matrix column
n_{nzr}		average number of non-zero entries per sparse matrix row
P	FLOP/s	achieved double precision floating point performance

P^*	FLOP/s	performance limit according to the Roofline model
$P_{\mathbf{max}}$	FLOP/s	applicable maximum floating point performance
$P_{\mathbf{peak}}$	FLOP/s	theoretical peak double precision floating point performance
σ		sorting scope of a SELL-C-σ matrix
V	B	data volume
$v_{\mathbf{el}}$		bytes needed to store a single matrix/vector value
$v_{\mathbf{idx}}$		bytes needed to store a single index
Ω		general data traffic overhead factor

1 Introduction

Sparse matrices are omnipresent in numerical analysis and scientific computing. Unlike dense matrices, most of sparse matrices' entries are zero. A common definition of "sparse" is that only $\mathcal{O}(n)$ entries of an $n \times n$ matrix are non-zero. Or, inverting the definition about dense matrices from WILKINSON and REINSCH, a matrix is "sparse" if it has enough zeros that it is economical to take advantage of their presence [177]. In a general sense, sparse matrices represent loosely coupled systems. They arise in numerous application fields, from quantum physics and chemistry to structural mechanics to fluid dynamics and graph analysis. Sparse linear algebra is an immensely important field of research and it has been identified as one of the "seven dwarfs" of High-Performance Computing (HPC), i.e., one of the seven fields that are believed to "be the computational kernels of many future applications" [13]. Although the tasks arising in sparse linear algebra are manifold, including the solution of linear systems or the computation of eigenvalues and -vectors, a frequent recurrence of numerical building blocks can be observed in this field. With the ever-increasing power of compute systems comes an ever-increasing demand from application scientists for solving larger and increasingly complex problems. The concomitantly growing size of sparse system matrices prohibits the use of direct methods, which steers the focus of this work to iterative solvers. Among the most important classes of iterative numerical methods for large sparse problems are Krylov subspace methods, as well as filtering and restarting techniques [147, 148]. In either of them, the multiplication of the sparse system matrix with one or more vectors is frequently the most time-consuming building block. The orthogonalization of vectors is a further compute-intensive and frequently occurring component of such solvers.

The importance of efficiency in numerical software grows with the scale on which this software is used. Assessing the efficiency of an implementation necessitates the use of analytical performance models. Performance engineering is a structured approach to software development, which is mostly

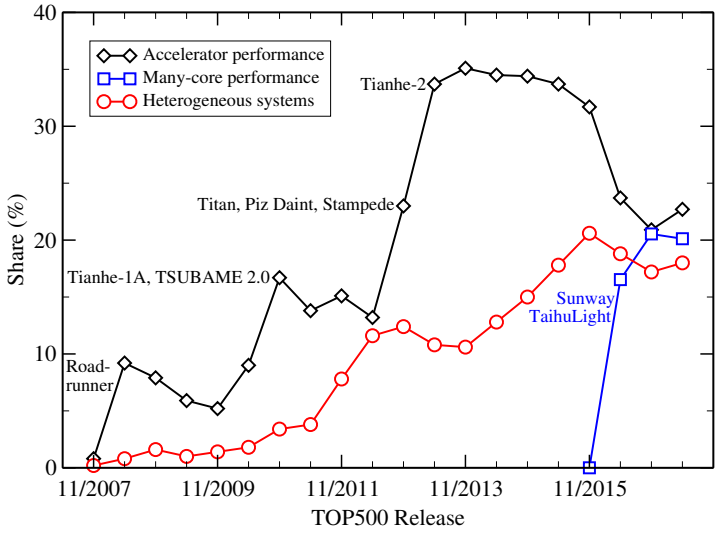


Figure 1.1: Development of the share of accelerators and many-core architectures (i.e., CPUs with at least 60 cores on a chip) with respect to the aggregated performance of TOP500 [167] systems, as well as the share of heterogeneous systems with respect to all TOP500 systems. Notable additions of heterogeneous or many-core systems to the TOP500 list are noted explicitly.

guided by performance models to obtain a deep understanding of the interplay between hard- and software [170]. Most importantly, the quality of an implementation is understood and evaluated by means of performance models, rather than by comparison against other implementations. A crucial aspect of sustainability in high-performance software development is performance portability, i.e., the software's potential to retain high efficiency across hardware architectures and generations. Obviously, detailed understanding of performance through performance modeling is a necessity for achieving performance-portable code.

Although the focus of this work is on hardware-efficient software for the current and next generation of supercomputers, a brief retrospect of the history of computer architectures is helpful to understand today's world of supercomputing. Over many decades, the development of computer architectures has followed Moore's law [129], which stated in 1965 that the number of transistors of integrated circuits is projected to double every year (or rather, as Moore correct himself ten years later, every other year [130]). It turned out that the actual development kept step with Moore's projection, and over the next decades more and more powerful single-core Central Processing Units (CPUs) were developed, thanks to a combination of better processor architectures, enhanced manufacturing processes and growing clock speeds. Even-

tually, this rapid development has come to an end due to growing manufacturing costs, physical limitations, and the need for limiting the heat emission and power dissipation of chips. This led to the advent of multi-core CPUs, which are operated at moderate clock frequencies but contain several cores on a single chip. The first non-embedded multi-core chip was IBM's POWER4 in 2001, featuring two cores with a shared level 2 cache. It was not until 2005 that Intel and AMD followed with the release of their first multi-core chips. Since then, an ever-increasing number of cores on multi-core CPUs has been observed, and hardware developers often have to trade off core complexity and features against a larger core count [2]. The appearance of accelerators and heterogeneous systems is just a natural consequence of this evolution. One of the first chips of this kind was the Cell processor in 2005 [93], which offered a heterogeneous architecture on a single chip. The first concept of a many-core CPU was Intel's Larrabee in 2008 [153] (which is the foundation for the many-core CPUs regarded in this work) and already in the year 2006, NVIDIA launched the Tesla architecture which enabled Graphics Processing Units (GPUs) for general-purpose computing [116]. The descendants of Larrabee and NVIDIA Tesla GPUs are ubiquitous in today's world of supercomputing, trading off core complexity and general applicability against a large number of simple cores to enable high throughput, as well as unparalleled performance and energy efficiency. A general view of the world of supercomputers is best obtained by analyzing the TOP500 list [167]. This database, established in 1993 and updated twice a year, ranks the 500 most powerful supercomputers of the world known to the list administrators according to their peak floating point performance. Figure 1.1 indicates that heterogeneous and many-core systems are a reality in today's supercomputers. Other than homogeneous multi-core CPU systems (which consist of nodes with one or more commodity multi-core CPUs), heterogeneous systems additionally feature one or more "accelerators" in their nodes. The Roadrunner system, being built on AMD Opteron CPUs accelerated by Cell processors, entered the TOP500 list in June 2008 as the first system to achieve a peak floating point performance of 1 PFLOP/s. Notable systems which are accelerated by GPUs are Tianhe-1A, TSUBAME 2.0, Titan and Piz Daint, whereas Stampede and Tianhe-2 use accelerators of the Intel Knight's Corner architecture. As of November 2015, more than one fifth of all published systems are heterogeneous and their accelerators account for about a third of the entire aggregated TOP500 performance. Recently, homogeneous systems comprised of self-hosted many-core chips, such as Sunway TaihuLight (based on 260-core Sunway SW26010 CPUs), Cori and Oakforest-PACS (both based on

68-core Intel Knight's Landing CPUs), are rapidly gaining importance in supercomputing. Consequently, as of November 2016 more than 40% of the aggregated TOP500 performance stems from accelerators or many-core CPUs. Although one cannot foresee the future, there is no prospect of an end of the relevance of such systems, which justifies the special attention given to them in this thesis.

An important challenge in today's world of supercomputing is the presence of legacy software and data structures in a rapidly changing hardware environment. This work is an attempt to design, develop and implement numerical building blocks and data structures from scratch, being able to cope with the challenges posed by both the constantly changing hardware environment as well as increasing demands from the application side. This thesis has been conducted within the Equipping Sparse Solvers for Exascale (ESSEX) project. ESSEX is a joint project funded by the German Research Foundation (DFG) under the priority program 1648 "Software for Exascale Computing." The focus of ESSEX is on efficient, scalable, and robust numerical algorithms and programming concepts for iterative sparse matrix applications on next-generation supercomputers. The numerical building blocks developed within this work form the foundation of the ESSEX project.

1.1 Related Work

Due to the indisputable relevance of high-performance sparse linear algebra, work related to all parts of this thesis is abundant and will be reviewed here. Although this section uses several notions and conceptualities which are explained in later sections, it is useful to provide an overview about related work already in the beginning to put the contributions summarized in section 1.2 into the global context.

SpMV. As will be shown in this work, the Sparse Matrix-Vector Multiplication (SpMV) is one of most important compute kernels in sparse linear algebra, and the efficiency of this operation has undergone intense research over many years. Comprehensive performance studies are available [76], and extensive overviews of optimization techniques for SpMV can be found [174]. The main object of research about SpMV performance is often the storage format of sparse matrices. Unlike dense matrices, sparse matrices offer abundant possibilities to store them, and the storage format directly influences the performance of operations like SpMV. A common finding is that

the performance-optimal storage format for a given sparse matrix depends on its structure as well as the target hardware.

As the performance of SpMV is typically limited by the memory bandwidth, the exploitation of matrix structures is a promising way to alleviate this bottleneck and enable efficient SpMV. Although this work does not take into account special matrix structures, they are certainly relevant and a short survey of related work is justified. One such structure is the presence of dense (sub-)diagonals in the matrix, whose positive effect on SpMV performance has been noted long ago [122]. Another possibly present regular structure in sparse matrices is the presence of dense sub-blocks, enabling high-performance SpMV kernels with blocking [173, 178]. Automatic determination and handling of such matrix structures is a way to high-performance SpMV if no a priori knowledge about the matrix structure is present. Having its roots in work conducted in the early 1990s by Agarwal et al. [3], this idea is still developed further today by Kourtis et al. [99], who pursue a comprehensive approach to the exploitation of special matrix structures with the Compressed Sparse eXtended (CSX) storage format. The central idea is to detect characteristic sparsity patterns in a pre-processing step. Such patterns are then encapsulated in a certain format, e.g., a dense matrix format for dense sub-blocks. Re-ordering of matrix rows and columns can be applied, e.g., to generate and enlarge dense sub-blocks [138]. A further important class of "special" sparse matrices are the ones which are symmetric. Obviously, only half of the matrix needs to be stored if it is symmetric. In contrast to non-symmetric SpMV, parallel SpMV kernels exploiting matrix symmetry require a special implementation to avoid race conditions and work at high efficiency. This is due to the fact that, whenever a matrix entry gets multiplied with the input vector, also its "sibling" (its corresponding symmetric entry which is located at a different matrix row) needs to be multiplied with the vector, leading to conflicts with the thread currently active in the sibling's row. There are several ideas to cope with this issue. For instance, Buluc et al. have developed the idea of "compressed sparse blocks" [33], and their symmetric SpMV algorithm requires non-zero entries to be close to the diagonal or else, atomic store instructions (which are rarely efficient) will be used. In contrast to this, Martone uses locking in his "recursive sparse blocks" approach [123]. The large thread count on GPU architectures demands different solutions to this problem, like the task-scheduling approach by Mironowicz et al. [127].

However, not always does a sparse matrix contain special structures like dense diagonals or sub-blocks or is it possible to re-structure the matrix. In this case, one speaks of *general* sparse matrices. Research about general

SpMV primarily revolves around the storage format of the matrix. Today, the variety of sparse matrix storage formats is tremendous, which is partly founded on the fact that the optimal choice of sparse matrix storage format depends on the used compute architecture. FILIPPONE et al. review over 70 published formats tailored to GPU architectures alone [68], not to mention the countless storage formats and variations tailored towards cache-based CPUs, vector computers, or more exotic architectures. Compressed Row Storage (CRS) [21] is arguably the most widely known storage format for general sparse matrices, and it is often a candidate for high-performance SpMV on multi-core CPUs. With the emergence of accelerator architectures, extensive research about suitable data formats for them has been conducted, demonstrating the deficiencies of CRS on such architectures and proposing alternatives such as ELLPACK and its variants [24, 119].

At this point it should be noted that SpMV performance is not the only evaluation criterion of sparse matrix storage formats. LANGR and TVRDIK present a comprehensive review of existing sparse matrix storage formats and propose new evaluation criteria which they would like to see seized by the SpMV performance community, such as the runtime and memory requirements of finding optimal parameters for a storage format [113].

Block algorithms. Block algorithms are algorithms which work with a block of vectors rather than a single vector. Many block algorithms, like the Block Lanczos method developed by CULLUM and DONATH [46] or the Block Conjugate Gradient method by O'LEARY [135], offer numerical benefits compared to their single-vector counterparts. Beyond that, blocking bears the potential of performance improvements by reducing data traffic.

The resulting blocks of vectors – due to their shape often referred to as "tall & skinny" dense matrices – allow the use of optimized routines, such as Sparse Matrix-Multiple Vectors Multiplication (SpMMV). This operation is the block-vector counterpart of SpMV and often the key operation in block sparse linear algebra algorithms. Applying a sparse matrix to multiple vectors at once has the potential to reduce the overall data transfers needed and increase the performance. While this positive effect has been noted long ago, it was first quantified by GROPP et al. [78], who present a first analytical model for the speedup of SpMMV over SpMV. Besides the storage format of the sparse matrix, also the storage format of tall & skinny matrices can be decided upon. In the case of SpMMV, row-major (i.e., interleaved) storage of vectors in the block usually yields highest performance, as observed, among others, by IM et al. [89].

Another commonly used procedure in block algorithms is the orthogonalization of a block of vectors. This can be done, e.g., using Tall Skinny QR (TSQR) as presented by Demmel et al. [54] or Singular Value QB (SVQB) as proposed by Stathopoulos and Wu [160]. An important building block of those orthogonalization schemes are multiplications of block vectors. Even though General Dense Matrix-Matrix Multiplication (GEMM) operations are a well-studied, -understood, and -tuned numerical kernel, this is only true for matrices which are large in both dimensions, and existing and established software libraries often show significant performance deficiencies for Tall & Skinny General Dense Matrix-Matrix Multiplication (T&S-GEMM) operations. This has been noted, for example, in the context of Communication-Avoiding QR (CAQR) [54] (which uses TSQR for its panel factorization) [7, 181]. Numerical building block developers have also taken note of this and approaches towards enhanced performance for tall & skinny matrix multiplications exist. For instance, LIBXSMM [81] is a software library for small GEMM operations on Intel Architectures using enhanced techniques like just-in-time compilation. Although this library assumes that all input data are small, it could be harnessed for block vector operations in "batch mode." Recent work from Abdelfattah et al. about "performance, design, and autotuning of batched GEMM for GPUs" [1] shows that the GPU computing community is aware of these issues as well.

Kernel fusion. Although the data traffic can be reduced by using block algorithms, it commonly remains the relevant bottleneck for execution performance. Computational kernels of scientific computing are frequently formulated as loops. If the same data is used in different loops of a solver, loop fusion can be employed if certain prerequisites are fulfilled (e.g., there are no dependencies between the loops). Loop fusion is one of the most fundamental optimization techniques in computing [16, 58, 95] and capable compilers can employ it automatically in sufficiently simple cases.

On a higher level, i.e., if the solver is not comprised just of loops but of separate kernels (which may be subject to execution on an accelerator), one speaks of kernel fusion rather than loop fusion. In recent years, this idea has attracted new attention from the GPU programming community [145, 162, 175]. In the simplest case, kernel fusion implies manual implementation of tailored kernels. There exist also approaches which try to generalize kernel fusion. For instance, "Build To Order BLAS" implements a compiler to automatically generate fused kernels from a high level representation [25]. However, it is limited to CPU architectures and only implements the dense Basic

Linear Algebra Subprograms (BLAS), which form perhaps the most prominent set of kernels in scientific computing. An extension to sparse linear algebra kernels is planned for the future [132]. Russell et al. follow a similar approach, but kernels are generated at runtime rather than compile time here [146]. Besides those efforts, which are targeting CPU architectures, there are similar approaches for GPUs like the one by Filipovič et al., whose source-to-source compiler can automatically generate fused kernels for sequences of dense BLAS-1/2 calls [67]. A major drawback of those approaches towards kernel fusion is the lack of sparse matrix kernels. Automatic fusion of general, non-BLAS kernels is a much harder task, especially if the resulting kernels should operate at near-optimal performance.

Sparse linear algebra software packages. Optimized SpMV, SpMMV, and T&S-GEMM routines as well as fused kernels need to be offered such that they can be used by application and algorithm developers in an easy and efficient way. Due to the very high scientific relevance of sparse linear algebra, there is a large interest in developing numerical software for this kind of applications. Sparse linear algebra software libraries provide interfaces to such kernels, but their modus operandi may differ significantly. First, there exist basic building block libraries, in many cases developed by hardware vendors, which often provide implementations of the BLAS functionality as well as the sparse matrix extensions of the BLAS [59, 62]. Obviously, such vendor-supplied libraries, like the Intel Math Kernel Library (MKL) [90] or the cuBLAS [44] and cuSPARSE [47] libraries from NVIDIA, are tuned to the vendor's compute architectures. The vendors make big efforts in developing such toolkits, which results in a usually high efficiency for many use cases.

Besides those vendor-supplied libraries, numerous alternatives providing similar or extended functionality exist. The development of scientific compute platforms towards heterogeneous and many-core architectures has steered the development focus of such software towards this kind of architectures. Perhaps the most prominent building block library for heterogeneous architectures is MAGMA [166]. It follows a task-parallel approach to map heterogeneous workloads to heterogeneous architectures. ViennaCL [144] is another, similar software package. Using CUDA, Open Computing Language (OpenCL), or Open Multi-Processing (OpenMP), the same application code can run on a wide range of different architectures. ViennaCL's missing support for complex numbers greatly limits its applicability in the scope of the present work.

A common property of the so-far named building block libraries is that they do not feature distributed-memory parallelism. However, this is an indispensable prerequisite for large-scale computations. Hence, if distributed memory systems should be harnessed, other tools are required on top. A library containing distributed memory parallel sparse iterative solvers and preconditioners is PARALUTION [136]. However, multi-process and complex number support are restricted to its commercial version. LAMA is another numerical library, which promises "hardware independent code for multiple platforms" and "full cluster support" [109]. It features the CRS and ELLPACK sparse matrix storage formats and calls cuSPARSE for its CRS SpMV kernel. The Distributed and Unified Numerics Environment (DUNE) [28] is a library featuring parallelism using the Message Passing Interface (MPI) and further layers (which is known as "MPI+X") as well as accelerator support, tailored towards the solution of partial differential equations. Two of the most prominent numerical libraries featuring sparse matrix kernels as well as accelerator support and distributed-memory parallelism are the Portable, Extensible Toolkit for Scientific Computation (PETSc) [20] and Trilinos [82]. Both rely on distributed-memory parallelism via MPI and at least one more level of parallelism underneath it. PETSc "is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations" [19]. It features MPI+X parallelism and offers GPU support through CUDA or OpenCL [126]. The Trilinos packages Tpetra [17] and Kokkos [36] also provide parallel numerical building blocks using MPI+X. Trilinos is a selection of numerous packages, which is beneficial in terms of distributed development and maintenance. PETSc, on the other hand, features a tighter integration between the different software layers. A common and inherent drawback of existing and widely used software packages is the need to retain backward compatibility which often restricts their flexibility, for instance when it comes to adapt to various levels of potential heterogeneity.

Sparse linear algebra algorithms. Efficient software is best developed with a holistic view of all involved layers, from applications to algorithms and implementations. Even though this work is by no means about the theory and development of sparse linear algebra algorithms, progressions and trends in this field must not be ignored. For instance, recent advancements of sparse linear algebra algorithms include pipelined methods that try to hide communication (e.g., pipelined Generalized Minimum Residual (GMRES) [74]), methods that try to avoid communication (e.g., block Krylov

methods and Communication-Avoiding Generalized Minimum Residual (CA-GMRES) [53]), and fully asynchronous methods like the asynchronous Incomplete LU (ILU) preconditioner suggested by Chow and Patel [41]. Furthermore, methods which are particularly suited for current hardware architectures like, e.g., polynomial filtering techniques have recently gained new attraction [65]. Besides the aforementioned peculiarities of current hardware architectures, any software which claims usefulness for sparse linear algebra algorithms needs to keep those algorithmic advances in mind.

Performance modeling. During the last years, the development and application of performance models has gained increasing importance in the field of HPC. Arguably the most prominent performance model is the Roofline model developed by Williams et al. [179], which relates processor performance to off-chip memory bandwidth. The principal ideas of this performance model date back to early work conducted by Kung [108], Callahan et al. [35], and Hockney and Curington [83]. Performance models based on the Roofline model have been developed for a large number of numerical kernels, e.g., the SpMV operation, taking its special re-use characteristics into account [118, 152]. In recent years, the Roofline model has been refined and extended to a wider set of hardware bottlenecks. This contains, for example, taking latency effects into account [34, 120] or fine-grained modeling of caches [34, 88]. A major shortcoming of the original Roofline model is the lack of explanation of multi-core scaling behavior. The Execution Cache Memory (ECM) performance model as first introduced by Treibig and Hager [168] and later refined by Stengel et al. [161] and Hager et al. [79] poses an option to overcome this issue by providing deep insight into the scaling behavior and in-cache data traffic.

1.2 Contributions

The main contributions of this work concern several of the described aspects of high-performance sparse linear algebra and are briefly summarized in the following.

Holistic performance engineering. Unlike related work which often has a restricted view of performance, e.g., by conducting performance analyses only for single compute-intensive kernels, this work strives to retain a holistic view of all layers involved in sparse linear algebra algorithms, from applications on the highest level to algorithms and further down to basic numerical

building blocks. On top of performance-engineered basic building blocks, having in mind the other layers allows for the development of highly efficient numerical software.

A unified sparse matrix storage format for high-performance SpMV. As mentioned above, SpMV is one of the key operations in the field of sparse linear algebra. The dependence of the optimal sparse matrix storage format on the compute platform is especially unwanted facing today's heterogeneous systems, and having a unified sparse matrix storage format which yields efficient SpMV on many different architectures is desirable in many regards. For example, the complexity of sparse linear algebra software for heterogeneous systems can be reduced greatly and moreover, a platform-agnostic data structure allows for low-overhead dynamic load balancing on such systems. This work presents SELL-C-σ as an attempt towards a unified sparse matrix storage format. By identifying commonalities between architectures which are very different at first sight, SELL-C-σ is designed as a data structure inherently suitable for heterogeneous computing. SELL-C-σ yields competitive or superior SpMV performance even compared to vendor-supplied implementations with device-optimized formats. Hence, it is a very promising format to mitigate the burden of selecting the optimal sparse matrix storage formats in the heterogeneous computing era. This work analyzes the SpMV performance based on established and refined performance models, which helps in understanding matrix-dependent performance properties of this kernel.

Performance engineering for SpMMV. Although the potential performance benefit of SpMMV over SpMV is well known, structured performance analyses of this operation are rare. This work attempts to close this gap by providing a detailed performance analysis for the SpMMV kernel, including performance models and speedup estimations, which helps in understanding the performance-limiting factors of this operation. The influence of the storage formats of the block vectors as well as the sparse matrix are given special consideration.

Performance engineering for T&S-GEMM operations. Due to the high relevance of block algorithms in sparse linear algebra it is worthwhile to closely examine the efficiency of T&S-GEMM operations. This work reveals flaws of existing implementations, which lead to the manual development of such kernels. Their implementation is described and their high efficiency with respect to a Roofline performance model, as well as their superiority

over a vendor-tuned BLAS library is demonstrated. Besides block orthogonalization using TSQR or SVQB, those kernels may turn out useful for many more applications.

Custom compute kernels. The entire development process in this work is guided by a selection of sparse eigenvalue solvers. A holistic approach to performance engineering allows to mitigate relevant bottlenecks already at the algorithmic level. Mostly founded on the idea of kernel fusion, custom compute kernels for selected sparse eigenvalue solvers are developed. Despite the fact that manual kernel fusion is employed, the resulting kernels can easily be altered or serve as blueprints to be applicable for further applications and algorithms.

An open-source software library. All developed building blocks and custom kernels are part of the open-source software library General, Hybrid, and Optimized Sparse Toolkit (GHOST), which is the major "tangible" outcome of this work. On top of the aforementioned optimized kernels, it features distributed-memory parallelism and data-parallel heterogeneous execution. Further features of GHOST which contribute to its high efficiency are automatic code generation and the presence of an affinity-aware tasking mechanism. In the world of numerical libraries for sparse matrix algorithms, the feature set of GHOST is unique and performance experiments of real-world applications at the end of this work demonstrate its capabilities. A key advantage of GHOST over the previously described software packages is that it was designed and developed from scratch in the ESSEX project. The absence of the need to retain backward compatibility and the presence of full freedom regarding the design of data structures and a holistic view of all layers of numerical software greatly support the development of highly efficient software.

Efficient and large-scale eigenvalue solvers. The eigenvalue solver implementations based on the developed building blocks and GHOST stand out in terms of efficiency and scalability. Eigenvalue computations on several thousand homogeneous and heterogeneous nodes demonstrate the ability to bring the developed performance-engineered building blocks to a very large scale. At this point, it should also be noted that the inclusion of preconditioning in the developed algorithms is an ongoing research topic in the ESSEX project. However, the outcomes of this work are certainly applicable also to preconditioned sparse linear algebra algorithms.

1.3 List of Publications

In this section, publications which have been prepared within the scope of this thesis and are relevant to the presented results are listed according to their year of publication and briefly summarized. For those where the author of this thesis is not the first author, specific contributions are specified. For each publication, references to associated parts of this thesis are given.

- Kreutzer, Hager, Wellein, Fehske, Basermann, and Bishop, "Sparse Matrix-Vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation" [103]. In: *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. In this paper, both the pJDS sparse matrix storage format (as briefly explained in section 5.1) and a scalable MPI+CUDA SpMV implementation as used in GHOST (see section 9.4) are introduced.

- Kreutzer, Hager, Wellein, Fehske, and Bishop, "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units" [104]. In: *SIAM Journal on Scientific Computing* (2014). This journal article introduces the SELL-C-σ storage format and provides an in-depth performance analysis of the SpMV operation using this and other formats (see chapter 5).

- Alvermann, Basermann, Fehske, Galgon, Hager, Kreutzer, Krämer, Lang, Pieper, Röhrig-Zöllner, Shahzad, Thies, and Wellein, "ESSEX: Equipping Sparse Solvers for Exascale" [6]. In: *Euro-Par 2014: Parallel Processing Workshops*. This overview paper describes the ESSEX project, which is closely connected to this thesis. It gives a good hint for the classification of this work. The author of this thesis contributed descriptions of GHOST's tasking mechanism (see section 9.2), a brief performance survey of SELL-C-σ SpMV (see chapter 5), and an analysis of algorithmic optimizations and performance gains of the KPM (see section 8.1).

- Röhrig-Zöllner, Thies, Kreutzer, Alvermann, Pieper, Basermann, Hager, Wellein, and Fehske, "Increasing the Performance of the Jacobi–Davidson Method by Blocking" [142]. In: *SIAM Journal on Scientific Computing* (2015). Development and performance analysis of the BJDQR algorithm (see sections 3.4 and 10.3) are covered in this

journal article. This thesis' author contributed a model-guided development of the involved building blocks, including highly optimized implementations.

- Kreutzer, Pieper, Hager, Wellein, Alvermann, and Fehske, "Performance Engineering of the Kernel Polynomial Method on Large-Scale CPU-GPU Systems" [100]. In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. This paper describes the development, node-level and large-scale performance analysis of a fully heterogeneous KPM (see section 3.2) implementation on the heterogeneous supercomputer Piz Daint (see sections 3.2, 8.1 and 10.1).

- Pieper, Kreutzer, Alvermann, Galgon, Fehske, Hager, Lang, and Wellein, "High-performance implementation of Chebyshev filter diagonalization for interior eigenvalue computations" [137]. In: *Journal of Computational Physics* (2016). In this journal article, a detailed description including performance engineering of the ChebFD algorithm as analyzed in sections 3.3, 8.2 and 10.2 can be found. The contributions of this thesis' author mainly consist in a detailed performance analysis of the involved building blocks and highly efficient implementations of them.

- Kreutzer, Thies, Pieper, Alvermann, Galgon, Röhrig-Zöllner, Shahzad, Basermann, Bishop, Fehske, Hager, Lang, and Wellein, "Performance Engineering and Energy Efficiency of Building Blocks for Large, Sparse Eigenvalue Computations on Heterogeneous Supercomputers" [106]. In: *Software for Exascale Computing - SPPEXA 2013-2015*. This is a project overview and progress report of the ESSEX project. The KPM energy analysis in section 10.4 and some further performance data presented in chapter 10 are related to this publication.

- Thies, Galgon, Shahzad, Alvermann, Kreutzer, Pieper, Röhrig-Zöllner, Basermann, Fehske, Hager, Lang, and Wellein, "Towards an Exascale Enabled Sparse Solver Repository" [163]. In: *Software for Exascale Computing - SPPEXA 2013-2015*. This book chapter focuses on the software infrastructure of the ESSEX project. It is partly related to chapter 9 of this work. The author of this thesis contributed a short overview of GHOST as a major component of the ESSEX software environment.

- Anzt, Kreutzer, Ponce, Peterson, Wellein, and Dongarra, "Optimization and performance evaluation of the IDR iterative Krylov

solver on GPUs" [11]. In: *The International Journal of High Performance Computing Applications* (2018). In this paper a performance analysis of an Induced Dimension Reduction (IDR) iterative Krylov solver can be found. While the solver itself is no part of this thesis, several ideas and aspects, like the SELL-C-σ storage format (section 5.2), Roofline performance modeling (section 4.2), and kernel fusion similar to the custom kernels in chapter 8 are covered. This publication also substantiates the applicability of this thesis' results to further algorithms above eigenvalue solvers. The author of this thesis contributed a performance model which is used to asses the efficiency of the developed IDR solver.

- Anzt, Dongarra, Kreutzer, Wellein, and Köhler, "Efficiency of General Krylov Methods on GPUs – An Experimental Study" [9]. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. This publication builds on the IDR implementation developed in [11] and has the same links to this thesis. This paper presents an experimental study and comparison of different Krylov methods on GPU architectures for a wide range of test matrices. For such a study to be valuable, the implementations of the methods need to be assessed by means of performance models. The author of this thesis contributed the aforementioned performance analysis of the IDR solver. As the other considered solvers use similar building blocks, also their efficiency can be assured.

- Kreutzer, Thies, Röhrig-Zöllner, Pieper, Shahzad, Galgon, Basermann, Fehske, Hager, and Wellein, "GHOST: Building Blocks for High Performance Sparse Linear Algebra on Heterogeneous Systems" [107]. In: *International Journal of Parallel Programming* (2016). This paper provides an in-depth presentation of the GHOST library. Chapter 9 is closely related to this publication.

- Anzt, Gates, Dongarra, Kreutzer, Wellein, and Köhler, "Preconditioned Krylov solvers on GPUs" [10]. In: *Parallel Computing* (2017). This article extends previously published work [9] by taking preconditioning into account, having the same links to this thesis and contributions by its author.

Besides those research papers, several posters related to this work have been presented at conferences:

- Kreutzer, Hager, and Wellein, "A unified sparse matrix storage format for heterogeneous systems" [102]. This poster presents early

work on the development of the SELL-C-σ storage format (addressed in chapter 5).

- Kreutzer, Pieper, Alvermann, Fehske, Hager, Wellein, and Bishop, "Efficient Large-Scale Sparse Eigenvalue Computations on Heterogeneous Hardware" [105]. This poster gives a combined presentation of the KPM and ChebFD algorithms, early work on T&S-GEMM performance and extensions of previously published KPM scaling experiments [100] (see section 10.1).
- Kreutzer, "GHOST: General, Hybrid, and Optimized Sparse Toolkit" [101]. This is an overview poster of GHOST, briefly summarizing all its key features and aspects as described in chapter 9.

1.4 Structure of this Thesis

The contents of each of this work's chapters are briefly summarized in the following.

Chapter 2 The hard- and software test bed, as well as an overview of all test cases used throughout this work, are presented in chapter 2. This chapter, which also briefly describes relevant hardware concepts and programming techniques, is a useful resource for reference when reading through this work.

Chapter 3 The motivating application of this work is the solution of large and sparse eigenvalue problems. Chapter 3 briefly introduces four solvers addressing different classes of eigenvalue problems and providing different algorithmic approaches. While omitting most of the numerical details which are out of scope for this work, this overview is an essential foundation for the rest of this work. From the selected eigenvalue solvers, relevant numerical building blocks are identified which are further analyzed later in this thesis.

Chapter 4 Performance modeling is an integral part of this thesis and the ESSEX project in general. Chapter 4 motivates the choice of the Roofline model as the preferred performance model in this work. Its application to the previously identified relevant building blocks as described.

Chapter 5 In chapters 3 and 4, it becomes apparent that the SpMV is one of the most important building blocks. It is often the most time-consuming operation in sparse linear algebra algorithms, and its particular performance properties make it worth investigating. This is done in chapter 5. A small survey shows that the SpMV performance is strongly connected to the storage format of the sparse matrix, which itself is often hardware-specific. This leads to the development of the SELL-C-σ format as a unified and hardware-agnostic format. SELL-C-σ is analyzed in detail and a thorough performance analysis of SpMV with different data formats is given.

Chapter 6 It frequently occurs in sparse linear algebra that computations are done with a single sparse system matrix but multiple vectors. This leads to the occurrence of the SpMMV operation, a numerical building block of similar importance as SpMV. The SpMV performance analysis is extended to multiple vectors and the performance properties of SpMMV are discussed in chapter 6.

Chapter 7 Another important and worth investigating building block is the multiplication of tall & skinny dense matrices. In the context of this work, such matrices originate from vector blocking. Deficiencies of existing building block libraries together with special performance properties of tall & skinny matrix multiplications motivate a close investigation as done in chapter 7.

Chapter 8 To achieve optimal efficiency for the examined sparse eigenvalue solvers, it is often not sufficient to optimize the performance of each involved building block. On top of efficient building blocks, the implementation of custom kernels is often a key to further significant performance increases. Such implementations in view of the presented eigenvalue solvers are examined in Chapter 8. The implementation of custom kernels is justified and the resulting kernels are closely analyzed by means of performance models.

Chapter 9 The open-source software library GHOST, which has a special focus on high performance for large-scale sparse linear algebra, contains all software development efforts which have been conducted within this work, is presented in chapter 9. Important

design principles and programming paradigms which are relevant to this work are described there.

Chapter 10 GHOST contains all optimized building blocks and kernels from chapters 5 to 8 and can be used to obtain high-performance implementations for real-world applications as summarized in chapter 10.

Chapter 11 The thesis is briefly summarized and concluded in chapter 11.

Chapter 12 This chapter provides an outlook to potential future work concerning this thesis.

Figure 1.2 tries to put this work into perspective by giving a "big picture" of the various components of a large-scale interior eigenvalue computation using a specific solver, namely Chebyshev Filter Diagonalization (ChebFD). Based on properties of the sparse system matrix and the present hardware architectures, a bottleneck analysis of the "textbook" ChebFD algorithm is conducted. Kernel fusion and vector blocking are identified as promising algorithmic optimization techniques, resulting in an optimized version of the algorithm. The system matrix is stored in the platform-agnostic SELL-C-σ format (which enables high efficiency on all relevant architectures) and fed into the optimized algorithm, which itself consists of two major steps: the application of the polynomial filter (which is done by means of a custom kernel) and the orthogonalization of search vectors (which builds on T&S-GEMM operations). All involved building blocks require proper performance engineering to work at best efficiency. On top of those building blocks, large-scale computations require a further software layer implementing distributed-memory parallel and scalable compute kernels for heterogeneous architectures: GHOST. Gluing all components together enables highly efficient interior eigenvalue computations at a very large scale.

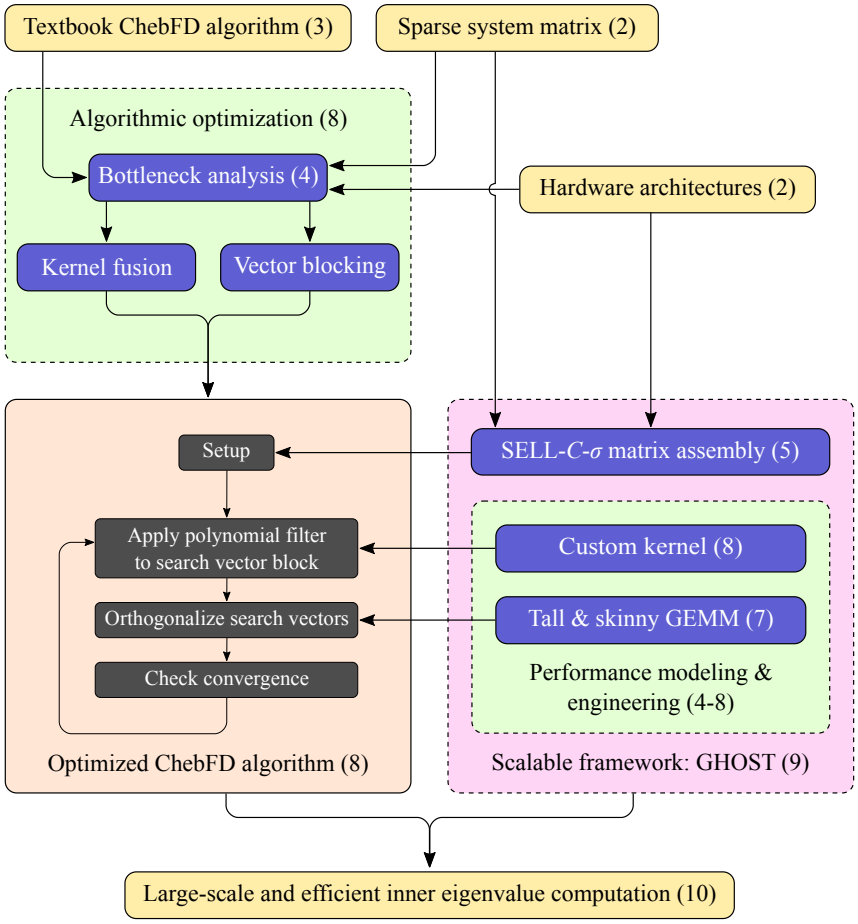


Figure 1.2: Schematic view of a large-scale interior eigenvalue computation on the basis of the ChebFD algorithm. References of selected components to chapters of this work are given in parentheses.

2 Test Bed

This chapter is an important reference for the rest of this thesis, as it presents the complete hard- and software test bed as well as all test matrices used throughout this work.

2.1 Node-Level Hardware Architectures

Benchmark and solver runs are executed on different hardware architectures. As this work is about heterogeneous computing, different classes of compute architectures are used. Table 2.1 summarizes all hardware architectures which are used in this work, differentiating between "standard" multi-core Central Processing Units (CPUs) (table 2.1a) and more novel compute platforms like many-core CPUs and Graphics Processing Units (GPUs), all of which will be referred to as "accelerators" (table 2.1b). The reason for this naming is that, originally, the only operating mode for these architectures consisted in accelerating the work of a host CPU. Recent advances are, for example, self-hosted many-core CPUs, which are no longer "accelerators" in the strict sense. Still, for simplicity in this work, they will be referred to as such. Not all results will be shown for all platforms, as this would merely add more volume to this work without providing more insight.

This section provides a brief overview of hardware features and concepts relevant to this work. The book by Hager and Wellein [80] can be consulted for further reading on basic concepts of current hardware architectures. Detailed analyses of all relevant architectures are abundantly available by hardware vendors as well as from the scientific community [85, 155].

Multi-core CPU architectures. Intel Xeon E5-2670 "Sandy Bridge" (SNB), Intel Xeon E5-2660v2 "Ivy Bridge" (IVB) and Intel Xeon E5-2697v3 "Haswell" (HSW) represent three generations of Intel multi-core CPUs. Especially the number of cores, Last Level Cache (LLC) size, and maximum attainable main memory bandwidth b_s have increased throughout these generations. For the

		SNB	IVB(*)	HSW
Model		Intel Xeon E5-2670	Intel Xeon E5-2660v2	Intel Xeon E5-2697v3
Microarchitecture		Sandy Bridge	Ivy Bridge	Haswell
Launch		Q1'12	Q3'13	Q3'14
Cores		8	10	14
SMT		2	2	2
Clock	(GHz)	2.60^{+}	2.20	2.60
P_{peak}	(GFLOP/s)	166.4	176	582.4
b_{s}	(GB/s)	45	52 (47)	65
	SMT	1	1	1
	Bench	MM$\{64, 4\}$	MM$\{256, 4\}$	MM$\{64, 4\}$
I_{M}	(FLOP/B)	4.0	3.4 (3.9)	9.0
SIMD	(B)	32	32	32
LLC	(MiB)	20	25	35
MEM	(GiB)	32	32	32
L_C	(B)	64	64	64

(a) Multicore CPU architectures

		KNC	KNL	K20	K20m	P100
Model		Intel Xeon Phi 5110P	Intel Xeon Phi 7250	NVIDIA Tesla K20X	NVIDIA Tesla K20m	NVIDIA Tesla P100
Microarchitecture		Knight's Corner	Knight's Landing	Kepler	Kepler	Pascal
Launch		Q4'12	Q2'16	Q4'12	Q4'12	Q2'16
Cores		60	68	2688	2496	3584
SMT		4	4	-	-	-
Clock	(GHz)	1.05	1.40^{+}	0.74^{+}	0.71^{+}	1.33^{+}
P_{peak}	(GFLOP/s)	1010	3046	1310	1175	4760
b_{s}	(GB/s)	174	509	182	152	587
	SMT	3	4	-	-	-
	Bench	MM$\{32, 8\}$	MM$\{1, 8\}$	COPY	COPY	DOT
I_{M}	(FLOP/B)	6.9	6.0	7.3	7.7	8.1
SIMD	(B)	64	64	-	-	-
LLC	(MiB)	30	34	1.5	1.25	4
MEM	(GiB)	8	16	6	5	16
L_C	(B)	64	64	32	32	32

(b) Many-core and GPU ("accelerator") architectures

Table 2.1: Hardware architectures used in this work. "SMT" denotes the number of hardware threads per core, P_{peak} the theoretical peak double precision floating point performance, b_{s} the maximum attainable main memory bandwidth, as well as the number of threads per core, and respective microbenchmark to achieve it (see section 4.1). I_{M} denotes the inverse machine balance which is defined as $P_{\text{peak}}/b_{\text{s}}$, LLC (MEM) the size of the last level cache (main memory) and L_C the cache line size. Clock frequencies suffixed with "$^{+}$" are not fixed and subject to vary at runtime.

IVB chip, two variants with different memory clock speed (1866 MHz and 1600 MHz) are considered and the slower one is named Intel Xeon E5-2660v2 "Ivy Bridge" (IVB*).

The Single Instruction, Multiple Data (SIMD) width has not changed, although HSW implements the Advanced Vector Extensions 2 (AVX2) instruction set (as compared to Advanced Vector Extensions (AVX) on SNB and IVB). Each core of SNB and IVB is capable of computing one SIMD addition and multiplication each per cycle. AVX2 adds support for Fused Multiply-Add (FMA) instructions, and HSW is capable of execution two FMA instructions per cycle and core. This leads to a doubling of core performance and – together with the increased core count – to a significant increase of theoretical peak double precision floating point performance P_{peak} between SNB/IVB and HSW.

Although two hardware threads can be active at the same time on each core using Simultaneous Multithreading (SMT) (i.e., placing more than one hardware thread on each physical core), the highest main memory bandwidth on all three multi-core CPU architectures is achieved with only a single thread active per core. Hence, the number of threads per core will be fixed to one in this thesis if not noted otherwise.

NUMA designs. Non-Uniform Memory Access (NUMA) denotes a memory design in multi-core processing where a single address space is provided using several, physically disjoint, memory domains. LAMETER provides a comprehensive overview of many NUMA-related issues [110]. Although all memory addresses can be accessed by all cores transparently, significant losses of bandwidth can be expected when accessing memory located in a remote memory domain. The location of allocated memory usually gets determined when the data is initialized: a memory page gets placed in the domain of the first core to write to it ("first touch policy"). Even though page migration, i.e., transparent migration of memory pages to other memory domains when accessed by remote cores, is implemented by some operating systems (such as Linux, which is arguably the most relevant operating system in High-Performance Computing (HPC)), it is advised to access only local memory in NUMA nodes, if possible. Careful scheduling of threads is crucial to achieve this. Note that purely local access might not be possible under specific workloads with irregular data access patterns. For ease of programmability, NUMA designs are usually equipped with mechanisms to assure cache coherency, denoted as cache-coherent Non-Uniform Memory Access (ccNUMA). This thesis is restricted to such systems, and any mention of NUMA designs implies cache coherency.

NUMA designs naturally occur when several CPUs are installed in a single, shared-memory, compute node. In that case, the memory domains are usually connected via a proprietary interconnect such as Intel's QuickPath Interconnect (QPI). Some recent CPU architectures feature NUMA designs already on a single chip: an important new architectural feature of HSW is Cluster on Die (CoD), which divides the CPU socket into two equally sized NUMA domains. Hence, in case of the present HSW chip with 14 cores, each NUMA domain contains 7 cores. This feature can be disabled manually at boot time, but it has been enabled throughout this work. The vendor claims that the performance of NUMA-aware workloads which are limited by the last level cache or main memory bandwidth can be increased by using CoD. However, on the other hand, severe performance degradation can occur for non-NUMA-aware workloads like bandwidth-limited applications using inappropriate thread scheduling.

Accelerator architectures. The accelerator architectures Intel Xeon Phi 5110P "Knight's Corner" (KNC), Intel Xeon Phi 7250 "Knight's Landing" (KNL), NVIDIA Tesla K20X "Kepler" (K20), NVIDIA Tesla K20m "Kepler" (K20m), and NVIDIA Tesla P100 "Pascal" (P100) are significantly different than the multi-core CPUs from an architectural point of view. One obvious difference is the number of cores: for instance, KNL features 68 cores which is well beyond the maximum core count of even current multi-core CPUs. The core counts of the NVIDIA architectures always include all (very lightweight) "CUDA cores." On the Kepler GPUs, 192 CUDA cores are present on each Streaming Kepler Multiprocessor (SMX), of which there are 14 (13) on K20 (K20m). On Pascal, each Streaming Pascal Multiprocessor (SM) (of which P100 features 56) consists of 64 CUDA cores. The clock frequency of the accelerator architectures is significantly lower than for the multi-core CPUs. However, their large core count including wide SIMD units results in a considerably higher peak floating point performance. In contrast to the considered multi-core CPUs, KNC and KNL feature up to four SMT threads per core. For optimal memory bandwidth, multiple threads have to be placed on each core. One major difference between the Intel many-core CPUs and the NVIDIA GPUs is the significantly larger last level cache of the former.

The memory bandwidth of the accelerators exceeds the multi-core CPUs by a large margin. This is particularly the case for the newest accelerator architectures KNL and P100 which are equipped with novel memory technologies. In that respect, NVIDIA relies on the High Bandwidth Memory 2 (HBM2) technology whereas Intel bets on Multi-Channel Dynamic Random

Access Memory (MCDRAM). Besides 16 GB of MCDRAM, the KNL chip features 96 GB of Double Data Rate Fourth-Generation (DDR4) Synchronous Dynamic Random Access Memory (SDRAM) memory (at less than 100 GB/s saturated peak memory bandwidth), with the former functioning either as a cache for the latter ("cache mode"), or as a separate NUMA domain ("flat mode"). Highest performance can be achieved if the whole data set resides in the MCDRAM and the chip is operated in flat mode. This is also the preferred operating mode of KNL in this work, and the DDR4 memory partition is dismissed in table 2.1b and in all experiments. In contrast to this, the P100 does not feature lower-bandwidth memory besides its HBM2. However, CUDA offers a feature called "Unified Memory," which allows to regard the memory of the host CPU and the GPU as a single memory pool with a shared address space and transparent page migration between the two domains. The potential overhead of this mechanism is significant [112, 115], and it has not been used in this work. The relatively low Main Memory (MEM) capacity of the accelerators often poses an obstacle when it comes to programming this kind of architectures. While this is especially true on the older architectures KNC and K20(m), the vendors have taken note of this issue and equipped their latest accelerator generations with a significantly larger main memory. Note that Error-Correcting Code (ECC) memory protection has always been enabled on the Kepler GPUs. Those are the only considered devices where this can be controlled – all other hardware architectures have always enabled ECC.

The accelerator architectures KNC, K20(m), and P100 cannot be operated standalone, but require a host CPU to which the accelerator is connected via Peripheral Component Interconnect Express (PCIe). The maximum theoretical bidirectional bandwidth of this interconnect (using the maximum of 16 PCIe lanes) sums up to 8 GB/s for PCIe 2.0 (KNC and K20(m)) and 16 GB/s for PCIe 3.0 (P100). As this is much lower than all on-chip bandwidths, data transfers between the host CPU and the accelerator should be avoided if possible. The "accelerator" operating mode suggests that the code is still executed on the host CPU, and selected compute- or bandwidth-intensive and time-consuming kernels are executed on the accelerator. The KNC, however, can also be operated in "native" mode, which means that the entire program is executed on the accelerator card. Due to the weak single-core performance of KNC, the degree of parallelism should be as high as possible. As this is the case to a sufficient degree for the present applications, KNC has been operated in native mode throughout this work. This operating mode also helps in avoiding host-accelerator data transfers and contributes to ease of programmability. KNL breaks with this tradition, as it can also be operated

self-hosted, i.e., without the need for a host CPU. This allows for a homogeneous node architecture. On the one hand, this comes with significant advantages in usability and low overhead for well-parallelized workloads. On the other hand, if an application contains a significant serial portion, the relatively low single-core performance of this architecture quickly turns out to be hazardous for overall performance. Due to this trade-off, it is hard to tell whether one or the other operating mode can be considered better, and only the future will show whether homogeneous or heterogeneous node designs are going to prevail in the long run.

SIMD and SIMT processing. SIMD processing is an elementary feature of current CPU architectures. The basic idea is that a single instruction works with multiple data elements. The number of data elements processed in this way is called the "SIMD width." On the considered CPU architectures the SIMD width for double precision data is between four (real data) and two (complex data). For single precision data, those numbers double. A necessary premise for SIMD processing to work is vectorization, i.e., the formulation of the code in a way which allows single instructions to work on multiple data. This holds both for arithmetic operations and loading/storing of data. A related concept present on GPUs is Single Instruction, Multiple Threads (SIMT) processing. As opposed to the SIMD width, SIMT features the concept of "warps," with a warp being a group of threads which execute one common instruction at a time. While being similar to SIMD, the CUDA programming guide explains that "a key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread" [45]. The analogy between SIMT and SIMD processing has been noted already in the early days of general purpose computing on GPUs by Volkov and Demmel, stating that a "warp is a stream of vector instructions" and "scalar threads are then vector elements" [172]. On all current NVIDIA architectures, the warp size is 32. This is significantly larger than the aforementioned SIMD width of the considered CPU architectures. While it is certainly possible to program GPU architectures on a thread-level granularity, any divergence of threads in a warp causes a serialization of execution. Hence, optimal execution efficiency is achieved only if all threads of the warp have the same execution path. For instance, loading data from main memory is best done in a coalesced way, i.e., subsequent threads of a warp access subsequent memory locations. One peculiarity on GPU architectures regarding data accesses is that memory request by a warp are split into separate 128-byte memory requests if each

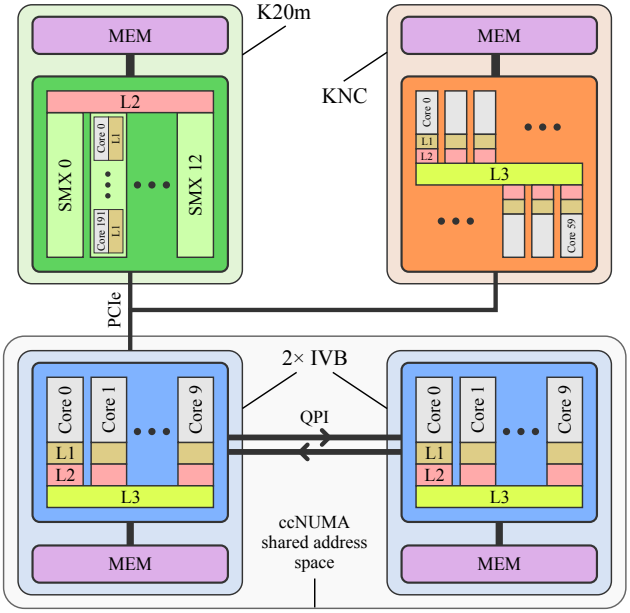


Figure 2.1: A heterogeneous compute node.

thread accesses more than 4 bytes [45]. This holds for all GPU architectures considered in this work. The SIMD equivalent of coalesced loads are vectorized load instructions. In summary, although being fundamentally different, SIMD and SIMT processing are similar in a sense that data structures and algorithms which are equipped for efficient execution with one of both are good candidates to also work well with the other.

Heterogeneous node architectures. Figure 2.1 shows a strongly heterogeneous compute node with a very complex architecture, as it consists of two IVB CPUs, and one K20m and KNC accelerator each. Each component has its own physical main memory ("MEM") domain. The IVB sockets are connected via two unidirectional QPI links providing a cache coherent shared address space. The NUMA architecture becomes obvious from the figure. Despite the shared address space, careful memory placement across the IVB sockets is important. The QPI links can deliver up to 32 GB/s in total. Despite the fact that this is not much lower than the main memory bandwidth of IVB, frequent remote memory accesses are likely to cause performance problems as each memory interface is shared between all local and remote accesses. The accelerators are connected to the host CPUs via PCIe with a maximum bidi-

rectional bandwidth of 8 GB/s. In the context of often data-transfer-intensive sparse linear algebra algorithms, transfers over those slow data paths should be avoided. Hence, the data should be kept on an accelerator or a single CPU socket if possible.

2.2 Large-Scale Compute Systems

Large-scale experiments are conducted on five compute systems, some of which differ substantially in terms of their architecture. An important means for the classification of supercomputers is the aforementioned TOP500 list, which sorts systems according to their achieved High Performance Linpack (HPL) [87] performance. HPL is a scalable and compute-intensive benchmark that solves a dense linear system in double precision floating point arithmetic. It usually gives a more realistic estimate about the practically achievable floating point performance of a computer than the theoretical P_{peak} value, which is the reason why it is used as the sorting criterion in the TOP500 list. However, floating point arithmetic performance may not be a critical bottleneck for many applications. (This also holds for many sparse linear algebra algorithms, where the execution speed is frequently limited by main memory bandwidth as will be shown in chapter 4.) This led to the proposition of further prospective standard benchmark codes beyond HPL to obtain more realistic evaluations of compute systems for this kind of applications. One of the most prominent HPL alternatives is the High Performance Conjugate Gradient (HPCG) benchmark as proposed by Dongarra et al. [60]. It uses a Preconditioned Conjugate Gradient (PCG) algorithm to solve the Poisson differential equation on a regular 3-dimensional grid which is discretized with a 27-point stencil. The involved building blocks are similar to the ones present in this work, which qualifies HPCG as a further evaluation criterion for compute systems here, although HPCG is not of the same prominence as HPL today. While the coverage of published HPCG performance numbers does not match all TOP500 systems, the HPCG website [86] lists the HPCG performance of some relevant TOP500 systems. Another important classification criterion of compute clusters is the bisection bandwidth, which is the minimum bandwidth between two partitions of the network. The used systems will be briefly explained in the following, and key performance indicators like the HPL/HPCG performance and bisection bandwidth are given if available.

SuperMUC Phase 2 [121]. This Lenovo NeXtScale nx360M5 WCT system, installed at the Leibniz Supercomputing Center (LRZ) in Garching, Germany, is a typical representative of the class of standard multi-core CPU clusters. Each of its 3072 nodes features 2 HSW chips, and the nodes are connected with an Infiniband FDR14 network with a bisection bandwidth of 5.1 TB/s. Each group of 512 nodes forms an "island," and the intra-island bandwidth is by a factor of $4\times$ larger than the inter-island bandwidth. The HPL performance sums up to 2.81 PFLOP/s, which resulted in a TOP500 debut at position 21 in 06/2015. In the latest edition of the list (06/2017), SuperMUC Phase 2 still holds position 41.

Emmy [143]. Emmy is a NEC LX-2400 machine installed at the Erlangen Regional Computing Center (RRZE) in Erlangen, Germany. Similarly to SuperMUC Phase 2, it is primarily a classical multi-core CPU cluster with 2 IVB* sockets installed in 544 nodes. On top of that, the system contains 16 specialized nodes, some of which are equipped with 2 KNC accelerators, some with 2 K20m GPUs, and some with one accelerator of each kind. All of those accelerated nodes are driven by two IVB host CPUs. The nodes are connected with a QDR Infiniband network. With a HPL performance of 197 TFLOP/s, it debuted in the TOP500 in 11/2013 at position 210. As of 11/2015, it is no longer among the TOP500 systems.

Piz Daint [43]. Piz Daint is a Cray XC30 system located at the Swiss National Supercomputing Centre (CSCS) in Lugano, Switzerland. It consists of 5272 nodes, each equipped with one K20 GPU and one SNB CPU. In contrast to SuperMUC Phase 2 and Emmy, this system is a typical representative of GPU-accelerated systems. The nodes are connected with a Cray Aries network with a dragonfly topology and a total bisection bandwidth of 33 TB/s. With its HPL performance of 6.27 PFLOP/s, it entered the TOP500 list in 11/2013 at position 6, and it would still hold position 13 in the 06/2017 edition, if it were not replaced by Piz Daint v2 (see next paragraph). The HPCG performance adds up to 125 TFLOP/s which puts Piz Daint on position 13 of the HPCG ranking as of November 2016.

Piz Daint v2 [42]. Piz Daint was replaced by a Cray XC/40XC50 in late 2016 to a system which will be called "Piz Daint v2" in this work. In its final configuration, it contains 5320 hybrid nodes (each with a P100 GPU and Intel Haswell host CPUs) and 1431 CPU-only nodes with Intel Broadwell chips. Neither the host CPUs of the hybrid partition nor the CPU-only nodes are taken into

account in this work. A first, smaller configuration entered the TOP500 list in 11/2016 at position 8 with a HPL performance of 9.78 PFLOP/s, and the final configuration achieves list position 3 in 06/2017 with a HPL performance of 19.59 PFLOP/s. The HPCG performance of the final configuration adds up to 477 TFLOP/s which puts it on position 4 of this ranking as of June 2017. The network topology is the same as in its predecessor, and CSCS does not publish the bisection bandwidth.

Oakforest-PACS [92]. Besides multi-core CPU and GPU-accelerated clusters, there is a third important class of supercomputers: clusters comprised of self-hosted many-core nodes. Oakforest-PACS, a Fujitsu PRIMERGY CX1640 M1 machine located at the Joint Center for Advanced High Performance Computing (JCAHPC) in Tokyo, Japan, is one of the largest machines of this kind. (The world's fastest public supercomputer as of 11/2016, Sunway TaihuLight, is of similar architecture.) Each of the 8208 nodes of Oakforest-PACS is equipped with a KNL chip and Intel Omni-Path is used as the interconnect. It has debuted in the TOP500 list in 11/2016 on position 6 with a HPL performance of 13.55 PFLOP/s. As of 06/2017, it still holds position 7. The HPCG performance of 386 TFLOP/s results in position 5 of this ranking as of 06/2017.

2.3 Software Test Bed and Methodology

Several programming models and Application Programming Interfaces (APIs) are used throughout this work. They are very briefly introduced in the following. Afterwards, relevant software versions are specified, together with some notes about the benchmarking and measurement methodology used in this work. This enables reproducibility of the presented performance results.

MPI. The Message Passing Interface (MPI) [131] is a standardized message passing system for parallel computing, developed since 1992. Communication and data exchange between processors is done via explicit message passing. "The MPI standard includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and a profiling interface." [131] Numerous

implementations of the standard are available, and the ones relevant to this work are summarized below.

OpenMP. The development of the Open Multi-Processing (OpenMP) standard started in 1997. The intention was to create an alternative to message passing (e.g., via MPI) to be used in conjunction with cache-coherent shared-memory multiprocessor architectures. It primarily consists of compiler directives used for thread creation and work sharing. Library functions and environment variables can be used to influence the runtime behavior. In a typical cluster of shared-memory compute nodes, OpenMP can be used together with MPI.

SIMD programming. Besides writing source code in a high-level language, such as C or C++, many compilers offer so-called intrinsic functions. In this work, such functions are used for the creation of code which the compiler cannot map to a processor's SIMD units efficiently. Besides, there exist compiler directives to guide the compiler to generate efficient SIMD code. In this work, both techniques – intrinsics and directives – are used and marked accordingly.

CUDA. In 2007, NVIDIA launched its parallel programming interface CUDA which enables GPU architectures for general purpose processing. The standard approach follows a host-accelerator model, where selected functions are expressed as CUDA kernels which are launched from a thread on the host CPU.

On all current NVIDIA architectures, 32 threads form a warp, which is the basic unit of SIMT processing. On top of that, a number of threads (usually several warps) are grouped in a *thread block* of size `blockDim.x` $\times$ `blockDim.y` $\times$ `blockDim.z`, which organizes threads in a one-, two-, or three-dimensional way and assigns each thread a unique index in this block `threadIdx.x/y/z`. The threads blocks are organized in a one- or two-dimensional *grid* of size `gridDim.x` $\times$ `gridDim.y`, in which each block has a unique `blockIdx.x/y`. The dimensions of the grid and thread blocks have to be specified at the time of kernel launch.

CUDA offers extensions to standard programming languages like C/C++ for the implementation of kernels. In addition, CUDA includes several software libraries to enable easy GPU utilization, e.g., cuSPARSE for sparse matrix computations.

Software versions. The Intel C Compiler (ICC) is always used for compilation of CPU code throughout this work. The present ICC versions are 15.0.1 on SNB, 16.0.3 on IVB(*), HSW, and KNC, and 17.0.1 on KNL. For the compilation of GPU code, CUDA version 7.0 is used on K20 and K20m, and version 8.0 on P100. Basic Linear Algebra Subprograms (BLAS) calls are handled by the Intel Math Kernel Library (MKL) 11.3 on the multi-core CPUs and KNC, MKL 2017.0.1 on KNL, and the CUDA-appendant cuBLAS on the GPUs. Interprocess communication via message passing is achieved using IBM MPI 1.4 on SuperMUC Phase 2, Cray MPICH 7.2.2 on Piz Daint (7.5 on Piz Daint v2), and Intel MPI 5.1.3 on Emmy (2017.0.1 on Oakforest-PACS).

Benchmarking methodology. Besides specifying the hard- and software test bed, the benchmarking methodology must be well defined in order to obtain reproducibility of the presented performance results. Guidelines for the publication of performance data with special consideration of reproducibility and interpretability are given, e.g., by Hoefler and Belli [84]. If not noted otherwise, all presented performance data in this work represents average performance of a sufficiently large number of test executions, i.e., large enough such that no significant fluctuations of observed performance occur between distinct test runs. Having iterative sparse solvers in mind, it is a valid approach to report the average performance of computational building blocks, as there are also many subsequent executions of them when used in solvers. Detailed information about the methodology (like, e.g., the number of iterations) are given in the according sections of this work.

2.4 Test Matrices

The analysis of this work is done by means of a set of test matrices which are summarized in table 2.2. In all cases, real or complex double precision values are used, i.e., a single matrix value occupies 8 (real) or 16 (complex) bytes. Index values are always stored using 4-byte integers. A matrix is considered "large" if it does not fit into any cache of the used hardware architectures. According to table 2.1, the HSW architecture has the largest available LLC with 35 MiB. The absolute minimum storage requirement for a general sparse matrix is to store the value and column index of each non-zero element (see section 4.2.2). Assuming real double precision (i.e., 8-byte) data and 4-byte indices, this adds up to 12 bytes per element, and a test matrix must contain at least $n_{\text{nz}} \geq (35 \times 10^6)/12 \approx 3 \times 10^6$ elements to be qualified for the group of "large" matrices.

#	Test case	n ($\times m$)	n_{nz}	n_{nzr}	$\beta^{C=16}_{\sigma=1\rightarrow256}$	$\sigma^{C=16}_{\text{opt}}$	c_v
1	Spin-N_{Up}	$\frac{N_{\text{Up}}!}{(N_{\text{Up}}/2)!^2}$	nn_{nzr}	$\frac{N_{\text{Up}}}{2}+1$	$0.88 \rightarrow 0.98$	2^6	0.178
2	Topi-N_x-N_y-N_z	$4N_xN_yN_z$	nn_{nzr}	≈ 13	$1.00 \rightarrow 1.00$	2^0	0.022
3	Graphene-N_x-N_y	N_xN_y	nn_{nzr}	≈ 4	$1.00 \rightarrow 1.00$	2^0	0.012
4	MATPDE-N_x-N_y	N_xN_y	nn_{nzr}	≈ 5	$1.00 \rightarrow 1.00$	2^0	0.006
5	ML_Geer	1,504,002	110,879,972	73.72	$1.00 \rightarrow 1.00$	2^0	0.034
6	RM07R	381,689	37,464,962	98.16	$0.63 \rightarrow 0.93$	2^9	0.700
7	kkt_power	2,063,494	14,612,663	7.08	$0.54 \rightarrow 0.92$	2^9	1.045
8	Hamrle3	1,447,360	5,514,242	3.81	$1.00 \rightarrow 1.00$	2^0	0.407
9	pwtk	217,918	11,634,424	53.39	$0.99 \rightarrow 1.00$	2^0	0.089
10	shipsec1	140,874	7,813,404	55.46	$0.89 \rightarrow 0.98$	2^7	0.200
11	consph	83,334	6,010,480	72.13	$0.94 \rightarrow 0.97$	2^7	0.265
12	pdb1HYS	36,417	4,344,765	119.31	$0.84 \rightarrow 0.97$	2^8	0.267
13	cant	62,451	4,007,383	64.17	$0.90 \rightarrow 0.98$	2^7	0.219
14	webbase-1M	1,000,005	3,105,536	3.11	$0.45 \rightarrow 0.67$	2^{15}	8.161
15	rail4284	4,284 × 1,092,610	11,279,748	2,632.99	$0.28 \rightarrow 0.73$	n	1.598
16	dense2	2,000	4,000,000	2,000.00	$1.00 \rightarrow 1.00$	2^0	0.000
17	cop20k_A	121,192	2,624,331	21.65	$0.86 \rightarrow 0.98$	2^7	0.479
18	rma10	46,835	2,374,001	50.69	$0.70 \rightarrow 0.96$	2^8	0.548
19	mc2depi	525,825	2,100,225	3.99	$1.00 \rightarrow 1.00$	2^0	0.019
20	qcd5_4	49,152	1,916,928	39.00	$1.00 \rightarrow 1.00$	2^0	0.000
21	mac_econ_fwd500	206,500	1,273,389	6.17	$0.37 \rightarrow 0.82$	2^{11}	0.719
22	scircuit	170,998	958,936	5.61	$0.49 \rightarrow 0.83$	2^{12}	0.783

Table 2.2: Summary of basic matrix characteristics. Matrices #1-3 are quantum physics application matrices, matrix #4 is a scalable test case from the Matrix Market [30] (see text for description of #1-4), matrices #5-8 are large corner case benchmark matrices, matrices #9-14 are the large, sparse and square matrices of the Williams group of the University of Florida sparse matrix collection [51] and matrices #16-22 are the remaining matrices of this group. The matrices have dimension $n \times m$ ($n \times n$ is m is omitted) with a total number of non-zero entries n_{nz} and an average number of non-zero entries per row $n_{\text{nzr}} = n_{\text{nz}}/n$. The last columns describe SELL-C-σ-specific properties (see section 5.2): β denotes the chunk occupancy of each matrix without ($\sigma = 1$) and with sorting ($\sigma = 256$) for a chunk height of $C = 16$. The sorting scope to achieve a chunk occupancy of 95% with $C = 16$ is denoted by σ_{opt}, and c_v labels the coefficient of non-zero count variation of a sparse matrix.

Matrices #1-3 arise from applications relevant to the Equipping Sparse Solvers for Exascale (ESSEX) project [6]. Spin-N_{Up} resembles the Hamilton operator for the Heisenberg XXZ spin chain model with S_z symmetry and N_{Up} spins [142]. Topi-N_x-N_y-N_z is the Hamilton operator of an $N_x \times N_y \times N_z$ sample of a topological insulator material [100]. This matrix is complex-valued. Finally, Graphene-N_x-N_y relates to the standard tight-binding Hamiltonian on the honeycomb lattice with an on-site disorder term [37, 137].

Matrix #4, MATPDE-N_x-N_y, is a scalable test matrix from the Matrix Market [30]. This non-symmetric matrix represents a five-point finite difference discretization of a two-dimensional variable-coefficient linear elliptic equation on an $N_x \times N_y$ grid with Dirichlet boundary conditions.

To demonstrate wider applicability of the developed building blocks, a range of benchmark matrices (#5-22) have been selected which will be used in the according building block's performance analyses. Regardless of their original data type, they will be used with real double precision data. Matrices #5-8 represent 4 corner cases in terms of low/high average number of nonzero entries per sparse matrix row n_{nzr} and chunk occupancy of a SELL-C-σ matrix β (see chapter 5), which is very useful for performance analysis and benchmarking. The remaining matrices constitute the "Williams" group of matrices in the University of Florida sparse matrix collection [51]. This collection has originally been proposed by Williams et al. and represents "a wide variety of actual applications, including finite element method-based modeling, circuit simulation, linear programming, a connectivity graph collected from a partial web crawl, as well as a dense matrix stored in sparse format" [178]. Nowadays, this set of matrices is commonly used in publications about sparse matrix algorithm performance [24, 40, 104]. Although those matrices are not necessarily directly relevant for the present algorithms and applications, they are useful for evaluating wider applicability of the developed building blocks. Furthermore, the very diverse nature of those matrices is helpful in identifying strengths and weaknesses of different implementations and hardware architectures.

3 Iterative Sparse Eigenvalue Solvers

The determination of relevant numerical building blocks in this thesis is guided by a representative selection of iterative sparse eigenvalue solvers. The goal of such solvers is to solve the standard eigenvalue problem,

$$H\vec{x} = \lambda\vec{x}\,, \tag{3.1}$$

where H is a sparse, potentially large, and assumed to be square matrix of size $n \times n$. The scalars $\lambda_1 \leq \lambda_i \leq \lambda_n$ are called the eigenvalues of H and $\vec{x}_1, \ldots, \vec{x}_n$ are the corresponding eigenvectors. The set of all eigenvalues $\lambda_1, \ldots, \lambda_n$ builds the *spectrum* of the matrix. An eigenvalue is called an *exterior* eigenvalue if it is located at the end of the spectrum, i.e., close to λ_1 or λ_n. Eigenvalues in the interior of the spectrum are called *interior* eigenvalues. An approximation to the full spectrum, i.e., a quantification of the number of eigenvalues per interval, is called the Density Of States (DOS).

The identification of performance-critical sparse linear algebra building blocks is done by means of some selected methods, which will be briefly introduced in the following sections. The level of detail of mathematical foundations is kept to a minimum and suitable references are given for further reading. The goal here is not to facilitate a deep understanding of those methods, but rather to obtain a sufficient overview and understanding of various sparse eigenvalue solvers. The presented methods in sections 3.1 to 3.4 reflect a wide range of solvers for different classes of eigenvalues.

3.1 Lanczos Method

The Lanczos method is an iterative algorithm proposed by Cornelius Lanczos [111]. The method is known to be especially useful in situations where only a few of H's exterior eigenvalues are sought, as information about those tends to emerge fairly early during the iterative process [75, Chapter 10]. A sketch of the Hermitian Lanczos algorithm (i.e., assuming Hermitian H) is

Algorithm 1 The Hermitian Lanczos algorithm.

1: $\vec{v}_1 \leftarrow$ random initial vector with norm 1
2: $\vec{v}_0 \leftarrow 0$
3: $\beta_1 \leftarrow 0$
4: **while** not converged **do**
5: $\quad \vec{w}_j \leftarrow H\vec{v}_j - \beta_j \vec{v}_{j-1}$ ▷ `spmv()`
6: $\quad \alpha_j \leftarrow \langle \vec{w}_j, \vec{v}_j \rangle$ ▷ `dot()`
7: $\quad$ IMTQLI(m,α,β) ▷ Find eigenvalues of T_{jj} and check for convergence
8: $\quad \vec{w}_j \leftarrow \vec{w}_j - \alpha_j \vec{v}_j$ ▷ `axpy()`
9: $\quad \beta_{j+1} \leftarrow ||\vec{w}_j||_2$ ▷ `nrm2()`
10: $\quad \vec{v}_{j+1} \leftarrow \vec{w}_j / \beta_{j+1}$ ▷ `scal()`
11: **end while**

shown in algorithm 1. The iterative process generates a sequence of orthogonal "Lanczos vectors" $\vec{v}_j$, which span a Krylov subspace of H. Using those, one can successively build a symmetric and tridiagonal matrix

$$T_{jj} = V_j^H H V_j = \begin{pmatrix} \alpha_1 & \beta_2 & & & & 0 \\ \beta_2 & \alpha_2 & \beta_3 & & & \\ & \beta_3 & \alpha_3 & \ddots & & \\ & & \ddots & \ddots & \beta_{j-1} & \\ & & & \beta_{j-1} & \alpha_{j-1} & \beta_j \\ 0 & & & & \beta_j & \alpha_j \end{pmatrix} \tag{3.2}$$

with $V_j = (\vec{v}_1, \ldots, \vec{v}_j)$ a dense matrix of size $n \times j$ and V_j^H its conjugate transpose. The eigenvalues of T_{jj} are approximate eigenvalues of H, with the exterior eigenvalues converging most quickly to the exterior eigenvalues of H. The eigenvalues of T_{jj} can be computed directly, e.g. using the implicit QL [61] method. This comes at very low cost since T_{jj} is tridiagonal and $j \ll n$.

The computational building blocks required in the presented Hermitian Lanczos algorithm are the Sparse Matrix-Vector Multiplication (SpMV) operation $\vec{y} \leftarrow H\vec{x} + \beta\vec{y}$ in line 5 of algorithm 1, as well as the BLAS-1 operations `dot()`, `axpy()`, `nrm2()`, and `scal()` in lines 6 and 8 to 10. Note that a non-Hermitian Lanczos algorithm would require similar building blocks, but also the multiplication of the conjugate transpose of H, H^H, with a vector. In a simple implementation, H^H can be stored explicitly to avoid the requirement of a further compute kernel for this operation.

3.2 Kernel Polynomial Method

The Kernel Polynomial Method (KPM) is a well-established polynomial expansion technique for the computation of the eigenvalue density and spectral properties of large sparse matrices. Being originally developed for the computation of eigenvalue densities and spectral functions [154], the KPM soon found applications in physics and chemistry. In recent applications the KPM has been used, e.g., for large-scale data analysis [27] and the counting of eigenvalues for the pre-determination of sub-space sizes in projection-based eigensolvers [57] (see also section 3.3). While a detailed analysis of the KPM can be found in the review by Weisse et al. [176], a brief overview will be given in the following.

One application of the KPM is the computation of the DOS, which is a basic quantity of interest for physics applications defined as

$$\rho(\lambda) = \sum_{i=1}^{n} \delta(\lambda - \lambda_i)\,, \tag{3.3}$$

with δ the Dirac delta function. A direct method for the computation of $\rho(\lambda)$ as given in eq. (3.3) would have to compute all eigenvalues λ_i of H, which is not feasible for large matrices. Instead, the KPM works with an alternative expression,

$$\rho(\lambda) = \mathrm{tr}\left[\delta(\lambda\mathbb{1} - H)\right]\,, \tag{3.4}$$

where $\mathbb{1}$ denotes the identity matrix and the sum of the trace $\mathrm{tr}[\dots]$ runs over all eigenvalues λ_i of H. Basically, the KPM works with a systematic expansion of the δ function in eq. (3.4). The KPM exploits the orthogonality properties and two-term recurrence for Chebyshev polynomials $T_m(x)$ of the first kind to successively compute the vectors

$$\vec{v}_m = T_m\left(\tilde{H}\vec{v}_0\right) \tag{3.5}$$

from a starting vector $\vec{v}_0$ with $1 \leq m \leq M/2$ and prescribed M through the recurrence

$$\vec{v}_1 = \tilde{H}\vec{v}_0\,, \tag{3.6}$$

$$\vec{v}_{m+1} = 2\tilde{H}\vec{v}_m - \vec{v}_{m-1}\,. \tag{3.7}$$

$\tilde{H}$ indicates a re-scaled and shifted matrix H with

$$\tilde{H} = a(H - b\mathbb{1})\,, \tag{3.8}$$

which is necessary because the spectrum of $\tilde{H}$ must be contained in the interval of orthogonality of the Chebyshev polynomials [-1;1]. The Lanczos method as briefly introduced in section 3.1 is a suitable way for the determination of the exterior eigenvalues λ_1 and λ_n which can be used to compute the scaling factor a and shift b as

$$a = (2 - \epsilon)/(\lambda_n - \lambda_1) \text{ and} \tag{3.9}$$
$$b = (\lambda_n + \lambda_1)/2 \tag{3.10}$$

with ϵ a small cut-off used to avoid stability problems. To avoid saving all $\vec{v}_m$ and restrict the number of saved vectors to two, the two scalar products corresponding to the Chebyshev moments

$$\eta_{2m} = \langle \vec{v}_m, \vec{v}_m \rangle \text{ and} \tag{3.11}$$
$$\eta_{2m+1} = \langle \vec{v}_{m+1}, \vec{v}_m \rangle \tag{3.12}$$

are computed in each iterative step. After the iterative and computationally intensive scheme is finished, spectral quantities are constructed in a computationally undemanding step from averaged Chebyshev moments using kernel polynomials to suppress Gibbs oscillations. Concretely, for the computation of the spectrally averaged DOS, the trace in eq. (3.4) can be approximated by a sum over R independent random initial vectors $\left\{ \vec{v}_0^{(1)}, \ldots, \vec{v}_0^{(R)} \right\}$ using the relation that

$$\mathrm{tr}\,[H] \approx (1/R) \sum_{r=1}^{R} \left\langle \vec{v}_0^{(r)}, H\vec{v}_0^{(r)} \right\rangle . \tag{3.13}$$

The independence of the random initial vectors allows to combine all of them into a single KPM iteration. As will be motivated in sections 8.1 and 10.1, this can be done to increase the performance of the computation. The final KPM algorithm for DOS computation is shown in algorithm 2, where $\vec{W}$ and $\vec{V}$ denote blocks of R vectors. The computational building blocks needed in algorithm 2 are a shifted and general Sparse Matrix-Multiple Vectors Multiplication (SpMMV) kernel in line 5 and block vector versions for the calculation of norms (line 6) and dot products (line 7). Although the presented version of the KPM is specialized towards the computation of the DOS, the short name "KPM" (instead of, e.g., KPM-DOS [100]) is used throughout this work for the sake of brevity.

Algorithm 2 Blocked version of the KPM.

1: $\vec{V} \leftarrow R$ random initial vectors
2: $\vec{W} \leftarrow R$ zero vectors
3: Initialization and computation of η_0, η_1
4: **for** $m = 1$ to $M/2$ **do**
5: $\quad \vec{W} \leftarrow 2a(H - b\mathbb{1})\vec{V} - \vec{W}$ ▷ Equations (3.7) and (3.8)
6: $\quad \eta_{2m} \leftarrow \langle \vec{V}, \vec{V} \rangle$ ▷ Equation (3.11)
7: $\quad \eta_{2m+1} \leftarrow \langle \vec{W}, \vec{V} \rangle$ ▷ Equation (3.12)
8: $\quad$ SWAP$(\vec{W}, \vec{V})$
9: **end for**
10: Inexpensive computation of DOS from $\eta_{0,\ldots,M}$

3.3 Chebyshev Filter Diagonalization

The problem of finding interior eigenvalues of a symmetric (or Hermitian) matrix can be specified as follows: Given a symmetric (or Hermitian) matrix H, compute the N_T eigenpairs $(\lambda_i, \vec{v}_i)$ within a prescribed target interval, $\lambda_i \in I_T = [\underline{\lambda}, \overline{\lambda}]$. In current applications, the target interval typically contains up to a few hundred eigenvalues of a matrix H which has approximately $10^6, \ldots, 10^{10}$ rows [137]. Chebyshev Filter Diagonalization (ChebFD) is a straightforward scheme for the solution of this problem. While this work only presents a brief overview of this method, a thorough explanation of the ChebFD method is available by PIEPER et al. [137].

Being based on polynomial filter functions, ChebFD has a lot in common with the KPM. In ChebFD, filter polynomials of degree N_p, which are obtained from Chebyshev expansions of window functions with expansion coefficients c_n and damping factors g_n for $0 \leq n \leq N_p$, are used to project the subspace onto the target space of sought eigenvectors [137]. The precise value of eigenvalues in the target interval N_T is normally not known prior to the computation, but can be estimated from the DOS (e.g., by applying the KPM, see section 3.2). The eigenpairs in the target interval can be obtained as follows: Start with sufficiently many random search vectors $\vec{x}_k$, for $k = 1, \ldots, N_S$ with $N_S \geq N_T$, and compute the sought eigenvalues and -vectors from the filtered vectors $p[H]\vec{x}_k$. The basic procedure of ChebFD can be summarized as:

Algorithm 3 Blocked version of the polynomial filter application procedure in ChebFD.

1: $\vec{U} \leftarrow (\alpha H + \beta \mathbb{1})\vec{X}$
2: $\vec{W} \leftarrow 2(\alpha H + \beta \mathbb{1})\vec{U} - \vec{X}$
3: $\vec{X} \leftarrow g_0 c_0 \vec{X}_k + g_1 c_1 \vec{U} + g_2 c_2 \vec{W}$
4: **for** $n = 3$ to N_p **do**
5: swap$(\vec{W}, \vec{U})$
6: $\vec{W} \leftarrow 2(\alpha H + \beta \mathbb{1})\vec{U} - \vec{W}$
7: $\vec{X} \leftarrow \vec{X} + g_n c_n \vec{W}$
8: **end for**

Pre-processing

1. Determine the parameters a, b such that the spectrum of H is included in $[a, b]$. This can be done, e.g., using the Lanczos method as described in section 3.1.
2. Compute the DOS of H, e.g., using the KPM as presented in section 3.2. Estimate the number of target vectors N_T from it and choose the number of search vectors $N_S \gtrsim 2N_T$ accordingly.
3. Estimate the size of the search interval I_S, which includes the target interval I_T, from N_S and choose the filter polynomial degree N_P accordingly.

Polynomial filtering

4. Construct N_S random search vectors $\vec{x}_1, \ldots, \vec{x}_{N_S}$.
5. Apply the polynomial filter: $\vec{x}_k \leftarrow p[H]\vec{x}_k$ for $k = 1, \ldots, N_S$ as illustrated in algorithm 3 with $\alpha = 2/(b-a)$ and $\beta = (a+b)/(a-b)$.
6. Orthogonalize the filtered search vectors and compute the Ritz pairs $(\tilde{\lambda}_k, \tilde{\vec{v}}_k)$ together with their residuals
7. Restart from step 5 if not converged.

The computationally most expensive step is the application of the polynomial filter (step 5). Similar to the KPM, the independence of random initial vectors allows for a straightforward formulation of this step to use block vectors. In sections 8.2 and 10.2, the use of block vectors in ChebFD is motivated by estimating and demonstrating the concomitant performance gains.

Algorithm 3 illustrates the application of the polynomial filter (step 5 above) to a block of vectors $\vec{X}$ using the auxiliary block vectors $\vec{U}$ and $\vec{W}$. Similar to the KPM, the block vector formulation necessitates a shifted and general SpMMV kernel (lines 1, 2 and 6). Besides, a block vector implementation of the BLAS-1 kernel `axpy()` is required (lines 3 and 7).

After the application of the filter, the filtered search vectors must be orthogonalized (step 6 in the above-illustrated scheme). This can be achieved, e.g., using Singular Value QB (SVQB) [160], a method which demands matrix-matrix multiplications on block vectors, i.e., Tall & Skinny General Dense Matrix-Matrix Multiplication (T&S-GEMM) operations. The computational demand of this step in the context of ChebFD depends in the polynomial degree N_P: the larger it is, the more dominating is the application of the filter compared to the other steps of ChebFD. For constant N_T, the polynomial degree scales with the problem size, and already for medium-scale matrices, N_P is in the range of thousands [137]. Hence, the presence of highly efficient routines for T&S-GEMM operations may not be critical for high ChebFD performance, and potentially inefficient "standard" General Dense Matrix-Matrix Multiplication (GEMM) implementations could be used if N_P is high enough. Nevertheless, it is worthwhile to investigate the efficiency of the involved T&S-GEMM operations to operate at high efficiency on all scales.

3.4 Block Jacobi-Davidson QR Method

The last considered problem is to find l exterior eigenvalues of the matrix H with $l \ll n$. A suitable but basic method for the solution of this is Davidson's method [50], which is a generalization of the Lanczos method (see section 3.1) and can be seen as a preconditioned version of it. Combining outer iterations of Davidson type with efficient preconditioning and inner iterations to solve auxiliary linear systems yields the Jacobi-Davidson QR (JDQR) method as first proposed by Fokkema et al. [69]. JDQR is significantly faster than Davidson's method for finding several exterior eigenvalues. However, this method reveals numerical weaknesses when it comes to multiple or clustered eigenvalues. In this case, block variants of Jacobi-Davidson (JD) methods (which were originally addressed by Stathopoulos and McCombs [157]) are known to be more robust than their single vector counterparts. This work considers the Block Jacobi-Davidson QR (BJDQR) method for the solution of the aforementioned exterior eigenvalue problem where n_b denotes the number of vectors in a block. Contrary to KPM (section 3.2) and ChebFD (section 3.3), both of which are originally single vector methods which can be formulated as block algorithms easily, BJDQR is designed as a block algorithm from the start. A theoretical foundation of the BJDQR method is available by Röhrig-Zöllner et al. [142], and a short overview of this method is given in the following.

The starting point is the invariant subspace $\mathcal{V} = \text{span}\{\vec{v}_1, \ldots, \vec{v}_l\}$ spanned by the l exterior eigenvectors which are sought. Considering an orthonormal basis of this subspace, the standard eigenvalue problem as given in eq. (3.1) can be re-formulated as

$$\begin{cases} HQ - QR & = 0\,, \\ -\frac{1}{2}Q^H Q + \frac{1}{2}\mathbb{1} & = 0\,, \end{cases} \tag{3.14}$$

where $HQ = QR$ is a partial Schur decomposition of H with an orthogonal matrix Q of size $n \times l$ and an upper triangular matrix R of size $n \times n$. The diagonal entries of R are the eigenvalues of H. While omitting the mathematical derivation, it can be said that one of the main contributors to the overall runtime is the application of the JD operator to a block vector $\vec{X}$ containing n_b vectors with $n_b \ll n$ in each iteration of an inner iterative sparse solver, e.g., Generalized Minimum Residual (GMRES) [142],

$$\vec{Y} \leftarrow \underbrace{\left(\mathbb{1} - QQ^T\right)}_{\text{projection}} \underbrace{\left(H - \tilde{\lambda}\mathbb{1}\right)\vec{X}}_{\text{shifted SpMMV}}\,. \tag{3.15}$$

This can be split into the shifted SpMMV,

$$\vec{Y} \leftarrow \left(H - \tilde{\lambda}\mathbb{1}\right)\vec{X}\,, \tag{3.16}$$

where the Ritz values $\tilde{\lambda}$ act as shift and $\tilde{\lambda}$ is different for each vector in the block, and the projection,

$$S \leftarrow Q^T\vec{Y}\,, \tag{3.17}$$

$$\vec{Y} \leftarrow \vec{Y} - QS\,. \tag{3.18}$$

As $l, n_b \ll n$, it holds that Q, $\vec{X}$ and $\vec{Y}$ are very "tall & skinny" dense matrices and S is a small matrix of size $l \times n_b$. Besides the block vector width n_b, BJDQR features an additional parameter m which determines the current dimension of the search space. One characteristic of the BJDQR algorithm is that this dimension is subject to growth during the iteration. When it grows too large, it needs to be shrunken to its original value $m_{\text{min}} \leq m$. Speaking in terms of numerical kernels, this procedure requires an in-place multiplication of a block vector $\vec{W} = (\vec{w}_1, \ldots, \vec{w}_m)$ with a small matrix C of size $m \times m_{\text{min}}$ to update the first m_{min} vectors of $\vec{W}$,

$$(\vec{w}_1, \ldots, \vec{w}_{m_{\text{min}}}) \leftarrow \vec{W}C\,. \tag{3.19}$$

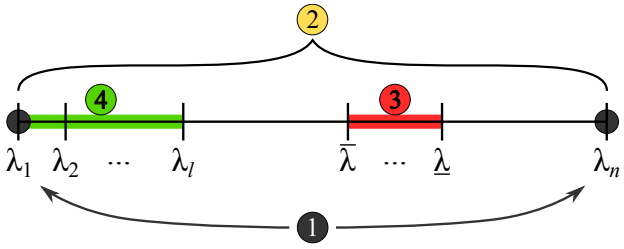


Figure 3.1: Considered classes of sought eigenvalues and corresponding methods used in this work: (1) denotes a few exterior eigenvalues which can be found with the Lanczos method (see section 3.1), (2) is the DOS which can be estimated with the KPM (see section 3.2), (3) highlights "some" interior eigenvalues which can be found using ChebFD (see section 3.3), and (4) marks "some" exterior eigenvalues for which BJDQR is a suitable method (see section 3.4).

Both the projection and subspace shrinking involve GEMM operations on tall & skinny dense matrices. In eq. (3.17), an inner product of tall & skinny dense matrices is computed which yields the small dense matrix S. In further reference, this operation will be called Tall & Skinny Matrix Transposed-Tall & Skinny Matrix Multiplication (TSMTTSM). The multiplication of a tall & skinny with a small matrix as needed in eq. (3.18) will be called Tall & Skinny Matrix-Small Matrix Multiplication (TSMM) and its in-place counterpart as required in eq. (3.19) TSMM-inplace. A peculiarity of an implementation of eq. (3.19) is the aliasing of the output with the first input matrix. This is against the BLAS standard, which states that "unless specified otherwise, only input-only arguments (specified with the `const` qualifier), may be legally aliased on a call to the C interface to the BLAS" [8]. Thus, a copy of $\vec{W}$ needs to be passed as the input matrix if off-the-shelf BLAS implementations were used.

Conclusively, the most important numerical kernels of the BJDQR method are shifted SpMMV, and the three aforementioned T&S-GEMM operations.

3.5 Summary

This chapter introduced four different numerical methods for different classes of eigenvalue problems. Figure 3.1 visualizes which part of the spectrum is addressed by the different solvers. While each of the methods has its own characteristics, recurring building blocks can be found. This thesis covers hardware-efficient and highly parallel implementations of those building blocks for all current HPC architectures. Although a selection of sparse eigenvalue solvers has been used to identify relevant building blocks, the results of this thesis are not limited to those methods. In fact, the selected building

blocks appear in many more sparse linear algebra algorithms, which makes the results of this work applicable to a much wider range of solvers. For instance, linear solvers based on Krylov subspaces, like the selection analyzed by Anzt et al. containing IDR(s), QMR, BiCGSTAB, and others [9], consist of similar building blocks. A brief summary of the occurring building blocks in the presented eigenvalues solvers, as well as their typical communication patterns in distributed memory environments, can be given as follows:

1. **BLAS-1 operations**
   These involve, e.g., operations like `axpy()` and `dot()`. Being part of the BLAS, efficient library implementations of those methods are widely available. They do not require further regard in this work, as there is no further optimization potential.

   Only the BLAS-1 operations containing reductions (i.e., `dot()` and `nrm2()`) require distributed memory communication. However, for those operations the distributed memory communication follows a very simple pattern as it merely requires a reduction of all partial results.

2. **Block vector-enabled BLAS-1 operations**
   The use of block vectors often requires block vector-enabled BLAS-1 operations, to be called Block-BLAS-1 operations. The idea is to apply a BLAS-1 operation to a block of vectors at once. Simple operations like `axpy()` and `scal()` can be executed on the full block at once, regardless of the block's storage order. However, this does not hold for operations including reductions, like `dot()` and `nrm2()`. In this case, n_b calls to standard BLAS-1 functions are sufficient if the vectors are stored one after another. If the vectors are stored in an interleaved way, this is not possible and one either has to execute n_b successive BLAS-1 operations passing a stride of n_b between successive vector elements to them, or provide a manual implementation. The first approach using a stride of n_b is very likely to lead to performance issues due to an unfavorable data access pattern. Hence, a manual implementation is required for high efficiency.

   In terms of communication patterns, similar points as for the BLAS-1 operations above can be made. Solely the block vector versions of `dot()` and `nrm2()` require communication of all partial results, with the only difference being that there are now n_b partial results.

3. **(Shifted) SpMV**
   The standard SpMV is the key operation of many (single vector) sparse linear algebra algorithms. Hence, implementations are available in many libraries, e.g. the vendor-supplied Intel MKL and NVIDIA cuSPARSE. The performance of this operation strongly depends on the structure of the sparse matrix, and it is not safe to assume high efficiency from available software libraries for all matrices (see chapter 5). Adding a shift causes only a small change to the SpMV and all its peculiarities considering implementation and performance are retained. However, implementations of the shifted SpMV are scarce in available software libraries which often necessitates a manual implementation if chaining the SpMV with an `axpy()` operation should be avoided.

   The communication patterns of this operation in distributed memory depend on the structure of the matrix. Usually, the involved vectors are distributed in the same way as the matrix. Hence, some elements of the input vector need to be gathered from remote locations before the SpMV operation can be carried out. Section 9.4 will describe this procedure in more detail.

4. **(Shifted) SpMMV**
   The SpMMV is the block vector version of the SpMV and it usually appears if a sparse linear algebra algorithm is formulated as a block algorithm. Doing so can be motivated purely by potential performance gains (as in KPM and ChebFD) or also by numerical benefits (as in BJDQR). The performance gains are quantified in section 4.2.3 and the basic properties of this operation are analyzed in chapter 6.

   Just as for its single-vector counterpart, the communication patterns depend on the matrix structure. The only difference is that n_b vector elements need to be transferred for each matrix element corresponding to a remote location. This leads to larger message sizes for increasing n_b, which is important for mitigating the impact of latency and makes SpMMV favorable over n_b SpMV calls with respect to communication behavior.

5. **Tall & skinny GEMM operations**
   This includes the operations TSMTTSM, TSMM, and TSMM-inplace as required by the BJDQR solver. Beyond this, block vector orthogonalization, e.g., using SVQB, requires similar operations. As shown in chapter 7, these operations have interesting performance properties and a

careful manual implementation potentially bears significant potential for speedup compared to available implementations.

Similar to the Block-BLAS-1 operations explained above, the only operation requiring communication is the one which comes with a reduction: TSMTTSM. The result of this operation is a small matrix which needs to be reduced over all involved processes.

4 Performance Modeling of Sparse Linear Algebra Building Blocks

Performance modeling is a central component of performance engineering. The main target of performance modeling is to find expressions for performance behavior of a given code or algorithm on a given computer architecture. There is a wide variety of approaches towards performance modeling. Performance models based on stochastic analysis or statistics are in the best case able to give accurate predictions of program performance [164, 165]. However, they rarely provide insight into how the performance can be improved. Opposed to that, the analysis of bounds and bottlenecks "provides valuable insight into the primary factors affecting the performance of computer systems. In particular, the critical influence of the system bottleneck is highlighted and quantified" [114, p. 70]. By following a "white box" approach and revealing relevant bottlenecks of a kernel on a given hardware platform, one can assess the quality of a specific implementation and derive potential optimization strategies. The Roofline performance model is arguably the most prominent performance model based on the "bounds and bottlenecks" idea. It is the main tool for performance modeling used in this work and will be introduced briefly in the following section. Afterwards, in section 4.2 it will be specifically applied to the present building blocks (see section 3.5) and a selection of the hardware platforms introduced in section 2.1.

4.1 The Roofline Performance Model

The Roofline performance model as popularized by Williams et al. is entitled as an "insightful visual performance model for multi-core architectures" [179]. It relates processor performance to memory traffic, assuming a single-level memory hierarchy. Usually, this level is the main memory and the Roofline model only considers traffic between main memory and the cores, ignoring all caches. The central assumption of the Roofline model is that in-core execution overlaps perfectly with data transfers in the memory

hierarchy. Latency effects of memory accesses are assumed to be not present. As this work focuses on *large-scale* sparse linear algebra, the assumption that the data set exceeds the cache and resides in main memory is applicable, which qualifies Roofline as a suitable performance model.

The Roofline model delivers an upper bound ("light speed estimate") for the attainable performance, i.e., the Roofline performance limit P^* of a kernel, based on its arithmetic intensity I, applicable maximum floating point performance P_{max}, and maximum attainable main memory bandwidth b_s as

$$P^* = \min\left(I \times b_{\text{s}}; P_{\text{max}}\right) . \tag{4.1}$$

In eq. (4.1) one can already recognize two major classes of kernels: "memory bound" kernels with $P^* = I \times b_{\text{s}}$ and "compute bound" kernels with $P^* = P_{\text{max}}$. It will be shown that many kernels and algorithms of sparse linear algebra are memory bound, i.e., limited by main memory bandwidth. The same conclusion can be drawn by comparing I and the inverse machine balance I_{M} (which is defined as $P_{\text{peak}}/b_{\text{s}}$): If I is smaller than I_{M}, the kernel is memory bound and the performance is limited by b_{s}. Otherwise, it is limited by P_{peak} (whose relation to P_{max} is described below). Due to b_{s} being fixed for a given architecture, the only way to increase the performance of a bandwidth-limited kernel on a given hardware platform is to increase the kernel's arithmetic intensity I. In this work, the term "Roofline efficiency" is used for the achieved double precision floating point performance P of a kernel relative to its Roofline limit P^*.

More elaborate performance models like the Execution Cache Memory (ECM) model [168] require detailed knowledge about the kernel's data traffic and location of data accessed in the kernel, which can usually not be supplied for the irregular data access patterns in sparse linear algebra. This, and the fact that the Roofline model delivers sufficient insight in the scope of this work, justifies the choice of Roofline as a suitable tool for performance modeling. The input parameters to the Roofline model are briefly explained in the following.

Arithmetic intensity I. Williams et al. define the term "operational intensity to mean operations per byte of DRAM traffic" [179]. As all relevant operations of sparse linear algebra involve floating point arithmetic, the term "arithmetic intensity" can be used over "operational intensity," and be defined as the ratio between executed Floating Point Operations (FLOPs) per byte of Dynamic Random Access Memory (DRAM) traffic.

A straightforward way to determine the arithmetic intensity of a kernel is by manually counting the FLOPs and memory transfers. This requires knowledge about data locality (i.e., whether some data may be located in a cache and should not be considered) and the size of the used data types. In some cases, a kernel's arithmetic intensity cannot be obtained by only investigating the code. For example, due to the irregular memory access pattern of to the input vector of an SpMV, it is only possible to specify, e.g., an upper bound for the arithmetic intensity (I_{max}) assuming perfect re-use of input vector data. See section 5.4 for a detailed analysis of this topic.

It is always possible to get precise values for I from measurements, e.g., using Hardware Performance Monitoring (HPM). Using tools like LIKWID [169] on CPUs or nvprof [134] on NVIDIA GPUs, one can get precise measurements of the executed FLOPs and memory transfers which gives a realistic image of a kernel's arithmetic intensity.

Attainable saturated peak memory bandwidth b_s. Obviously, this is the hardware metric which determines the maximum performance of bandwidth-limited kernels. A first approximation for the maximum attainable main memory bandwidth is the theoretical memory bandwidth as given in a processor's data sheet, which yields an absolute upper bound. However, in reality this value is rarely achieved even for very simple kernels, and the realistically attainable peak memory bandwidth is significantly lower. For instance, the theoretical peak memory bandwidth of K20 is 250 GB/s, which deviates significantly from the actually attainable maximum memory bandwidth b_s of 182 GB/s (which is measured as described below).

A simple way to obtain meaningful values for b_s is to run a set of microbenchmarks like STREAM [125] on the system. STREAM consists of four separate benchmarks which are summarized in table 4.1. The STREAM benchmarks show different properties in terms of arithmetic intensity and the ratio between load and store operations. Note that for all STREAM benchmarks, the target array `a[]` is not read from main memory and thus, an additional "write allocate" may occur on cache-based systems [80]. Consequently, to minimize data transfers, *streaming store* instructions, which avoid a redundant transfer of `a[]` from main memory to the core, should be used if the architecture implements them. As this is the case for all considered Intel architectures, streaming stores have been enabled for the STREAM benchmarks at compile time for them, delivering the "B/Iter" ratios as given in table 4.1. For measuring the attainable bandwidth on the GPU architectures, a GPU port of STREAM called GPU-STREAM [52] is used. It

	Name	Kernel	B/ Iter	FLOP/ Iter	Load/ Store
STREAM	COPY	`a[i] = b[i]`	16	0	1
	SCALE	`a[i] = q*b[i]`	16	1	1
	ADD	`a[i] = b[i] + c[i]`	24	1	2
	TRIAD	`a[i] = b[i] + q*c[i]`	24	2	2
	DOT	`q += b[i] * c[i]`	16	2	∞
	MM$\{k, s\}$	cf. listing 4.1	$8(2+k)$	$2k$	$k+1$

Table 4.1: Summary of memory microbenchmarks with `a[]`, `b[]`, and `c[]` large arrays of length n and `q` a scalar value. All data are of real double precision type and the STREAM benchmarks feature a loop over `i` which has been omitted for brevity. "B/Iter" ("FLOP/Iter") denotes the minimum amount of bytes to be transferred (floating point operations to be executed) per loop iteration, and "Load/Store" is the ratio between load and store operations, assuming minimal data traffic.

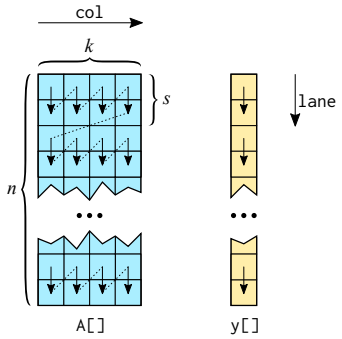


Figure 4.1: Data storage of the MM$\{k, s\}$ microbenchmark with $k = 4$ and $s = 2$.

implements the aforementioned STREAM benchmarks and augments the set with a read-only benchmark "DOT," which is similar to the computation of a scalar product.

A closer investigation reveals that on some architectures, none of the STREAM benchmarks achieves the maximum practically achievable bandwidth. Especially on CPU architectures, read-dominated or read-only memory access may yield highest bandwidth. As many sparse linear algebra algorithms are read-dominated (due to the fact that the sparse matrix rarely gets written to), this is of special importance for the present work. This can be demonstrated with the help of a custom microbenchmark which is called MM$\{k, s\}$. The benchmark kernel is shown in listing 4.1 and the layout of data is visualized in fig. 4.1. Section 5.4 will reveal that MM$\{k, s\}$ mimics to some extent the memory access pattern of SpMV for a matrix with k

Listing 4.1 MM$\{k, s\}$ microbenchmark kernel with $k > 0$ and n a multiple of s.

```
1 double y[n], A[n*k], tmp[s];
2 // initialize data...
3
4 for (int row = 0; row < n; row += s) {
5   for (int lane = 0; lane < s; lane++)
6     tmp[lane] = 0.;
7
8   for (int col = 0; col < k; k++)
9     for (int lane = 0; lane < s; lane++)
10      tmp[lane] += A[row*k+col*s+lane] * y[row+lane];
11
12  for (int lane = 0; lane < s; lane++)
13    y[row+lane] = tmp[lane];
14 }
```

non-zero entries per row, which qualifies it as a useful microbenchmark for this work. However, the absence of the potentially irregular matrix access patterns of SpMV makes it a suitable tool for obtaining upper bandwidth bounds. If $k = 1$, the operation is similar to a diagonal matrix-vector multiplication. The parameter s corresponds to the SIMD width of the examined architecture. It ensures perfectly vectorized access in the innermost loops in listing 4.1 and is in some sense related to the C parameter of the SELL-C-σ storage format (see section 5.2). As the load/store ratio of MM$\{k, s\}$ increases with increasing k, it is a useful microbenchmark for kernels with different load/store ratios, ranging from balanced to very load-dominated.

Figure 4.2 shows the main memory bandwidth of HSW and KNL for the MM$\{k, s\}$ microbenchmark, varying k and s chosen according to the SIMD width of each architecture as given in table 2.1 on page 22. Thread parallelism is employed along the outer loop and the parameters k and s are known at compile time, allowing the compiler to create efficient code. It can be observed that the main memory bandwidth of MM$\{k, s\}$ increases for increasing load dominance (i.e., for increasing k) on HSW until it saturates at approximately 65 GB/s, which is at about 120% of the bandwidth at $k = 1$. Although not shown in the graph, similar behavior can also be observed on the further considered multi-core CPU architectures as well as KNC. However, probably due to its novel memory technology, KNL shows a somewhat inverse behavior. The highest bandwidth is achieved with $k = 1$, and from this point it declines with increasing k until a certain level, which is at only 75% of the highest bandwidth.

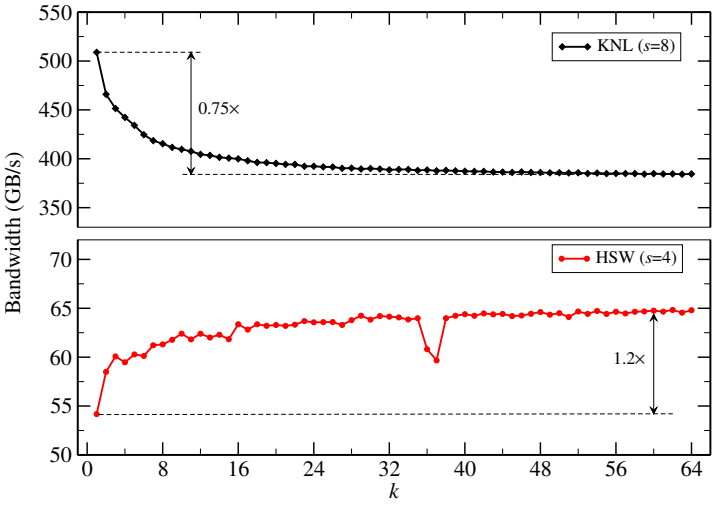


Figure 4.2: Memory bandwidth of the MM$\{k, s\}$ microbenchmark on HSW and KNL with $n = 30,000,000$. Note that the ordinate does not start from zero.

As the performance modeling approach strives to define an absolute upper performance bound, b_s is chosen to be the highest achieved bandwidth from any of the presented benchmarks. One may argue that this is not always a realistic upper bound for the read-dominated kernels of sparse linear algebra. However, a finer approach would entail having a separate performance model for each kernel and, in case of SpM(M)V, for each test case. Obviously, this does not seem expedient, and the approach of setting b_s to the highest achieved bandwidth is pursued while still keeping the sometimes surprising results from above in mind. The achieved bandwidth and respective benchmark for each architecture is shown in table 2.1.

Applicable maximum floating point performance $P_{\max}$. Speaking in Roofline terms, the second important class of kernels besides bandwidth-limited kernels are compute-limited kernels, i.e., kernels which are limited by $P_{\max}$. $P_{\max}$ is the maximum performance of the investigated kernel if all data is accessible in the level 1 cache. The simplest approximation of $P_{\max}$ is to set it equal to the processor's theoretical peak performance P_{peak}.

In order to get more precise values, it is necessary to investigate the resources used by the kernel and available by the processor in more detail. For example, many current processors (like all regarded architectures in this work) can only deliver full peak performance if the executed FLOPs are balanced between additions and multiplications. A kernel which executes only

either of them can only run at half of P_{peak} in the best case. HSW even relies on FMA instructions to deliver its maximum speed. Several other potential bottlenecks, like loads and stores to the level 1 cache, lack of vectorization, etc. exist which can yield a P_{max} significantly lower than P_{peak}.

Again, it should be noted that the scope of performance modeling in this work is to define upper performance limits. It will be shown in the next chapters that the kernels' performance is usually not limited by P_{max}. Hence, it is not essential to define precise values of P_{max} for each regarded kernel. Consequently, for ease of analysis, P_{max} is set to P_{peak} throughout this work.

4.2 Application of the Roofline Model to Sparse Linear Algebra

In the following, the Roofline performance model will be applied to the relevant building blocks as summarized in section 3.5. As mentioned above, the parameters b_{s} and P_{max} relate to the hardware rather than the kernel. Hence, in this section, expressions for the arithmetic intensity I of the involved building blocks are derived.

4.2.1 (Block-)BLAS-1 Roofline Model

Level 1 BLAS include scalar, vector and vector-vector operations. In the presented applications, the operations `nrm2()`, `dot()`, `axpy()`, and `scal()` are used. Applying these operations to block vectors does not change their computational intensity. Without going into further detail, it can be stated that the maximum arithmetic intensity over all of those building blocks is achieved by the `nrm2()` operation in real (complex) double precision, where $I = 0.25$ FLOP/B. This is much smaller than the I_{M} values of all regarded architectures as given in table 2.1 on page 22. Thus, all required BLAS-1 operations are strongly bandwidth-limited on all architectures considered in this work (and on many more beyond that).

4.2.2 Sparse Matrix-Vector Multiplication Roofline Model

The SpMV Roofline model developed herein is constructed along results previously published [103, 104]. Performance modeling is done for the general SpMV operation

$$\vec{y} \leftarrow \vec{y} + H\vec{x} \tag{4.2}$$

Listing 4.2 Minimal data structures for SpMV with "`data`" being either "`double`" or "`double complex`."

```
1 data x_val[m];
2 data y_val[n];
3 data H_val[nnz];
4 int H_col[nnz];
```

with H being a general $n \times m$ sparse matrix with n_{nz} non-zero entries and an average non-zero count of $n_{\text{nzr}} = n_{\text{nz}}/n$ per row and $n_{\text{nzc}} = n_{\text{nz}}/m$ per column.

Given that H is stored as a sparse matrix, it is a valid assumption that at the very least a 4-byte column index needs to be stored which each non-zero element. If n_{nzr} is reasonably large, the storage of matrix row information can be omitted. The description of sparse matrix storage formats in section 5.1 will substantiate this assumption. Listing 4.2 summarizes the minimum working set of data needed for an SpMV operation. Given that v_{el} denotes the bytes needed to store a single matrix/vector value (i.e., $v_{\text{el}} = 8$ (16) for real (complex) double precision values), the minimum data volume $V_{\text{SpMV,min}}$ to be transferred in this operation is defined by the size of the involved data structures, assuming that each needed element is only accessed once:

$$V_{\text{SpMV,min}} = \Big[\underbrace{(v_{\text{el}} + 4)\, n_{\text{nz}}}_{\substack{H \text{ values and} \\ \text{column indices}}} + \underbrace{2 v_{\text{el}} n}_{\vec{y}\text{ values}} + \underbrace{v_{\text{el}} m}_{\vec{x}\text{ values}}\Big]\ \text{B}\,. \tag{4.3}$$

Assuming a large data set, at least this data needs to be transferred between the cores and main memory. Equation (4.3) assumes that the input vector $\vec{x}$ gets read from memory only once. However, this may not be the case for sparse matrices with an irregular access pattern and in reality, (at least part of) $\vec{x}$ has to be read more often. To account for this overhead, the right hand side vector data traffic overhead factor α is introduced which quantifies the efficiency of the $\vec{x}$ access in terms of data traffic. The impact of the matrix structure on the transferred data volume has been noted long ago, and first similar SpMV Roofline models have been conducted by Schubert et al. using the factor κ [152] and Liu et al. using the factor $k(m)$ [118]. In this work, the overhead factor α as first used by Kreutzer et al. [103] is used.

The factor α can be implemented in the SpMV memory volume as

$$V_{\text{SpMV}} = \Big[\underbrace{(v_{\text{el}} + 4)\, n_{\text{nz}}}_{\substack{H \text{ values and} \\ \text{column indices}}} + \underbrace{2 v_{\text{el}} n}_{\vec{y}\text{ values}} + \underbrace{\alpha v_{\text{el}} n_{\text{nz}}}_{\vec{x}\text{ values}}\Big]\ \text{B}\,. \tag{4.4}$$

The size of α depends on the hardware (particularly the cache size) and the nonzero pattern of the matrix. Some special values it can attain are as follows:

- $\alpha = 0$
  For relatively small data sets and certain iterative algorithms, it is possible that $\vec{x}$ remains in cache between subsequent SpMV iterations. In this case, which is not relevant for this work, $\vec{x}$ has no impact on the memory traffic. Note that in this case also the $\vec{y}$ traffic may be nonexistent.

- $\alpha = 1/n_{\text{nzc}}$
  As $n_{\text{nzc}} = n_{\text{nz}}/m$ is the average number of non-zero entries per column, this value of α indicates that each $\vec{x}$ value gets loaded from main memory only once. This the best case for large data set scenarios as it yields the minimum data volume $V_{\text{SpMV,min}}$ as given in eq. (4.3).

- $\alpha = 1$
  Each load of $\vec{x}$ goes to main memory. This corresponds to the case where there is no cache available.

- $\alpha > 1$
  If the processor's cache is organized in cache lines which contain L elements, an unfavorable matrix structure which causes a cache miss for each $\vec{x}$ value can result in a value of α of up to L.

The factor α is usually not known from the analysis. Instead, it has to be determined by measuring the actual SpMV data volume and solving eq. (4.4) for α. Experiments on this are conducted in section 5.4.

In contrast to the data volume, the number of executed double precision floating point operations F_{SpMV} does not depend on the matrix structure, but only on the non-zero count of the matrix. Given that each non-zero matrix entry needs to be multiplied with a vector entry, and that the results of this multiplication need to be summed up in each matrix row, it can be specified as

$$F_{\text{SpMV}} = (f_{\text{MUL}} + f_{\text{ADD}})\, n_{\text{nz}}\ \text{FLOP}\,, \tag{4.5}$$

with f_{MUL} the number of double precision floating point operations for a multiplication (1 for real, 6 for complex numbers) and f_{ADD} the number of double precision floating point operations for an addition (1 for real, 2 for complex

numbers). The maximum arithmetic intensity of the SpMV operation can now be given as

$$I_{\text{SpMV,max}} = \frac{F_{\text{SpMV}}}{V_{\text{SpMV,min}}} \tag{4.6}$$

$$= \frac{(f_{\text{MUL}} + f_{\text{ADD}})\, n_{\text{nz}}}{(v_{\text{el}} + 4)\, n_{\text{nz}} + 2v_{\text{el}}n + v_{\text{el}}m} \frac{\text{FLOP}}{\text{B}} \tag{4.7}$$

$$= \frac{f_{\text{MUL}} + f_{\text{ADD}}}{v_{\text{el}} + 4 + 2v_{\text{el}}/n_{\text{nzr}} + v_{\text{el}}/n_{\text{nzc}}} \frac{\text{FLOP}}{\text{B}}. \tag{4.8}$$

According the Roofline model as given in eq. (4.1), an upper bound for the SpMV performance can be given as

$$P^*_{\text{SpMV}} = \min\left(I_{\text{SpMV,max}} \times b_{\text{s}}; P_{\text{max}}\right) \tag{4.9}$$

$$= \min\left(\frac{f_{\text{MUL}} + f_{\text{ADD}}}{v_{\text{el}} + 4 + 2v_{\text{el}}/n_{\text{nzr}} + v_{\text{el}}/n_{\text{nzc}}} \frac{\text{FLOP}}{\text{B}} \times b_{\text{s}}; P_{\text{max}}\right). \tag{4.10}$$

For all considered test matrices and hardware architectures, P^*_{SpMV} will never be limited by P_{max}. Hence,

$$P^*_{\text{SpMV}} = \frac{f_{\text{MUL}} + f_{\text{ADD}}}{v_{\text{el}} + 4 + 2v_{\text{el}}/n_{\text{nzr}} + v_{\text{el}}/n_{\text{nzc}}} \frac{\text{FLOP}}{\text{B}} \times b_{\text{s}} \tag{4.11}$$

$$= \begin{cases} \dfrac{2}{12 + 8\,(2/n_{\text{nzr}} + 1/n_{\text{nzc}})} \dfrac{\text{FLOP}}{\text{B}} \times b_{\text{s}} & \text{for real values, and} \\ \dfrac{8}{20 + 16\,(2/n_{\text{nzr}} + 1/n_{\text{nzc}})} \dfrac{\text{FLOP}}{\text{B}} \times b_{\text{s}} & \text{for complex values.} \end{cases} \tag{4.12}$$

Taking the correction factor α into account, a more precise expression for the arithmetic intensity of SpMV is

$$I_{\text{SpMV}} = \frac{F_{\text{SpMV}}}{V_{\text{SpMV}}} \tag{4.13}$$

$$= \frac{f_{\text{MUL}} + f_{\text{ADD}}}{v_{\text{el}} + 4 + \alpha v_{\text{el}} + 2v_{\text{el}}/n_{\text{nzr}}} \frac{\text{FLOP}}{\text{B}} \tag{4.14}$$

$$= \begin{cases} \dfrac{2}{12 + 8\,(\alpha + 2/n_{\text{nzr}})} \dfrac{\text{FLOP}}{\text{B}} & \text{for real values, and} \\ \dfrac{8}{20 + 16\,(\alpha + 2/n_{\text{nzr}})} \dfrac{\text{FLOP}}{\text{B}} & \text{for complex values.} \end{cases} \tag{4.15}$$

An important means to increase bandwidth-limited performance is to increase a kernel's arithmetic intensity. Obviously, in the SpMV case this can be done by reducing α, which in turn can be achieved, e.g., by reducing the matrix bandwidth as described in section 5.6.

4.2.3 Sparse Matrix-Multiple Vectors Multiplication Roofline Model

While the potential of performance gains by vector blocking has been noted long ago, first work on SpMMV performance models has been conducted by Gropp et al. who have estimated the speedup of using n_b vectors in an SpMMV operation over n_b successive SpMV operations [78]. Recently, this idea has attracted new interest, refining the original model and taking new bottlenecks like cache bandwidth into account [4, 118].

The Roofline model for the SpMMV kernels is very similar to the SpMV kernel. Considering the operation $\vec{Y} \leftarrow \vec{Y} + H\vec{X}$ with $\vec{X}, \vec{Y}$ blocks of n_b vectors and 4-byte indices, the minimum SpMMV data volume $V_{\text{SpMMV,min}}$ can be given analogously to $V_{\text{SpMV,min}}$ as

$$V_{\text{SpMMV,min}} = \Big[\underbrace{(v_{\text{el}}+4)\,n_{\text{nz}}}_{\substack{H \text{ values \&} \\ \text{column indices}}} + \underbrace{2v_{\text{el}}nn_b}_{\vec{Y}\text{ values}} + \underbrace{v_{\text{el}}mn_b}_{\vec{X}\text{ values}}\Big]\ \text{B}\,, \tag{4.16}$$

and the actual SpMMV data volume is

$$V_{\text{SpMMV}} = \Big[\underbrace{(v_{\text{el}}+4)\,n_{\text{nz}}}_{\substack{H\text{values \&} \\ \text{column indices}}} + \underbrace{2v_{\text{el}}nn_b}_{\vec{Y}\text{values}} + \underbrace{\alpha v_{\text{el}}n_{\text{nz}}n_b}_{\vec{X}\text{values}}\Big]\ \text{B}\,. \tag{4.17}$$

The overhead factor α now also depends on n_b and can attain the same possible values as in the SpMV case. Note that only the data traffic contribution from the vectors gets multiplied with n_b, as the matrix has to be read only once for n_b vectors. Given that

$$F_{\text{SpMMV}} = n_b F_{\text{SpMV}}\,, \tag{4.18}$$

the maximum arithmetic intensity of the SpMMV operation can be specified as

$$I_{\text{SpMMV,max}} = \frac{F_{\text{SpMMV}}}{V_{\text{SpMMV,min}}} \tag{4.19}$$

$$= \frac{n_b\,(f_{\text{MUL}} + f_{\text{ADD}})}{v_{\text{el}} + 4 + 2n_b v_{\text{el}}/n_{\text{nzr}} + v_{\text{el}}n_b/n_{\text{nzc}}}\,\frac{\text{FLOP}}{\text{B}}\,. \tag{4.20}$$

According to the Roofline model as given in eq. (4.1), an upper bound for the SpMV performance can be given as

$$P^*_{\text{SpMMV}} = \min\left(I_{\text{SpMMV,max}} \times b_s; P_{\text{max}}\right) \tag{4.21}$$

$$= \min\left(\frac{n_b(f_{\text{MUL}} + f_{\text{ADD}})}{v_{\text{el}} + 4 + 2n_b v_{\text{el}}/n_{\text{nzr}} + v_{\text{el}} n_b/n_{\text{nzc}}} \frac{\text{FLOP}}{\text{B}} \times b_s; P_{\text{max}}\right). \tag{4.22}$$

The corrected arithmetic intensity, taking the overhead factor α into account, is

$$I_{\text{SpMMV}} = \frac{F_{\text{SpMMV}}}{V_{\text{SpMMV}}} \tag{4.23}$$

$$= \frac{n_b\left(f_{\text{MUL}} + f_{\text{ADD}}\right)}{v_{\text{el}} + 4 + v_{\text{el}} n_b\left(\alpha + 2/n_{\text{nzr}}\right)} \frac{\text{FLOP}}{\text{B}} \tag{4.24}$$

$$= \begin{cases} \dfrac{2n_b}{12 + 8n_b\left(\alpha + 2/n_{\text{nzr}}\right)} \dfrac{\text{FLOP}}{\text{B}} & \text{for real values, and} \\ \dfrac{8n_b}{20 + 16n_b\left(\alpha + 2/n_{\text{nzr}}\right)} \dfrac{\text{FLOP}}{\text{B}} & \text{for complex values.} \end{cases} \tag{4.25}$$

Obviously, all derived equations for SpMMV are valid for SpMV in case $n_b = 1$.

Speedup from vector blocking. The arithmetic intensity of SpMMV with $n_b > 1$ is larger than I_{SpMV}. For square matrices (i.e., $n_{\text{nzr}} = n_{\text{nzc}}$) and under the assumptions that the Roofline performance prediction P^*_{SpMV} as defined in eq. (4.9) is limited by the "$I \times b_s$" term for any value of n_b, and that α always attains its optimal value of $1/n_{\text{nzr}}$, the maximum potential speedup from using a single SpMMV operation instead of n_b SpMV operations (i.e., the speedup from vector blocking), can be estimated as

$$S^* = \frac{I_{\text{SpMMV,max}}}{I_{\text{SpMV,max}}} \tag{4.26}$$

$$= \frac{n_{\text{nzr}}(v_{\text{el}} + 4) + 3v_{\text{el}}}{n_{\text{nzr}}(v_{\text{el}} + 4)/n_b + 3v_{\text{el}}}. \tag{4.27}$$

If n_b is very large, the asymptotic maximum speedup is

$$S^*_{n_b \gg 1} = \frac{n_{\text{nzr}}}{3}\left(1 + \frac{4}{v_{\text{el}}}\right) + 1 \tag{4.28}$$

$$= \begin{cases} \dfrac{n_{\text{nzr}}}{2} + 1 & \text{for real values, and} \\ \dfrac{5n_{\text{nzr}}}{12} + 1 & \text{for complex values.} \end{cases} \tag{4.29}$$

However, in reality this speedup will rarely be achieved because increasing n_b usually involves an increase of α due to a higher memory and cache footprint of the vectors. Chapter 6 provides an analysis of this issue.

4.2.4 General Dense Matrix-Matrix Multiplication Roofline Model

GEMM operations, being a common building block for dense linear algebra, occur as a result of vector blocking in the considered sparse linear algebra algorithms. The GEMM operation can be expressed as

$$C \leftarrow \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C \tag{4.30}$$

with scalar $\alpha \neq 0$ and β, and the dense matrices A, B, and C having the following dimensions:

$$C \in K^{\bar{m} \times \bar{n}}\,, \tag{4.31}$$

$$\text{op}(A) \in K^{\bar{m} \times \bar{k}}\,, \text{ and} \tag{4.32}$$

$$\text{op}(B) \in K^{\bar{k} \times \bar{n}}\,. \tag{4.33}$$

The term $\text{op}(A)$ indicates one of A, A^T, A^H (similar for B). For the quantification of the amount of FLOPs to be done in GEMM, one has to distinguish between several cases, depending on whether $\alpha = 1$ and $\beta = 0$. The number of FLOPs per GEMM for each case can be given as

$$F_{\text{GEMM},\alpha=1,\beta=0} = \left[\left(f_{\text{ADD}} + f_{\text{MUL}}\right)\bar{m}\bar{n}\bar{k}\right] \text{ FLOP}\,, \tag{4.34}$$

$$F_{\text{GEMM},\alpha\neq1,\beta=0} = \left[\left(f_{\text{ADD}} + f_{\text{MUL}}\right)\bar{m}\bar{n}\bar{k} + \left(f_{\text{MUL}}\right)\bar{m}\bar{n}\right] \text{ FLOP}\,, \tag{4.35}$$

$$F_{\text{GEMM},\alpha=1,\beta\neq0} = \left[\left(f_{\text{ADD}} + f_{\text{MUL}}\right)\bar{m}\bar{n}\bar{k} + \left(f_{\text{ADD}} + f_{\text{MUL}}\right)\bar{m}\bar{n}\right] \text{ FLOP}\,, \tag{4.36}$$

$$F_{\text{GEMM},\alpha\neq1,\beta\neq0} = \left[\left(f_{\text{ADD}} + f_{\text{MUL}}\right)\bar{m}\bar{n}\bar{k} + \left(f_{\text{ADD}} + 2f_{\text{MUL}}\right)\bar{m}\bar{n}\right] \text{ FLOP}\,, \tag{4.37}$$

Data	$I^{\alpha=1,\beta=0}_{\text{GEMM,max}}$	$I^{\alpha\neq1,\beta=0}_{\text{GEMM,max}}$	$I^{\alpha=1,\beta\neq0}_{\text{GEMM,max}}$	$I^{\alpha\neq1,\beta\neq0}_{\text{GEMM,max}}$
real	$\frac{1}{4\Theta}$	$\frac{1+1/(2k)}{4\Theta}$	$\frac{1+1/k}{4\Theta'}$	$\frac{1+3/(2k)}{4\Theta'}$
complex	$\frac{1}{2\Theta}$	$\frac{1+3/(4k)}{2\Theta}$	$\frac{1+1/k}{2\Theta'}$	$\frac{1+7/(4k)}{2\Theta'}$

Table 4.2: Maximum arithmetic intensity in FLOP/B for the GEMM operation for real and complex data as well as different α, β combinations with $\Theta = 1/\bar{m} + 1/\bar{n} + 1/\bar{k}$ and $\Theta' = \Theta + 1/\bar{k}$ if C is not aliased with A ($\Theta' = \Theta$ otherwise).

and, assuming streaming stores to C if $\beta = 0$, the minimum amount of bytes to be transferred for each case is

$$V_{\text{GEMM,min},\beta=0} = v_{\text{el}} \left(\bar{n}\bar{m} + \bar{n}\bar{k} + \bar{m}\bar{k}\right) \text{ B}, \text{ and} \tag{4.38}$$

$$V_{\text{GEMM,min},\beta\neq0} = v_{\text{el}} \left(2\bar{n}\bar{m} + \bar{n}\bar{k} + \bar{m}\bar{k}\right) \text{ B}. \tag{4.39}$$

Table 4.2 summarizes the maximum arithmetic intensity of GEMM, i.e., $F_{\text{GEMM}}/V_{\text{GEMM,min}}$, for different data types and α,β values. In the following, the application to the specific T&S-GEMM operations as presented in section 3.5 is done. Further analysis of the operations can be found in chapter 7.

TSMTTSM. This corresponds to an inner product of block vectors and it holds that $\bar{m}, \bar{n} \ll \bar{k}$. In a sparse solver, the small dimensions $\bar{m}$ and $\bar{n}$ correspond to block vector widths, and $\bar{k}$ to the number of matrix/vector rows n. The arithmetic intensity as given in table 4.2 can be approximated as

$$I_{\text{TSMTTSM,max}} \approx \begin{cases} \dfrac{1}{4(1/\bar{m} + 1/\bar{n})} \dfrac{\text{FLOP}}{\text{B}} & \text{for real values, and} \\ \dfrac{1}{2(1/\bar{m} + 1/\bar{n})} \dfrac{\text{FLOP}}{\text{B}} & \text{for complex values.} \end{cases} \tag{4.40}$$

Due to the result matrix being very small, the values of α and β do not have a significant influence on the arithmetic intensity.

TSMM. For the multiplication of a tall & skinny with a small matrix, it holds that $\bar{n}, \bar{k} \ll \bar{m}$. The long dimension $\bar{m}$ corresponds to the number of vector/matrix rows n and the short dimensions $\bar{n}$ and $\bar{k}$ are related to block vector widths. The arithmetic intensity can be read from table 4.2 with $\Theta \approx 1/\bar{n} + 1/\bar{k}$. Further simplifications to the arithmetic intensity (similar to TSMTTSM) cannot be applied here.

TSMM-inplace. Again, it holds that $\bar{n}, \bar{k} \ll \bar{m}$ and additionally $\bar{n} \leq \bar{k}$. This is not a GEMM operation as C is aliased with A, which is not allowed according to the BLAS standard [8]. Thus, a manual implementation is required if copying of C should be avoided. Regarding the transferred data volume, eq. (4.38) applies, regardless of the actual value of β. In the $\beta \neq 0$ case, the same minimum data volume has to be transferred due to the aliasing. The arithmetic intensity can be read from table 4.2. Out of all three operations, TSMM-inplace with $\alpha \neq 1$, $\beta \neq 0$, $\bar{n} = \bar{k}$, and complex values has the largest arithmetic intensity for given $\bar{m}$, $\bar{n}$, and $\bar{k}$.

4.3 Summary

In this chapter, the Roofline model was identified as a suitable tool for performance modeling in this work and respective expressions for the arithmetic intensity of the previously identified relevant building blocks were derived. For all kernels besides (Block-)BLAS-1 operations, the arithmetic intensity depends on the respective test case's properties like, i.e., the matrix/vector dimensions m and n, the average number of non-zeros per sparse matrix row (column) n_{nzr} (n_{nzc}), and the number of vectors in a block n_b.

Figure 4.3 visualizes the Roofline performance model for selected architectures. Some of the described building blocks are plotted in the graph as vertical lines at their specific arithmetic intensity. A kernel is bandwidth-limited on an architecture if it intersects with the architecture's characteristic roofline curve in its increasing region. If the intersection coincides with the architecture's knee point, the kernel's arithmetic intensity is equal to the inverse machine balance, i.e., assuming that b_{s} and P_{max} are the only present bottlenecks, this kernel can make perfect use of both hardware capabilities. Otherwise, i.e., "right" of the knee point, the kernel is compute-limited. Due to their architectural differences, a single kernel can be bandwidth-limited on some and compute-limited on other architectures. This is for example the case for the TSMTTSM example displayed in fig. 4.3, which is (barely) bandwidth-limited on HSW and compute-limited on all other architectures.

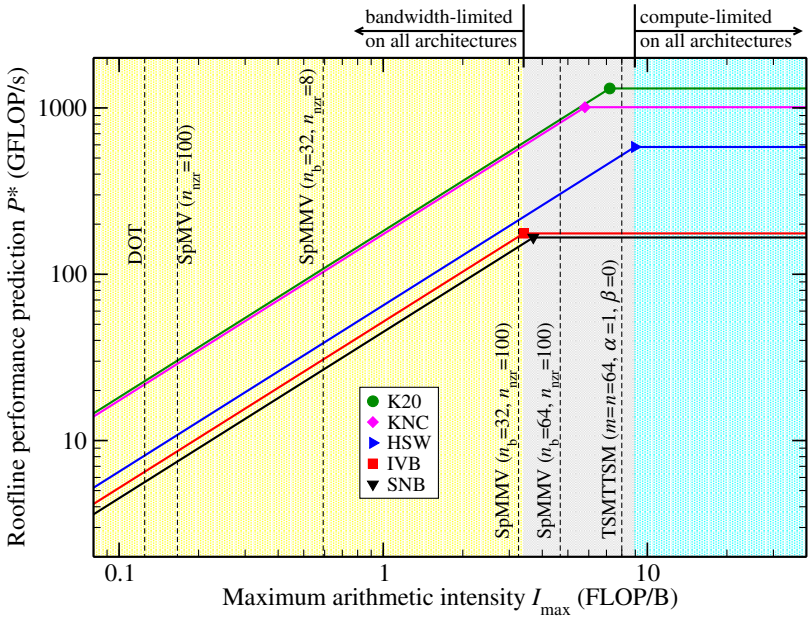


Figure 4.3: Roofline performance model for selected hardware architectures and building blocks in real double precision arithmetic. Note the double logarithmic scale.

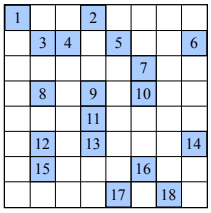
5 A Unified Sparse Matrix Storage Format for High-Performance Sparse Matrix-Vector Multiplication

The SpMV is a key operation not only in the presented eigenvalue solvers, but in a wide variety of sparse linear algebra algorithms. Specifically, many sparse linear algebra solvers (be it for eigenvalues or systems of equations) are composed of iterative methods where SpMV is frequently one of the most time-consuming building blocks at the lowest level. The high relevance of this operation drives ongoing and intense research on efficient implementations using all different kinds of hardware architectures, with the central topic often being the storage format of the sparse matrix. In this work, the focus is on *general* sparse matrices, i.e., no assumptions on special matrix structures like constant (sub-)diagonals, dense blocks or symmetry are taken.

This chapter first presents a brief overview of such formats for different hardware architectures in section 5.1. It becomes clear that the optimal format often depends on the hardware platform under consideration. In the era of heterogeneous computing, such a dependence calls for hardware-agnostic data structures. An effort towards a hardware-agnostic sparse matrix storage format, SELL-C-σ, is introduced in section 5.2 and has previously been published by Kreutzer et al. [104]. The discussion of storage formats is followed by a thorough explanation of the SELL-C-σ SpMV kernel in section 5.3 and a deep, model-guided performance analysis in section 5.4. This chapter closes with a performance comparison of the unified SELL-C-σ format with device-specific formats on various architectures in section 5.5, which demonstrates the wide applicability and high efficiency of SELL-C-σ.

5.1 General Sparse Matrix Storage Formats

In this section, selected established and widely used sparse matrix storage formats are briefly introduced and important features, which lead to the development of SELL-C-σ, are highlighted.




```
   val = [  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 ]

   col = [  0 |  3 |  1 |  2 |  4 |  7 |  5 |  1 |  3 |  5 |  3 |  1 |  3 |  7 |  1 |  5 |  4 |  6 ]

rowptr = [  0 |  2 |  6 |  7 | 10 | 11 | 14 | 16 | 18 ]
```

Figure 5.1: The CRS data structures for an example matrix.

Listing 5.1 CRS SpMV kernel.

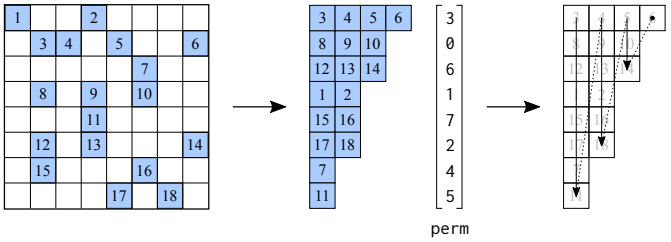
```
1 for (int i = 0; i < n; i++) {
2   double tmp = y[i];
3
4   for (int j = rowptr[i]; j < rowptr[i+1]; j++)
5     tmp += val[j] * x[col[j]];
6
7   y[i] = tmp;
8 }
```

COO and CRS. The most basic sparse matrix storage format is the Coordinate (COO) format, which stores each non-zero matrix entry together with its row and column index. A derivate of it, and arguably the most widely known and used sparse matrix storage format, is the Compressed Row Storage (CRS) format, which compresses the storage of the non-zero's row indices (hence the name). CRS (for reference, see the book by Barrett et al. [21]) is the widely accepted standard sparse matrix storage format for cache-based (multi-core) CPUs. SpMV implementations based on CRS are commonly available in numerical kernel libraries like Intel MKL and NVIDIA cuSPARSE. As illustrated in fig. 5.1, CRS stores the matrix' non-zero entries (`val[]`) and column indices (`col[]`) row-wise, and in addition, for each row the accumulated number of non-zero entries of all previous rows (`rowptr[]`). It is suitable for general sparse matrices as it does not pose any requirements to the matrix structure. A further important property is that it only stores n_{nz} matrix values, i.e., there is no overhead introduced by storing zeros, e.g., required by padding.

Listing 5.2 CRS SpMV kernel with four-way unrolling.

```
1 for (int i = 0; i < n; i++) {
2   double tmp0 = 0., tmp1 = 0., tmp2 = 0., tmp3 = 0.;
3
4   for (int j = rowptr[i]; j < rowptr[i+1]; j+=4) {
5     tmp0 += val[j+0] * x[col[j+0]];
6     tmp1 += val[j+1] * x[col[j+1]];
7     tmp2 += val[j+2] * x[col[j+2]];
8     tmp3 += val[j+3] * x[col[j+3]];
9   }
10
11  for (j = j-4; j < rowptr[i+1]; j++) // remainder loop
12    tmp0 += val[j] * x[col[j]];
13
14  y[i] = y[i] + tmp0+tmp1+tmp2+tmp3;
15 }
```

The CRS SpMV kernel is shown in listing 5.1. Note that the access to the input vector `x[]` is indirect due to the sparse nature of the matrix. This is always the case, regardless of the sparse matrix storage format. On a multi-core system, thread parallelism can be achieved along the outer loop. The inner loop execution can be vectorized to further increase the core execution efficiency. For this to work, the inner loop must be unrolled (manually or by the compiler) according to the SIMD width. Listing 5.2 shows the SIMD-friendly CRS SpMV kernel with four-way unrolling of the inner loop (which corresponds to, e.g., real double precision data with AVX(2) vectorization as present in the HSW, IVB, and SNB architectures). Note that current compilers can often do this re-formulation by themselves. Loop peeling to fulfill alignment constraints is omitted for brevity. In this version of the CRS SpMV kernel, the access to `val[]` and `col[]` in the inner loop (lines 4 to 9) is vectorized. Note that the indirect load of `x[]` data generally remains scattered, as it is always the case for general SpMV regardless of the matrix storage format. A potential problem of this kernel is that the grade of vectorization depends on the number of non-zero elements in the matrix rows. If the matrix contains only few non-zero elements in each row, vectorized execution of the CRS SpMV kernel may be inefficient. This is due to a limited number of iterations of the vectorized loop and an increasing dominance of the scalar remainder loop (lines 11 to 12). Beyond that, this kernel introduces a constant overhead due to the in-register reduction ("horizontal add") in line 14 which is needed for summing up the partial results. Note that all discussed overhead costs grow with the SIMD width. For instance, the KNC architecture features 512 bit

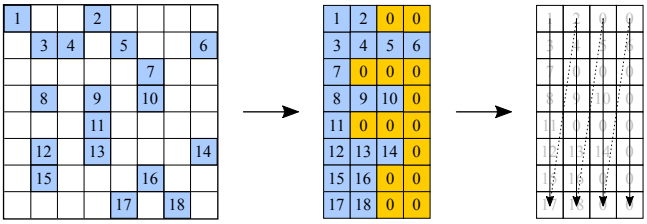



```
  val = [  3 |  8 | 12 |  1 | 15 | 17 |  7 | 11 |  4 |  9 | 13 |  2 | 16 | 18 |  5 | 10 | 14 |  6 ]
  col = [  0 |  0 |  0 |  3 |  0 |  7 |  2 |  1 |  6 |  1 |  1 |  1 |  2 |  4 |  7 |  2 |  5 |  5 ]
jdptr = [  0 |  8 | 14 | 17 ]
```

Figure 5.2: The JDS data structures for an example matrix.

wide SIMD registers, which allows for 8-way SIMD processing of real double precision values (see table 2.1 on page 22). Obviously, a value of $n_{\text{nzr}} \lesssim 8$ (which is realistic according to table 2.2 on page 33) leads to inefficient vectorization in this case. On top of that, the 8-way horizontal add and this architecture's strict alignment requirements add additional overhead, which disqualifies CRS as an efficient sparse matrix storage format for SpMV on this architecture. Similar observations have been made by Saule et al. [150].

CRS on GPU architectures. For GPU architectures, there is a wide variety of available sparse matrix storage formats which are summarized and classified in an extensive review by Filippone et al. [68]. Bell and Garland have investigated the suitability of CRS for SpMV on GPUs and find that this storage format is not appropriate for this kind of hardware [24]. A parallelization approach similar to the vectorized kernel as described above comes with the same drawbacks as discussed there. In fact, the potential overhead will be even larger on a GPU because a warp (which is processing in a SIMT manner) contains 32 threads (compared to 8-way SIMD on the KNC for real double precision data). Another approach is to assign one thread to each matrix row. However, in this case, non-coalesced access to the matrix data leads to inefficient execution [24]. Efforts to alleviate this effect using shared memory have already been made in the early years of GPU computing [71]. In recent years, several attempts have been made to adapt the CRS storage format for highly efficient SpMV on many-core architectures like KNC and K20, e.g., CSR-Adaptive [49, 77], ACSR [14], and CSR5 [117].



```
val = [  1 |  3 |  7 |  8 | 11 | 12 | 15 | 17 |  2 |  4 |  0 |  9 |  0 | 13 | 16 | 18 | ...
         0 |  5 |  0 | 10 |  0 | 14 |  0 |  0 |  0 |  6 |  0 |  0 |  0 |  0 |  0 |  0 ]

col = [  0 |  1 |  5 |  1 |  3 |  1 |  1 |  4 |  3 |  2 |  0 |  3 |  0 |  3 |  5 |  6 | ...
         0 |  4 |  0 |  5 |  0 |  7 |  0 |  0 |  0 |  7 |  0 |  0 |  0 |  0 |  0 |  0 ]

maxrowlen = 4
```

Figure 5.3: The ELLPACK data structures for an example matrix.

JDS and pJDS. Due to the high relevance of vectors computers in supercomputing in the 1980s era, dedicated sparse matrix storage formats for this kind of architectures were developed at that time. The most important ones are the Jagged Diagonal Storage (JDS) format [149] and the ITPACK/ELLPACK (in the following, only the latter term will be used) format [98]. In both, the sparse matrix is stored along "jagged diagonals." JDS, as illustrated in fig. 5.2, requires a global re-ordering of matrix rows according to their non-zero count to avoid any storage overhead, which is expressed as a permutation stored in the vector `perm[]`. The starting index of each jagged diagonal into the matrix data is stored in `jdptr[]`. The padded Jagged Diagonal Storage (pJDS) format as introduced by Kreutzer et al. [103] is an attempt to exploit the similarities between vector computers and GPUs as recognized, among others, by Volkov and Demmel [172]. It adopts the idea of global sorting from JDS, but introduces row padding to obtain aligned and coalesced memory access for vectorized loads. Global sorting of the matrix rows can potentially harm the non-zero pattern of matrices, leading to an increase of the α parameter in the arithmetic intensity of SpMV as defined in eq. (4.13). Thus, global sorting is usually not desirable and neither JDS nor pJDS will further be regarded in this work. Experiments on the influence of global sorting on the performance will be shown in section 5.4.

ELLPACK and derivatives. The ELLPACK storage format is visualized in fig. 5.3. While being similar to JDS as it stores values and column indices along jagged diagonals, ELLPACK does not re-order the rows which introduces a

Listing 5.3 ELLPACK SpMV kernel with four-way unrolling and n a multiple of four.

```
1 for (int i = 0; i < n/4; i++) {
2   double tmp0 = y[i*4+0];
3   double tmp1 = y[i*4+1];
4   double tmp2 = y[i*4+2];
5   double tmp3 = y[i*4+3];
6
7   for (int j = 0; j < maxrowlen; j++) {
8     tmp0 += val[j*n+i*4+0] * x[col[j*n+i*4+0]];
9     tmp1 += val[j*n+i*4+1] * x[col[j*n+i*4+1]];
10    tmp2 += val[j*n+i*4+2] * x[col[j*n+i*4+2]];
11    tmp3 += val[j*n+i*4+3] * x[col[j*n+i*4+3]];
12  }
13
14  y[i*4+0] = tmp0;
15  y[i*4+1] = tmp1;
16  y[i*4+2] = tmp2;
17  y[i*4+3] = tmp3;
18 }
```

storage overhead depending on the matrix row with the largest number of non-zero elements. Similar to JDS/pJDS, the similarities between vector computers and GPUs led to a rediscovery of the ELLPACK format with the advent of GPUs for general purpose computing [24]. More recently, the advent of many-core CPU with wide SIMD units led to an adaption of ELLPACK-like storage formats for SpMV on those architectures [119].

As vectorization can be done along (jagged) diagonals, ELLPACK does not carry the performance pitfalls of the CRS format for architectures with wide SIMD units. Listing 5.3 shows the ELLPACK SpMV kernel for four-way SIMD-processing. Again, thread parallelism can be achieved along the outer loop. Note the absence of both an inner loop remainder and the reduction with horizontal add. Thus, this storage format is significantly more SIMD-friendly compared to CRS. However, as it can be seen in fig. 5.3, it involves a potentially large overhead in storage and data traffic, depending on the row with the highest number of non-zero entries. As SpMV is usually a bandwidth-limited operation, this overhead will directly be translated to a performance degradation in most cases. To alleviate this overhead, Monakov et al. came up with the idea of "Sliced ELLPACK," in which the matrix is divided row-wise into slices with $S < n$ rows each [128]. Then, each slice is stored in ELLPACK format. Obviously, in Sliced ELLPACK the storage and traffic overhead depends only on the "longest" row in each slice, rather than the globally

longest row. The overhead can further be decreased by moving similarly long matrix rows close to each other, an approach which already MONAKOV et al. have proposed [128]. Despite the potential overhead, (Sliced) ELLPACK is a sparse matrix storage format which is suitable for many matrices and all relevant hardware architectures.

In this work, the idea of Sliced ELLPACK is further pursued and refined, leading to the SELL-C-σ storage format as a "catch all" storage format for all relevant architectures.

5.2 The SELL-C-σ Sparse Matrix Storage Format

SELL-C-σ is a generalization of the Sliced ELLPACK storage format with the chunk height C equal to the slice size S. Row sorting according to non-zero counts is done motivated by JDS, but not on the global matrix but rather in scopes of σ rows each. Figure 5.4 shows the SELL-C-σ matrix for a given source matrix and different C and σ values. For simplicity in the following analysis, it will be assumed that the row count n is a multiple of C. If this is not the case, appropriate row padding or remainder loops have to be introduced. Besides the matrix values and column indices, which are stored along jagged diagonals within a chunk, SELL-C-σ stores the so-called "chunk pointer" (`chunkptr[]`), i.e., the starting offset of each chunk into the full matrix (which corresponds to the row pointer of the CRS format). For some kernels it may be useful to store additional metadata, such as the non-zero count of each row (cf. section 5.3).

SELL-C-σ corner cases and selection of C. Basically, SELL-C-σ evolves from a combination of existing ideas, and some of the previously discussed established formats are included in SELL-C-σ. Notable corner cases of SELL-C-σ are SELL-1-1, which is equal to CRS, and SELL-n-1, which is equal to ELLPACK. As discussed above, CRS (SELL-1-1) incurs no data overhead at all and ELLPACK (SELL-n-1) may incur significant overhead. On the other hand, ELLPACK was shown to be significantly more SIMD-friendly than CRS for SpMV. Thus, C should be chosen as small as possible to reduce the overhead, but large enough to enable efficient SIMD vectorization and meet possible alignment restrictions in SpMV. The main idea is to chose C equal to (a small multiple of) the architecture's SIMD width. Consequently, if a single SELL-C-σ matrix should be used on different architectures, a sensible value of C would be equal to the largest SIMD/SIMT width over all architectures.

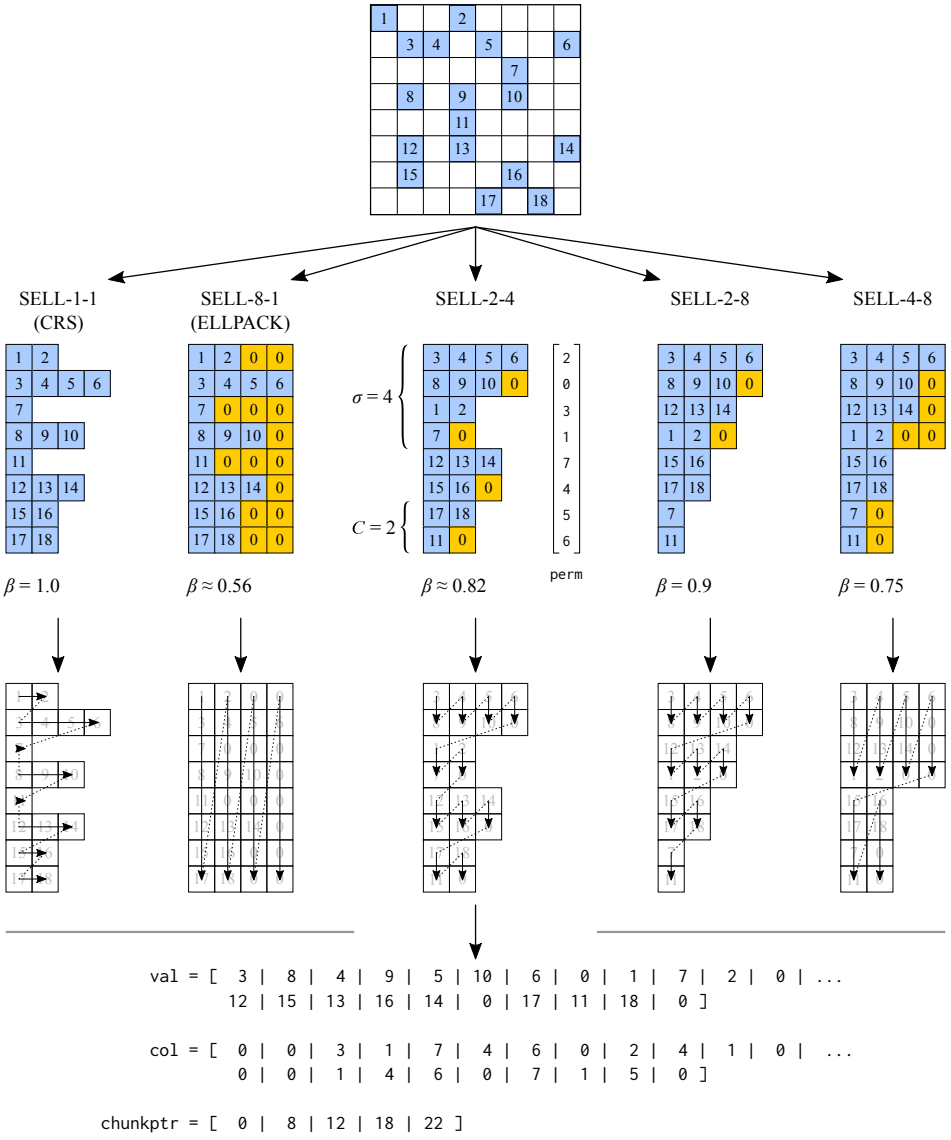


Figure 5.4: Variants of the SELL-C-σ sparse matrix storage format for an example matrix and minimum data needed to be stored. The contents of the data structure are shown for SELL-2-4.

The influence of C on the SpMV performance on different architectures will be analyzed in section 5.4.

Quantification of storage overhead. In order to quantify the overhead incurred by the zero-padding in the SELL-C-σ format, the chunk occupancy β is defined as the ratio between useful (i.e., non-zero) matrix entries n_{nz} and elements stored in the SELL-C-σ format as

$$\beta = \frac{n_{\text{nz}}}{\texttt{chunkptr}[n_C]} \tag{5.1}$$

with $n_C = n/C$ the number of chunks of the matrix. Obviously, in the best case $\beta = 1$ and only non-zero elements are stored. In a worst-case scenario, a single row of each chunk has m non-zero entries (i.e., it is fully occupied) while all other rows have only one non-zero element. In this case, $n_{\text{nz}} = n_C(m + C - 1)$ and the SELL-C-σ matrix is stored as if it was dense, i.e., the number of stored entries is nm. Hence,

$$\beta_{\text{worst}} = \frac{n_C(m + C - 1)}{nm} = \frac{n(m + C - 1)}{Cnm} \xrightarrow{n,m \gg C} \frac{1}{C}\,. \tag{5.2}$$

Refined arithmetic intensity. Depending on the implementation and further factors like C and the cache line size (detailed analysis to be found in section 5.3), a non-optimal value of $\beta < 1$ may directly translate to an increase of data traffic for the matrix. In this case, the matrix-induced data volume has to be multiplied with the reciprocal of β which yields a maximum data volume to be transferred in an SpMMV operation of

$$V_{\text{SpMMV,SELL-}C\text{-}\sigma\text{,max}} = \left[(v_{\text{el}} + 4)\, n_{\text{nz}}/\beta + n_b(v_{\text{el}}\alpha n_{\text{nz}} + 2v_{\text{el}}n)\right]\ \text{B}\,, \tag{5.3}$$

and the minimal arithmetic intensity adds up to

$$I_{\text{SpMMV,SELL-}C\text{-}\sigma\text{,min}} = \frac{n_b\,(f_{\text{MUL}} + f_{\text{ADD}})}{(v_{\text{el}} + 4)/\beta + n_b\,(v_{\text{el}}\alpha + 2v_{\text{el}}/n_{\text{nzr}})}\,\frac{\text{FLOP}}{\text{B}}\,. \tag{5.4}$$

In case of SpMV, the expressions hold for $n_b = 1$. Note that the memory traffic from the vectors does not increase, as all padding elements have column index zero and access the same cache line of $\vec{x}$. In addition, the number of FLOPs should not be increased as no useful operations are added.

Row sorting and selection of σ. A small value of β can be increased by sorting rows across chunks, such that equally "long" (in terms of non-zero count) rows get close to each other. This is done by means of a permutation vector, which can be seen in fig. 5.4 for SELL-2-4. Obviously, it holds that the lowest overhead for a given C can be achieved if rows are sorted globally, i.e., SELL-C-n is used. However, global re-ordering is costly for large matrices and may substantially change the access pattern to $\vec{x}$, which could cause that possible temporal or spatial locality arising from the physical problem gets harmed. As a consequence of this, the α parameter of the SpMV Roofline model (eq. (4.13)) may increase which would lower the arithmetic intensity and the kernel's performance. Experimental results on this are shown in section 5.4. This problem can be ameliorated by not sorting the rows globally, but only in scopes of σ consecutive rows. If C is a multiple of σ, no change of chunk occupancy due to sorting can be expected. However, depending on the implementation (cf. section 5.3), it can lead to a decrease of memory traffic due to untouched cache lines. Typically, σ is chosen to be a multiple of C. The effect of increasing σ can also be seen in fig. 5.4 (comparing SELL-2-4 to SELL-2-8). An optimal choice of σ would be large enough to increase β to an appropriate value close to one, but small enough not to worsen the access pattern of $\vec{x}$ too much. However, the optimal σ is usually not known a priori. The β values for all test matrices, $C = 16$, and the two different σ values 1 and 256 can be found in table 2.2 on page 33.

The sorting of matrix rows is done as a pre-processing step. In iterative solvers, the sorting has to be done only once, which makes the sorting overhead insignificant if sufficiently many iterations are done. Furthermore, the usually small sorting scope and the existence of several independent sorting scopes make the sorting procedure efficient and easily parallelizable. Note that a row permutation of the matrix usually necessitates a column permutation as well. This is due to the fact that the in- and output vectors of SpMV operations often get interchanged in between successive solver iterations. Hence, the entire solver should run in the permuted index space to avoid excessive vector permutations. Another reason for permuting the matrix columns in addition to the rows is to avoid possible escalation of the matrix bandwidth (i.e., the maximum distance of non-zero elements from the matrix' diagonal). Due to column permutation, a diagonal matrix element will still be on the diagonal in the permuted matrix. As many sparse matrices from relevant applications have the majority of their non-zero entries centered around the diagonal, applying a column permutation helps to preserve this structure for $\sigma > 1$. Figure 5.21 on page 101 shows a concrete ex-

Listing 5.4 SELL-4-σ SpMV kernel with four-way unrolling.

```
for (int i = 0; i < n/4; i++) {
  double tmp0 = y[i*4+0];
  double tmp1 = y[i*4+1];
  double tmp2 = y[i*4+2];
  double tmp3 = y[i*4+3];

  for (int j = 0; j < chunklen[i]; j++) {
    tmp0 += val[chunkptr[i]+j*4+0] * x[col[chunkptr[i]+j*4+0]];
    tmp1 += val[chunkptr[i]+j*4+1] * x[col[chunkptr[i]+j*4+1]];
    tmp2 += val[chunkptr[i]+j*4+2] * x[col[chunkptr[i]+j*4+2]];
    tmp3 += val[chunkptr[i]+j*4+3] * x[col[chunkptr[i]+j*4+3]];
  }

  y[i*4+0] = tmp0;
  y[i*4+1] = tmp1;
  y[i*4+2] = tmp2;
  y[i*4+3] = tmp3;
}
```

ample for a row- and column-permutation of the Spin-N_{Up} test case. Note that the SELL-2-4 data structures shown in the lower part of fig. 5.4 also have permuted columns.

5.3 SELL-C-σ SpMV Implementation

In the following listings and analyses, some metadata about the SELL-C-σ matrix beyond what was shown in fig. 5.4 will be used. For convenience and brevity in the kernel code, the length of each chunk (i.e., the non-zero count of its longest row) will be stored in an additional array `chunklen[]` where `chunklen[i] = (chunkptr[i+1]-chunkptr[i])/C`. Similar to this, the largest non-zero count of each group of four rows will be stored in an additional array `rowlen4[]`, which just points to `chunklen[]` if $C = 4$ and the non-zero count of each single row will be stored in an array `rowlen[]`. Note that for an actual implementation of SELL-C-σ SpMV, those auxiliary arrays are not necessary and the data structures as shown in fig. 5.4 are sufficient.

SELL-C-σ SpMV on CPU architectures. In the simplest case, C is equal to the SIMD length and there is exactly one SIMD iteration in the inner loop. Listing 5.4 shows the corresponding SELL-C-σ SpMV kernel with $C = 4$. This would be a minimal choice of C for, e.g., real double precision data on

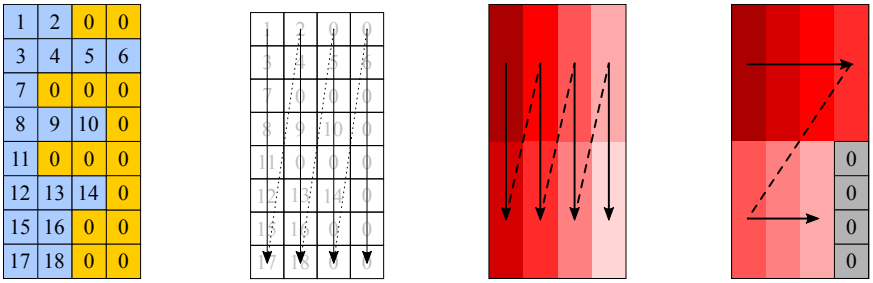
Listing 5.5 SELL-4-σ SpMV kernel implemented with AVX intrinsics.

```
1 for (int i = 0; i < n/4; i++) {
2   __m256d vtmp = _mm256_load_pd(&y[i*4]);
3
4   for (int j = 0; j < chunklen[i]; j++) {
5     __m256d vval = _mm256_load_pd(&val[chunkptr[i]+j*4]);
6     __m256d vx   = _mm256_set_pd(x[col[chunkptr[i]+j*4+3]],
7                                  x[col[chunkptr[i]+j*4+2]],
8                                  x[col[chunkptr[i]+j*4+1]],
9                                  x[col[chunkptr[i]+j*4+0]]);
10            vtmp = _mm256_add_pd(vtmp,_mm256_mul_pd(vval,vx));
11  }
12
13  _mm256_store_pd(&y[i*4],vtmp);
14 }
```

an AVX(2) machine. Note that the SELL-C-σ SpMV kernel is oblivious of σ. The access to `val[]` and `col[]` is vectorizable. As four rows are processed in a vectorized manner, this kernel's vectorization efficiency is much less sensitive to short rows as it is the case for CRS. In addition, reading and writing `y[]` is also vectorized. Thus, streaming stores to `y[]` can be employed if $\vec{y} \leftarrow H\vec{x}$ should be computed. The access to `x[]` is scattered, as it is always case for all general SpMV implementations, regardless of the sparse matrix storage format.

An implementation of listing 5.4 using AVX compiler intrinsics for explicit vectorization is shown in listing 5.5. The only part of this kernel subject to scalar execution is the scattered loading of $\vec{x}$ elements, which is done element-wise. This can be avoided on architectures which implement the *gather* instruction, which is part of, e.g., the AVX2 and Many Integrated Core (MIC) instruction sets. Listing 5.6 shows an AVX2 version of listings 5.4 and 5.5. The use of the gather instruction by the compiler intrinsic `_mm256_i32gather_pd()` becomes obvious. Moreover, the addition and multiplication in line 10 of listing 5.5 can be replaced by the FMA instruction in line 8 of listing 5.6.

SELL-C-σ SpMV scheduling options on CPU architectures. If C is larger than the SIMD width, there are two possibilities of implementing the SELL-C-σ SpMV kernel. First, the outer loop can be unrolled by a factor of C, and each inner loop iteration covers the full height of the chunk C. This scheduling, which will be called "C-first" scheduling, is visualized in



(a) SELL-8-1 matrix as of fig. 5.4 **(b)** Storage of the matrix **(c)** C-first scheduling **(d)** `rowlen4[]`-first scheduling

Figure 5.5: SELL-C-σ SpMV scheduling options for $C = 8$ and a SIMD width of 4 words. A block of one color depicts a SIMD access and the color intensity decreases with inner iteration count.

Listing 5.7 SELL-8-σ SpMV kernel with C-first scheduling (see fig. 5.5c).

```
for (i = 0; i < n/8; i++) {

  double t0 = y[i*8+0];
  double t1 = y[i*8+1];
  ...
  double t7 = y[i*8+7];

  for (j = 0;
       j < chunklen[i];
       j++) {
    int o = chunkptr[i] +
            j*8;
    t0 += val[o+0]*x[col[o+0]];
    t1 += val[o+1]*x[col[o+1]];
    ...
    t7 += val[o+7]*x[col[o+7]];
  }

  y[i*8+0] = t0;
  y[i*8+1] = t1;
  ...
  y[i*8+7] = t7;

}
```

Listing 5.8 SELL-8-σ SpMV kernel with `rowlen4[]`-first scheduling (see fig. 5.5d).

```
for (i = 0; i < n/8; i++) {
  for (b = 0; b < 8/4; b++) {
    double t0 = y[i*8+4*b+0];
    double t1 = y[i*8+4*b+1];
    double t2 = y[i*8+4*b+2];
    double t3 = y[i*8+4*b+3];

    for (int j = 0;
             j < rowlen4[i*2+b];
             j++) {
      int o = chunkptr[i] +
              j*8 + 4*b;
      t0 += val[o+0]*x[col[o+0]];
      t1 += val[o+1]*x[col[o+1]];
      t2 += val[o+2]*x[col[o+2]];
      t3 += val[o+3]*x[col[o+3]];
    }

    y[i*8+4*b+0] = t0;
    y[i*8+4*b+1] = t1;
    y[i*8+4*b+2] = t2;
    y[i*8+4*b+3] = t3;
  }
}
```

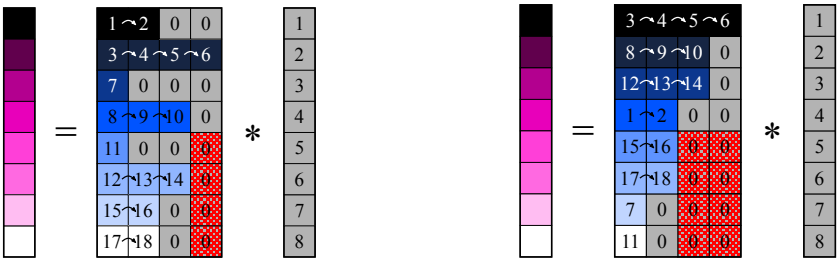
Listing 5.6 SELL-4-σ SpMV kernel implemented with AVX2 intrinsics.

```
1 for (int i = 0; i < n/4; i++) {
2   __m256d vtmp = _mm256_load_pd(&y[i*4]);
3
4   for (int j = 0; j < chunklen[i]; j++) {
5     __m256d vval = _mm256_load_pd(&val[chunkptr[i]+j*4]);
6     __m128i vcol = _mm_load_si128(&col[chunkptr[i]+j*4]);
7     __m256d vx   = _mm256_i32gather_pd(x,vcol,8);
8              vtmp = _mm256_fmadd_pd(vval,vx,vtmp);
9   }
10
11  _mm256_store_pd(&y[i*4],vtmp);
12 }
```

fig. 5.5c for $C = 8$ and a SIMD width of 4. The resulting kernel can be seen in listing 5.7. As the inner loop runs to `chunklen[i]`, all SELL-C-σ matrix elements are loaded and a reduced chunk occupancy will have a direct negative impact on the SpMV performance due to increased data traffic.

One approach to potentially minimize the data transfer overhead is to use the previously described `rowlen4[]` instead of `chunklen[]` for the inner loop, leading to "`rowlen4[]`-first" scheduling. As it can be seen in fig. 5.5d, one can possibly avoid loading some zero entries. However, this is only possible if an entire cache line (i.e., 8 successive matrix elements in real double precision on the regarded CPU architectures) contains only zero entries. The requirement on successive zero entries may be doubled if "Adjacent Cache Line Prefetching" is enabled on the system (which is usually the case on current Intel CPUs). This mechanism automatically prefetches a cache line from main memory if the previous cache line was loaded.

An SpMV implementation using `rowlen4[]`-first scheduling can justify a choice of $\sigma \leq C$ as it brings zero-only cache lines close to each other. However, it has turned out in a number of experiments for some test matrices that C-first scheduling yields equal or better performance than `rowlen4[]`-first scheduling for $C \leq 32$ and $\sigma \leq C$. In section 5.4 it will be motivated that $C = 32$ is the maximum sensible choice for the present hardware architectures. Hence, C-first scheduling will always be used in the following (even if `rowlen4[]`-first scheduling may still deliver better performance in some corner cases). The explicit unrolling of SELL-C-σ SpMV kernels with C-first scheduling as done in listing 5.7 can be accomplished with automatic code generation for arbitrary combinations of C and the SIMD width (cf. section 9.5).



(a) SELL-8-1 **(b)** SELL-8-8

Figure 5.6: Thread mapping for the CUDA SELL-8-σ SpMV operation for a single warp (assumed to contain 8 threads). Each thread has a different color and a cache line consists of 4 elements. Zero elements which are not loaded from main memory are indicated by the checkered pattern.

Listing 5.9 SELL-8-σ SpMV kernel implemented in CUDA with one thread per row and no excess threads/rows.

```
1 int row = threadIdx.x + blockIdx.x*blockDim.x; // 1 thread/ row
2 int chk = row/8; // chunk
3 int ric = row&7; // row in chunk, equiv. to 'row%8' but faster
4 double tmp = y[row];
5
6 for (int j=0; j<rowlen[row]; j++) {
7   tmp += val[chunkptr[chk] + j*8 + ric] *
8         x[col[chunkptr[chk] + j*8 + ric]];
9 }
10
11 y[row] = tmp;
```

SELL-C-σ SpMV on GPU architectures. Figure 5.6 depicts the thread mapping of the SELL-8-σ SpMV kernel on a GPU architecture. One thread is scheduled per row and no excess threads or rows are assumed. The corresponding kernel code is shown in listing 5.9. On GPU architectures, SIMD processing happens on a thread base ("SIMT") with the SIMD width being equal to the warp size. On all current NVIDIA GPU architectures, a warp consists of 32 threads. However, for the sake of easier illustration a warp size of eight is assumed in fig. 5.6. The fact that each matrix row is covered by a separate thread allows for fine-grained inner loops which only run to the according length of each row instead of the length of the entire chunk. This is to some extent similar to `rowlen4[]`-first scheduling as discussed above for the CPU implementation. Note that the "early-stopping" threads of a warp stall until the warp's last thread is finished and are not available for further

computation. However, if all threads accessing a cache line have stopped early, a load of this cache line (which contains only padding elements) can be avoided. This decreases the data traffic overhead of the SELL-C-σ matrix storage format and usually results in higher performance. Similar to the `rowlen4[]`-first CPU SELL-C-σ SpMV implementation, such fine-grained inner loops can justify a choice of $\sigma \leq C$. As a cache line on the GPU architectures contains only 4 real double precision elements (compared to 8 for the other architectures), and sensible choices of C on GPUs are 16 or 32 (see section 5.4), a benefit from row sorting inside a chunk is much more likely to be achieved on this architecture. Figure 5.6b demonstrates this effect: Due to row sorting in a scope of $\sigma = C = 8$, two cache line loads can be avoided instead of only one if $\sigma = 1$ (fig. 5.6a). Experimental results on this effect are shown in section 5.4.

5.4 SELL-C-σ SpMV Tuning Factors

The performance of the SpMV operation on different architectures will be discussed in the following. For each of the measurements, 100 SpMV operations $\vec{y} \leftarrow \vec{y} + H\vec{x}$ are performed and the average time t of the last 90 iterations (the first 10 iterations are considered as "warm up" phase) is measured. Using the number of useful FLOPs from eq. (4.5), the reported performance is computed as

$$P_{\text{SpMV}} = \frac{F_{\text{SpMV}}}{t} \tag{5.5}$$

$$= \begin{cases} \dfrac{2n_{\text{nz}}\,\text{FLOP}}{t} & \text{for real values, and} \\ \dfrac{8n_{\text{nz}}\,\text{FLOP}}{t} & \text{for complex values.} \end{cases} \tag{5.6}$$

As the name suggests, the SELL-C-σ storage format contains the two tuning factors C and σ. An additional tuning factor for OpenMP-parallel SpMV execution with any storage format is the OpenMP scheduling. In all implementations, the outermost loop over the number of matrix rows n is subject to OpenMP parallelization. In the following, the influence of all those tuning factors on the SpMV performance on various architectures is analyzed. The selection of corner case test matrices (cf. section 2.4) allows for a study of those parameters with a limited number of test matrices, from which the obtained knowledge can be transferred to further test cases. The final performance is resulting from a subtle interplay of all tuning factors. First, each

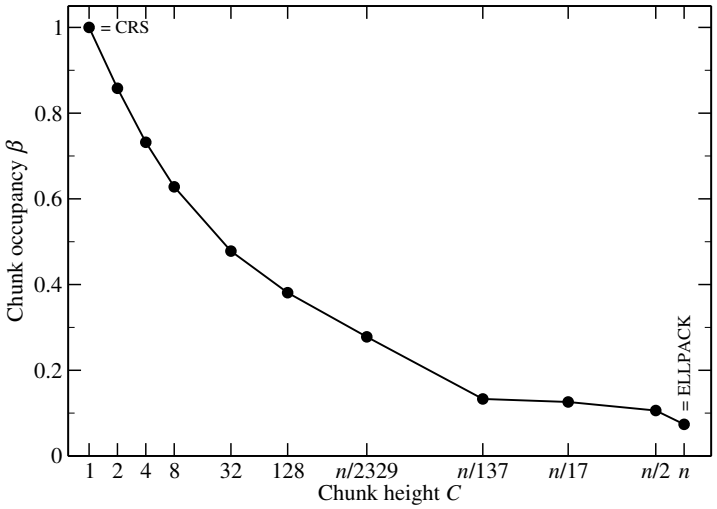
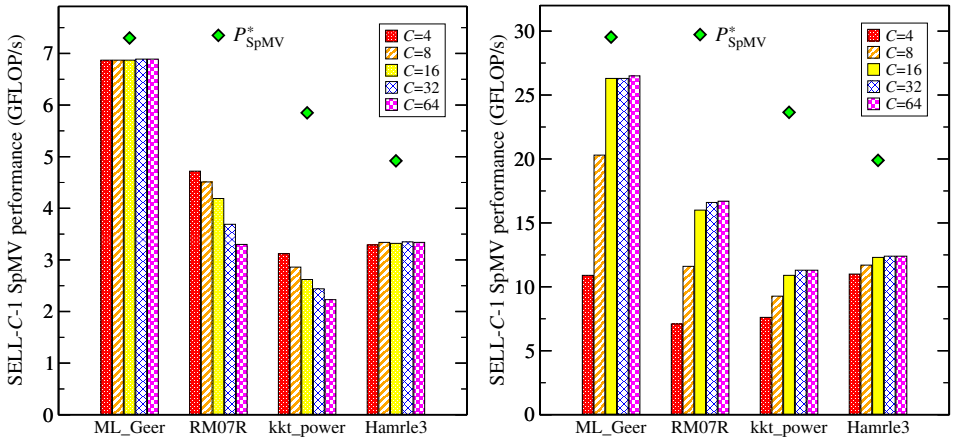


Figure 5.7: Influence of C on the chunk occupancy β for the kkt_power matrix with $\sigma = 1$. Note the logarithmic scale on the abscissa.

of them is considered individually and in the end, the combined tuned performance is shown.

5.4.1 Chunk Height C

As described in section 5.2, C should be a multiple of the SIMD width to enable efficiently vectorized execution. However, the chunk occupancy β of a SELL-C-σ matrix can never increase with increasing C. As the arithmetic intensity of the SELL-C-σ SpMV as given in eq. (5.4), and with it the maximum attainable performance in the bandwidth-limited case, decreases with β, it should be kept as large as possible. In the best case, each row of the matrix is equal in non-zero count and $\beta = 1$ for all values of $1 \leq C, \sigma \leq n$. According to table 2.2 on page 33, one test matrix where this is clearly not the case is kkt_power. Figure 5.7 visualizes the chunk occupancy β as a function of C for this matrix. The entire C range from 1 (corresponding to CSR) to n (corresponding to ELLPACK) is shown. Obviously, β attains its optimal value of 1 for $C = 1$, which is always true for all matrices. It also holds for all matrices that β can never increase for increasing C. In the following, a sensible choice of C will be analyzed for different architectures. The sorting scope σ will be kept constant to 1 in the following experiments and analyzed separately at a later point. Similarly, the OpenMP scheduling is fixed to the compiler default value, i.e., `STATIC`.



(a) SNB **(b)** K20

Figure 5.8: Influence of C on the SpMV performance on SNB and K20 for the corner case test matrices. P^*_{SpMV} is determined as specified in eq. (4.9). Note the different value ranges on the ordinates.

CPU architectures. Figure 5.8a shows the SpMV performance of the corner case test matrices on SNB subject to different C values. For this architecture, the minimal C which enables efficient vectorization using AVX instructions (similar to listing 5.5) is four. This motivates the set of C values chosen in fig. 5.8. For RM07R and kkt_power the performance decreases with increasing C. This is due to a decreasing chunk occupancy and holds for all matrices with $\beta < 1$ in table 2.2. A sensible choice of σ can counteract this behavior and will be discussed at a later point. On the contrary, matrices with $\beta = 1$ (ML_Geer, Hamrle3) show a constant performance for increasing C. Although those findings have been obtained on the SNB architecture, they directly apply to other CPU architectures as well.

KNC and KNL. On KNC and KNL, each memory data to be loaded in a vectorized manner must be aligned on a 64-byte boundary. For the matrix entries, setting $C = 8$ on this architecture meets the SIMD width as well as the alignment requirements and appears like a reasonable choice. However, the alignment requirements turn out to be critical for the access to the matrix' column indices (which is similar to line 6 of listing 5.6). Each column index is only 4 bytes in size. Hence, the minimal C value on KNC is not 8, but rather 16 for the present data types. Beyond that, similar quantitative results as for SNB can be observed also on KNC and KNL.

GPU architectures. Figure 5.8b shows the SpMV performance on K20 for the same matrices and C values as used in fig. 5.8a. Despite a warp size of 32, the relevant SIMD width for memory accesses of matrix values is 16. The reason for this is stated in section 2.1: global memory accesses are split into 128-byte requests if threads access more than 4 bytes each, which can lead to coalesced loads with less than a full warp. Obviously, this is the case for double precision matrix data, where each thread accesses 8 bytes of matrix data at a time. This explains that a close-to-optimal (with respect to $C = 32$) performance can be achieved already with $C = 16$ for all test cases in fig. 5.8b. Note that the access to the 4-byte column indices is not perfectly coalesced. However, the performance data indicate that the negative effect of this is very limited. It can further be seen that the performance increases for all matrices if C gets increased from 4 to 16, i.e., when the SIMT efficiency is increased to its optimum. This effect is most distinct for large-n_{nzr} matrices RM07R and ML_Geer. This is due to the fact that the matrix traffic plays a larger role than the vector traffic (which is usually non-coalesced for $\vec{x}$ anyway due to indirect addressing) in those two cases. A further observation is that the performance for RM07R and kkt_power does not decrease for $C > 16$ although the chunk occupancy β gets smaller. This is due to the aforementioned effect of fine-grained inner loops which avoids the loading of zero-only cache lines (see section 5.3).

Best practices for the selection of C. The above analysis motivates a choice of C "as small as possible" to maximize β if no knowledge about a matrix' non-zero variance is present. This reasons to choose C equal to the relevant SIMD/SIMT width of the investigated hardware architecture. If heterogeneous hardware architectures are present, C should meet the largest SIMD/SIMT width of them. As shown above, a chunk size of 16 already yields satisfying results on GPU architectures (which have the largest SIMD/SIMT width of all investigated architectures). Hence, regarding the whole range of considered architectures, a sensible choice to construct a unified storage format is $C = 16$.

5.4.2 Sorting Scope σ.

The next tuning factor to be analyzed concerns the sorting scope σ. Again, in this analysis the OpenMP scheduling is fixed to the compiler default value, i.e., `STATIC`, if not noted otherwise.

Figure 5.9 shows the chunk occupancy β as a function of C for the kkt_power matrix and different σ values. The data for "$\sigma = 1$ or 2" is the

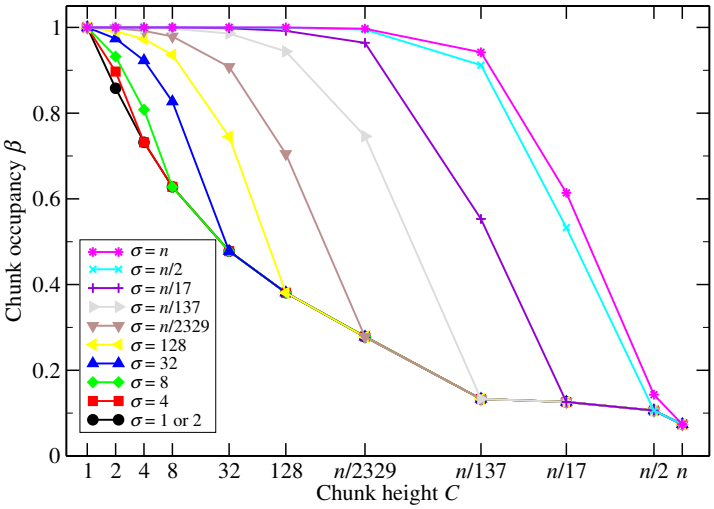
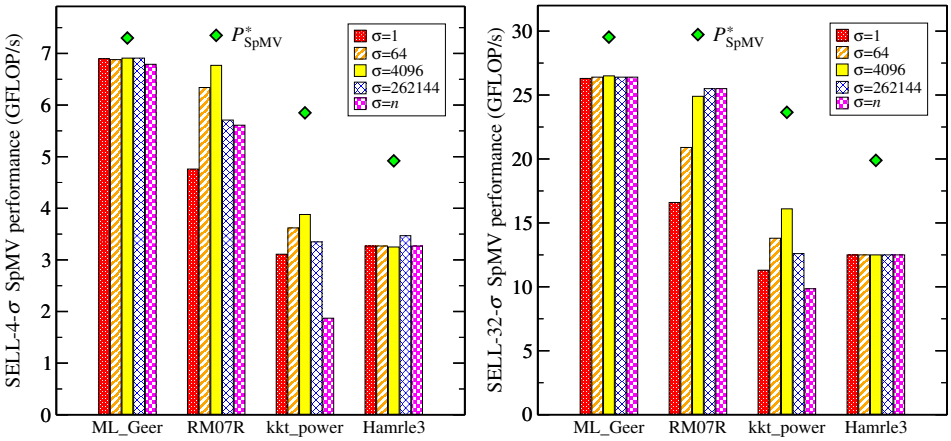


Figure 5.9: Influence of C on the chunk occupancy β for the kkt_power matrix with different σ values. The "$\sigma = 1$ or 2" curve corresponds to fig. 5.7. Note the logarithmic scale on the abscissa.

same as shown in fig. 5.7. Note that a sorting scope of two can never increase the chunk occupancy over a sorting scope of one. Furthermore, β can never decrease if σ is increased. However, sorting in a scope of σ can only increase the chunk occupancy if $\sigma > C$. Clearly, $\sigma = n$ maximizes β throughout the entire C range. However, a value of σ too large may entail a performance hazard as it potentially increases the overhead factor α in the arithmetic intensity of the SpMV operation as given in eq. (4.13).

Performance impact of σ on CPU and GPU architectures. The impact of σ on the SpMV performance of the four corner case matrices for SNB and K20 is shown in fig. 5.10. The "$\sigma = 1$" bar corresponds to the bar with the respective C value of fig. 5.8. Row sorting does not have a significant influence on the performance for the high-β matrices ML_Geer and Hamrle3 on either architecture. In contrast to this, the SpMV performance for the low-β matrices RM07R and kkt_power can be increased by setting $\sigma > 1$. A too large value of σ involves the danger of severe performance degradation, which is the case for kkt_power on both architectures and RM07R on SNB. The reason for that will be analyzed below for the kkt_power matrix. However, as it is the case for RM07R on K20, a large σ value does not *necessarily* lead to lower performance. Still, it is a good practice to keep σ moderate, as a performance degradation for large values cannot be ruled out.



(a) SNB **(b)** K20

Figure 5.10: Influence of σ on the SpMV performance on SNB and K20 for the corner case test matrices. P^*_{SpMV} is determined as specified in eq. (4.9). Note the different value ranges on the ordinates.

Analysis methodology. As the kkt_power matrix reveals the most interesting characteristics and a large enough row count n to demonstrate the cache effects it will be analyzed in more detail on both architectures in the following. To quantify the influence of σ on the $\vec{x}$ overhead factor α, the actual SpMV data volume $V_{\text{SpMV,meas}}$ has to be measured using HPM, e.g., using LIKWID [169] on CPU architectures and nvprof [134] on GPUs. Then, following from eq. (5.3), it holds that

$$\alpha = \frac{V_{\text{SpMV,meas}} - \left[(v_{\text{el}} + 4)\, n_{\text{nz}}/\beta - 2 v_{\text{el}} n\right] \text{ B}}{(v_{\text{el}} n_{\text{nz}}) \text{ B}} \,. \tag{5.7}$$

The factor α can only be computed in this way if it can be assured that the matrix-induced memory traffic $(v_{\text{el}} + 4)\, n_{\text{nz}}$ scales with $1/\beta$. This is only the case on the CPU architectures, as C-first scheduling is employed for the SELL-C-σ SpMV implementation on them (cf. section 5.3). Otherwise, it is hard to tell whether additional memory traffic comes from the matrix or the input vector. However, the general data traffic overhead factor Ω can always be computed. It is defined as the ratio between the actual and minimal memory traffic of the entire SpMV operation,

$$\Omega = \frac{V_{\text{SpMV,meas}}}{V_{\text{SpMV,min}}} \geq 1 \,. \tag{5.8}$$

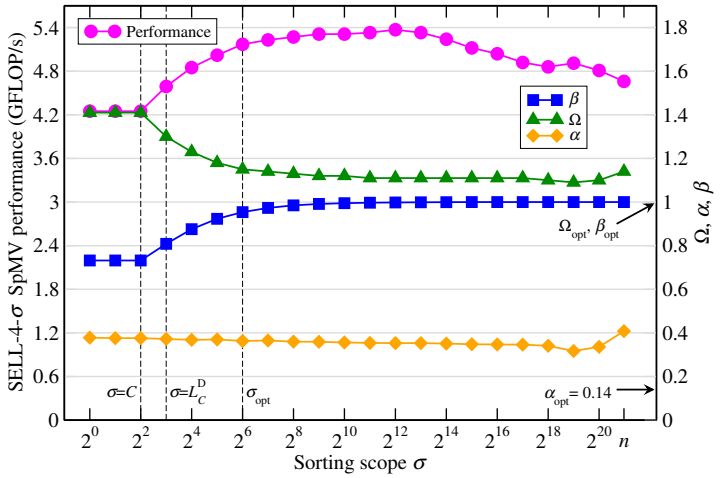
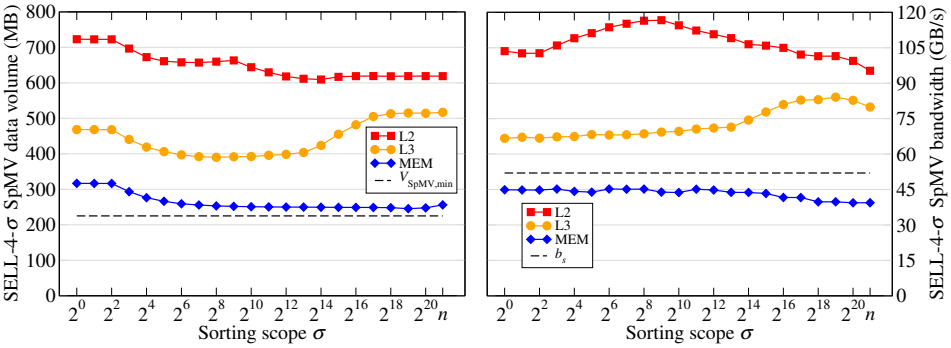


Figure 5.11: Influence of the sorting scope σ on the SpMV performance, the chunk occupancy β, the general overhead factor Ω and the SpMV overhead factor α on IVB for the kkt_power matrix and `STATIC,100` OpenMP scheduling. Note the logarithmic scale on the abscissa.

If ECC is enabled on GPU architectures, the measured data volume has to be scaled with a factor of 7/8, as one bit of each transferred byte contains parity information and is not "useful." This correction factor is not documented by the vendor, but it will be used as it gives precise results for low-level measurements.

Analysis on IVB. Figure 5.11 shows the σ-dependence of several metrics for the kkt_power matrix and $C = 4$ on IVB. The left axis indicates the SpMV performance while the right axis shows the chunk occupancy β, the $\vec{x}$ reuse factor α, and the SpMV data traffic overhead Ω. The first observation is that β increases once σ exceeds C and it approaches its optimal value of $\beta_{\text{opt}} = 1$, reaching a value of 0.95 at $\sigma_{\text{opt}} = 2^6$. On the other hand, α stays roughly constant up to $\sigma = 2^{20}$, indicating that sorting within this scope does not harm the access pattern to $\vec{x}$ in a way that it leads to an increase of vector-induced memory traffic. While α stays constant, the general overhead factor Ω and β are inversely proportional to each other. This is an obvious result, as any increase of chunk occupancy directly translates to a decrease of matrix-induced memory traffic and hence, to a decrease of the general traffic overhead Ω. With increasing β also the performance increases as the amount of "useless" transferred matrix data gets smaller. As σ approaches n, an increase of α, accompanied by an increase of Ω, can be observed. Hence, global sorting leads



(a) Data volume **(b)** Bandwidth

Figure 5.12: Influence of σ on the transferred data volume and bandwidth for different levels of the memory hierarchy on IVB for the kkt_power matrix and `STATIC,100` OpenMP scheduling. Note the logarithmic scale on the abscissas.

to a malicious access pattern to $\vec{x}$ which leads to increased memory traffic. In the same range, a performance decrease can be observed.

However, the performance descent starts even earlier, at $\sigma \gtrsim 2^{12}$. As α and Ω stay constant in this range, this initial decline cannot be attributed to increased memory traffic. A closer analysis of the data transfer volumes for main memory ("MEM"), L3 and L2 cache, as well as the minimum SpMV data volume $V_{\text{SpMV,min}}$ as given in eq. (4.3), is shown in fig. 5.12a. Figure 5.12b shows the respective bandwidth measurements. As expected, the MEM data volume curve is similar to the Ω curve of fig. 5.11. (Remember that $\Omega = V_{\text{meas}}/V_{\text{min}}$.) Once σ exceeds 2^{10}, the memory bandwidth decreases from about 45 GB/s to less than 40 GB/s, while the memory data volume stays constant. Obviously, a decline of memory bandwidth with constant data volume yields a performance decrease if memory bandwidth is the bottleneck. Hence, the drop of performance in this region as observed above can be attributed to the decrease of memory bandwidth, which itself is most probably due to a malicious data access pattern. A further observation in fig. 5.12a is a distinct increase of the L3 data volume for $\sigma \gtrsim 2^{12}$. However, fig. 5.12b shows that the L3 bandwidth increases at the same time, which makes it hard to draw credible conclusions from the data. To get a detailed understanding of the involved effects and bottlenecks one would have to conduct a deeper analysis. However, as those effects happen far beyond σ_{opt} (i.e., far beyond the relevant range as discussed below), this analysis is omitted here. All in all, it can be stated that although effects like worsening access patterns are hardly predictable, it is always a good advice to keep σ as small as possible.

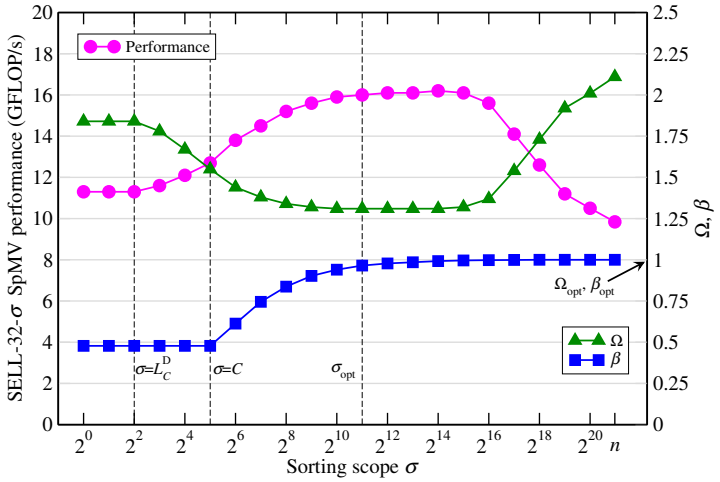


Figure 5.13: Influence of the sorting scope σ on the SpMV performance, the chunk occupancy β and the general overhead factor Ω on K20 for the kkt_power matrix. Note the logarithmic scale on the abscissa.

Analysis on K20. The influence of the sorting scope σ on the performance, the SpMV data overhead factor Ω and the chunk occupancy β for the kkt_power matrix on K20 can be seen in fig. 5.13. It is not possible to plot α due to the aforementioned fact that it is unknown whether any additional traffic in the "$\beta < 1$" range is matrix- or vector-induced. Again, the first observation is that β increases as soon as σ exceeds C and it approaches its optimal value of $\beta_{\text{opt}} = 1$. However, Ω decreases already for $\sigma < C$, namely at the point where σ exceeds the number of real double precision data items in a cache line L_C^{D}. This is due the aforementioned fine-grained inner loop of the CUDA SELL-C-σ SpMV kernel. In this case, a choice of $\sigma > L_C^{\text{D}}$ can lead to the emergence of zero-only cache lines which do not have to be loaded, leading to a decrease of Ω. Note that this effect is independent of the chunk height C. Obviously, a choice of $\sigma \leq L_C^{\text{D}}$ cannot have such an effect. At the same time as Ω decreases, an increasing performance can be observed. As this kernel's execution is limited by main memory bandwidth, any decrease in memory traffic can directly increase the performance. On the other hand, any increase of memory traffic or Ω, potentially leads to a performance degradation. This effect can be observed for $\sigma > 2^{15}$ in fig. 5.13. As no increase in matrix traffic occurs, it can be said that the increasing Ω in this region is solely due to an increasing α, i.e., due to a worse access pattern to $\vec{x}$. In general it can be seen that the performance and Ω show an inversely proportional behavior.

Best practices for the selection of σ. An "optimal" choice of σ_{opt} for a given C could be made by taking the minimum σ which results in a chunk occupancy larger than some threshold. The values of σ_{opt} for all test matrices, $C = 16$ and the constraint that β must be larger than 0.95 are given in table 2.2 on page 33. Those values have been obtained experimentally and the search space of σ has been restricted to powers of 2 (as well as n for global sorting) to reduce the time needed for the study. A more fine-grained search for σ_{opt} could deliver better results. Also, it could happen that the performance increases "randomly" for some value of $\sigma \neq \sigma_{\text{opt}}$ because of a more favorable access pattern or load balancing.

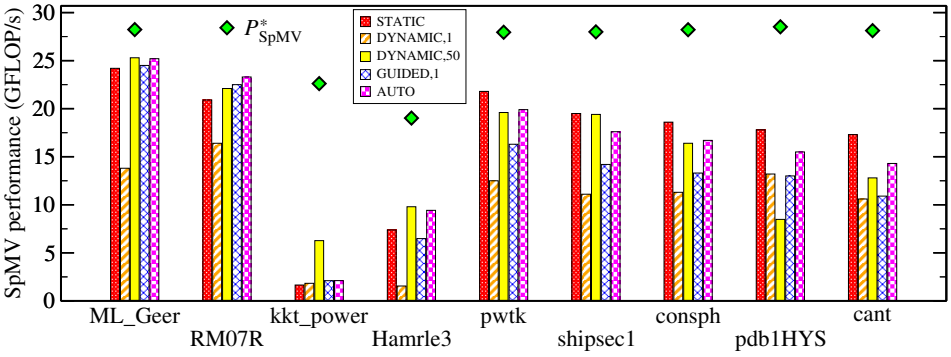
5.4.3 OpenMP Scheduling

Another important tuning factor for thread-parallel SpMV on CPUs is the workload scheduling of OpenMP threads. In this respect, often a trade-off has to be made between load balancing and efficiency. As thread parallelism is achieved using OpenMP in this work, the OpenMP scheduling terminology is used in the following. If each thread gets assigned a single portion of work, i.e., `STATIC` OpenMP scheduling is applied, there is a risk of load imbalance if the number of non-zeros differs significantly between the portions. On the other hand, a scheduling policy of `DYNAMIC,1` causes each loop iteration to be processed by an available thread, which makes it much less prone to load imbalances than `STATIC` scheduling but turns out to be inefficient because, e.g., the amount of work done by each thread at a time is too low compared to the OpenMP scheduling overhead. Hence, if `DYNAMIC` scheduling should be employed, it is preferred to chose a "sensible" chunk size which is large enough to enable efficient execution but small enough to retain the good load balancing properties. According to the OpenMP nomenclature, the portions of work assigned to a thread at a time will be called "chunks" in the following (mind the difference to chunks of C rows in a SELL-C-σ matrix).

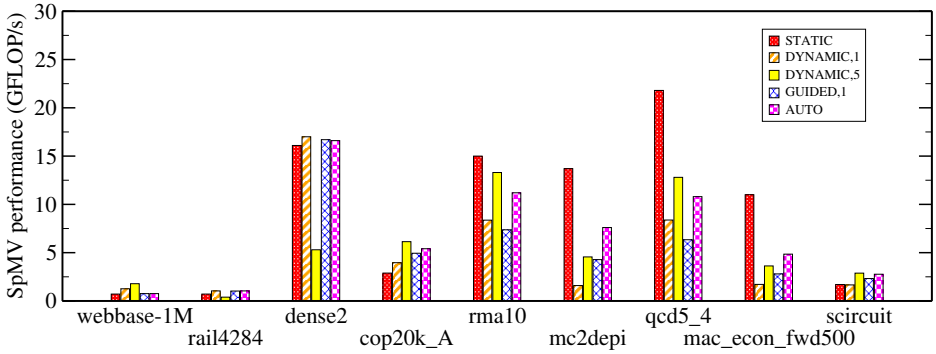
Susceptibility to load imbalance. The coefficient of variation c_v is defined as the ratio between the standard deviation and the mean, i.e., if the standard deviation is larger than the mean, c_v is larger than one. The c_v with respect to non-zero counts per matrix row is given in table 2.2 on page 33. A large value indicates that this matrix is susceptible to load imbalance if the wrong OpenMP scheduling (e.g., `STATIC`) is chosen. On the other hand, a matrix with $c_v = 0$ has the same non-zero count in each row and, if it also has a regular non-zero pattern, will not face load imbalance problems with any sensible scheduling strategy.

Workload scheduling on NUMA architectures. If the OpenMP threads span multiple NUMA domains, any scheduling strategy involves the danger of low performance due to remote data access. This is especially true for non-`STATIC` scheduling strategies. Concretely, if first-touch allocation of the matrix data happens in a parallel region and the matrix data is used (e.g., in an SpMV kernel) in another parallel region, it is likely to happen for non-`STATIC` scheduling that the thread mappings differ and threads access remote data elements. However, even in case of `STATIC` scheduling, remote data accesses may hamper performance. The indirect access to the right hand side vector may prevent an optimal mapping of this vector's data to the NUMA domains. However, even though `STATIC` scheduling does not guarantee optimal data access, the use of any non-`STATIC` scheduling is especially undesired if the OpenMP threads span several NUMA domains. Although page migration is supported by modern Linux kernels [110] and may alleviate the effects of bad NUMA placement, correct page placement is always preferable. Experimental results on the influence of OpenMP scheduling on NUMA architectures are presented in section 10.2.

Analysis on KNC. The effect of OpenMP scheduling is best studied on a many-core architecture like KNC. As noted in table 2.1, enabling three threads per core yields the highest main memory bandwidth, and analogously, in many cases the best SpMV performance on this architecture. to the large thread count of 180, deficiencies or strengths of one or the other scheduling strategy are likely to be revealed clearly. The sensitivity of this architecture with respect to OpenMP scheduling has also been noted in earlier studies [104]. The results of this analysis are also applicable to CPU architectures with a single NUMA domain and lower thread counts. Figure 5.14 depicts the SpMV performance with SELL-16-σ_{opt} (σ_{opt} was chosen as given in table 2.2) on KNC for all non-application test matrices with different OpenMP scheduling strategies. First, the simple and, in case of low c_v, likely to be efficient `STATIC` scheduling was chosen. The second choice is `DYNAMIC,1`, which always yields the best possible load balancing, but may be inefficient due to too little work per OpenMP chunk if n_{nzr} is not large enough. To overcome this inefficiency, `DYNAMIC` scheduling with a sensible chunk size of "`*`," which should depend on the number of matrix rows, is the next choice. This chunk size was chosen to be 50 for large matrices and 5 for small matrices, which turns out to be a good compromise between efficiency and load balancing. Further choices are `GUIDED,1` (which is equal to `DYNAMIC` but with a chunk size which decreases down to 1 between iterations) and `AUTO`, which delegates



(a) Large and well-behaved matrices. P^*_{SpMV} is determined as specified in eq. (4.9).



(b) Small and pathological matrices.

Figure 5.14: SELL-16-σ_{opt} SpMV performance with different OpenMP scheduling strategies on KNC.

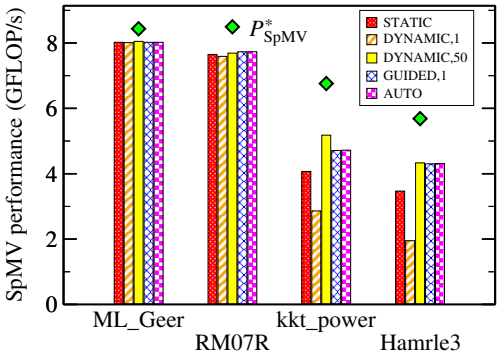


Figure 5.15: SELL-16-σ_{opt} SpMV performance with different OpenMP scheduling strategies on IVB. P^*_{SpMV} is determined as specified in eq. (4.9).

the decision to the compiler. While it is certainly possible that another, unregarded scheduling strategy yields better performance than this choice, the search space has to be limited and qualitative conclusions can be drawn. The set of matrices is split into the two groups of large and small matrices as already suggested in section 2.4, with one exception: the webbase-1M test case shows a pathological performance behavior and is shifted to the "small and pathological" matrices group. Roofline performance limits are only given for the "large and well-behaved" matrices. For the pathological test cases, the achieved performance is far below the Roofline limit, and for small matrices, the Roofline limit is not an applicable upper performance bound as the execution is not strongly limited by main memory bandwidth.

It can be seen that the best strategy strongly depends on the matrix. The fact that `STATIC` scheduling yields good performance for low-c_v matrices (e.g., ML_Geer, pwtk, dense2, mc2depi, qcd5_4) matches the expectation. Those matrices have a relatively constant non-zero count per row which prevents potential load balancing problems with `STATIC` scheduling. On the other hand, this strategy underperforms for matrices with a large coefficient of variation (e.g., kkt_power, Hamrle3, cop20k_A). `DYNAMIC,1` scheduling only shows good performance for high-n_{nzr} matrices (e.g., dense2) where enough work is done per loop iteration. `DYNAMIC,*` scheduling shows good or superior performance on matrices with a rather large non-zero count per row and a rather large c_v. Especially for kkt_power, it is significantly better than the other choices. `GUIDED,1` scheduling is never the best choice, but rarely the worst. The good load balancing properties cannot compensate for the potentially low efficiency. Lastly, `AUTO` scheduling seems to be a safe choice for decent performance, especially if no knowledge about the matrix is present. It only falls behind the best choice significantly for kkt_power and some of the small matrices. It is never the worst choice, and it is always better than `DYNAMIC,1` and `GUIDED,1`. As `AUTO` scheduling delegates the decision to the compiler, it may happen that future compiler versions can further increase the performance of this scheduling strategy.

Analysis on IVB. Figure 5.15 displays the influence of OpenMP scheduling on the SpMV performance for the corner case matrices on IVB. For the high-n_{nzr} matrices ML_Geer and RM07R scheduling does barely have an influence on performance. In those cases, even `DYNAMIC,1` scheduling is on par with the other strategies due to the fact that enough work is being done per loop iteration. `DYNAMIC` scheduling with a reasonable chunk size of 50 outperforms the other strategies for the low-n_{nzr} and high-c_v matrices kkt_-

power and Hamrle3. Also, it can be observed that `AUTO` scheduling is never a bad choice for the four corner case matrices. All in all, the same qualitative conclusions as above for KNC can be drawn, although the differences in performance are significantly smaller due to a lower thread count and a more "robust" architecture.

Best practices for OpenMP thread scheduling. Conclusively, good SpMV performance can usually be achieved with either `STATIC` or `DYNAMIC` scheduling with a reasonable chunk size. If the matrix has a low coefficient of variation, not enough rows to enable efficient `DYNAMIC` scheduling, or the threads span multiple NUMA domains, `STATIC` scheduling is a good choice. Otherwise, `DYNAMIC` with a chunk size large enough to occupy each thread sufficiently, but small enough to overcome possible load balancing problems can be recommended. Consequently, in the present set of test matrices, `STATIC` scheduling is used for ML_Geer, pwtk, shipsec1, consph, pdb1HYS, cant, dense2, mc2depi and qcd5_4, Graphene-N_x-N_y, and Topi-N_x-N_y-N_z. For the remaining large (small) matrices, `DYNAMIC,50` (`DYNAMIC,5`) scheduling is used.

5.5 Unified SELL-C-σ SpMV Performance

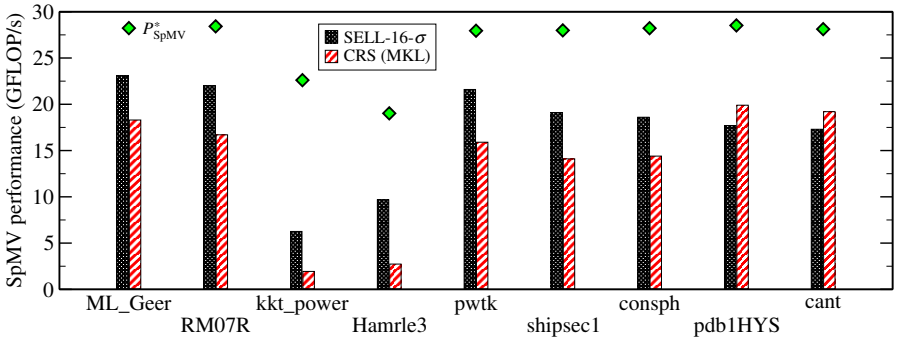
In this section, the performance of the unified and platform-agnostic storage format is compared against device-specific storage formats. The matrices are classified in the same two sets as described above. In addition, the application matrices relevant to the ESSEX project are taken into consideration. As this work focuses on sparse linear algebra towards the exascale era, the performance results from the "large and well-behaved" benchmark matrices are more relevant than the "small and pathological" group, and the application matrices are by far the most relevant ones.

Selection of tuning factors. The parameters of SELL-C-σ are chosen in a way that the same matrix can be used on all present architectures at high efficiency. The chunk height C is chosen to be 16. As described above, this may come at slightly lower performance than $C = 32$ on K20. However, this is accepted as $C = 16$ is expected to have better performance than $C = 32$ on the other architectures for aforementioned reasons. The sorting scope σ was set to σ_{opt} as given in table 2.2 on page 33. The OpenMP scheduling is chosen as described above.

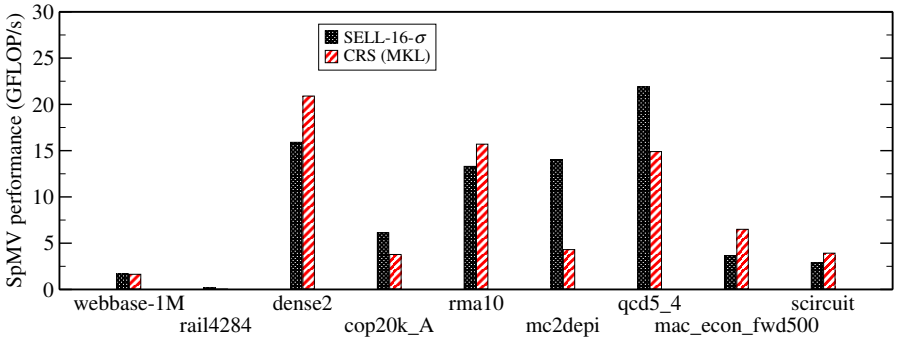
Baseline performance. On the CPU architectures IVB, HSW, KNC and KNL, the CSR storage format is selected as the baseline. This is a valid choice, as CSR is by far the most relevant and widely used format on those architectures. The Intel MKL offers an SpMV kernel based on CRS. As it is implemented by the CPU vendor, high performance can be expected and it is qualified as a valid baseline for performance comparison. Similarly, on K20, NVIDIA cuSPARSE [47] using CRS as well as Hybrid (HYB) is used for comparison. HYB combines ELLPACK and the COO format: matrix rows up to a certain threshold are stored in ELLPACK and "overlong" parts of rows are stored in COO. The format has been proposed as the GPU standard format for general sparse matrices by Bell and Garland [24]. The reason to also regard CRS on GPUs is that it allows to analyze the performance of a single storage format (i.e., CRS) on heterogeneous architectures and draw a direct comparison to the unified SELL-C-σ format.

Performance on KNC. The performance comparison on KNC of SELL-C-σ against CRS is shown in fig. 5.16. In the first group of matrices (fig. 5.16a), SELL-C-σ usually outperforms CSR. Due to its SIMD friendliness, the benefit of SELL-C-σ is most distinct for low-n_{nzr} matrices like kkt_power and Hamrle3, where SELL-C-σ delivers $3.3\times$ and $3.6\times$ the performance of CRS. SELL-C-σ falls behind CRS for the pdb1HYS and cant test cases. Both of those matrices have a relatively large n_{nzr} and a rather small row count. The large n_{nzr} enables high SIMD efficiency also with CRS, which diminishes the potential advantage of SELL-C-σ in this respect. The small row count gives CRS an advantage in terms of load balancing: As thread parallelism in CRS is done on a per-row base (instead of a per-chunk base in SELL-C-σ), it is easier to evenly distribute the work to the large number of 180 threads on this architecture. Still, the maximum benefit of CRS is only in the range of 10% for the large and well-behaved matrices. For all large and well-behaved matrices except kkt_power, SELL-C-σ achieves between 50% and 80% of the Roofline performance limit.

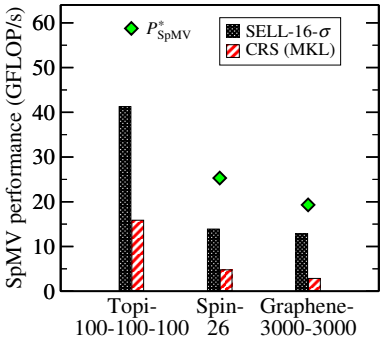
The group of small and pathological matrices reveals more diverse performance characteristics as can be seen in fig. 5.16b. The webbase-1M matrix has an extremely high coefficient of row length variation. In fact, one of its rows is fully populated while the average number of non-zeros per row is only 3.11. This characteristic is very likely to lead to performance hazards for any implementation which does not explicitly handle this case, e.g., due to a severe load imbalance. As neither the present CRS nor SELL-C-σ SpMV implementations take special precautions in this regard, the performance of



(a) Large and well-behaved matrices.



(b) Small and pathological matrices.



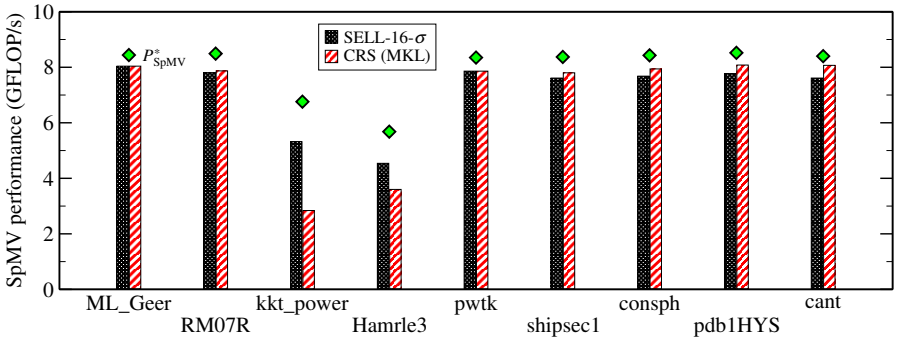
(c) Application matrices.

Figure 5.16: SpMV performance comparison of the unified SELL-16-σ_{opt} storage format and Intel MKL on KNC. P^*_{SpMV} is determined as specified in eq. (4.9). Note the different value ranges.

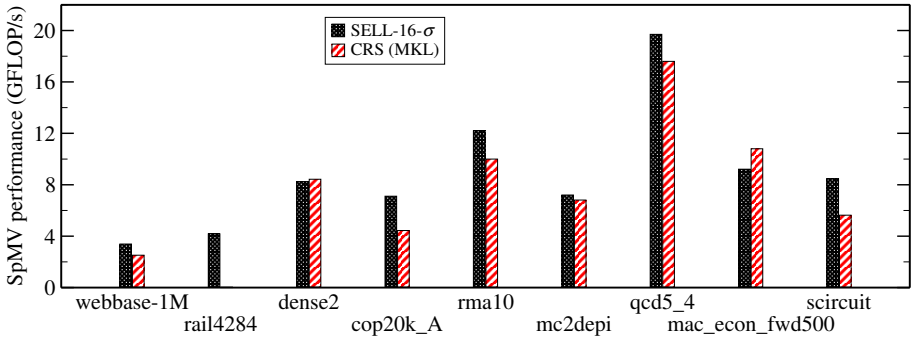
both formats for this matrix is very low. This matrix models web connectivity, which explains its very irregular structure. In contrast to this, the application matrices which are relevant for this work have far more regular structure and equally distributed non-zero counts. In case of the non-square rail4284 matrix, Intel MKL aborts the execution without further notice. This is the reason for "zero performance" of MKL for this matrix on all CPU architectures. The SELL-C-σ performance is very low for rail4284, which has several reasons: the low row count prohibits efficient use of all 180 threads and the large coefficient of variation impedes good load balancing. For the remaining small matrices, there is no clear outcome about which is the better storage format. The SpMV kernel with the dense2 matrix has 2000/C=125 outer loop iterations. However, 180 threads are present on the KNC. Hence, the low row count obviously prohibits an efficient execution of the SELL-16-σ SpMV with this matrix on this architecture. This could be enhanced by adapting the implementation, e.g., by letting more than one thread working on a chunk. The performance of both formats is especially low for the cop20k_A, mac_econ_fwd500 and scircuit, all of which show unfavorable patterns like a low row count and large coefficient of variation.

The most relevant matrices for this work are the three application matrices. Figure 5.16c reveals that SELL-C-σ outperforms CRS significantly for all of them. The Topi-N_x-N_y-N_z matrix has a regular structure and a moderate n_{nzr}. A possible explanation for the significant performance benefit of SELL-C-σ is that this matrix is complex-valued and the MKL implementation lacks efficiency for complex data types on KNC. In the Spin-N_{Up} case, the performance difference can also be explained by an inefficient CRS SpMV implementation and for Graphene-N_x-N_y, the low n_{nzr} of 4 comes into play which prohibits an efficiently vectorized CRS SpMV on KNC.

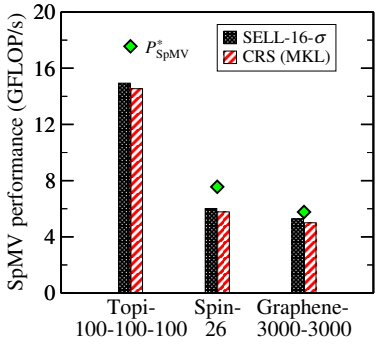
Performance on IVB. On IVB (fig. 5.17), the performance of SELL-C-σ and CRS is comparable for almost all test matrices. The largest deviation occurs for kkt_power, where SELL-C-σ outperforms CRS by a factor of almost $2\times$. Although it may appear that the choice of storage format is arbitrary in this case, an energy study in section 10.4 will reveal that using SELL-C-σ instead of CRS can offer possibilities for significant energy savings. Hence, even if there is no performance benefit on a full socket (as it is the case here), it may be useful to chose a SIMD-friendly storage format like SELL-C-σ. In contrast to KNC it can be seen that the achieved performance is closer to the Roofline performance limit. This indicates that standard CPU architectures like IVB are easier to harness at high efficiency compared to many-core architectures



(a) Large and well-behaved matrices.



(b) Small and pathological matrices.



(c) Application matrices.

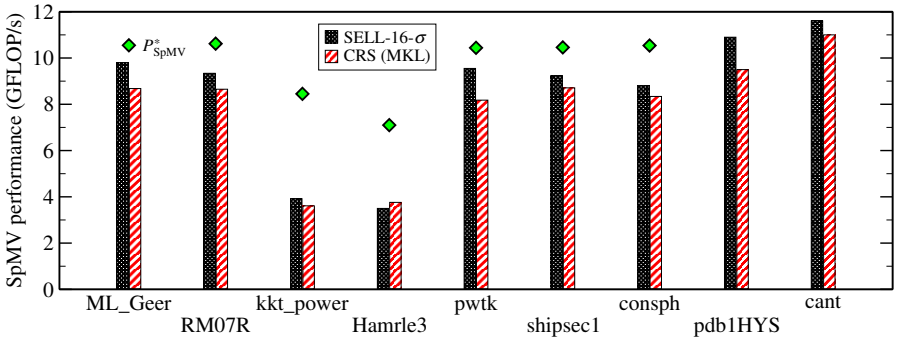
Figure 5.17: SpMV performance comparison of the unified SELL-16-σ_{opt} storage format and Intel MKL with CSR on IVB. P^*_{SpMV} is determined as specified in eq. (4.9). Note the different value ranges.

like KNC. Similar to KNC, SELL-C-σ yields on average a better performance than CRS on IVB.

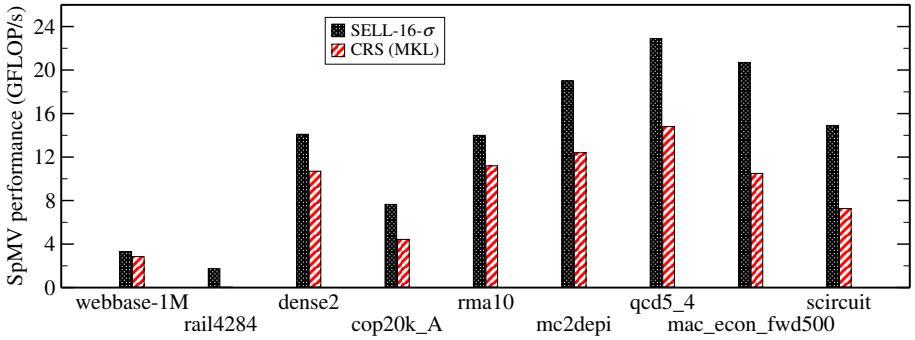
Performance on HSW. As described in section 2.1, the HSW architecture features AVX2 instructions as well as the CoD feature. The latter yields NUMA characteristics within a single socket, which entails potential performance hazards for thread-parallel programming. The OpenMP scheduling strategy has to be chosen with care in this case, as any kind of `DYNAMIC` scheduling will likely lead to a high ratio of remote memory accesses. Hence, simple `STATIC` scheduling has been chosen for all SELL-C-σ runs although this may not be the optimal choice in terms of load balancing (cf. section 5.4). For the MKL runs, NUMA interleaving as suggested in the documentation [90] has been employed. By this, memory pages will be placed across the NUMA domains in a round-robin manner, which usually leads to a balanced ratio of local and remote accesses. For SELL-C-σ, however, memory is allocated following the "NUMA first touch" policy. By this, the ratio of local accesses can be increased which leads to the performance benefit to be seen in fig. 5.18.

Note that the measured performance would exceed the Roofline limit for the large matrices with smallest row count cant and pdb1HYS. Although the entire working set exceeds the last level cache size of 35 MB, there is apparently still some re-use of data between successive iterations. This is due to the cache architecture and the validity of this statement can be confirmed with HPM measurements. Hence, if performance should be compared against a memory-limited Roofline model, it has to be assured that the execution is in fact limited by main memory bandwidth. The Roofline predictions are omitted for those two test cases in fig. 5.18a for this reason. Figure 5.18b and section 5.5 show that SELL-C-σ outperforms CRS for almost all small, pathological, and application matrices on HSW. It only falls behind by a small margin for Hamrle3 and the (complex-valued) Topi-N_x-N_y-N_z matrix.

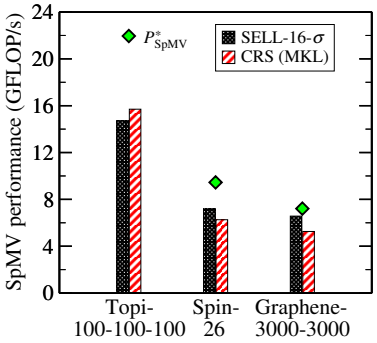
Due to the aforementioned NUMA characteristics, it is more difficult to match the Roofline limit on HSW than on IVB. Also, the benefit of SELL-C-σ over CRS, e.g., for the kkt_power matrix cannot be transferred from IVB to HSW due to this. One way to alleviate the NUMA problems is to confine the process to a single NUMA domain, i.e. half of HSW. Obviously, in order to use the full HSW chip, the code has to be enabled for inter-process parallelism, e.g., using MPI. If this is done, a straightforward comparison to MKL is no longer possible as it does not offer distributed memory parallel compute kernels. Experimental results on this topic can be found in section 10.2.



(a) Large and well-behaved matrices.



(b) Small and pathological matrices.



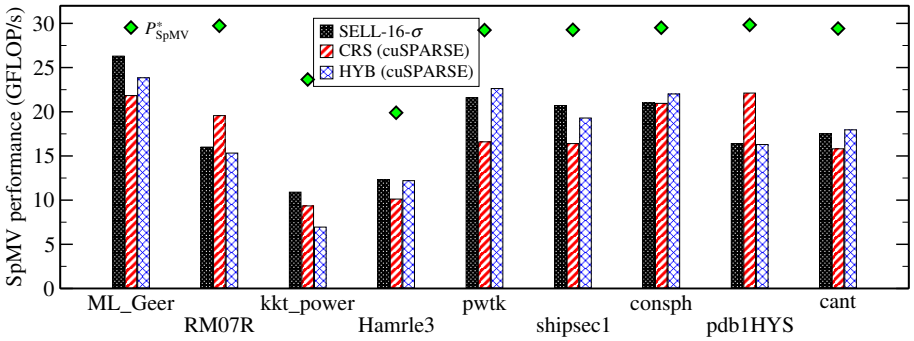
(c) Application matrices.

Figure 5.18: SpMV performance comparison of the unified SELL-16-σ_{opt} storage format and Intel MKL with CSR on HSW. P^*_{SpMV} is determined as specified in eq. (4.9). Note the different value ranges.

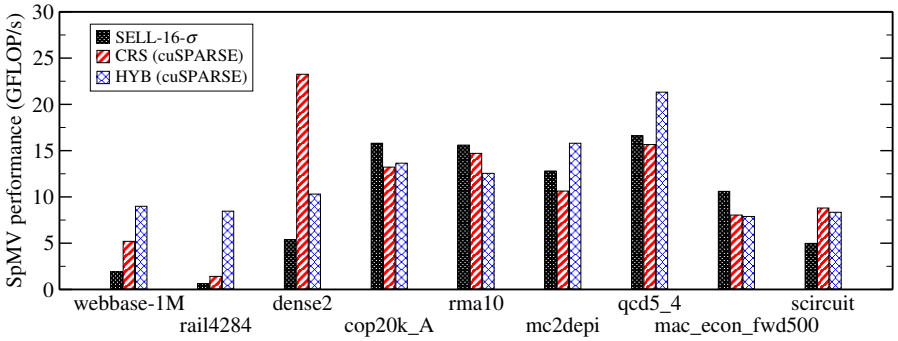
Performance on K20. On K20, the SELL-C-σ performance is compared against CRS as well as the HYB format in fig. 5.19. SELL-C-σ is slightly better or roughly on par with HYB for all large and application matrices. The largest benefit of SELL-C-σ can be observed for kkt_power, where it outperforms HYB by a factor of $1.6\times$. HYB reveals its strengths especially on matrices with a very malicious pattern, such as webbase-1M, rail4284 and scircuit.

Considering the large and application matrices, CRS can outperform SELL-C-σ only for RM07R and pdb1HYS by a factor of $1.2 - 1.3\times$. In all other cases, SELL-C-σ yields higher SpMV performance by up to a factor of $1.3\times$ for the pwtk test case. In the group of small matrices, CRS outperforms SELL-C-σ significantly for some test cases. The most distinct one in this respect is dense2. The SELL-C-σ implementation uses a single thread per matrix row, i.e., 2,000 threads are active for this test case. This number is too low for having an efficient execution which saturates the GPU bandwidth. One way to alleviate this problem is to launch multiple threads per matrix row. Earlier experiments have shown that using 16 threads per row on a dense matrix with 8,000 rows delivers optimal performance for SELL-C-σ according to the Roofline model [104]. Again, it should be noted that this deficiency of the present SELL-C-σ SpMV implementation is of not much relevance for the present work, as the focus is on *large-scale* and *sparse* linear algebra. The same problem is a cause for the low SELL-C-σ performance on rail4284. Furthermore, CRS outperforms SELL-C-σ for the aforementioned malicious test cases webbase-1M and scircuit. In summary, SELL-C-σ is on average the best choice for the relevant matrix groups of "large and well-behaved" and application matrices on this architecture.

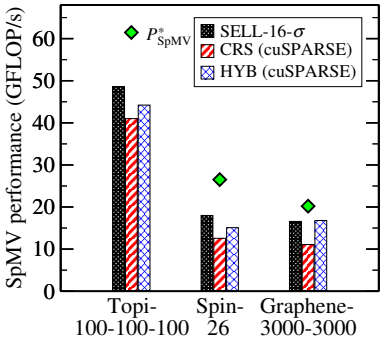
Performance on KNL and P100. To demonstrate performance portability of the SELL-C-σ SpMV operation across generations of hardware architectures, fig. 5.20 shows the SpMV performance comparison of SELL-16-σ_{opt} with device-specific formats for the large and well-behaved matrices on the newer hardware architectures KNL and P100. The key findings made above for the other architectures are still valid on the new generation architectures: SELL-C-σ is on par with or superior to device-specific formats for this set of matrices. On the GPU architecture P100, a good accordance with the Roofline limit P^*_{SpMV} can be observed, specifically for the very large and high-n_{nzr} matrices ML_Geer and Hamrle3. For the smaller matrices further right on the abscissa, it is harder to expose enough parallelism to match the Roofline expectation to a similar extent. In contrast to this, the performance on KNL falls behind the Roofline expectation more significantly. This can (at least



(a) Large and well-behaved matrices.



(b) Small and pathological matrices.



(c) Application matrices.

Figure 5.19: SpMV performance comparison of the unified SELL-16-σ_{opt} storage format and cuSPARSE with HYB and CSR on K20. P^*_{SpMV} is determined as specified in eq. (4.9). Note the different value ranges.

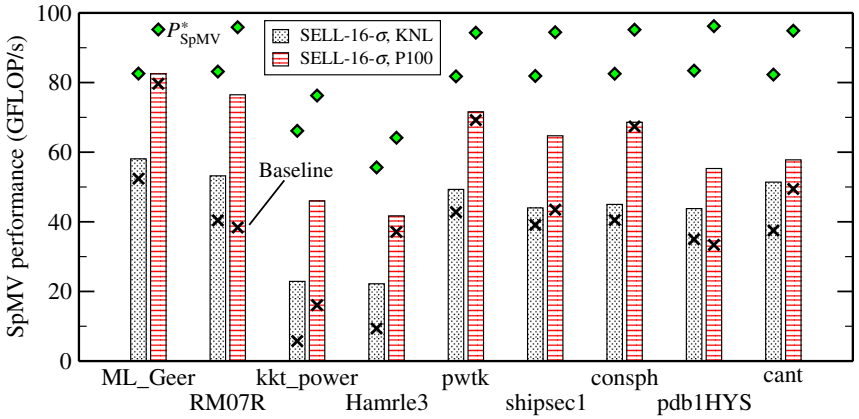
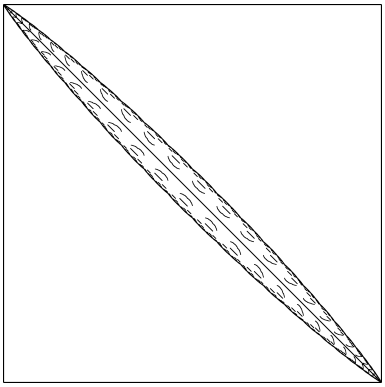


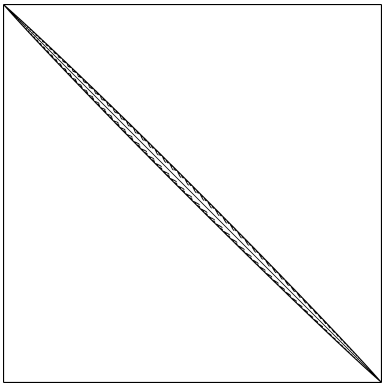
Figure 5.20: SpMV performance comparison of the unified SELL-16-σ_{opt} storage format and a device-specific baseline on KNL and P100. The baseline is CRS on KNL and HYB on P100. P^*_{SpMV} is determined as specified in eq. (4.9).

partly) be attributed to the fact that the achievable memory bandwidth of this architecture is much lower than b_s for read-intensive workloads such as SpMV (see fig. 4.2). In this case, a refined Roofline model taking the actually achievable bandwidth according to the read/write ratio for each test matrix into account would yield a better accordance.

Unified performance conclusions. All in all, it can be observed that the unified SELL-C-σ format yields performance which is comparable to or better than device-specific formats in almost all cases for relevant application matrices and further large and well-behaved benchmark matrices. It is sometimes possible to find a specific matrix format which yields better performance for a certain test case. Hence, more meaningful than a comparison against some baseline implementation is the comparison against the maximum achievable performance P^*_{SpMV} as predicted by the Roofline performance model. Especially on multi-core CPU architectures with a single NUMA domain, close to optimal performance can usually be achieved for the memory-limited test cases. If the performance deviates substantially from the Roofline prediction, a plausible explanation can be found. The most important result for this work is that the unified SELL-C-σ format yields on average the best performance with high Roofline efficiency for the relevant application matrices on all architectures.



(a) Original matrix (bandwidth 709,995).



(b) Matrix with applied RCM reordering (bandwidth 211,828).

Figure 5.21: Bandwidth reduction using RCM re-ordering with the Spin-26 matrix.

5.6 Matrix Bandwidth Reduction

The $\vec{x}$ overhead factor α should be reduced if possible to increase the arithmetic intensity of the SpMV operation (eq. (4.13)). The factor α exceeds its optimal value if $\vec{x}$ elements get evicted from the cache before they are re-used. This is closely related to the bandwidth of the matrix. The bandwidth of a matrix (not to be confused with memory bandwidth) is defined as "upper bandwidth + lower bandwidth + 1." The upper (lower) bandwidth is the maximum distance of a non-zero element of the matrix' upper (lower) triangular part to the matrix diagonal. For instance, a diagonal matrix has a bandwidth of 1, a tridiagonal matrix of 3 and a dense triangular matrix of n. In case of a diagonal matrix, α always attains its optimal value, as each $\vec{x}$ element is loaded and used exactly once from main memory. In case of a tridiagonal matrix an $\vec{x}$ element is firstly used in row i, then again in row $i+1$ and lastly in row $i+2$. In very simple cases like those it is easy to make predictions about α or set conditions on cache sizes if a certain α value should be achieved. However, in reality this is often not possible. Obviously, the access patterns and re-use intervals are far more complex for general sparse matrices. Also, the matrix bandwidth merely defines an upper bound for the off-diagonal distance. For many sparse matrices, the distance is lower for a significant amount of rows. However, reducing the matrix bandwidth is always a promising way to reduce α.

The perhaps most prominent algorithm for sparse matrix bandwidth reduction is Reverse Cuthill-McKee (RCM) as proposed by George and

Liu [73]. It is based on the original Cuthill-McKee (CM) algorithm [48] with reversed index numbers. In recent years, the research on this method has gained new interest and parallel RCM algorithms have been developed for shared [94] as well as distributed memory [15]. RCM is limited to symmetric sparse matrices, but similar approaches exist for the non-symmetric case as well [140]. Figure 5.21 visualizes the impact of RCM re-ordering on the Spin-26 matrix. Note that the qualitative results will transfer to other configurations of this test case, i.e., other values of N_{Up}. Using RCM re-ordering, the bandwidth of this matrix could be reduced by a factor of approximately $3.3\times$. The SpMV performance on IVB using Spin-26 and `DYNAMIC,50` OpenMP scheduling increases by 6% and α decreases to 47% of its original value. The rather small performance gain can be explained by the already relatively small bandwidth of the original matrix. The positive influence of bandwidth reduction gets amplified in the block vector case (see section 6.3).

Potential performance gains from matrix re-ordering are not restricted to a specific matrix format. Hence, it is not useful to include re-ordering in comparative studies of sparse matrix storage formats. Also note that matrix re-ordering is sometimes prohibited by the application or can lead to worse numerical behavior of, e.g., preconditioners. Hence, no matrix bandwidth reduction techniques are applied in the present performance studies.

5.7 Summary

In this chapter, SELL-C-σ was presented as a candidate for a platform-agnostic, unified sparse matrix storage format for high-performance SpMV on all relevant HPC architectures. Having a unified sparse matrix storage format for a wide range of architectures is beneficial in multiple ways. Besides obvious gains like simplified interfaces to and reduced code bases in numerical kernel libraries, it opens further possibilities in terms of usability and efficiency. For instance, exchange of matrix data between heterogeneous processors is greatly simplified, which opens possibilities for easy and low-overhead load balancing. The same holds for the implementation of fault tolerance mechanism like checkpointing in heterogeneous settings. Low-overhead exchange of matrix data is especially relevant for current and future heterogeneous architectures with a coherent, shared memory space like, e.g., NVIDIA Pascal GPUs using "Unified Memory" with their host CPUs (especially if they are connected via the proprietary NVLink interconnect [70] which is significantly faster than PCIe) or the heterogeneous exascale architecture proposed by Vijayaraghavany et al. [171]. Lastly, SELL-C-σ appears

to be future-proof. As shown in this work, its good performance properties can be transferred between generations of heterogeneous hardware architectures, and current trends in supercomputer hardware give rise to the assumption that this will hold true in the future.

The suitability of SELL-C-σ as a high-performance general sparse matrix storage format is also reflected in the inclusion of it into several numerical software frameworks. The developers of DUNE, a modular toolbox for solving partial differential equations with grid-based methods, have adopted SELL-C-σ and extended it to a block version to exploit block structures in the sparse matrix [22]. A further extension of SELL-C-σ is the SELL-P format as proposed by Anzt et al. [12]. In this variant, additional column padding is introduced which opens the possibility for launching more than one thread per matrix row which can enhance the performance for high-n_{nzr} matrices. This generalized version of SELL-C-σ has been included in the "sparse" module of the originally dense linear algebra library MAGMA [166]. SELL-C-σ has also found its way into ViennaCL, a linear algebra library for computations on many-core architectures and multi-core CPUs [144]. Graph algorithms are a topic which is related to sparse linear algebra, as graphs can be expressed as sparse matrices and vice versa. SELL-C-σ has recently been introduced in the graph algorithm community, being the underlying data format for the vectorizable graph representation "SlimSell" [26].

6 High-Performance Sparse Matrix-Multiple Vectors Multiplication

In this chapter, the general SpMMV operation $\vec{Y} \leftarrow \vec{Y} + H\vec{X}$ with $\vec{X}, \vec{Y}$ being blocks of n_b vectors is analyzed. Besides its single-vector counterpart, the SpMMV operation is one of the most crucial and time-consuming compute kernels in many sparse linear algebra algorithms. Block vectors may appear due to several reasons. For example, if a polynomial filter like in ChebFD should be applied to multiple vectors, those could be packed into a block vector and operations like SpMMV could be used in the solver. Furthermore, block vectors may arise from algorithmic optimizations, e.g., as it is done in BJDQR (cf. section 3.4) for increasing robustness [142]. In principle, an SpMMV operation could be implemented as a sequence of n_b SpMV calls. The use of custom SpMMV kernels is motivated by potential performance gains. As shown in section 4.2.3, the arithmetic intensity increases with n_b and thus, a performance gain can be expected in the bandwidth-limited case.

Although the SpMV and SpMMV performance analyses are similar to some extent, there are some peculiarities to the SpMMV operation which will be discussed in the following. Besides the sparse matrix storage format, the SpMMV performance depends the storage format of the block vectors as a crucial tuning factor. Possibilities in this regard and their respective SpMMV kernel implementation are analyzed in section 6.1. Section 6.2 analyzes the relation between the sparse matrix storage format and the SpMMV performance. Afterwards, the idea of matrix bandwidth reduction and its influence on SpM(M)V performance as discussed in section 5.6 is taken up and analyzed in section 6.3.

6.1 Block Vector Storage and SpMMV Implementation

One of the most fundamental design decisions when it comes to the implementation of block algorithms affects the storage order of block vectors.

Listing 6.1 SELL-1-σ SpMMV kernel with column-major block vector storage.

```
1 for (int i = 0; i < n; i++) {
2   double tmp[n_b];
3   for (int b = 0; b < n_b; b++) {
4     tmp[b] = y[b*n+i]; // y[] gather
5   }
6
7   for (int j = chunkptr[i]; j < chunkptr[i+1]; j++) {
8     double mval = val[j]; // scalar access to matrix values
9     int    xrow = col[j]; // scalar access to column indices
10    for (int b = 0; b < n_b; b++) {
11      tmp[b] += mval * x[b*n+xrow]; // x[] gather
12    }
13  }
14
15  for (int b = 0; b < n_b; b++) {
16    y[b*n+i] = tmp[b]; // scalar write of y[] data
17  }
18 }
```

Column-major (one-by-one) storage. A naïve approach is to store the vectors one after the other, i.e., the block vector would have column-major storage order. Listing 6.1 shows the SELL-1-σ SpMMV kernel with column-major block vector storage. As can be seen in lines 4, 11 and 16, the multiplication of a single matrix element with n_b $\vec{X}$ elements and the update of $\vec{Y}$ causes that the block vector elements are accessed with a large stride of n, which hampers vectorized processing and efficient data access. Despite the unfavorable data access pattern, column-major storage of block vectors may carry a further performance hazard which concerns the re-use of input vector data. Whenever an $\vec{X}$ elements gets accessed, a full cache line containing the element and some further elements of the same vector is loaded from main memory. As it is not clear whether those additional elements are needed as long as they stay in the cache, this bears a potential overhead, which can be avoided with row-major storage (see next paragraph).

Row-major (interleaved) storage. In contrast to column-major storage, interleaved (i.e., row-major) storage of block vectors allows for vectorized loads of $\vec{X}$ data (with remainder loops if n_b is not a multiple of the SIMD width) and perfect use of loaded cache lines if n_b is a multiple of the cache line size (disregarding the potential effect of adjacent cache line prefetching). This limits the maximum value of α in eqs. (4.17) and (4.23) to 1.

Listing 6.2 SELL-1-σ SpMMV kernel with row-major block vector storage.

```
1 for (int i = 0; i < n; i++) {
2   double tmp[n_b];
3   for (int b = 0; b < n_b; b++) {
4     tmp[b] = y[i*n_b+b]; // packed y[] load
5   }
6
7   for (int j = chunkptr[i]; j < chunkptr[i+1]; j++) {
8     double mval = val[j]; // scalar access to matrix values
9     int    xrow = col[j]; // scalar access to column indices
10    for (int b = 0; b < n_b; b++) {
11      tmp[b] += mval * x[xrow*n_b+b]; // packed x[] load
12    }
13  }
14
15  for (int b = 0; b < n_b; b++) {
16    y[i*n_b+b] = tmp[b]; // packed y[] store
17  }
18 }
```

The SELL-1-σ SpMMV kernel for row-major block vectors is shown in listing 6.2. Contrary to the column-major case, all block vector accesses (lines 4, 11 and 16) have a stride of 1, i.e., vectorized access can easily be achieved using compiler intrinsics or directives. The fact that the Intel MKL and NVIDIA cuSPARSE include row-major versions of SpMMV confirms the relevance of this operation.

CPU implementation. The CPU implementation of the row-major SpMMV kernel basically follows listing 6.2. Thread parallelism is achieved in the outer loop, and the innermost loops can be equipped with compiler pragmas to guide vectorization. The frequent occurrence and relatively small iteration count of loops over n_b give rise to potential performance benefits if the loop trip count n_b is known at compilation time. Automatically generating kernels for user-defined values of n_b helps to retain code flexibility and maintainability while increasing the performance. Details on this mechanism can be found in section 9.5.

Performance comparison. The expected performance benefit of row- over column-major block vector storage for SpMMV is substantiated by the measurements shown in fig. 6.1, which compares the two block vector storage variants for a custom SELL-C-σ SpMMV kernel and Intel MKL with

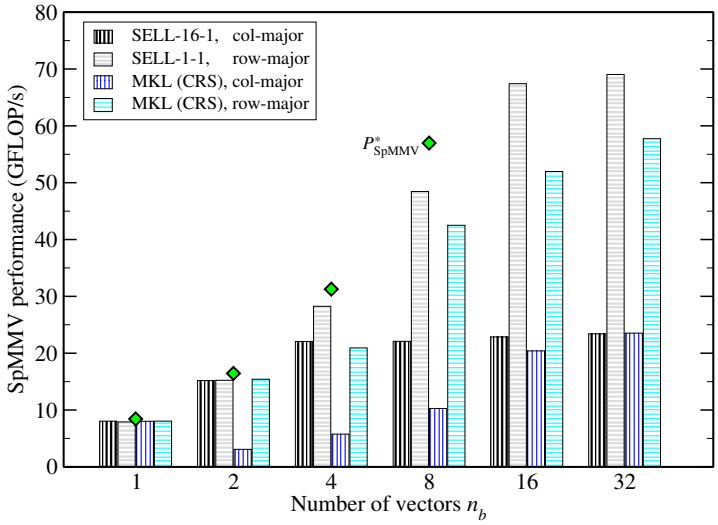


Figure 6.1: SpMMV performance for row- and column-major block vectors with the ML_Geer test case on IVB. The Roofline performance limits P^*_{SpMMV} are computed according to eq. (4.21) and are omitted for the largest n_b values (97 GFLOP/s at $n_b = 16$ and 150 GFLOP/s at $n_b = 32$) for the sake of illustration. Note the logarithmic scale on the abscissa.

CRS. For the SELL-C-σ implementation, automatically generated SpMMV kernels for discrete values of n_b as described above are used.

The performance at $n_b = 1$ (i.e., SpMV) is the same for all variants, which meets the experience and expectation from the SpMV performance analysis in chapter 5. For larger n_b counts, row-major storage is always better than column-major storage due to the aforementioned reasons. Furthermore, the SELL-C-σ implementation outperforms MKL for all $n_b \geq 4$, which is at least partly due to a systematic advantage from automatically generated tailored kernels with compile-time-defined n_b. However, the comparison between generated kernels and general implementations as in MKL can still be considered valid or "fair," as there are no reasons which prevent library implementers from including SpMMV kernels with hard-coded n_b for moderate sized block vectors (where such an optimization obviously has the largest advantage). A comparison to the Roofline performance limit P^*_{SpMMV} reveals very high efficiency for low values of n_b. For higher values, the execution decouples from main memory bandwidth and the Roofline prediction (which assumes strong limitation by main memory bandwidth in our case) is not matched anymore. A refined performance model for the KPM algorithm, which is similar to plain SpMMV, is given in section 8.1.2 and explains this effect in more detail.

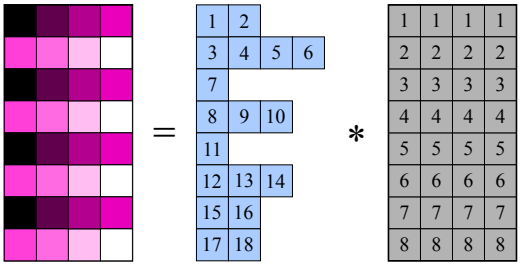
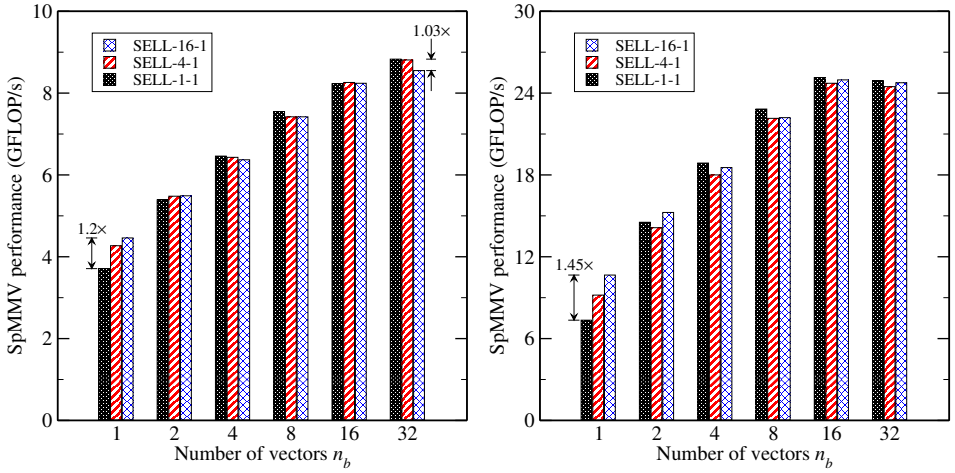


Figure 6.2: Thread mapping for the SELL-1-σ SpMMV operation on a GPU. Each thread computes one element of the results block vector and has a different color. The warp size is assumed to be 8.

All in all, row-major storage is preferred if high SpMMV performance should be achieved. However, using row-major storage could sometimes problematic or even prohibited, e.g., if block operations should be integrated in larger software stacks where such fundamental changes of data structures cannot be done. However, this is not the case in the present work and row-major storage will be used for all algorithms.

GPU implementation. Figure 6.2 illustrates the GPU SpMMV implementation for row-major block vectors. In terms of parallelization and having in mind the SIMD/SIMT analogy, it is similar to the CPU case. Perfectly coalesced access to $\vec{X}$ and $\vec{Y}$ data is easily achieved if n_b is a multiple of the SIMT width, i.e., $n_b = 16i,\ i \in \mathbb{N}^+$ for real double precision data on the present GPU architectures (note the global memory coalescing rules as explained in section 2.1). Assuming $n_b = 16$, all threads of a half warp load the same matrix value and column index and consequently, access 16 consecutive $\vec{X}$ elements.

Consequences for block algorithms. As it has become clear that SpMMV favors row-major storage of block vectors, the question about the influence of block vector storage on the performance of block vector-vector (Block-BLAS-1) operations remains open. The arithmetic intensity of other operations such as Block-BLAS-1 operations does not depend on n_b. Hence, there is no performance difference between one Block-BLAS-1 operation compared to a sequence of n_b "normal" BLAS-1 operations. Despite the fact that the BLAS-1 interface exposes the possibility to define strides between vector elements, row-major storage of block vectors necessitates a manual implementation of BLAS-1 operations which contain reductions, if high efficiency should be



(a) IVB **(b)** K20m

Figure 6.3: SpMMV performance for Hamrle3, different storage formats and different numbers of vectors on IVB with `AUTO` OpenMP scheduling and K20m. Note the logarithmic scale on the abscissas and the different value ranges on the ordinates.

achieved. This is due to the fact that the strided BLAS-1 kernels inhibit inherent performance hazards such as scalar memory operations and inefficient cache line usage. Also, in case of reduction operations, the cost of distributed-memory communication favors a single Block-BLAS-1 operation over a sequence of n_b BLAS-1 operations, as the cost of latency only has to be paid once, instead of n_b times.

6.2 Performance Influence of the Sparse Matrix Storage Format

In contrast to SpMV, the SpMMV performance is likely to be less sensitive to the sparse matrix storage format, as the matrix traffic plays a smaller role as n_b (and consequently, the vector traffic) increases. The relative impact of matrix and vector traffic depends on the values of n_{nzr} and n_b. Furthermore, vectorized access to vector data is easily achieved for row-major block vectors and sufficiently large n_b as shown above in listing 6.2. Without sacrificing performance, the matrix access can then be done in a non-vectorized fashion, which alleviates the necessity of a SIMD-friendly sparse matrix storage format.

Performance comparison. Figure 6.3a shows the SpMMV performance on IVB for three different storage formats and a range of n_b values. The OpenMP scheduling policy has been set to "`AUTO`." The performance at $n_b = 1$ agrees with previous results as shown in fig. 5.17a, although a custom SELL-1-1 (CRS) SpMV kernel has been used here instead of MKL. This matrix has a very low n_{nzr} value of 3.81 and thus requires a SIMD-friendly sparse matrix storage format for optimal SpMV performance due to the reasons explained in section 5.1. Due to this, SELL-4-1 and SELL-16-1 outperform SELL-1-1 at $n_b = 1$ by approximately 20% on IVB and 45% on K20m. This effect vanishes for larger vector counts and the performance of all three storage formats is comparable for all values of $n_b \geq 2$. Note that the $n_b = 2$ kernel uses Streaming SIMD Extensions (SSE) instructions on IVB (instead of AVX which would have double the SIMD width). If possible, the block vector size should be chosen to be a multiple of the SIMD width to guarantee optimal vectorization, i.e., n_b should be a multiple of 4 on IVB and a multiple of 16 on K20m for real double precision data. In this experiment, SELL-16-1 even falls behind SELL-1-1 by 3% for $n_b = 32$ on IVB, which could be explained by better spacial locality properties of the SELL-1-1 kernel.

Best practices for the selection of sparse matrix storage formats. Similar results can be obtained for all test matrices, with the exception that there may be a smaller or even no performance difference at $n_b = 1$ for large-n_{nzr} matrices. Conclusively, it can be said that the choice of the sparse matrix storage format is not crucial for SpMMV performance in case of sufficiently large n_b. In other words, even CRS (SELL-1) may be a good choice for optimal performance on a wide range of architectures for SpMMV in this case.

6.3 Performance Influence of Matrix Bandwidth Reduction

As explained in section 5.6, the SpMV performance can often be increased by reducing the overhead factor α (see eq. (4.13)). One way to achieve this is by decreasing the matrix bandwidth, for example using RCM re-ordering as explained in section 5.6. For $n_b > 1$, this is even more promising as will be shown in the following. Equation (5.3) indicates that the $\vec{X}$-induced data traffic scales with a factor of αn_b. Hence, minimizing α becomes more and more important for increasing values of n_b. Using HPM, the transferred data

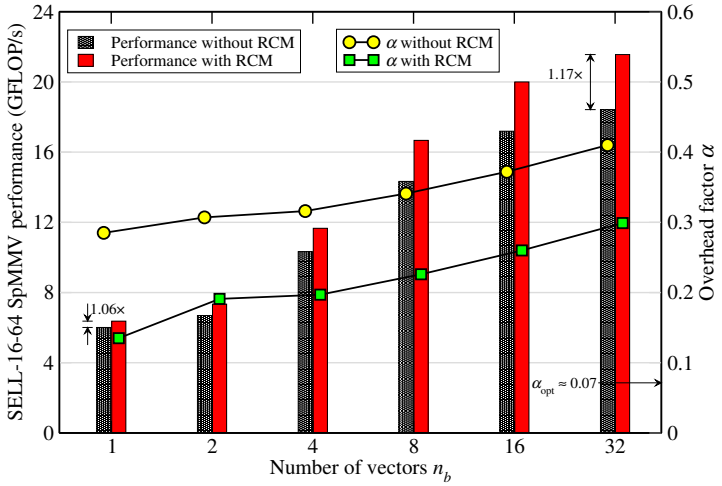


Figure 6.4: SpMMV performance and measured overhead factor α for Spin-26 with SELL-16-64 on IVB and `DYNAMIC,50` OpenMP scheduling. Note the logarithmic scale on the abscissa.

volume $V_{\text{SpMMV,meas}}$ can be measured and eq. (5.3) can be solved for α,

$$\alpha = \frac{V_{\text{SpMMV,meas}} - [(v_{\text{el}}+4)n_{\text{nz}}/\beta - 2n_b v_{\text{el}} n]\ \text{B}}{(n_b v_{\text{el}} n_{\text{nz}})\ \text{B}}\,. \tag{6.1}$$

Figure 6.4 depicts the SpMMV performance and α for varying block vector size on IVB using the original Spin-26 test case as well as the same matrix with an RCM re-ordering applied (as shown in fig. 5.21). It can be seen that the SpMMV performance for the matrix with applied RCM re-ordering is always higher than with the original matrix. The performance benefit increases with n_b, from 6% at $n_b = 1$ to 17% at $n_b = 32$. The $\vec{X}$ overhead factor α is always smaller for the matrix with applied RCM re-ordering, which is a result of the better cache locality coming from the bandwidth reduction.

6.4 Summary

In this chapter, the performance of the SpMMV operation has been analyzed and its characteristics have been described in comparison to SpMV. A crucial design decision in block algorithms affects the storage order of block vectors. It has been shown and explained that row-major storage yields higher SpMMV performance than column-major storage. If this is done, the performance in comparison to a sequence of SpMV operations can be increased

significantly, although the achieved speedup may not meet the model expectation. In contrast to SpMV, the storage format of the sparse matrix has a smaller impact on the SpMMV performance, especially for large values of n_b. It has also been shown, that a reduction of matrix bandwidth is a promising way to increased SpMMV performance, especially for large vector counts.

7 High-Performance Tall & Skinny Matrix-Matrix Multiplication Operations

Block sparse linear algebra algorithms frequently contain multiplications of block vectors, which can be seen as tall & skinny dense matrices. Besides the operations needed in BJDQR (see section 3.4), the orthogonalization of block vectors, e.g. using SVQB [160], requires similar kernels. While any GEMM implementation can handle such multiplications, severe performance drawbacks can be revealed, which requires manually optimized implementations. The goal of this chapter is not to re-implement GEMM or outperform established libraries which address the general case. Rather, it should be made clear through performance models that even established libraries like Intel MKL can have a significant lack of performance for some problem settings of sparse linear algebra. In this case, manual implementations are necessary for high performance.

This work does not cover T&S-GEMM operations on GPUs, which is the topic of an associated Master's thesis [63].

7.1 Tall & Skinny Matrix-Matrix Multiplication Performance

The GEMM operation is the main building block of the HPL benchmark [87] and highly optimized implementations for square matrices exist for all relevant architectures. For large and square matrices, this benchmark is usually compute-limited on all relevant current architectures and its performance typically gives a good estimate about the practically attainable peak performance on a given architecture. However, as already described in section 4.2.4, the arithmetic intensity of GEMM depends on the matrix dimensions. Typically, the kernel gets more and more bandwidth-limited as the matrices become more and more non-square.

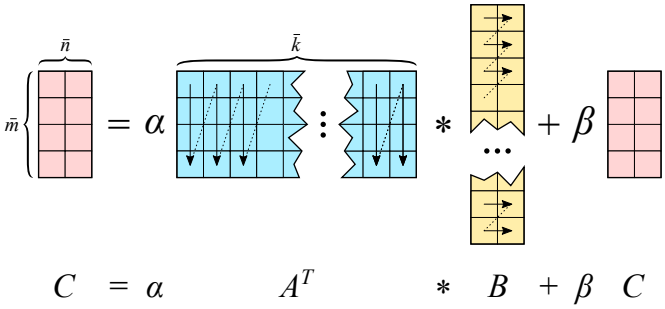


Figure 7.1: Schematic of the TSMTTSM operation.

Methodology. All presented T&S-GEMM kernels are implemented in plain C and make use of automatic code generation (see section 9.5). Compiler directives are used to achieve efficient code vectorization. Although hand-vectorized code using compiler intrinsics can lead to slightly better performance for specific input data, benchmark experiments have shown that for the present T&S-GEMM kernels and parameter ranges, high performance can be also be achieved with compiler-vectorized code. Especially in case of odd dimensions, manual vectorization is sometimes hard to implement and the high efficiency of the compiler-vectorized code alleviates the necessity for a manually vectorized implementation. It should also be noted that plain, vectorizable C code is more portable than hand-vectorized code using compiler intrinsics which correspond to a specific instruction set.

In this analysis, block vectors are always stored row-major, as this yields higher performance of SpMMV (see section 6.1) which is often the key operation in block algorithms. Moreover, all block vectors considered in this chapter have a row count of $10,000,000$ (which does not restrict generality, as long as it is sufficiently large) and the short dimensions will be varied from $1, \ldots, 32$. The short dimensions are assumed to be always non-padded, i.e., the stride is equal to the dimension.

Intel MKL will be used as the baseline BLAS implementation. It is usually considered very efficient for many applications and a significant development effort is being put into it. This qualifies it as a suitable baseline for this work.

7.2 Tall & Skinny Matrix Transposed-Tall & Skinny Matrix Multiplication

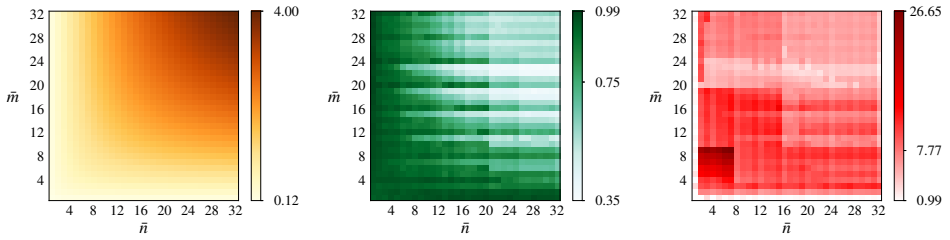
The TSMTTSM operation can be considered as an inner product of block vectors where $\bar{k} \gg \bar{m}, \bar{n}$. In a sparse linear algebra algorithm, $\bar{k}$ corresponds

Listing 7.1 TSMTTSM kernel for real double precision data with row-major block vector storage of A and B, column-major storage of C, $\alpha = 1$, $\beta = 1$, and $\bar{k}$ a multiple of two. Possibly required alignment/padding of data is omitted here for brevity.

```
1 #pragma omp parallel
2 {
3   double c_tmp[n*m] = {0.};
4
5 #pragma omp for
6   for (int row = 0; row < k-1; row+=2) {
7     for (int bcol = 0; bcol < n; bcol++) {
8 #pragma simd
9       for (int acol = 0; acol < m; acol++) {
10        c_tmp[bcol*m+acol] +=
11          a[(row+0)*m + acol] * b[(row+0)*n + bcol] +
12          a[(row+1)*m + acol] * b[(row+1)*n + bcol];
13      }
14    }
15  }
16
17 #pragma omp critical
18  for (int bcol = 0; bcol < n; bcol++) {
19 #pragma simd
20    for (int acol = 0; acol < m; acol++) {
21      c[bcol*m+acol] += c_tmp[bcol*m+acol];
22    }
23  }
24 }
```

to the dimension of the system matrix n and $\bar{m}$ and $\bar{n}$ are column counts n_b of block vectors. A schematic of the operation is shown in fig. 7.1. In BJDQR, it holds that $\bar{m} \geq \bar{n}$ and in SVQB, $\bar{m} = \bar{n}$. Nevertheless, also the performance for the case that $\bar{m} < \bar{n}$ will be shown and analyzed, as it may be of relevance for other algorithms. The result of this operation is a small matrix whose storage order is irrelevant as it can be easily transposed at low overhead.

Implementation. Listing 7.1 shows a TSMTTSM kernel using OpenMP for thread parallelism and "`#pragma simd`" for compiler-driven vectorization. Thread-parallel work sharing is achieved along the $\bar{k}$ dimension (lines 5 to 15) and each thread owns a partial result matrix (line 3) which get reduced at the end of the kernel in a serial way (lines 17 to 23). As can be seen in lines 6, 11 and 12, the kernel is two-way unrolled in the long (i.e., $\bar{k}$) dimension to decrease transfers of partial result matrix data. The values of α and β have no significant influence on the TSMTTSM performance because there are only



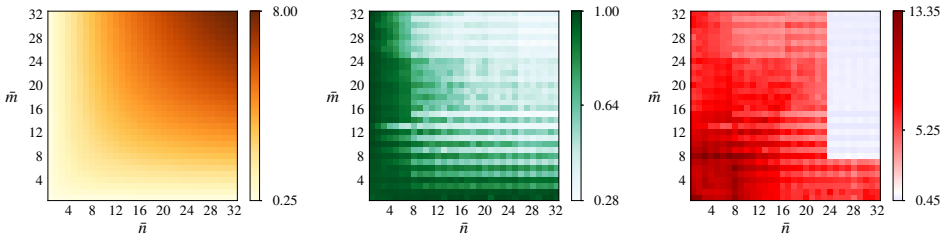
(a) Arithmetic intensity **(b)** Roofline efficiency **(c)** Speedup over Intel MKL

Figure 7.2: Arithmetic intensity, Roofline efficiency, and speedup over Intel MKL for the TSMTTSM operation with real arithmetic on IVB.

$\bar{m} \times \bar{n}$ multiplications with each of them. Hence, the most simple scenario, i.e., $\alpha = 1$ and $\beta = 0$ will be used for the following performance analysis.

Performance analysis. Figure 7.2b shows the TSMTTSM efficiency on IVB. In general, at least 35% of Roofline performance can be achieved in the entire $\bar{m}, \bar{n}$ range. Especially in the strongly bandwidth-limited region, i.e., in case either $\bar{m}$ or $\bar{n}$ is very small (see fig. 7.2a), the manual implementation proves to be very efficient. The comparison with Intel MKL in fig. 7.2c shows that the manual implementation is on par with MKL for the case where either $\bar{m}$ or $\bar{n}$ is 1. Otherwise, significant speedups of up to $27\times$ can be achieved and the average speedup in the entire $\bar{m}, \bar{n}$ range is around $8\times$. In general, larger speedups are obtained in the bandwidth-limited region. This agrees with the aforementioned statement that MKL features efficient GEMM kernels for compute-limited scenarios. Note that the performance of some data points could be increased even further if the two input block vectors in the kernel get swapped. This requires an inexpensive transposition of the result matrix.

Figure 7.3 demonstrates that the findings can be transferred to a different hardware architectures as well as complex numbers/arithmetic. Note that in the complex case, the first input matrix is conjugated as well as transposed. Concerning Roofline efficiency (cf. fig. 7.3b), similar observations as in the real case can be made: The TSMTTSM kernel ranges between 28-100% of Roofline efficiency, with the larger numbers being obtained in the strongly bandwidth-limited areas with a low arithmetic intensity. It can be seen in fig. 7.3c that the manual implementation still outperforms Intel MKL by a factor of up to $13\times$ and $5\times$ on average. The arithmetic intensity of TSMTTSM is twice as high for complex numbers compared to real numbers (see eq. (4.40)).



(a) Arithmetic intensity **(b)** Roofline efficiency **(c)** Speedup over Intel MKL

Figure 7.3: Arithmetic intensity, Roofline efficiency, and speedup over Intel MKL for the TSMTTSM operation with complex arithmetic on HSW.

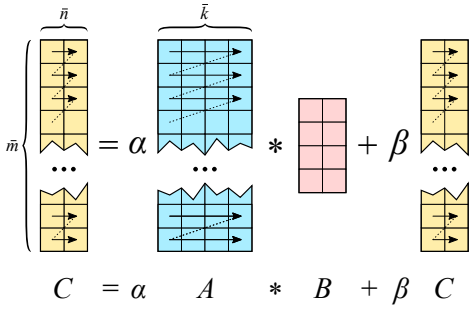


Figure 7.4: Schematic of the TSMM operation.

In fact, fig. 7.3a reveals that for large $\bar{n}$ and $\bar{m}$, the arithmetic intensity gets close to the inverse machine balance of this architecture of 9 FLOP/B. The kernel can no longer be regarded as strongly bandwidth-limited. Hence, it is plausible that MKL, which is optimized for the compute-limited, high-intensity case, does not fall so far behind the manual implementation. In fact, MKL outperforms the manual kernel by a factor of $2\times$ for $\bar{n} \geq 24$ and $\bar{m} \geq 8$. The fact that there is only a significant step in the MKL comparison, but not in the TSMTTSM efficiency, indicates that MKL switches to a significantly more efficient implementation for those dimensions, rather than the manual implementation is getting worse.

7.3 Tall & Skinny Matrix-Small Matrix Multiplication

A schematic sketch of the TSMM operation can be seen in fig. 7.4. For this operation, it holds that $\bar{m} \gg \bar{k}, \bar{n}$. Moreover, in BJDQR, $\bar{k} \geq \bar{n}$ and in SVQB it holds that $\bar{k} = \bar{n}$. However, the analysis will be carried out for the entire $\bar{k}, \bar{n}$

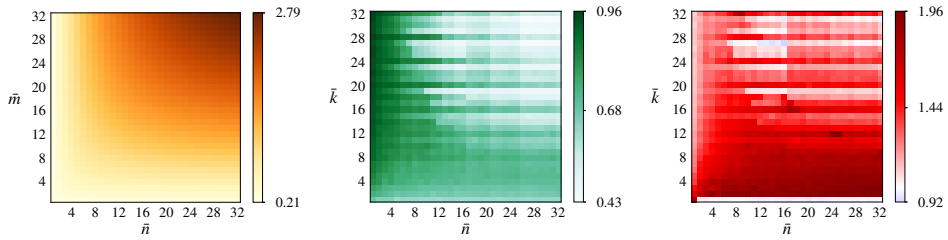
Listing 7.2 TSMM kernel for real double precision data with row-major block vector storage of A and C, column-major storage of B, $\alpha = 1$, $\beta = 1$, and $\bar{m}$ a multiple of two. Possibly required alignment/padding of data is omitted here for brevity.

```
1 double tmp[2];
2
3 #pragma omp parallel for private(tmp)
4 for (int row = 0; row < m-1; row+=2) {
5   for (int ccol = 0; ccol < n; ccol++) {
6     tmp[0] = c[(row+0)*n + ccol];
7     tmp[1] = c[(row+1)*n + ccol];
8
9 #pragma simd
10    for (int acol = 0; acol < k; acol++) {
11      tmp[0] += a[(row+0)*k + acol] * b[ccol*k + acol];
12      tmp[1] += a[(row+1)*k + acol] * b[ccol*k + acol];
13    }
14
15    c[(row+0)*n + ccol] = tmp[0];
16    c[(row+1)*n + ccol] = tmp[1];
17  }
18 }
```

range for the same reasons as in the TSMTTSM analysis in section 7.2. The small input matrix is not a block vector but possibly the result of a TSMTTSM operation (see above). The storage order of this matrix is irrelevant, as transposing it could be done at very low overhead. If $\beta = 0$, streaming stores to the result matrix should be employed to avoid a redundant transfer of this matrix ("write allocate").

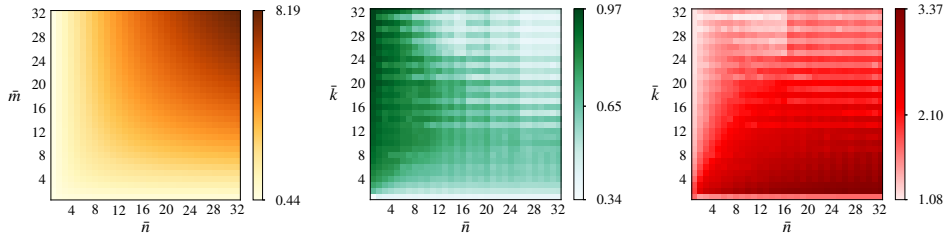
Implementation. Listing 7.2 shows the TSMM kernel using OpenMP for thread parallelism along the $\bar{m}$ dimension (lines 3 to 18) and "`#pragma simd`" for compiler-driven vectorization of the inner loop (lines 9 to 13). As can be seen in lines lines 4, 11 and 12, the kernel is two-way unrolled along the m (long) dimension to decrease the pressure on the load unit (only one load of `b[]` data is needed per inner loop iteration).

Performance analysis. Figure 7.5 shows the performance of TSMM in real arithmetic and for general α and β. Figure 7.5b demonstrates that for the entire range of $1 \leq \bar{n}, \bar{k} \leq 32$, the TSMM kernel reaches at least 43% and on average 68% of the Roofline performance. The kernel is most efficient in the range where either $\bar{k}$ or $\bar{n}$ is small, i.e., in the strongly bandwidth-limited regime. A comparison to Intel MKL is shown in fig. 7.5c. It can be seen that,



(a) Arithmetic intensity **(b)** Roofline efficiency **(c)** Speedup over Intel MKL

Figure 7.5: Arithmetic intensity, Roofline efficiency, and speedup over Intel MKL for the TSMM operation with real arithmetic and $\alpha \neq 1, \beta \neq 0$ on IVB.



(a) Arithmetic intensity **(b)** Roofline efficiency **(c)** Speedup over Intel MKL

Figure 7.6: Arithmetic intensity, Roofline efficiency, and speedup over Intel MKL for the TSMM operation with complex arithmetic and $\alpha = 1, \beta = 0$ on HSW.

especially in the strongly bandwidth-limited region (i.e., where the arithmetic intensity is low), TSMM outperforms the GEMM implementation of the MKL. MKL has an efficient GEMM implementation for $\bar{n} > 1$ and $\bar{k} = 1$, where it is roughly on par with the manual implementation. The maximum performance benefit of the manual implementation is $1.96\times$, while in the worst case ($\bar{n} = 16$, $\bar{k} = 27$) only 8% of performance are lost with respect to MKL. The average performance advantage of TSMM compared to MKL is $1.44\times$.

The performance analysis and findings can be transferred to the HSW architecture and complex arithmetic as shown in fig. 7.6. In this experiment, β has been set to zero which opens the possibility to employ streaming stores of the result matrix. The Roofline efficiency as shown in fig. 7.6b reveals similar behavior to the case shown in fig. 7.5b. Compared to MKL, the manual implementation achieves a maximum speedup of more than $3\times$ and $2\times$ on average. The fact that the relative speedup is smallest for $\bar{n} = 1$ and large $\bar{k}$

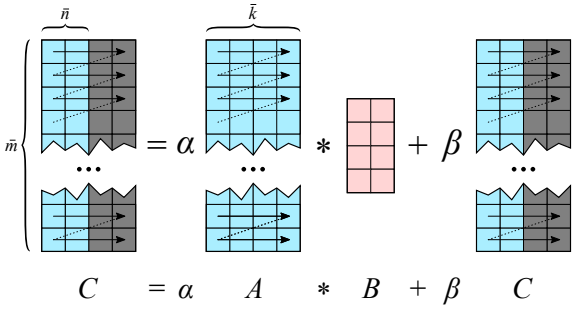


Figure 7.7: Schematic of the TSMM-inplace operation. Note that $\bar{n} < \bar{k}$ and C overlaps with A.

indicates that the MKL implementation does not employ streaming stores: the potential positive effect of them is smallest in exactly this region. Measurements of the transferred data volume using HPM verify this assumption. However, this analysis is omitted here as the intention of this work is not to reveal implementation drawbacks of existing BLAS implementations in great detail. Note that streaming stores need to be done to aligned addresses. For odd values of $\bar{n}$, this requires a careful implementation and can be achieved together with the aforementioned two-way loop unrolling along the m dimension.

7.4 In-Place Tall & Skinny Matrix-Small Matrix Multiplication

This kernel is similar to TSMM, with the difference that the output block vector C is aliased with the first input A. Figure 7.7 shows a schematic sketch of this operation. Again, it holds that $\bar{m} \gg \bar{k}, \bar{n}$ and the storage order of B is irrelevant, as it can be arbitrarily transformed at very little overhead. As C views some columns of A, it holds that $\bar{n} \leq \bar{k}$. The aliasing of in- and output data is prohibited in BLAS GEMM [8] which requires a temporary matrix of size $\bar{m} \times \bar{n}$. A manual implementation can avoid this by taking the aliasing into account. This is somehow related to the idea of "kernel fusion" (see chapter 8), as a copy and a GEMM kernel are fused into a single one.

Implementation. An implementation of this operation for real double precision data is shown in listing 7.3. Again, two-way outer loop unrolling is implemented to decrease the loads of `b[]` data (see lines 4, 11 and 12). Work sharing among threads is achieved along the $\bar{m}$ dimension (lines 3 to 20) and

Listing 7.3 TSMM-inplace kernel for real double precision data with row-major block vector storage of A and C, column-major storage of B, $\alpha = 1$, $\beta = 0$, and $\bar{m}$ a multiple of two. Note the absence of `c[]` which is aliased with `a[]`. Possibly required alignment/padding of data is omitted here for brevity.

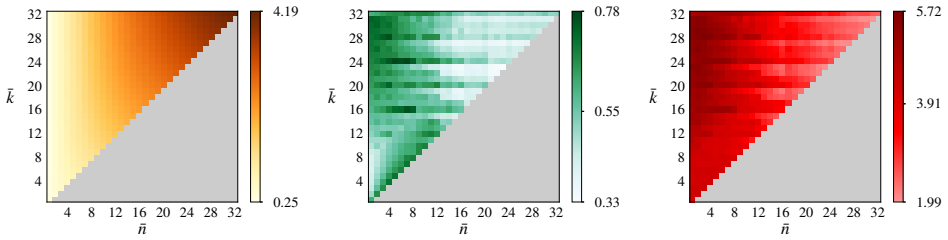
```
1 double tmp[2*n];
2
3 #pragma omp parallel for private(tmp)
4 for (int row = 0; row < m-1; row+=2) {
5   for (int ccol = 0; ccol < n; ccol++) {
6     tmp[0*n+ccol] = 0.;
7     tmp[1*n+ccol] = 0.;
8
9 #pragma simd
10    for (int acol = 0; acol < k; acol++) {
11      tmp[0*n+ccol] += a[(row+0)*k + acol] * b[ccol*k + acol];
12      tmp[1*n+ccol] += a[(row+1)*k + acol] * b[ccol*k + acol];
13    }
14  }
15
16  for (int ccol = 0; ccol < n; ccol++) {
17    a[(row+0)*k + ccol] = tmp[0*n+ccol];
18    a[(row+1)*k + ccol] = tmp[1*n+ccol];
19  }
20 }
```

the aliasing of in- and output data necessitates a temporary copy of two (due to unrolling) rows of C data as shown in line 1.

Performance analysis. Figure 7.8b reveals that the maximum TSMM-inplace efficiency (78%) is lower compared to the previously analyzed operations. Still, an average efficiency of 55% is achieved and the benefit over Intel MKL (fig. 7.8c) is between $2\times$ and $5.7\times$ with an average of $3.9\times$. The larger benefit over MKL compared to the non-aliased TSMM operation is easily explained by the avoidance of the block vector copy.

7.5 Summary

In this chapter, careful performance modeling and analysis revealed significant drawbacks for tall & skinny matrices of otherwise highly efficient GEMM implementations. This gives rise to specialized libraries like LIBXSMM [81], which features high-performance GEMM operations for small matrices and can be driven in "batch mode" for tall & skinny data. However, in its current



(a) Arithmetic intensity **(b)** Roofline efficiency **(c)** Speedup over Intel MKL

Figure 7.8: Arithmetic intensity, Roofline efficiency, and speedup over Intel MKL for the TSMM-inplace operation with real arithmetic and $\alpha \neq 1, \beta \neq 0$ on IVB. Data points where $\bar{k} \leq \bar{n}$ are grayed out.

state, it lacks some functionality which makes it not suitable for the kernels used in this work. For example, support for complex arithmetic is missing entirely. Moreover, the result matrix cannot be stored with streaming stores (which is needed for high TSMM performance if $\beta = 0$). Among other reasons, this motivates an implementation of T&S-GEMM kernels from scratch to yield high performance in data ranges where GEMM implementations fail. A comparison for relevant kernels and data ranges with Intel MKL indicates an average speedup in the range of $1.4 - 7\times$ and maximum speedup of $27\times$.

8 Custom Compute Kernels

The performance of bandwidth-limited algorithms can often be increased by reducing data transfers. Given the Roofline performance model from eq. (4.1), this corresponds to an increase of arithmetic intensity I and therefore an increase of the performance limit P^*. One way to achieve this is by reducing the data transfers within individual kernels, for example by reducing the factor α in SpMV operations (see section 4.2.2).

Beyond this, it is sometimes possible to reduce the total data transfers of an algorithm if the same data is used in different kernels by fusing those kernels. Kernel fusion is a very common optimization technique for bandwidth-limited algorithms. In this work, manual kernel fusion is employed. As the central and most complex single operation is often SpM(M)V, this operation gets augmented with one or more simpler operations which are executed in temporal proximity to SpM(M)V. To retain some degree of re-usability and flexibility, the level of augmentation can be changed incrementally. Needless to say, this approach cannot be considered "general," but it meets the specific requirements for the considered sparse linear algebra algorithms as introduced in chapter 3, and the implemented kernels can either be further augmented or serve as blueprints for further development.

All building blocks which are fused into tailored kernels in this work carry out computations on a row-wise basis. This makes it straightforward to fuse them, as it only requires to add more and more operations per row. However, there are certain peculiarities which require careful performance engineering if efficient tailored kernels should be implemented. As already mentioned in section 7.4, the TSMM-inplace operation, as it is required in BJDQR, can be seen as a fused kernel. Besides, kernel fusion has been used for the KPM and ChebFD solvers. The resulting implementations will be discussed in the following sections and performance models for the Topi-N_x-N_y-N_z test case will be developed. The same techniques can be applied to the simpler Lanczos algorithm (which is omitted here) and many more iterative schemes. On top of the performance data presented in this chapter, more experimental

Algorithm 4 "Vanilla" version of the KPM assuming that only BLAS-1 operations and a general SpMV kernel $\vec{y} \leftarrow H\vec{x}$ are available.

1: **for** $r = 0$ to $R - 1$ **do**
2: $\vec{v} \leftarrow$ rand()
3: $\vec{w} \leftarrow \vec{0}$
4: Initialization and computation of η_0, η_1
5: **for** $m = 1$ to $M/2$ **do**
6: $\vec{u} \leftarrow H\vec{v}$ ▷ `spmv()`, $V = [(v_{\text{el}} + v_{\text{idx}})n_{\text{nz}} + 2nv_{\text{el}}]$ B
7: $\vec{u} \leftarrow \vec{u} - b\vec{v}$ ▷ `axpy()`, $V = 3nv_{\text{el}}$ B
8: $\vec{w} \leftarrow -\vec{w}$ ▷ `scal()`, $V = 2nv_{\text{el}}$ B
9: $\vec{w} \leftarrow \vec{w} + 2a\vec{u}$ ▷ `axpy()`, $V = 3nv_{\text{el}}$ B
10: $\eta_{2m} \leftarrow \langle \vec{v}, \vec{v} \rangle$ ▷ `nrm2()`, $V = nv_{\text{el}}$ B
11: $\eta_{2m+1} \leftarrow \langle \vec{w}, \vec{v} \rangle$ ▷ `dot()`, $V = 2nv_{\text{el}}$ B
12: SWAP$(\vec{w}, \vec{v})$
13: **end for**
14: **end for** ▷ $V_{\text{KPM}} = RM/2[(v_{\text{el}} + v_{\text{idx}})n_{\text{nz}} + 13nv_{\text{el}}]$ B

performance results and comparison studies between standard and tailored implementations can be found in chapter 10.

8.1 KPM Operator

A direct implementation of the KPM scheme as briefly described in section 3.2 using only basic BLAS-1 calls and a basic SpMV operation $\vec{y} \leftarrow H\vec{x}$ results in the "vanilla" version of the KPM algorithm as illustrated in algorithm 4. The lack of scaling factors α and β in SpMV (as in $\vec{y} \leftarrow \alpha H\vec{x} + \beta\vec{y}$) and the missing possibility to apply a shift γ (as in $(H - \gamma\mathbb{1})\vec{x}$) necessitates the sequence of BLAS-1 operations in lines 7 to 9. The minimum data to be transferred is annotated after each function call. Here, v_{idx} denotes the bytes needed to store a single index. This is usually 4 bytes, but it may be larger for very large problems on a shared memory node or large-scale distributed memory computations without mixed index types (cf. section 9.3.1). The overall minimum data volume is annotated as V_{KPM} at the end of algorithm 4.

Kernel fusion. Assuming that a general and shifted SpMV operation is available or has been implemented by hand, some operations of algorithm 4 can be fused together, resulting in line 6 of algorithm 5. In a high-performance implementation, lines 6 to 8 can be further fused into a single kernel, which avoids the re-loading of the vectors $\vec{v}$ and $\vec{w}$. The overall data

Algorithm 5 "Fused" version of the KPM using a general and shifted SpMV operation which is fused with the `nrm2()` and `dot()` kernels (lines 6 to 8).

1: **for** $r = 0$ to $R - 1$ **do**
2: $\vec{v} \leftarrow$ rand()
3: $\vec{w} \leftarrow \vec{0}$
4: Initialization and computation of η_0, η_1
5: **for** $m = 1$ to $M/2$ **do**
6: $\vec{w} \leftarrow 2a(H - b\mathbb{1})\vec{v} - \vec{w}$
7: $\eta_{2m} \leftarrow \langle \vec{v}, \vec{v} \rangle$
8: $\eta_{2m+1} \leftarrow \langle \vec{w}, \vec{v} \rangle$
} ▷ $V = [(v_{\text{el}} + v_{\text{idx}})n_{\text{nz}} + 3nv_{\text{el}}]$ B
9: SWAP$(\vec{w}, \vec{v})$
10: **end for**
11: **end for** ▷ $V_{\text{KPM}} = RM/2[(v_{\text{el}} + v_{\text{idx}})n_{\text{nz}} + 3nv_{\text{el}}]$ B

volume V_{KPM} indicates that ten vector transfers can be saved by combining all kernels into a single fused "KPM operator."

Vector blocking. A further possible improvement to the KPM algorithm is to combine n_b of the R random initial vectors into a single block vector, which can be seen in algorithm 6. Here, and if not noted otherwise in all further occurrences, $n_b = R$. By this, the outer loop over r vanishes and the presence of a general, block vector-enabled, and shifted SpMMV is required, enabling streaming access to vector data and saving $(n_b - 1)$ loads of matrix data. In addition, block versions of the vector norm and dot product operations are needed. Similarly to the non-blocked version, lines 7 to 9 can be fused into a single kernel, resulting in the "fused+blocked" KPM operator. It should be noted that algorithms 4 to 6 are mathematically equivalent.

Increased arithmetic intensity. The primary goal of the presented alterations to the KPM algorithm from algorithm 4 to algorithm 6 is to increase the arithmetic intensity and with it the performance. The arithmetic intensity is increased by reducing the overall transferred data volume V_{KPM} while keeping the number of FLOPs constant at

$$F_{\text{KPM}} = RM/2\left[n_{\text{nz}}\left(f_{\text{ADD}} + f_{\text{MUL}}\right) + n\left(\lceil 7f_{\text{ADD}}/2 \rceil + \lceil 9f_{\text{MUL}}/2 \rceil\right)\right] \text{ FLOP}\,. \tag{8.1}$$

The KPM performance and algorithmic optimization will be analyzed by means of the Topi-N_x-N_y-N_z test case, where $n_{\text{nzr}} \approx 13$ (see table 2.2 on page 33 for details on this matrix). Matrix and vector values are of complex

Algorithm 6 "Fused+blocked" version of the KPM using a general and shifted SpMMV operation which is fused with block versions of the `nrm2()` and `dot()` kernels (lines 7 to 9, equal to algorithm 2 on page 39).

1: $\vec{V} := \vec{v}_{1,\dots,R}$ ▷ *Define block vector*
2: $\vec{W} := \vec{w}_{1,\dots,R}$ ▷ *Define block vector*
3: $\vec{V} \leftarrow$ rand()
4: $\vec{W} \leftarrow \vec{0}$
5: Initialization and computation of η_0, η_1
6: **for** $m = 1$ to $M/2$ **do**
7: $\quad \vec{W} \leftarrow 2a(H - b\mathbb{1})\vec{V} - \vec{W}$
8: $\quad \eta_{2m} \leftarrow \langle \vec{V}, \vec{V} \rangle$
9: $\quad \eta_{2m+1} \leftarrow \langle \vec{W}, \vec{V} \rangle$
} ▷ $V = [(v_{\text{el}} + v_{\text{idx}})n_{\text{nz}} + 3Rnv_{\text{el}}]$ B
10: $\quad$ SWAP$(\vec{W}, \vec{V})$
11: **end for** ▷ $V_{\text{KPM}} = M/2[(v_{\text{el}} + v_{\text{idx}})n_{\text{nz}} + 3Rnv_{\text{el}}]$ B

double precision which results in $v_{\text{el}} = 16$, $f_{\text{ADD}} = 2$ and $f_{\text{MUL}} = 6$. Moreover, v_{idx} is set to 4. The arithmetic intensity I of the three versions of this algorithm can be given as follows:

$$I^{\text{Topi}}_{\text{KPM}} = \frac{F_{\text{KPM}}}{V_{\text{KPM}}} \tag{8.2}$$

$$I^{\text{Topi}}_{\text{KPM,vanilla}} = \frac{138}{468}\frac{\text{FLOP}}{\text{B}} \approx 0.29\,\frac{\text{FLOP}}{\text{B}} \quad \text{for algorithm 4,} \tag{8.3}$$

$$I^{\text{Topi}}_{\text{KPM,fused}} = \frac{138}{308}\frac{\text{FLOP}}{\text{B}} \approx 0.45\,\frac{\text{FLOP}}{\text{B}} \quad \text{for algorithm 5,} \tag{8.4}$$

$$I^{\text{Topi}}_{\text{KPM,fused+blocked}} = \frac{138}{260/R + 48}\frac{\text{FLOP}}{\text{B}} \overset{R\to\infty}{\approx} 2.88\,\frac{\text{FLOP}}{\text{B}} \quad \text{for algorithm 6.} \tag{8.5}$$

Using a fused kernel as it is done in algorithm 5 increases the arithmetic intensity by approximately 50% (from 0.29 to 0.45 FLOP/B). Hence, a performance increase of up to 50% can be expected from kernel fusion if the optimized algorithm retains its original bottleneck, i.e., main memory bandwidth.

The arithmetic intensity of the fused+blocked version depends on the number of random vectors R. At the limit, i.e. for very large R, almost a factor of $10\times$ can be gained in arithmetic intensity compared to the baseline

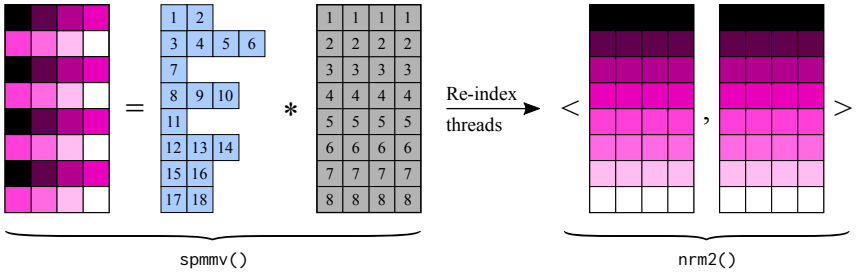


Figure 8.1: Fused "`spmmv()+nrm2()`" kernel on a GPU. Each thread computes one result block vector element and has a different color. The warp size is assumed to be eight.

implementation. The asymptotic intensity of 2.88 FLOP/B is already reasonably close to the inverse machine balance of some considered architectures such as IVB with $I_{\mathrm{M}} = 3.4$ FLOP/B, i.e., the code can no longer be considered strongly bandwidth-limited. This is especially true considering the unbalanced ratio of additions and multiplications in eq. (8.1), which reduces the applicable maximum floating point performance P_{max} of the Roofline model and causes a further shift of this kernel towards the compute bound region. Furthermore, vector blocking has some implications on the performance of operations like SpMMV as explained in chapter 6. This makes the appearance of new bottlenecks likely and a ten-fold speedup (as one could suggest from the arithmetic intensity) is hardly reached.

8.1.1 Implementation

As mentioned above, custom kernels like the fused or the fused+blocked KPM operator are usually no part of standard libraries. Consequently, they need to be implemented manually, which involves certain challenges and requires a careful approach to obtain high performance.

CPU implementation. The increased arithmetic intensity involves that the kernel's performance becomes sensitive to non-optimal in-core execution. In particular, efficient code vectorization gains importance as the code becomes more compute-limited. Especially for complex arithmetic (as it is the case here for the Topi-N_x-N_y-N_z matrix), compilers often fail to generate highly efficient code from a high level implementation. Thus, the CPU version of the KPM operator is implemented using compiler intrinsics to ensure efficient vectorization on the respective instruction set.

GPU implementation. Reduction operations are notoriously hard to implement on GPU architectures and prone to performance hazards [133]. Reductions inside a thread block can be achieved with the use of shared memory or shuffle instructions (which are available on NVIDIA GPUs since the Kepler architecture). The latter option usually leads to better performance, since it has lower overhead and shared memory is a scarce resource. Inter-block reductions require a synchronization of thread blocks, which cannot be avoided. The use of shuffle instructions for intra-block reductions requires careful thread scheduling, especially in the present case where the reduction operations `dot()` and `nrm2()` are fused with SpMMV. The thread mapping in a pure SpMMV kernel is done as shown in fig. 6.2, i.e., the threads of a block are mapped row first to block vectors. However, a transposed thread mapping is needed for the reduction, i.e., successive threads of a block must be mapped to successive rows of the block vectors. Figure 8.1 tries to visualize this procedure. Note that no data gets transposed, only the thread assignment gets altered.

8.1.2 Performance Modeling

The KPM performance using the Topi-100-100-40 test case is analyzed on various architectures. The physical domain results in a matrix with roughly 1.6×10^6 rows.

Shift of CPU bottleneck. Figure 8.2 clearly shows the shift of architectural bottleneck on IVB when blocking is employed. The performance of the fused variant is limited by main memory bandwidth and saturates at a performance which is reasonably close to the prediction from the Roofline model as given in eq. (4.1),

$$P^*_{\text{KPM,fused}} = \min\left(I^{\text{Topi}}_{\text{KPM,fused}} \times b_\text{S}; P_\text{max}\right) \tag{8.6}$$

$$= \min\left(0.45\frac{\text{FLOP}}{\text{B}} \times 52\,\frac{\text{GB}}{\text{s}}; 176\,\frac{\text{GFLOP}}{\text{s}}\right) \tag{8.7}$$

$$= 23.4\,\frac{\text{GFLOP}}{\text{s}}\,. \tag{8.8}$$

On the contrary, the fused and blocked kernel scales almost linearly within a socket. This suggests that the relevant architectural bottleneck is no longer the main memory bandwidth (which is a shared and non-scalable resource among the threads), but rather some scalable resource like the bandwidth of some cache level or the in-core execution.

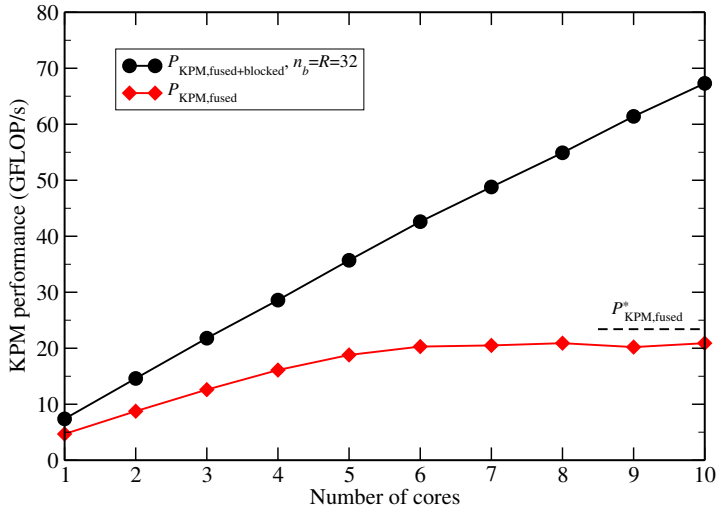


Figure 8.2: Intra-socket scaling performance of the "fused" KPM implementation (algorithm 5) and the "fused+blocked" KPM implementation (algorithm 6) on IVB. Figure adapted from [100].

CPU performance model for fused+blocked KPM. To identify the architectural bottleneck of the fused+blocked KPM, the Roofline model needs to be refined. For brevity, the subscript "fused+blocked" is omitted in the following.

A first refinement of the performance model is to include the data traffic overhead Ω into the bandwidth-limited performance prediction. In this case, Ω is defined as the ratio of actual, measured memory traffic and V_{KPM} as specified in algorithm 6. Dividing the bandwidth-limited Roofline prediction P^*_{KPM} with Ω gives a more realistic estimate as can be seen in fig. 8.3. However, especially for large n_b, the measured performance P_{KPM} still deviates significantly from P^*_{KPM}/Ω.

The KPM Roofline model can be further refined by considering a configuration which is decoupled from main memory,

$$P^*_{\text{KPM,refined}} = \min\left(P^*_{\text{KPM}}/\Omega; P_{\text{KPM,L3}}\right) , \tag{8.9}$$

where $P_{\text{KPM,L3}}$ is an upper performance limit for the L3 cache. In this case, it is obtained by performing the KPM using a test case which is reduced to a size where the entire working set fits into the L3 cache of IVB. The matrix should be kept as similar to the original (large) matrix as possible to have a good representation of the actual access patterns. Note that this does not necessarily

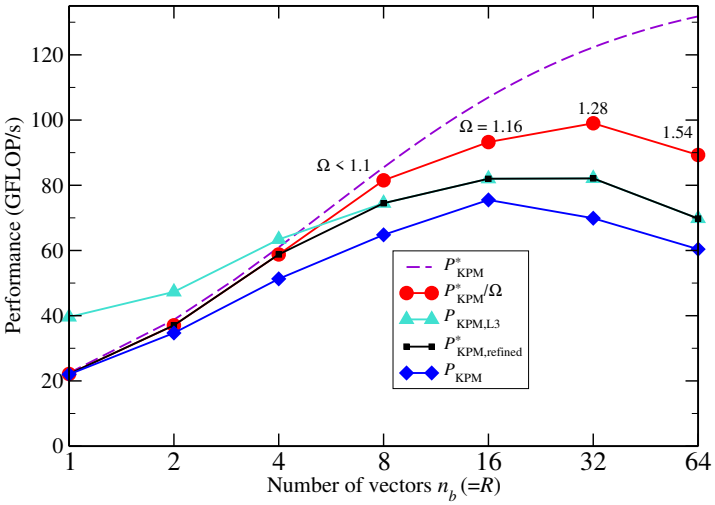


Figure 8.3: Refined Roofline model for the fused+blocked KPM operator on IVB with a Topi-100-100-40 test case. Note the logarithmic scale on the abscissa. Figure adapted from [100].

mean that the execution speed is limited by the L3 cache bandwidth, but the bottleneck can be anything except the main memory bandwidth.

The prediction of the original and refined Roofline models as well as the measured performance P_{KPM} for different values of n_b is shown in fig. 8.3. Clearly, for the evidently bandwidth-limited $n_b = 1$ case (where Ω is negligible), the original Roofline prediction P^*_{KPM} yields a precise performance prediction. At larger n_b, the kernel slowly decouples from main memory and at $n_b \geq 8$, main memory bandwidth is no longer the bottleneck, which is why the standard and Ω-correct Roofline models' predictions deviate more and more from the measured performance. However, the refined Roofline model, using the minimum of the Ω-corrected memory-limited and the memory-decoupled (L3) performance, delivers a reasonably precise prediction throughout the entire n_b range and never deviates from the measured performance by more than 15%.

GPU performance model. On a GPU, creating a similar refined Roofline model is substantially more difficult because one cannot use the GPU to full efficiency with a data set which fits into the small last level (L2) cache. However, a qualitative performance model can be established for K20m which explains the observed behavior and justifies the algorithmic optimizations.

For the blocked KPM kernel, which is similar to SpMMV, two caches of K20m are relevant. First, the L2 cache serves to alleviate the penalty of un-

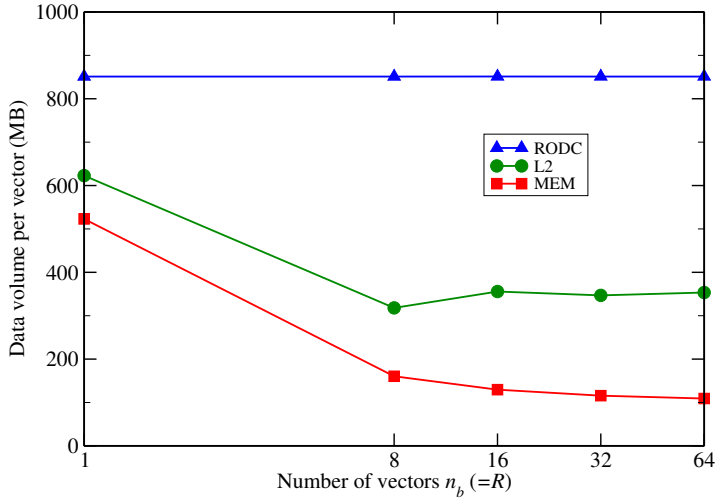


Figure 8.4: Transferred data volume per vector for a plain SpMMV kernel ($\vec{Y} \leftarrow H\vec{X}$) from different memory subsystem components on K20m using the Topi-100-100-40 matrix. Note the logarithmic scale on the abscissa. Figure adapted from [100].

structured accesses to the input block vector (just like a cache on a CPU does). Second, the 48KiB (per SMX) Read-Only Data Cache (RODC) (or texture cache) can be used for transparently broadcasting data to all threads of a warp due to its relaxed memory coalescing rules. In case of SpMMV, each matrix entry needs to be broadcast to a number of threads, which poses a very good usage scenario of the read-only data cache. Detailed information about the Kepler architecture and its caches can be found in the vendor's documentation [96, 97].

In a first step, the data volume from different memory subsystem components for a plain SpMMV kernel with varying block vector width n_b is analyzed in fig. 8.4. The fact that the RODC volume scales linearly with the block vector width reflects the fact that this cache is used for broadcasting matrix data to the vectors of a block. The accumulated data volume across all considered memory components decreases for increasing block vector width, which is due to the fact that the matrix needs to be loaded only once for all vectors in the block.

Figure 8.5 presents bandwidth measurements for three different kernel variants, going from a simple SpMMV kernel to the fully fused+blocked KPM operator. A first observation is that the plain (a), as well as the general and shifted (b) SpMMV achieve a main memory bandwidth of about 150 GB/s in the single vector case. This is in accordance with the attainable saturated

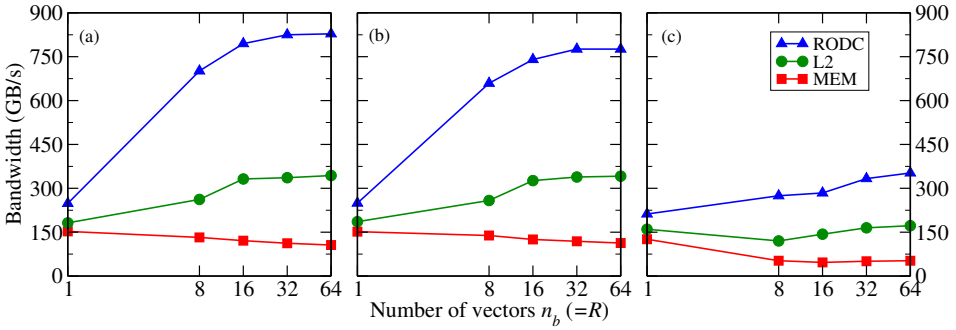


Figure 8.5: Memory bandwidth on K20m on different memory components for a Topi-100-100-40 matrix and (a), the plain SpMMV kernel ($\vec{Y} \leftarrow H\vec{X}$), (b), the general and shifted SpMMV kernel ($\vec{Y} \leftarrow \alpha(H - \gamma\mathbb{1})\vec{X} + \beta\vec{Y}$), and (c), the (b) kernel but fused with the dot product and norm computation, i.e., the fused+blocked KPM operator. Note the logarithmic scale on the abscissas. Figure adapted from [100].

peak memory bandwidth of this architecture of 152 GB/s (according to table 2.1 on page 22). The fact that the L2 and RODC bandwidths do not exceed the main memory bandwidth significantly in this case indicates that the kernel is limited by main memory bandwidth. With growing block vector width, a decrease of main memory bandwidth can be observed for (a) and (b), which is accompanied with increasing and eventually saturating L2 and RODC bandwidths. Thus, the relevant bottleneck shifts from main memory to cache bandwidth as the number of block vectors increases. This is a result which is similar to the findings made on IVB above.

For the fused+blocked kernel (c), a qualitatively similar behavior can be observed. However, all measured memory bandwidths are at a significantly lower level compared to (a) and (b). This is due to the issues with reduction operations mentioned in section 8.1.1. Note that the fully fused kernel, despite its performance issues, still yields a significantly higher performance than an implementation with separate computation of dot products and norms.

All observations coincide with findings that could be made using the NVIDIA Profiler's bottleneck analysis [134]: For all kernels, the main memory bandwidth is the reported bottleneck in the single vector case. For (a) and (b) this shifts to L2 bandwidth for larger block vector widths while instruction latency becomes the bottleneck for (c).

Algorithm 7 "Vanilla" version of the application of the polynomial filter in ChebFD.

1: **for** $k = 1$ to N_S **do** ▷ *First two recurrence steps*
2: $\vec{u}_k \leftarrow (\alpha H + \beta \mathbb{1})\vec{x}_k$ ▷ `spmv()`
3: $\vec{w}_k \leftarrow 2(\alpha H + \beta \mathbb{1})\vec{u}_k - \vec{x}_k$ ▷ `spmv()`
4: $\vec{x}_k \leftarrow g_0 c_0 \vec{x}_k + g_1 c_1 \vec{u}_k + g_2 c_2 \vec{w}_k$ ▷ `axpy() & scal()`
5: **end for**
6: **for** $n = 3$ to N_p **do** ▷ *Remaining recurrence steps*
7: **for** $k = 1$ to N_S **do**
8: swap$(\vec{w}_k, \vec{u}_k)$ ▷ swap pointers
9: $\vec{w}_k \leftarrow 2(\alpha H + \beta \mathbb{1})\vec{u}_k - \vec{w}_k$ ▷ `spmv()`
10: $\vec{x}_k \leftarrow \vec{x}_k + g_n c_n \vec{w}_k$ ▷ `axpy()`
11: **end for**
12: **end for**

Algorithm 8 "Fused+blocked" application of the polynomial filter in ChebFD. This is a blocked version of algorithm 7, omitting the first two recurrence steps for brevity. Other than that, this algorithm is equal to algorithm 3 on page 40. $\vec{U}, \vec{W}, \vec{X}$ are blocks of N_S vectors each. Lines 3 to 4 can be fused in a single iteration.

1: **for** $n = 3$ to N_p **do**
2: swap$(\vec{W}, \vec{U})$
3: $\vec{W} \leftarrow 2(\alpha H + \beta \mathbb{1})\vec{U} - \vec{W}$
4: $\vec{X} \leftarrow \vec{X} + g_n c_n \vec{W}$
} ▷ $V_{\text{ChebFD}} = [(v_{\text{el}} + v_{\text{idx}})n_{\text{nz}} + 5nN_s v_{\text{el}}]$ B
5: **end for**

8.2 ChebFD Operator

Besides KPM, ChebFD as described in section 3.3 is another important and computationally demanding eigenvalue algorithm which requires careful implementation in order to achieve high performance.

In the scope of this work, it is sufficient to say that the computationally most intensive step is the application of the polynomial filter, to be seen in algorithm 7. The two loops over k can each be re-formulated using block vectors containing n_b vectors each. In the simplest case $n_b = N_S$, but it may be wise to choose a smaller value of n_b if it yields higher performance. The polynomial degree N_p is usually in the range of thousands or larger for the relevant applications, i.e., the first two recurrence steps in algorithm 7 can be neglected in the performance analysis.

Applying the aforementioned blocking over N_s vectors, the core of the method can now be written as shown in algorithm 8. Assuming that this is done, the method requires a shifted and general SpMMV as well as a block-vector-enabled `axpy()` kernel. This is similar to the KPM operator analyzed in section 8.1. A crucial difference between the KPM and ChebFD algorithms is the absence of scalar product and norm calculations in ChebFD. However, kernel fusion can still be employed to increase the performance: lines 3 and 4 of algorithm 8 can be fused into a single kernel which avoids re-loading $\vec{W}$.

Increased arithmetic intensity. Regardless of algorithmic optimizations, the number of floating point operations in each iteration adds up to

$$F_{\text{ChebFD}} = N_s \left[n_{\text{nz}} \left(f_{\text{ADD}} + f_{\text{MUL}} \right) + n \left(3 f_{\text{ADD}} + 4 f_{\text{MUL}} \right) \right] \text{ FLOP} \,. \tag{8.10}$$

A possible algorithmic improvement is to compute the Chebyshev moments $\langle \vec{U}, \vec{U} \rangle$ and $\langle \vec{W}, \vec{U} \rangle$ in the iterative scheme similar to the KPM. This algorithm version, to be denoted as "ChebFD+KPM," allows to monitor the spectral density in the current search space and check, e.g., the choice of N_T during the ChebFD scheme [137]. As those additional operations are also fused into the ChebFD operator, this does not change the data traffic but merely increases the number of floating point iterations to

$$F_{\text{ChebFD+KPM}} = N_s \left[n_{\text{nz}} \left(f_{\text{ADD}} + f_{\text{MUL}} \right) + n \left(\lceil 9 f_{\text{ADD}} / 2 \rceil + \lceil 11 f_{\text{MUL}} / 2 \rceil \right) \right] \text{ B} \,. \tag{8.11}$$

Due to the aforementioned issues with fused reductions on Kepler GPUs, this should be only done if necessary, i.e., if the spectral density certainly needs to be monitored. However, CPU implementations do usually not entail this kind of problems which enables the inclusion of the moment computation at virtually no additional cost. This approach has been pursued in the CPU-only experiments in this work and previously published [137].

In conjunction with the minimum data volume as annotated in algorithm 8, the arithmetic intensity of the fused and blocked ChebFD algorithm for the Topi-N_x-N_y-N_z matrix ($n_{\text{nzr}} \approx 13$, complex double precision data) can be determined as

$$I^{\text{Topi}}_{\text{ChebFD}} = \frac{F_{\text{ChebFD}}}{V_{\text{ChebFD}}} = \frac{134}{260/N_S + 80} \frac{\text{FLOP}}{\text{B}} \overset{N_S \to \infty}{\approx} 1.67 \frac{\text{FLOP}}{\text{B}} \,, \tag{8.12}$$

and in case the Chebyshev moments get computed for monitoring as described above,

$$I^{\text{Topi}}_{\text{ChebFD+KPM}} = \frac{F_{\text{ChebFD+KPM}}}{V_{\text{ChebFD}}} = \frac{146}{260/N_S + 80}\,\frac{\text{FLOP}}{\text{B}} \overset{N_S\to\infty}{\approx} 1.83\,\frac{\text{FLOP}}{\text{B}}\,. \tag{8.13}$$

Implementation. For the ChebFD+KPM scheme, the implementation of the KPM operator as shown in section 8.1.1 can be used as a blueprint and fused with the additional scaled block vector addition. This fusion does not introduce new bottlenecks on the present compute architectures and a performance analysis yields qualitatively the same results as presented in section 8.1.2. For the standard ChebFD method (which does not include the computation of Chebyshev moments), the implementation and also the performance analysis simplify. Due to the absence of the reduction operation, the GPU performance and Roofline efficiency of the ChebFD operator exceed the (ChebFD+)KPM operator on Kepler GPUs (not shown in this work).

8.3 Summary

In this chapter, custom compute kernels for two of the investigated eigenvalue solvers, namely KPM and ChebFD, have been developed. A bottleneck analysis led to the identification of suitable optimization techniques: kernel fusion and vector blocking. The arithmetic intensity can be increased by those techniques which leads to a shift of hardware bottlenecks and higher performance of originally bandwidth-limited kernels. Similarly fused kernels can be employed for the Lanczos method which is not shown here. Additionally, the BJDQR implementation also uses fused kernels in the form of a shifted SpMMV and TSMM-inplace (see section 3.4 and the article by Röhrig-Zöllner et al. [142]). Performance results of the tailored compute kernels will be presented in chapter 10.

9 GHOST: General, Hybrid, and Optimized Sparse Toolkit

The General, Hybrid, and Optimized Sparse Toolkit (GHOST) is an open-source software library which comprises all programming efforts of this work. While a detailed description of its key features and design principles has previously been published [107], this chapter focuses on the basic software infrastructure and further aspects which give rise to a software library with a focus on high performance from the core to the peta-scale level. All kernels and optimization efforts described in the preceding chapters of this work are included in GHOST, and all presented performance data can be reproduced with it. The library, documentation material, and several example applications are available for download from the ESSEX website [64].

While section 9.1 provides details about data-parallel heterogeneous execution, section 9.2 describes the tasking capabilities of GHOST. In section 9.3, information about the basic data structures involved in GHOST is given. Section 9.4 describes the parallel SpMV implementation of GHOST, which is a premise for scalable sparse linear algebra software. Automatic code generation is a major contributor to high performance in GHOST. Section 9.5 describes the involved mechanisms and gives insight into how users can feed their knowledge about applications and algorithms into GHOST to generate highly efficient and tailored compute kernels. To give the reader an impression about the API of GHOST, section 9.6 provides a commented example application which uses GHOST to compute an SpMV operation. This is followed by a section covering various aspects about how to integrate GHOST into larger software stacks in section 9.7.

9.1 Data-Parallel Heterogeneous Execution

As this work's focus is on heterogeneous compute systems in the widest sense, GHOST has to be equipped with mechanisms which enable high performance on this kind of architectures. In the preceding chapters, the

performance properties of building blocks on different architectures have been analyzed separately. However, many current compute systems feature different kinds of architectures. For example, each node of the Piz Daint supercomputer (see section 2.2) is equipped with a host CPU and a GPU accelerator. To harness the full power of such heterogeneous nodes, it is inevitable to use all parts of it at high efficiency. Although the performance of the accelerator is higher than the performance of the host CPU by definition, and one could consider ignoring the CPU altogether, including both in the computation can lead to substantial performance gains (see section 10.1).

Parallelization paradigm. GHOST builds on the MPI+X parallelization paradigm, i.e., coarse-grained parallelism is done via MPI accompanied by fine-grained and device-specific parallelization strategies "X." Considering the architectures present in this work, "X" maps to CUDA on the GPU architectures K20, K20m, and P100. On the remaining architectures KNL, KNC, HSW, IVB, and SNB, "X" is a combination of thread-level parallelism with OpenMP and SIMD parallelism, the latter being achieved with compiler-friendly code and pragmas or explicit code vectorization using compiler intrinsics. If required, it is possible to omit the OpenMP layer and employ MPI-only parallelism on the CPU architectures. However, combining MPI with threading opens interesting possibilities, e.g., in terms of communication hiding and load balancing [139].

Data and task parallelism. There exist two general classes of parallel execution: data and task parallelism. In data-parallel execution, the workers execute identical tasks on different data sets concurrently. On the other hand, task-parallel execution involves workers processing completely independent and possibly even unrelated tasks at the same time. GHOST implements both on different levels: data parallelism is employed between MPI processes while inside a process, it is possible to have task-parallel execution (see section 9.2). Some of the aforementioned software libraries like, e.g., MAGMA and ViennaCL, employ task parallelism to map heterogeneous workloads to heterogeneous architectures. However, once kernel fusion (as done, e.g., in chapter 8) is implemented in sparse linear algebra algorithms, one often has a single monolithic kernel in which the majority of the work is done. Obviously, this impedes the use of task parallelism and hence, data parallelism is used in GHOST for heterogeneous execution.

Process distribution. GHOST detects the presence of compute devices at compile time and creates a custom build depending on that. If CUDA libraries are detected, the according kernels and features are compiled into GHOST on top of the standard CPU code and can be used transparently. Heterogeneous execution is transparent in a sense that a single executable can do the computational work on either the host CPU, an accelerating GPU, or both of them concurrently. Note that KNC and KNL are considered CPU architectures here, as they are operated in native and stand-alone mode (see section 2.1).

The execution mode of GHOST is described by a so-called "type," which can be determined automatically by GHOST or set by the user via a function call or an environment variable. Possible types are "CPU" and "GPU," the first one indicating that this GHOST instance is running on a standard multi-core CPU, a many-core CPU accelerator in native execution mode or a self-hosted many-core CPU like KNL. A GHOST instance of "GPU" type drives a GPU accelerator using CUDA.

On a typical CPU+GPU node containing several CPUs with C NUMA domains in total and G GPUs, the automatic type determination will cause the first process P0 to be of type "CPU" and cover all of the node's CPUs. Processes 1 to G are of type "GPU." For each process of this type, a CPU core is subtracted from P0's available resources for GPU management purposes. Any further added process causes a division of P0's CPU set into equally sized smaller sets. The recommended operating mode is to put $C + G$ processes on the node, resulting in one process per NUMA domain and one process per GPU.

Heterogeneous load balancing. Heterogeneous execution requires special attention towards load balancing. In the present data-parallel heterogeneous execution approach, the sparse system matrix (and associated vectors) are distributed among processes in a way that ensures balanced load of all processes (see section 9.3). If the computational power differs between the processes, this has to be considered in matrix/vector distribution. The number of matrix/vector rows on process p is denoted as n_p and can be computed as $n \times w_p$, where w_p is the process-specific weight and $\sum_p w_p = 1$. Analogously, the number of non-zero elements can be taken as the distribution criterion instead of the number of rows. In this case, the local number of rows n_p is chosen such that they contain $n_{\text{nz}} \times w_p$ non-zero elements. This is especially useful for matrices with irregular non-zero counts and can improve load balancing in such cases. GHOST can automatically determine w_p

Listing 9.1 Single-architecture SpMV benchmark application with SELL-32-1 and GHOST.

```
1 > ./spmvbench -m ML_Geer.mtx -c 10
2 [GHOST] PERFWARNING: The number of ranks (2) on this node is not
3 optimal! Suggested number: 3 (2 NUMA domains + 1 CUDA device)
4 [GHOST] PERFWARNING: There is 1 KNC in the set of active nodes
5 but only 0 are used!
6 7.94 GFLOP/s
7 > GHOST_TYPE=CPU mpirun -nopin -np 2 ./spmvbench -m ML_Geer.mtx
8 15.73 GFLOP/s
9 > GHOST_TYPE=GPU ./spmvbench -m ML_Geer.mtx
10 22.64 GFLOP/s
11 > export I_MPI_MIC=enable # enable MPI with MIC
12 > export I_MPI_POSTFIX=.mic # use the MIC-compatible executable
13 > mpirun -nopin -np 1 -host `hostname`-mic0 ./spmvbench -m \
14 ML_Geer.mtx
15 24.35 GFLOP/s
```

by means of micro-benchmarks. On top of that, a user can define custom weights at run-time.

Example: heterogeneous SpMV. In the following, the heterogeneous execution capabilities of GHOST are demonstrated based on a heterogeneous node as illustrated in fig. 2.1 on page 27, containing two IVB sockets as well as one K20m and one KNC. The presence of two different accelerators in a single node is rather atypical and hardly to be found in large-scale production clusters. However, this node is very well suited for demonstrating the mechanisms of heterogeneous execution in GHOST. If a complex node like this can be utilized efficiently and conveniently for a user, there is a high chance that the same holds for a "real world" node with a simpler setup.

The heterogeneous execution capabilities are demonstrated by means of the simple benchmark application "`spmvbench`," which measures the SELL-32-1 SpMV performance for a given sparse matrix using GHOST. The source code of this application is similar to the minimal example application described in section 9.6, equipped with parsing of command line parameters, sparse matrix read-in, a benchmark loop for SpMV and time measurements. Listing 9.1 shows the most simple execution mode, using only a single architecture at a time. In the first run shown in line 1, the execution should be done on a single IVB socket. As there are two sockets present with 20 cores in total, the number of cores used by GHOST has to be limited to ten in this case ("`-c 10`"). Assuming a best case arithmetic intensity of 1/6 FLOP/B according

to eq. (4.13), the effective memory bandwidth drawn in this experiment sums up to $7.94\,\text{GFLOP/s} \times 6\,\text{B/FLOP} \approx 48$ GB/s which is reasonably close to the maximum attainable bandwidth of IVB of 52 GB/s (see table 2.1 on page 22), resulting in a Roofline efficiency of $48/52 \approx 92\%$. The performance warnings issued by GHOST indicate an inefficient use of this node's resources and are omitted in subsequent experiments.

The preferred operation mode for a NUMA system (like the two present IVB sockets) is hybrid MPI+OpenMP, i.e., one MPI process should be running on each socket and thread parallelism using OpenMP should be used inside those processes. The respective invocation of the benchmark application is shown in line 7 of listing 9.1. To prevent GHOST from harnessing the K20m with the second process, the type is explicitly set to CPU with the "`GHOST_TYPE`" environment variable. The launched processes will be placed to the node's NUMA domains automatically by GHOST and automatic thread pinning by the MPI startup script should be disabled ("`-nopin`"). OpenMP is used in each MPI process and the 10 worker threads are automatically pinned to exclusive cores in each socket. The obtained performance of 15.73 GFLOP/s indicates a $1.98\times$, i.e., near-perfect, speedup from a single socket and very efficient usage of resources.

Without the need of re-compilation or any other adaption of the code, the benchmark can be executed on the K20m by setting the type to GPU as shown in line 9. The achieved performance of 22.64 GFLOP/s indicates a high Roofline efficiency of 89%.

As the KNC is operated in native mode, it has to be considered a node on its own with the hostname suffix "-mico." After some preparations, a single process can be started on the KNC as shown in lines 13 to 14. Here, the benchmark application yields a performance of 24.35 GFLOP/s, or 84% of Roofline efficiency.

The so far presented `spmvbench` performance numbers basically correlate to the numbers from section 5.4, with the difference that SELL-32-1 has been used here instead of SELL-16-1 (which does not increase the storage overhead for the well-behaved ML_Geer matrix).

If both IVB sockets and the K20m should be used, it is sufficient to start the executable with three processes as shown in line 1 of listing 9.2. GHOST prints information about each process' type. In this example, the process weights are determined automatically by means of a simple "UPDATE" microbenchmark which performs `a[i] = s*a[i]` for a large array of double numbers `a[]` and a scalar `s`. This gives a good impression of the relative bandwidth numbers of the involved hardware architectures, which are used for

Listing 9.2 Heterogeneous SpMV benchmark application with SELL-32-1 and GHOST.

```
1 > mpirun -nopin -np 3 ./spmvbench -m ML_Geer.mtx
2 [GHOST] PE0 INFO: Setting GHOST type to CPU.
3 [GHOST] PE1 INFO: Setting GHOST type to GPU.
4 [GHOST] PE2 INFO: Setting GHOST type to CPU.
5 [GHOST] PE0 INFO: Setting weight according to 39.9 GB/s
6 UPDATE bandwidth.
7 [GHOST] PE1 INFO: Setting weight according to 148.6 GB/s
8 UPDATE bandwidth.
9 [GHOST] PE2 INFO: Setting weight according to 37.4 GB/s
10 UPDATE bandwidth.
11 27.59 GFLOP/s
12 > mpirun -nopin -np 3 ./spmvbench -m ML_Geer.mtx -w 1:2.6
13 31.14 GFLOP/s
14 > # prepare machine file 'mf' for heterogeneous run with KNC
15 > echo -e "`hostname`:3\n`hostname`-mic0:1" > mf
16 > mpirun -nopin -np 4 -machine mf ./spmvbench -m \
17 ML_Geer.mtx -w 1:3.2:3.2
18 33.13 GFLOP/s
```

heterogeneous load balancing in GHOST. To get the relative process weight w_p, the measured bandwidth has to be divided by the accumulated bandwidth over all processes. Hence, $w_p \approx 0.17$ on each CPU process and $w_p \approx 0.66$ on the GPU process, or in other words, the GPU gets assigned approximately two thirds of the entire working set, while each CPU works on only one sixth of the global matrix. The heterogeneous performance sums up to 27.59 GFLOP/s. Note that this is a strong scaling scenario which includes communication over the slow PCIe bus. The parallel efficiency is $27.59/(15.73 + 22.64) \approx 72\%$. Although the automatically determined process weights usually lead to reasonably good load balancing, manual tuning is always an option. A short parameter scan reveals that a GPU/CPU weight ratio of 2.6 (instead of $0.66/0.17 \approx 3.9$), passed to the executable with the `-w` flag as demonstrated in line 12, significantly increases the heterogeneous performance to 31.14 GFLOP/s (which corresponds to 81% parallel efficiency).

Including the KNC and assigning manual process weights as shown in lines 16 to 17 yields a total node performance of 33.13 GFLOP/s, which is barely larger than the CPU+GPU performance. This indicates that the execution has reached the scalability limit. Note that due to software incompatibilities in the operating system concerning MPI communication between the KNC and host CPUs, the Transmission Control Protocol (TCP) has been used as the MPI fabric in this case, which may incur performance hazards.

9.2 Affinity-Aware Tasking

On top of the described inter-process data parallel approach, GHOST organizes work in tasks within a process. This is based on the necessity for sensible hardware affinity and the avoidance of resource conflicts, especially in view of complex node architectures and more and more complex and inherently asynchronous algorithms. GHOST uses hwloc [32] to get an abstract view of hierarchical topologies of current node architectures. Hwloc names hardware units that act as workers Processing Units (PUs). On the CPU architectures, a PU is equal to a hardware thread.

OpenMP compatibility. An important requirement for the tasking mechanism is to support OpenMP parallelism with pinned threads inside tasks. OpenMP is in wide use in the scientific community and the goal is that tasking can be incorporated without breaking existing code bases. This prerequisite rules out a number of existing task parallelism implementations like Intel Cilk [29] or Threading Building Blocks (TBB) [141], both of which warn in their user's manuals about using them together with OpenMP. Specifically, the Cilk developers warn that "the user must be extremely careful about controlling the number of threads to avoid oversubscription, especially when OpenMP is used at the innermost level" [66]. For TBB, the vendor warns that "oversubscription is possible because TBB and OpenMP run-time libraries create separate thread pools" and recommends "rewriting OpenMP code using Intel TBB if the use of TBB fits the design criteria for the application" [91].

Task design. Technically, a GHOST task is an arbitrary function in which OpenMP parallelism can be used without having to worry about resource conflicts or thread affinity. The task is implemented as a function that takes a single "`void *`" argument and returns a "`void *`." This is the most general and flexible type of function. The GHOST task data structure stores a pointer to this function and the argument.

A number of flags can be set to control the task's behavior. Most importantly, the user can control the hardware resources available to the task at creation time. Concretely, the number of threads and the NUMA domain on which these threads should be present can be defined. Normally, the definition of the NUMA domain is interpreted only as a hint, i.e., a task may be executed on a different domain if all other requirements for execution are met but the requested NUMA domain does not offer enough free resources. This behavior can be bypassed by setting a flag to force task execution on the

defined NUMA domain. Lastly, it is possible to pass a list of tasks which need to be finished before this task can be processed. This allows the implementation of simple dependency chains. It is possible to create nested tasks, i.e., tasks running inside other tasks. Whether or not a child task can occupy the parent's resources can be controlled by the parent's flags: by default, it is assumed that children are allowed to run on PUs belonging to their parent.

Minimal tasking. The simplest case of tasking is to create only a single task which contains basically all of the application's work. By this, no precautions concerning resource management (like manual pinning of threads) have to be taken by the user. Meaningful resource management is inevitable for obtaining high, explicable and reproducible performance. Moreover, performance analyses using HPM require controlled thread placement.

GHOST tasking can be disabled globally by setting an environment variable. By this, all enqueued tasks will be executed immediately and no resource management and thread pinning will be done. For sensible performance measurements, this has to be taken care of by the user.

Lifetime of a task-parallel application. Figure 9.1 visualizes the lifetime of a simple application using a single GHOST task to perform some work in parallel with pinned threads. The main application thread is shown on the left side and work happening inside the GHOST library on the right.

The first GHOST function called by the application is `ghost_init()`. Here, a number of *shepherd threads* get created with `pthread_create()`, which are later responsible for processing enqueued tasks. The number of such shepherd threads is arbitrary, but a reasonable guess is to set them to the number of PUs initially. More shepherd threads are spawned on demand. After creation, the shepherd threads immediately start waiting for a condition variable using `pthread_cond_wait()`.

When a user-created task gets enqueued using `ghost_task_enqueue()`, the condition on which the shepherd threads are waiting gets signaled which causes one of them being woken up. On the application side, the user's call to `ghost_task_enqueue()` returns immediately and the main thread continues processing. The woken shepherd thread decides whether the task's resource requirements can be met. If this is the case, an initial OpenMP parallel region gets created in which the worker threads get pinned to available PUs. Those PUs are marked as *busy* in the GHOST-internal PU map. After this is done, the user-defined task function is called by the shepherd thread.

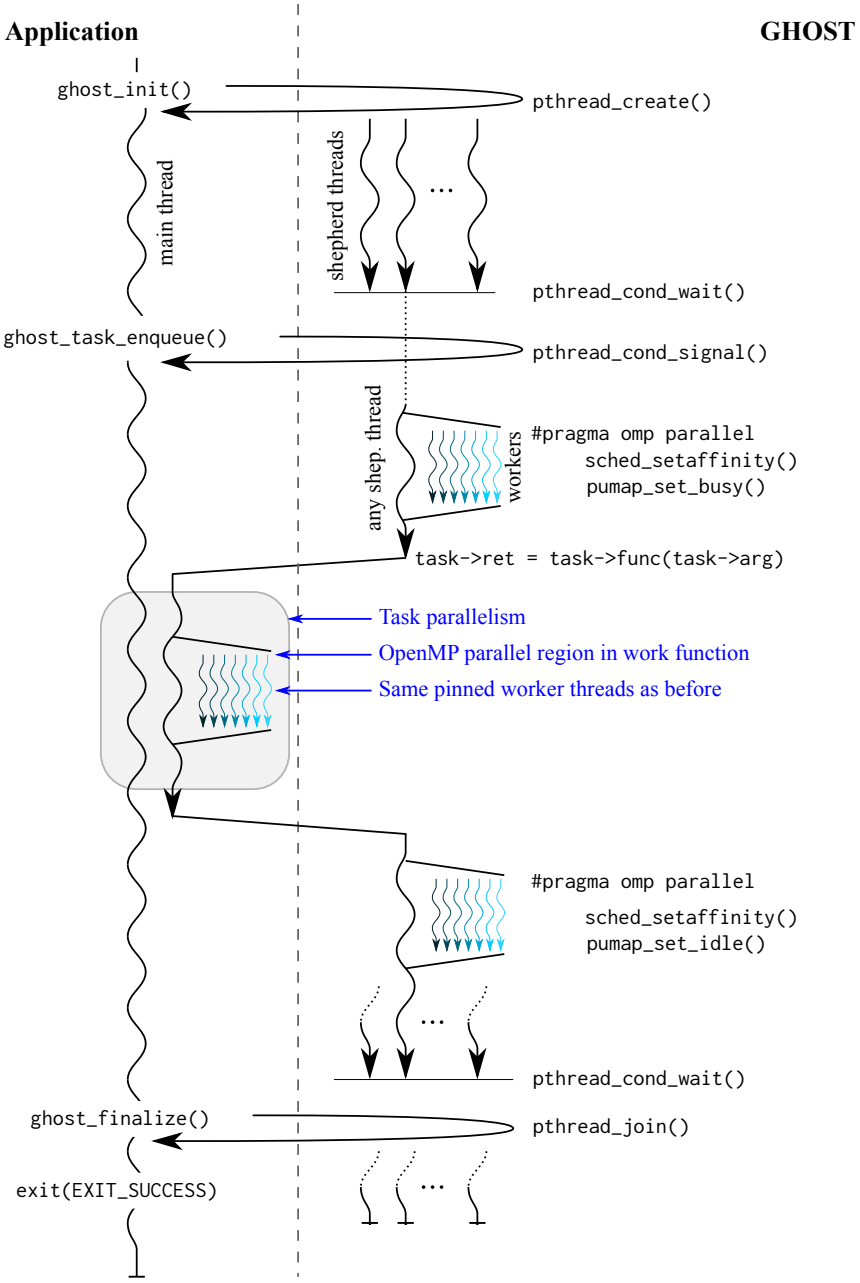


Figure 9.1: Lifetime of a simple application using a GHOST task (adapted from [107]).

At this point, the application code following `ghost_task_enqueue()` is executed in parallel to the user-defined task.

Within the task, OpenMP can be used and the threads will be pinned to the same PUs as in the initial OpenMP parallel region on the GHOST side, due to being the same threads as before. This persistence of OpenMP threads is not defined by the standard, but can be observed in the most important OpenMP implementations Intel OpenMP and GOMP. If the present OpenMP library does not have persistent threads, a fallback mechanism could confine the task threads to a CPU set by setting the shepherd thread's affinity mask. This is possible since "a child created via fork(2) inherits its parent's CPU affinity mask" [151]. This fallback mechanism works on a coarser granularity level and might be inappropriate for scenarios where a single task spans several NUMA domains. Hence, the aforementioned, non-standard-guaranteed but working mechanism is implemented in the current version of GHOST.

Once the task is finished, its PUs are marked as idle again, and its OpenMP threads are unpinned from their hardware resources. The shepherd thread starts waiting again on the condition of new tasks together with all other shepherd treads. At finalization, the shepherd threads are terminated.

9.3 Data Structures

GHOST is a building block library for sparse linear algebra algorithms. The central data structures involved in those are sparse matrices, as well as dense vectors and matrices. Detailed information about assembly, storage, and handling of either structure is provided in the following.

9.3.1 Sparse Matrices

The central data structure in many sparse linear algebra applications is the sparse system matrix. GHOST explicitly stores this matrix in the SELL-C-σ storage format, which was described in detail in section 5.2. At this point, it should be noted again that the widely used CRS storage format is just a special case of SELL-C-σ and hence, also supported by GHOST.

Distribution. The sparse matrix is the central data structure in GHOST and the inter-process work distribution is based on it. Parallel data structures are distributed on a per-row basis and the number of rows per process, which is reflected by the weights as described in section 9.1, depends on the sparse matrix and the present compute architectures.

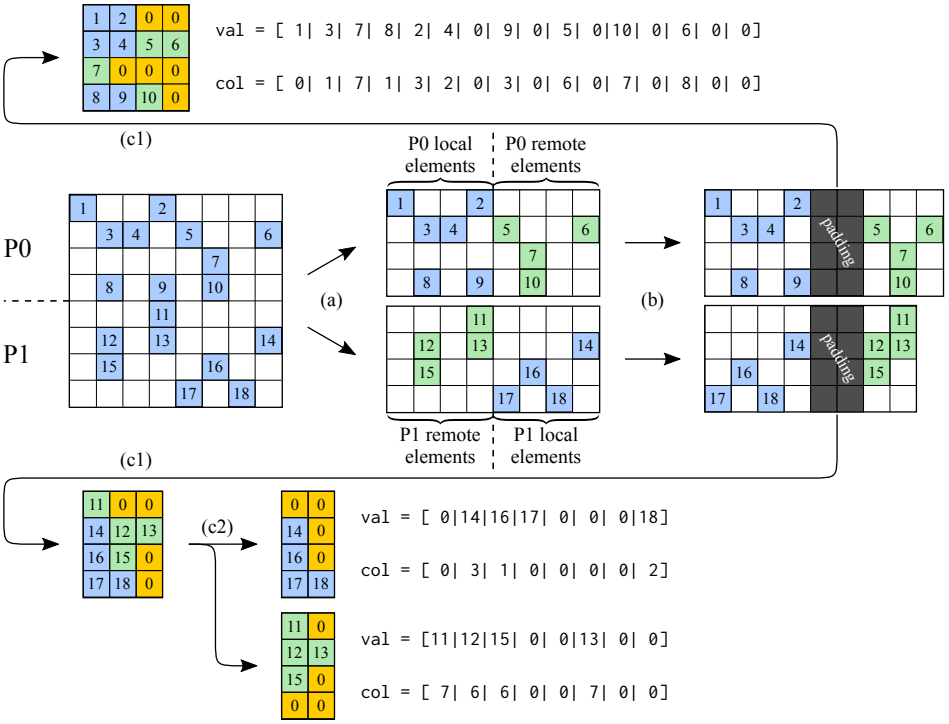


Figure 9.2: Distribution of a SELL-4-1 sparse matrix for two homogeneous processes and 6-element padding. Step (a) is splitting up the global matrix, (b) is the compression and padding of column indices, (c1) is the resulting combined process-local matrix and (c2) shows the splitting of it into a local and remote part.

Figure 9.2 illustrates the distribution of an example matrix on two homogeneous processes. Due to the homogeneity, the weight of each process is 0.5 and the number of rows per process is 4, as shown in step (a). The sparse matrix on each process consists of "local" and "remote" matrix elements. As vectors are distributed in the same, row-wise, way, this corresponds to SpMV input vector elements which are present on the same process versus those which are present on a remote process (see section 9.4). A matrix element is a local element if for its column index c it holds that "first row of this process" $\leq c \leq$ "last row of this process". Otherwise, the element is a remote element.

On each process, local column indices get shifted and remote column indices get compressed (step (b)). Shifting the local column indices causes a subtraction of "first row of this process" from each of them, i.e., the local part always starts with column index zero. Hence, the first possible entry of the

remote part is at column index "last row of this process." In addition, padding may be introduced, causing a further shift of the remote part "to the right." There are several potential reasons for the inclusion of padding, which are all related to the way remote matrix entries are handled in vectors, e.g., in distributed SpMV operations (see section 9.4). Note that padding does not contain the inclusion of new matrix elements, but only a shift of column indices. In the example in fig. 9.2, the first remote column index needs to be aligned to a multiple of six, which causes the introduction of two padding columns.

Compressing remote elements eliminates column gaps between them. By this, column indices of the process-local matrix can usually be stored as 4-byte integer values, even if the global matrix dimensions exceeds the value range of this type. In order for this to be possible, the sum of "local number of rows" + "padding" + "number of distinct remote indices" must not exceed the value range of 4-byte integers, which is a realistic constraint which was met in all experiments conducted throughout this work. Using 4- instead of 8-byte integers for column indices increases the arithmetic intensity of SpMV as shown in eq. (4.13) significantly and is an important factor for high performance.

Lastly, the process local matrices get created in step (c) with the modified column indices. On both processes, a combined matrix is created containing all local and remote entries (c1). On P1, this matrix is further split up into two separate matrices, one for the local and one for the remote elements (c2). This allows for an overlapping parallel SpMV computation (see section 9.4).

Pre-processing. GHOST supports several matrix pre-processing manipulations, all of which can be expressed by a permutation of matrix rows (and possibly columns). In this regard, a distinction between process-local and global permutations has to be made.

Global permutations can be employed, e.g., for communication reduction in parallel SpMV operations. For this purpose, GHOST can be linked against Zoltan [31] which implements hypergraph partitioning [38, 56] or PT-Scotch [39]. If such a global permutation should be employed, matrix information is passed to the third-party library which results in a permutation vector containing global indices. The GHOST matrix is then assembled in parallel on each process, according to the global permutation.

On top of a global permutation, one or more local permutations can be applied to the matrix. The most obvious local permutation is due to row sorting in the SELL-C-σ format. Further possibilities for local permutations are RCM re-ordering (for which the Sparse Matrix Pre-Processing (SpMP) library [156]

Listing 9.3 Assembly of a diagonal matrix using a callback function in GHOST.

```
int diag(
  ghost_gidx row,  // in: global matrix row
  ghost_lidx *nnz, // out: number of non-zero entries in row
  ghost_gidx *col, // out: column indices of non-zero entries
  void *val,       // out: values of the non-zero entries
  void *arg        // in/out: arbitrary
  ) {
    *nnz = 1;      // the row contains a single non-zero entry
    col[0] = row;  // the only entry is on the diagonal
    if (arg == NULL) return 1;           // error
    ((double *)val)[0] = *(double *)arg; // use *arg as value
    return 0;      // success
}
```

can be linked against GHOST) or multi-coloring using the ColPack [72] library.

Further local or global permutation methods like the recently proposed distributed-memory parallel RCM re-ordering [15] can be easily implemented in the existing software structure.

Assembly by callback function. The preferred method of matrix assembly is by means of a user-provided callback function. This function gets called by GHOST for every matrix row, and potentially by many MPI processes and threads in parallel. Listing 9.3 shows the implementation of such a callback function for the simple case of a diagonal matrix with a constant value, which is provided by the user in the last argument. The user has to specify the matrix dimensions, data type and an upper bound for the rows' non-zero count before matrix assembly. Note that in parallel runs, the matrix row and column indices need to be in the *global* scope (hence the `ghost_gidx` index type). In addition to the current row, an additional "`void *`" argument can be used to the user's needs. The callback function returns the number of non-zeros of the row `nnz`, and their according column indices `col[]` and values `val[]`.

Assembly from files. Besides callback functions, sparse matrices can be provided as files. GHOST supports matrix read-in from either the Matrix Market file format [124] or a custom binary CRS-like format. In the latter case, the matrix can be read in parallel from multiple MPI processes. Sparse matrices can also be stored in files of either format in a parallel way. Due to the

inherent scalability and performance hazards of file I/O, callback functions as described above should be used whenever possible.

Conversion from CRS. If GHOST should be used together with other software, it sometimes is required to create a SELL-C-σ matrix from an existing CRS matrix. For instance, constructing a SELL-32-128 representation of the ML_Geer test case (present in CRS) on two IVB sockets with one MPI process each consumes as much time as 48 SpMV operations (including row sorting, permutation, communication data setup, and copying of data). Note that setting up the communication data structures, which in some way has to be done by all parallel compute libraries regardless of the storage format, accounts for 78% of the entire construction time, which reduces the initial, pure CRS-to-SELL-C-σ conversion overhead to an equivalent of 11 SpMV operations. In realistic, large-scale scenarios (see, e.g., section 10.3), this SpMV count is easily reached and the overhead of SELL-C-σ matrix construction is quickly amortized. If the values of an existing SELL-C-σ matrix need to be updated from a present CRS matrix and both matrices have the same nonzero pattern, this takes as long as 2 SpMV operations. This is plausible as the CRS matrix needs to be read, the SELL-C-σ matrix needs to be written, and the SpMV traffic for this (high-n_{nzr}) matrix is dominated by matrix data.

9.3.2 Dense Matrices

Dense matrices can be of various kinds, which are all represented by a single type in GHOST. The simplest case of a dense matrix is a dense vector, where the column count is simply set to one. For any other kind of dense matrices, GHOST fully supports row- and column-major storage. For parallel runs, the sparse system matrix is distributed among the processes as shown in fig. 9.2, and with it the (block) vectors. Dense matrices can also be specified to be stored redundantly on each process, which only makes sense if they do not scale with the global matrix dimension. A common source for this kind of matrices is the result of an inner product of block vectors (TSMTTSM).

Views. GHOST supports dense matrix views, i.e., dense matrices which only store metadata but view actual data of another dense matrix. This allows, e.g., to perform an operation on a subset of a block of vectors, which is a frequent pattern in sparse linear algebra. For instance, converged eigenvalues in iterative eigenvalue solvers like BJDQR can be locked by creating a view of the full block vector except the vector corresponding to the converged

eigenvalue. This allows to ignore the converged eigenvector in subsequent iterations without the need of copying data.

In case of row-major block vectors, a view is called "compact" if it views subsequent vectors of the block, and "scattered" otherwise. Scattered views require a bit mask indicating the viewed vectors. Additionally, scattered views are likely to hinder efficient vectorization. Hence, they should be avoided whenever possible. Note that even compact views may carry performance hazards compared to non-views. For example, if a row-major view contains one cache line of data per row, and the viewed dense matrix has twice the number of columns, adjacent cache line prefetching (as it is present in many current CPUs) would cause the non-viewed data to be transferred even if only the elements of the view are requested in a computation.

Permutations. Potential matrix permutations as listed in section 9.3.1 necessitate permuted dense matrices. GHOST supports permuted dense matrix assembly (using an existing permutation) and features functions for permuting dense matrices back and forth, locally and globally.

Heterogeneous processing. If a GHOST run includes accelerators, dense matrix data will be exclusively stored on the host or accelerator, depending on the process type. This behavior can be changed by the user, i.e., the user can specify the "data location" to store redundant copies of dense matrix data on both the device and accelerator. Functions for up- and download are available to the user. In addition to the data location, it is also possible to control the "compute location" for data which is stored redundantly. This is especially useful for heterogeneous runs, where the user could prefer to perform computations with small dense matrices on the host instead of the accelerator.

9.4 Parallel SpMV

A scalable sparse linear algebra library requires distributed-memory parallel implementations of the numerical kernels. For many of the present building blocks this is easy to achieve, as no or only trivial inter-process communication is required (see section 3.5). For Block-BLAS-1 operations, only dot product and norm computations require a global reduction of the partial results after the process-local operation. Considering the T&S-GEMM operations, merely TSMTTSM requires a global reduction of the result matrix (which is obvious as it resembles an inner product of block vectors). The only

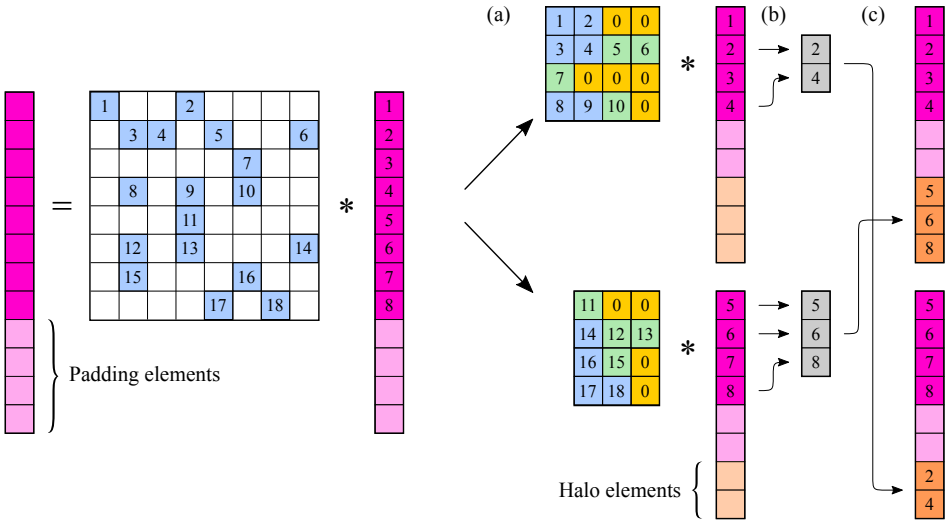


Figure 9.3: Communication and computation scheme of the "vector mode" distributed SpMV operation. Step (a) is the distribution of the matrix, (b) involves the assembly of communication buffers by locally gathering elements to be transferred and (c) is the communication via MPI.

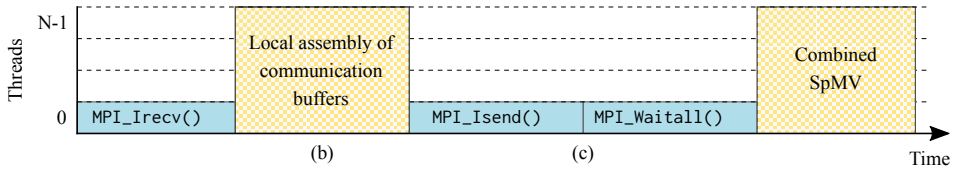


Figure 9.4: Timeline of the "vector mode" distributed SpMV operation. The annotating letters correspond to fig. 9.3. Checked panels indicate process-local operations and solid panels indicate communication.

presented building block which requires more elaborate communication is SpM(M)V, and obviously all custom kernels (see chapter 8) that it is part of.

All previous discussion about SpMV was limited to the single-node/-device level. In this section, the implementation of this kernels in distributed-memory systems is explained. All involved data structures are distributed among MPI ranks. This necessitates communication of input vector elements. Obviously, the parallel SpMV performance depends on the process-local performance, as well as on the communication effort and performance. Several execution modes for parallel SpMV are implemented in GHOST, basically based on the suggestions from Schubert et al. [152], which will be briefly described in the following.

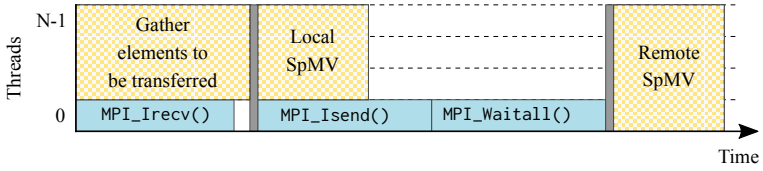


Figure 9.5: Timeline of the "overlap" distributed SpMV operation. Checked panels indicate process-local operations and solid panels indicate communication.

Vector mode SpMV. This is the default parallel SpMV mode which is illustrated in fig. 9.3. The sparse matrix, its distribution, and the number of vector padding elements are the same as in fig. 9.2. In this mode, the local and remote part of the matrix are stored in a combined manner, i.e., as a single process-local matrix as shown in step (c1) of fig. 9.2. In step (a) of fig. 9.3, the matrix gets distributed as previously described. The vectors get distributed using the same number of rows on each process. The right hand side vector $\vec{x}$ needs to have additional storage for storing the halo elements. Hence, matrix information needs to be known when a vector gets allocated that should be used in a parallel SpMV operation. Step (b) is the first part of the communication, where each process gathers $\vec{x}$ elements which are required by another process and stores them in a communication buffer. This buffer is sent to the target process using MPI in step (c) where it is attached to the local vector data in the space reserved for the halo elements. Once the communication is done, each process has a valid right hand side vector corresponding to the process-local sparse matrix and can perform the SpMV using them.

Figure 9.4 shows a timeline of the same operation. Note that the "widths" of the contributions are not to scale. Most importantly, the MPI communication does not overlap with computation and there is only a single, combined SpMV operation executed.

Overlap mode SpMV. If the communication time in a parallel SpMV operation is significant, it is sometimes possible to overlap (part of) it with computation time. Obviously, for the combined process-local SpMV operation, all $\vec{x}$ elements need to be present, i.e., the communication needs to be finished beforehand. Hence, the SpMV operation must be split into a purely local and a purely remote part. This is shown for process P1 in step (c2) of fig. 9.2. In this case, the local SpMV does not require remote vector elements, and it can be overlapped with the transfer of vector data. Once the remote vector data is present, the remote partial SpMV can be done. The partial results of both

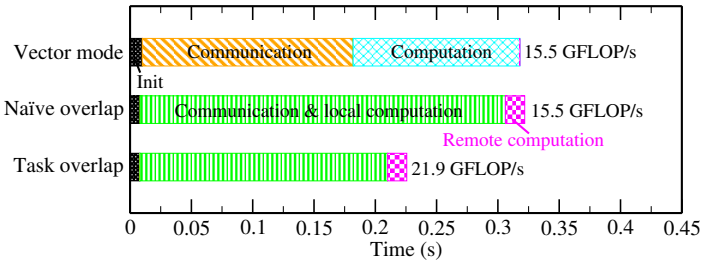


Figure 9.6: Runtime breakdown of different parallel SpMV variants on 4 Emmy nodes with one process per IVB* socket using the Spin-30 test case stored in SELL-16-64 and `DYNAMIC,50` OpenMP scheduling.

sub-operations need to be combined into the final result vector $\vec{y}$. Figure 9.5 shows a schematic timeline for this operation.

The overlap of local SpMV and communication can be achieved explicitly by using a dedicated thread for the MPI communication or by relying on non-blocking MPI calls (to be called "naïve overlap"). Non-blocking MPI calls are not guaranteed to be processed asynchronously. This depends on the MPI implementation and several other factors like message size. Previous research has shown that in fact, non-blocking MPI calls are often processed synchronously [55, 180]. This is why GHOST implements both a version relying on asynchronous MPI, as well as a version with explicit overlap using GHOST tasks.

The overlap SpMV mode bears a potential performance hazard due to multiple transfers of the result vector $\vec{y}$. In case of the simple $\vec{y} \leftarrow H\vec{x}$ operation, the split computation results in the two sub-operations $\vec{y}_{\text{local}} = H_{\text{local}}\vec{x}$ and $\vec{y} = \vec{y}_{\text{local}} + H_{\text{remote}}\vec{x}$, requiring three transfers of $\vec{y}$ compared to only one for the vector mode SpMV. For the $\vec{y} \leftarrow \vec{y} + H\vec{x}$ case, four instead of two transfers are required. In either case, the overhead may be significant and overcompensate performance gains by the communication overlap, especially for large data sets, where $\vec{y}$ does not fit into the cache and is written back to memory after the local operation and re-read in the remote operation. In general, one cannot determine an optimal SpMV mode for all sparse matrices, as it depends on many factors like matrix structure (and hence, communication effort), network speed, data set size, and cache sizes.

Comparison of SpMV modes. One test case where overlapping communication and computation pays off is the communication-intensive Spin-N_{Up} matrix. Figure 9.6 shows the runtime breakdown for the three aforementioned parallel SpMV modes on 4 Emmy nodes with the Spin-30 matrix.

Clearly, explicit overlap using GHOST tasks yields the smallest runtime and best performance. The naïve overlap mode, which relies on asynchronous MPI calls, yields the same performance as vector mode. Obviously, the present MPI library (Intel MPI 5.1.3 according to section 2.3) either does not handle non-blocking calls asynchronously, or it does it in a very inefficient manner.

As mentioned above, overlapping communication with computation in SpMV only pays off if the negative impact from multiple $\vec{y}$ transfers is overcompensated by the gains from communication hiding. This is only the case if the communication effort is large enough. For example, executing a parallel SpMV for the Spin-26 test case (similar to the above, but smaller) on a single Emmy node (i.e., 2 IVB* sockets) yields a performance of 7.5 GFLOP/s with vector mode and 6.6 GFLOP/s with task overlap. Hence, there is no "best" parallel SpMV mode and special care has to be taken when selecting it.

9.5 Code Generation

Code generation is a crucial ingredient for high performance of many of the discussed building blocks. In GHOST, it is done in an automatic and user-guided way. Code generation can help to generate compiler-friendly code. Most importantly, GHOST generates numerical kernels with hard-coded small dimensions, loop trip counts, if-conditions or explicit unrolling. To that end, there exist "template" files for several building blocks out of which different specializations are generated at compile time.

Kernel lookup and selection. Algorithm 9 shows the principle of kernel selection in GHOST, assuming that a SELL-32-σ SpMMV operation with a block vector width of four, properly aligned data, and complex double precision data should be executed on an AVX2-capable machine.

All generated variants of a certain kernel are put into a map, which gets created at the first call of this kernel (line 2). The map assembly code is generated at compile time and injected in the source file. The key of the map is a combination of all kernel properties. In the shown example, it is a quintuplet containing the chunk height C of the SELL-C-σ matrix, the number of vectors in the block, the implementation variant, the type of the matrix and block vector data, and the alignment of block vector data. Each of those key components can attain several values. The chunk height C could be fixed to 32 in the SpMMV kernel, or it could be arbitrary (and similarly for the block vector size). The implementation of the kernel could be conducted using

Algorithm 9 SpMMV kernel selection for aligned complex double precision data, SELL-32-σ matrix storage, and a block vector width of 4 on an AVX2-capable machine.

```
 1: if Map is not initialized then
 2:     Include generated file for map assembly
 3: end if
 4:
 5: Kernel ← NULL;
 6: Optimal ← true;
 7:
 8: for each SELL chunk height C in {32,arbitrary} do
 9:     for each Vector block size B in {4, arbitrary} do
10:         for each Implementation I in {AVX2, AVX, SSE, Plain} do
11:             for each Data type D in {double complex, arbitrary} do
12:                 for each Vector alignment A in {aligned, unaligned} do
13:                     Kernel ← map{C,B,I,D,A}
14:                     if Kernel then
15:                         go to line 24;
16:                     end if
17:                     Optimal ← false;
18:                 end for
19:             end for
20:         end for
21:     end for
22: end for
23:
24: if Kernel then
25:     if Optimal == false then
26:         Echo "Performance warning: calling possibly non-optimal kernel!";
27:     end if
28:     Call kernel;
29: else
30:     Echo "Performance warning: calling possibly fallback kernel!";
31:     Call fallback kernel;
32: end if
```

AVX2, AVX, or SSE compiler intrinsics or plain C code. The kernel could be specialized for complex double precision data, or the data type could be arbitrary in the kernel. And lastly, the kernel could assume (and exploit) the alignment of block vector data, or it could make no assumptions about that.

For each key component, the possible values are iterated, going from the most specific and appropriate to more general configurations. In the above example, the first check whether a kernel exists in line 13 would try to find an explicitly AVX2-vectorized SELL-32 SpMMV kernel assuming aligned block vectors with width 4 and complex double precision values. If no such kernel is found, the alignment requirement is dropped and a kernel is sought which meets all of the above criteria except aligned block vectors. This procedure is done iteratively until a suitable kernel is found. Once this is the case, the loop nest is left and the kernel is executed in line 28. If the kernels is not entirely specialized, a performance warning is printed in line 26. In case no generated kernel can be found, GHOST issues a performance warning and falls back to the highly general fallback kernel (lines 30 and 31). Note that the order of the loops in lines 8 to 12 can vary arbitrarily, and the optimal order is hard to determine.

User guidance. If one or more specializations are missing, the user may consider re-configuring and -building GHOST passing the respective hints to generate specialized kernels. To help with that, GHOST prints a warning at the end of the run with an appropriate command to generate a custom GHOST build for the present application. For example, if GHOST detects missing kernels for SELL-32-σ SpMV and SELL-1-σ SpMMV with a block vector width of 4, GHOST advises to re-configure the build specifying the string "`-DGHOST_GEN_SPMMV=32,1;1,4`" on the command line.

9.6 An Example Application Using GHOST

Listing 9.4 shows a small example application using GHOST. The application computes an SpMV of an identity matrix with a random vector and prints the matrix, as well as the in- and output vectors. Note that tasking as explained in section 9.2 is not employed here, i.e., GHOST does not take care of thread placement and affinity. The different stages of the application will be briefly explained in the following.

First, some variables needed in the application are declared in lines 2 to 5. The variables `zero` and `one` are required for initializing data. The `char` pointers are later used for storing strings of the involved sparse and dense matrices.

Listing 9.4 A minimal GHOST application to compute an SpMV of an $n \times n$ identity matrix with a random vector.

```
int main(int argc, char **argv) {
  ghost_sparsemat *H;
  ghost_densemat *y, *x;
  double zero = 0., one = 1.;
  char *Hstr, *xstr, *ystr;

  // initialize GHOST
  ghost_init(argc,argv);
  // use default traits for dense and sparse matrices
  ghost_sparsemat_traits mtraits = GHOST_SPARSEMAT_TRAITS_INITIALIZER;
  ghost_densemat_traits vtraits = GHOST_DENSEMAT_TRAITS_INITIALIZER;
  // sparse matrix and vectors are of real double precision data
  mtraits.datatype = (ghost_datatype)(GHOST_DT_REAL|GHOST_DT_DOUBLE);
  vtraits.datatype = (ghost_datatype)(GHOST_DT_REAL|GHOST_DT_DOUBLE);
  // save uncompressed column indices for printing matrix
  mtraits.flags = GHOST_SPARSEMAT_SAVE_ORIG_COLS;
  // create sparse matrix source
  ghost_sparsemat_src_rowfunc matsrc =
    GHOST_SPARSEMAT_SRC_ROWFUNC_INITIALIZER;
  matsrc.func = diag;   // callback function for matrix creation
  matsrc.arg = &one;    // an identity matrix should be created
  matsrc.gnrows = n;    // global number of rows (defined elsewhere)
  matsrc.maxrowlen = 1; // maximum non-zero count per row
  // create sparse matrix H and initialize from row-wise source function
  ghost_sparsemat_create(&H, NULL, &mtraits, 1);
  ghost_sparsemat_init_rowfunc(H,&matsrc,MPI_COMM_WORLD,0.);
  // create and initialize input vector x and output vector y
  ghost_densemat_create(&x, ghost_context_max_map(H->context), vtraits);
  ghost_densemat_create(&y, ghost_context_max_map(H->context), vtraits);
  ghost_densemat_init_rand(x);      // x = random
  ghost_densemat_init_val(y,&zero); // y = 0
  // compute y = H*x
  ghost_spmv(y,H,x,GHOST_SPMV_OPTS_INITIALIZER);
  // print y, H and x
  ghost_sparsemat_string(&Hstr,H,1);
  ghost_densemat_string(&xstr,x);
  ghost_densemat_string(&ystr,y);
  printf("%s\n=\n%s\n*\n%s\n",ystr,Hstr,xstr);
  // clean up: free strings and destroy sparse and dense matrices
  free(Hstr); free(xstr); free(ystr);
  ghost_sparsemat_destroy(H);
  ghost_densemat_destroy(x);
  ghost_densemat_destroy(y);
  // finalize GHOST
  ghost_finalize();

  return 0;
}
```

Dense vectors are represented as dense matrices of type `ghost_densemat`, and the sparse matrix H is stored as a `ghost_sparsemat`.

In most cases, the first GHOST function called by an application is `ghost_init()` as done in line 8. This function includes the complete set-up phase of GHOST, including the gathering of hardware information using hwloc and the determination of process types based on this (see section 9.1). If MPI is not yet initialized, it is done in this function.

The next step in lines 9 to 16 is to set up traits for the involved data structures, i.e. sparse (`mtraits`) and dense (`vtraits`) matrices. The initial traits contain reasonable default values, which can be altered by the user. This includes setting the data type of the matrix and vector values as done in lines 13 and 14. In line 16 of the given example, a flag is passed to the `mtraits` which prevents deletion of the original, uncompressed column indices (see section 9.3.1). This is not needed for the computation, but only for printing the matrix later. The traits can be modified in many ways to control the layout and behavior of sparse and dense matrices. For instance, the sparse matrix storage format can be explicitly specified by the user in the `mtraits`, or the storage layout and column count of the dense matrices (which corresponds to the number of vectors in a block) can be set in the `vtraits`.

The sparse matrix should be assembled from a callback function as it was explained and defined section 9.3.1. This function creates a diagonal matrix with a user-defined value. Lines 17 to 23 demonstrate how this function is used with GHOST. First, a variable `matsrc` is declared and initialized. This structure gets now assigned specific values like the callback function `diag()` as defined in listing 9.3, the argument which will be passed to it as a "`void *`" (as the matrix should contain ones on the diagonal, a pointer to the previously declared variable `one` is passed here), the global number of rows `n` (which is defined elsewhere) and the maximum non-zero count of all matrix rows (which is one for a diagonal matrix).

In the next step, the sparse matrix can be created as shown in line 25. A `ghost_context` is a data structure which holds information about communication patterns for sparse matrix computations. Two sparse matrices with the same non-zero pattern can share a `ghost_context`. As no previously created matrix exists in the present example, `NULL` is passed to `ghost_sparsemat_create()`. Additionally, a pointer to the `mtraits` and the number of trait data structures is passed. If different traits should be specified for the local and remote sub-matrices (see section 9.3.1), an array of traits could be passed here instead. Once the matrix is created, it gets initialized in line 26 using the described `matsrc`. An MPI communicator, in which the

matrix should be distributed, is passed to this function. The last argument to `ghost_sparsemat_init_rowfunc()` determines the process weight as explained in section 9.1. In this example, zero is passed which indicates that GHOST should determine the weight automatically.

The vectors are created in lines 28 and 29. In addition to the traits, `ghost_densemat_create()` gets passed a `ghost_map`. This data structure holds information about matrix and vector distribution in parallel runs. At vector creation, it is used for determining the amount of memory to be reserved for the vector. The map can be extracted from the previously created sparse matrix. Although the row- and column count of the sparse matrix may differ (and with it, the required memory for the in- and output vectors), the maximum of the two is determined via `ghost_context_max_map()` and passed to `ghost_densemat_create()` for simplicity. Note that this may incur storage overhead for non-square matrices if vectors are always used as either in- or output vectors in SpMV operations. However, using the maximum map is a necessity in iterative algorithms where in- and output vectors are swapped between iterations. The input vector $\vec{x}$ is initialized with random numbers and $\vec{y}$ is initialized with zero in lines 30 and 31.

At this point, all required data structures are set up and the SpMV can be computed in line 33. In addition to the matrix and vectors, a structure containing options is passed to `ghost_spmv()`. This structure contains information about scaling factors, augmentation with more operations (as used in chapter 8), and memory locations where to store, e.g., the result of a fused scalar product. In the present example, only the operation $\vec{y} \leftarrow H\vec{x}$ should be computed, i.e., the `ghost_spmv_opts` need not be changed and the initial value can be passed to `ghost_spmv()`.

In the next step, shown in lines 34 to 38, the matrix and vectors should be printed. First, the respective strings have to be created using the functions `ghost_sparsemat/densemat_string()`. Memory gets allocated in these functions which later needs to be freed. For the sparse matrix, in the last argument it needs to be specified whether it should be printed in a dense (1) or sparse format (0).

Finally, the strings are freed and the created GHOST data structures are destroyed in lines 39 to 43. Note that data structures like the `ghost_context` and `ghost_map` rely on reference counting, i.e., they are automatically destroyed once the last reference to them is deleted. As a last step in line 45, `ghost_finalize()` is called, cleaning up all data created by GHOST (like, e.g., task queues and hardware topology information) and performing further tasks like finalizing MPI (if it has been initialized in `ghost_init()`.

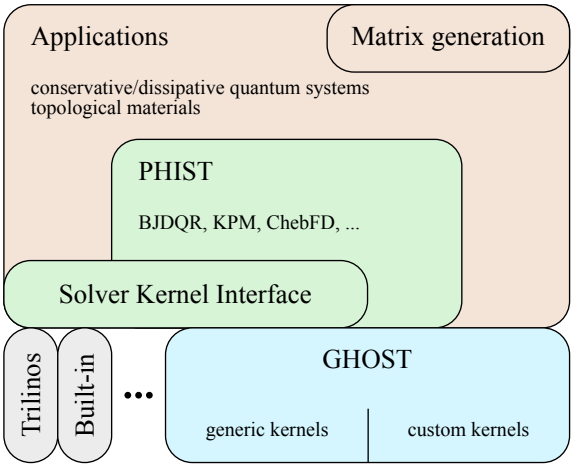


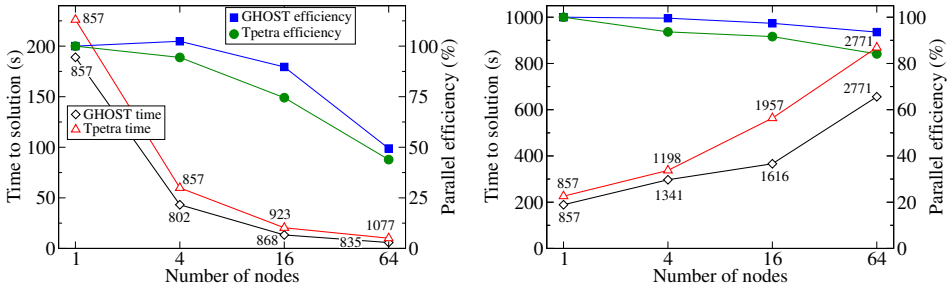
Figure 9.7: Software structure in the ESSEX project. Figure adapted from the ESSEX-II project proposal.

9.7 Integration with Other Software

To be applicable for a wider audience, GHOST and the ESSEX software stack offer several ways to connect to other software or be integrated into existing software environments.

ESSEX software stack. Figure 9.7 visualizes the software structure of the ESSEX project. The overall frame of the project activities is defined by the applications layer, which delivers scalable routines for matrix generation as explained in section 9.3.1. Besides GHOST, a second major software development effort in the ESSEX project is the Pipelined Hybrid Iterative Solver Toolkit (PHIST). PHIST is developed by Thies et al. and provides iterative solvers through an abstract kernel interface, which basically means that different building block libraries can interact with PHIST itself or other solver libraries [163].

The project-related algorithms like BJDQR, KPM, and ChebFD are implemented in PHIST and access compute kernels through the abstract solver kernel interface. Besides those algorithms, adapters to several other algorithm libraries exist. For instance, PHIST provides GHOST adapters for the Trilinos packages Anasazi, which is "an extensible and interoperable framework for large-scale eigenvalue algorithms" [18], and Belos, which "provides next-generation iterative linear solvers and a powerful linear solver developer framework" [23].



(a) Strong scaling, $N_x = N_y = 2^{12}$ **(b)** Weak scaling, $N_x = N_y = 2^{12,\ldots,15}$

Figure 9.8: Scaling performance of GHOST and Tpetra+Kokkos on Emmy for the Anasazi implementation of the Krylov-Schur method through PHIST using the MATPDE-N_x-N_y test case. Iterations counts until convergence are annotated and equal iteration counts are assumed for computing the parallel efficiency. Note the logarithmic scale on the abscissas. Figures adapted from [107].

The PHIST software infrastructure enables algorithm developers to work at high levels of abstraction and have the option to replace kernel libraries easily. A typical development workflow consists in using a robust kernel library like the Trilinos package Tpetra for algorithm development, debugging, and validation, before switching to a highly optimized, yet maybe less robust, kernel library like GHOST.

Use case: an eigenvalue solver with PHIST, GHOST and Trilinos. To demonstrate the interoperability and high efficiency of GHOST, the Anasazi implementation of the Krylov-Schur method for finding a few eigenpairs on the Emmy cluster (see section 2.2) is used. PHIST serves as the interface layer between Anasazi and two different kernel libraries, namely GHOST and Tpetra+Kokkos. Anasazi, Tpetra, and Kokkos are taken from Trilinos version 11.12.1. For both kernel libraries, OpenMP is used inside each IVB* socket and MPI on the inter-socket and -node level. Using a search space of 20 vectors, the 10 eigenvalues with largest real part of the MATPDE-N_x-N_y test case are sought with a residual tolerance of 10^{-6} as stopping criterion.

Figure 9.8 summarizes the experimental results, both for strong (fig. 9.8a) and weak scaling (fig. 9.8b). While PHIST enables straightforward use of both GHOST and Tpetra+Kokkos, one can state that, without going into further detail, GHOST turns out to be preferable both in terms of performance and scalability. This outcome is a result of many of the aforementioned aspects, like the SELL-C-σ format, hand-tuned T&S-GEMM kernels, and careful resource management. Granted that the presented example covers only a

small field of possibilities, the easy and straightforward implementation and the continued development effort in PHIST and GHOST give rise to the well-founded hope that these mechanism could prove useful for a wide range of applications.

9.8 Summary

This chapter presented the open-source software library GHOST, which combines all programming efforts conducted within this work into an efficient, scalable, and inherently heterogeneous software framework. In respect of the numerous existing software packages for sparse linear algebra mentioned in section 1.1, the existence of GHOST is justified by its unique feature set – consisting of distributed and exascale-enabled data structures and compute kernels, hybrid data-parallel processing, automatic and user-guided code generation as well as affinity-aware task parallelism – and consequent, model-guided focus on optimal performance. Out of the many existing packages named above, two which deliver similar features as GHOST are the Trilinos packages Kokkos and Tpetra. Given those, one distinct difference to GHOST is that Trilinos clearly separates between the node level (Kokkos) and the MPI level (Tpetra), whereas GHOST enforces tight integration and a holistic view of all involved levels of parallelism. This allows for easier implementation of, e.g., improved asynchronous MPI communication and high-performance custom compute kernels. The combined software development efforts in the ESSEX project enable straightforward use of GHOST in conjunction with a wide range of other software packages.

10 Node-Level and Large-Scale Application Performance

In this chapter, full application performance for selected sparse eigenvalue solvers as introduced in chapter 3 on various hardware architectures is shown. The presented performance numbers are the result of all described development and performance engineering efforts as presented in chapters 4 to 8 and can be reproduced using GHOST as detailed in chapter 9.

Section 10.1 presents and analyzes performance data of the KPM, whose optimization and implementation has been described in section 8.1. Node-level and large-scale ChebFD+KPM performance with real-world applications is shown in section 10.2, and finally, section 10.3 provides some insight about the performance properties of the developed BJDQR implementation. The performance of the Lanczos method is not covered in this chapter. As mentioned in section 3.1, the most important building block of this method is an SpMV operation, whose performance has already been analyzed in great detail in chapter 5. Hence, analyzing the performance of the Lanczos method would not yield further insight beyond what is discussed there.

Another aspect not covered so far is the energy efficiency of the investigated eigenvalue solvers. While energy efficiency is closely related to performance, a short energy analysis as presented for the KPM in section 10.4 offers interesting insight into this relevant topic.

10.1 KPM Performance Analysis

In this section, the performance of the KPM as introduced in section 3.2 and further investigated in section 8.1 is analyzed by means of the Topi-N_x-N_y-N_z test case.

Node level performance. The performance of the KPM on single nodes of Piz Daint, Piz Daint v2, SuperMUC Phase 2 and Oakforest-PACS is shown in fig. 10.1. The performance numbers obtained on SNB and K20 have been

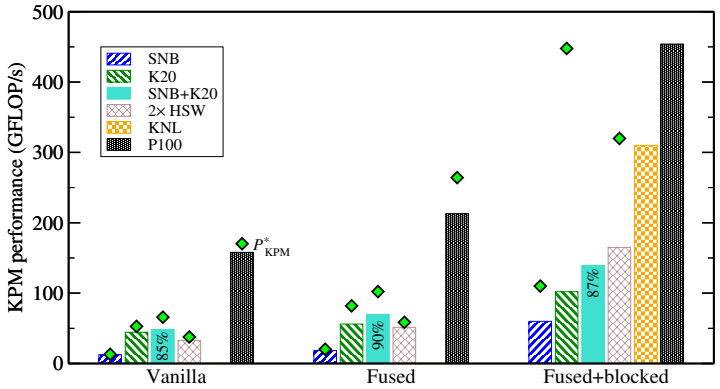


Figure 10.1: Node-level performance and Roofline limit of the KPM with $R = 32$, $M = 2000$, and a Topi-200-100-40 test case (Topi-400-100-40 for heterogeneous runs). The parallel efficiency of heterogeneous runs is annotated in the respective bars. Roofline limits P^*_{KPM} are given according to the arithmetic intensities in eqs. (8.3) to (8.5) scaled with each architecture's b_S. They are omitted for the "fused+blocked" variant on SNB+K20, KNL, and P100 at 558, 1252, and 1444 GFLOP/s for the sake of readability. Figure adapted from [100] and augmented with new results and Roofline limits.

published before by Kreutzer et al. [100] and will be discussed first. The "SNB+K20" bars show the performance of fully heterogeneous data-parallel runs on Piz Daint using GHOST. The parallel efficiency of the heterogeneous runs with respect to the accumulated single-device performance amounts to 85-90%. Reasons for the non-perfect efficiency are communication over the slow PCIe bus as well as the fact that one CPU core is sacrificed for driving the GPU, which diminishes the performance of the non-bandwidth-limited CPU kernel. Considering the measurements on SNB (K20), a significant speedup of $4.8\times$ ($2.3\times$) can be achieved by algorithmic optimization. Given the fully optimized, i.e., "fused+blocked" algorithm, fully heterogeneous execution using GHOST on SNB+K20 yields a performance gain of $1.4\times$ over GPU-only execution on K20. A naïve, yet realistic, usage scenario for accelerated compute clusters like Piz Daint is the K20-only variant of the vanilla algorithm. Compared to this, a performance increase of $3.1\times$ can be achieved by algorithmic optimization and fully heterogeneous execution.

As mentioned in the introduction, performance portability is an important aspect of scientific software development. Without going into great detail, KPM performance on the newer architectures HSW, P100, and KNL is reported in fig. 10.1. Significant performance improvements can be observed, peaking in a performance of around 450 GFLOP/s on P100. Due to the very high performance of P100 compared to the CPU installed in Piz Daint v2, no

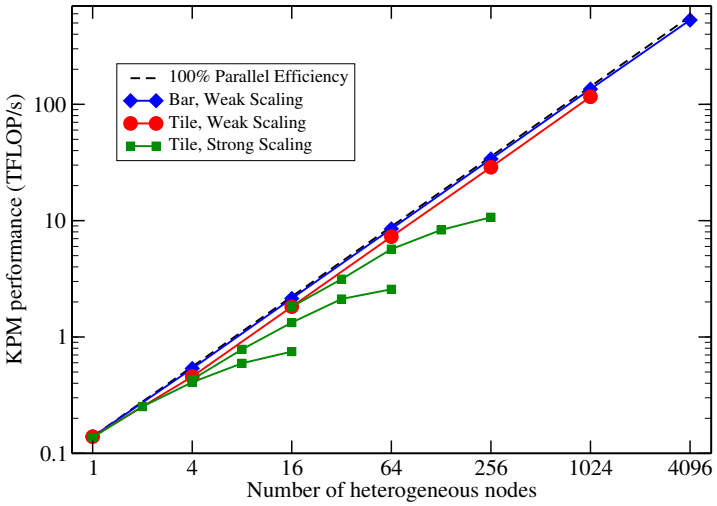


Figure 10.2: Scaling performance of the heterogeneous, "fused+blocked" KPM implementation for $R = 32$, $M = 2000$, and different physical domain geometries on Piz Daint (one SNB and one K20 per node). Both geometries have a size of $400 \times 100 \times 40$ at a single node. At $N \geq 4$ nodes, a "bar" has a size of $(400 \times N) \times 100 \times 40$ and a "tile" has a size of $(200 \times \sqrt{N}) \times (200 \times \sqrt{N}) \times 40$. Note the double logarithmic scale. Figure adapted from [100, 105].

speedup can be gained by heterogeneous execution on this machine as the aforementioned overhead from data-parallel heterogeneous execution cannot be overcompensated by the relatively small performance contribution from the CPU.

Note that on all architectures, the Roofline limit P^*_{KPM} is best matched with the simplest (vanilla) implementation. The close relation of the observed performance with the Roofline limit indicates an efficient implementation of the involved building blocks in terms of main memory bandwidth utilization. For the more complex and compute-intensive "fused" variant, the measured performance falls a bit further behind the Roofline limit on all architectures, but is still within a reasonable level of this best-case performance estimate. Only when blocking is employed, a significant deviation can be observed. In this case, other bottlenecks are likely to become relevant for code execution as discussed in section 8.1.2, and the simple Roofline model assuming pure limitation by main memory bandwidth yields too optimistic estimates.

Scaling performance on Piz Daint. Figure 10.2 depicts the scaling performance of the fully optimized and heterogeneous KPM implementation on Piz Daint. A domain of size $400 \times 100 \times 40$ is solved at a single node and

Version	TFLOP/s	Nodes	Node hours
Fused	14.9	288	164
Fused+blocked*	107	1024	81
Fused+blocked	116	1024	75

Table 10.1: Overview of required resources for applying the KPM to a Topi-(1024×400)-100-40 with $R = 32$ and $M = 2000$. The "fused" version has been run in throughput mode. "Fused+blocked*" indicates a version where a global reduction over the Chebyshev moments is done in each iteration instead of once at the very end.

the z-dimension of the physical domain stays constant at $N_z = 40$ for all variants. The "bar" geometry represents a domain with fixed width $N_y = 100$ and ever-increasing length $N_x = 400 \times$ "number of nodes. The "tile" represents a domain of $400 \times 400 \times 40$ at four nodes with equally increasing N_x and N_y at larger node counts. Due to the regular and scalable structure of this matrix, the weak scaling performance turns out to be close to optimal, especially for the communication-undemanding "bar" geometry. In case of the "tile" geometry, the communication effort increases when going from one to four nodes and stays constant afterwards. Hence, the parallel efficiency drops in this region and shows constant behavior for larger node counts. The strong scaling curves correspond to domain sizes at their respective first data point. As expected, non-optimal, yet adequate, scaling performance can be observed for strong scaling due to increasing communication overhead.

Comparison to throughput mode. Table 10.1 substantiates the performance gains which can be achieved by blocking. Investigating the non-blocked version of the KPM in algorithm 5, one might observe the independence of outer loop iterations and draw the conclusion that highly efficient parallelization could easily be achieved by running R instances of the loop code in "throughput mode." However, table 10.1 reveals that this involves a resource overconsumption in node hours of $2.2\times$, compared to the "fused+blocked" version.

Global reductions. A further interesting observation concerns the reduction of the Chebyshev moments η in lines 8 and 9 of algorithm 6 on page 128. As they are only required at the very end of the computation, it is not necessary to perform a global reduction in each loop iteration. A naïve implementation, which is denoted as "fused+blocked*" in table 10.1 may ignore this fact, while the optimized "fused+blocked" version performs the global reduc-

tion over all nodes only at the very end. The results in table 10.1 indicate a resource overconsumption of 8% of the naïve variant.

10.2 ChebFD+KPM Performance Analysis

Next, the node-level and large-scale parallel performance of the combined ChebFD+KPM kernel as introduced in section 3.3 and detailed in section 8.2 is analyzed.

NUMA-domain-level performance on HSW. The performance properties of ChebFD+KPM, are first analyzed on the HSW architecture. The presented performance data on HSW and SuperMUC Phase 2 have previously been published by Pieper et al. [137]. As mentioned in section 2.1, this architecture features the CoD feature, which basically splits up a CPU socket into two NUMA domains to achieve higher main memory bandwidth on the full chip. As a basic performance analysis is best carried out on a single NUMA domain, only half of a HSW socket, i.e., 7 cores, is used at first.

Figure 10.3 verifies the performance of the implemented ChebFD+KPM kernel, $P_{\text{ChebFD+KPM}}$, against the predicted performance $P^*_{\text{ChebFD+KPM}}$. At first glance, one can observe that the achieved performance deviates significantly from the Roofline limit, especially for large values of n_b. As mentioned in section 8.2, the ChebFD Roofline model assumes perfect re-use of input vector data. As n_b increases, this assumption gets more and more likely to be violated as the cache size is limited. For quantification of this effect, the general data traffic overhead factor Ω gets employed, which is the ratio between the measured and the minimum main memory data traffic. In the upper panel of fig. 10.3, it becomes obvious that Ω indeed increases with n_b. There is little to no overhead, i.e., $\Omega \approx 1$, and the achieved performance is very close to the Roofline limit, at low n_b counts. Throughout the regarded n_b range, Ω increases to approximately 1.75. Without taking further measures on trying to decrease Ω, a more realistic performance bound can be defined as the ratio between the Roofline limit $P^*_{\text{ChebFD+KPM}}$ and Ω. The achieved performance is always within 80% of the corrected Roofline limit, which proves the high efficiency of the implementation also for larger n_b counts.

In the correction factor Ω, only the main memory traffic has been taken into account. The fact that the achieved performance gets so close to the corrected model leads to the conclusion that the execution of the ChebFD filter on HSW is always limited by main memory bandwidth, and any other hardware component poses no bottleneck. This contrasts with the findings about

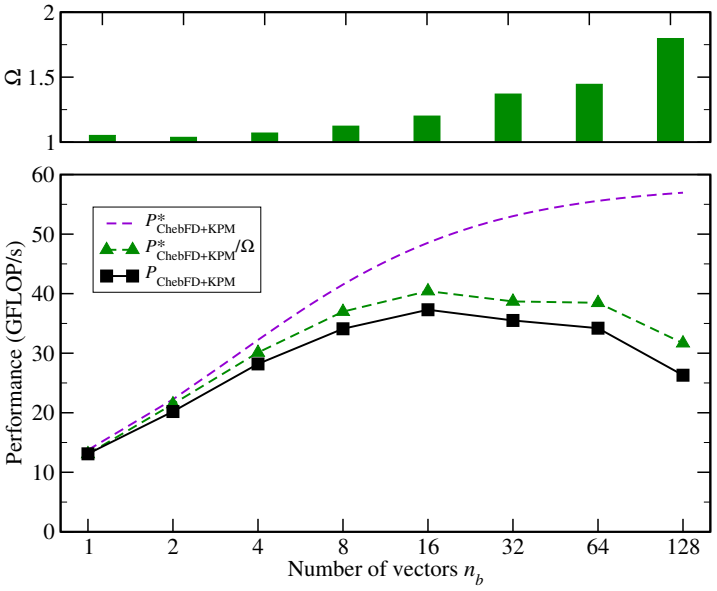


Figure 10.3: Performance and data traffic overhead of the ChebFD+KPM kernel applied to the Topi-128-64-64 test case for varying block vector size on 7 cores (1 NUMA domain) of HSW, using SELL-32-1 with `DYNAMIC,1000` (SELL-1-1 with `DYNAMIC,10000`) scheduling for $n_b = 1$ ($n_b > 1$). $P^*_{\text{ChebFD+KPM}}$ is computed using the arithmetic intensity given in eq. (8.13) multiplied with half (as only one of two NUMA domains is used) the maximum attainable bandwidth of HSW as given in table 2.1 on page 22. The average performance of 400 iterations is reported. Note the logarithmic scale on the abscissa. Figure adapted from [137].

the very similar KPM operator on IVB in section 8.1.2, and can be subjected to architectural changes.

Socket-level performance on HSW. The special architecture of the HSW chip, having two NUMA domains on a single socket due to the CoD feature, demands special attention to OpenMP scheduling on this architecture. Figure 10.4 shows the ChebFD+KPM performance scaled across a full socket of HSW. The transition from the first to the second NUMA domain at 7 cores can be clearly seen. The best performance on a single NUMA domain is achieved with `DYNAMIC` OpenMP scheduling. However, using dynamic scheduling across NUMA domains likely leads to severe performance degradation due to non-local memory accesses. This can be seen in fig. 10.4, where the pure OpenMP variant with dynamic scheduling reveals bad scaling performance at the second NUMA domain.

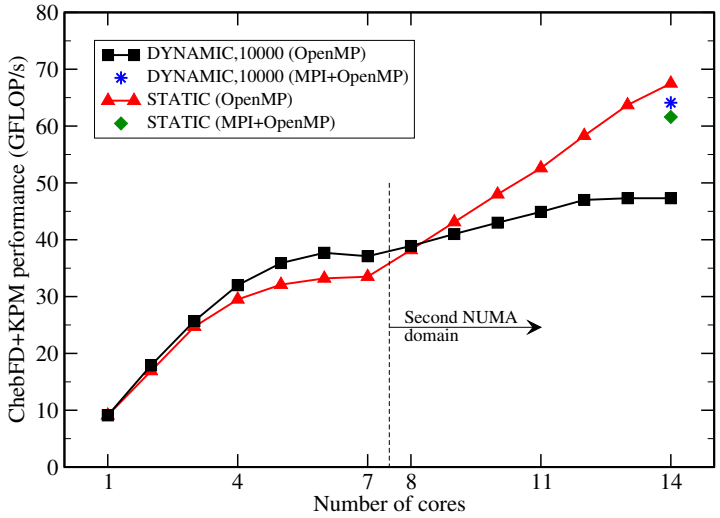


Figure 10.4: Performance of the ChebFD+KPM kernel for different OpenMP scheduling strategies and execution modes on HSW at $n_b = 16$. The runs using MPI used one process per NUMA domain. The average performance of 400 iterations is reported. Figure adapted from [137].

This undesirable behavior can be avoided in two different ways: First, `STATIC` scheduling can be employed. Given that the data is carefully initialized in a NUMA-friendly way, i.e., following the "first touch" policy, good scaling across NUMA domains can be achieved. Second, a hybrid MPI+OpenMP implementation can be used. In the latter case, it is often a good practice is to place one MPI process on each NUMA domain and employ the optimal scheduling strategy, i.e., `DYNAMIC,10000` in this case, in each of them. This is especially advisable if no knowledge about NUMA-aware initialization and data handling of the underlying code is present. The performance of both execution modes is shown in fig. 10.4 and it turns out that the pure OpenMP execution with `STATIC` scheduling yields the best performance of all variants, thanks to the NUMA-aware implementation in GHOST. This execution mode will be used in the following large-scale experiments.

Scaling performance on SuperMUC Phase 2. Figure 10.5 shows the performance as a function of n_b on different node counts with a fixed problem size per node (weak scaling). It becomes obvious that speedups from choosing an optimal block vector size amplify on large node counts: While only a factor of $1.3\times$ can be gained by choosing the optimal block size on a single node, the speedup adds up to $1.6\times$ on 512 nodes. This is due to the fact that

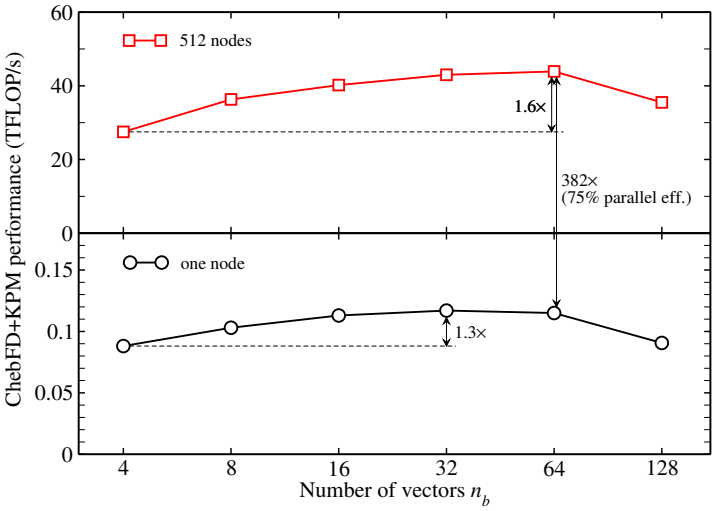


Figure 10.5: ChebFD+KPM performance as a function of n_b on different node counts for a fixed problem size (Topi-128-64-64) per node and a quadratic tile-shaped global domain on SuperMUC Phase (2× HSW per node). The average performance of 400 iterations is reported. Note the logarithmic scale on the abscissa. Figure adapted from [137].

MPI message sizes also increase with n_b, reducing the communication effort in terms of message count and latency. In contrast to fig. 10.3, where the optimal n_b was determined as 16 when only a single NUMA domain (i.e., 7 cores) is used, fig. 10.5 indicates an optimal n_b of 32 on a single node, and 64 on 512 nodes.

Weak scaling experiments on SuperMUC Phase 2 are shown in fig. 10.6. The first observation is the overall good weak scalability, regardless of n_b. The inlet shows the relative performance per node, which is directly related to the parallel efficiency. Here, two major drops in parallel efficiency on the scaling curve can be observed. From one to two nodes, MPI communication over the network is enabled which causes the first drop in efficiency. The second distinct drop happens when going from 128 to 512 nodes due to the special network architecture of SuperMUC Phase 2. As mentioned in section 2.2, this machine consists of 512-node islands with a quarter of the bandwidth between them compared to the intra-island bandwidth. The executed 512-node jobs were not confined to a single island, but spread across two islands by the batch system. Hence, the drop in parallel efficiency to the final value of 75% (which is still a satisfiable parallel efficiency under these hardware constraints) at $n_b = 64$ can be observed.

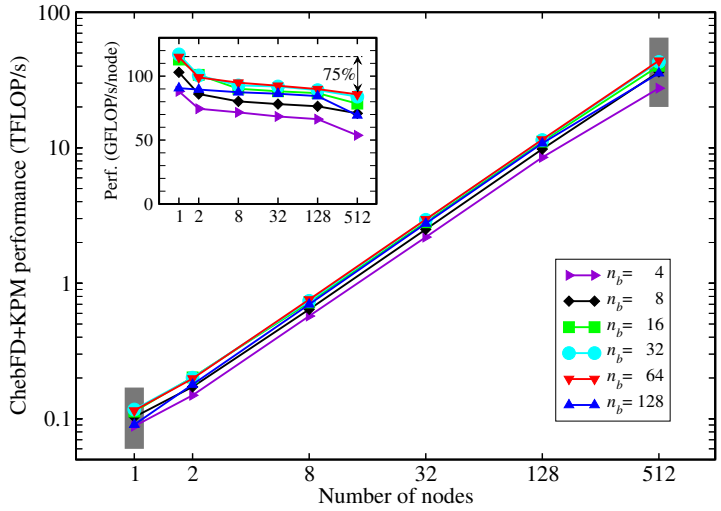
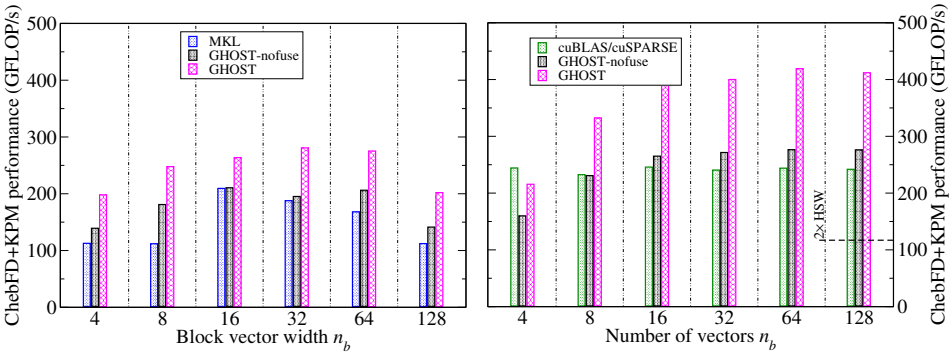


Figure 10.6: ChebFD+KPM weak scaling performance on SuperMUC Phase 2 with a fixed problem size (Topi-128-64-64) per node and a quadratic tile-shaped global domain on SuperMUC Phase (2× HSW per node). The average performance of 400 iterations is reported. The data points shaded in gray are also shown in fig. 10.5. Note the double logarithmic scale. Figure adapted from [137].



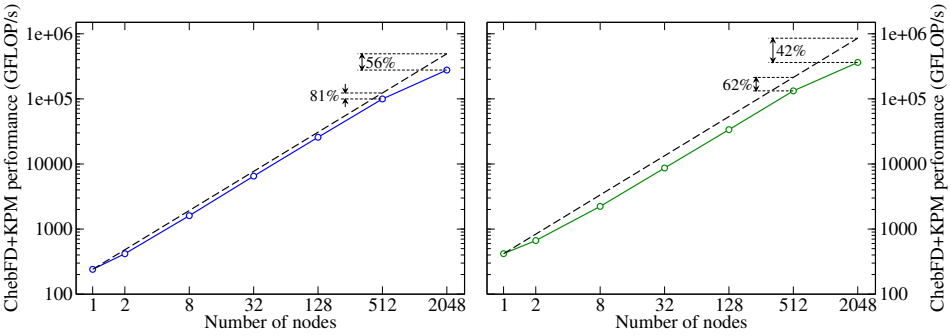
(a) KNL **(b)** P100

Figure 10.7: Median ChebFD+KPM performance for a Topi-128-64-64 matrix and 10 iterations with a polynomial degree $N_p = 500$ on KNL and P100. Note the logarithmic scale on the abscissa. The "2× HSW" performance corresponds to the lower panel of fig. 10.5.

Performance on P100 and KNL. Figure 10.7 demonstrates the performance portability of the present implementation and data structures to a new generation of architectures represented by KNL and P100. Concerning, the relation between n_b and $P_{\text{ChebFD+KPM}}$, similar observations as on the formerly investigated architectures can be made. This figure also demonstrates the performance advantage of the GHOST implementation with respect to a reasonable baseline, namely the vendor-supplied and -tuned numerical building block libraries Intel MKL and NVIDIA cuBLAS/cuSPARSE. Additionally, a version based on GHOST, but not using its fused kernels, is taken into account ("GHOST-nofuse"), which allows a fair comparison of raw building block performance without the systematic advantage of GHOST due to kernel fusion. The KNL performance numbers in fig. 10.7a indicate that the optimized GHOST version always yields the best performance and tops out at around 260-280 GFLOP/s at $n_b = 16 - 64$. The dominance of GHOST over MKL is mainly due to kernel fusion, but even the non-fused GHOST version outperforms MKL for most values of n_b, which indicates a more efficient implementation of the basic building blocks.

Figure 10.7b shows that the performance on P100 is on average on a higher level than on KNL. At $n_b = 4$, the GHOST implementations show a relatively low absolute performance and fall behind cuSPARSE/cuBLAS. This is due to the fact that it is not optimized for this range of low n_b, which is irrelevant for the present ChebFD application. Although a search space of only four vectors is not a realistic choice for ChebFD, a tuned version for this kernel and low n_b values could be added once a relevant application comes up. In the more relevant large-n_b region, GHOST-nofuse performs slightly better than cuBLAS/cuSPARSE while the optimized GHOST version outperforms both significantly due to kernel fusion and an efficient implementation. The performance tops out at around 420 GFLOP/s.

Compared to the performance on a node with two HSW chips which adds up to 117 GFLOP/s, a speedup of $2.4\times$ ($3.6\times$) can be achieved on KNL (P100). Assuming bandwidth-limited execution on all architectures, the maximum speedup results from the ratio of b_s values and adds up to $3.9\times$ ($4.5\times$) on KNL (P100). However, the analysis for the similar KPM kernel in section 8.1.2 indicates that this is not necessarily a valid assumption on all compute architectures and a detailed analysis (which is omitted here) is required for deeper understanding of the performance on KNL and P100. Although this analysis is omitted here, it can be stated that the performance achieved on those two architectures seems to be adequate, given the relatively sensitive performance characteristics of accelerator architectures which makes it harder to

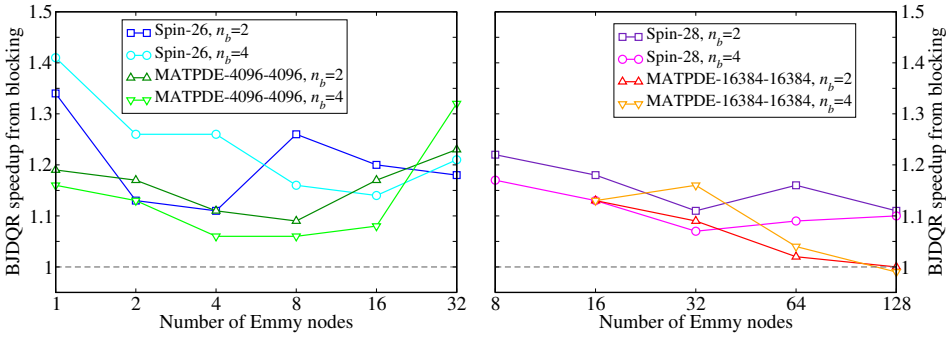


(a) Oakforest-PACS, $n_b = 16$ **(b)** Piz Daint v2, $n_b = 64$

Figure 10.8: ChebFD+KPM weak scaling performance with a Topi-128-64-64 domain on one node, Topi-128-128-64 on two nodes, and equally increasing N_x and N_y on larger node counts. The median performance of 10 iterations with a polynomial degree $N_p = 500$ is shown on Oakforest-PACS and Piz Daint v2. No significant fluctuations were observed. The dashed lines show ideal speedup from a single node. Note the double logarithmic scale.

reach optimal performance (as it can be seen, e.g., in the SpMV performance analysis in section 5.4).

Scaling performance on Piz Daint v2 and Oakforest-PACS. Weak scaling performance for a "tile" geometry on Oakforest-PACS and Piz Daint v2 is shown in Figure 10.8. The block vector widths n_b are chosen according to best performance achieved on a single node as shown in fig. 10.7. On two nodes, the physical domain represents a quadratic tile of size $128 \times 128 \times 64$, which gets double in each dimension for each larger node count. These experiments are similar to what is shown for SuperMUC Phase 2 in fig. 10.6, but scaling up to 2048 nodes. The parallel efficiency at 512 and 2048 nodes, both with respect to the single-node performance, is annotated in the figure. The efficiencies at 512 nodes of 81% and 62% compare to the 75% parallel efficiency obtained on SuperMUC Phase 2 shown in fig. 10.6. A significant drop in parallel efficiency can be observed on both machines when going from 512 to 2048 nodes. A closer analysis reveals that the communication performance decreases in this region and hence, the efficiency drop can likely be attributed to the network architecture. Generally, the parallel efficiency on the KNL machine Oakforest-PACS is higher than on the P100 machine Piz Daint v2. However, on the largest node count, Piz Daint v2 still yields $1.3\times$ the performance of Oakforest-PACS. On both machines, the performance on 2048 nodes is in the range of hundreds of TFLOP/s.



(a) Small test cases **(b)** Large test cases

Figure 10.9: BJDQR speedup from blocking in a strong and weak scaling scenario for searching 20 exterior eigenvalues at the lower end of the spectrum. Note the logarithmic scale on the abscissa. Figures adapted from [142].

10.3 BJDQR Performance Analysis

To demonstrate the benefits of performance engineering for another class of eigenvalue solvers, this section provides a brief analysis of the performance of the BJDQR method. The central statement is that, although blocking due to numerical optimizations leads to a higher SpMV count, an efficient implementation (guided by performance models) can overcompensate this effect and lead to a shorter overall time to solution for a given problem. Further information can be found in the article by Röhrig-Zöllner et al. [142]. The present BJDQR eigenvalue solver is implemented in PHIST, using GHOST kernels for numerical operations. All experiments have been performed on IVB*. As described in section 3.4, the most time intensive building blocks of BJDQR are a shifted SpMMV and T&S-GEMM operations, whose performance has been extensively analyzed and whose efficiency with respect to the Roofline model has been demonstrated before. In this section, no performance numbers in terms of FLOP/s are reported, but rather absolute runtimes and speedup numbers. Analyzing the time needed for solving a fixed problem is intuitive and allows for easy comparison of different approaches. This section presents another showcase for performance opportunities opened by efficient implementations of block algorithms.

Speedup from blocking. The speedup from blocking with respect to a single-vector variant of a full BJDQR eigensolver is shown in fig. 10.9. Figure 10.9a shows a strong scaling scenario for two different test cases with two

Method	n_b	# SpMV	Walltime (s)	Time/ SpMV (ms)	Block speedup
PRIMME (a)	1	1387	326	80 (34%)	1.00
	2	1688	370	80 (37%)	0.82
	4	1811	403	81 (39%)	0.77
PRIMME (b)	1	1395	327	80 (34%)	1.00
	2	1487	326	80 (37%)	0.94
	4	1746	366	81 (39%)	0.80
PHIST/ GHOST	1	1476	343	53 (23%)	1.00
	2	1562	286	37 (20%)	1.20
	4	1774	252	26 (19%)	1.36

Table 10.2: Comparison of the PHIST/GHOST BJDQR implementation with PRIMME for the Spin-26 test case with RCM re-ordering. The column 'Time/SpMV' shows the average time per single SpMV product and the contribution to the overall runtime in percent. The (a) configuration is similar to BJDQR, while (b) uses a more sophisticated stopping criterion [142].

block sizes each and fig. 10.9b shows similar data for larger matrices and node counts. The experiments have been conducted on the Emmy cluster (i.e., with 2 IVB* sockets per node), 2 MPI processes per node and OpenMP underneath. All in all, it can be observed that blocking yields a shorter time to solution in almost all displayed cases. Only for the MATPDE-N_x-N_y matrix on 128 nodes it is on par with the non-blocked variant, but it never falls behind it. Using a block vector algorithm in distributed memory has two counteracting effect. Clearly, the total communication volume increases with the number of SpMV operations, but on the other hand, message aggregation leads to larger messages and a smaller influence of latency effects. The sparsity pattern and distribution of the sparse matrix among the processes are further factors which influence the communication performance and contribute to the non-smooth block speedup in the strong scaling runs.

Comparison to PRIMME. Table 10.2 shows a comparison between the developed BJDQR implementation and an eigenvalue solver software named Preconditioned Iterative Multimethod Eigensolver (PRIMME) [158], which is based on the Jacobi-Davidson Quasi-Minimal Residual (JDQMR) method and uses SpM(M)V kernels from the MPI-parallel Trilinos package Epetra [82]. For this experiment, 20 eigenpairs are sought with an accuracy of 10^{-8}, and both methods are re-started from 28 vectors if the basis reaches a size of 60.

There is no claim that the present BJDQR implementation is in some way superior over the certainly more sophisticated implementation in PRIMME. JDQMR has obvious advantages over JDQR [159], but those are complementary to the performance advantages due to blocking in BJDQR. The primary goal here is not to compare the overall runtimes of the two solvers, which is difficult as they are very different in nature, and PRIMME does not use blocking in the inner solves. It should rather be demonstrated that blocking pays off performance-wise. PRIMME is only capable of using column-major block vectors for SpMMV which has known implications on performance as discussed in chapter 6. Furthermore, it relies on the BLAS library's GEMM implementation for T&S-GEMM operations, which is problematic as demonstrated in chapter 7.

Table 10.2 shows that in the single vector case, PRIMME outperforms the PHIST/GHOST implementation, which is due to a lower SpMV count and a probably smarter implementation of the algorithm itself [142]. PRIMME does not benefit from blocking, due to the aforementioned issues and the fact that the used JDQMR method does not solve the correction equation in a blocked way. Hence, the block speedup of PRIMME just reflects the increasing SpMV count. On the contrary, the PHIST/GHOST implementation benefits from blocking, thanks to the highly efficient shifted SpMMV and T&S-GEMM operations. Despite a growing SpMV count (which is due to numerical reasons), the overall runtime decreases and blocking, which is known to yield a more robust algorithm when it comes to multiple or clustered eigenvalues [142], can be employed and even yield higher performance.

Conclusion. The BJDQR method as developed by Röhrig-Zöllner et al. [142] and shortly reviewed here is the first block version of the JD eigenvalue solver which yields shorter time to solution compared to a respective single-vector version. Performance engineering of the involved compute kernels is a necessity for achieving this. Having a block version which outperforms the single-vector algorithm is especially useful in light of the fact that blocking is beneficial from a numerical point of view. The example of the presented BJDQR algorithm also demonstrates that performance engineering is inevitable also in view of numerical optimizations if high performance should be achieved.

10.4 KPM Energy Analysis

The algorithmic optimization of the KPM and the development as well as use of the SELL-C-σ format give rise to interesting opportunities for further analysis regarding performance and energy efficiency.

While performance models are extensively used in this work, in-depth modeling and analysis of power dissipation is no part of it. HAGER et al. [79] have developed a phenomenological model for power and energy consumption which provides useful guidelines for energy-optimal operating points of a code, in terms of the number of active cores and their clock frequency. An analysis of the energy properties of the KPM in view of this model has previously been published by KREUTZER et al. [106] and is briefly reviewed here.

Even without further knowledge about energy modeling, it is obvious that the energy to solution E is a key metric when it comes to the evaluation of an implementation's energy efficiency. On the basis of the KPM, its interplay with the performance will be briefly analyzed in the following, focusing on the influence of SMT and the selected sparse matrix storage format. A parameter omitted here is the influence of clock frequency on the performance and energy consumption. In this analysis, the energy to solution is measured using LIKWID on the CPU package level, i.e., ignoring contributions from RAM, I/O, disks, and so on.

Z-plots. The interplay between socket scaling, performance, and energy consumption is analyzed by means of so-called *Z-plots*, plotting performance on the abscissa and energy to solution on the ordinate. Each set of data points represents intra-socket scaling for a fixed problem size. In such a plot, vertical lines are performance iso-lines, horizontal lines are energy iso-lines, and hyperbolas are power iso-lines. In case of saturating performance, the curve bends upwards, indicating that more resources are used without increasing performance, leading to a higher energy consumption. In case of scalable performance, the curve is expected to stay flat or decrease throughout the scaling range.

Influence of SMT. A factor which has been ignored until now is the influence of SMT on performance and energy consumption. Figure 10.10 shows the intra-socket scaling behavior of the three KPM variants on IVB*, considering both the performance and energy to solution E. The first observation is the strong performance saturation of the non-blocked variants at 5-6 cores, which is due to their strong limitation by main memory bandwidth (cf.

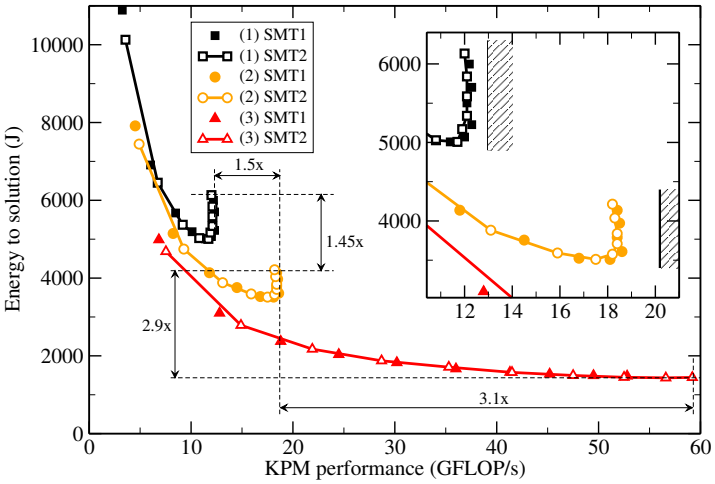


Figure 10.10: Intra-socket performance and energy consumption of the "vanilla" (1), "fused" (2), and "fused+blocked" (3) KPM implementations with $R = 32$ and $M = 100$ for one (SMT1) and two (SMT2) threads per core on IVB*. Line segments between points indicate intra-socket scaling. The hatched bars represent upper Roofline performance limits according to the arithmetic intensities specified in eqs. (8.3) to (8.5). Figure adapted from [106].

section 8.1.2 for performance models). Both saturate at a performance level which is reasonably close to the simple bandwidth-limited Roofline model. The "fused" kernel requires more cores to reach saturation due to its higher arithmetic intensity. Contrary to those two variants, the blocked version is not limited by main memory bandwidth and can profit from all of the socket's cores. The influence of SMT for the saturating cases is negligible, which is in accordance with the frequent observation that SMT cannot improve performance in the presence of a strong memory bottleneck. For the blocked code, SMT comes with a $1.12\times$ performance gain (indicating that the bottleneck is not the L3 cache bandwidth in this case), but without any energy savings. Hence, using a second thread per core obviously increases the power dissipation, which also complies with the small energy increase in the saturated cases for SMT2.

It can be observed that performance gains due to algorithmic or implementation enhancements translate into almost proportional energy savings, which is a consequence of the low dynamic power of the regarded IVB* chip. Concretely, the performance gain (energy saving factor) of the fully optimized version compared to the fused version is $3.1\times$ ($2.9\times$), and $1.5\times$ ($1.45\times$) of the fused version compared to the vanilla version. This agreement would get even more accurate if the full baseline power of all system components was

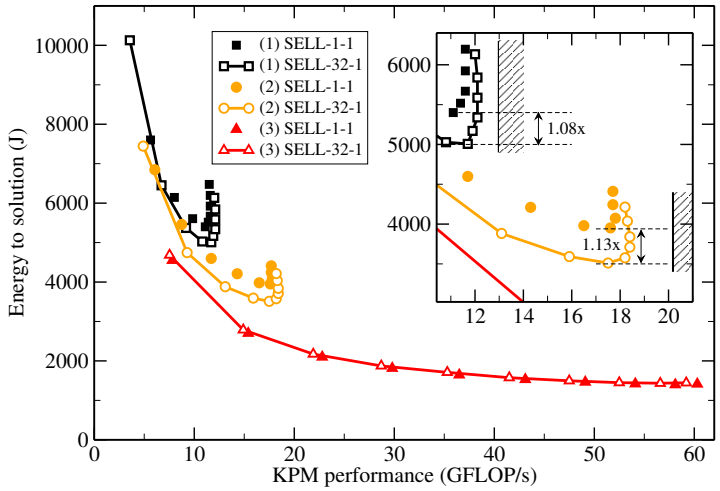


Figure 10.11: Intra-socket performance and energy consumption of the "vanilla" (1), "fused" (2), and "fused+blocked" (3) KPM implementations with $R = 32$ and $M = 100$ using different sparse matrix storage formats on IVB*. Line segments between points indicate intra-socket scaling. The hatched bars represent upper Roofline performance limits according to the arithmetic intensities specified in eqs. (8.3) to (8.5). Figure adapted from [106].

taken into account. For the saturating variants, 20% of energy can be saved by choosing the minimum number of cores which lead to performance saturation, instead of using the full chip.

Influence of the sparse matrix storage format. An analysis of the KPM performance and energy efficiency using different sparse matrix storage formats, namely SELL-1-1 (i.e., CRS) and SELL-32-1, can be seen in fig. 10.11. A major motivation for the development of the SELL-C-σ format was its SIMD-friendliness. However, on current multi-core CPUs this does not necessarily lead to performance benefits in bandwidth-limited scenarios, an observation which has also been made in section 5.5. This is due to the fact that the main memory bandwidth can often be saturated also with non-vectorized, i.e., inefficient on a core level, code. However, even for bandwidth-limited kernels efficient vectorization becomes important if energy efficiency is a target metric. Using SELL-32-1 instead of SELL-1-1, one can save 8% (13%) of energy for the vanilla (fused) variant. This is mainly due to the higher single-core performance and a smaller number of cores needed for saturation as a consequence thereof. The small performance advantage of SELL-32-1 in the saturated case can be attributed to a more beneficial access pattern to the right hand side vector. For the blocked variant, using SELL-32-1 does not entail

a performance benefit due to the negligible influence of matrix data access in this case. This coincides with the findings about SpMMV performance in section 6.2.

10.5 Summary

This chapter presented performance and energy efficiency of the KPM, ChebFD+KPM, and BJDQR eigenvalue solvers on several high-end compute systems. The results demonstrate the high efficiency, both in terms of performance and energy consumption, and real-world applicability of the developed building blocks and parallel software components for highly efficient eigenvalue computations. The high performance and efficiency results from several ingredients, from performance-engineered data structures like SELL-C-σ and building blocks like custom compute kernels to data-parallel heterogeneous execution and parallel compute kernels as available in GHOST. The developed KPM implementation is the first of its kind to perform fully heterogeneous computations on a CPU+GPU machine. Being similar to KPM, also the implemented ChebFD+KPM method stands out in terms of efficiency. Using the developed BJDQR method and performance-engineered kernels, it was for the first time possible to obtain a shorter time to solution for a blocked JD method compared to single-vector implementations. All presented solvers show highly efficient scaling performance, enabling sparse eigensolver algorithms (which are originally strongly limited by main memory bandwidth and low in FLOP rate) at hundreds of TFLOP/s on some of the world's largest homogeneous and heterogeneous supercomputers.

11 Summary

This work addresses the development of numerical building blocks for sparse linear algebra algorithms on current and next-generation supercomputers. The large scale of computations impedes the use of direct methods and steers the focus towards iterative solvers. In such solvers, basic operations like Sparse Matrix-Vector Multiplication (SpMV), Sparse Matrix-Multiple Vectors Multiplication (SpMMV) and Tall & Skinny General Dense Matrix-Matrix Multiplication (T&S-GEMM) are widely required and representative for many challenges regarding the efficiency of sparse linear algebra algorithms. Besides the presence of optimized implementations of such kernels, a holistic view of applications, algorithms, and implementations gives rise to further optimization potential, for example by the implementation of application-tailored variants of those basic operations. Besides the development of performance-engineered basic compute kernels and application-specific variants, this thesis presents a scalable and heterogeneous open-source software library which opens the use of the developed software to a wide audience and range of supercomputers. From an algorithmic perspective, the standard sparse eigenvalue problem is addressed. A selection of iterative solvers is identified, representing different classes of algorithms for different classes of target eigenvalues. However, due to a frequent occurrence of similar patterns and building blocks in a much wider range of sparse linear algebra algorithms, the discussed results are certainly not restricted to the selected eigenvalue solvers.

Model-guided performance engineering. In this work, the entire process of development is guided by performance models. The Roofline model is presented as a suitable tool for performance modeling in the field of sparse linear algebra and refined models are presented for the considered basic building blocks and custom compute kernels. Performance modeling allows to assess the efficiency of an implementation and is helpful in identifying

relevant architectural bottleneck from which promising performance optimization strategies can be derived.

SpMV. The SpMV is perhaps the most prominent building block of sparse linear algebra and it is granted special consideration in this work. The performance of this operation strongly depends on the sparse matrix storage format and its interplay with the compute architecture as well as the matrix structure. Obviously, the dependence of the optimal sparse matrix storage format on the architecture is greatly undesired, especially in view of today's heterogeneous compute platforms. This dependence can be eliminated by a unified, hardware-agnostic storage format. One such format – SELL-C-σ– is developed within this work and its high efficiency across a range of relevant, heterogeneous architectures is demonstrated. It turns out that not only does SELL-C-σ enable cross-platform SpMV operations with a single matrix data structure, but it also outperforms device-specific storage formats for a wide range of sparse matrices.

SpMMV and T&S-GEMM. Main memory data traffic is frequently the limiting hardware bottleneck for sparse linear algebra algorithms. Consequently, any method to save on data transfers can potentially increase performance. Vector blocking, which is also used for increasing robustness in some algorithms, is an important means to alleviate the data transfer bottleneck by saving on sparse matrix data transfers. Algorithms with block vectors necessitate building blocks like SpMMV and T&S-GEMM, both of which are analyzed in this work. For SpMMV, it is shown in this work that the above-mentioned influence of the sparse matrix storage format on the performance is mitigated compared to SpMV, and further bottlenecks beyond main memory bandwidth come into play. Furthermore, an analysis of block vector layouts reveals that interleaved storage of vectors yields best SpMMV performance by achieving regular data access (as compared to the inherently irregular patterns of SpMV). The considered T&S-GEMM kernels are of special interest as severe performance drawbacks of off-the-shelf General Dense Matrix-Matrix Multiplication (GEMM) implementations can be revealed. Performance experiments conducted in this work demonstrate that custom implementations of such kernels increase the performance for five different kernels and configurations by a factor of $4\times$ on average and up to a factor of $27\times$ compared to the Intel Math Kernel Library (MKL).

Custom compute kernels. Another measure to alleviate the aforementioned bottleneck of data traffic is the development of custom, i.e., fused compute kernels. In this work, manual kernel fusion for selected eigenvalue solvers is employed. The performance potential of this optimization is significant, but it requires a careful, and in the best case model-guided, implementation. To give an example, a speedup of $4.8\times$ due to kernel fusion and vector blocking is demonstrated for an implementation of the Kernel Polynomial Method (KPM). While the presented kernels are specifically developed for selected solvers, they can serve as blueprints for further algorithms.

General, Hybrid, and Optimized Sparse Toolkit (GHOST). All developed building blocks are part of the open-source software library GHOST. On top of performance-engineered numerical kernels, GHOST features data-parallel heterogeneous execution of kernels in distributed memory using the MPI+X programming paradigm, affinity-aware task level parallelism, and further features like automatic code generation. Implementations of relevant eigenvalue solvers like KPM, Chebyshev Filter Diagonalization (ChebFD) and Block Jacobi-Davidson QR (BJDQR) based on GHOST are publicly available. Its unique feature set enables highly efficient spare linear algebra computations from a single core to thousands of nodes on some of the world's largest supercomputers with completely different architectures, such as the CPU-GPU system Piz Daint and the many-core system Oakforest-PACS.

Large-scale application performance. The final outcome of this work are highly efficient and scalable eigenvalue solver implementations. The ChebFD implementation based on GHOST enables the computation of hundreds of interior eigenpairs of a matrix with billions of rows, which is the first interior eigenvalue computation at this scale. Thanks to performance engineering, the presented BJDQR algorithm is the first of its kind to reveal a reduction in time to solution for a block implementation (which is in some cases favorable from a numerical point of view) compared to a single-vector version. The implemented KPM scheme is successfully employed on up to 4096 nodes of a heterogeneous CPU-GPU machine, using GHOST to make efficient use of both architectures to compute the eigenvalue density of a matrix with 26 billion rows, which is the first fully heterogeneous computation of this kind and on this scale.

12 Outlook

The results of this work offer a wide field of interesting possibilities for future work. While the developed software underlies constant improvement in terms of supporting new hardware architectures and extending the feature set, several more methods and techniques are worth investigating to further enhance applicability and efficiency. In the following, a selection of topics for further research based on the presented work is given.

Scalability. A constantly relevant topic in high-performance computing is the improvement of scaling performance. In sparse linear algebra, scalability often depends strongly on the sparse system matrix, as it directly influences the communication effort of kernels like SpMV. In the best case, the system matrix has a scaling-friendly structure without requiring further manipulation. If this is not the case, techniques like hypergraph partitioning [38] can be employed in order to increase the scalability. Within this work, preliminary experiments with hypergraph partitioning led to mixed results: While the SpMV performance could certainly be increased by reducing the communication volume, the partitioning procedure itself consumed a significant amount of runtime and its scalability appeared to be limited. The distributed memory-parallel Reverse Cuthill-McKee (RCM) as recently suggested by Azad et al. [15] has not yet been investigated in the context of the present algorithms and applications. In future work, those techniques should be analyzed thoroughly as they potentially offer a real chance to bring difficult problems like the Spin-N_{Up} matrix to a very large scale.

A further idea for better scalability in the context of block algorithms is pipelined communication of vector data. For instance, it is possible to consider a subdivision of block vectors. In a parallel SpMMV operation, only the first part of the block vector is communicated initially. The process-local SpMMV is then done using this part, and at the same time the next part of the block vector is communicated. Obviously, the sparse matrix needs to be read for each part of the block vector in this case. However, communication can

be overlapped with computation. Hence, for certain matrices and block vector sizes there is a chance that the positive effect from communication hiding outweighs the overhead of this technique. In future work, this should be investigated and implemented if applicable.

Special matrix structures. An important aspect which is out of scope for this work is the consideration of special matrix structures. In this regard, especially the presence of symmetry in the sparse matrix is of relevance for the present applications and algorithms. Besides what is presented in the related work section on this topic, a promising approach is to employ Block Multi-Coloring (BMC), a technique to increase the performance of sparse matrix operations with data dependencies [5]. On such operation is symmetric SpMV, and BMC is a promising approach to significantly increase the performance of it.

Dynamic load balancing. One of the crucial issues in the context of heterogeneous computing is load balancing. Currently, the work distribution of compute kernels is static and done by means of micro-benchmarks or the decision of the user. A leap forward in this respect would be the implementation of dynamic load balancing. This could involve, e.g., the automatic transfer of matrix and vector data to under-employed devices. One important prerequisite for this to work is already present, namely the platform-agnostic SELL-C-σ sparse matrix storage format.

Bibliography

[1] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. "Performance, Design, and Autotuning of Batched GEMM for GPUs." In: *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*. Ed. by J. M. Kunkel, P. Balaji, and J. Dongarra. Cham: Springer International Publishing, 2016, pp. 21–38. ISBN: 978-3-319-41321-1. DOI: 10.1007/978-3-319-41321-1_2.

[2] A. Agarwal and M. Levy. "The KILL Rule for Multicore." In: *Proceedings of the 44th Annual Design Automation Conference*. DAC '07. San Diego, California: ACM, 2007, pp. 750–753. ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278668.

[3] R. C. Agarwal, F. G. Gustavson, and M. Zubair. "A High Performance Algorithm Using Pre-Processing for the Sparse Matrix-Vector Multiplication." In: *Proceedings Supercomputing '92*. Nov. 1992, pp. 32–41. DOI: 10.1109/SUPERC.1992.236712.

[4] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang. "Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations." In: *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1213–1222. ISBN: 978-1-4799-3800-1. DOI: 10.1109/IPDPS.2014.125.

[5] C. L. Alappat. "Implementation and Performance Engineering of the Kaczmarz Method for Parallel Systems." MA thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2016.

[6] A. Alvermann, A. Basermann, H. Fehske, M. Galgon, G. Hager, M. Kreutzer, L. Krämer, B. Lang, A. Pieper, M. Röhrig-Zöllner, F. Shahzad, J. Thies, and G. Wellein. "ESSEX: Equipping Sparse Solvers for Exascale." In: *Euro-Par 2014: Parallel Processing Workshops*. Ed. by L. Lopes, J. Žilinskas, A. Costan, R. Cascella, G.

Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander. Vol. 8806. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 577–588. isbn: 978-3-319-14312-5. doi: 10.1007/978-3-319-14313-2_49.

[7] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. "Communication-Avoiding QR Decomposition for GPUs." In: *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*. May 2011, pp. 48–58. doi: 10.1109/IPDPS.2011.15.

[8] "Annex B Legacy Blas." In: *The International Journal of High Performance Computing Applications* 16.1 (2002), pp. 94–107. doi: 10.1177/109434200201600011001.

[9] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler. "Efficiency of General Krylov Methods on GPUs – An Experimental Study." In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2016, pp. 683–691. doi: 10.1109/IPDPSW.2016.45.

[10] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler. "Preconditioned Krylov solvers on GPUs." In: *Parallel Computing* (2017). issn: 0167-8191. doi: 10.1016/j.parco.2017.05.006.

[11] H. Anzt, M. Kreutzer, E. Ponce, G. D. Peterson, G. Wellein, and J. Dongarra. "Optimization and performance evaluation of the IDR iterative Krylov solver on GPUs." In: *The International Journal of High Performance Computing Applications* 32.2 (2018), pp. 220–230. doi: 10.1177/1094342016646844.

[12] H. Anzt, S. Tomov, P. Luszczek, W. Sawyer, and J. Dongarra. "Acceleration of GPU-based Krylov solvers via data transfer reduction." In: *The International Journal of High Performance Computing Applications* 29.3 (2015), pp. 366–383. doi: 10.1177/1094342015580139.

[13] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dec. 2006. url: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[14] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. "Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. New Orleans, Louisana: IEEE Press, 2014, pp. 781–792. ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.69.

[15] A. Azad, M. Jacquelin, A. Buluç, and E. G. Ng. "The Reverse Cuthill-McKee Algorithm in Distributed-Memory." In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2017, pp. 22–31. DOI: 10.1109/IPDPS.2017.85.

[16] D. F. Bacon, S. L. Graham, and O. J. Sharp. "Compiler Transformations for High-performance Computing." In: *ACM Comput. Surv.* 26.4 (Dec. 1994), pp. 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406.

[17] C. G. Baker and M. A. Heroux. "Tpetra, and the Use of Generic Programming in Scientific Computing." In: *Sci. Program.* 20.2 (Apr. 2012), pp. 115–128. ISSN: 1058-9244. DOI: 10.1155/2012/693861.

[18] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist. "Anasazi Software for the Numerical Solution of Large-scale Eigenvalue Problems." In: *ACM Trans. Math. Softw.* 36.3 (July 2009), 13:1–13:23. ISSN: 0098-3500. DOI: 10.1145/1527286.1527287.

[19] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, K. Rupp, B. Smith, S. Zampini, H. Zhang, and H. Zhang. *PETSc Web page*. 2017. URL: http://www.mcs.anl.gov/petsc (visited on 08/01/2017).

[20] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. "Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries." In: *Modern Software Tools for Scientific Computing*. Ed. by E. Arge, A. M. Bruaset, and H. P. Langtangen. Boston, MA: Birkhäuser Boston, 1997, pp. 163–202. ISBN: 978-1-4612-1986-6. DOI: 10.1007/978-1-4612-1986-6_8.

[21] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 1994. DOI: 10.1137/1.9781611971538.

[22] P. Bastian, C. Engwer, D. Göddeke, O. Iliev, O. Ippisch, M. Ohlberger, S. Turek, J. Fahlke, S. Kaulmann, S. Müthing, and D. Ribbrock. "EXA-DUNE: Flexible PDE Solvers, Numerical Methods and Applications." In: *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*. Ed. by L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander. Cham: Springer International Publishing, 2014, pp. 530–541. isbn: 978-3-319-14313-2. doi: 10.1007/978-3-319-14313-2_45.

[23] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist. "Amesos2 and Belos: Direct and Iterative Solvers for Large Sparse Linear Systems." In: *Sci. Program.* 20.3 (July 2012), pp. 241–255. issn: 1058-9244. doi: 10.1155/2012/243875.

[24] N. Bell and M. Garland. "Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors." In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon: ACM, 2009, 18:1–18:11. isbn: 978-1-60558-744-8. doi: 10.1145/1654059.1654078.

[25] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. "Automating the Generation of Composed Linear Algebra Kernels." In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon: ACM, 2009, 59:1–59:12. isbn: 978-1-60558-744-8. doi: 10.1145/1654059.1654119.

[26] M. Besta, F. Marending, E. Solomonik, and T. Hoefler. "SlimSell: A Vectorizable Graph Representation for Breadth-First Search." In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2017, pp. 32–41. doi: 10.1109/IPDPS.2017.93.

[27] O. Bhardwaj, Y. Ineichen, C. Bekas, and A. Curioni. "Highly Scalable Linear Time Estimation of Spectrograms - A Tool for Very Large Scale Data Analysis." Poster at the 2013 ACM/IEEE International Conference on High Performance Computing Networking, Storage and Analysis. 2013. url: http://sc13.supercomputing.org/sites/default/files/PostersArchive/post255.html (visited on 02/13/2017).

[28] M. Blatt, A. Burchardt, A. Dedner, C. Engwer, J. Fahlke, B. Flemisch, C. Gersbacher, C. Gräser, F. Gruber, C. Grüninger, D. Kempf, R. Klöfkorn, T. Malkmus, S. Müthing, M. Nolte, M. Piatkowski, and O. Sander. "The Distributed and Unified Numerics Environment, Version 2.4." In: *Archive of Numerical Software* 4.100 (2016), pp. 13–29. ISSN: 2197-8263. DOI: 10.11588/ans.2016.100.26526.

[29] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. "Cilk: An Efficient Multithreaded Runtime System." In: *SIGPLAN Not.* 30.8 (Aug. 1995), pp. 207–216. ISSN: 0362-1340. DOI: 10.1145/209937.209958.

[30] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra. "Matrix Market: a web resource for test matrix collections." In: *Quality of Numerical Software: Assessment and enhancement*. Ed. by R. F. Boisvert. Boston, MA: Springer US, 1997, pp. 125–137. ISBN: 978-1-5041-2940-4. DOI: 10.1007/978-1-5041-2940-4_9.

[31] E. G. Boman, Ü. V. Çatalyürek, C. Chevalier, and K. D. Devine. "The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering and Coloring." In: *Sci. Program.* 20.2 (Apr. 2012), pp. 129–150. ISSN: 1058-9244. DOI: 10.1155/2012/713587.

[32] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications." In: *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. Ed. by IEEE. Pisa, Italy, Feb. 2010. DOI: 10.1109/PDP.2010.67.

[33] A. Buluc, S. Williams, L. Oliker, and J. Demmel. "Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication." In: *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 721–733. ISBN: 978-0-7695-4385-7. DOI: 10.1109/IPDPS.2011.73.

[34] V. C. Cabezas and M. Püschel. "Extending the Roofline Model: Bottleneck Analysis with Microarchitectural Constraints." In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. Oct. 2014, pp. 222–231. DOI: 10.1109/IISWC.2014.6983061.

[35] D. Callahan, J. Cocke, and K. Kennedy. "Estimating interlock and improving balance for pipelined architectures." In: *Journal of Parallel and Distributed Computing* 5.4 (1988), pp. 334–358. ISSN: 0743-7315. DOI: 10.1016/0743-7315(88)90002-0.

[36] H. Carter Edwards, C. R. Trott, and D. Sunderland. "Kokkos." In: *J. Parallel Distrib. Comput.* 74.12 (Dec. 2014), pp. 3202–3216. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003.

[37] A. H. Castro Neto, F. Guinea, N. M. R. Peres, K. S. Novoselov, and A. K. Geim. "The electronic properties of graphene." In: *Rev. Mod. Phys.* 81 (1 Jan. 2009), pp. 109–162. DOI: 10.1103/RevModPhys.81.109.

[38] U. Catalyurek and C. Aykanat. "Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication." In: *IEEE Trans. Parallel Distrib. Syst.* 10.7 (July 1999), pp. 673–693. ISSN: 1045-9219. DOI: 10.1109/71.780863.

[39] C. Chevalier and F. Pellegrini. "PT-Scotch: A Tool for Efficient Parallel Graph Ordering." In: *Parallel Comput.* 34.6-8 (July 2008), pp. 318–331. ISSN: 0167-8191. DOI: 10.1016/j.parco.2007.12.001.

[40] J. W. Choi, A. Singh, and R. W. Vuduc. "Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs." In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '10. Bangalore, India: ACM, 2010, pp. 115–126. ISBN: 978-1-60558-877-3. DOI: 10.1145/1693453.1693471.

[41] E. Chow and A. Patel. "Fine-Grained Parallel Incomplete LU Factorization." In: *SIAM Journal on Scientific Computing* 37.2 (2015), pp. C169–C193. DOI: 10.1137/140968896.

[42] *Cscs: Piz Daint*. 2017. URL: http://www.cscs.ch/computers/piz_daint/index.html (visited on 02/18/2017).

[43] *Cscs: Piz Daint & Piz Dora*. 2017. URL: http://www.cscs.ch/computers/piz_daint_piz_dora/index.html (visited on 02/18/2017).

[44] *cuBLAS*. 2017. URL: http://docs.nvidia.com/cuda/cublas (visited on 03/07/2017).

[45] *CUDA C Programming Guide*. 2016. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide (visited on 07/13/2016).

[46] J. Cullum and W. E. Donath. "A block Lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse, real symmetric matrices." In: *1974 IEEE Conference on Decision and Control including the 13th Symposium on Adaptive Processes*. Nov. 1974, pp. 505–509. DOI: 10.1109/CDC.1974.270490.

[47] *cuSPARSE*. 2016. URL: http://docs.nvidia.com/cuda/cusparse (visited on 09/15/2016).

[48] E. Cuthill and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices." In: *Proceedings of the 1969 24th National Conference*. ACM '69. New York, NY, USA: ACM, 1969, pp. 157–172. DOI: 10.1145/800195.805928.

[49] M. Daga and J. L. Greathouse. "Structural Agnostic SpMV: Adapting CSR-Adaptive for Irregular Matrices." In: *High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on*. Dec. 2015, pp. 64–74. DOI: 10.1109/HiPC.2015.55.

[50] E. R. Davidson. "The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices." In: *Journal of Computational Physics* 17.1 (1975), pp. 87–94. ISSN: 0021-9991. DOI: 10.1016/0021-9991(75)90065-0.

[51] T. A. Davis and Y. Hu. "The University of Florida Sparse Matrix Collection." In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 1:1–1:25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

[52] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. "GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models." In: *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P$\hat{3}$MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers*. Ed. by M. Taufer, B. Mohr, and J. M. Kunkel. Cham: Springer International Publishing, 2016, pp. 489–507. ISBN: 978-3-319-46079-6. DOI: 10.1007/978-3-319-46079-6_34.

[53] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. "Avoiding Communication in Sparse Matrix Computations." In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. Apr. 2008, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536305.

[54] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. "Communication-optimal Parallel and Sequential QR and LU Factorizations." In: *SIAM Journal on Scientific Computing* 34.1 (2012), A206–A239. DOI: 10.1137/080731992.

[55] A. Denis. "POSTER: a Generic Framework for Asynchronous Progression and Multithreaded Communications." In: *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*. Sept. 2014, pp. 276–277. DOI: 10.1109/CLUSTER.2014.6968752.

[56] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek. "Parallel Hypergraph Partitioning for Scientific Computing." In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. Apr. 2006. DOI: 10.1109/IPDPS.2006.1639359.

[57] E. Di Napoli, E. Polizzi, and Y. Saad. "Efficient estimation of eigenvalue counts in an interval." In: *Numerical Linear Algebra with Applications* 23.4 (2016). nla.2048, pp. 674–692. ISSN: 1099-1506. DOI: 10.1002/nla.2048.

[58] C. Ding and K. Kennedy. "Improving Effective Bandwidth Through Compiler Enhancement of Global Cache Reuse." In: *J. Parallel Distrib. Comput.* 64.1 (Jan. 2004), pp. 108–134. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2003.09.005.

[59] D. S. Dodson, R. G. Grimes, and J. G. Lewis. "Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms." In: *ACM Trans. Math. Softw.* 17.2 (June 1991), pp. 253–263. ISSN: 0098-3500. DOI: 10.1145/108556.108577.

[60] J. Dongarra, M. A. Heroux, and P. Luszczek. "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems." In: *The International Journal of High Performance Computing Applications* 30.1 (2016), pp. 3–10. DOI: 10.1177/1094342015593158.

[61] A. Dubrulle, R. S. Martin, and J. H. Wilkinson. "The Implicit QL Algorithm." In: *Handbook for Automatic Computation: Volume II: Linear Algebra*. Ed. by F. L. Bauer, A. S. Householder, F. W. J. Olver, H. Rutishauser, K. Samelson, and E. Stiefel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1971, pp. 241–248. ISBN: 978-3-642-86940-2. DOI: 10.1007/978-3-642-86940-2_15.

[62] I. S. Duff, M. Marrone, G. Radicati, and C. Vittoli. "Level 3 Basic Linear Algebra Subprograms for Sparse Matrices: A User-level Interface." In: *ACM Trans. Math. Softw.* 23.3 (Sept. 1997), pp. 379–401. ISSN: 0098-3500. DOI: 10.1145/275323.275327.

[63] D. Ernst. "Performance Engineering for Block Vector Matrix Multiplications on GPUs." MA thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2017.

[64] *ESSEX-II - Equipping Sparse Solvers for Exascale*. 2017. URL: https://blogs.fau.de/essex/ (visited on 08/06/2017).

[65] H.-r. Fang and Y. Saad. "A Filtered Lanczos Procedure for Extreme and Interior Eigenvalue Problems." In: *SIAM Journal on Scientific Computing* 34.4 (2012), A2220–A2246. DOI: 10.1137/110836535.

[66] *FAQs | CilkPlus*. 2017. URL: https://www.cilkplus.org/faq/24 (visited on 08/06/2017).

[67] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska. "Optimizing CUDA code by kernel fusion: application on BLAS." In: *The Journal of Supercomputing* 71.10 (2015), pp. 3934–3957. ISSN: 1573-0484. DOI: 10.1007/s11227-015-1483-z.

[68] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo. "Sparse Matrix-Vector Multiplication on GPGPUs." In: *ACM Trans. Math. Softw.* 43.4 (Jan. 2017), 30:1–30:49. ISSN: 0098-3500. DOI: 10.1145/3017994.

[69] D. R. Fokkema, G. L. G. Sleijpen, and H. A. van der Vorst. "Jacobi–Davidson Style QR and QZ Algorithms for the Reduction of Matrix Pencils." In: *SIAM J. Sci. Comp.* 20.1 (Jan. 1998), pp. 94–125. ISSN: 1064-8275. DOI: 10.1137/S1064827596300073.

[70] D. Foley and J. Danskin. "Ultra-Performance Pascal GPU and NVLink Interconnect." In: *IEEE Micro* 37.2 (Mar. 2017), pp. 7–17. ISSN: 0272-1732. DOI: 10.1109/MM.2017.37.

[71] M. Garland. "Sparse Matrix Computations on Manycore GPU's." In: *Proceedings of the 45th Annual Design Automation Conference*. DAC '08. Anaheim, California: ACM, 2008, pp. 2–6. ISBN: 978-1-60558-115-6. DOI: 10.1145/1391469.1391473.

[72] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen. "ColPack: Software for Graph Coloring and Related Problems in Scientific Computing." In: *ACM Trans. Math. Softw.* 40.1 (Oct. 2013), 1:1–1:31. ISSN: 0098-3500. DOI: 10.1145/2513109.2513110.

[73] A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall Professional Technical Reference, 1981. ISBN: 0131652745.

[74] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose. "Hiding Global Communication Latency in the GMRES Algorithm on Massively Parallel Machines." In: *SIAM Journal on Scientific Computing* 35.1 (2013), pp. C48–C71. DOI: 10.1137/12086563X.

[75] G. H. Golub and C. F. Van Loan. *Matrix Computations (4th Ed.)* Baltimore, MD, USA: Johns Hopkins University Press, 2013. ISBN: 1-4214-0794-9.

[76] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. "Performance evaluation of the sparse matrix-vector multiplication on modern architectures." In: *The Journal of Supercomputing* 50.1 (2009), pp. 36–77. ISSN: 1573-0484. DOI: 10.1007/s11227-008-0251-8.

[77] J. L. Greathouse and M. Daga. "Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. New Orleans, Louisana: IEEE Press, 2014, pp. 769–780. ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.68.

[78] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. "Towards Realistic Performance Bounds for Implicit CFD Codes." In: *Proceedings of Parallel CFD'99*. Elsevier, 1999, pp. 233–240. DOI: 10.1016/B978-044482851-4.50030-X.

[79] G. Hager, J. Treibig, J. Habich, and G. Wellein. "Exploring performance and power properties of modern multi-core chips via simple machine models." In: *Concurrency and Computation: Practice and Experience* 28.2 (2016), pp. 189–210. ISSN: 1532-0634. DOI: 10.1002/cpe.3180.

[80] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2010. ISBN: 9781439811924.

[81] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. "LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16. Salt

Lake City, Utah: IEEE Press, 2016, 84:1–84:11. ISBN: 978-1-4673-8815-3. DOI: 10.1109/SC.2016.83.

[82] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. "An overview of the Trilinos project." In: *ACM Trans. Math. Softw.* 31.3 (2005), pp. 397–423. ISSN: 0098-3500. DOI: http://doi.acm.org/10.1145/1089014.1089021.

[83] R. W. Hockney and I. J. Curington. "f1/2: A parameter to characterize memory and communication bottlenecks." In: *Parallel Computing* 10.3 (1989), pp. 277–286. ISSN: 0167-8191. DOI: 10.1016/0167-8191(89)90100-2.

[84] T. Hoefler and R. Belli. "Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: ACM, 2015, 73:1–73:12. ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807644.

[85] J. Hofmann, G. Hager, G. Wellein, and D. Fey. "An Analysis of Core- and Chip-Level Architectural Features in Four Generations of Intel Server Processors." In: *High Performance Computing: 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017, Proceedings*. Ed. by J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes. Cham: Springer International Publishing, 2017, pp. 294–314. ISBN: 978-3-319-58667-0. DOI: 10.1007/978-3-319-58667-0_16.

[86] *HPCG*. 2017. URL: http://www.hpcg-benchmark.org/ (visited on 08/06/2017).

[87] *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. 2017. URL: http://www.netlib.org/benchmark/hpl/ (visited on 02/22/2017).

[88] A. Ilic, F. Pratas, and L. Sousa. "Cache-aware Roofline model: Upgrading the loft." In: *IEEE Computer Architecture Letters* 13.1 (Jan. 2014), pp. 21–24. ISSN: 1556-6056. DOI: 10.1109/L-CA.2013.6.

[89] E.-J. Im, K. Yelick, and R. Vuduc. "Sparsity: Optimization Framework for Sparse Matrix Kernels." In: *Int. J. High Perform. Comput. Appl.* 18.1 (Feb. 2004), pp. 135–158. ISSN: 1094-3420. DOI: 10.1177/1094342004041296.

[90] *Intel Math Kernel Library*. 2016. URL: https://software.intel.com/en-us/intel-mkl (visited on 09/13/2016).

[91] *Intel® Threading Building Blocks, OpenMP, or native threads? | Intel® Software*. 2017. URL: https://software.intel.com/en-us/intel-threading-building-blocks-openmp-or-native-threads (visited on 08/06/2017).

[92] *JCAHPC: Joint Center for Advanced HPC*. 2017. URL: http://jcahpc.jp/eng/ofp_intro.html (visited on 02/18/2017).

[93] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. "Introduction to the Cell multiprocessor." In: *IBM Journal of Research and Development* 49.4.5 (July 2005), pp. 589–604. ISSN: 0018-8646. DOI: 10.1147/rd.494.0589.

[94] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali. "Parallelization of Reordering Algorithms for Bandwidth and Wavefront Reduction." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. New Orleans, Louisana: IEEE Press, 2014, pp. 921–932. ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.80.

[95] K. Kennedy and K. S. McKinley. "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution." In: *Languages and Compilers for Parallel Computing: 6th International Workshop Portland, Oregon, USA, August 12–14, 1993 Proceedings*. Ed. by U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 301–320. ISBN: 978-3-540-48308-3. DOI: 10.1007/3-540-57659-2_18.

[96] *Kepler GK110 Architecture Whitepaper*. NVIDIA. 2016. URL: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (visited on 11/20/2016).

[97] *Kepler Tuning Guide*. NVIDIA. 2016. URL: http://docs.nvidia.com/cuda/kepler-tuning-guide (visited on 11/20/2016).

[98] D. R. Kincaid, T. C. Oppe, and D. M. Young. *ITPACKV 2D User's Guide*. Report CNA-232. The University of Texas at Austin, May 1989.

[99] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris. "CSX: An Extended Compression Format for Spmv on Shared Memory Systems." In: *SIGPLAN Not.* 46.8 (Feb. 2011), pp. 247–256. ISSN: 0362-1340. DOI: 10.1145/2038037.1941587.

[100] M. Kreutzer, A. Pieper, G. Hager, G. Wellein, A. Alvermann, and H. Fehske. "Performance Engineering of the Kernel Polynomial Method on Large-Scale CPU-GPU Systems." In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. May 2015, pp. 417–426. DOI: 10.1109/IPDPS.2015.76.

[101] M. Kreutzer. "GHOST: General, Hybrid, and Optimized Sparse Toolkit." Poster presented at the PhD forum at ISC High Performance 2016. 2016. URL: http://www.isc-hpc.com/isc16_ap/presentationdetails.htm?t=presentation&o=999&a=select&ra=personendetails (visited on 03/28/2017).

[102] M. Kreutzer, G. Hager, and G. Wellein. "A unified sparse matrix storage format for heterogeneous systems." Poster presented at the Early Research Showcase track at the 2013 ACM/IEEE International Conference on High Performance Computing Networking, Storage and Analysis. 2013. URL: https://blogs.fau.de/essex/files/2012/11/sc13_poster-final.pdf (visited on 02/13/2017).

[103] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop. "Sparse Matrix-Vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation." In: *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IPDPSW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1696–1702. ISBN: 978-0-7695-4676-6. DOI: 10.1109/IPDPSW.2012.211.

[104] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop. "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units." In: *SIAM Journal on Scientific Computing* 36.5 (2014), pp. C401–C423. DOI: 10.1137/130930352.

[105] M. Kreutzer, A. Pieper, A. Alvermann, H. Fehske, G. Hager, G. Wellein, and A. R. Bishop. "Efficient Large-Scale Sparse Eigenvalue Computations on Heterogeneous Hardware." Poster at the 2015 ACM/IEEE International Conference on High Performance Computing Networking, Storage and Analysis. 2015. URL:

http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post205.html (visited on 02/13/2017).

[106] M. Kreutzer, J. Thies, A. Pieper, A. Alvermann, M. Galgon, M. Röhrig-Zöllner, F. Shahzad, A. Basermann, A. R. Bishop, H. Fehske, G. Hager, B. Lang, and G. Wellein. "Performance Engineering and Energy Efficiency of Building Blocks for Large, Sparse Eigenvalue Computations on Heterogeneous Supercomputers." In: *Software for Exascale Computing - SPPEXA 2013-2015*. Ed. by H.-J. Bungartz, P. Neumann, and W. E. Nagel. Cham: Springer International Publishing, 2016, pp. 317–338. ISBN: 978-3-319-40528-5. DOI: 10.1007/978-3-319-40528-5_14.

[107] M. Kreutzer, J. Thies, M. Röhrig-Zöllner, A. Pieper, F. Shahzad, M. Galgon, A. Basermann, H. Fehske, G. Hager, and G. Wellein. "GHOST: Building Blocks for High Performance Sparse Linear Algebra on Heterogeneous Systems." In: *International Journal of Parallel Programming* (2016), pp. 1–27. ISSN: 1573-7640. DOI: 10.1007/s10766-016-0464-z.

[108] H. Kung. "Memory Requirements for Balanced Computer Architectures." In: *Journal of Complexity* 1.1 (1985), pp. 147–157. ISSN: 0885-064X. DOI: 10.1016/0885-064X(85)90026-3.

[109] *LAMA*. 2017. URL: https://www.libama.org/ (visited on 03/01/2017).

[110] C. Lameter. "NUMA (Non-Uniform Memory Access): An Overview." In: *Queue* 11.7 (July 2013), 40:40–40:51. ISSN: 1542-7730. DOI: 10.1145/2508834.2513149.

[111] C. Lanczos. "An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators." In: *Journal of Research of the National Bureau of Standards* 45 (1950), pp. 255–282.

[112] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt. "An Investigation of Unified Memory Access Performance in CUDA." In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2014, pp. 1–6. DOI: 10.1109/HPEC.2014.7040988.

[113] D. Langr and P. Tvrdik. "Evaluation Criteria for Sparse Matrix Storage Formats." In: *Parallel and Distributed Systems, IEEE Transactions on* PP.99 (2015), pp. 1–1. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2401575.

[114] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984. ISBN: 0-13-746975-6.

[115] W. Li, G. Jin, X. Cui, and S. See. "An Evaluation of Unified Memory Technology on NVIDIA GPUs." In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. May 2015, pp. 1092–1098. DOI: 10.1109/CCGrid.2015.105.

[116] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture." In: *Micro, IEEE* 28.2 (Mar. 2008), pp. 39–55. ISSN: 0272-1732. DOI: 10.1109/MM.2008.31.

[117] W. Liu and B. Vinter. "CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication." In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. Newport Beach, California, USA: ACM, 2015, pp. 339–350. ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751209.

[118] X. Liu, E. Chow, K. Vaidyanathan, and M. Smelyanskiy. "Improving the Performance of Dynamical Simulations Via Multiple Right-Hand Sides." In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. May 2012, pp. 36–47. DOI: 10.1109/IPDPS.2012.14.

[119] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. "Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors." In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. Eugene, Oregon, USA: ACM, 2013, pp. 273–282. ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465013.

[120] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera. "3DyRM: a dynamic roofline model including memory latency information." In: *The Journal of Supercomputing* 70.2 (2014), pp. 696–708. ISSN: 1573-0484. DOI: 10.1007/s11227-014-1163-4.

[121] *LRZ: SuperMUC Petascale System*. 2017. URL: https://www.lrz.de/services/compute/supermuc/ (visited on 02/18/2017).

[122] N. K. Madsen, G. H. Rodrigue, and J. I. Karush. "Matrix multiplication by diagonals on a vector/parallel processor." In: *Information Processing Letters* 5.2 (1976), pp. 41–45. ISSN: 0020-0190. DOI: http://dx.doi.org/10.1016/0020-0190(76)90077-6.

[123] M. Martone. "Efficient multithreaded untransposed, transposed or symmetric sparse matrix–vector multiplication with the Recursive Sparse Blocks format." In: *Parallel Computing* 40.7 (2014). 7th Workshop on Parallel Matrix Algorithms and Applications, pp. 251–270. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2014.03.008.

[124] *Matrix Market: File Formats*. 2016. URL: http://math.nist.gov/MatrixMarket/formats.html (visited on 12/12/2016).

[125] J. D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers." In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

[126] V. Minden, B. Smith, and M. G. Knepley. "Preliminary Implementation of PETSc Using GPUs." In: *GPU Solutions to Multi-scale Problems in Science and Engineering*. Ed. by D. A. Yuen, L. Wang, X. Chi, L. Johnsson, W. Ge, and Y. Shi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 131–140. ISBN: 978-3-642-16405-7. DOI: 10.1007/978-3-642-16405-7_7.

[127] P. Mironowicz, A. Dziekonski, and M. Mrozowski. "A Task-Scheduling Approach for Efficient Sparse Symmetric Matrix-Vector Multiplication on a GPU." In: *SIAM Journal on Scientific Computing* 37.6 (2015), pp. C643–C666. DOI: 10.1137/14097135X.

[128] A. Monakov, A. Lokhmotov, and A. Avetisyan. "Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures." In: *High Performance Embedded Architectures and Compilers*. Ed. by Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell. Vol. 5952. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 111–125. ISBN: 978-3-642-11514-1. DOI: 10.1007/978-3-642-11515-8_10.

[129] G. E. Moore. "Cramming More Components onto Integrated Circuits." In: *Electronics* 38.8 (Apr. 1965), pp. 114–117. ISSN: 0018-9219. DOI: 10.1109/jproc.1998.658762.

[130] G. E. Moore. "Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]" In: *IEEE Solid-State Circuits Society Newsletter* 20.3 (Sept. 2006), pp. 36–37. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4804410.

[131] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.1*. 2015. URL: http://www.mpi-forum.org (visited on 06/29/2017).

[132] T. Nelson, G. Belter, J. G. Siek, E. Jessup, and B. Norris. "Reliable Generation of High-Performance Matrix Algebra." In: *ACM Trans. Math. Softw.* 41.3 (June 2015), 18:1–18:27. ISSN: 0098-3500. DOI: 10.1145/2629698.

[133] J. Nickolls, I. Buck, M. Garland, and K. Skadron. "Scalable Parallel Programming with CUDA." In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500.

[134] *NVIDIA Profiler*. NVIDIA. 2016. URL: http://docs.nvidia.com/cuda/profiler-users-guide (visited on 07/20/2016).

[135] D. P. O'Leary. "The Block Conjugate Gradient Algorithm and Related Methods." In: *Linear Algebra and its Applications* 29 (1980), pp. 293–322. ISSN: 0024-3795. DOI: http://dx.doi.org/10.1016/0024-3795(80)90247-5.

[136] *PARALUTION - The Library for Iterative Sparse Methods on CPU and GPU*. 2017. URL: http://www.paralution.com/ (visited on 03/01/2017).

[137] A. Pieper, M. Kreutzer, A. Alvermann, M. Galgon, H. Fehske, G. Hager, B. Lang, and G. Wellein. "High-performance implementation of Chebyshev filter diagonalization for interior eigenvalue computations." In: *Journal of Computational Physics* 325 (2016), pp. 226–243. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2016.08.027.

[138] A. Pinar and M. T. Heath. "Improving Performance of Sparse Matrix-vector Multiplication." In: *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. SC '99. Portland, Oregon, USA: ACM, 1999. ISBN: 1-58113-091-0. DOI: 10.1145/331532.331562.

[139] R. Rabenseifner, G. Hager, and G. Jost. "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes." In: *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. Feb. 2009, pp. 427–436. DOI: 10.1109/PDP.2009.43.

[140] J. K. Reid and J. A. Scott. "Reducing the Total Bandwidth of a Sparse Unsymmetric Matrix." In: *SIAM Journal on Matrix Analysis and Applications* 28.3 (2006), pp. 805–821. DOI: 10.1137/050629938.

[141] J. Reinders. *Intel Threading Building Blocks*. 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007. ISBN: 9780596514808.

[142] M. Röhrig-Zöllner, J. Thies, M. Kreutzer, A. Alvermann, A. Pieper, A. Basermann, G. Hager, G. Wellein, and H. Fehske. "Increasing the Performance of the Jacobi–Davidson Method by Blocking." In: *SIAM Journal on Scientific Computing* 37.6 (2015), pp. C697–C722. DOI: 10.1137/140976017.

[143] *RRZE - Emmy Cluter*. 2017. URL: https://www.rrze.fau.de/dienste/arbeiten-rechnen/hpc/systeme/emmy-cluster.shtml (visited on 02/20/2017).

[144] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, and S. Selberherr. "ViennaCL—Linear Algebra Library for Multi- and Many-Core Architectures." In: *SIAM Journal on Scientific Computing* 38.5 (2016), S412–S439. DOI: 10.1137/15M1026419.

[145] K. Rupp, J. Weinbub, A. Jüngel, and T. Grasser. "Pipelined Iterative Solvers with Kernel Fusion for Graphics Processing Units." In: *ACM Trans. Math. Softw.* 43.2 (Aug. 2016), 11:1–11:27. ISSN: 0098-3500. DOI: 10.1145/2907944.

[146] F. P. Russell, M. R. Mellor, P. H. J. Kelly, and O. Beckmann. "DESOLA: An Active Linear Algebra Library Using Delayed Evaluation and Runtime Code Generation." In: *Sci. Comput. Program.* 76.4 (Apr. 2011), pp. 227–242. ISSN: 0167-6423. DOI: 10.1016/j.scico.2008.06.002.

[147] Y. Saad. "Krylov Subspace Methods for Solving Large Unsymmetric Linear Systems." In: *Mathematics of Computation* 37.155 (1981), pp. 105–126. ISSN: 00255718, 10886842. DOI: 10.2307/2007504.

[148] Y. Saad. *Numerical Methods for Large Eigenvalue Problems*. Society for Industrial and Applied Mathematics, 2011. DOI: 10.1137/1.9781611970739.

[149] Y. Saad. "Krylov Subspace Methods on Supercomputers." In: *SIAM Journal on Scientific and Statistical Computing* 10.6 (1989), pp. 1200–1232. DOI: 10.1137/0910073.

[150] E. Saule, K. Kaya, and Ü. V. Çatalyürek. "Parallel Processing and Applied Mathematics: 10th International Conference, PPAM 2013, Warsaw, Poland, September 8-11, 2013, Revised Selected Papers, Part I." In: ed. by R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. Chap. Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi, pp. 559–570. ISBN: 978-3-642-55224-3. DOI: 10.1007/978-3-642-55224-3_52.

[151] *sched_setaffinity(2) - Linux man page*. 2017. (Visited on 07/31/2017).

[152] G. Schubert, H. Fehske, G. Hager, and G. Wellein. "Hybrid-Parallel Sparse Matrix-Vector Multiplication with Explicit Communication Overlap on Current Multicore-Based Systems." In: *Parallel Processing Letters* 21.03 (2011), pp. 339–358. DOI: 10.1142/S0129626411000254.

[153] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. "Larrabee: A Many-core x86 Architecture for Visual Computing." In: *ACM SIGGRAPH 2008 Papers*. SIGGRAPH '08. Los Angeles, California: ACM, 2008, 18:1–18:15. ISBN: 978-1-4503-0112-1. DOI: 10.1145/1399504.1360617.

[154] R. Silver, H. Roeder, A. Voter, and J. Kress. "Kernel Polynomial Approximations for Densities of States and Spectral Functions." In: *J. Comput. Phys.* 124.1 (Mar. 1996), pp. 115–130. ISSN: 0021-9991. DOI: 10.1006/jcph.1996.0048.

[155] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. "Knights Landing: Second-Generation Intel Xeon Phi Product." In: *IEEE Micro* 36.2 (Mar. 2016), pp. 34–46. ISSN: 0272-1732. DOI: 10.1109/MM.2016.25.

[156] *SpMP: sparse matrix pre-processing library*. 2016. URL: https://github.com/IntelLabs/SpMP (visited on 12/12/2016).

[157] A. Stathopoulos and J. R. McCombs. "Nearly Optimal Preconditioned Methods for Hermitian Eigenproblems Under Limited Memory. Part II: Seeking Many Eigenvalues." In: *SIAM J. Sci. Comp.* 29.5 (Jan. 2007), pp. 2162–2188. DOI: 10.1137/060661910.

[158] A. Stathopoulos and J. R. McCombs. "PRIMME: PReconditioned Iterative MultiMethod Eigensolver–Methods and Software Description." In: *ACM Trans. Math. Softw.* 37.2 (Apr. 2010), pp. 1–30. ISSN: 0098-3500. DOI: 10.1145/1731022.1731031.

[159] A. Stathopoulos. "Nearly Optimal Preconditioned Methods for Hermitian Eigenproblems under Limited Memory. Part I: Seeking One Eigenvalue." In: *SIAM Journal on Scientific Computing* 29.2 (2007), pp. 481–514. DOI: 10.1137/050631574.

[160] A. Stathopoulos and K. Wu. "A Block Orthogonalization Procedure with Constant Synchronization Requirements." In: *SIAM Journal on Scientific Computing* 23.6 (2002), pp. 2165–2182. DOI: 10.1137/S1064827500370883.

[161] H. Stengel, J. Treibig, G. Hager, and G. Wellein. "Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model." In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. Newport Beach, California, USA: ACM, 2015, pp. 207–216. ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751240.

[162] S. Tabik, G. Ortega, and E. M. Garzón. "Performance evaluation of kernel fusion BLAS routines on the GPU: iterative solvers as case study." In: *The Journal of Supercomputing* 70.2 (2014), pp. 577–587. ISSN: 1573-0484. DOI: 10.1007/s11227-014-1102-4.

[163] J. Thies, M. Galgon, F. Shahzad, A. Alvermann, M. Kreutzer, A. Pieper, M. Röhrig-Zöllner, A. Basermann, H. Fehske, G. Hager, B. Lang, and G. Wellein. "Towards an Exascale Enabled Sparse Solver Repository." In: *Software for Exascale Computing - SPPEXA 2013-2015*. Ed. by H.-J. Bungartz, P. Neumann, and W. E. Nagel. Cham: Springer International Publishing, 2016, pp. 295–316. ISBN: 978-3-319-40528-5. DOI: 10.1007/978-3-319-40528-5_13.

[164] A. Thomasian and P. F. Bay. "Analytic Queueing Network Models for Parallel Processing of Task Systems." In: *IEEE Transactions on Computers* C-35.12 (Dec. 1986), pp. 1045–1054. ISSN: 0018-9340. DOI: 10.1109/TC.1986.1676712.

[165] M. M. Tikir, L. Carrington, E. Strohmaier, and A. Snavely. "A Genetic Algorithms Approach to Modeling the Performance of Memory-bound Computations." In: *Proceedings of the 2007 ACM/IEEE Con-*

ference on Supercomputing. SC '07. Reno, Nevada: ACM, 2007, 47:1–47:12. ISBN: 978-1-59593-764-3. DOI: 10.1145/1362622.1362686.

[166] S. Tomov, J. Dongarra, and M. Baboulin. "Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems." In: *Parallel Computing* 36.5-6 (June 2010), pp. 232–240. ISSN: 0167-8191. DOI: 10.1016/j.parco.2009.12.005.

[167] *TOP500 Supercomputer Sites*. 2017. URL: http://www.top500.org (visited on 06/23/2017).

[168] J. Treibig and G. Hager. "Introducing a Performance Model for Bandwidth-Limited Loop Kernels." In: *Parallel Processing and Applied Mathematics: 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009. Revised Selected Papers, Part I*. Ed. by R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 615–624. ISBN: 978-3-642-14390-8. DOI: 10.1007/978-3-642-14390-8_64.

[169] J. Treibig, G. Hager, and G. Wellein. "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments." In: *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*. ICPPW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 207–216. ISBN: 978-0-7695-4157-0. DOI: 10.1109/ICPPW.2010.38.

[170] J. Treibig, G. Hager, and G. Wellein. "Performance Patterns and Hardware Metrics on Modern Multicore Processors: Best Practices for Performance Engineering." In: *Euro-Par 2012: Parallel Processing Workshops: BDMC, CGWS, HeteroPar, HiBB, OMHI, Paraphrase, PROPER, Resilience, UCHPC, VHPC, Rhodes Islands, Greece, August 27-31, 2012. Revised Selected Papers*. Ed. by I. Caragiannis, M. Alexander, R. M. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. L. Scott, and J. Weidendorfer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 451–460. ISBN: 978-3-642-36949-0. DOI: 10.1007/978-3-642-36949-0_50.

[171] T. Vijayaraghavany, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan. "Design and Analysis of an APU for Exascale

Computing." In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2017, pp. 85–96. DOI: 10.1109/HPCA.2017.42.

[172] V. Volkov and J. W. Demmel. "Benchmarking GPUs to Tune Dense Linear Algebra." In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008, 31:1–31:11. ISBN: 978-1-4244-2835-9. DOI: 10.1109/SC.2008.5214359.

[173] R. W. Vuduc and H.-J. Moon. "Fast Sparse Matrix-vector Multiplication by Exploiting Variable Block Structure." In: *Proceedings of the First International Conference on High Performance Computing and Communications*. HPCC'05. Sorrento, Italy: Springer-Verlag, 2005, pp. 807–816. DOI: 10.1007/11557654_91.

[174] R. W. Vuduc. "Automatic Performance Tuning of Sparse Matrix Kernels." AAI3121741. PhD thesis. 2003.

[175] M. Wahib and N. Maruyama. "Scalable Kernel Fusion for Memory-bound GPU Applications." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. New Orleans, Louisana: IEEE Press, 2014, pp. 191–202. ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.21.

[176] A. Weisse, G. Wellein, A. Alvermann, and H. Fehske. "The kernel polynomial method." In: *Rev. Mod. Phys.* 78 (1 Mar. 2006), pp. 275–306. DOI: 10.1103/RevModPhys.78.275.

[177] J. H. Wilkinson and C. Reinsch, eds. *Linear Algebra*. Vol. II. Handbook for Automatic Computation, Editors: F. L. Bauer, A. S. Householder, F. W. J. Olver, H. Rutishauser, K. Samelson and E. Stiefel. Berlin, Germany / Heidelberg, Germany / London, UK / etc.: Springer-Verlag, 1971, pp. viii + 439. ISBN: 978-3-540-05414-6.

[178] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. "Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms." In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. SC '07. Reno, Nevada: ACM, 2007, 38:1–38:12. ISBN: 978-1-59593-764-3. DOI: 10.1145/1362622.1362674.

[179] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures." In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

[180] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. "Asynchronous MPI for the Masses." preprint. 2013. URL: http://arxiv.org/abs/1302.4280.

[181] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra. "Improving the Performance of CA-GMRES on Multicores with Multiple GPUs." In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. May 2014, pp. 382–391. DOI: 10.1109/IPDPS.2014.48.

The increasing demand for solving larger and more complex problems in computational science and engineering is a major driving factor to deploy computer systems with ever-advancing performance capabilities. To increase the available performance, modern HPC platforms come with multiple levels of parallelism, complex memory hierarchies, heterogeneous architectures, and extreme scales. To match the need for sustainable and efficient software under these premises, special value has to be attached to the inherent challenges like efficiency on all scales and performance portability across heterogeneous architectures.

This work addresses the development of high-performance scientific software for sparse linear algebra, which is an important field of research and forms the foundation of many applications of computational science and engineering, with a special focus on sparse eigenvalue solvers on current and future supercomputers. Consequent employment of performance models as well as a holistic view on applications, algorithms, and hardware architectures enable the creation of basic computational building blocks, custom compute kernels, and optimized algorithmic formulations with provably high efficiency.

To demonstrate the applicability of the developed software components, full-application performance of selected sparse eigenvalue solvers for real-world problems on some of the world‘s largest supercomputers with completely different hardware architectures – including homogeneous multi-core CPU clusters, GPU-accelerated clusters, and self-hosted many-core CPU clusters – is presented.

ISBN 978-3-96147-103-4

FAU UNIVERSITY PRESS 2018