UNIVERSIDAD JAUME I DE CASTELLÓN
E. S. DE TECNOLOGÍA Y CIENCIAS EXPERIMENTALES

**SPARSE LINEAR SYSTEM SOLVERS
ON GPUS:
PARALLEL PRECONDITIONING,
WORKLOAD BALANCING, AND
COMMUNICATION REDUCTION**

CASTELLÓN DE LA PLANA, MARCH 2019

TESIS DOCTORAL PRESENTADA POR:	GORAN FLEGAR
DIRIGIDA POR:	ENRIQUE S. QUINTANA-ORTÍ
	HARTWIG ANZT

UNIVERSIDAD JAUME I DE CASTELLÓN
E. S. DE TECNOLOGÍA Y CIENCIAS EXPERIMENTALES

**SPARSE LINEAR SYSTEM SOLVERS
ON GPUS:
PARALLEL PRECONDITIONING,
WORKLOAD BALANCING, AND
COMMUNICATION REDUCTION**

GORAN FLEGAR

Abstract

With the breakdown of Dennard scaling during the mid-2000s, and the end of Moore’s law on the horizon, hardware vendors, datacenters, and the high performance computing community are turning their attention towards unconventional hardware in hope of continuing the exponential performance growth of computational capacity. Among the available hardware options, a new generation of graphics processing units (GPUs), designed to support a wide variety of workloads in addition to graphics processing, is achieving the widest adoption. These processors are employed by the majority of today’s most powerful supercomputers to solve the world’s most complex problems in physics simulations, weather forecasting, data analytics, social network analysis, and machine learning, among others. The potential of GPUs for these problems can only be unleashed by developing appropriate software, specifically tuned for the GPU architectures. Fortunately, many algorithms that appear in these applications are constructed out of the same basic building blocks. One example of a heavily-used building block is the solution of large, sparse linear systems, a challenge that is addressed in this thesis.

After a quick overview of the current state-of-the-art methods for the solution of linear systems, this dissertation pays detailed attention to the class of Krylov iterative methods. Instead of deriving new methods, improvements are introduced to components that are already widely used in existing methods, and therein account for a significant fraction of the overall runtime cost. The components are designed for a single GPU, while scaling to multiple GPUs can be achieved by either generalizing the same ideas, or by decomposing the larger problem into multiple independent parts which can leverage the implementations described in this thesis.

The most time-consuming part of a Krylov method is often the matrix-vector product. Two improvements are suggested in this dissertation: one for the widely-used compressed sparse row (CSR) matrix format, and an alternative one for the coordinate (COO) format, which has not yet achieved such ample adoption in numerical linear algebra. The new GPU implementation for the CSR format is specifically tuned for matrices with irregular sparsity patterns and, while experiencing slowdowns of up to 3x compared with the vendor library implementation for regular patterns, it achieves up to 100x speedup for irregular ones. However, the slowdown can be eliminated by using a simple heuristic that selects the superior implementation based on the sparsity pattern of the matrix. The new COO algorithm is suggested as the default matrix-vector product implementation for cases when a specific matrix sparsity pattern is not known in advance. This algorithm achieves 80% higher minimal and 22% higher average performance than the newly developed CSR algorithm on a variety of large matrices arising from real-world applications, making it an ideal default choice for general-purpose libraries.

The second component addressed in this dissertation is preconditioning. It explores the relatively simple class of block-Jacobi preconditioners, and shows that these can significantly increase the robustness and decrease the total runtime of Krylov solvers for a certain class of matrices. Several algorithmic realizations of the preconditioner are evaluated, and the one based on Gauss-Jordan elimination is identified as performance winner in most problem settings. The variant based on the LU factorization can be attractive for problems that converge in few iterations.

In this dissertation, block-Jacobi preconditioning is analyzed further via an initial study of the effects that single and half precision floating-point storage have on this type of preconditioners. The resulting adaptive precision block-Jacobi preconditioner dynamically assigns storage precisions to individual blocks at runtime, taking into account the numerical properties of the blocks. A sequential implementation in a high-level lan-

guage, backed by a theoretical error analysis, shows that this preconditioner reduces the total memory transfer volume, while maintaining the preconditioner quality of a full precision block-Jacobi. A theoretical energy model predicts that the adaptive variant can offer energy savings of around 25% in comparison to the full precision block-Jacobi.

Acknowledging that new algorithms or optimized implementations are only useful for the scientific computing community if they are available as production-ready open source code, the final part of this dissertation presents a possible design of a sparse linear algebra library, which effectively solves the problem of excessive manifoldness of components for the iterative solution of linear systems. These ideas represent the backbone of the open source Ginkgo library, which also includes successful implementations of matrix-vector product algorithms and preconditioners described in this thesis.

Con el final de la ley de escalado de Dennard a mitad de la pasada década, y el fin de la ley de Moore en el horizonte, los vendedores de sistemas hardware, los grandes centros de datos y la comunidad que trabaja en computación de altas prestaciones están fijando su atención en nuevas tecnologías no convencionales, con la esperanza de mantener el crecimiento exponencial de la capacidad computacional. Entre las diferentes opciones hardware disponibles, la nueva generación de procesadores gráficos (o GPUs, del término en inglés *Graphics Processing Units*), diseñados para ejecutar de manera eficiente una gran variedad de aplicaciones además del procesamiento gráfico, está consiguiendo una amplia aceptación. Hoy en día, estos procesadores se emplean en la mayor parte de los supercomputadores más potentes, para resolver problemas enormemente complejos relacionados con simulaciones de fenómenos físicos, predicción climática, análisis de datos, análisis de redes sociales y aprendizaje máquina, entre otros. El potencial de las GPUs para tratar estos problemas solo puede aprovecharse mediante el desarrollo de programas eficientes, específicamente optimizados para este tipo de arquitecturas. Por fortuna, muchos de los algoritmos que aparecen en estas aplicaciones se construyen a partir de un conjunto reducido de bloques básicos. Un ejemplo de bloque básico, comúnmente usado, es la solución de sistemas lineales dispersos de gran dimensión, un reto que se afronta en esta tesis.

Tras una breve revisión del estado del arte en métodos para la resolución de sistemas lineales, esta tesis doctoral presta especial atención a la familia de métodos iterativos de Krylov. Sin embargo, en lugar de intentar derivar nuevos métodos, en este trabajo se introducen mejoras en los componentes que se usan ampliamente en los métodos ya existentes, y que suponen una parte importante de su coste de ejecución total. Los componentes están diseñados para una única GPU, pero escalarlos a un sistema con múltiples aceleradores gráficos puede conseguirse generalizando las mismas ideas, o descomponiendo el problema en múltiples partes independientes que puedan aprovechar las implementaciones descritas en esta tesis.

A menudo, la parte computacionalmente más costosa de los métodos de Krylov es el producto matriz-vector. En esta tesis se sugieren dos mejoras para esta operación: una para el formato matrix-vector CSR (*compressed sparse row*), y otra para el formato alternativo COO (coordinado), que no ha logrado una aceptación tan amplia como el CSR en álgebra lineal numérica. La nueva implementación del formato CSR para GPUs está diseñada para ser especialmente eficiente con matrices con un patrón de dispersidad irregular y, si bien sufre una reducción de rendimiento en un factor 3x comparada con la implementación de las bibliotecas estándar para patrones regulares, también es cierto que ofrece una aceleración de 100x para los patrones irregulares. Además, la merma en las prestaciones puede eliminarse mediante una heurística simple que selecciona la mejor implementación en función del patrón de dispersidad de la matriz. Este algoritmo consigue, como mínimo un 80% y como media un 22% mejor rendimiento medio que el nuevo algoritmo basado en CSR en una evaluación con una variedad de matrices de gran tamaño, que surgen en aplicaciones reales, ofreciendo una muy buena opción por defecto para bibliotecas de propósito general.

El segundo componente que se aborda en esta tesis doctoral es el preconditionado. Nuestro trabajo explora la clase relativamente simple de preconditionadores de Jacobi por bloques, y muestra que estos pueden mejorar la robustez y reducir el tiempo de ejecución de los métodos de Krylov para un determinado tipo de matrices. En este trabajo se evalúan algunas realizaciones del preconditionador, y se identifica una, basada en la eliminación de Gauss-Jordan, como aquella que ofrece mejores prestaciones en la mayor parte de escenarios. La variante

basada en la factorización LU, en cambio, puede ser una buena opción para problemas donde el método de Krylov converge en pocas iteraciones.

En esta tesis doctoral se analizan los preconditionadores de Jacobi por bloques elaborando un estudio en detalle de los efectos que tiene sobre este tipo de preconditionadores el almacenamiento de información en precisión reducida. El preconditionador de Jacobi por bloques resultante, con precisión adaptativa, asigna de manera dinámica la precisión a utilizar en el almacenamiento de bloques individuales en tiempo de ejecución, teniendo en cuenta las propiedades numéricas de los bloques. Una implementación en un lenguaje de alto nivel, complementada por un análisis teórico del error, muestra que este tipo de preconditionador reduce el volumen total de datos transferidos, en tanto que mantiene la calidad de los preconditionadores convencionales con precisión plena.

A modo de reconocimiento de que los nuevos algoritmos o las implementaciones optimizadas solo son útiles para la comunidad científica si están disponibles como código abierto en producción, la parte final de esta tesis presenta un posible diseño de biblioteca de álgebra lineal dispersa, que resuelve el problema de la explosión de componentes de manera efectiva para la resolución iterativa de sistemas lineales dispersos. Estas ideas representan la columna vertebral de la biblioteca de código abierto Ginkgo, que también incluye las implementaciones eficientes de algoritmos para el producto matriz-vector y los preconditionadores introducidos en esta tesis.

I	Prologue	1
1	Introduction	3
1.1	Linear Systems	3
1.2	Sparse Matrices	7
1.3	Preconditioning	9
1.4	Numerical Methods in High Performance Computing	10
1.5	This Work	11
II	Sparse Matrix Formats and Matrix-Vector Product	15
2	Balanced Sparse Matrix-Vector Product for the CSR Matrix Format	17
2.1	Introduction	17
2.2	CSR-Based Formats and Algorithms for SpMV	18
2.3	Balanced SpMV kernel	19
2.3.1	General idea	19
2.3.2	Achieving good performance on GPUs	20
2.3.3	Determining the first row of each segment	20
2.4	Experimental Evaluation	21
2.4.1	Setup and metrics	21
2.4.2	Memory consumption	23
2.4.3	Global comparison	24
2.4.4	Detailed comparison of CSR and CSR-I.	24
2.5	Conclusions	26
3	Balanced Sparse Matrix-Vector Product for the COO Matrix Format	27
3.1	Introduction	27
3.2	Related Work	28
3.2.1	Sparse Matrix Formats	28
3.2.2	SpMV on manycore architectures	28
3.3	Design of the COO SpMV GPU kernel	29
3.3.1	COO SpMV	29
3.3.2	CUDA realization of COO SpMV	30
3.4	Performance Assessment	31
3.4.1	Test matrices	31
3.4.2	Experiment setup	32
3.4.3	Experimental results	32
3.5	Summary and Outlook	36

4	Matrix-Vector Product Algorithms for Individual Streaming Multiprocessors	39
4.1	Introduction	39
4.2	Related Work	40
4.2.1	SpMV on manycore architectures	40
4.2.2	Batched routines	40
4.3	Design of flexible batched SpMV kernels for GPUs	41
4.3.1	Flexible batched SpMV	41
4.3.2	GPU kernel design	41
4.3.3	COO	41
4.3.4	CSR	42
4.3.5	CSR-I	43
4.3.6	ELL	43
4.4	Performance Evaluation	43
4.4.1	Experiment setup	43
4.4.2	Experimental results	45
4.5	Summary and Outlook	47
III	Preconditioning	49
5	Block-Jacobi Preconditioning Using Explicit Inversion	51
5.1	Introduction	51
5.2	Background and Related Work	52
5.2.1	Block-Jacobi preconditioning	52
5.2.2	GJE for matrix inversion	53
5.2.3	GJE with implicit pivoting	53
5.2.4	Batched GPU routines	54
5.3	Design of CUDA Kernels	55
5.3.1	Variable-size batched Gauss-Jordan elimination	55
5.3.2	Data extraction from the sparse coefficient matrix	56
5.3.3	Preconditioner application	57
5.4	Experimental Evaluation	58
5.4.1	Hardware and software framework	58
5.4.2	Batched matrix inversion	58
5.4.3	Block-Jacobi generation	60
5.4.4	Block-Jacobi application	64
5.4.5	Convergence in the context of an iterative solver	64
5.5	Concluding Remarks	65
6	Block-Jacobi Preconditioning Based on Gauss-Huard Decomposition	69
6.1	Introduction	69
6.2	Background and Related Work	70
6.2.1	Block-Jacobi preconditioning	70
6.2.2	Solution of linear systems	70
6.2.3	GH with implicit pivoting	71
6.2.4	Related work on batched routines	72
6.3	Design of CUDA Kernels	72
6.3.1	Variable Size Batched Gauss-Huard decomposition	72
6.3.2	Batched Gauss-Huard application	72
6.3.3	Batched data extraction and insertion	73
6.4	Numerical Experiments	73
6.4.1	Hardware and software framework	73
6.4.2	Performance of BGH	74
6.4.3	Performance of block-Jacobi application	74

6.4.4	Iterative solver analysis	75
6.5	Concluding Remarks	76
7	Block-Jacobi Preconditioning Based on LU Factorization	79
7.1	Introduction	79
7.2	Background and Related Work	80
7.2.1	Block-Jacobi preconditioning	80
7.2.2	Solution of linear systems via the LU factorization	81
7.2.3	Batched solution of small linear systems	81
7.3	Design of CUDA Kernels	82
7.3.1	Batched LU factorization (GETRF)	83
7.3.2	Batched triangular system solves (TRSV)	84
7.3.3	Block-Jacobi preconditioning using batched LU	86
7.4	Numerical Experiments	86
7.4.1	Hardware and software framework	87
7.4.2	Performance of batched factorization routines	87
7.4.3	Performance of batched triangular solves	88
7.4.4	Analysis of block-Jacobi preconditioning for iterative solvers	88
7.5	Concluding Remarks and Future Work	90
IV	Towards Adaptive Precision Methods	95
8	Leveraging Adaptive Precision in Block-Jacobi Preconditioning	97
8.1	Introduction	97
8.2	Reduced Precision Preconditioning in the PCG Method	99
8.2.1	Brief review	99
8.2.2	Orthogonality-Preserving Mixed Precision Preconditioning	99
8.3	Block-Jacobi Preconditioning	101
8.4	Adaptive Precision Block-Jacobi Preconditioning	101
8.5	Rounding Error Analysis	104
8.6	Experimental Analysis	105
8.6.1	Experimental framework	105
8.6.2	Reduced precision preconditioning	105
8.6.3	Energy model	107
8.7	Concluding Remarks and Future Work	109
V	Epilogue	111
9	Into the Great Unknown	113
9.1	Designing Scientific Software for Sparse Computations	113
9.1.1	Matrices	113
9.1.2	Linear Systems	115
9.1.3	Preconditioners	116
9.1.4	Linear Operators — Towards a Generic Interface for Sparse Computations	117
9.1.5	Ginkgo — A High Performance Linear Operator Library	117
9.2	Conclusions and Open Research Directions	119
9.3	Conclusiones y Líneas abiertas de Investigación	121
	List of Figures	128
	List of Tables	129

Acknowledgements

As any work of such magnitude, a thesis cannot be completed by the efforts of one individual alone. While there is room for only one name on the front cover, this section allows the author to mention all the other people whose actions contributed to the creation of this dissertation.

First and foremost, I would like to thank my advisors, Hartwig Anzt and Enrique Quintana-Ortí for taking on the challenge and the risk of guiding a fresh graduate through the maze of academic research: your advice about every aspect of this field was invaluable; our frequent discussions were the most enjoyable part of my job; and you selflessly invested more time than anyone could have asked of you in smoothing my relocation and stay abroad.

This thesis would not be possible without the involvement of my instructors at the University of Zagreb: Neven Krajina, Vedran Novaković, and Sanja Singer, who first introduced me to and provided detailed training in high performance computing.

I am also grateful to my colleagues at Universidad Jaume I: Maria Barreda Vayà, Rocío Carratalá Sáez, Adrián Castelló Gimeno, Sandra Catalán Pallarés, Sergio Iserte Agut, and Andrés Enrique Tomás Domínguez, who would often spend their valuable time as my advisors and translators in dealings with local institutions. I especially want to thank Rocío, for making me feel less of an outsider in a predominantly homogeneous society.

My time at the Karlsruhe Institute of Technology would not be as pleasant, and the Ginkgo library would certainly not be what it is today without my colleagues from my “other home institution”: Terry Cojean, Thomas Grützmacher, Pratik Nayak, and Tobias Ribizel; and our visitors and contributors from the National Taiwan University: Yen-Chen Chen and Yaohung Mike Tsai.

I am deeply grateful to my parents and my family for their help in preserving my emotional well-being during this adventure. I will always remember all the family gatherings you organized in the rare occasions when I was able to come home.

Finally, I thank my beloved Jelena. You have always been my moral and emotional pillar, despite the strain of prolonged physical separation and the fact that other plans had to be put “on hold” while I was pursuing my dreams. This thesis represents the end of this chapter of my life, and the beginning of a new one where we can start realizing new dreams together.

*To my friend . . .
right till the end.*

Part I

Prologue

The solution of linear systems is one of the most fundamental problems in computer science, with application areas ranging from physical simulations to computer graphics, social network analysis, and artificial intelligence [11, 26]. It is also a key component of many methods for high order linear algebra problems, such as the eigenvalue problem [12, 25]. Major contributing factors to such a widespread of linear systems are their developed theoretical foundations, and the abundance of practical methods for their solution, making them an ideal building block and approximation tool for more complex applications [12, 17].

Despite their ubiquity, there are still significant efforts focusing on the development of efficient methods for linear systems. One reason is the sheer scale of the systems that need to be solved [5], which either stems from the amount of data that has to be processed, or from the desire to better approximate a continuous equation. Fortunately, a majority of such problems exhibit certain structural properties, *e.g.*, their system matrices contain a high percentage of zero entries (*sparse matrices*) or low-rank matrix blocks (*hierarchical matrices*), enabling the development of various data compression techniques and accompanying algorithms that leverage the compressed data directly [7, 13, 16, 26].

In addition to the special properties of the problem instance, another consideration in algorithm design are the architectural features of computing platforms that will be used to run it. Recently, as physical limitations undermined Dennard scaling, further hardware improvements have turned to non-conventional, special-purpose chip designs, such as Graphics Processing Units (GPUs), Intel Xeon Phi and Field-programmable gate arrays (FPGAs). Among the available alternatives, GPUs achieve the widest adoption, with 5 of the world's 10 most powerful systems featuring GPUs as the main contributor to their performance and energy efficiency [27].

High levels of hardware parallelism offered by the GPUs proved to be a good match for many methods for the solution of general systems, and a high-performance port was quickly developed [23]. In contrast, methods for compressed, especially sparse systems pose a greater challenge since the appropriate algorithms are bound by memory bandwidth, and the system matrices often feature highly irregular distribution of nonzero values. While there are libraries providing support for the basic methods [15, 23, 24, 28], more advanced algorithms are either not suitable for GPUs, not yet ported to GPUs, or only available as special-purpose implementations, part of a domain-specific software. New methods tailored specifically for GPUs are another area of current research.

Considering the relative novelty combined with the wide-spread usage of the GPU hardware, the resulting landscape offers a plethora of possible research directions. This thesis explores one of these directions: the study of Krylov subspace-based methods and related building blocks. The rest of this chapter introduces linear systems, sparse storage formats, Krylov methods, preconditioners and the limitations of current high performance computing (HPC) hardware in more detail, while the remaining chapters present the original contributions of the thesis.

1.1 Linear Systems

This section presents a short overview of various methods for solving linear systems. As there already is literature providing extensive descriptions and theoretical analyses of the methods, this section only aims at outlining their general classification, and introducing the reader with the main ideas underlying them. For an

Name	Mathematical description	Supported matrix types
LU factorization	$A = LU$ L unit lower triangular U upper triangular	general
Cholesky factorization	$A = LL^*$ L lower triangular	symmetric positive definite
LDL factorization	$A = LDL^*$ L unit lower triangular D diagonal	symmetric
QR factorization	$A = QR$ Q unitary R upper triangular	general

Table 1.1: Common direct methods for the solution of linear systems.

introductory text on an undergraduate level, readers are referred to the book by Ipsen [18], which describes the basic direct methods and provides the most important parts of their theoretical analysis. Demmel's graduate-level book [12] presents the same topics in more detail, and also provides a significant amount of material on iterative methods. Finally, advanced material on the topic can be found in the following trio: Higham's book [17] describes the error analysis of direct methods and iterative relaxation methods; Duff et al. [13] provide an extensive text on sparse direct methods; and a detailed description of iterative methods is available in Saad's book [26].

Solving a linear system refers to finding a vector $x \in \mathbb{F}^n$ such that $Ax = b$ for a known matrix $A \in \mathbb{F}^{n \times n}$ and a vector $b \in \mathbb{F}^n$ (right-hand side). Here, $\mathbb{F}$ is either the field of real numbers ($\mathbb{R}$), or the field of complex numbers ($\mathbb{C}$), and n is a positive integer which determines the size of the system. If the system matrix is nonsingular, the unique solution is equal to $x = A^{-1}b$ [12]. While a straightforward approach would be to compute the matrix inverse A^{-1} and apply it to b , this strategy suffers from numerical instability, and needs unnecessary floating-point operations [12, 18]. In practice, depending on the numerical properties of the system matrix, one can choose an alternative with higher numerical stability and lower computational cost. There are two main approaches for solving linear systems, resulting in a distinction between direct and iterative methods [12].

Direct methods exploit the fact that systems with matrices of some special structure are relatively easy to solve. For example, a system with a diagonal matrix ($A_{ij} = 0$ for $i \neq j$) can be solved by dividing the entries of b with the corresponding diagonal entries of A ; an upper ($A_{ij} = 0$ for $i < j$) or lower triangular system ($A_{ij} = 0$ for $i > j$) is easily solved via forward or backward substitutions, respectively [12, 18]; a unitary system ($A^*A = I$, where $(A^*)_{ij} = \overline{A_{ji}}$) is solved by multiplying the right-hand side with A^* . Systems that do not belong to one of these categories are handled by factorizing the original system matrix into a product of two or more matrices that do:

$$A = F_1 \cdot F_2 \cdot \dots \cdot F_k, \quad (1.1)$$

where F_i is diagonal, triangular or unitary, for $i = 1, 2, \dots, k$, and solving a series of simple systems:

$$F_1 x_1 = b, \quad (1.2)$$

$$F_2 x_2 = x_1, \quad (1.3)$$

$$\vdots$$

$$F_k x = x_{k-1}. \quad (1.4)$$

The most popular direct methods are listed in Table 1.1. LU factorization is the most common form and can be used on all nonsingular matrices. The Cholesky factorization exists only for symmetric positive definite matrices [12], while the LDL factorization relaxes this requirement to symmetric matrices, regardless of their definiteness. The QR factorization works for general matrices. It provides better error bounds than the LU factorization and can also be used to solve non-square, overdetermined and underdetermined systems, but

Name	Splitting	Supported matrix types
Richardson	$M = \frac{1}{\alpha}I$	general
Jacobi	$M = D$	general
Gauss-Seidel	$M = D - L$	general
SOR(ω)	$M = \frac{1}{\omega}(D - \omega L)$	general
SSOR(ω)	$M = \frac{1}{\omega(2-\omega)}(D - \omega L)D^{-1}(D - \omega U)$	symmetric

Table 1.2: Common relaxation methods for the solution of linear systems. The matrix D is the diagonal, $-L$ the strict lower triangle and $-U$ the strict upper triangle of the system matrix A . α and ω are scalar values.

needs more operations than LU [12]. Many direct methods need to be augmented with a pivoting strategy to ensure the existence and numerical stability of the factorizations listed above, which includes permuting the rows and columns of the matrix during the factorization process [12, 13]. Effectively, this results in the factorization being done on the permuted matrix $B = PAQ^T$, where P and Q are matrices defining the row and column permutations, respectively. Assuming the matrix is stored in full, uncompressed form, all of these methods require $O(n^3)$ floating point operations (*flops*) to produce the factorization and $O(n^2)$ flops to solve the system for one right-hand side. However, they have different constant factors hidden underneath the big- O notation [21].

Iterative methods, in contrast, produce a sequence $x_0, x_1, x_2, \dots$ of approximations to the solution x , starting from an initial guess x_0 . The hope is that the approximation sequence converges towards x , and that the approximation is good enough after a reasonable amount of iterations. Theoretical analysis only guarantees convergence for some methods and for matrices with certain properties. Nevertheless, iterative methods offer some attractive properties [26]: 1) they converge for many classes of real-world problems; 2) the quality of the solution is proportional to the time invested in computing it, enabling performance gains over direct methods if only a solution of reduced accuracy is required; 3) for some matrices and a suitable initial guess, even a fully accurate solution can be obtained in only a couple of iterations; 4) the matrix A is invariant; and 5) cost per approximation is in general low.

Relaxation methods are the oldest and simplest class of iterative methods. The idea is to split the system matrix A into the sum of two matrices $A = M - N$, where M is nonsingular and a system with matrix M is relatively easy to solve. Then, the problem can be rewritten as

$$Ax = b \tag{1.5}$$

$$(M - N)x = b \tag{1.6}$$

$$Mx = Nx + b \tag{1.7}$$

$$x = M^{-1}Nx + M^{-1}b \tag{1.8}$$

yielding an iterative method via the recurrence relation

$$x_{k+1} = M^{-1}Nx_k + M^{-1}b. \tag{1.9}$$

This class of methods converges for any right-hand side b and any initial guess x_0 if and only if the spectral radius $\rho(A) = \max\{|\lambda| : \lambda \in \mathbb{C}, \exists x \in \mathbb{F}^n, Ax = \lambda x\}$ of the matrix $M^{-1}N$ is strictly less than 1 [12, 26]. Table 1.2 lists the most common relaxation methods, together with a matrix M which defines the splitting. The properties required to fulfill the spectral radius condition differ among the methods, and depend on the properties of the system matrix and the choice of the open parameters α and ω [7]. All these methods can be transformed into their blocked variant by replacing the diagonal D with the block-diagonal, the strict lower triangle $-L$ with the strict lower block-triangle and the upper triangle $-U$ with the strict upper block-triangle of the system matrix A , which can significantly increase the convergence rate of the solver [26]. Since each iteration consists of several matrix-vector products, solutions of simple systems, and vector operations, the complexity of the method is

```

Initialize  $x_0, r_0 := b - Ax_0, p_0 := r_0, \tau_0 := r_0^* r_0$ 
 $k := 0$ 
while not converged
     $q_{k+1} := Ap_k$ 
     $\eta_k := p_k^* q_{k+1}$ 
     $\alpha_k := \tau_k / \eta_k$ 
     $x_{k+1} := x_k + \alpha_k p_k$ 
     $r_{k+1} := r_k - \alpha_k q_{k+1}$ 
     $\tau_{k+1} := r_{k+1}^* r_{k+1}$ 
     $\beta_{k+1} := \tau_{k+1} / \tau_k$ 
     $p_{k+1} := r_{k+1} + \beta_{k+1} p_k$ 
     $k := k + 1$ 
endwhile

```

Figure 1.1: A pseudocode of the Conjugate Gradient Krylov method.

$O((n^2 + s)m)$ flops, where s is the cost of solving a system with M , and m is the number of iterations required to achieve a good enough approximation. Thus, speedups over direct methods are possible if m is significantly smaller than n .

A relatively newer, and usually more effective class of iterative methods is the class of methods based on Krylov subspaces. Since every square matrix A satisfies its own characteristic equation $k_A(\lambda) = 0$ (*i.e.* $k_A(A) = 0$), where k_A is the characteristic polynomial $k_A(\lambda) := \det(\lambda I - A) = \alpha_0 + \alpha_1 \lambda + \dots + \alpha_n \lambda^n$ of the matrix A (a property known as the Cayley-Hamilton theorem), multiplying the equation with the solution x of the linear system results in the following formula:

$$k_A(A)x = 0, \quad (1.10)$$

$$\alpha_0 x + \alpha_1 Ax + \dots + \alpha_n A^n x = 0, \quad (1.11)$$

$$\alpha_0 x + \alpha_1 b + \dots + \alpha_n A^{n-1} b = 0, \quad (1.12)$$

$$x = -\frac{1}{\alpha_0}(\alpha_1 b + \dots + \alpha_n A^{n-1} b), \quad (1.13)$$

where the last equation holds since A is non-singular, *i.e.* $\alpha_0 = k_A(0) = \det(A) \neq 0$. Thus, the solution is in one of the Krylov subspaces $\mathcal{K}_{A,b}^m := \text{span}\{b, Ab, \dots, A^{m-1}b\}$, $m = 1, \dots, n$. In practice, finding the coefficients α_i of the characteristic polynomial is far more difficult than solving the system. Instead, practical Krylov methods construct a series of subspaces $\mathcal{K}_{A,b}^m$ and find a projection (orthogonal or oblique) of the solution x onto that subspace. By using a clever definition of the inner product, this projection can be obtained without knowing x itself [12, 26].

If one of the Krylov subspaces is invariant for A , *i.e.* $A\mathcal{K}_{A,b}^m \subseteq \mathcal{K}_{A,b}^m$, then the sequence folds onto itself ($\mathcal{K}_{A,b}^{m+1} = \mathcal{K}_{A,b}^m \cup A\mathcal{K}_{A,b}^m = \mathcal{K}_{A,b}^m$) and the exact solution is found after m steps in $\mathcal{K}_{A,b}^m$. Finding an invariant subspace early in the iterative process is the hope of Krylov subspace-based methods, since in that case, only m multiplications with the system matrix are needed. Even if the sequence does not fold onto itself early, the hope is that the projection of the solution x to the subspaces $\mathcal{K}_{A,b}^m$ is close enough to x , so that the method finds this solution. Similarly to relaxation methods, the complexity of Krylov methods is $O(n^2 m)$ flops. Usually, the number of iterations m needed is much smaller than that required by relaxation methods, and, assuming exact arithmetic and no breakdowns in the orthogonalization process, m is bounded by the size of the system n . Another appealing property of Krylov subspace-based methods is the fact that the system matrix is used indirectly, as part of the Krylov subspace construction, and to define the inner product. This is especially appealing for a prospective software library designer, as the only operations where the system matrix is required is its application to a vector (*i.e.* a matrix-vector product). Thus, different matrix storage formats and corresponding matrix-vector product implementations can easily be swapped and used with the same implementation of the Krylov method.

Figure 1.1 shows the pseudocode of the Conjugate Gradient (CG) Krylov method, suitable for symmetric positive definite matrices. The information about the current Krylov subspace is stored implicitly as part of

the auxiliary vectors p , q and r . The main components of the method can be clearly seen: the matrix-vector product used to construct the next vector in the subspace; and vector operations used to orthogonalize the subspace and construct the projection of x onto that subspace. The convergence of the method depends on the spectral properties of the system matrix [7, 12, 26], with the method converging by a factor of $(\sqrt{\kappa_2(A)} - 1)/(\sqrt{\kappa_2(A)} + 1)$ per iteration, where $\kappa_2(A) := \|A\|_2\|A^{-1}\|_2$ is the spectral condition number of A [7, 12, 26]. Other Krylov methods contain similar components, with some of them requiring the conjugate matrix-vector product ($y = A^*x$) as well. Similarly to CG, their theoretical convergence can usually be bound by some polynomial of the system matrix' spectrum. The pseudocode for those methods, as well as their derivation and theoretical analysis can be found in other literature [7, 12, 26].

The last iterative method discussed in this section is iterative refinement. This is not a standalone method, but can be used to improve the accuracy of other methods. A coarse, less accurate method for solving $Ax = b$ produces a result $\tilde{x}_0 = x + e$, where e is the error in the solution. The error e can be approximated by solving a new system $Ac = r_0$ using the coarse method, where $r_0 = b - A\tilde{x}_0$, obtaining $c = A^{-1}r_0 = A^{-1}b - \tilde{x}_0 = x - \tilde{x}_0 = -e$. Then, c can be used to correct the solution $\tilde{x}_0$, since $x = \tilde{x}_0 + c$. However, as c is only approximated using the coarse method, $\tilde{c} = c + e_c$ is obtained instead of c . Thus, the corrected solution is actually $\tilde{x}_1 = \tilde{x}_0 + \tilde{c} = x + e_c$. Nevertheless, as long as the residual r_0 and the update $\tilde{x}_1$ is computed more accurately than the solution of the system, the new error e_c will be several orders of magnitude smaller than e [12, 26]. The process can be repeated iteratively to decrease the error further. Iterative refinement is usually used as a way to obtain a solution better than the working precision of the coarse method, either by 1) using a lower precision arithmetic in the coarse method to accelerate the solution process [4, 10], or 2) by using non-IEEE compliant or software-defined arithmetic for residual calculation and solution updates, resulting in a more accurate solution than possible using the standard floating point types [12].

For completeness, it is worth mentioning there are more advanced methods for solving linear systems, which yield significant performance improvements in some special cases, or even enable solving problems which are otherwise not solvable via standard techniques. Notably, these are the multigrid and domain decomposition methods. However, these methods are not in the scope of this thesis, so the interested reader is referred to other literature describing them [12, 26].

1.2 Sparse Matrices

Many problems, such as finite element and finite difference discretizations [13, 26] or problems arising from graph applications [19], result in system matrices where the majority of elements are zero with each row having only a few nonzero elements on average. Storing all these zeros in an uncompressed, dense matrix is wasteful both in terms of storage and computational cost, since the majority of the computations will be multiplications or additions with zero. Furthermore, some of these systems contain a very large number of unknowns (more than a million), so just storing the matrix in double precision would exceed the memory capacity of a standard computer server.

A common approach for dealing with such problems is to exploit the high fraction of zeros in the matrix by storing only the nonzero elements, thus significantly reducing the memory usage. In addition, if all operations that depend on the matrix are implemented to directly use the compressed data, computational savings can be obtained by skipping the operations involving zero elements which are not stored.

Over the years, a variety of sparse matrix storage formats has been developed, which aim at balancing the storage savings, and the efficient access of stored data when performing the necessary operations. Derivation of the most basic ones is shown graphically on Figure 1.2. The simplest one is the Coordinate format (COO). Its derivation is straightforward: all nonzero entries are stored in sequence, and accompanied by two indices which determine the position of that element in the matrix. The information is stored in structure-of-arrays form — a common approach when storing sparse matrices so as to maximize cache efficiency. Another option is to compress each row individually, only adding additional information about the column index. The resulting data structure can then either be embedded into square matrices by adding dummy elements as padding and storing them in one of the dense matrix formats (usually in column-major order, that is column-by-column), or individual rows can be stored in sequence and augmented with pointers to the starting position of each row. This results in the so-called ELLPACK (ELL) and Compressed Sparse Row (CSR) formats, respectively. Alternatively, one could compress each column instead of each row, resulting in a column variant of ELLPACK and

Figure 1.2: Derivation of common sparse matrix storage formats.

the Compressed Sparse Column (CSC) format [26]. In addition to these basic formats, a number of advanced formats were developed to offer additional memory and computational savings with certain algorithms, applications, or on certain hardware. Common approaches include blocked versions of basic formats [9, 29], their various combinations [1, 8, 20], and their unconventional enhancements [14, 22].

Even with a variety of available matrix formats, using them as part of method implementations is nontrivial. In the context of direct methods, the first challenge is the fact that factorizations do not preserve the sparsity pattern (*i.e.* the locations of nonzero elements) of the system matrix. Indeed, if the system matrix contains a nonzero element on position (i, j) , the L factor in the LU, Cholesky and LDL factorizations will generally have nonzeros in all positions (i, k) where $j \leq k \leq i$. Similarly, the U factor in LU factorizations will contain a nonzero in all positions (k, j) where $i \leq k \leq j$ [13]. A similar situation occurs with a matrix product AB of matrices A and B — which is a common building block of QR factorization algorithms [12] — where $(AB)_{ij}$ is nonzero if there exists an index k such that both A_{ik} and B_{kj} are nonzero. Consequently, depending on the sparsity pattern of the matrix, its factorization may have a significantly larger proportion of nonzero elements, as some positions that were zero in the original matrix become nonzero in the factorized form (fill-in positions). To alleviate the problem, sparse direct methods use various reordering (pivoting) strategies, such as the Reverse Cuthill McKee (RCM), Minimum Degree (MD) and Nested Dissection (ND) orderings, with the aim of moving the nonzeros closer to the diagonals and thus reducing the amount of fill-in [13, 26]. However, the strategies have to be balanced with the preservation of numerical stability, and a good reordering is not always easy to find, making the use of direct methods for sparse systems far more limited than in the dense case [13, 26]. An additional difficulty is the fact that the exact amount and locations of the fill-in elements are usually not known in advance, so sparse direct methods use several phases combined with specialized sparse storage formats that allow for the easy insertion of new elements to the data structure [13, 26].

The situation is somewhat better in the case of iterative methods. Instead of factorizing the matrix, relaxation methods split the matrix into the sum of two matrices. In addition, the operation $y = M^{-1}Nx$ is usually

calculated as a matrix-vector product $z = Nx$, followed by a solution of the linear system $My = z$, avoiding the need for inversions and matrix-matrix products which would destroy the sparsity pattern. Thus, the only operations required for relaxation methods are the matrix-vector products with sparse matrices, solutions of linear systems with sparse triangular or diagonal matrices, and the extraction of the upper and lower triangles and the diagonals of the matrix from the sparse data structure, which, at least for conventional storage formats, can usually be realized by augmenting the sparse structure with pointers to the diagonal elements. The only difficulty that occurs in both the relaxation methods, as well as the solution step of direct methods, is the parallelization of triangular system solves. While the operation is considered well-parallelizable in the dense case, some sparsity structures make the operation hard to parallelize, or even inherently sequential. An example of a matrix structure that offers no parallelism whatsoever is the bidiagonal structure, which has nonzero elements only on the main diagonal and the first diagonal below it. For such a matrix, each stage of the forward substitution contains only one operation. Furthermore, different stages cannot be parallelized, as each subsequent stage depends on the result of the previous one. In contrast to direct and relaxation methods, Krylov methods present the ultimate solution when dealing with sparse matrices. With the only requirement being the matrix-vector product — and in some cases the conjugate matrix-vector product — it is relatively simple to design a sparse matrix format which allows computing this single operation efficiently. Even if the format does not allow for efficient conjugate matrix-vector product, it is often possible to explicitly store the conjugate transpose of the matrix in memory and use the same algorithm as for the regular matrix-vector product.

In both direct and iterative methods, coming up with a single storage format for all operations is extremely difficult. This is especially the case for highly complicated and specialized formats, which are usually only designed to speed up the matrix-vector product. Other operations are either not possible to realize efficiently, or require significant implementation effort. Instead, most software will rely on sometimes expensive format conversion procedures to provide other operations when a specialized storage format is used. An additional difficulty is interoperability between libraries, as every format allows for slight variations of the basic scheme. As a result, the adoption of specialized formats is extremely slow, with the majority of software packages continuing to rely on basic storage formats [15, 23, 24, 28]. Among them, the CSR format is considered the de-facto standard, as it usually offers the best storage efficiency, and has historically provided the best performance on conventional processor architectures.

1.3 Preconditioning

As outlined in Section 1.1, the convergence rate of Krylov methods is tied to a function of its spectrum. Thus, if the original system is replaced with a different system that has the same solution but better spectral properties, the method will converge in fewer iterations. This can be achieved by using a preconditioner matrix M to transform the original system $Ax = b$ into a *preconditioned system* in one of the following ways:

- *left preconditioning*,

$$M^{-1}Ax = M^{-1}b; \quad (1.14)$$

- *right preconditioning*,

$$AM^{-1}y = b, \quad (1.15)$$

where $Mx = y$; and

- *two-sided preconditioning*,

$$M_1^{-1}AM_2^{-1}y = M_1^{-1}b, \quad (1.16)$$

where $M_2x = y$ and $M = M_1M_2$.

To make sure the preconditioned system is easier to solve than the original one, the *preconditioner* M should be chosen such that $M^{-1}A$, AM^{-1} or $M_1^{-1}AM_2^{-1}$ is better conditioned than A , or at least has fewer extreme eigenvalues. Additionally, one needs to compute $M^{-1}b$ and a series of matrix-vector products $z = M^{-1}Ay$, so M should be chosen in a way that makes computing $z = M^{-1}w$ easy. Unfortunately, these two requirements are mutually exclusive. The first one is optimized by setting $M = A$, resulting in perfectly conditioned system matrix $M^{-1}A = I$, but the operation $z = M^{-1}w = A^{-1}w$ is as difficult to compute as the original problem. On the other hand, the second is optimized by setting $M = I$, which does not yield any improvement of the spectral

properties. Thus, an effective preconditioner balances the trade-offs between the two extremes, and provides moderate improvements of the spectrum, while keeping its structure simple enough for computing $z = M^{-1}w$ cheaply. Finding efficient preconditioners is an area of active research and, while there are no methods which would find perfect ones, there are several heuristics that generate good preconditioners for certain types of problems.

One category of heuristics is derived directly from relaxation methods [26]. By setting $G := M^{-1}N$ and $f := M^{-1}b$, the relaxation method equation (1.9) can be rewritten as:

$$x_{k+1} = Gx_k + f. \quad (1.17)$$

This is in fact the Richardson iteration (see Table 1.2) with parameter $\alpha = 1$ for the system

$$(I - G)x = f. \quad (1.18)$$

Using the equalities $I - G = I - M^{-1}N = M^{-1}(M - N) = M^{-1}A$ and $f = M^{-1}b$, it can be rewritten as

$$M^{-1}Ax = M^{-1}b. \quad (1.19)$$

This shows that any relaxation method is just Richardson iteration on a preconditioned system, where the preconditioner M is the same matrix that defines the splitting in Table 1.2. Thus, every matrix that defines a relaxation method can also be used as a preconditioner, defining the Jacobi, Gauss-Seidel, $\text{SOR}(\omega)$ and $\text{SSOR}(\omega)$ preconditioners, and their blocked variants.

Another class of preconditioners is obtained by using the ideas from sparse direct methods [26]. Instead of completing the (often expensive) full factorization, one can limit the amount of fill-in to obtain an approximate factorization $A = F_1 \cdot \dots \cdot F_k - R$, where R is the residual of the approximation. The approximate factorization is then used as the preconditioner $M = F_1 \cdot \dots \cdot F_k$. These ideas can be combined with various ways of controlling the fill-in of the factors and result in families of Incomplete LU (ILU) and Incomplete Cholesky (IC) preconditioners [3, 26].

Other preconditioning heuristics include methods for approximating the inverse A^{-1} , using a small number of iterations of another method, or leveraging problem-specific knowledge to construct a suitable preconditioner.

1.4 Numerical Methods in High Performance Computing

While the previous sections mostly focused on numerical aspects of various methods for the solution of linear systems, this section briefly outlines additional considerations that have to be taken into account when designing high performance software for current hardware.

The defining characteristic of today's systems is the discrepancy between processor and memory speed, with most systems being able to perform between 10 and 100 operations for every byte fetched from memory. To resolve the issue, systems are designed with a hierarchy of increasingly smaller, faster, and more expensive memories placed between the main memory and the processor. The idea is to hide the slow main memory by placing often-accessed data into these faster memories, which is either done automatically by the hardware (*cache*) or manually by the programmer (*scratchpad memory*). Multiple memory- and processor modules are connected together to form *nodes*. The memory modules are often presented as a single unified memory. Nevertheless, depending on the node configuration, a group of processors can exhibit lower bandwidth with some memory modules (*e.g.*, NUMA configurations and heterogeneous systems with Unified Memory enabled), or even have no access to them (*e.g.*, heterogeneous systems with Unified Memory disabled). State-of-the-art processors can roughly be divided into two categories:

- *general-purpose processors*, which are installed in 1–4 groups (*i.e.* *sockets*) of 2–30 processors (*i.e.* *cores*) per node, with each one able to perform 8–32 operations (combined into vector instructions) in parallel; and
- *accelerators*, which are installed in 0–16 groups (*e.g.* *GPUs*) of 1–80 processors (*e.g.* *Streaming Multiprocessors*) per node, with each one able to perform 64–196 operations (combined into vector instructions) in parallel.

Compared with accelerators, general-purpose processors usually operate on higher frequencies and expose less parallelism, but feature higher energy cost per operation ratio. As a result, new systems follow a trend where an increasingly larger proportion of total compute power is supplied by the accelerators, while the general-purpose processors are used for managing I/O devices, networking, copying memory and orchestrating computation. As a final layer of the hierarchy, large systems are formed by connecting multiple nodes into a network (*cluster*). The largest systems contain between 1,000 and 100,000 nodes, with the number converging towards the lower end, due to the current trend of reducing the total number of nodes by using more powerful “*fat*” nodes [27].

Direct methods for dense systems offer a relatively straightforward mapping to modern hardware. The volume of data is a lower degree polynomial ($O(n^2)$) than the amount of computation ($O(n^3)$) needed to perform the factorization, so there are plenty of opportunities for data reuse and effective utilization of caches and scratchpad memories. In addition, the regular structure of these problems implies that the amount of computation needed to process each row, column or block of the system matrix is roughly constant, so the work can be easily distributed evenly among processors.

On the other hand, direct and iterative methods for sparse systems suffer from more serious issues. The total data volume, $O(n_{nz})$, where n_{nz} is the total number of nonzeros in the system, is usually significantly closer to the total amount of computation, which is $O(n_{nz}m)$ for iterative methods and ranges between $O(n_{nz})$ and $O(n^3)$ for direct methods, resulting in far less opportunities for data reuse. Iterative methods in particular proceed in a sequence of iterations that cannot be combined, and each one requires the complete problem data. There are ways, however, to slightly alleviate the issue by processing multiple systems that have the same system matrix at the same time, or adding multiple vectors to the Krylov subspace in each iteration [30]. Work distribution is another issue for both direct and iterative methods. This is a direct consequence of often imbalanced distribution of nonzeros in the system matrix, which hampers the development of efficient building blocks, such as the matrix-vector product and factorization algorithms. Especially difficult are the algorithms for triangular solves since, depending on the sparsity pattern of the matrix, they can exhibit virtually no parallelization potential [2]. These methods are often the Achilles’ heel of accelerator-focused systems, as load balancing and low parallelization potential present significant difficulties on highly-parallel hardware. As a result of these issues, state-of-the art methods for sparse systems are limited by the memory bandwidth and only use a fraction of the processing power available in today’s systems [27].

1.5 This Work

With the shift towards fat, heterogeneous nodes, efficient accelerator-focused algorithms are becoming increasingly important. Significant improvement is especially possible on methods for sparse systems. The rest of this thesis deals with the category of iterative methods, and mostly considers the Krylov method subcategory. Our goal is not to develop new methods, but to improve existing ones by accelerating their building blocks. Optimizations of vector operations are not discussed, since an abundance of recent research already dealt with this aspect [6]. Instead, this work is focused towards algorithms for preconditioner and matrix-vector product computations. The hardware considered is not a full node, but a single accelerator processor group (that is, a single GPU) together with the memory directly attached to it. All lower levels of the hierarchy are considered, including individual processors and vector units, and there are no assumptions about the availability of lower granularity operations. Thus, the algorithms presented here constitute the lowest granularity building blocks of applications which utilize the full system or simplified studies on the way towards larger building blocks. As such, they are crucial for the design of larger software, since any performance issues on a processor group level will be transferred to higher levels of the software.

This work is designed as a collection of standalone articles. Each chapter constitutes of one such article and can be read independently of the rest of the thesis. The first section of each chapter contains introductory remarks which establish the context of that chapter and provide references to related research. Thus, there is a fair amount of repetition in these sections, which means that readers interested in the entire thesis may want to skip them during the first read. The chapters are organized into thematic parts and generally increase in complexity towards the end of the thesis. The reason for this organization is to form a coherent story line, as opposed to presenting the chronological history of the research. Thus, some chapters may refer to work presented later in the text, but that information will not be necessary to understand the current chapter. The rest of this work is organized as follows:

- Part II deals with algorithms for the computation of sparse matrix-vector products. Special attention is given to irregular matrices and standard, well-established compression formats, since they constitute the case where current research is somewhat lacking. Chapter 2 describes a load-balanced matrix-vector algorithm for the widely used CSR storage format, which achieves superior performance on irregular matrices compared to conventional algorithms. Chapter 3 explores the potential of the COO format, which fell out of favor in numerical linear algebra, and shows that it becomes relevant once more on modern accelerator hardware. Finally, Chapter 4 describes various algorithms and storage formats in cases when the full problem is decomposed into smaller problems whose granularity fits a single processor (*i.e.*, Streaming Multiprocessor), instead of a processor group (*i.e.*, GPU).
- In Part III, the attention is shifted towards preconditioning. The discussion is restricted to block-Jacobi preconditioners, which in itself already offer an abundance of possible algorithms. Chapter 5 describes an algorithm which uses explicit inversion techniques to construct the preconditioner. Its idea is to optimize the preconditioner application process by expressing it as a batched dense matrix-vector product, while allowing for a slightly longer preconditioner generation step and ignoring possible instabilities from the inversion. These issues are dealt with in Chapter 6, which demonstrates that the instabilities do not occur in real-world problems and that the inversion-based preconditioner is superior to the numerically stable version for moderate to large number of outer solver iterations. In addition, it also explores the potential of another forgotten method for the solution of dense linear systems. In this case, however, the standard LU-factorization based method can be implemented in a superior way by replacing the conventional “lazy” triangular solves with the “eager” variant, which is the topic of Chapter 7.
- Part IV takes the block-Jacobi preconditioning idea one step further by exploring the possibilities of low-precision storage. Its only chapter establishes a new research direction of adaptive precision preconditioning techniques by providing a theoretical analysis of the adaptive precision block-Jacobi preconditioner. It lays the groundwork for practical implementations and theoretical analysis of other preconditioners that automatically adapt their storage precision to the numerical properties of the problem.
- Part V provides a summary of avenues that remain open after this work. It proposes a novel sparse linear algebra library design motivated by experience gained from writing and using existing high performance software. It also presents current and future research that resulted, or is a natural extension of this thesis.

Bibliography

- [1] H. Anzt, S. Tomov, and J. Dongarra. Implementing a sparse matrix-vector product for the SELL-C/SELL-C- σ formats on NVIDIA GPUs. Technical report, The University of Tennessee at Knoxville, 2014.
- [2] H. Anzt, E. Chow, and J. Dongarra. Iterative sparse triangular solves for preconditioning. In *Proceedings of the 21st International European Conference on Parallel and Distributed Computing*, Euro-Par 2015, pages 650–661. Springer, 2015.
- [3] H. Anzt, E. Chow, and J. Donagarra. ParILUT—a new parallel threshold ILU factorization. *SIAM Journal on Scientific Computing*, 40(4):C503–C519, 2018.
- [4] H. Anzt, G. Flegar, V. Novaković, E. S. Quintana-Ortí, and A. E. Tomás. Residual replacement in mixed-precision iterative refinement for sparse linear systems. In *High Performance Computing*, pages 554–561. Springer, 2018.
- [5] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegel, F. Streitz, A. White, and M. Wright. The opportunities and challenges of exascale computing. Technical report, U.S. Department of Energy, 2010.
- [6] J. P. Badenes. *Consumo Energético de Métodos Iterativos Para Sistemas Dispersos en Procesadores Gráficos*. PhD thesis, Universitat Jaume I, 2016.

-
- [7] R. Barrett, M. Berry, T. F. 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*. SIAM, 2nd edition, 1994.
 - [8] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA Corporation, 2008.
 - [9] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA’09, pages 233–244. ACM, 2009.
 - [10] E. Carson and N. J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing*, 40:A817–A847, 2018.
 - [11] C. de Boor. *A Practical Guide to Splines*. Springer-Verlag, 1st edition, 1978.
 - [12] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1st edition, 1997.
 - [13] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, 2nd edition, 2017.
 - [14] J. P. Ecker, R. Berrendorf, and F. Mannuss. New efficient general sparse matrix formats for parallel SpMV operations. In *Proceedings of the 23rd International European Conference on Parallel and Distributed Computing*, Euro-Par 2017, pages 523–537. Springer, 2017.
 - [15] Ginkgo. <https://ginkgo-project.github.io>, 2019.
 - [16] W. Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*. Springer-Verlag, 1st edition, 2015.
 - [17] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002.
 - [18] I. C. F. Ipsen. *Numerical Matrix Analysis: Linear Systems and Least Squares*. SIAM, 1st edition, 2009.
 - [19] J. Kepner and J. Gilbert, editors. *Graph Algorithms in the Language of Linear Algebra*. SIAM, 1st edition, 2011.
 - [20] 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. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014.
 - [21] E. Landau. *Foundations of Analysis*. AMS, 3rd edition, 1966.
 - [22] 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, pages 339–350. ACM, 2015.
 - [23] MAGMA 2.5.0. <http://icl.cs.utk.edu/magma/>, 2019.
 - [24] PARALUTION. <http://www.paralution.com/>, 2015.
 - [25] E. Polizzi. Density-matrix-based algorithm for solving eigenvalue problems. *Phys. Rev. B*, 79(11):115112, 2009.
 - [26] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.
 - [27] The Top 500 List. <http://www.top.org/>, 2019.
 - [28] ViennaCl. <http://viennacl.sourceforge.net/>, 2015.
 - [29] R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, 2003.
 - [30] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra. Improving the performance of CAGMRES on multicores with multiple GPUs. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS’14, pages 382–391. IEEE, 2014.
-

Part II

Sparse Matrix Formats and Matrix-Vector Product

Balanced Sparse Matrix-Vector Product for the CSR Matrix Format

Published as: G. Flegar and E. S. Quintana-Ortí. Balanced CSR sparse matrix-vector product on graphics processors. In *Proceedings of the 23rd International European Conference on Parallel and Distributed Computing*, Euro-Par 2017, pages 697–709. Springer, 2017

We propose a novel parallel approach to compute the sparse matrix-vector product (SpMV) on graphics processing units (GPUs), optimized for matrices with an irregular row distribution of the non-zero entries. Our algorithm relies on the standard CSR format to store the sparse matrix, requires an inexpensive pre-processing step, and consumes only a minor amount of additional memory compared with significantly more expensive GPU-specific sparse matrix layouts. In addition, we propose a simple heuristic to determine whether our method or the standard CSR SpMV algorithm should be used for a specific matrix. As a result, our proposal, combined with the standard CSR SpMV, can be adopted as the default choice for the implementation of SpMV in general-purpose sparse linear algebra libraries for GPUs.

2.1 Introduction

The sparse matrix-vector product (SpMV) is a classical yet pivotal kernel for the solution of numerical linear algebra problems via iterative methods [10]. In the last years, this operation has also gained relevance for big data analytics [4] and web search [8]. It is thus natural that, over the past decades, a considerable research effort has been applied to design specialized data structures that offer a compact representation of the problem data to reduce the storage requirements, facilitate its manipulation, and diminish the volume of data movements for sparse computational kernels such as SpMV.

Among the variety of storage layouts for sparse matrices, the CSR (Compressed Sparse Row) format [10] conforms the current standard layout because of its storage efficiency which, in general, results in faster serial algorithms [3]. For this reason, CSR has become ubiquitous in sparse matrix computations [3, 6, 10].

For graphics processing units (GPUs), CSR can be outperformed by specialized sparse matrix layouts that sacrifice storage efficiency for fast (coalesced) memory access. Among these GPU-oriented formats, ELLPACK, ELLR-T [11] and SELL-C- σ [1, 7] have shown notable performance. Unfortunately, none of these formats is truly general. Some suffer from increased memory consumption, which can grow significantly for irregular sparsity patterns, while others (like NVIDIA’s HYB [5]) are only suitable for a few types of matrix operations (computational kernels) and/or require expensive format conversions. Another common issue arising in SpMV computations on GPUs is load imbalance. This has been a topic of some recent research, resulting in new matrix formats like CSR5 [9] and BCCOO [12], which enable well-balanced SpMV algorithms.

In this paper, we re-visit the CSR format, proposing a CSR-based SpMV variant that provides increased efficiency on GPUs and offers the following properties compared with standard CSR algorithm and GPU-specific solutions:

- Our balanced CSR algorithm for irregular matrices (CSR-I) ensures an even distribution of the workload among the CUDA threads participating in the SpMV, at the cost of using atomic updates to avoid race conditions.

Figure 2.1: Data layouts for an 8×8 sparse matrix with $n_z=9$ nonzero entries.

- CSR-I maintains the same data structure as CSR, and augments this with an additional vector of a dimension that is linear in the amount of available parallelism. For moderate to large-scale problems this introduces a negligible storage overhead, in general much lower than that incurred by ELLPACK-type formats and sliced versions (SELL-*).
- The additional data structure leveraged by CSR-I can be built at execution time, e.g. the first time an SpMV is invoked, for a very small computational cost, similar to that of reading once the solution vector for the SpMV.
- Our experiments with a subset of a sparse matrix benchmark show that CSR-I outperforms CSR for about 40–50% of the cases on NVIDIA architectures providing hardware support for atomic updates. Furthermore, it is easy to detect *a priori* when CSR-I should be the preferred option. This property leads to an optimal hybrid kernel that employs either CSR-I or CSR, depending on the target problem.

2.2 CSR-Based Formats and Algorithms for SpMV

CSR represents a sparse matrix $A \in \mathbb{R}^{m \times n}$ in compact form using three arrays: vector `val` stores the n_z non-zero values by rows; `colidx` keeps the column index for each value in the same order; and `rowptr` contains $m+1$ row pointers that mark the initial and final position of each row within the other two arrays; see Figure 2.1. Storing the sparse matrix A in CSR format thus requires $\mathcal{S}_{CSR} = n_z(s_v + s_i) + (m+1)s_i$ bytes, where s_v and s_i respectively denote the number of bytes occupied by each value and integer index.

In [2], Bell and Garland (BG) explored the performance of different sparse formats and implementations of SpMV for throughput-oriented GPUs from NVIDIA. BG’s SpMV kernels based on CSR parallelize the product across the matrix rows, with one CUDA thread assigned to each row in the *scalar* kernel (CSR-s) or, alternatively, one warp per row in the *vector* kernel (CSR-v). CSR-s has two major issues though: first, for sparse matrices with an irregular row distribution of their non-zero entries, many threads within a warp will likely remain idle. Second, since each thread of a warp works on a different row, the memory accesses are noncoalesced. CSR-v aims to amend the second issue, though it requires that the rows contain a number of nonzeros greater than the warp size in order to deliver high performance [2].

A couple of examples illustrate the advantages/deficiencies of CSR-s and CSR-v. Consider first an arrow-head matrix, with all its nonzero entries lying on the main diagonal and the last column/row of the matrix. (This problem type appears in domain decomposition methods, when discretizing partial differential equations.) This matrix structure poses an ill-case scenario for both BG kernels, as it produces a highly unbalanced mapping of the workload to the threads. In contrast, a tridiagonal matrix (often encountered in computational physics), results in an almost perfectly balanced distribution of the workload for both BG CSR kernels, but yields a significant waste of resources for CSR-v.

Figure 2.2: Left: sparsity pattern for FREESCALE/TRANSIENT. Right: execution time (blue) and memory consumption (red) on a GTX1080 using different SpMV kernels.

Figure 2.2 provides a motivating example for our work. The left-hand side plot there shows the sparsity pattern for matrix FREESCALE/TRANSIENT from the SuiteSparse Matrix Collection.¹ The distribution of the nonzeros for this problem, arising in circuit simulation, shows quite an unbalanced pattern, with most of the elements concentrated in a few rows of the matrix. Concretely, more than 95% of the rows contain 10 or less nonzeros; 99.95% comprise 100 or less nonzeros; only 5 rows contain more than 10^3 nonzeros; and only 2 more than 10^4 , with the densest row comprising 60,423 nonzero entries.

The right-hand side plot in Figure 2.2 reports the execution time on an NVIDIA GTX1080 GPU for double-precision SpMV kernels based on CSR and HYB (implemented in cuSPARSE [5]), SELL-P (from MAGMA-sparse²), and our balanced version of CSR (CSR-I). ELL and ELLR-T are not included because, for this problem instance, they both need to store an $m \times 60,423$ matrix just for the array `val` (i.e., more than 79.5 Gbytes in double precision), which exceeds the memory of the target GPU. For this particular matrix both CSR and SELL-P exhibit poor performance compared with HYB and CSR-I. SELL-P also suffers from considerably higher memory consumption than other implementations. CSR-I is the best performing algorithm in this case, achieving slightly better performance than HYB, while maintaining the storage efficiency of CSR.

2.3 Balanced SpMV kernel

The culprit for the unbalance in the SpMV implementations discussed in Section 2.2 is the irregular distribution of arrays `val` and `colidx` (and therefore workload) among the threads. This irregularity can result in significant performance loss, since the two vectors comprise the majority of CSR’s data structure. Hence, the key objective of our kernel is to attain a balanced distribution of these arrays among the threads. The trade-off for this comes in the form of an increased number of integer operations and the introduction of potential race conditions, which may result in slightly lower performance on regular sparsity patterns.

2.3.1 General idea

In order to distribute the arrays `val` and `colidx`, both of size n_z , evenly among T threads, thread $k \in \{0, 1, \dots, T-1\}$ is given a segment of the arrays starting at $\lfloor kn_z/T \rfloor$ (inclusive) and ending at $\lfloor (k+1)n_z/T \rfloor$ (exclusive). During the execution of the SpMV $y := Ax + y$, with an $m \times n$ matrix A , each thread multiplies the elements in its segment of `val` with those of the input vector x determined by the corresponding indices in `colidx` (dot product). The result has to be accumulated into the correct position of the output vector y . Thus, the thread has to keep track of the current row it is operating on, as well as the last entry of the row, in order to detect a change to the next row once this entry is reached. The sequential C routine in Figure 2.3 illustrates this idea. Since there can be multiple threads operating on the same row, the updates on the solution vector y have to

¹Formerly known as the University of Florida Matrix Collection: <http://www.cise.ufl.edu/research/sparse/matrices/>.

²<http://icl.cs.utk.edu/magma/>.

```

1  const int T = thread_count; // degree of thread concurrency
2  void SpMVI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
3      int row = -1, next_row = 0, nnz = rowptr[m];
4      for (int k = 0; k < T; ++k) { // this loop should be parallelized
5          for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
6              while (i >= next_row) next_row = rowptr[ ++row+1 ];
7              y[row] += val[i] * x[ colidx[i] ];
8          }
9      }
10 }

```

Figure 2.3: A (sequential) C implementation of the CSR-I algorithm. In a parallel implementation each thread needs to efficiently determine its starting value of the row variable. This is discussed in Section 2.3, “Determining the first row of each segment”.

be implemented as atomic transactions, resulting in one transaction per matrix element which rapidly becomes a performance bottleneck. However, this problem can be amended by accumulating the result to a thread-local variable, and updating the output vector only after the thread has finished processing the row. With this option, the upper limit on the number of atomic transactions is reduced to $m + T$.

2.3.2 Achieving good performance on GPUs

Although the approach underlying CSR-I does result in a balanced distribution of the data among the threads, it is not suitable for GPUs in such form. Concretely, since each thread operates on a different matrix segment, with that formulation the memory accesses of the threads within a warp will be noncoalesced. In a memory-bounded kernel like SpMV, this severely reduces performance. To tackle the issue, the segments can be distributed at the warp-level, so that each segment is assigned to one warp (instead of to one thread). The warp then reads its segment in chunks of 32 elements, with each thread within the warp reading one value of the chunk and accumulating the result into its local registers. After reaching the end of a row, all threads need to write their results to the output vector. If this was realized using atomic instructions, it would cause significant overhead, as the threads inside one warp are synchronized and all of them would then try to update the result at exactly the same time. Instead, the results are first accumulated by a single thread, using reduction via warp shuffles, and then a single atomic addition updates the result in global memory.

A second question arising from the warp-level segment distribution is how to handle rows that end in the middle of a chunk. Waiting for the entire warp to complete the processing of a row before moving to the next one would cause a partial serialization in case the rows consisted of a few elements only. To address this, the threads are allowed to work on different rows at the same time, and the information about the current and the next rows becomes thread-specific. As a consequence, the algorithm to accumulate the results before writing to main memory needs to be changed. Each time at least one of the threads moves to a different row, the entire warp executes a segmented scan (instead of a reduction) which accumulates the result for each row in the register of the first thread working on that particular row. At this point the local results of the remaining threads are reset to zero, while the first threads will update the global output vector once they are finished with their row. This eliminates all race conditions inside a warp, since each thread updates a different location of the output vector. Determining whether at least one thread moved to the next row can be realized in only one instruction per chunk by using warp vote functions.

Warp-level segment distribution also causes additional reads from `rowptr`, since each thread may need to move multiple rows after each chunk. However, as the last thread in a warp always has the most up-to-date information about the starting row of the next chunk, the number of reads can be reduced by broadcasting this information to the other threads within the warp using a single warp shuffle. Finally, in order to ensure aligned accesses to arrays `val` and `colidx`, and fully utilize each fetched cache line, the segment sizes can be restricted to an integer multiple of the chunk size. Since the chunk size is a multiple of the cache-line size, if `val` and `colidx` arrays are aligned, the start of each segment will also be aligned.

2.3.3 Determining the first row of each segment

At the beginning of the CSR-I algorithm each warp has to determine the first row of its segment. This can be done by first constructing a histogram of `rowptr` with T bins associated with the segments of `val` and `colidx`.

The number of elements n_i in each bin corresponds to the number of rows which end in the segment associated with this particular bin. Since the first row of segment k is equal to the number of rows ending in previous segments (i.e., $\text{srow}_k = n_1 + n_2 + \dots + n_{k-1}$), the indices of these first rows can be determined by computing the exclusive scan of the histogram.

In order to avoid repeating this computation at each SpMV invocation, the array `srow` can be saved and “attached” to the CSR matrix structure. We note that the optimal number of warps T does not depend on the matrix A , but only on the hardware-specific degree of thread-concurrency, adding a constant amount of storage overhead. Even though the procedure can be realized on the GPU in parallel, this is generally not needed, as its computational cost is very low compared with that of SpMV: the entire computation requires only one pass over `rowptr` and one over the resulting histogram, comprising a total of $m + T$ data accesses and integer operations. Instead, it can be performed sequentially on the CPU and overlapped with the (initial) memory transfer of matrix A to the GPU.

2.4 Experimental Evaluation

2.4.1 Setup and metrics

The GPUs used in the experiments cover a fair subset of recent compute capabilities from NVIDIA: 3.5 (“Kepler” K40) and 6.1 (“Pascal” GTX1080). Since the experiments run only on the GPU, the details of the host CPU are not relevant. Experiments on even older architectures (Fermi and earlier) are not possible since these GPUs do not support warp shuffle instructions required by the CSR-I algorithm. We use NVIDIA’s compilers in the CUDA toolkit 8.0, and report numbers for single precision (SP) and double precision (DP) arithmetic. All kernels are implemented using the CUDA programming model and are designed to be integrated into the MAGMA-sparse library, which is also leveraged as a testing environment. In addition, the CSR-I algorithm will be publicly available in a future version of MAGMA.

Among the implementations of SpMV based on CSR we compare the two variants from BG: CSR-s, which is implemented in MAGMA-sparse and an implementation of CSR-v taken from BG’s article, as well as the CSR algorithm from NVIDIA’s cuSPARSE library. Among the specialized formats, we include the implementations of SpMV for ELLPACK, ELLR-T and SELL-P from MAGMA-sparse, and that for the HYB format from cuSPARSE.

In order to obtain a comprehensive evaluation we compare the storage formats and SpMV implementations from the perspectives of performance and storage cost. For the performance, we report either the speed-up/slow-down relative to the CSR kernel from cuSPARSE or the absolute performance in terms of GFLOPS rate (billions of floating-point arithmetic operations, or flops, per second). The flop count for SpMV used for all examples is $2n_z$, even though some of the implementations of SpMV may actually perform a larger number of flops (because they operate with zero entries). All experiments are repeated 1,000 times and the average time of these runs is used in the calculations.

The CSR-I algorithm has one tunable parameter: the number of warps T launched to compute the SpMV. The optimal value for T is proportional to the degree of hardware concurrency, i.e. $T = l \cdot n_C / 32$, where n_C is the number of CUDA cores available on the GPU and l is the desired load per core. Our experiments reveal that the optimal load is $l=64$ for both the K40 and GTX1080 architectures and this setting is used for all experiments in this section.

We determine the storage requirements for the CSR, ELLPACK and ELLR-T formats from the basic properties of the matrix A as:

$$\mathcal{S}_{CSR}, \quad \mathcal{S}_{ELL} = m \cdot l_M (s_v + s_i), \quad \text{and} \quad \mathcal{S}_{ELLR-T} = \mathcal{S}_{ELL} + m \cdot s_i,$$

where l_M is the number of nonzero elements in the densest matrix row. Determining the storage requirements for the remaining two formats is more involved. For SELL-P we use a conversion routine from CSR to SELL-P implemented in MAGMA-sparse and modify each memory allocation to instead increase a counter by the amount that it was supposed to allocate. This is not possible for HYB, as the source code is not available. For this case, we use the `cudaMemGetInfo` routine from the CUDA Runtime API to get the total amount of free device memory before and after allocating the matrix in HYB format. The difference between the two values is the actual storage required by HYB. This strategy allows us to measure the storage consumption without

Figure 2.4: Storage consumption of different sparse matrix formats (left) and overhead compared to CSR for these formats (right). The data is shown for 100 selected matrices from SMC, assuming $s_v=8$ (double precision) and $s_i=4$.

Figure 2.5: GFLOPS distribution of SpMV implementations on K40 (top) and GTX1080 (bottom), using SP and DP arithmetic (left and right, respectively).

Figure 2.6: Comparison of SpMV implementations on K40 (top) and GTX1080 (bottom), using SP and DP arithmetic (left and right, respectively).

actually allocating the required data for all formats except HYB. Thus, we are able to evaluate the cost even if the problem does not fit into the GPU memory.

The experiments are carried out using a subset of the SuiteSparse Matrix Collection (SMC). Concretely, we first filtered the complete collection (2,757 problem instances) keeping only real-valued instances with $10^6 \leq n_z < 10^8$ (491 problems), and then randomly selected 100 cases³ among these (about 20% of the filtered problems and 3.6% of the complete collection). The limits for n_z were chosen to allow the utilization of the full processing potential of GPUs, while keeping the storage requirements low enough to fit the matrix into the GPU memory. We believe this is a representative subset of the problems for which a GPU accelerator can be beneficial, not being biased to any particular format.

2.4.2 Memory consumption

We commence our evaluation with an analysis of storage consumption of different matrix formats for the 100 selected matrices from SMC. Figure 2.4 shows that, for most cases, CSR is the format that requires the lowest amount of memory, and the additional storage required to save the `srow` array in CSR-I is negligible. HYB requires some additional memory, but this is still within a limit of $2\times$ compared with CSR. SELL-P performs quite poorly for some cases, consuming up to $11\times$ more memory than CSR; while ELLPACK and ELLR-T consume even up to 5 orders of magnitude more storage space in some cases. As a result, even though the storage required by CSR and HYB is under 1 Gbyte for all selected problems, the storage requirements can grow to 3 Gbytes for SELL-P and even to 100 Tbytes for ELLPACK and ELLR-T. This shows that the last two layouts cannot be considered as general formats. Since the focus of this work is on SpMV algorithms for general matrices, possibly with an irregular nonzero distribution, we omit ELLPACK and ELLR-T from the following experiments.

³The list of cases employed in the experiments can be downloaded from http://www3.uji.es/~flegar/2017_csri/matrices.txt.

Figure 2.7: Comparison between CSR and CSR-I implementations of SpMV on K40 (top) and GTX1080 (bottom), using SP and DP arithmetic (left and right, respectively).

2.4.3 Global comparison

The results in Figure 2.5 show the distribution of the GFLOPS rates by means of “box-and-whisker” plots. This experiment reveals that the median GFLOPS rate for our CSR-I format (red line inside the blue boxes) is similar to those of the specialized kernel for CSR in cuSPARSE, HYB and SELL-P; and all four present considerably higher GFLOPS medians than those observed for CSR-s and CSR-v. For this reason, we omit CSR-s and CSR-v from further discussion. Furthermore, the lower “whisker” attached to the boxes in Figure 2.5, which comprises the first quartile (i.e. 25%) of the cases, show that both CSR and SELL-P encounter a considerable number of “ill-conditioned” cases from the point of view of performance, delivering notably lower GFLOPS rates for those. In contrast, CSR-I and HYB feature a more consistent performance rate. This behaviour can also be observed in Figure 2.6. (The problem instances in this figure are ordered by speed-up/slow-down of CSR-I over cuSPARSE CSR.) For regular cases, appearing in the left-hand side of the plots, CSR-I is outperformed by all implementations due to its higher arithmetic intensity and use of atomic operations. In contrast, for irregular problems, in the right-hand side of the plots, the only implementation that matches its performance is HYB, which, in addition to higher storage consumption, is not suitable for other types of operations. We do not evaluate the cost of transformation from CSR to the other formats included in our experiments. For CSR-I, as discussed in the previous section, this cost is small, or even negligible if this transformation is overlapped with the first transfer of the matrix data to the GPU memory.

2.4.4 Detailed comparison of CSR and CSR-I.

As argued at the beginning of this paper, the specific goal for our CSR-I variant is to ensure an efficient execution of SpMV when the matrix exhibits an irregular row distribution of its nonzero entries, while maintaining the data layout of the regular version of CSR (and roughly its memory requirements). To close the experiments, we evaluate the performance of these two formats in more detail. Figure 2.7 illustrates the throughput of the CSR-I

Figure 2.8: Relationship between $s[n_{zr}]/E[n_{zr}]$ (x -axis) and speed-up/slow-down of CSR-I over CSR (y -axis) on K40 (top) and GTX1080 (bottom) using SP and DP arithmetic (left and right, respectively).

variant with respect to that of CSR from cuSPARSE for each problem instance. In these plots, we employ a logarithmic scale for the y -axis, and the problems instances are sequenced in the x -axis in increasing order of difference in favour of CSR-I. For three of the configurations: K40-SP and GTX 1080-SP/DP, CSR-I outperforms CSR in about 40–50% of the problems, and the difference in favour of the former comes in a factor that can raise more than $100\times$. Compared with this, the highest loss of CSR-I shows a factor that is at most $0.3\times$. For K40-DP CSR-I is superior for 24% of the problem instances. This is explained by the lack of hardware support for DP atomic updates in this architecture.

Even though CSR-I shows notable acceleration over CSR for a fair fraction of the problem instances, an optimal hybrid strategy is obtained if CSR-I is applied to compute SpMV for matrices in this subset only, while the operation relies on CSR for the remaining cases. Note that this is possible because CSR-I maintains the same structure as CSR, with just an additional vector to store the starting rows of each segment. In contrast, an attempt to combine CSR with any of the other GPU-specialized formats (HYB, SELL-P, ELLPACK, ELLR-T) would incur a considerable increase in the amount of stored information (even a complete duplication). Still, a relevant question is whether we can choose *a priori* to rely on either CSR or CSR-I for a particular SpMV. Figure 2.8 shows that this is indeed the case if we have a rough statistical estimation of the distribution of the number of nonzero entries per row n_{zr} . Concretely, the figure depicts the relationship between the performance of CSR-I over CSR and the standard deviation-to-mean ratio: $s[n_{zr}]/E[n_{zr}]$. The plots in the figure show a clear separation at $s[n_{zr}]/E[n_{zr}]=1$ for both architectures and precisions. For ratios greater than one, CSR-I is slightly slower for only one test matrix and shows a significant acceleration for the rest of the cases on GTX1080. The K40 GPU exhibits a similar behaviour, with only several cases slightly slower and the majority achieving significantly higher performance for ratios above this threshold. For ratios between 0.1 and 1, the faster algorithm depends on the matrix, but the majority of cases favour cuSPARSE CSR. For extremely regular sparsity patterns, with ratios below 0.1, cuSPARSE CSR is the clear winner.

2.5 Conclusions

We have re-formulated the parallelization of SpMV based on the CSR sparse matrix format to enforce a balanced partitioning of the data (and workload) on GPUs, optimized for matrices with an irregular row distribution of the non-zero entries. Our approach departs from the conventional parallelization across matrix rows advocated by standard implementations of CSR SpMV and other GPU-specific formats, instead facing potential race conditions via atomic transactions (supported by hardware in recent GPU architectures). Furthermore, our algorithm preserves the standard CSR format to store the sparse matrix, augmented with a vector which holds the row indexes of some key matrix elements. This additional array can be built inexpensively and consumes only a minor amount of additional memory.

Our experiments on two recent GPU architectures from NVIDIA, using both single and double precision arithmetic, show that our algorithm can be composed with the standard CSR SpMV to yield a GPU kernel that becomes a strong candidate for the implementation of SpMV in general-purpose sparse linear algebra libraries for this type of accelerators.

Acknowledgements

This work was supported by the CICYT project TIN2014-53495-R of the MINECO and FEDER and the EU H2020 project 732631 “OPRECOMP. Open Transprecision Computing”.

Bibliography

- [1] H. Anzt, S. Tomov, and J. Dongarra. Implementing a sparse matrix-vector product for the SELL-C/SELL-C- σ formats on NVIDIA GPUs. Technical report, The University of Tennessee at Knoxville, 2014.
- [2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA Corporation, 2008.
- [3] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA’09, pages 233–244. ACM, 2009.
- [4] D. Buono, J. A. Gunnels, X. Que, F. Checconi, F. Petrini, T.-C. Tuan, and C. Long. Optimizing sparse linear algebra for large-scale graph analytics. *Computer*, 48(8):26–34, 2015.
- [5] cuSPARSE. <http://docs.nvidia.com/cuda/cusparse/>, 2017.
- [6] T. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 1st edition, 2006.
- [7] 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. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014.
- [8] A. Langville and C. Meyer. *Google’s PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2009.
- [9] 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, pages 339–350. ACM, 2015.
- [10] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.
- [11] F. Vázquez, J. J. Fernández, and E. M. Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience*, 23(8):815–826, 2011.
- [12] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaSpMV: yet another SpMV framework on GPUs. *ACM SIGPLAN Notices - PPOPP’14*, 49(8):107–118, 2014.

Balanced Sparse Matrix-Vector Product for the COO Matrix Format

Published as: G. Flegar and H. Anzt. Overcoming load imbalance for irregular sparse matrices. In *Proceedings of the 7th Workshop on Irregular Applications: Architectures and Algorithms*, IA3'17, pages 2:1–2:8. ACM, 2017

In this paper we propose a load-balanced GPU kernel for computing the sparse matrix vector (SpMV) product. Making heavy use of the latest GPU programming features, we also enable satisfying performance for irregular and unbalanced matrices. In a performance comparison using 400 test matrices we reveal the new kernel being superior to the most popular SpMV implementations.

3.1 Introduction

Applying a discretized operator in terms of a sparse matrix-vector product (SpMV) is a heavily-used operation in many scientific applications. An example are the Krylov subspace methods relying on the SpMV kernel to generate the Krylov subspaces in which the solutions to linear and eigenvalue problems are approximated. At the same time, the SpMV is a frequent bottleneck in complex applications, as it is notorious for sustaining low fractions of peak processor performance. This is partly due to the low arithmetic intensity making the SpMV kernel memory bound on literally all modern architectures, the access overhead induced by storing only the nonzero elements in the matrix, the (in many cases random) access to the input vector. Given the importance of this building block, significant effort is spent on finding the best way to store sparse matrices and optimizing the SpMV kernel for different nonzero patterns and hardware architectures [2, 6, 11]. For sparse matrices where the nonzeros are distributed in a very structured fashion, it is often possible to derive problem-tailored storage formats, like, e.g., the DIA format for matrices with a tridiagonal structure [6]. A similar situation is given if the pattern is not very structured, but the nonzero elements are distributed equally across the rows (each row contains a similar number of nonzero elements). Most challenging are the sparsity patterns that are irregular (no recurring sub-pattern can be identified), and unbalanced (the distinct rows have a very different number of nonzero elements). Problems with these characteristics are typical for, e.g., social network representations. For these irregular problems, standard parallelization strategies, like assigning rows to the parallel resources, inevitably result in heavy load imbalance. Furthermore, unstructured sparsity patterns often promote random memory access to the vector values.

In this paper we present a GPU implementation of the sparse matrix-vector product (Section 3.3) that addresses the challenge of overcoming the load imbalance in unstructured matrices. The kernel is based on the coordinate (COO) format [6], leverages the latest features of the CUDA programming model, and succeeds in achieving high performance for unstructured matrices. In a comprehensive evaluation in Section 3.4 we identify the developed kernel as competitive or superior to the existing routines. Prior to presenting the new implementation, in Section 3.2 we review existing efforts for optimizing the sparse matrix vector product on manycore architectures.

Figure 3.1: Different storage formats for a sparse matrix of dimension $m \times n$ containing nnz nonzeros along with the memory consumption.

3.2 Related Work

3.2.1 Sparse Matrix Formats

In the BLAS and LAPACK [1] standard for dense linear algebra, matrices are stored as a sequence of columns, with each column stored as an array of its elements. This allows to easily locate or identify any matrix entry in memory. For matrices where most elements are zero, which is typical for, e.g., finite element discretizations, storing all matrix entries results in significant storage overhead. The computational cost of a matrix vector product increases as well, as a result of explicitly multiplying the zero entries with vector values. Sparse matrix formats aim at reducing the memory footprint (and computational) cost by storing only the nonzero matrix values. Some formats additionally store a moderate amount of zero elements to enable faster processing when computing matrix vector products. Obviously, storing only a subset of the elements requires to accompany these values with information that allows to deduce their location in the original matrix.

A straight-forward idea is to explicitly store only the nonzero elements, along with the row and column index of each element. This storage format, known as coordinate (COO [5]) format, allows to determine the original position of any element in the matrix without processing other entries.

Further reduction of the storage cost is possible if the elements are sorted row-wise, and with increasing column-order in every row. (The latter assumption is technically not required, but it usually results in better performance.) Then, the Compressed Sparse Row (CSR [5]) format can replace the array containing the row indexes with a pointer to the beginning of the distinct rows. While this reduces the data volume, the CSR format requires extra processing to determine the row location of a certain element.

On SIMD architectures, a performance-relevant aspect is to have uniform operations across the SIMD unit. This makes the ELL format [6] attractive for these architectures: In this format, each row is compressed to contain only the nonzero entries and some explicit zeros that are used for padding to enforce an equal length for all rows. The resulting value matrix is accompanied with a column index matrix which stores the column position of each entry in the compressed matrix. While typically increasing the storage cost compared to the CSR format, this removes the need for explicitly storing the row pointers. Furthermore, the column indexes (and values) in the distinct rows can be processed in SIMD fashion. Coalescent (SIMD-friendly) memory access is enabled if the value and column index matrices are stored in column-major order.

To reduce the memory overhead, the ELL format can be truncated to a version where only row-blocks with the height of the SIMD-length are padded to the same number of nonzero elements, but the rows in distinct blocks can differ in the number of nonzero elements. This “sliced ELL” (SELL-p [10]) format can be viewed as splitting the matrix into row-blocks and storing each block in ELL format. The formats discussed in this section are visualized in Figure 3.1.

Aside from these basic formats, there exist variants which arise as combinations of these basic formats: e.g., the hybrid format which stores the matrix partly in ELL and partly in CSR or COO.

3.2.2 SpMV on manycore architectures

Related to the storage format is the question of how to process the multiplication with a vector in parallel. The main challenges in this context are: 1) balancing the workload among the distinct cores/threads; and 2) allowing for efficient access to the matrix entries and the vector values. The second aspect is relevant in particular on NVIDIA GPUs where each memory access reads 128 contiguous bytes of memory [7]. In case of fine-grained

```

1 // input: A, x, y
2 // output: y = y+A*x
3 void coo_spmv(int nnz, const int *rowidx,
4               const int *colidx, const float *val,
5               const float *x, float *y)
6 {
7     for (int i = 0; i < nnz; ++i) {
8         y[rowidx[i]] += val[i] * x[colidx[i]];
9     }
10 }

```

Figure 3.2: COO SpMV kernel design.

parallelism, balancing the workload naturally results in multiple threads computing partial sums for one row, which requires careful synchronization when writing the resulting vector entry back into main memory.

The standard approach of parallelizing the CSR, ELL and SELL-p formats is to distribute the rows among distinct threads (or groups of threads) [6, 12]. For the CSR format, this works fair well for balanced sparsity patterns, but it can lead to severe load imbalance otherwise. Recently, a strategy for a load balanced CSR SpMV was proposed that parallelizes across the nonzero elements instead the rows [9]. The SpMV kernel we present in this paper is based on the COO format, which comes with the advantage of the row index of an element being readily available.

3.3 Design of the COO SpMV GPU kernel

The specific SpMV operation we target in this paper is $y := A \cdot x + y$, for $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$. This routine updates the vector y by adding the product of the sparse matrix A and the vector x , and allows for flexibility in terms of scaling y prior to the operation (i.e. scaling y with 0 to compute $y = A \cdot x$). It comprises $2 \cdot nnz$ arithmetic operations (with nnz being the number of nonzero elements in A).

In the rest of this section we describe the design of the COO SpMV kernel we propose for manycore architectures. Subsection 3.3.1 presents the general algorithmic idea, without introducing hardware-specific optimizations. There, we only assume the target device to be a shared memory architecture, with a relatively large number of computational elements (cores) and support for atomic addition of floating point values. The last assumption is required to resolve race conditions which can occur if multiple computational elements are assigned to the same matrix row, i.e. contribute to the computation of the same entry in the output vector. Furthermore, we assume that the memory design favours data locality over random data access, which is true for virtually all modern hardware. Subsection 3.3.2 describes the hardware-specific optimizations we employ when realizing the COO SpMV algorithm on NVIDIA GPUs using the CUDA programming model.

3.3.1 COO SpMV

The most natural approach to exploit hardware concurrency in an SpMV routine based on the COO format is to parallelize the loop traversing the nonzero elements in the matrix (line 7 in Figure 3.2) by splitting the workload into similar-sized subsets, and distributing those among the parallel resources. This corresponds to assigning a contiguous “chunk” of the array containing the matrix values (along with the corresponding chunks of column and row index arrays), to each computational element. While it is possible to use non-contiguous chunks (e.g., distribute the data in round-robin fashion) this could break data locality, resulting in performance loss. To ensure load balancing among different computational elements, all chunks should be of similar size. In addition, to reduce the number of memory transactions from main memory, it is important to aim for aligned memory access and to use every data element brought into cache. This is crucial when implementing the memory-bound SpMV kernel as its performance is largely dependent on the data access efficiency. Therefore, assuming that all three arrays describing the matrix in COO format are aligned in memory, each chunk should start at a cache line boundary, and ideally comprise an integer number of cache lines.

In summary, efficient data access and optimal load balancing imposes two restrictions on chunk sizes: 1) each chunk should be an integer multiple of the cache line size, and 2) the sizes of any two chunks should differ

by at most one cache line.

While the above strategy yields a perfect distribution of the input matrix A , which typically comprises the majority of the data, it has several implications we discuss next.

The core operation (line 8 in Figure 3.2) of COO SpMV is composed of a (fused) multiply-add between an element of A , and elements of x and y , indexed by the values in arrays `rowidx` and `colidx`. The element of y is then updated with the result of this operation. Since multiple computational elements can operate on matrix elements which have the same `rowidx` values (which corresponds to matrix elements located in the same row), this update is prone to race conditions. To resolve write conflicts, the (fused) multiply-add can be replaced by multiplication, followed by an atomic addition of the result into the correct position of vector y . This, however, requires more arithmetic instructions than the original approach, and uses a large number of expensive atomic operations, which may cause significant overhead in case of atomic collisions (i.e. multiple atomic operations requesting the same data entity at the same time). To alleviate the problem, we accumulate the results of several iterations of the for loop with the same update destination in registers private to the computational element. If the `rowidx` value of the next data element is different to what was processed previously, the accumulated results are written back to main memory using an atomic addition.

In this strategy, to decrease the number of atomic operations, (and increase the amount of computation handled in registers,) the three arrays comprising the matrix data should be sorted with respect to the increasing `rowidx` values. This will ensure that each computational element performs only one atomic operation per `rowidx` value, while also ensuring data locality for vector y (which can be beneficial on hardware that implements atomic operations in a shared cache file, like used in NVIDIA and AMD GPUs). Finally, a good heuristic to improve the access pattern to the input vector x is to additionally sort each set of matrix elements with the same `rowidx` value with respect to increasing `colidx` value. The effectiveness of this approach is highly sensitive to the sparsity pattern of the matrix, but this is a common problem of virtually all SpMV formats and algorithms. (The only exception to this problem known to the authors is the CSC format, where structured access to the vector x is ensured at the price of complicated access to the vector y .)

3.3.2 CUDA realization of COO SpMV

We realize the specific implementation of the general approach described in the previous subsection using the CUDA programming model. This model has all of the features required by the introductory paragraph of this section, with atomic instructions being the only critical component. While older generations of NVIDIA GPUs emulate double precision atomics in software (by using 64-bit atomic CAS), the new Pascal architecture offers native support for double precision atomics.

A naive implementation assigns one chunk of memory to each CUDA thread. This, however, inevitably results in non-coalescent reads to the matrix A , which is detrimental to performance. To ensure coalescent memory access, each chunk should be assigned to one warp (a group of 32 threads), and a thread i of the warp should read elements at positions $32j + i$, $j = 0, 1, \dots$ of the chunk. (This is the motivation to use a more platform-agnostic term “computational element” in the previous subsection, as the terms “thread” or “core” may have different meanings in distinct programming models and/or hardware.)

The main problem with this “warp-level approach” is the use of atomic operations. If each thread in the warp attempts to issue an atomic addition whenever it progresses to the next row, this will cause a large number of atomic collisions, since all the threads in a warp execute in lockstep (i.e. perfectly synchronized). A workaround is to conclude each iteration of the loop (line 7–9 in Figure 3.2) with a “warp vote function” in which all threads in a warp decide whether there is at least one of them that needs to write its results into global memory. If such a thread is identified, all threads collectively execute a warp-level segmented scan operation on their private registers, with segments defined by the distinct values of the `rowidx` array currently processed by the warp. The segmented scan (as opposed to a simple reduction) is required to ensure correct operation if the warp handles multiple rows of the matrix. For each of the handled `rowidx` values, the segmented scan determines the thread with the lowest thread index among all the threads that operate on this row. Then, these threads accumulate the partial sums present in their registers, and each of the threads with the lowest index will issue a global atomic addition before it progresses to a different `rowidx` value. This strategy avoids atomic collisions between threads of the same warp. Atomic collisions between threads of distinct warps are still possible, but these are unlikely as 1) threads of distinct warps operate on distinct data chunks (so the number of overlapping rows is limited) and 2) distinct warps are not perfectly synchronized. The segmented scan approach radically reduces the number of

Figure 3.3: Nonzero count vs. size for the considered test matrices. For convenience we added density baselines for 3 nnz/row, 10 nnz/row, and 50 nnz/row.

atomic collisions, however increases the number of arithmetic operations in the algorithm. As the SpMV kernel is heavily memory bound, it can be expected that additional arithmetic operations rarely impact the overall performance, as long as they can be overlapped with memory operations. For completeness, we mention that this approach of avoiding intra-warp atomic collisions was also used to construct a balanced CSR SpMV kernel in Flegar et al. [9].

An additional optimization on NVIDIA GPUs is related to the choice of the number of chunks. In contrast to the classical latency-minimizing CPU hardware, enabled by sophisticated cache hierarchy, NVIDIA GPUs use a latency-hiding approach where the computational units are oversubscribed with warps. The intention of having a larger number of warps active is to quickly switch in-between them to cover memory latency [3]. If threads in a warp issue a memory operation, those threads will stall while waiting for the memory transaction to complete. To combat this, rather than allowing the hardware to stall, the warp scheduler may find a warp that is not waiting for a memory operation to complete, and issue the execution of this warp instead. This constant juggling of active warps allows the GPU to tolerate the high memory latency and keep the compute cores occupied. To enable latency hiding, the number of generated chunks on this hardware should be higher than the amount of parallel processing resources. On the other hand, a high number of chunks increases the chance of atomic collisions as more chunks may contain data located in the same matrix row. We introduce an “oversubscribing” parameter ω that determines the number of threads allocated per each physical core (e.g., $\omega = 2$ means that the number of threads is two times larger than the number of physical cores, while $\omega = 4$ means that there are four threads assigned to each physical core). The oversubscribing parameter is subject to hardware- and problem-specific optimization, and we experimentally identify reasonable choices in Section 3.4.

3.4 Performance Assessment

3.4.1 Test matrices

For the experimental performance analysis, we use a set of 400 matrices from the SuiteSparse matrix collection [8]. This collection comprises a large number of matrices that differ in the algebraic field (real, complex, pattern), the shape (square, rectangular), and matrix-specific characteristics such as size and nonzero pattern. For the performance assessment we focus on real, square matrices that have a pairwise different nonzero pattern. In Figure 3.3 we visualize the size and nonzero count of the chosen test matrices. In order to quantify the imbalance of the nonzero distribution of a matrix, we use the standard deviation of the nonzero-per-row metric, see Figure 3.4.

Figure 3.4: Histogram for the (standard deviation / avg) of the nonzero-per-row metric. Few problems have a higher standard deviation than 10^2 .

3.4.2 Experiment setup

All experiments were conducted on the GPU-accelerated compute nodes of the PizDaint supercomputer at the Swiss National Computing Centre (CSCS). Although irrelevant for the performance analysis, we mention that the host composes of an Intel E5-2690 v3 processor (codename Haswell) with 12 cores running at 2.6 GHz. All computations are executed by the NVIDIA Tesla P100 GPU (compute capability 6.0) for which NVIDIA lists a double precision peak performance of 5.3 TFLOPs (10^{12} floating point operations per second). We use double precision in all experiments. The P100 is equipped with 16 GB main memory, which is accessed at a theoretical bandwidth of 732 GB/s. Using the bandwidth test that ships with CUDA 8.0 (and that puts equal pressure on memory reads and memory writes) we were able to achieve 497 GB/s. Using NVIDIA’s CUDA toolkit version 8.0, we design the C00 kernel to integrate into the MAGMA-sparse software library [4]. MAGMA-sparse is also used as experiment ecosystem, and provided the SpMV reference implementations. Specifically, the reference implementations are:

CSR The CSR-based SpMV kernel we consider is part of NVIDIA’s cuSPARSE library.

CSR5 The CSR5 SpMV kernel is based on modifying the CSR format for achieving higher performance. The implementation is part of the MAGMA-sparse software stack, details about the kernel are presented in Liu et al. [11].

CSRI The design of the CSRI SpMV kernel is very similar to the C00 kernel we propose. It tries to enable load balancing for unbalanced matrices stored in CSR by using atomic addition operations [9].

HYB The hybrid SpMV kernel we consider combines the ELL format for the regular (balanced) part of the matrix with the COO format for the irregular part of the matrix. We use the implementation available in NVIDIA’s cuSPARSE library.

SELLP The SELL-p kernel is also part of the MAGMA-sparse software ecosystem, and has proven to be very efficient for balanced problems [12].

3.4.3 Experimental results

In a first experiment, we analyze the effect of the oversubscribing-parameter ω on the performance of the C00 kernel. In Figure 3.5 we order the matrices for increasing nonzero count, and report the performance for the ω -values 2, 4, 8, 16, 32, 64, and 128. We notice that the performance differences increase with the nonzero count. For small nonzero counts, the differences are negligible. The optimal choice for the ω parameter then takes turns: Moderate oversubscribing with $\omega = 8/\omega = 16$ is the performance winner for systems with about

Figure 3.5: Evaluating the effect of the parameter ω on the performance of the C00 kernel. The matrices are ordered according to increasing nonzero count.

Figure 3.6: Performance of the distinct SpMV kernels for all problems included in the test suite. The matrices are ordered according to increasing nonzero count.

10^5 nonzeros, $\omega = 32/\omega = 64$ is superior for systems with about 10^6 nonzeros, and $\omega = 128$ seems to be the best parameter choice beyond that. The nonzero count of a matrix is one of the characteristics known prior to the SpMV invocation. Hence, a straight-forward optimization step for the C00 kernel is given by choosing ω on a heuristic derived from Figure 3.5. In the rest of the paper we define the C00 as the kernel that chooses

$$\omega = \begin{cases} 8 & \text{for } nnz < 10^5, \\ 32 & \text{for } 10^5 \leq nnz < 10^6, \\ 128 & \text{for } 10^6 \leq nnz. \end{cases}$$

Next, we compare the C00 kernel with the reference kernels previously listed. In Figure 3.6 we order the test matrices according to increasing nonzero count, and visualize the performance of all SpMV kernels we consider in this analysis.

Independent of the SpMV kernel, the performance linearly increases with the nonzero count of the problems until it stagnates around 90 GFLOPs.

Furthermore, the visualization suggests that the C00 kernel is the fastest kernel for almost all test matrices with less than $2 \cdot 10^5$ nonzero elements. For problems containing more nonzero elements it is difficult to identify an overall winner. This is partly because multiple performance indicators are covering each other, and distinct problems, although different in sparsity pattern, may have the same nonzero count, and are therefore arranged in the same place on the x-axis. Overall, it is difficult to extract from this figure for how many problems a specific kernel is the fastest. We answer this question in Figure 3.7 where we report for how many problems a certain kernel was the performance winner (blue bar) and for how many problems a certain kernel gave the worst performance (red bar). If the kernel was neither the fastest nor the slowest for a certain problem, it is counted as “ballpark.” In the end, for each kernel we get a bar of the same length split into three colors; the sum of all blue parts and the sum of all red parts equals the number of test cases, respectively.

Figure 3.7: Fastest kernel comparison: blue bars represent the number of problems for which the kernel was the fastest, red bars count the problems for which the kernel was the slowest in the comparison.

Figure 3.8: Performance statistics for the distinct kernels over the test cases containing more than $2 \cdot 10^5$ nonzero elements.

Overall, the COO kernel wins most cases, and it is followed by the SELLP SpMV. CSR, CSRI, and HYB win only few cases, CSR5 not a single one. On the other hand, CSR5 and CSR are rarely the slowest kernels, while COO, CSRI, SELLP, and HYB lose significantly more cases.

Looking at the complete test suite containing all 400 matrices, we include problems that are “small” with respect to the computational workload of the SpMV. For those, even the winning kernel in Figure 3.7 achieves only low execution performance. In scientific applications, these “easy” problems are typically rather handled via a direct solver than an iteration method based on the SpMV kernel. As we are in particular interested in the performance for problems that are the characteristic target we limit the further analysis to the 248 problems containing more than $2 \cdot 10^5$ nonzero elements.

In Figure 3.8 we compare the distinct SpMV kernels in the GFLOPs metric for the problems containing more than $2 \cdot 10^5$ nonzero elements. We accompany this graph with some numeric information in Table 3.1 where we additionally list the average performance. For this metric, the COO format turns out to be the overall winner with an average 38.86 GFLOPs. Looking at the median, the SELLP and COO kernels achieve the highest execution rate (38.64 GFLOPs and 37.24 GFLOPs, respectively). They outperform the closest competitor CSR by about 25%. The lowest median performance of 18.74 GFLOPs is achieved by the HYB kernel. At the same time, the HYB kernel achieves significantly higher performance (up to 82.43 GFLOPs) for specific test cases, see Table 3.1. Only the SELLP and the CSR kernel achieve higher performance for balanced and regular problems (82.62 GFLOPs and 87.43 GFLOPs, respectively).

Most noticeable in Figure 3.8 is the variation of the COO performance being radically smaller than for any of the other formats: 50% of the performance numbers are within a 6 GFLOPs range from the median, the standard deviation is 9.16. The lowest performance number for the COO kernel is 24 GFLOPs. Only the CSRI

Kernel	min	max	average	median	standard-dev.
C00	24.29	64.32	38.86	37.24	9.16
CSR	0.07	87.43	32.77	30.43	20.07
CSR5	9.66	75.56	31.79	27.15	15.58
CSRI	13.47	81.21	31.85	26.84	14.44
HYB	6.64	82.43	27.98	18.74	20.22
SELLP	0.06	82.62	36.42	38.64	22.46

Table 3.1: Statistical information on the GFLOPs metric of the SpMV kernel for the 248 matrices containing more than $2 \cdot 10^5$ nonzero elements.

Figure 3.9: Runtime overhead of the distinct SpMV kernels. For each kernel, the matrices are ordered with respect to increasing overhead. Only test matrices with more than $2 \cdot 10^5$ nonzero elements are considered.

kernel is competitive in handling unbalanced problems, with 50% of the performance numbers between 20 GFLOPs and 60 GFLOPs, a standard deviation of 14.44, and the lowest performance being 13.47 GFLOPs (see Table 3.1). For SELLP and CSR, the performance values spread across a large range. In particular, the lowest performance numbers (0.06 GFLOPs for SELLP and 0.07 GFLOPs for CSR) are far from the median. The central box (upper/lower quantiles) are for these kernels a multiple of those for C00. Also the upper and lower whiskers are significantly further apart. For SELLP, this is expected as, for unbalanced matrices, the nonzero padding to a block-uniform nonzero count introduces significant performance-detrimental overhead, as well as load imbalance in-between the matrix blocks. Load imbalance is also the culprit for CSR’s poor performance. Hence, the formats delivering the best performance for balanced test matrices are not suitable for irregular unbalances problems. The C00 format, although achieving only 64.32 GFLOPs for the best case, proves to handle irregular problems well, with the highest average performance, a competitive median, the smallest variation, and the highest minimal performance.

Finally, we want to assess the performance penalty of choosing one specific kernel vs. choosing the problem-specific best kernel. Obviously, the problem-specific best format is unknown a priori, and one would have to test all kernels prior to the the performance-relevant run, or use machine learning techniques for making a good guess. Again, we focus on the problems containing more than $2 \cdot 10^5$ nonzero elements. In the analysis we identify the optimal format for each test matrix, and scale the performance of all kernels to this baseline. We then sort the matrices with increasing overhead for every kernel individually, and visualize this characteristic curve in Figure 3.9. Hence, the order of the matrices is different for each kernel, but the overheads are increasing in all datasets. The objective of minimizing the slowdown corresponds to minimizing the area below the curve. The longer a curve stays at 1, the more test cases a certain kernel wins. We notice that the overhead stays low particularly for C00, CSRI, and CSR5. For SELLP and CSR, the initially moderate overhead for balanced problems quickly grows for irregular problems. HYB deals better with these cases, however it already starts of with a larger overhead than C00 and the CSR variants. Overall, C00 has a radically lower overhead than any of the competitors. Another key observation is that (ignoring two outliers) the C00 kernel never exhibits

a slowdown factor larger than two. This implies that choosing the COO format results in an SpMV kernel which is in the worst case two times slower than the (unknown) optimal choice. This clearly makes COO the overall winner in this metric as well.

3.5 Summary and Outlook

We addressed the challenge of overcoming the load-imbalance in the sparse matrix vector product for irregular matrices. We developed and implemented an SpMV kernel for GPUs that is based on the COO format. Using a large collection of test matrices we compared the performance of the new kernel to the (up to our knowledge) best SpMV kernels available: the CSR and hybrid kernels which are part of NVIDIA’s cuSPARSE library, the SELL-P and CSR5 kernels part of the MAGMA-sparse software library, and the CSRI kernel, which balances the workload via atomic operations. We used different metrics to quantify the performance: median absolute performance in GFLOPs and its variation, kernel winning most test cases, and smallest overhead compared to the best kernel included in the test suite. For the chosen test suite containing 400 matrices, the proposed COO-based SpMV performs radically better on irregular matrix structures, and ultimately wins all considered performance metrics.

In the future we want to focus on multi-GPU architectures and optimize the developed kernel for hybrid (Multicore+Manycore) execution. Furthermore, we are convinced that the strategies reducing the impact of global write conflicts via warp-vote functions and introducing additional operations are also applicable to other computational problems of irregular nature.

Acknowledgments

H. Anzt was supported by the “Impuls und Vernetzungsfond” of the Helmholtz Association under grant VH-NG-1241. G. Flegar was supported by projects TIN2014-53495-R of the Spanish *Ministerio de Economía y Competitividad* and the EU H2020 project 732631 OPRECOMP.

The authors want to acknowledge the access to the PizDaint supercomputer at the Swiss National Supercomputing Centre granted under the project #d65.

Furthermore, the authors would like to thank Enrique S. Quintana-Ortí for comments on an earlier version of the paper.

Bibliography

- [1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, SC ’90, pages 2–11. IEEE, 1990.
- [2] H. Anzt, S. Tomov, and J. Dongarra. Implementing a sparse matrix-vector product for the SELL-C/SELL-C- σ formats on NVIDIA GPUs. Technical report, The University of Tennessee at Knoxville, 2014.
- [3] H. Anzt, E. Chow, and J. Dongarra. On block-asynchronous execution on GPUs. Technical report, LAPACK Working Note, 2016.
- [4] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler. Preconditioned Krylov solvers on GPUs. *Parallel Computing*, 68:32–44, 2017.
- [5] R. Barrett, M. Berry, T. F. 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*. SIAM, 2nd edition, 1994.
- [6] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA Corporation, 2008.
- [7] CUDA Toolkit v8.0. <https://docs.nvidia.com/cuda/>, 2017.

- [8] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.
- [9] G. Flegar and E. S. Quintana-Ortí. Balanced CSR sparse matrix-vector product on graphics processors. In *Proceedings of the 23rd International European Conference on Parallel and Distributed Computing, Euro-Par 2017*, pages 697–709. Springer, 2017.
- [10] 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. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014.
- [11] 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*, pages 339–350. ACM, 2015.
- [12] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers, HiPEAC’10*. Springer-Verlag, 2010.

Matrix-Vector Product Algorithms for Individual Streaming Multiprocessors

Published as: H. Anzt, G. Collins, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. Flexible batched sparse matrix-vector product on GPUs. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA'17, pages 3:1–3:8. ACM, 2017

We propose a variety of batched routines for concurrently processing a large collection of small-size, independent sparse matrix-vector products (SpMV) on graphics processing units (GPUs). These batched SpMV kernels are designed to be flexible in order to handle a batch of matrices which differ in size, nonzero count, and nonzero distribution. Furthermore, they support three most commonly used sparse storage formats: CSR, COO and ELL. Our experimental results on a state-of-the-art GPU reveal performance improvements of up to 25× compared to non-batched SpMV routines.

4.1 Introduction

Applying an operator discretized as a sparse matrix in terms of a sparse matrix-vector product (SpMV) is a heavily utilized kernel in many scientific applications. A practical example are Krylov subspace methods, which rely on SpMV to generate the Krylov subspaces used to approximate the solution of linear systems and eigenvalue problems. At the same time, SpMV frequently poses a performance bottleneck of sparse linear algebra algorithms, as this memory-bounded operation is notorious for delivering low fractions of peak performance on current computer architectures. Given the importance of SpMV, significant effort has been spent on finding the best strategy to store sparse matrices and optimizing this kernel for distinct nonzero distributions and hardware architectures, including multicore processors and graphics processing units (GPUs).

In general, scientific applications require the multiplication of a single, large and sparse matrix with an input vector. In this paper, we address a different scenario composed of the multiplication of a large set of “small” sparse matrices with their corresponding vectors. Although this use case is less prominent, it occurs for example in the context of astrophysics simulations. Our goal is to make the community aware that, under these circumstances, replacing a standard routine with a “batched” SpMV kernel often results in significant performance improvements. Following a brief discussion of related work in Section 4.2, Section 4.3 presents different strategies for processing a batch of SpMV calls/sparse matrices on GPUs via a number of flexible routines that are designed to handle the most commonly-used sparse matrix storage formats. In Section 4.4 we then assess the performance of the new kernels, by comparing them against the standard implementations of SpMV in cuSPARSE [7] and MAGMA-sparse [2]. While all kernels are tested on a production line GPU, it can be expected that the kernel design as well as the benefits carry over to other architectures. We conclude in Section 4.5 with some remarks and an outlook on future research directions.

Figure 4.1: Basic storage formats for an $m \times n$ sparse matrix with nnz nonzeros along with memory consumption.

4.2 Related Work

4.2.1 SpMV on manycore architectures

Improving the performance of SpMV on modern architectures is an active field of research. A critical factor is the selection of an appropriate sparse matrix format, which reduces the storage cost by maintaining only the nonzero values but has to keep some additional information in order to derive the location of the elements.

The simplest idea is to explicitly store only the nonzero elements along with the row and column indices (i.e. coordinates) of each element. This coordinate (COO) format [5] allows to determine the original position of any element in the matrix without processing any other entries.

If the elements are sorted row-wise and, for performance reasons, in increasing column-order within each row, the storage cost can be reduced. The “Compressed Sparse Row” (CSR [5]) format replaces the array containing the row indices with a pointer to the beginning of the distinct rows, reducing the amount of data required to store the row information, at the cost of additional processing necessary to determine the row location of the elements.

While being very popular for manycore architectures in general, the ELL format [6] is particularly suited for GPUs. In this layout, the distinct rows are padded with zeros to ensure they are all of the same length. While this typically increases the storage cost, it removes the need to maintain the row pointers, and enables processing the column-indices (and values) in distinct rows in SIMD fashion. Furthermore, coalescent memory access is attained if the matrix containing the nonzero elements is stored in column-major order.

The three basic formats targeted in our batched kernels are illustrated in Figure 4.1. In addition to these basic formats, there exist many other variants, which often arise as a combination of the basic formats. For example, the hybrid format stores the matrix partly in ELL and partly in CSR/COO; and the sliced ELL format (SELL-p) [12] chops the matrix into row blocks and stores each block in ELL format.

Related to the storage format is the question of how to parallelize SpMV. The main challenges in this context are: 1) balancing the workload among the distinct cores/threads; and 2) ensuring an efficient access to the matrix entries and the vector values. The second aspect is in particular relevant on NVIDIA GPUs where each memory access reads 128 contiguous bytes of memory [7]. In case of fine-grained parallelism, balancing the workload naturally results in multiple threads computing partial sums for one row, which requires careful synchronization when writing the result entry back into main memory. In this paper, we exclusively focus on (batched) SpMV implementations for GPU architectures. For a comprehensive overview about the CUDA programming model and its implications, see [3, 7].

4.2.2 Batched routines

The development of specialized routines for an operation involving many problems of small size that are pairwise independent, and can thus be handled simultaneously, has recently gained a lot of attention due to their heavy application in machine learning [1]. The motivation for designing these kernels is that the amount of hardware concurrency in manycore processors such as GPUs often exceeds the level of parallelism exploited by conventional routines. Consequently, handling the distinct problems in sequence utilizes only a fraction of the hardware resources and incurs a significant kernel launch overhead. In response to this, there are several efforts among the high performance computing community to standardize the interface for a batched version of the basic linear algebra subprograms (BLAS) and more complex functionality built on top of it [10].

4.3 Design of flexible batched SpMV kernels for GPUs

4.3.1 Flexible batched SpMV

A batched sparse SpMV for scientific applications often comes with some boundary conditions, which allows optimizing the kernel for this specific setting. Examples are situations where all SpMV operations of the batch have:

- the same system size (which, as long as the sparsity pattern does not change too drastically, allows to use explicit zero padding to fix the sparsity pattern);
- the same nonzero-per-row distribution (allows reuse of row pointers/row indices);
- the same nonzero locations (allows reuse of row pointers/row indices and column indices);
- the same values but distinct sparsity patterns (allows reuse of the values);
- the same matrix scaled by a scalar (which allows to rewrite the batched SpMV as a sparse matrix-matrix product and scaling the column of the distinct vectors).

In our case, we design our *flexible* kernels to tackle the most general case: the systems can differ in size, nonzero count, nonzero distribution, and values. This solution offers greater flexibility, and even allows to process a batch of problems coming from several concurrently-running applications.

4.3.2 GPU kernel design

A central aspect in the design of the batched kernels is the optimization of the access to the vectors. For this purpose we initially read the vectors into shared memory, which significantly reduces the cost of accessing them in the multiplication phase. As shared memory access is limited to the thread block, it is a natural choice to assign one thread blocks to each problem of the matrix batch. We design the batched SpMV kernels with focus on matrices of size up to 1,024 rows. Technically it is possible to process also larger systems, but targeting batches containing small matrices makes this a reasonable design choice.

In this paper we implement and compare six kernels for processing a batch of multiple SpMVs, with matrices stored in either CSR, COO or ELL format. While the properties of the COO and ELL formats inherently result in balanced workload distributions for each problem, we consider four implementations for the CSR format that use different strategies to balance the work.

Currently, all implementations use one CUDA thread block per problem. Thus, the same amount of resources, in terms of shared memory and number of threads, is allocated to each problem in the batch. We recognize this can result in workload imbalance in-between the distinct problems. However, optimizing the resource allocation to the characteristics of the distinct matrices in the batch remains outside the scope of this work.

4.3.3 COO

The first listing in Figure 4.2 (routine SpMV_COO) offers a sequential implementation of the SpMV kernel based on COO. The natural approach to exploit hardware concurrency in this case is to parallelize the loop traversing the nonzero elements in the matrix (line 2 in the code). This strategy comes with two advantages: 1) the data access to the matrix is coalescent; and 2) the workload is perfectly balanced. The disadvantage is that multiple threads may be assigned to elements located in the same row, and careful synchronization is necessary to ensure the partial sums are handled correctly. In order to ensure correct synchronization, in our batched implementation we combine atomic operations with thread-local partial sums. Moreover, each thread, typically handling multiple elements located in the same row, does not write its partial sum to global memory until all elements of the row are processed. In addition, intra-warp atomic collisions are avoided using warp-local segmented scans before each write. In the remainder of the paper, we use the abbreviation COO when referring to the flexible batched SpMV routine based on COO.

```

1 void SpMV_COO(int nnz, int *rowidx, int *colidx, float *val, float *x, float *y) {
2     for (int i = 0; i < nnz; ++i) {
3         y[ rowidx[i] ] += val[i] * x[ colidx[i] ];
4     }
5 }

1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2     for (int i = 0; i < m; ++i) {
3         for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
4             y[i] += val[j] * x[ colidx[j] ];
5     }
6 }

1 const int T = thread_count;
2 void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
3     int row = -1, next_row = 0, nnz = rowptr[m];
4     for (int k = 0; k < T; ++k) {
5         for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
6             while (i >= next_row) next_row = rowptr[++row+1];
7             y[row] += val[i] * x[ colidx[i] ];
8         }
9     }

1 void SpMV_ELL(int m, int max_nnz, int *colidx, float *val, float *x, float *y) {
2     for (int i = 0; i < m; ++i) {
3         for (int j = 0; j < max_nnz; ++j) {
4             y[i] += val[i+j*m] * x[ colidx[i+j*m] ];
5         }
6     }
7 }

```

Figure 4.2: Sequential C implementations of basic SpMV algorithms.

4.3.4 CSR

A simple parallelization of SpMV based on CSR is obtained by mapping the distinct rows to different threads. This corresponds to parallelizing the outer for-loop in the second listing in Figure 4.2 (SpMV_CSR). This variant was first described for GPUs in [6], under the name CSR-scalar. Even though CSR-scalar does not require any synchronization, it typically suffers from noncoalescent memory accesses for matrices containing more than one nonzero per row. This flaw becomes more apparent with increasing matrix density. Additionally, for unbalanced nonzero distributions, CSR-scalar exhibits severe workload imbalance as, after processing their rows, all threads of a warp remain idle until the thread processing the densest row has completed its work.

To alleviate the issues with CSR-scalar, the authors of [6] proposed an alternative implementation: CSR-vector, which maps each row to one warp (group of 32 threads). This strategy removes two drawbacks at a time: Assigning a warp to a row allows for coalescent memory access; and it improves the workload balancing for irregular sparsity patterns. However, for “very sparse” matrices containing only few nonzeros per row, CSR-vector wastes a significant amount of computational resources.

CSR-smart aims to alleviate the drawbacks of CSR-scalar and CSR-vector while combining their strengths. This kernel, recently implemented in the CUSP library [8], allocates a “vector” of threads (a subset of a warp) to process each row. The number of threads in a vector is determined at runtime. The strategy extracts the average number of nonzeros per row from the input matrix, and sets the vector size to the smallest power of two equal to or larger than this number (up to 32). Although this approach may render some workload imbalance for irregular sparsity patterns, it resolves both the CSR-scalar’s noncoalescent memory reads as well as the problem of idle resources in CSR-vector’s. In our batched CSR-smart kernel we calculate the vector length for each problem individually. This allows that thread blocks launched by the same kernel process different matrices of the batch with different vector lengths.

For convenience, we will use the abbreviations CSR_scal, CSR_vec and CSR_smart to refer to the flexible batched implementations of the CSR-scalar, CSR-vector and CSR-smart kernels, respectively.

4.3.5 CSR-I

A reorganization of the SpMV loops as shown in the third listing of Figure 4.2 (SpMV_CSR-I) can yield a perfectly balanced implementation for the CSR format [11]. Concretely, by parallelizing the outer loop of this variant (line 4), each warp is assigned the same percentage of nonzero elements. This mimics C00, and makes CSR-I especially appealing for irregular sparsity patterns (hence the “I” in the name). However, other CSR variants can be expected to outperform CSR-I [11] for regular sparsity patterns as the latter: 1) requires atomic operations for synchronizing the distinct warps writing to the same output vector location; 2) exhibits higher arithmetic intensity to minimize the amount of atomic collisions; 3) potentially reads some elements of the row pointer multiple times if the majority of rows have few nonzeros; and 4) requires a preprocessing step to determine the starting value of the row variable for each warp. In the batched CSR-I implementation (CSRI), the preprocessing step occurs once for each matrix of the batch, while every invocation of the CSRI kernel on the same matrix batch will reuse this information. As many applications require a high number of SpMV calls, (e.g., iterative solvers,) we do not account for the runtime of this preprocessing step in the performance measurements in Section 4.4.

In the original non-batched CSR-I implementation, the optimal level of thread concurrency is selected depending on the (single) problem characteristics and the hardware resources. A straight-forward approach in the batched CSR-I implementation distributes the resources equally across the problems in the batch. However, in the limit of increasing batch size, this strategy assigns one warp to each problem. At the same time, the amount of shared memory required per problem remains constant, i.e. equal to the size of the problem. The shared memory thus becomes the factor that constrains the number of thread blocks which can run concurrently on each multiprocessor of the GPU. Therefore, to prevent low occupancy, the number of threads assigned to each problem is not allowed to drop below the point where the shared memory becomes the occupancy-limiting factor.

4.3.6 ELL

The implementation of the flexible batched SpMV kernel based on the ELL format (we denote the batched kernel “ELL”) is an immediate derivation of the standard ELL SpMV from [6]: Each thread of the block processes a different row, forming the partial sums of its row in thread-local memory, while the thread block traverses the column indices and values from left to right. After completion, the intermediate results are written into the output vector locations in global memory. Conversely to the standard ELL kernel, the values of the input vector are read from shared memory, and the thread block size is adjusted to the matrix size such that each thread block handles one problem of the matrix batch.

4.4 Performance Evaluation

4.4.1 Experiment setup

For the performance analysis, we use the following test benchmark consisting of 32 matrices from the SuiteSparse matrix collection [9]: PIVTOL, TOMOGRAPHY, WEST0989, G45, GD02_A, SI2, CK656 EX25, JGL011 LOCK_700, FS_541_1, MBEACXC, DWT_918, MCFE, DWT_607, RBSA480, GR_30_30, BCSSTK34, CAGE8, EX27, CAN_838, BCSSTM34, USAIR97, BP_1600, ROTOR2, MSC00726, NOS3, G2, DWT_992, EX2, TOLS90, BCSSTK02. We focus on square matrices of order up to 1024, with real entries, including a variety of problems to cover a large spectrum. Concretely, this subset contains matrices with size $n \in [11, 1015]$, number of nonzeros $nnz \in [76, 38352]$, and $nnz/n \in [3.0, 66.0]$. The specific operation we target is $y := A \cdot x + y$, which requires $2 \cdot nnz$ floating-point arithmetic operations (flops), and allows for much flexibility in terms of scaling y before the operation (e.g., scaling y with 0 to compute $y = A \cdot x$).

The performance evaluation is split into three parts. In the first experiment, we quantify the performance advantages that the flexible batched routines provide over the standard SpMV kernels when processing a homogeneous collection (batch) consisting of an increasing number of identical matrices.

In the second part, we compare the batched kernels against each other, using homogeneous batches consisting either of the problems from the test benchmark, or custom-engineered matrices where we control the density and nonzero distribution. Although it is possible to write a much more efficient kernel for this problem

Figure 4.3: Performance of the standard and the flexible batched SpMV routines for homogeneous batches.

setting (which, in particular, reuses the information about the problem size and the nonzero locations), we argue that this experiment is useful to extract information about which format and kernel to choose for batches containing similar problems.

In the third part of the experimental analysis, we consider batches comprising problems with different characteristics. First, we look into settings where all matrices in the batch are of similar size but differ in the sparsity pattern. For this purpose, we select 12 matrices from the test benchmark of order 800–1,024, and create the batch by appending these problems in random order. Second, we consider batches containing any of the matrices in the test benchmark.

All experiments were conducted on the compute nodes of the PizDaint supercomputer at the Swiss National Computing Centre (CSCS). Although irrelevant for the performance analysis, the host contains an Intel E5-2690 v3 (Haswell) processor with 12 cores. All computations were executed by the NVIDIA Tesla P100 GPU (compute capability 6.0), using double precision arithmetic, for which NVIDIA lists a peak performance of 5.3 TFLOPs (10^{12} flops/second). The P100 is equipped with 16 GB of main memory that are accessed at a theoretical peak bandwidth of 732 GB/s. Using NVIDIA’s CUDA toolkit version 8.0, we designed the flexible batched routines to be integrated into the MAGMA-sparse software library [4]. MAGMA-sparse was also used

Figure 4.4: Performance of the standard and flexible batched CSR-based SpMV routines for a homogeneous batch consisting of 1000 square matrices of order 1024 with controlled density and nonzero distribution. The nonzeros are either distributed equally among the rows (left) or accumulated in few rows (right).

as experiment ecosystem, and provided the standard SpMV reference implementations.

4.4.2 Experimental results

We first quantify the benefits of leveraging custom-designed batched kernels over the standard SpMV routines when processing batches of small matrices. For this purpose, we select 12 problems with between 800 and 1,024 unknowns from the test benchmark, and create homogeneous batches. In Figure 4.3 we visualize the performance achieved by the standard SpMV routines versus the flexible batched SpMV kernels when processing batches of increasing size. In order to process a batch with multiple matrices via the standard SpMV kernels, we loop over the kernel invocations. For the standard CSR kernel, MAGMA-sparse simply interfaces to NVIDIA’s cuSPARSE library [7]; the standard ELL kernel is the implementation available in MAGMA-sparse [2].

The results of this experiment reveal that the performance of the standard SpMV kernels never exceeds 5 GFLOPs. Furthermore, although there is no clear winner among the batched SpMV kernels, they all complete the operation at least $10\times$ faster than their standard counterparts. For balanced problems, such as DWT992, GR_30_30, and NOS3, the performance of the ELL kernel surpasses 70 GFLOPs, a rate which is unmatched by any other kernel. At the other end of the spectrum, the CSRI kernel achieves very good performance for unbalanced problems containing many nonzero elements, such as EX25 and MSC. In these cases, the ELL kernel suffers from a significant zero-padding overhead. For the very sparse problem WEST0989 the fastest options are CSR_scal and C00. Overall, we acknowledge that the C00 kernel achieves very good performance across the complete test suite. In addition to being the fastest option in most of the cases, the C00 kernel is the second-best choice in all remaining cases where a different format is superior.

Next, we focus on the CSR format, for which we developed four kernels that differ in how they balance the workload. For reference, we also include the standard CSR SpMV from NVIDIA’s cuSPARSE in this analysis. For the next experiment we generate a homogeneous batch containing 1,000 square matrices of size 1,024 and vary the density and nonzero distribution. We analyze the performance in relation to the average number of nonzero elements per row. On the left-hand side of Figure 4.4, we test a balanced nonzero distribution. The results show that the batched CSR_scal offers very good performance for low nonzero-per-row ratios. This comes from the fact that the data reads are mostly coalescent. The performance of CSR_scal drops in case there are more than 4 nonzeros per row, while the performance of CSR_vec continues to improve. CSR_smart is a good trade-off between CSR_scal and CSR_vec, as it provides between 35 and 60 GFLOPs for most of the tested scenarios. The sequence of standard CSR SpMV calls never delivers more than 5 GFLOPs. Especially for increasing nonzero count, the performance of CSRI is competitive or even superior to CSR_smart. However, for low nonzero-per-row ratios, CSR_scal and CSR_smart are faster. Conversely, on the right-hand side plot of Figure 4.4, CSRI gives the best performance in all cases. This is expected as this experiment configures a batch of extremely unbalanced matrices with the nonzeros accumulated in a few rows. We recall that the good performance that CSRI achieves for this problem comes at the price of a preprocessing step to calculate the

Figure 4.5: Performance of the flexible batched SpMV routines for all matrices in the test benchmark ordered in increasing nonzero-per-row ratio.

balancing information.

In Figure 4.5 we analyze the performance of the flexible batched SpMV kernels achieved for a homogeneous batch of 10000 matrices of the test benchmark. The matrices are ordered in the x -axis according to increasing nonzero-per-row ratio. A larger value of this parameter makes the CSR_scal kernel less attractive while the performance of CSR_vec increases with the density. CSRI and CSR_smart outperform CSR_vec and CSR_scal in most cases. None of the CSR-based kernels is competitive to the COO kernel, which can be identified as overall winner in this experiment. Only for balanced matrices, the ELL kernel outperforms all other competitors. However, the performance of ELL is very problem-dependent, and for unbalanced nonzero distributions, it yields low performance.

We now turn to heterogeneous batches. First, we compose a batch with the matrices we analyzed in Figure 4.3. These matrices are very different in their nonzero pattern, but they all share similar size (800–1024 rows/columns). In Figure 4.6 we show performance (left) and bandwidth (right) for the distinct batched SpMV kernels. In the latter, we also account for explicit zeros read into the multiprocessors. We notice that the ELL kernel achieves memory access rates beyond 500 GB/s, which is about 70% of the theoretical peak [13]. COO attains around 450 GB/s; and CSR_smart and CSRI deliver around 300 GB/s. However, the memory bandwidth is not the relevant factor in terms of runtime performance and, although the ELL kernel was the performance winner for selected problems in Figure 4.3, it only achieves about 40 GFLOPs in this experiment. Higher throughput is achieved by CSR_smart (45 GFLOPs), CSRI (50 GFLOPs), and COO (55 GFLOPs). We mention that it is possible to improve the ELL kernel by using the sliced ELL format instead [2]. There, the overhead introduced from zero-padding is reduced by enforcing the same nonzero count only for those rows located in the same block. However, we refrain from this optimization step as we expect the benefit to be moderate: the height of the distinct row-blocks should at least match the warp size (32), which is relatively large compared to the small matrices we focus on.

Finally, we consider batches containing all of the matrices in the test benchmark, arranged in random order. Figure 4.7 (right) shows that the ELL kernel sustains a memory access rate around 500 GB/s. At the same time, the performance drops from 40 to 30 GFLOPs, which is likely due to the large number of small and unbalanced test matrices in the batch. Conversely, the performance of the other formats is not affected, and CSR_smart, CSRI and COO exceed 45, 50 and 55 GFLOPs, respectively.

We conclude that across all sparsity formats, the COO kernel achieves the best performance for heteroge-

Figure 4.6: Performance (left) and sustained memory bandwidth (right) of the flexible batched SpMV routines applied to a heterogeneous batch consisting of a random compilation of the matrices analyzed in Figure 4.3.

Figure 4.7: Performance (left) and sustained memory bandwidth (right) of the flexible batched SpMV routines applied to a heterogeneous batch consisting of a random compilation of all the matrices in the test benchmark.

neous batches. If the batch consists of balanced matrices only, the ELL kernel becomes the preferred choice, achieving up to 80 GFLOPs. In a one-touch-only scenario where all matrices are stored in CSR format, the CSR_smart kernel is the best option. If a preprocessing step is justified by a high number of kernel invocations, the CSRI kernel is much faster for batches consisting of unbalanced problems.

4.5 Summary and Outlook

We have developed and implemented a set of flexible batched SpMV kernels that accommodate the CSR, COO and ELL sparse matrix storage formats. The routines can efficiently process matrix batches where each problem is different in terms of size, nonzero count and nonzero pattern. Although the performance of the distinct kernels is very problem-dependent, our experimental results on an NVIDIA P100 GPU, using batches comprising very different matrices, revealed that the developed kernels based on COO and CSR are able to sustain a performance of about 50 GFLOPs. This corresponds to a $25\times$ speed-up compared to the use of a sequence of invocations to standard implementations of SpMV.

In the future we plan to further optimize the formats by determining the resources allocated to the distinct problems based on the matrix characteristics. Furthermore, we want to extend the performance assessment to also account for the energy usage, and compare resource efficiency with other manycore architectures that feature a more sophisticated cache hierarchy.

Acknowledgements

This work was partly funded by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC-0010042. H. Anzt was supported by the “Impuls und Vernetzungsfond” of the Helmholtz Association under grant VH-NG-1241. G. Flegar and E. S. Quintana-Ortí were supported by projects TIN2014-53495-R of the Spanish *Ministerio de Economía y Competitividad* and the EU H2020 project 732631 OPRECOMP.

The authors want to acknowledge the access to the PizDaint supercomputer at the Swiss National Supercomputing Centre granted under the project #d65.

Bibliography

- [1] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. Performance, design, and autotuning of batched GEMM for GPUs. In *In Proceedings of the 31st International Conference on High Performance Computing*, ISC 2016, pages 21–38. Springer, 2016.
- [2] H. Anzt, S. Tomov, and J. Dongarra. Implementing a sparse matrix-vector product for the SELL-C/SELL-C- σ formats on NVIDIA GPUs. Technical report, The University of Tennessee at Knoxville, 2014.
- [3] H. Anzt, E. Chow, and J. Dongarra. On block-asynchronous execution on GPUs. Technical report, LAPACK Working Note, 2016.
- [4] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler. Preconditioned Krylov solvers on GPUs. *Parallel Computing*, 68:32–44, 2017.
- [5] R. Barrett, M. Berry, T. F. 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*. SIAM, 2nd edition, 1994.
- [6] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA Corporation, 2008.
- [7] CUDA Toolkit v8.0. <https://docs.nvidia.com/cuda/>, 2017.
- [8] CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. <http://cusplibrary.github.io/>, 2014.
- [9] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.
- [10] J. Dongarra, I. S. Duff, M. Gates, A. Haidar, S. Hammerling, J. Higham, J. Hogg, P. Valero-Lara, D. Relton, S. Tomov, and M. Zounon. A proposed API for batched basic linear algebra subprograms. Technical report, The University of Manchester, 2016.
- [11] G. Flegar and E. S. Quintana-Ortí. Balanced CSR sparse matrix-vector product on graphics processors. In *Proceedings of the 23rd International European Conference on Parallel and Distributed Computing*, Euro-Par 2017, pages 697–709. Springer, 2017.
- [12] 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. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014.
- [13] NVIDIA Tesla P100. WP-08019-001_v01.1, 2016. Whitepaper.

Part III

Preconditioning

Block-Jacobi Preconditioning Using Explicit Inversion

Published as: H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. Variable-size batched Gauss-Jordan elimination for block-Jacobi preconditioning on graphics processors. *Parallel Computing*, 81:131–146, 2019

In this work, we address the efficient realization of block-Jacobi preconditioning on graphics processing units (GPUs). This task requires the solution of a collection of small and independent linear systems. To fully realize this implementation, we develop a variable-size batched matrix inversion kernel that uses Gauss-Jordan elimination (GJE) along with a variable-size batched matrix–vector multiplication kernel that transforms the linear systems’ right-hand sides into the solution vectors. Our kernels make heavy use of the increased register count and the warp-local communication associated with newer GPU architectures. Moreover, in the matrix inversion, we employ an implicit pivoting strategy that migrates the workload (i.e., operations) to the place where the data resides instead of moving the data to the executing cores. We complement the matrix inversion with extraction and insertion strategies that allow the block-Jacobi preconditioner to be set up rapidly. The experiments on NVIDIA’s K40 and P100 architectures reveal that our variable-size batched matrix inversion routine outperforms the CUDA basic linear algebra subroutine (cuBLAS) library functions that provide the same (or even less) functionality. We also show that the preconditioner setup and preconditioner application cost can be somewhat offset by the faster convergence of the iterative solver.

5.1 Introduction

Solving large, sparse-linear systems of equations is a prevailing problem in scientific and engineering applications that involve the discretization of partial differential equations (PDEs). A standard approach to tackle these problems combines a Krylov method with a preconditioner that accelerates the iterative solution process [14]. In the context of high-performance computing, the efficiency of the preconditioner depends on the parallel scalability of both the preconditioner generation (prior to the iterative solve) and the preconditioner application (at each step of the iterative solve).

Using preconditioners based on Jacobi (diagonal scaling) and block-Jacobi typically renders moderate improvements to the convergence of the iterative solver [14]. These acceleration techniques are nevertheless reasonably attractive as block-diagonal scaling introduces very small computational overhead to the solver iteration. Furthermore, the application of a Jacobi-type preconditioner is inherently parallel and, therefore, highly appealing for massively parallel architectures.

In [5] we proposed a batched routine for the generation of a block-Jacobi preconditioner using the explicit inversion of the diagonal blocks. Precisely, we designed a variable-size batched routine for matrix inversion on graphics processing units (GPUs) based on Gauss-Jordan elimination (GJE) [10]. Furthermore, we introduced an implicit pivoting strategy in the GJE procedure that replaces row-swapping with a migration of the workload (i.e., operations) to the thread that owns the necessary data, which allowed us to realize the complete inversion process in thread-local registers. For the block-Jacobi preconditioner generation, the inversion process needs to be combined with routines that extract the diagonal blocks from the sparse data structure that stores the coefficient matrix. This extraction step can be costly, particularly for matrices with an unbalanced nonzero

distribution. In response, we developed an extraction routine that balances coalescent data access, workload imbalance, and use of shared memory.

In this paper, we extend our previous work in [5] with new contributions, listed below.

- We propose a modified version of our variable-size batched GJE (BGJE) inversion routine for GPUs that can invert several blocks per warp. This avoids idle CUDA cores and operations on dummy values when processing a matrix batch where each matrix is less than or equal to 16×16 .
- We introduce a new variant of the extraction procedure that requires a much smaller amount of shared memory. This strategy transposes the diagonal blocks at the time they are extracted from the sparse coefficient matrix, inverts the transposed diagonal block, and ultimately writes the inverse of the transpose in transposed mode. The result provides a functionality equivalent to the original block inversion but reduces the amount of shared memory in the block size, utilized during the inversion procedure, from quadratic to linear only.
- We replace the general sparse matrix–vector multiplication kernel in the preconditioner application with a specialized variant that exploits the block-diagonal structure of the preconditioner matrix. This accelerates the application of the block-Jacobi preconditioner in the iterative solution process.

Our results revealed that these modifications can render significant performance improvements, particularly when targeting batches consisting of small blocks like those appearing in block-Jacobi preconditioning for problems arising from finite element method (FEM) discretizations.

The rest of the paper is structured as follows. In Section 5.2 we offer a short review of Jacobi-type iterative solvers and batched routines for linear algebra. In Section 5.3 we further elaborate on the batched Gauss-Jordan elimination (BGJE) procedure presented in [5], and we describe the batched kernels and highlight the major improvements in the block-Jacobi generation step, extraction step, and preconditioner application. In Section 5.4 we report on our extensive evaluation of the new BGJE routine on NVIDIA’s K40 and P100 GPUs. Particularly, we focus on the performance acceleration produced by the modifications of the original BGJE. Finally, in Section 5.5 we summarize our contributions and the insights gained from the experimental evaluation.

5.2 Background and Related Work

5.2.1 Block-Jacobi preconditioning

Consider the linear system $Ax = b$, with the coefficient matrix $A \in \mathbb{R}^{n \times n}$, the right-hand side vector $b \in \mathbb{R}^n$, and the sought-after solution $x \in \mathbb{R}^n$. The block-Jacobi method partitions the entries of the coefficient matrix as $A = L + D + U$, where $D = (D_1, D_2, \dots, D_N)$ contains a collection of blocks (of variable-size) located on the diagonal of A , while L and U comprise the entries of A below and above those in D , respectively. For a starting solution guess $x^{(0)}$, the iterative Jacobi solver can then be formulated as:

$$\begin{aligned} x^{(k)} &:= D^{-1} (b - (A - D)x^{(k-1)}) \\ &= D^{-1} b + Mx^{(k-1)}, \quad k = 1, 2, \dots, \end{aligned} \quad (5.1)$$

where the convergence is ensured if the spectral radius of the iteration matrix $M = I - D^{-1}A$ is smaller than one [14]. This occurs, for instance, in diagonally-dominant systems [14].

In case it is well-defined, the (block-)Jacobi matrix can be used as preconditioner, transforming the original system $Ax = b$ into either the left-preconditioned system

$$D^{-1}Ax = c \quad (= D^{-1}b), \quad (5.2)$$

or the right-preconditioned system

$$AD^{-1}y = b, \quad (5.3)$$

with $x = D^{-1}y$. Hereafter, we will consider the left-preconditioned case.

```

1 % Input : m x m nonsingular matrix block Di.
2 % Output : Matrix block Di overwritten by its inverse
3 p = [1:m];
4 for k = 1 : m
5     % explicit pivoting
6     [abs_ipiv, ipiv] = max(abs(Di(k:m,k)));
7     ipiv = ipiv+k-1;
8     [Di(k,:), Di(ipiv,:)] = swap(Di(ipiv,:), Di(k,:));
9     [p(k), p(ipiv)] = swap(p(ipiv), p(k));
10
11     % Jordan transformation
12     d = Di(k,k);
13     Di(:,k) = -[Di(1:k-1,k); 0; Di(k+1:m,k)] / d; % SCAL
14     Di = Di + Di(:,k) * Di(k,:); % GER
15     Di(k,:) = [Di(k,1:k-1), 1, Di(k,k+1:m)] / d; % SCAL
16 end
17 % Undo permutations
18 Di(:,p) = Di;

```

Figure 5.1: Simplified loop-body of the basic GJE implementation in Matlab notation using standard pivoting.

When integrated into a Krylov subspace-based solver, the application of a block-Jacobi preconditioner in (5.2) requires the solution of the block-diagonal linear system (i.e., a linear system for each block D_i). Alternatively, assuming the block-inverse matrix $\hat{D} = D^{-1}$ is available, the block-diagonal scaling in (5.2) can be realized in terms of a matrix-vector multiplication with the inverse blocks $\hat{D}_i = D_i^{-1}, i = 1, 2, \dots, N$. In general, pre-computing explicitly the block-inverse $\hat{D}$ during the preconditioner setup allows for faster preconditioner application in the iterative solver. However, when dealing with large blocks and sparse data structures, the inversion of matrix D can become a bottleneck. On parallel architectures, it is possible to exploit the pairwise independence of the diagonal blocks in D by generating their individual inverses in parallel.

5.2.2 GJE for matrix inversion

GJE has been proposed in the last years as an efficient method for matrix inversion on clusters of multicore processors and many-core hardware accelerators [6, 13]. In addition, in [5] we demonstrate the benefits of leveraging GJE for block-Jacobi preconditioning on GPUs. When combined with partial pivoting, GJE is as stable as matrix inversion via the LU factorization. At the same time, GJE avoids the workload imbalance that occurs in the LU-based approach due to computations with triangular factors.

The basic algorithm for matrix inversion via GJE consists of a loop that comprises a pair of vector scalings (SCAL) and a rank-1 update (GER); see Figure 5.1. The unblocked version of the GJE algorithm in this figure, based on Level-2 BLAS operations, generally yields a poor exploitation of the memory hierarchy on current processors. However, this formulation can deliver good performance when computing the inverses of small matrices, like those that usually appear in the context of block-Jacobi preconditioning. Finally, the Level-2 BLAS version of GJE allows the integration of an implicit pivoting strategy, which dramatically reduces explicit data movements.

5.2.3 GJE with implicit pivoting

To ensure numerical stability, GJE needs to include a pivoting strategy. On parallel architectures, the row swaps required in the standard partial pivoting technique (Figure 5.1, line 8) can be costly. This is particularly the case if the data is distributed row-wise among the processor cores. In this scenario, the two cores holding rows k and $ipiv$ need to exchange their data, while the other cores remain idle. Although distributing the matrix column-wise resolves this problem, the load imbalance is then just shifted to the pivot selection (line 6). As a response, in [5] we presented an implicit pivoting procedure which avoids explicitly swapping data. Instead, it accumulates all swaps, and combines them when completing the GJE algorithm. The paradigm underlying implicit pivoting is to move the workload to the thread owning the data, instead of keeping the workload fixed to the thread index and reshuffling the data.

```

1 % Input : m x m nonsingular matrix block Di.
2 % Output : Matrix block Di overwritten by its inverse
3 p = zeros(1, m);
4 for k = 1 : m
5     % implicit spivoting
6     abs_elems = abs(Di(:, k));
7     abs_elems(p > 0) = -1; % exclude already pivoted rows
8     [abs_ipiv, ipiv] = max(abs_elems);
9     p(ipiv) = k;
10
11     % Jordan transformation
12     d = Di(ipiv, k);
13     Di(:, k) = -[Di(1:ipiv-1, k); 0; Di(ipiv+1:m, k)] / d; % SCAL
14     Di = Di + Di(:, k) * Di(ipiv, :); % GER
15     Di(ipiv, :) = [Di(ipiv, 1:k-1), 1, Di(ipiv, k+1:m)] / d; % SCAL
16 end
17 % Undo permutations
18 Di(p, :) = Di(:, p);

```

Figure 5.2: Simplified loop-body of the basic GJE implementation in Matlab notation using implicit pivoting.

In standard GJE with explicit pivoting, the data required for operations performed on each row at iteration k (lines 12–15) is located only in that particular row and the current pivot row $ipiv$ (which was swapped with row k at the beginning of the iteration). The operation applied on the distinct rows only depends on whether or not a certain row is the current pivot row. Concretely, if a row is the current pivot (i.e., it lies on position k) the operation involves diagonal scaling; otherwise, it requires the scaling of element k followed by an AXPY of the remaining elements. Hence, the actual order of the rows is not important during the application of the Gauss-Jordan transformations, and the swaps can be postponed until the algorithm is completed. This idea is illustrated in Figure 5.2.

The selection of the pivot entry has to be modified when pivoting implicitly. In explicit pivoting, at iteration k , all previous pivots are located above the k -th entry of the diagonal, and the potential pivot rows for the current iteration lie in positions $k:m$. When using implicit pivoting, none of the rows have been swapped, so we need to keep track of the previously chosen pivots. At step k , the next pivot is chosen among the set of rows that were not yet selected as pivots. In Figure 5.2, the potential pivots are the entries in rows i with “ $p(i) = 0$ ” in lines 6–9.

Since implicit pivoting does not change the execution order of the operations applied or the numerical values, this variant of pivoting preserves the numerical properties of the algorithm.

5.2.4 Batched GPU routines

The qualifier “batched” identifies a procedure that applies the same operation to a large collection of data entities. In general, the subproblems (i.e., the data entities) are all small and independent, asking for a parallel formulation that simultaneously performs the operation on several/all subproblems in order to yield a more efficient exploitation of the computational resources. Batched routines are especially attractive in order to reduce the overall kernel launch overhead on GPUs, as they replace a sequence of kernel calls with a single kernel invocation. In addition, if the data for the subproblems is conveniently stored in the GPU memory, a batched routine can orchestrate a more efficient (coalesced) memory access.

In recent years, the development of batched routines for linear algebra operations has received considerable interest because of their application in machine learning, astrophysics, quantum chemistry, hydrodynamics, and hyperspectral image processing, among others. Examples of batched kernels for the dense BLAS appear in [9, 11], and there exists a strong community effort on designing a interface standard for these routines [8]. Aside from block-Jacobi, the adoption of batched routines for efficient preconditioner generation has also been recently studied in the context of using approximate triangular solves for incomplete factorization preconditioning [2, 5].

Figure 5.3: Generation of the block-Jacobi preconditioner: 1) data extraction; 2) variable-size BGJE; 3) data insertion. The block structure is indicated with orange circles, the original nonzero pattern with blue dots, and the block inverses with purple circles.

5.3 Design of CUDA Kernels

In [5] we designed a set of routines for the generation and application of block-Jacobi preconditioners via variable-size BGJE. In this section we review the key concepts in [5], and introduce several improvements to further accelerate both the generation and application of the preconditioner.

The generation of an inversion-based block-Jacobi preconditioner can be decomposed into three distinct steps: 1) extraction of the diagonal blocks; 2) inversion of these blocks; and 3) insertion of the inverse blocks into the preconditioner matrix. We visualize these steps for non-uniform block sizes in Figure 5.3. The three steps can be realized as three separate CUDA kernels, or in terms of a single kernel doing all steps in sequence. The experimental results in [5] suggest that, in general, merging all operations into a single kernel results in higher performance. A reason for this is the reduced memory transfer, as realizing the operations in a single kernel avoids the main memory accesses that are necessary to transfer data between separate kernels. In this paper we therefore focus on merged kernels for generating block-inverse matrices.

The question of how to identify a convenient block structure for a given coefficient matrix and an upper bound limiting the size of the diagonal blocks remains outside the focus of this paper. Here, for all experiments we use the supervariable blocking routine available in MAGMA-sparse [12].

5.3.1 Variable-size batched Gauss-Jordan elimination

The central operation in the generation of an inversion-based block-Jacobi preconditioner is the inversion of the diagonal blocks in D . These blocks are all square, of small dimension, and independent. In [5] we designed a variable-size BGJE routine that assigns one CUDA warp (a group of 32 threads) to invert each diagonal block. The kernel is launched on a grid with the number of warps covering the number of diagonal blocks. Within a warp, parallelism is realized by each thread handling one row of the diagonal block. This limits the scope of the kernel to matrix batches where no matrix is of dimension larger than 32. As blocks of larger dimension are rarely encountered in the context of block-Jacobi preconditioning, the variable-size batched GJE kernel perfectly fits this application scope [5].

Handling the inversion with a single warp allows us to use two recent features of NVIDIA’s GPU architectures: increased register count and warp shuffle instructions. In some detail, the data required by each thread (up to 32 data elements belonging to the matrix row) is first read into registers; the inversion is then computed using this data and communication occurs via the warp shuffle instruction (avoiding main memory access during the inversion process); and finally, the computed inverse is written back to main memory. In general, even though the diagonal blocks are sparse, their inverses are dense. We therefore handle and store the diagonal blocks in dense format during the complete inversion process.

The pivoting process ensuring numerical stability requires to identify the pivot element, see line 8 in Figure 5.2. Since the matrix is distributed row-wise among the threads, this requires a parallel reduction. We realize this step via warp shuffles. The same type of shuffles is also used to distribute the contents of the current pivot row $D_i(\text{ipiv}, k)$ required for the operations in lines 12–15.

Multiple problems per warp. The variable-size BGJE presented in [5] assigns one warp to each diagonal block. In that work, diagonal blocks of size $k < 32$ were padded with dummy values to that dimension and the threads only execute the first k iterations of the GJE algorithm. Obviously, for small blocks, this wastes a significant part of the computational resources, as most of the threads then operate on dummy data. In this work, we improve the algorithm by allowing one warp to handle multiple small problems simultaneously. Concretely, let k_m denote the size of the largest block in the matrix batch and p_m stand for the smallest power of 2 such that $p_m \geq k_m$; then, in our new approach, each warp processes $32/p_m$ blocks. Proceeding in this manner, each group of p_m threads – we call it *sub-warp* – is assigned to one problem. The first k_m threads in the sub-warp compute the inverse of a block of size $k \leq k_m$ by padding it with dummy values to size k_m , and computing only the first k steps of the inversion procedure. The rest of the threads in the sub-warp remain idle. The reason for choosing p_m as the sub-warp size is that the CUDA ecosystem supports warp shuffles for these sizes. Using k_m instead of p_m would require additional operations to calculate the thread index. Finally, we do not consider “packing” blocks of different sizes into one warp (e.g., one warp could process blocks of sizes 15 and 17), as this would require a preprocessing step in order to determine which warp can process which set of blocks. Furthermore, it would also result in thread divergence between the two parts of the warp.

5.3.2 Data extraction from the sparse coefficient matrix

As BGJE expects a collection of small dense blocks as input, these blocks need to be extracted from the sparse coefficient matrix stored in CSR format [14]. We next review the two extraction strategies we implemented and compared in [5].

The first approach, named *cached extraction*, is a straight-forward method where each thread traverses a single matrix row, (specifically, the row whose values will be required by this thread during inversion process,) and extracts the elements that lie on the corresponding diagonal block. Since the CSR format is designed to favor accessing sparse matrix by rows, (i.e., it keeps the matrix entries in row-major order,) this will most likely result in non-coalescent memory access. Furthermore, an unbalanced nonzero distribution in the coefficient matrix inevitably incurs load imbalance, as threads operating on short rows will remain idle while the remaining threads extract the data from their rows. Both effects impair the performance of the extraction step.

As a response to these issues, in [5] we proposed an alternative *shared extraction* method. The key idea is to eliminate non-coalescent memory access and (potential) load imbalance at the cost of using shared memory. Precisely, all threads of the warp collaborate on the extraction of the block by accessing each row containing part of the block in a coalesced mode (see Figure 5.4). The diagonal block is then converted to dense format and stored into shared memory by writing the extracted values into the appropriate locations in shared memory, see right-hand side in Figure 5.4. Once the extraction of a block is completed, each thread reads the values of the row assigned to it from shared memory. This strategy makes all memory accesses coalescent and alleviates load imbalance. The shared memory usage, however, can constrain the number of warps active per multiprocessor. On “older” GPU architectures we observed that the shared extraction strategy can result in lower performance due to this issue.

Reduced usage of shared memory. We improve the situation by radically reducing the amount of shared memory employed in the shared extraction step. This is possible because the inverse of a matrix A can be computed by first obtaining the inverse of A^T and then transposing the result (i.e., $((A^T)^{-1})^T = A^{-1}$). Extracting the transpose of the diagonal block is much easier as the i -th elements of all columns are available as soon as

Figure 5.4: Illustration of the memory requests for the cached extraction and shared extraction (left and right, respectively). We assume warps of 4 threads and memory transactions of 4 values. We only show the accesses to the vector storing the col-indices of the CSR matrix structure; the access to the actual values induces far less overhead, as these memory locations are accessed only if a location belonging to a diagonal block is found. In that case, the access pattern is equivalent to the one used for col-indices.

the i -th row is extracted to shared memory. This means that all threads can already read the i -th row-value of the transposed block into registers before proceeding with the extraction of the next row. Thus, the extraction of the transpose block from the sparse matrix structure can be “interleaved” with the retrieval of the values from shared memory into the registers. As a result, the same shared memory locations used to store row k of the diagonal block can be re-used in the following step of the extraction. This reduces the total amount of shared memory required to that necessary to keep a single row of the diagonal block.

In case multiple blocks are assigned to each warp, a straight-forward extension of the strategy is to let each sub-warp extract the block assigned to it. This would, however, result in non-coalesced memory access. Coalesced memory access can be preserved by extracting all blocks handled by the warp in sequence, using all threads of the warp and enough shared memory to store one row of the largest matrix in the batch.

After the inversion of a diagonal block is completed, the result is written back to main memory. Realizing the afore-described extraction step in reverse order, we store the inverse of the transposed block in transposed mode – which is the inverse of the original block.

5.3.3 Preconditioner application

Once the block-inverse is generated, the block-Jacobi preconditioner can be applied in terms of a sparse matrix-vector product (SPMV).

Structure-aware SPMV. We improve the performance of the block-Jacobi preconditioner by replacing the generic sparse matrix-vector product with a specialized kernel that exploits the block structure of the preconditioner.

tioner matrix. In detail, we use a variable-size batched dense matrix-vector multiplication (GEMV) to multiply the distinct block inverses D_i^{-1} with the appropriate vector segments. As in the preconditioner generation, the blocks are distributed among the (sub-)warps, with each (sub-)warp handling the multiplication for one vector segment. In contrast to the elements of the matrix, which are all used for only a single multiplication, the elements in the vector segment are reused in the multiplication with the distinct rows of the block. Hence, it is beneficial to read the elements of the vector segment into the registers of the distinct threads of the (sub-)warp (one element per thread) at the beginning of the routine. The performance of GEMV is constrained by the memory bandwidth. It is therefore essential to ensure coalesced memory accesses by forcing each (sub-)warp to read the diagonal blocks row-wise (the block-diagonal matrix is stored in the row-major based CSR matrix format). For each row of the block, the threads of the (sub-)warp then use warp shuffles to compute the dot product of the matrix row and the vector entries they keep in registers. Finally, the result is written to the appropriate position in the output vector.

5.4 Experimental Evaluation

In this section, we (1) benchmark gigaFLOPS enhanced version of the variable-size batched matrix inversion routine based on GJE; (2) analyze the performance of the complete block-Jacobi preconditioner (generation and application); and (3) assess the efficiency of block-Jacobi preconditioning in an iterative solver setting.

We begin by comparing the new BGJE kernel against the version presented in [5] and against two batched inversion routines available in NVIDIA’s cuBLAS library: `getriBatched` and `matinvBatched`. We note that both cuBLAS routines can only operate on batches of problems where all matrices are of the same size.

Next, to evaluate the the performance benefits for the block-Jacobi preconditioner generation stage, in isolation, we combine our variable-size BGJE routines with the improved extraction and insertion procedures, and we test the block-inverse generation for different sparsity structures and block sizes. For this purpose, we consider a set of test matrices from the SuiteSparse Matrix Collection¹ (formerly known as the University of Florida Sparse Matrix Collection). In addition to the preconditioner generation, we also compare the specialized block-Jacobi application kernel based on variable-size batched GEMV with the generic SPMV routine from MAGMA-sparse [12].

Finally, to analyze the practical effects of the block-Jacobi preconditioning on an iterative solver, we integrate the block-Jacobi preconditioner into an Induced Dimension Reduction Krylov solver with shadow space dimension 4 (IDR(4)) and demonstrate the time-to-solution improvements obtained by replacing a scalar-Jacobi preconditioner with a block-Jacobi variant.

5.4.1 Hardware and software framework

For the experiments, we use the two most recent NVIDIA GPU architectures, which have full support for double-precision computations: the Kepler K40 (Compute Capability 3.5) and the Pascal P100 (Compute Capability 6.0). We do not consider the older Fermi and Maxwell architectures, as the former lacks support for warp shuffle instructions, and the latter does not implement full double-precision support. Because the batched matrix inversion routines, the block-Jacobi generation kernel, and the iterative solvers proceed exclusively on the GPU, details about the node’s broader hardware specifications are irrelevant in the following experiments.

Our kernels are implemented using CUDA 8.0 and are designed to be integrated into the MAGMA-sparse library [12]. MAGMA-sparse also provides the testing environment, the block-pattern generation, and the sparse solvers used in our experiments. All computations use double-precision arithmetic—the standard in linear algebra.

5.4.2 Batched matrix inversion

This section analyzes the performance of four batched routines for matrix inversion on GPUs; BGJE, BGJE-MPW, `getriBatched`, and `matinvBatched`.

1. BGJE is the variable-size BGJE inversion kernel from [5].

¹Visit <http://www.cise.ufl.edu/research/sparse/matrices/>.

Figure 5.5: Performance comparison of batched matrix inversion routines for various batch sizes. Top row shows matrices of size 32×32 and the bottom row shows matrices of size 16×16 .

2. BGJE-MPW is the enhanced kernel that incorporates the BGJE improvements described in this paper.
3. getriBatched renders the batched matrix inversion using two functions from NVIDIA’s cuBLAS library: (1) getrfBatched computes the LU factorization of the matrix batch, then (2) getriBatched obtains the inverses using the results of the previous routine. All matrices in the batch are required to be of the same size.
4. matinvBatched is NVIDIA’s routine that merges the two calls of the getriBatched routine into a single kernel. Its functionality is limited to operating on batches of equal-size matrices with an upper bound of 32×32 .

We note that the scope of the distinct batched inversion routines is slightly different: BGJE, BGJE-MPW, and matinvBatched only support matrices of size up to 32×32 ; and neither matinvBatched nor getriBatched support batches containing matrices of different sizes. Therefore, we limit the performance comparison to batches composed of equal-size matrices of up to 32×32 . While this upper bound is usually not a problem in the context of block-Jacobi preconditioning, handling batches that contain variable-size matrices is essential to accommodating the inherent block structure of FEM discretizations. Consequently, the cuBLAS routines will not be considered in the complete preconditioner generation and application experiments.

Figure 5.5 compares the performance, in terms of gigaFLOPS (billions of arithmetic floating-point operations per second), for two fixed matrix sizes (32×32 and 16×16) while increasing the matrix count (batch size). In a case where the matrix order is 32, both BGJE and BGJE-MPW deliver the same performance because, in this scenario, BGJE-MPW also schedules a single problem per warp. For this matrix size, the performance of both variable-size BGJE routines exceeds 600 gigaFLOPS (13% of the theoretical peak) on P100 and around 125 gigaFLOPS (9% of peak) on K40. These rates correspond to a $6\times$ speedup over the batched inversion using getriBatched, and at least a $12\times$ speedup over matinvBatched.

The older K40 architecture has a significantly lower register-per-core ratio compared to the P100. Because our BGJE and BGJE-MPW routines make heavy use of registers, a reduced register count limits the number of

Figure 5.6: Performance comparison of batched matrix inversion routines for various matrix sizes.

threads/warps that can be active on a multiprocessor, which explains the large performance gap between the K40 and P100 GPUs.

The two graphs on the left side of Figure 5.5 clearly show that the registers are indeed a performance bottleneck on the K40. For batched problems consisting of 16×16 matrices, each thread only utilizes 16 registers (instead of 32 registers for 32×32 matrices), allowing more active threads—and therefore more active warps—per multiprocessor. As a result, the BGJE-MPW kernel delivers about 160 gigaFLOPS for the smaller matrix sizes but only around 125 gigaFLOPS for the larger matrices. In comparison, the BGJE kernel, which can only handle a single problem per warp, achieves a scant 40 gigaFLOPS for the small case. Moreover, both cuBLAS batched inversion routines, `getriBatched` and `matinvBatched`, deliver a meager 8 gigaFLOPS for this problem.

Again, note that the BGJE kernel pads the matrices with dummy elements to size 32×32 and inverts one system per warp. On the P100, this delivers less than 150 gigaFLOPS for a batch composed of matrices of size 16×16 . In contrast, the performance of the BGJE-MPW routine exceeds 550 gigaFLOPS in similar conditions. Thus, although the performance of BGJE-MPW is lower for a 16×16 matrix than for a 32×32 matrix (which was expected because the data-movement-to-floating-point-operation ratio grows with the matrix size), BGJE-MPW is about one order of magnitude faster than the matrix inversion functions provided in NVIDIA’s cuBLAS.

A detailed analysis for different matrix sizes is given in Figure 5.6. In this experiment we fixed the batch size to 500,000 matrix problems and varied the dimension of the matrices in the batch from 1×1 to 32×32 . For both architectures, BGJE exhibits a superlinear performance drop as the matrix size is reduced. This is because, for a batch with matrices of size k , each warp performs $2k^3$ useful operations, while the total volume of operations (including those on dummy data used for padding) is $2k \times 32^2$. In contrast, BGJE-MPW avoids most dummy operations and experiences only a linear performance loss—owing to inactive threads—between consecutive powers of two. Peaks for 16×16 matrices and 8×8 matrices clearly mark the thresholds where multiple small problems can be handled by a single warp without introducing any computational overhead. The performance lines for BGJE-MPW are more erratic than those observed for the other routines. The reason is that BGJE-MPW is implemented using C++ templates to generate a specialized version of the kernel for each matrix size. While this approach succeeds in minimizing the register count and the number of operations performed by the size-specific kernels, the kernel-specific resource requirements impact the number of warps that are active per multiprocessor and, ultimately, the kernel-specific performance.

5.4.3 Block-Jacobi generation

We now turn our attention to the complete block-inversion procedure that produces the block-Jacobi preconditioner. This includes the extraction of the diagonal blocks from a sparse data structure, followed by the explicit inversion, and then the insertion of the inverse blocks into the preconditioner matrix. As previously mentioned, the routines are “merged” into a single CUDA kernel that performs all three steps: (1) extraction from the sparse matrix structure, (2) inversion, and (3) insertion into the sparse preconditioner matrix. In this subsection, we compare three strategies for the generation of the block-Jacobi preconditioner, with the first strategy

Figure 5.7: Performance improvement from reducing the shared memory size in the block-Jacobi generation using shared extraction/insertion.

Figure 5.8: Sparsity plots of test matrices used to evaluate the diagonal block extraction.

corresponding to an implementation that was already proposed in [5], and the last two strategies realizing the improvements described in section 5.3:

1. CACHED: cached extraction/insertion with BGJE;
2. SHARED: shared extraction/insertion with BGJE; and
3. SHARED-MPW: shared extraction/insertion with BGJE-MPW.

Both SHARED and SHARED-MPW use the reduced memory shared extraction described in subsection 5.3.2. Figure 5.7 reveals that reducing the shared memory in the SHARED strategy can make the block-Jacobi generation up to four times faster. The problem-specific benefits depend on the the upper bound for the block size, the pattern of the system matrix determining the actual size of the distinct diagonal blocks, and the hardware characteristics determining how many thread blocks a multiprocessor can schedule in parallel.

Because the performance of the extraction strategies depends on the structure of the problem matrix, we consider four nonzero distributions that are characteristic in sparse linear algebra. In Figure 5.8, the arrow structure presents all nonzero entries on the (main) diagonal plus the last row/column of the matrix. In contrast, in the tridiagonal structure all nonzeros lie on the diagonal plus the diagonal immediately above/below it. These two structures are interesting, because they share the same nonzero count but exhibit different nonzero distributions. The other two examples correspond to a random block-diagonal matrix structure with nonzeros only in the diagonal blocks. The Laplace structure arises from the five-point stencil discretization of the Laplace equation.

In Figure 5.9, we report the total execution time of the three block-Jacobi generation strategies applied to the four matrix structures. In this experiment, we fix the size of the matrix to 1,000,000 and increase the size of the diagonal blocks from 1 to 32.

For the arrow sparsity structure, the SHARED strategy is much faster than its CACHED counterpart; see results in the first row of Figure 5.9. This result was expected because the arrow nonzero pattern contains a

Figure 5.9: Block-Jacobi generation time for increasing block sizes and nonzero distributions from top to bottom: arrow, tridiagonal, random block and Laplace.

Figure 5.10: Block-Jacobi generation time on NVIDIA P100 for a set of matrices taken from the SuiteSparse sparse matrix collection and varied block sizes: 4, 8, 12, 16, 24 and 32.

single dense row, which results in dramatic load imbalance if each row is traversed by a single thread, as is the case for CACHED. The SHARED alternative uses all threads of the warp to traverse this row, which alleviates the load imbalance and ensures coalescent access. For the other cases, the impact of non-coalescent memory access featured by CACHED is small as long as we consider small block sizes. This is because, for small blocks, only a few threads in each warp read data, which results in a reduced number of memory requests. Conversely, for large block sizes, the increase in memory requests impairs performance. Both strategies based on shared extraction eliminate load imbalance and non-coalescent memory access. Nonetheless, the reduced number of idle threads makes the SHARED-MPW version the overall winner.

We now assess the performance of the extraction routines for a set of test matrices from the SuiteSparse matrix collection. For brevity, we display the results for the P100 GPU only. The selected test matrices are listed along with some key properties in Table 5.1. In Figure 5.10, we report the runtime of the block-Jacobi preconditioner generation for different block sizes. In these tests, the block sizes only correspond to an upper bound, and the blocks are identified via supervariable blocking. Also, some blocks can be smaller to better reflect the block structure of the problem matrix [7]. We again identify SHARED-MPW as the overall winner.

[t]

Figure 5.11: Performance comparison of the block-Jacobi preconditioner application. SPMV is the generic sparse matrix-vector product routine from [5]. SA-SPMV is the specialized batched GEMV-based kernel developed as part of this work.

Figure 5.12: IDR(4) convergence and performance comparison for different block sizes used in the block-Jacobi preconditioner. The problem matrices are listed along with key characteristics in Table 5.1.

5.4.4 Block-Jacobi application

In an iterative solver setting, the efficiency of a preconditioner depends on the overhead of generating the preconditioner and, to even a larger extent, on the cost of applying it during the iterative solution process.

In Figure 5.11, we assess the performance of the preconditioner application using a generic SPMV kernel proposed in [5] versus our structure-aware SPMV (SA-SPMV) introduced in Section 5.3.3. On both architectures, SA-SPMV outperforms the initial SPMV kernel for the preconditioner application. For a block size of 32, this routine achieves about 32 gigaFLOPS on the K40 architecture, and around 80 gigaFLOPS on the P100 architecture. Local performance peaks can be identified for block sizes 8 and 16.

5.4.5 Convergence in the context of an iterative solver

Table 5.1 in the appendix details the convergence rate and execution time of an IDR(4) iterative solver [15] enhanced with either a scalar-Jacobi preconditioner or a block-Jacobi preconditioner for the selected cases from the SuiteSparse collection. The execution time includes both the preconditioner generation and the iterative solver execution. Detailed analysis reveals that in 88% of the tests, the preconditioner setup accounts for less than 1% of the total execution time. In all other cases, the block-inverse generation accounts for less than 5%. We combine the best kernels for the distinct preconditioner building blocks, i.e., SHARED-MPW, BGJE-MPW, and SA-SPMV. Other kernels are taken from the MAGMA-sparse open-source software package [12]. The IDR method [1] is among the most robust Krylov solvers [3]. The GPU-implementation of IDR available in MAGMA-sparse has been proven to achieve performance close to the hardware-imposed bounds [4]. For

brevity, we run these tests on the newer P100 architecture only. We start the iterative solution process with an initial guess, $x_0 = 0$, solve for a right-hand side composed of random values in the interval $[0, 1]$, and stop the iteration process once the relative residual norm has decreased by nine orders of magnitude. We allow for up to 50,000 iterations of the IDR(4) solver. In Figure 5.12, we summarize the results showing for how many problems a certain configuration was the best choice (i.e., provided the fastest time to solution), and for how many problems a certain configuration was “successful” (concretely, reduced the relative residual norm by nine orders of magnitude within the limit of 50,000 iterations).

The results reveal that the scalar version of Jacobi fails to sufficiently improve the convergence of IDR(4) for a significant fraction of the test matrices. For the test matrices where IDR(4) preconditioned with the scalar Jacobi converges, the faster convergence obtained from using a block-Jacobi preconditioner typically compensates for the higher costs of preconditioner setup and application. In Figure 5.13, we offer a head-to-head comparison of different block-size bounds for the block-Jacobi preconditioner used in IDR(4). The orange area in the plot at position “row Jacobi(x) vs. column Jacobi(y)” visualizes the number of matrices for which IDR(4) preconditioned with block-Jacobi of block size x converged, while it failed to converge with block size y . The opposite scenario, where block size y converged but block size x did not, is shown in green. Finally, the yellow area represents the number of matrices for which both methods converged—the area to the right of the center represents cases where block size y converges faster, while the area left of the center represents cases where block size x converges faster. The results suggest that adopting a larger block size usually leads to a more robust solver (i.e., convergence is achieved for a larger number of problems), and that a larger block size also improves the overall time-to-solution performance. However, in order to obtain the optimal performance for a specific problem, the block size should be tuned to the underlying block structure of the problem.

Overall, the results presented in this subsection offer strong evidence that the routines we developed provide an efficient approach to generating and applying a block-Jacobi preconditioner.

5.5 Concluding Remarks

In this paper, we presented an enhanced, variable-size batched matrix inversion routine for GPUs based on the GJE process. Our approach replaces explicit pivoting with a strategy that reassigns the workload instead of shuffling the data and relies heavily on CUDA’s latest warp-local communication features. As a result, our matrix inversion kernel is more flexible and significantly outperforms its counterparts in the cuBLAS library. In the framework of block-Jacobi preconditioning, we combined the batched matrix inversion procedure with efficient routines for extracting the diagonal blocks from the sparse data structures (where the problem matrix is stored) and inserting the inverse blocks back into the preconditioner. We also addressed the efficient preconditioner application by developing a structure-aware batched kernel for the sparse matrix-vector product that accommodates variable-size matrix operands. Finally, we demonstrated that block Jacobi can be significantly more efficient than a scalar Jacobi when preconditioning iterative solvers.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC-0010042. H. Anzt was supported by the “Impuls und Vernetzungsfond of the Helmholtz Association” under grant VH-NG-1241. G. Flegar and E. S. Quintana-Ortí were supported by project TIN2014-53495-R of the MINECO–FEDER; and project OPRECOMP (<http://oprecomp.eu> with the financial support of the Future and Emerging Technologies (FET) programme within the European Union’s Horizon 2020 research and innovation programme, under grant agreement No 732631. The authors would also like to acknowledge the Swiss National Computing Centre (CSCS) for granting computing resources in the Small Development Project entitled “Energy-Efficient preconditioning for iterative linear solvers” (#d65).

Figure 5.13: Detailed comparison of IDR(4) enhanced with block Jacobi using different block sizes.

Matrix	size	#nnz	ID	Jacobi		Block-Jacobi (8)		Block-Jacobi (12)		Block-Jacobi (16)		Block-Jacobi (24)		Block-Jacobi (32)	
				#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]
ABACUS_shell_ud	23,412	218,484	26	3703	5.09	2305	3.32	2829	4.09	3028	4.43	2858	4.27	2418	3.70
bcsstk17	10,974	428,650	6	1967	2.87	1174	1.78	901	1.38	792	1.23	735	1.20	879	1.40
bcsstk18	11,948	149,090	39	1491	1.90	933	1.37	653	0.98	591	0.92	440	0.65	532	0.85
bcsstk38	8,032	355,460	10	–	–	–	–	2459	4.21	4290	7.33	1878	3.26	2050	3.62
cbuckle	13,681	676,515	9	434	0.81	118	0.22	48	0.11	102	0.23	49	0.12	75	0.15
Chebyshev2	2,053	18,447	5	–	–	197	0.53	62	0.18	53	0.15	38	0.11	35	0.10
Chebyshev3	4,101	36,879	17	–	–	–	–	194	0.67	177	0.66	113	0.40	77	0.31
dc3	116,835	766,396	28	168	17.29	169	17.36	203	20.93	203	20.92	320	33.00	183	18.87
dw1024	2,048	10,114	30	–	–	169	0.23	148	0.27	163	0.30	92	0.18	65	0.13
dw2048	2,048	10,114	7	–	–	169	0.23	148	0.27	163	0.30	92	0.17	65	0.12
dw4096	8,192	41,746	18	–	–	–	–	–	–	6949	8.96	1732	2.33	1012	1.46
dw8192	8,192	41,746	36	–	–	–	–	–	–	6949	8.94	1732	2.35	1012	1.41
F1	343,791	26,837,113	27	3832	23.93	–	–	2460	16.34	2511	17.00	1895	13.23	2062	14.78
G3_circuit	1,585,478	7,660,826	31	2346	19.94	2069	19.52	2220	22.14	1935	20.09	2085	24.10	2198	26.83
gridgena	48,962	512,084	32	2265	3.51	1306	2.08	1766	2.84	1431	2.33	1028	1.70	1311	2.26
ibm_matrix_2	51,448	537,038	21	–	–	–	–	227	0.46	8992	17.45	254	0.55	2965	6.04
Kuu	7,102	340,200	3	162	0.29	103	0.19	95	0.18	84	0.16	94	0.18	84	0.17
LeGresley_2508	2,508	16,727	20	237	0.40	247	0.36	203	0.35	–	–	185	0.33	166	0.31
linverse	11,999	95,977	35	6185	7.80	–	–	–	–	6685	9.51	2175	3.07	933	1.44
matrix_9	103,430	1,205,518	2	1512	2.72	727	1.37	95	0.21	598	1.18	87	0.20	558	1.16
nasa2910	2,910	174,296	12	738	1.03	529	0.73	509	0.74	648	0.94	435	0.67	392	0.64
nd12k	36,000	14,220,946	19	–	–	6869	20.67	3183	9.67	2764	8.45	1543	4.81	1693	5.35
nd24k	72,000	28,715,634	11	–	–	4918	23.29	2906	13.85	2858	13.69	1916	9.35	1457	7.16
nd3k	9,000	3,279,690	33	–	–	–	–	4551	8.05	8270	14.55	2640	4.78	3196	5.87
nd6k	18,000	6,897,316	29	–	–	–	–	5142	11.14	9178	19.94	2589	5.76	2573	5.81
nemeth15	9,506	539,802	34	94	0.17	–	–	–	–	144	0.27	50	0.10	49	0.10
olm5000	5,000	19,996	15	–	–	1164	1.58	1049	1.44	256	0.41	545	0.80	178	0.33
Pres_Poisson	14,822	715,804	4	199	0.38	130	0.24	129	0.26	113	0.23	93	0.19	82	0.17
rail_79841	79,841	553,921	1	995	1.62	909	1.56	1013	1.78	880	1.52	862	1.58	810	1.52
s1rmt3m1	5,489	217,651	22	296	0.43	191	0.32	171	0.24	159	0.28	148	0.28	148	0.27
s2rmq4m1	5,489	263,351	37	708	0.97	231	0.39	223	0.36	262	0.43	214	0.36	198	0.36
s2rmt3m1	5,489	217,681	25	1016	1.35	327	0.50	209	0.35	218	0.36	178	0.32	220	0.40
s3rmq4m1	5,489	262,943	24	–	–	1387	1.95	599	0.89	1969	2.83	515	0.80	972	1.47
s3rmt3m1	5,489	217,669	16	–	–	–	–	693	0.99	2637	3.62	529	0.75	1142	1.71
s3rmt3m3	5,357	207,123	23	–	–	–	–	1995	2.74	2087	2.86	2229	3.14	784	1.18
saylr4	3,564	22,316	38	1907	2.46	387	0.59	246	0.40	281	0.38	163	0.30	170	0.32
ship_003	121,728	3,777,036	13	–	–	2058	4.89	1927	4.62	2849	6.97	1683	4.26	2160	5.75
sme3Dc	42,930	3,148,656	8	2680	5.66	4953	10.59	3101	6.74	3014	6.53	3566	7.92	4990	11.19
sts4098	4,098	72,356	14	135	0.26	113	0.23	78	0.12	94	0.19	76	0.16	64	0.11

Table 5.1: Iterations and execution time of IDR(4) enhanced with scalar Jacobi preconditioning or block-Jacobi preconditioning. The runtime combines the preconditioner setup time and the iterative solver execution time.

Bibliography

- [1] H. Anzt, E. Ponce, G. D. Peterson, and J. Dongarra. GPU-accelerated co-design of induced dimension reduction: Algorithmic fusion and kernel overlap. In *Proceedings of the 2nd International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC'15, pages 5:1–5:8. ACM, 2015.
- [2] H. Anzt, E. Chow, T. Huckle, and J. Dongarra. Batched generation of incomplete sparse approximate inverses on GPUs. In *Proceedings of the 7th Workshop on Scalable Algorithms for Large-scale Systems*, ScalA'16, pages 49–56. IEEE, 2016.
- [3] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler. Efficiency of general Krylov methods on GPUs — an experimental study. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops*, IPDPSW'16, pages 683–691. IEEE, 2016.
- [4] 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. *International Journal of High Performance Computing Applications*, 32(2):220–230, 2016.
- [5] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'17, pages 1–10. ACM, 2017.
- [6] P. Benner, P. Ezzatti, E. Quintana-Ortí, and A. Remón. Matrix inversion on CPU-GPU platforms with applications in control theory. *Concurrency and Computation: Practice and Experience*, 25(8):1170–1182, 2013.
- [7] E. Chow, H. Anzt, J. Scott, and J. Dongarra. Using Jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning. *Journal of Parallel and Distributed Computing*, 119:219–230, 2018.
- [8] J. Dongarra, I. S. Duff, M. Gates, A. Haidar, S. Hammerling, J. Higham, J. Hogg, P. Valero-Lara, D. Relton, S. Tomov, and M. Zounon. A proposed API for batched basic linear algebra subprograms. Technical report, The University of Manchester, 2016.
- [9] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *International Journal of High Performance Computing Applications*, 29(2): 193–208, 2015.
- [10] A. S. Householder. *The Theory of Matrices in Numerical Analysis*. Dover, 1st edition, 1964.
- [11] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra. Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 27(7): 2036–2048, 2016.
- [12] MAGMA 2.0.0. <http://icl.cs.utk.edu/magma/>, 2016.
- [13] E. S. Quintana-Ortí, G. Quintana-Ortí, X. Sun, and R. van de Geijn. A note on parallel matrix inversion. *SIAM Journal on Scientific Computing*, 22(5):1762–1771, 2001.
- [14] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.
- [15] P. Sonneveld and M. B. van Gijzen. IDR(s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. *SIAM Journal on Scientific Computing*, 31(2):1035–1062, 2009.

Block-Jacobi Preconditioning Based on Gauss-Huard Decomposition

Published as: H. Anzt, J. Dongarra, G. Flegar, E. S. Quintana-Ortí, and A. E. Tomás. Variable-size batched Gauss-Huard for block-Jacobi preconditioning. In *Procedia Computer Science*, volume 108, pages 1783–1792. Elsevier, 2017

In this work we present new kernels for the generation and application of block-Jacobi preconditioners that accelerate the iterative solution of sparse linear systems on graphics processing units (GPUs). Our approach departs from the conventional LU factorization and decomposes the diagonal blocks of the matrix using the Gauss-Huard method. When enhanced with column pivoting, this method is as stable as LU with partial/row pivoting. Due to extensive use of GPU registers and integration of implicit pivoting, our variable size batched Gauss-Huard implementation outperforms the batched version of LU factorization. In addition, the application kernel combines the conventional two-stage triangular solve procedure, consisting of a backward solve followed by a forward solve, into a single stage that performs both operations simultaneously.

6.1 Introduction

Iterative methods for the solution of sparse linear systems can strongly benefit from the integration of a preconditioner that is inexpensive to calculate and apply, and improves the convergence rate of the iterative scheme [14]. On a parallel system, the efficiency of the preconditioner also depends on how well these two building blocks, preconditioner calculation and application, can be formulated in terms of parallel algorithms.

Block-Jacobi preconditioners are more complex to handle than their (scalar) Jacobi counterparts, as they base the scaling on the inverse of the block diagonal. Nevertheless, the additional effort can be justified, as block-Jacobi preconditioners often provide faster convergence for problems that inherently carry a block structure.

The higher cost of block-Jacobi schemes comes from the extraction of the diagonal blocks from the coefficient matrix of the linear system, which is typically stored using a specialized data structure for sparse matrices, and the scaling with the inverse of this collection of small independent blocks. The latter can be realized by either explicit inversion in the preconditioner setup phase, or by generating LU factors that are then used in triangular solves during the preconditioner application (solve phase). Here, we note that parallelism in block-Jacobi preconditioners comes from the existence of multiple independent problems of small size.

In [4] we introduced a batched routine for GPUs that explicitly generates the block-inverse of a block-Jacobi preconditioner. Our implementation, based on Gauss-Jordan elimination (BGJE), 1) integrates an efficient scheme to extract the required matrix entries from the sparse data structures, 2) applies an implicit pivoting strategy during the inversion, and 3) computes the inverses using the GPU registers. As a result, our BGJE kernel clearly outperforms the batched LU-based factorization kernel in MAGMA [12].

In this paper we revisit the computation of block-Jacobi preconditioners on GPUs via variable size batched routines, making the following contributions:

- Our block-Jacobi preconditioner generation computes a triangular decomposition of the diagonal blocks, avoiding the computationally expensive and numerically dubious explicit generation of inverses.

- Instead of utilizing the conventional LU factorization for the decomposition, we rely on the Gauss-Huard (GH) algorithm [11]. The cost of this algorithm is analogous to that of the LU-based method, and much lower than the cost of an explicit inversion (such as GJE). Furthermore, when combined with *column* pivoting, GH offers a numerical stability that is comparable with that of the LU-based method with partial pivoting [9].
- In contrast to the batched LU factorization kernels in MAGMA, and the GPU parallel version of the GH algorithm in [6], we design our CUDA kernel to address the small problems of variable size arising in the computation of the block-Jacobi preconditioner by heavily exploiting the GPU registers. Furthermore, we reduce data movements by performing the column permutations (required for pivoting) implicitly.

6.2 Background and Related Work

6.2.1 Block-Jacobi preconditioning

The block-Jacobi method arises as a blocked variation of its (scalar) Jacobi counterpart that extends the idea of diagonal scaling to block-diagonal inversion with the diagonal blocks of $A \in \mathbb{R}^{n \times n}$ gathered into $D = \text{diag}(D_1, D_2, \dots, D_N)$, $D_i \in \mathbb{R}^{m_i \times m_i}$, $i = 1, 2, \dots, N$, $m_1 + \dots + m_N = n$.

An important question in this context is how to choose the diagonal blocks. In the optimal case, these blocks should reflect the properties of the coefficient matrix A . Fortunately, many linear systems exhibit some inherent block structure. For example, if A comes from a finite element discretization of a partial differential equation (PDE), each element typically has multiple variables associated with it [7]. These variables are typically tightly coupled, leading to dense diagonal blocks. As all variables belong to the same element, they share the same column sparsity pattern. Such sets of variables are often referred to as *supervariables*. A popular strategy to determine the blocks for block-Jacobi is *supervariable blocking* [7], which is based on identifying variables sharing the same column-nonzero-pattern. Depending on the predefined upper bound for the size of the blocks, multiple supervariables adjacent in the coefficient matrix can be agglomerated within the same diagonal block. To help ensure that supervariables accumulated into the same Jacobi block are coupled, the matrix should be ordered so that nearby variables are also close in the PDE mesh. This is satisfied for locality-preserving ordering techniques such as reverse Cuthill-McKee or natural orderings [7].

There exist different strategies of employing a block-Jacobi preconditioner in an iterative solver setting. One option is to explicitly compute the block-inverse before the iterative solution phase, and apply the preconditioner in terms of a matrix-vector multiplication. A second approach is to only extract the diagonal blocks in the preconditioner setup phase, and solve one linear system per block within the preconditioner application. In-between these strategies falls the idea (explored in this paper) of factorizing the diagonal blocks in the setup, and performing small triangular solves in the preconditioner application. These three strategies differ in the workload size, and how this work is distributed between the preconditioner setup phase and the preconditioner application phase.

6.2.2 Solution of linear systems

The conventional procedure to solve a linear system with an $m \times m$ coefficient matrix D_i commences with the computation of the LU factorization (with partial pivoting) $P_i D_i = L_i U_i$, where L_i is unit lower triangular, U_i is upper triangular, and P_i is a permutation [10]. This is followed by the application of P_i to the right-hand side vector; and the solution of two triangular systems with L_i and U_i . Assuming a single right-hand side vector, this four-stage procedure requires $2m^3/3$ floating-point operations (flops) for a linear system of order m .

Gauss-Jordan elimination (GJE) is an efficient procedure for matrix inversion. In terms of theoretical cost and practical performance, GJE is competitive with the standard approach based on the LU factorization [5, 13]. However, if the goal is not the matrix inversion but retrieving the solution of a linear system, GJE incurs significant overhead, requiring $2m^3$ flops. The *Gauss-Huard* (GH) algorithm is a variant of GJE for the solution of linear systems with a computational cost consistent with that of the LU-based approach. Furthermore, the GH-based solver can be combined with a column version of the standard partial (row) pivoting to offer a numerical stability that is comparable with that of the LU-based solver enhanced with partial pivoting [9]. Figure 6.1 illustrates a Matlab implementation of the GH solver for a system $D_i x_i = b_i$. Column-pivoting is applied to the coefficient matrix during the factorization, and undone in the final step of the algorithm.

```

1 % Input : m x m nonsingular matrix block Di, right-hand side bi
2 % Output : Di overwritten by the GH factorization, solution xi
3 p = [1:m];
4 for k = 1 : m
5     % Row elimination. Matrix-vector product (GEMV)
6     Di(k,k:m) = Di(k,k:m) - Di(k,1:k-1) * Di(1:k-1,k:m);
7     bi(k) = bi(k) - Di(k,1:k-1) * bi(1:k-1);
8     % Column pivoting (explicit)
9     [abs_ipiv, ipiv] = max(abs(Di(k,k:m)));
10    ipiv = ipiv+k-1;
11    [Di(:, ipiv), Di(:, k)] = swap(Di(:, ipiv), Di(:, k));
12    [p(ipiv), p(k)] = swap(p(ipiv), p(k));
13    % Diagonalization. Vector scaling (SCAL)
14    Di(k, k+1:m) = Di(k, k+1:m) / Di(k, k);
15    bi(k) = bi(k) / Di(k, k);
16    % Column elimination. Outer product (GER)
17    Di(1:k-1, k+1:m) = Di(1:k-1, k+1:m) - Di(1:k-1, k) * Di(k, k+1:m);
18    bi(1:k-1) = bi(1:k-1) - Di(1:k-1, k) * bi(k);
19 end
20 xi(p) = bi;

```

Figure 6.1: Simplified loop-body of the basic GH implementation in Matlab notation.

In the GH algorithm we distinguish a “decomposition phase”, operating exclusively on the matrix entries; and an “application phase”, which transforms the right-hand side vector of the linear system into the sought-after solution. Although we realize that a GH decomposition does not provide a factorization in the classical LU interpretation, we use this term to distinguish the operations on the system matrix (in the preconditioner setup phase) from those on the right-hand side vector (in the preconditioner application).

6.2.3 GH with implicit pivoting

Pivoting can be a costly operation on parallel architectures, as this process involves two different memory access patterns. For example, column pivoting requires the selection of the largest element in a single row, and this is followed by an exchange of two columns at each iteration of the algorithm. If the matrix rows are distributed among the processors column-wise, the selection can be performed using a parallel reduction, while the column swap exposes little parallelism since all but the two processors involved in the swap remain idle. If the matrix is distributed row-wise the situation is reversed.

To tackle this problem for GJE, in [4] we introduced implicit (row) pivoting, eliminating the sequential part of the process by postponing the exchange step, and applying the swaps from all iterations at the end of the algorithm in parallel. In GJE, implicit (row) pivoting is enabled by noticing that its iterations are row-oblivious (i.e., the operations performed do not depend on the actual position of the row in the matrix, but only on the currently selected pivot row).

This is different for GH, as the operations performed on the columns do depend on the column position in the matrix. Precisely, iteration i updates only the columns with index greater than i . By noticing that all the columns with smaller index were already chosen as pivots in an earlier iteration, this requirement can be reformulated in a column-oblivious manner: iteration i updates only the columns that have not yet been chosen as pivot. This binary predicate still does not provide enough information to compute the GEMV on line 6 of Figure 6.1, as the order of elements in the input vector depends on the order of columns in the matrix: j -th value in the vector is the i -th element of the j -th column (i.e., the i -th element of the column chosen as pivot in the j -th iteration). Without exchanging the columns, this information can be propagated by maintaining a list of previous pivots.

This implicit pivoting strategy incurs some instruction and memory overhead, however it is insignificant to the savings coming from omitting explicit column swapping. We note that introducing implicit pivoting in GH preserves the numerical stability of the original algorithm.

6.2.4 Related work on batched routines

The term “batched” refers to a setting where a certain computational kernel is applied to a large set of (independent) data items [1]. The motivation for having a special design (and interface) of those routines is that applying them to a single data item does not fully utilize the hardware resources, so handling the distinct problems serially leaves them unused. The independence of the data items allows the application of the operations to multiple data items in parallel. In particular, for architectures providing a vast amount of hardware concurrency, such as GPUs, replacing standard algorithms with data-parallel implementations can result in significant performance gains [2].

For the previously mentioned GJE, we demonstrated in [4] how the inversion of a set of small linear systems can be realized efficiently on NVIDIA GPUs. There we also described how to combine the batched routine with data extraction and insertion steps to efficiently generate a block-inverse matrix for block-Jacobi preconditioning.

6.3 Design of CUDA Kernels

Utilizing GH for a block-Jacobi preconditioner requires the initial extraction of the diagonal blocks from the matrix A stored in the compressed sparse row (CSR) format (as required by MAGMA [12]). After this, the sequence of blocks is fed into a batched version of the GH algorithm (BGH), and the resulting decomposition, along with the pivoting information, is inserted back into the preconditioner matrix and the pivot vector, respectively. The preconditioner application step uses this information to apply BGH to the right-hand side vector, ultimately turning each vector block into the solution of the corresponding small linear system. Details of the distinct steps and their efficient realization on GPUs are described in this section.

6.3.1 Variable Size Batched Gauss-Huard decomposition

The BGH kernel applies the GH algorithm to factorize a set of small independent blocks. Following the kernel design in [4], the BGH implementation takes advantage of the large register count and warp shuffle instructions in recent GPU architectures. The GH decomposition commences by reading the diagonal block, with each thread storing a single column in registers. The actual computation is realized entirely in the registers, using warp shuffles for inter-thread communication. This approach eliminates the latency of memory and caches, as well as the load and store instructions, decreasing the complexity of the kernel. Column permutations required to perform the pivoting can be avoided by using implicit pivoting as described in Section 6.2. The application of the pivoting is delayed until the preconditioner matrix is inserted into the sparse data structure. An additional register array is used to store the pivoting information, and this array is replicated in each thread for quick access during the GEMV step.

The use of warp shuffles and registers limits the scope of the kernel to blocks of size up to 32, as the number of threads cannot exceed the warp size. Nevertheless, this covers the typical application scenario for block-Jacobi preconditioning [4].

6.3.2 Batched Gauss-Huard application

The solution of the linear system lacks any reuse of matrix elements. Hence, the kernel has to be designed with focus on optimizing memory access. Since the solution vector is needed in each step of the GH algorithm, it is read into registers in an interleaved pattern, with each thread storing one component of the vector.

Each outer loop iteration k of the GH application algorithm updates the k -th element of the solution vector with the dot product between the first $k - 1$ elements of the k -th matrix row and the solution vector (Figure 6.1, line 7). This can be implemented as a parallel reduction using warp shuffles. After that, a parallel AXPY updates the first $k - 1$ elements of the solution vector using the first $k - 1$ elements of k -th column (Figure 6.1, line 17). To attain coalescent memory access for both operations, the matrix $\tilde{D}_i$ can be decomposed into lower and upper triangular parts: $\tilde{D}_i = L_i + U_i$, with the diagonal belonging to the former. Matrix L_i is stored in row-major order to enable coalescent access to its rows and U_i in column-major order to provide fast access to the columns. Alternatively, matrix U_i can be transposed with respect to the anti-diagonal to convert its columns into rows while preserving its upper triangular structure. The resulting matrix $\hat{D}_i = L_i + U_i^{AT}$ is then stored in row-major

Figure 6.2: Speedup of BGJE and BGH / BGHT over BLU taken from the MAGMA software package.

order. In this manner, both, the rows of L_i and the columns of U_i (i.e. the rows of U_i^{AT}) can be accessed in coalescent manner.

The application process is completed by writing the solution vector back to memory, taking into account the permutation generated in the decomposition step.

6.3.3 Batched data extraction and insertion

The extraction of a diagonal block from the sparse coefficient matrix is similar to the “shared extraction” strategy in [4]. A notable difference, though, is that the block has to be distributed among the threads in column-wise fashion. As a result, the strategy can be implemented using only a fraction of the amount of shared memory. Once the extraction of a single row to shared memory is completed, the elements of this row are immediately copied into the registers (one element per thread), making the shared memory locations available for the extraction of the next row. Thus, conversely to the shared memory extraction in [4], the shared memory has to hold only a single row of the diagonal block.

The insertion step writes the decomposed blocks $\tilde{D}_i$ (or $\hat{D}_i$) back into the sparse matrix structure. Storing $\tilde{D}_i$ is faster due to coalescent memory writes. At the same time, storing $\tilde{D}_i$ results in noncoalescent memory access to the matrix U_i during the preconditioner application. In contrast, storing $\hat{D}_i$ moves the cost of noncoalescent memory access from preconditioner application to preconditioner setup. The noncoalescent memory accesses when writing $\hat{D}_i$ can be avoided by first preparing the matrix in shared memory, and then writing it to the main memory in coalescent fashion. However, this significantly increases shared memory consumption, decreasing the number of warps that can be scheduled per SM, and ultimately achieves lower performance. Consequently, we refrain from presenting performance results for this approach.

6.4 Numerical Experiments

We evaluate the block-Jacobi preconditioner based on GH by comparing the performance of the BGH implementation against kernels offering similar functionality. Furthermore, we assess the effectiveness of the arising preconditioner within an iterative solver setting. We emphasize that the implementations are part of the same software stack, and that the kernel implementations are similar in design, and received the same level of tuning. This ensures a fair comparison and conclusions with fair credibility.

6.4.1 Hardware and software framework

We use an NVIDIA Tesla P100 GPU with full double precision support. We employ NVIDIA’s GPU compilers that are shipped with CUDA toolkit 8.0. All kernels are implemented using the CUDA programming model and are designed to integrate into the MAGMA-sparse library [12]. MAGMA-sparse is also leveraged to provide a testing environment, the block-pattern generation, and the sparse solvers. All computations use double precision

Figure 6.3: Left: Runtime of block-Jacobi application for distinct block sizes and a problem of row size 1,000,000. Right: BiCGSTAB convergence variations when using block-Jacobi based on BGJE or BGH.

arithmetic, as this is the standard in scientific computations. Since the complete algorithm is executed on the GPU, the details of the CPU are not relevant for the following experimentation.

6.4.2 Performance of BGH

Figure 6.2 compares the performance of our BGH implementation with alternative approaches providing similar functionality. The baseline implementation is the batched LU factorization (BLU) kernel provided in the MAGMA library (version 2.0 [12]), designed for the LU factorization of a large set of small problems. We note that, conversely to the BGJE and BGH kernels, the scope of the BLU kernel is limited to settings where all small systems are of the same size. The results in the figure are expressed in terms of speedup over this routine. BGJE is the implementation proposed in [4] which explicitly generates a block-inverse for Jacobi preconditioning. For GH, we also include data for the variant storing the upper triangular part transposed (BGHT), allowing for faster access during the preconditioner application. As BLU currently only supports batches of equal-size problems, while BGJE and BGH only work for problems of order up to 32, we limit the analysis to block sizes 16 and 32.

For block size 16 (see left plot in Figure 6.2) and large batch counts, the BGJE kernel is about $6\times$ faster than BLU; and we observe even larger speedups for smaller batch sizes. Adding the transposed storage to the BGH kernel has a minor impact: for relevant batch sizes, both BGH and BGHT are $12\text{--}20\times$ faster than BLU. This is different for block size 32 (see right-hand side plot in Figure 6.2): Adding the transposed storage to the BGH kernel reduces the speedup of BGHT over BLU to values around $8.5\times$, but BGH remains more than $10\times$ faster than BLU. Even though BGJE computes the explicit inverse, and hence executes more operations than BLU and GH, this kernel is more than $7.5\times$ faster than BLU. For completeness, we mention that for block size 32, BGJE delivers about 600 GFLOPS (billions of flops/second) on this architecture (see [4]).

6.4.3 Performance of block-Jacobi application

Figure 6.3 (left) shows the runtime of three preconditioner application strategies: sparse matrix-vector multiplication if the preconditioner was generated using GJE (BGJE), versus Gauss-Huard application using diagonal blocks $\tilde{D}_i$ (BGH) or $\hat{D}_i$ (BGHT). We fix the problem size to 1 million rows and consider different diagonal block sizes. As could be expected, the approach based on the sparse matrix-vector product is always faster than both GH-based application strategies, which can be attributed to the lower number of flops and better workload balance.

Even though half of the memory accesses in BGH are noncoalescent, we observe only minor performance deviations from the BGHT kernel for block sizes smaller than 16. An explanation is that smaller blocks fit into less cache lines, so they can be read only once, and kept in cache for the duration of the kernel. If this happens, the noncoalescent reads are as fast as the coalescent ones, and both strategies result in the same performance. Once the blocks become too large for cache, some cache lines need to be evicted during the GH application, and noncoalescent reads in BGH start to impact the performance of this approach.

Block size 16

Block size 32

Figure 6.4: Left: Total execution time (setup+solve) for BiCGSTAB enhanced with block-Jacobi preconditioning based on either BGJE, BGH or BGHT. Top-level results are for a block-size bound 16; bottom-level results are for a block-size bound 32. Right: Decomposition of the total execution time into preconditioner setup time and iterative solver runtime for the left-most test cases.

6.4.4 Iterative solver analysis

We next assess the efficiency of the distinct strategies for block-Jacobi preconditioning in an iterative solver framework. For this purpose we integrate the block-Jacobi preconditioner(s) into the BiCGSTAB iterative solver provided in MAGMA-sparse [3], and test the preconditioned solver setting for a variety of linear systems. The test matrices are chosen from the SuiteSparse matrix collection [8], combined with a right-hand side vector with all entries equal to one. We start the iterative solver with an initial guess of zero, and stop once the relative residual norm is decreased by six orders of magnitude. We allow for up to 10,000 iterations.

First, we evaluate whether the difference between GJE and GH in terms of numerical stability has any impact on the preconditioner efficiency. At this point, we recognize that rounding can have significant effect on a preconditioner's efficiency, and a more accurate preconditioner does not inevitably result in faster convergence of the iterative solver. Figure 6.3 (right) displays the convergence difference of BiCGSTAB depending on whether the block-Jacobi preconditioner is based on GH or GJE. The x-axis of the histogram reflects the iteration overhead; the y-axis shows the number of test cases for which GJE provided a “better” preconditioner (bars left of center) or GH did (bars right of center). For all block sizes, the majority of the problems is located in the center, reflecting the cases where both methods resulted in the same iteration count. Furthermore, the histogram exposes a high level of symmetry, suggesting that the numerical stability of the method based on

explicit inversion plays a minor role.

While being similar from the convergence rate point of view, the pending question is whether there exist any performance differences making GJE or GH superior. The plot in the left-hand side of Figure 6.4 arranges the test systems according to increasing execution time of the BiCGSTAB solver preconditioned with block-Jacobi based on BGJE. The block structure was generated via the supervariable blocking routine provided by MAGMA-sparse with a maximum block size of 16 (top) and 32 (bottom). The execution times comprise both the preconditioner setup and the iterative solver times. In addition to the block-Jacobi using BGJE, we also include the total solver runtime for the variants using BGH and BGHT.

In most cases, the BGH and BGHT execution times are close, or even match those of BGJE. In particular for block size 32, the results may suggest that BGJE is slightly better if the execution time is large (right-most data). Conversely, BGH and BGHT are faster if the solution time is small (left-most data). On the right of Figure 6.4 we decompose the total solution time into its preconditioner setup and iterative solver components for the first 10 test problems. For these instances, the preconditioner setup time accounts for a significant portion of the total solver execution time, and the higher cost of explicit block-inversion is not compensated for by the slightly faster preconditioner application.

6.5 Concluding Remarks

We have designed data-parallel GPU kernels for the efficient generation and application of block-Jacobi preconditioners, based on the Gauss-Huard method, which can be embedded into any Krylov-based iterative solver for sparse linear systems. Our kernels exploit the intrinsic parallelism in the two algorithmic steps, preconditioner generation and preconditioner application. In contrast, our kernel implementation is specifically designed for small block sizes, exploiting the GPU registers to outperform their MAGMA counterparts by a large margin. Furthermore, our variable size batched Gauss-Huard kernel integrates an implicit version of column pivoting to eliminate costly data movements due to column permutations, while delivering the same numerical stability. Compared with block-Jacobi based on our register-tuned batched Gauss-Jordan elimination, which is also designed for small block sizes, the variable size batched Gauss-Huard kernels offer faster preconditioner generation and higher numerical stability, but slower preconditioner application. Our experimental results, using a state-of-the-art NVIDIA's P100 GPU, are consistent with this analysis. Implementing the block-Jacobi preconditioner on top of the batched Gauss-Huard provides better performance if the iterative solver converges within a small number of iterations. In these cases the cost of the preconditioner generation is considerable compared with the whole iterative solve (including the preconditioner application).

Acknowledgments

This material is supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award #DE-SC-0010042. The researchers from UIJ were supported by project TIN2014-53495-R of MINECO and FEDER.

Bibliography

- [1] A. Abdelfattah, H. Anzt, J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. YarKhan. Linear algebra software for large-scale accelerated multicore computing. *Acta Numerica*, 25:1–160, 2016.
- [2] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. Performance tuning and optimization techniques of fixed and variable size batched Cholesky factorization on GPUs. *Procedia Computer Science*, 80: 119–130, 2016.
- [3] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler. Efficiency of general Krylov methods on GPUs — an experimental study. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW'16*, pages 683–691. IEEE, 2016.

-
- [4] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'17, pages 1–10. ACM, 2017.
 - [5] P. Benner, P. Ezzatti, E. Quintana-Ortí, and A. Remón. Matrix inversion on CPU-GPU platforms with applications in control theory. *Concurrency and Computation: Practice and Experience*, 25(8):1170–1182, 2013.
 - [6] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, and A. Remón. Revisiting the Gauss-Huard algorithm for the solution of linear systems on graphics accelerators. In *In Proceeding of the 11th International Conference on Parallel Processing and Applied Mathematics*, PPAM 2015, pages 505–514, 2016.
 - [7] E. Chow, H. Anzt, J. Scott, and J. Dongarra. Using Jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning. *Journal of Parallel and Distributed Computing*, 119:219–230, 2018.
 - [8] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.
 - [9] T. J. Dekker, W. Hoffmann, and K. Potma. Stability of the Gauss-Huard algorithm with partial pivoting. *Computing*, 58(3):225–244, 1997.
 - [10] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.
 - [11] P. Huard. La méthode simplex sans inverse explicite. *E.D.F. Bull. de la Direction des Études et des Recherches*, 2(2):79–98, 1979.
 - [12] MAGMA 2.0.0. <http://icl.cs.utk.edu/magma/>, 2016.
 - [13] E. S. Quintana-Ortí, G. Quintana-Ortí, X. Sun, and R. van de Geijn. A note on parallel matrix inversion. *SIAM Journal on Scientific Computing*, 22(5):1762–1771, 2001.
 - [14] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.

Block-Jacobi Preconditioning Based on LU Factorization

Published as: H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. Variable-size batched LU for small matrices and its integration into block-Jacobi preconditioning. In *Proceedings of the 46th International Conference on Parallel Processing, ICPP'2017*, pages 91–100. IEEE, 2017

We present a set of new batched CUDA kernels for the LU factorization of a large collection of independent problems of different size, and the subsequent triangular solves. All kernels heavily exploit the registers of the graphics processing unit (GPU) in order to deliver high performance for small problems. The development of these kernels is motivated by the need for tackling this embarrassingly-parallel scenario in the context of block-Jacobi preconditioning that is relevant for the iterative solution of sparse linear systems. Our experimental comparison with existing CUDA kernels offering similar functionality, including those in NVIDIA's cuBLAS library, on a state-of-the-art NVIDIA P100 GPU, reveal significant performance advantages of our kernels.

7.1 Introduction

The development of batched routines for linear algebra operations has received considerable interest in the past few years because of the hardware concurrency often exceeding the degree of parallelism present in the algorithms. At the same time, many problems arising in astrophysics, quantum chemistry, hydrodynamics and hyperspectral image processing, among others, require the application of the same computational kernel not only to one, but to a large number of data instances. Batched routines are optimized to tackle this embarrassingly-parallel scenario that comprises a large collection of independent problems, each of small dimension. Compared with conventional multi-threaded implementations of the *Basic Linear Algebra Subprograms* (BLAS) [5], optimized for moderate- and large-scale individual problem instances, batched routines may also exploit the parallelism available within the computational kernel, but mainly target the parallelism in-between the distinct data items.

With the increasing core-per-node ratio, there is an urging demand for batched routines, which are eventually expected to cover a significant fraction of the functionality currently supported by dense linear algebra libraries such as BLAS and the *Linear Algebra PACKage* (LAPACK) [1]. In addition, batched kernels are becoming a key ingredient for the solution of sparse linear systems via direct multifrontal solvers as well as for the efficient preconditioning of iterative solvers based on Krylov subspaces.

Preconditioning via a block-Jacobi scheme is a particularly simple technique that produces an effective acceleration of the iterative solve for some problem instances [13]. One option to realize block-Jacobi preconditioning requires to:

1. Extract multiple small-sized diagonal blocks of the sparse coefficient matrix of the linear system and factorize this collection of blocks during the preconditioner computation (setup).
2. Solve the resulting triangular systems during the preconditioner application (once per step of the iterative solve).

As the diagonal blocks are all pairwise independent, preconditioning via block-Jacobi naturally leads to a batched scenario. Our effort in this work is oriented towards elaborating efficient batched routines for these two steps, preconditioner setup and preconditioner application, yielding the following contributions:

- A batched LU factorization routine on GPUs tuned for small sizes.
- A complementary batched lower and upper triangular solve routine on GPUs tuned for small sizes.
- An implicit pivoting technique that preserves the stability of the factorization without explicitly swapping the matrix elements in memory.
- A routine for efficiently extracting the diagonal blocks from a sparse matrix layout that ensures a good workload balance even for matrices with an unbalanced nonzero distribution.
- A complete block-Jacobi preconditioner ecosystem based on batched LU factorization and batched triangular solves that improves time-to-solution of iterative Krylov solvers for a large set of test problems.

At this point, we note that the term “batched” has been frequently used to refer to a large collection of small-size problems. However, the definition of large and small are at best blurry. Here we target a realistic case study for block-Jacobi preconditioning, where small can be defined as the diagonal blocks that participate in steps 1)–2) above being in the range 4×4 to 32×32 , while large refers to thousands or even tens of thousands of independent problems. We consider this as scientifically relevant as the block-Jacobi preconditioner aims at reflecting the sparsity block structure of a finite element discretization.

Due to their small size, processing the problem instances usually involves memory-bound operations, which can question the utilization of discrete accelerators (with a memory detached from that of the host memory). For the particular case of block-Jacobi preconditioning, the use of a discrete hardware accelerator device is justified as the Krylov solvers require the coefficient matrix to reside in the device memory in order to build up the Krylov subspace via the Krylov iterations. Therefore, the cost of transferring this matrix from the host to the accelerator is quickly amortized, and the on-device generation of the batched block-Jacobi preconditioner incurs no additional host-to-device communication.

The rest of the paper is structured as follows. In Section 7.2 we review a few related works and offer a brief survey of (batched) factorization for the solution of linear systems. In Section 7.3 we describe the implementation of our CUDA kernels for batched LU factorization (paying special attention to the introduction of implicit pivoting), batched triangular system solves, and the extraction procedure. In Section 7.4 we assess the performance of the new batched kernels on an NVIDIA Tesla P100 (Pascal) GPU, and in Section 7.5 we close the paper with a discussion of remarks and future work.

7.2 Background and Related Work

7.2.1 Block-Jacobi preconditioning

For a coefficient matrix $A \in \mathbb{R}^{n \times n}$, the block-Jacobi method can be regarded as a straight-forward extension of its (scalar) Jacobi counterpart. Concretely, instead of splitting the coefficient matrix as $A = L + D + U$ (with diagonal $D = (\{a_{ii}\})$, lower triangular $L = (\{a_{ij} : i > j\})$ and upper triangular $U = (\{a_{ij} : i < j\})$), the block-Jacobi variant gathers the diagonal blocks of A into $D = (D_1, D_2, \dots, D_N)$, $D_i \in \mathbb{R}^{m_i \times m_i}$, $i = 1, 2, \dots, N$, with $n = \sum_{i=1}^N m_i$. (For simplicity, hereafter we assume that all blocks have the same size m and n is an integer multiple of m .) The remaining elements of A are then partitioned into matrices L and U such that L contains the elements below the diagonal blocks while U comprises those above them [3]. The block-Jacobi method is well-defined if all diagonal blocks are non-singular, and the resulting preconditioner is expected to work effectively if the blocks succeed in reflecting the nonzero structure of the coefficient matrix A .

Fortunately, many linear systems exhibit some inherent block structure, for example because they arise from a finite element discretization of a partial differential equation (PDE), with multiple variables associated to each element [3]. The variables belonging to the same element usually share the same column sparsity pattern, and the set of variables is often referred to as a *supervariable*. *Supervariable blocking* [6] aims to identify variables sharing the same column-nonzero-pattern, and turns this information into a block-structure that can be used, for example, in block-Jacobi preconditioning. Depending on the pre-defined upper bound for the size of the

diagonal blocks, multiple supervariables adjacent in the coefficient matrix can be clustered within the same diagonal block [6]. This is particularly efficient if the supervariables accumulated into the same Jacobi block are tightly coupled, which is the case if the variables ordered close-by in the matrix representation belong to elements that are nearby in the PDE mesh. Some reordering techniques such as reverse Cuthill-McKee or natural orderings preserve this locality [6].

Although there exist different strategies to integrate a block-Jacobi preconditioner into an iterative solver setting, in this paper we focus on an approach that factorizes the diagonal blocks in the preconditioner setup, and then applies the preconditioner in terms of triangular solves. Alternatively, it is possible to explicitly compute the block-inverse prior to the iterative solution phase, and apply the preconditioner as a matrix-vector multiplication. These two strategies primarily differ in the workload size, and how this work is distributed between the preconditioner setup and the preconditioner application. Additionally, the factorization-based approach might exhibit more favorable numerical stability as it avoids the explicit inversion of the blocks in D .

7.2.2 Solution of linear systems via the LU factorization

The standard procedure to solve a dense linear system $D_i x = b$, for a square block D_i of order m and vectors x, b each with m entries, consists of the following four steps [10]:

1. The computation of the LU factorization (with partial pivoting) $PD_i = LU$, where L is unit lower triangular, U is upper triangular, P is a permutation matrix, and all three matrices L, U, P are of the same order as D_i ;
2. the application of the permutation P to the right-hand side b ; i.e., $b := Pb$;
3. the solution of the unit lower triangular system $Ly = b$ for y ; and
4. the solution of the upper triangular system $Ux = y$ to obtain the sought-after vector x .

The computational cost of this four-step solution process is $\frac{2}{3}m^3 + \mathcal{O}(m^2)$ flops (floating-point arithmetic operations), where the dominating term $\frac{2}{3}m^3$ comes from the factorization step. Neglecting the pivoting process associated with the permutation matrix P can result in the triangular factors becoming singular and the breakdown of the algorithm [10]. Partial pivoting limits the process to row exchanges only, it is numerically stable in practice, and has become the norm in standard implementations of the LU factorization.

7.2.3 Batched solution of small linear systems

Batched routines for small-size problems play an important role in the context of preconditioning iterative methods for sparse linear systems. One example is the technique of block-Jacobi preconditioning, where the sparse coefficient matrix is scaled with its block-inverse [13]. This type of scaling requires the solution of a set of small linear systems induced by the diagonal blocks in D , which can be addressed via a factorization-based method. As argued earlier, for each block D_i , of dimension m , the cost of its LU factorization is $\frac{2}{3}m^3$ flops, while solving the triangular block system for a single block and right-hand side requires $2m^2$ flops. Alternatively, as the block-diagonal scaling is applied at each iteration of the solver, it may pay off to explicitly compute the block-inverse prior to the iteration process, at a cost of $2m^3$ flops (per block). With this approach, the preconditioner application can then be cast in terms of the matrix-vector product, with a cost of $2m^2$ flops (per block), but a much faster execution than a triangular block solve.

These two approaches, factorization-based and inversion-based, differ in the computational cost, numerical stability, and how they distribute the workload between preconditioner setup and preconditioner application. Which strategy is preferable depends on how often the preconditioner is applied and the size of the distinct diagonal blocks.

In block-Jacobi preconditioning, the diagonal blocks are typically chosen to be of small size, for example when reflecting the block structure of a system matrix coming from a finite element discretization. At the same time, the number of these small blocks is typically large, which motivates the use of batched routines. For GPU architectures, we showed in [3] how to realize an inversion-based block-Jacobi preconditioner efficiently using Gauss-Jordan elimination (GJE). As the explicit inversion may be questionable in terms of numerical stability,

```

1 % Input  : m x m nonsingular matrix block Di
2 % Output : Di overwritten by its L, U factors
3 p = [1:m];
4 for k = 1 : m
5     % Pivoting
6     [abs_ipiv , ipiv]      = max(abs(Di(k:m,k)));
7     ipiv                  = ipiv+k-1;
8     [Di(k,:), Di(ipiv,:)] = swap(Di(ipiv,:), Di(k,:));
9     [p(k),  p(ipiv)]      = swap(p(ipiv), p(k));
10
11     % Gauss transformation
12     d                    = Di(k,k);          % Pivot
13     Di(k+1:m,k)         = Di(k+1:m,k) / d;    % SCAL
14     Di(k+1:m,k+1:m)     = Di(k+1:m,k+1:m) ...
15                         - Di(k+1:m,k) * Di(k,k+1:m); % GER
16 end

1 % Input  : m x m nonsingular matrix block Di
2 % Output : Di overwritten by its L, U factors
3 p = zeros(1, m);
4 for k = 1 : m
5     % Implicit pivoting
6     abs_vals              = abs(Di(:,k));
7     abs_vals(p>0)         = -1; % exclude pivoted rows
8     [abs_ipiv , ipiv]     = max(abs_vals);
9     p(ipiv)               = k;
10
11     % Gauss transformation
12     d                    = Di(ipiv,k);        % Pivot
13     Di(p==0,k)           = Di(p==0,k) / d;    % SCAL
14     Di(p==0,k+1:m)       = Di(p==0,k+1:m) ...
15                         - Di(p==0,k) * Di(ipiv,k+1:m); % GER
16 end
17 % Combined row swaps
18 p(p) = 1:m; % Invert the permutation
19 Di   = Di(p,:);

```

Figure 7.1: Loop-body of the basic LU factorization in Matlab notation using explicit and implicit partial pivoting (top and bottom, respectively).

in [4] we compared this solution to a block-Jacobi preconditioning procedure based on the factorization of diagonal blocks. The factorization method we used in that comparison was the Gauss-Huard (GH) algorithm [11], which is algorithmically similar to GJE.

In this paper we extend our survey on using batched routines for block-Jacobi preconditioning by addressing the factorization of the diagonal blocks via the mainstream LU factorization. From the numerical perspective, the LU factorization (with partial pivoting) and the GH algorithm (with column pivoting) present the same properties. However, they build upon distinct algorithms, resulting in different implementations, and, consequently, distinguishable computational performance.

7.3 Design of CUDA Kernels

This section describes the implementation of efficient CUDA kernels for both batched LU factorization as well as triangular system solves specifically tuned for small problem sizes where the system matrix (corresponding to a diagonal block in block-Jacobi preconditioning) contains at most 32×32 elements. This is consistent with the batched implementation of GH in [4] which is also designed for small problems, of the type arising in block-Jacobi preconditioning.

Figure 7.2: Using batched factorization and batched triangular solve (TRSV) routines in a block-Jacobi preconditioner setting.

7.3.1 Batched LU factorization (GETRF)

A simplified MATLAB routine for the (right-looking) LU factorization of a square block D_i is shown in Figure 7.1 (top). This algorithmic variant lies in the foundations of our batched CUDA kernel for this factorization that we discuss next.

Recent GPU architectures from NVIDIA feature a large amount of registers per thread which makes it possible to assign problems of size up to 32×32 to a single warp. Each thread then stores one row of the system matrix D_i into the local registers, while warp shuffle instructions allow to access elements from other rows (e.g., when performing the updates in lines 13–15). Using this technique, it is possible to read the system matrix only once, and perform the whole factorization process in the registers, avoiding the latency of memory and caches, as well as additional load and store instructions.

A complementary optimization, also important, addresses the pivoting procedure ensuring the practical stability of the LU factorization. Even though the selection of the pivot row $ipiv$ (lines 6–7) in step k of the factorization can be realized efficiently using a parallel reduction, the actual exchange of rows k and $ipiv$ (lines 8–9) on a GPU is costly, as it involves only the two threads holding these rows, while the remaining threads stay idle. To tackle this issue, in [3, 4] we proposed an implicit pivoting procedure for GJE and GH, which avoids the explicit row swaps and combines them into a single, easily parallelizable permutation, which is performed after the main loop. This technique can also be applied to LU by observing the following:

- During step k of the factorization, the operation performed on each row of the matrix (lines 13–15) depends only on the elements in this row and the pivot row $ipiv$ (which was exchanged with row k when using standard, explicit pivoting as in Figure 7.1).
- The type of operation to perform on each row can be derived without knowing its position in the matrix: if the row has already been pivoted, then no operation is required (in Figure 7.1 such row was exchanged with one of the first k rows); otherwise the k -th element of the row has to be scaled (line 13), and an AXPY needs to be performed on the trailing vector (lines 14–15).

Figure 7.3: Illustration of the “lazy” and “eager” algorithmic variants (top and bottom, respectively) for the solution of a unit lower triangular system.

```

1 % Input  : m x m matrix L, rhs vector b
2 % Output : Vector b overwritten by the solution y
3 %         of Ly = b
4 for k = 2 : m
5     b(k) = b(k) - L(k,1:k-1) * b(1:k-1); % DOT
6 end

1 % Input  : m x m matrix L, rhs vector b
2 % Output : Vector b overwritten by the solution y
3 %         of Ly = b
4 for k = 1 : m-1
5     b(k+1:m) = b(k+1:m) - L(k+1:m,k) * b(k); % AXPY
6 end
    
```

Figure 7.4: Loop-body of the “lazy” and “eager” algorithmic variants (top and bottom, respectively) for the solution of a unit lower triangular system in Matlab notation.

These observations turn the implicit pivoting procedure for LU even more efficient than its counterpart applied in GH, as the operations performed by each thread do not depend on the previously selected pivot rows. Conversely, for GH a list of their indices has to be replicated in each thread [4]. As an illustration, the bottom factorization code in Figure 7.1 shows the implicit pivoting approach in the LU factorization. A final optimization can be made by combining the row swap with the off-load of L and U to main memory, hereby eliminating all inter-thread communication induced by row swaps. In addition to writing the triangular factors, the pivoting information also has to be stored in main memory for the subsequent triangular solves, see Figure 7.2.

7.3.2 Batched triangular system solves (TRSV)

Unlike the LU factorization, the triangular system solves offer only a limited amount of data reuse. Each element of the triangular factors is needed only once, so explicitly keeping this value in registers does not offer any advantage. In contrast, the right-hand side vector b , which is overwritten with the solution vector, is reused. Hence, it is beneficial to read and distribute this vector across the registers of the threads in the warp (one element per thread).

The permutation $b := Pb$ coming from pivoting is performed while reading b into the registers: Each element is stored to the registers of the correct threads. This step is followed by a unit lower triangular solve,

Figure 7.5: Illustration of the memory requests for the shared memory extraction. The elements part of the diagonal block are colored in light orange, the elements that have already been extracted in light blue. We assume warps of 4 threads, and visualize the data read by the distinct threads at each iteration with dashed cells. If an element part of the diagonal block is currently accessed (dark orange) it is extracted and stored in the correct location in shared memory (dark orange also in the shared memory). We only show the accesses to the vector storing the col-indices of the CSR matrix structure [13]; the access to the actual values induces far less overhead, as these memory locations are accessed only if a location belonging to a diagonal block is found. In that case, the access pattern is equivalent to the one used for col-indices.

and finally an upper triangular solve. As these operations are similar, we will for brevity only discuss the lower triangular solve (the solution of $Ly = b (= Pb)$) in detail. There exist different strategies for realizing the triangular solve: the “lazy” variant (Figure 7.3 and the code in Figure 7.4, top) relies on an inner (DOT) product to compute the final value of y_k at step k , while the “eager” one (Figure 7.3 and the code in Figure 7.4, bottom) leverages an AXPY to update the trailing vector $y_{k+1:m}$. In this case the latter variant is more convenient, as the parallelization of AXPY is straightforward, while the DOT product requires a reduction. The memory accesses to the system matrix are also different: the “lazy” variant reads one row per step, while the “eager” one reads one column. Therefore, assuming standard column-major storage, the “eager” variant also has the benefit of coalesced memory access.

7.3.3 Block-Jacobi preconditioning using batched LU

Using batched factorization routines for block-Jacobi preconditioning requires the extraction of the diagonal blocks from the sparse system matrix. This is a non-trivial step as accessing a (dense) diagonal block embedded in a sparse data structure (such as those typically used for storing the system matrix, e.g., CSR [13]) can be quite elaborate. Furthermore, exploiting the fine-grained parallelism provided by the GPU hardware in the extraction step makes this operation challenging for problems with an unbalanced sparsity pattern: Assigning the parallel resources to the distinct rows will inevitably result in severe work imbalance for problems with a very unbalanced nonzero distribution, like for example those arising in circuit simulation. Additionally, accessing the distinct rows in the row-major-based CSR layout in parallel results in non-coalescent data access. In [3] we proposed a strategy to overcome the latter drawback while simultaneously diminishing the effects of the former one by means of an intermediate step which stores the diagonal blocks in shared memory. Although we refrain from showing a comparison between the standard approach and the shared memory based strategy in the experimental section, we recall the central ideas of this *shared memory extraction* for convenience:

Instead of assigning the distinct threads within the warp to the distinct rows corresponding to the diagonal block, all threads of the warp collaborate to process each row. The threads accessing an element that is part of the diagonal block extract the respective value and store it into shared memory. This allows for coalescent access to the elements stored in CSR format, and avoids the load imbalance up to a level where load imbalance only occurs between threads of the same warp. After extracting the elements that are part of the diagonal block, they are copied into registers of the thread that will handle the respective row in the factorization process. Figure 7.5 visualizes this diagonal block extraction strategy [3].

7.4 Numerical Experiments

In this section we evaluate the performance of the batched LU factorization and triangular solve kernels tuned for small-size problems by comparing them against alternative kernels offering similar functionality. Concretely, our experimental analysis includes the following kernels:

- *Small-size LU*: The batched LU factorization and triangular solve kernels developed as part of this work.
- *Gauss-Huard*: The batched factorization and triangular solve kernels based on GH [4].
- *Gauss-Huard-T*: The factorization in this routine is identical to the GH kernels, except in that the triangular systems are stored in a transpose access-friendly mode to accelerate the triangular solves [4].
- *cuBLAS LU*: The batched LU factorization and triangular solve kernels available in NVIDIA’s cuBLAS package (version 8.0).

We point out that the first three implementations are part of the same software stack, the kernel implementations are similar in design, and received the same level of tuning. cuBLAS is a vendor-implementation, optimized specifically for the targeted architecture, but its source is not available. As variable block size is not supported by the batched kernels in cuBLAS, the experiments involving them were conducted using fixed block size for the entire batch. This ensures a fair comparison and credible conclusions.

In addition to the evaluation of the LU factorization and triangular solve kernels, we assess the effectiveness of the computed preconditioner integrated into the iterative IDR(4) solver for sparse linear systems [13].

Figure 7.6: Performance of batched factorization routines depending on the batch size.

7.4.1 Hardware and software framework

We employed an NVIDIA Tesla P100 GPU with full double-precision support in the experimentation together with NVIDIA’s GPU compilers that are shipped with the CUDA toolkit 8.0. Except for the GETRF and GETRS¹ routines taken from NVIDIA’s cuBLAS library [7], all kernels were designed to be integrated into the MAGMA-sparse library [12]. MAGMA-sparse was also leveraged to provide a testing environment, the block-pattern generation, and the sparse solvers. Since the complete algorithm is executed on the GPU, the details of the CPU are not relevant.

7.4.2 Performance of batched factorization routines

Figure 7.6 compares the performance of the four batched factorization routines in terms of GFLOPS (billions of flops per second). The left-hand side plots in the figure report the performance for a batch of matrices of size 16×16 . The reference implementation for batched LU-based GETRF taken from NVIDIA’s cuBLAS library achieves about 110 GFLOPS in single-precision (top row). In comparison, the small-size LU, Gauss-Huard and Gauss-Huard-T all achieve about 130 GFLOPS for this case. In double-precision (bottom row), the performance of the small-size LU is about 35% lower than that of the GH-based factorization routines, with the latter delivering about 100 GFLOPS. The scenario is different when the problem dimension is 32×32 (plots in the right-hand side of Figure 7.6): The performance of Gauss-Huard-T is then about 5% below that of Gauss-Huard, and the small-size LU outperforms both routines by a significant margin, achieving up to 600 GFLOPS in single-precision and 350 GFLOPS in double-precision. The cuBLAS counterpart providing the same functionality is $3.5\times$ slower, delivering about 100 GFLOPS only. The explanation for this block-size-dependent behavior is an implementation detail, which will be corrected as part of future work. Concretely, for block size $k < 32$, both the small-size LU and GH routines operate with a matrix of size 32×32 , padding the

¹Routine GETRS applies the sequence of permutations computed by the LU factorization routine to the right-hand side vector, followed by two triangular solves (TRSV).

Figure 7.7: Performance of batched factorization routines depending on the size of the matrices. The batch size is fixed to 40,000 systems.

input with zeros, but performing only the first k steps of the factorization. This benefits GH, since it implements a “lazy” factorization, while the “eager” (right-looking) algorithmic variant selected for the LU factorization performs more flops than its GH counterpart for block-size $k < 32$. By optimizing the algorithms specifically for smaller block sizes, we expect to observe the same behavior as that obtained for block size 32.

Figure 7.7 reports the performance as a function of the problem size. The results indicate that the non-coalescent writes in Gauss-Huard-T play a significant role only for problems of dimension larger than 16×16 . For single-precision, this also corresponds to the threshold from which the small-size LU starts to outperform the GH-type factorizations. In double-precision, the small-size LU is slower than the GH-based factorizations for problems smaller than 23×23 . The size-dependent results for cuBLAS LU reveal the system-specific optimizations: local performance peaks can be identified for sizes 8, 16, and 29 in single-precision arithmetic, and for dimensions 8 and 20 in double-precision arithmetic. Although we do not tune for specific sizes by handling multiple problems per warp, the small-size LU outperforms the cuBLAS LU for almost all sizes.

7.4.3 Performance of batched triangular solves

We next employ the same notation in order to distinguish the different batched implementations of the triangular solves that complement the factorization routines. On the left-hand side plot in Figure 7.8, we assess the performance of the triangular solves for a batch of matrices with size 16×16 . Unlike the factorization step, the performance for both GH variants and the small-size LU are almost identical in single- as well as double-precision arithmetic (44 GFLOPS and 37 GFLOPS, respectively).

For problems of size 32×32 (right-hand side plots in Figure 7.8), the more expensive Gauss-Huard-T factorization pays off by accelerating the triangular solve from 47 GFLOPS (for Gauss-Huard) to 80+ GFLOPS when using single-precision. In double-precision the triangular solves of Gauss-Huard-T are also about twice faster (70 GFLOPS) than those associated with the Gauss-Huard kernel (35 GFLOPS). The small-size LU achieves 90+ GFLOPS in single-precision and close to 80 GFLOPS in double-precision. This implies speed-up factors of $4.5\times$ and $4\times$ over cuBLAS, respectively.

Figure 7.9 analyzes the performance depending on the problem size. Conversely to the factorization step, the non-coalescent reads in the Gauss-Huard triangular solves harm the performance for problems larger than 16×16 . For Gauss-Huard-T, the price of non-coalescent access was paid in the factorization step. As a result, the Gauss-Huard-T triangular solves remain competitive with the small-size LU triangular solves. As in the factorization step, NVIDIA’s GETRS seems to be optimized for problem of dimension smaller than 16; nonetheless, this option achieves only a fraction of the performance of our small-size LU for all dimensions.

7.4.4 Analysis of block-Jacobi preconditioning for iterative solvers

In this section we assess the efficiency of batched factorization routines in the context of block-Jacobi preconditioning. For this purpose we enhance an IDR(4) Krylov solver (taken from the MAGMA-sparse open source software package [2]) with a block-Jacobi preconditioner that is generated via batched factorization

Figure 7.8: Performance of batched triangular solve routines depending on the batch size.

routines based on LU or GH, and applied in terms of triangular solves. The diagonal block structure is generated via the supervariable blocking routines available in MAGMA-sparse, and we only vary the upper bound for the size of the diagonal blocks. (At this point, we note that we do not include the cuBLAS batched LU in this comparison as it does not support variable problem size, which is needed for block-Jacobi preconditioning based on supervariable blocking.) We perform our tests using a set of 48 selected matrices from the SuiteSparse matrix collection [8] (see the column labeled as “Matrix in Table 7.1, and <http://www.cise.ufl.edu/research/sparse/matrices>). The test problems are listed along with some key characteristics in Table 7.1, and all carry some inherent block structure which makes them attractive targets for block-Jacobi preconditioning. We initialize the right-hand side vector with all its entries set to one, start the iterative solver with an initial guess of zero, and stop once the relative residual norm is decreased by six orders of magnitude. We allow for up to 10,000 iterations.

Although both the LU-based and GH-based factorizations present the same practical stability [9], we acknowledge the possibility of rounding effects. At this point, we note that rounding errors can have significant effect on the convergence rate of the Krylov solver, and a more accurate factorization (preconditioner setup) does not inevitably result in faster convergence of the preconditioned iterative solver. Figure 7.10 displays the convergence difference of IDR(4) depending on whether the block-Jacobi preconditioner is based on LU or GH. The x-axis of the histogram reflects the iteration overhead, while the y-axis shows the number of test cases for which LU provided a “better” preconditioner (bars left of center) or GH did (bars right of center). For all block sizes, the majority of the problems is located in the center, corresponding to those problem instances where both methods resulted in the same iteration count. Furthermore, the histogram exposes a remarkable level of symmetry, suggesting that, although rounding effects do occur, none of the factorization strategies is generally superior.

In addition to the convergence rate (iteration count), we are also interested in comparing the practical performance of the two factorization strategies, in terms of execution time, in a block-Jacobi setting. In Figure 7.11 we compare the total execution time (preconditioner setup time + iterative solver runtime) of the IDR(4) solver using a block-Jacobi preconditioner based on either LU, GH or GH-T. For this experiment, we use an upper

Figure 7.9: Performance of batched triangular solve routines depending on the size of the matrices. The batch size is fixed to 40,000 systems.

Figure 7.10: IDR(4) convergence using block-Jacobi preconditioning based on LU factorization or GH-factorization: Iteration overhead in % if LU provides the better preconditioner (left of center) or GH does (right of center).

bound of 32 for the supervariable agglomeration. In most cases, the performance differences between the three options are negligible. Only due to rounding-error, and derived iteration count differences, one of the methods becomes superior. The differences between GH and GH-T come from the faster preconditioner generation combined with a potentially-faster application of the latter. The matrices are ordered in the x-axis according to the total execution time of the solver and can be identified by the corresponding index in Table 7.1 (see column labelled as “ID”). The 4 missing cases correspond to matrices for which the solver did not attain convergence.

To close this section, Table 7.1 lists all test matrices along with the convergence behavior and execution time when using different upper bounds for the Jacobi blocks in an IDR(4) solver preconditioned with the small-size LU-based block-Jacobi. The results suggest that larger block sizes typically improve the solver convergence with respect to both iteration count and time-to-solution.

7.5 Concluding Remarks and Future Work

We have presented flexible-size batched CUDA kernels for the solution of linear systems via the LU factorization that are optimized for small-size problems and outperform existing counterparts offering the same functionality by a large margin. This performance is achieved by extensive use of the GPU registers, and the integration of an implicit pivoting technique that preserves numerical stability while removing the costly data movements due to the row exchanges.

Combined with an efficient strategy for the extraction of the diagonal blocks from a sparse data structure, we

Figure 7.11: Total execution time (setup+solve) for IDR(4) enhanced with block-Jacobi preconditioning based on either LU or GH factorization. The size of the distinct diagonal blocks is adapted to the system matrix via supervariable blocking with 32 as upper bound. The matrix indices correspond to the values in the column labeled as “ID” in Table 7.1.

have presented an ecosystem of a factorization-based block-Jacobi preconditioner which succeeds in reducing the time-to-solution of the iterative IDR(4) Krylov method for a large range of problems.

Future work will address the development of a Cholesky-based variant for symmetric positive definite problems and the optimization of the batched kernels for any problem size.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC-0010042. G. Flegar and E. S. Quintana-Ortí were supported by project TIN2014-53495-R of the MINECO and FEDER. The authors would also like to thank the Swiss National Computing Centre (CSCS) for granting computing resources in the Small Development Project entitled “Energy-Efficient preconditioning for iterative linear solvers” (d65).

Matrix	size	#nnz	ID	Jacobi		Block-Jacobi (8)		Block-Jacobi (12)		Block-Jacobi (16)		Block-Jacobi (24)		Block-Jacobi (32)	
				#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]
ABACUS_shell_ud	23,412	218,484	7	2074	3.38	2086	3.65	1717	3.07	1958	3.47	1889	3.48	1560	2.96
af_shell3	504,855	17,562,051	14	1443	8.91	1117	8.85	1148	9.09	953	7.66	968	7.95	1084	8.77
bcsstk17	10,974	428,650	13	1584	2.69	1524	2.73	797	1.50	736	1.38	612	1.13	927	1.79
bcsstk18	11,948	149,090	10	781	1.27	562	1.02	377	0.69	529	0.97	351	0.69	455	0.89
bcsstk38	8,032	355,460	48	—	—	—	—	—	—	7722	15.45	2875	5.86	2101	4.44
cbuckle	13,681	676,515	24	212	0.43	96	0.24	19	0.04	69	0.16	18	0.05	70	0.19
Chebyshev2	2,053	18,447	6	—	—	175	0.52	57	0.18	45	0.13	34	0.10	28	0.08
Chebyshev3	4,101	36,879	37	—	—	897	3.37	327	1.26	133	0.53	88	0.34	79	0.31
crankseg_1	52,804	10,614,210	36	180	0.61	125	0.48	123	0.47	144	0.54	126	0.48	126	0.51
CurlCurl_0	11,083	113,343	45	95	0.16	64	0.12	57	0.12	51	0.11	49	0.10	45	0.10
CurlCurl_1	226,451	2,472,071	35	262	0.73	191	0.72	182	0.67	132	0.50	117	0.45	106	0.43
dc3	116,835	766,396	8	183	18.92	116	12.10	142	14.76	131	13.68	109	11.36	168	17.52
dw1024	2,048	10,114	39	—	—	133	0.23	136	0.28	128	0.27	72	0.15	51	0.10
dw2048	2,048	10,114	25	—	—	133	0.22	136	0.22	128	0.25	72	0.15	51	0.09
dw4096	8,192	41,746	41	—	—	—	—	—	—	5035	7.94	1565	2.57	879	1.53
dw8192	8,192	41,746	40	—	—	—	—	—	—	5035	7.93	1565	2.57	879	1.56
ecology2	999,999	4,995,991	11	—	—	2826	27.58	3919	38.06	3539	34.92	3358	34.56	2990	30.17
F1	343,791	26,837,113	9	2906	19.17	1944	15.26	1422	11.16	1930	15.20	2065	16.63	1688	13.52
G3_circuit	1,585,478	7,660,826	30	1030	9.47	1079	15.82	1105	16.12	1070	15.85	1260	19.50	847	12.93
gas_sensor	66,917	1,703,365	4	—	—	—	—	—	—	—	—	—	—	9480	22.33
gridgena	48,962	512,084	42	1322	2.37	669	1.37	791	1.62	657	1.34	730	1.52	637	1.35
Hook_1498	1,498,023	59,374,451	38	7215	120.43	5017	109.84	2731	59.64	5358	117.96	3486	78.96	3747	83.84
ibm_matrix_2	51,448	537,038	21	—	—	—	—	211	0.54	—	—	300	0.76	—	—
Kuu	7,102	340,200	15	101	0.19	66	0.15	67	0.13	71	0.22	66	0.15	61	0.13
LeGresley_2508	2,508	16,727	16	266	0.49	189	0.36	166	0.30	171	0.34	172	0.30	151	0.32
linverse	11,999	95,977	43	2069	3.16	—	—	—	—	5981	10.19	2010	3.56	964	1.80
matrix_9	103,430	1,205,518	3	1425	2.96	—	—	469	1.19	—	—	508	1.34	—	—
matrix-new_3	125,329	893,984	26	—	—	—	—	408	1.09	—	—	452	1.24	—	—
ML_Geer	1,504,002	110,686,677	1	2092	44.06	1844	48.63	1917	50.43	1700	45.09	2111	57.35	1621	43.58
nasa2910	2,910	174,296	46	486	0.83	575	0.93	386	0.67	340	0.62	440	0.74	405	0.71
nd12k	36,000	14,220,946	5	—	—	3902	13.38	2863	9.81	2191	7.60	1487	5.21	1476	5.23
nd24k	72,000	28,715,634	31	—	—	3970	21.01	2542	13.46	2129	11.31	1691	9.04	1516	8.12
nd3k	9,000	3,279,690	34	—	—	—	—	3875	8.08	2354	4.94	2066	4.45	3145	6.87
nd6k	18,000	6,897,316	33	—	—	—	—	4173	10.39	3635	9.08	1945	5.01	2585	6.80
nemeth15	9,506	539,802	32	47	0.09	—	—	1298	2.28	392	0.73	280	0.53	156	0.31
olm5000	5,000	19,996	47	—	—	224	0.46	256	0.48	119	0.23	114	0.24	153	0.26
Pres_Poisson	14,822	715,804	2	177	0.32	136	0.29	97	0.19	88	0.21	81	0.17	73	0.26
rail_79841	79,841	553,921	19	915	1.70	987	2.20	825	1.91	658	1.55	801	1.88	593	1.34
s1rmt3m1	5,489	217,651	18	206	0.33	166	0.32	134	0.28	140	0.27	134	0.28	130	0.32
s2rmq4m1	5,489	263,351	12	722	1.13	190	0.35	197	0.37	200	0.38	188	0.38	172	0.39
s2rmt3m1	5,489	217,681	20	1131	1.75	268	0.52	217	0.40	206	0.38	200	0.40	180	0.36
s3rmq4m1	5,489	262,943	17	—	—	595	1.03	693	1.26	494	0.86	491	0.90	501	0.95
s3rmt3m1	5,489	217,669	29	—	—	1303	2.07	769	1.29	733	1.24	472	0.96	831	1.58
s3rmt3m3	5,357	207,123	22	—	—	—	—	—	—	—	—	9676	16.40	—	—
saylr4	3,564	22,316	27	1118	1.77	249	0.48	182	0.34	171	0.32	160	0.28	143	0.30
ship_003	121,728	3,777,036	23	—	—	2731	8.12	2488	7.37	3042	9.19	1739	5.38	1343	4.18
sme3Dc	42,930	3,148,656	28	3311	7.87	2945	7.58	3178	8.19	2397	6.22	2578	6.79	3113	8.32
sts4098	4,098	72,356	44	108	0.23	77	0.17	83	0.16	60	0.14	60	0.12	50	0.10

Table 7.1: Iterations and execution time of IDR(4) enhanced with scalar Jacobi preconditioning or block-Jacobi preconditioning. The runtime combines the preconditioner setup time and the iterative solver execution time.

Bibliography

- [1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 3rd edition, 1999.
- [2] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler. Efficiency of general Krylov methods on GPUs — an experimental study. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW'16*, pages 683–691. IEEE, 2016.
- [3] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'17*, pages 1–10. ACM, 2017.
- [4] H. Anzt, J. Dongarra, G. Flegar, E. S. Quintana-Ortí, and A. E. Tomás. Variable-size batched Gauss-Huard for block-Jacobi preconditioning. In *Procedia Computer Science*, volume 108, pages 1783–1792. Elsevier, 2017.
- [5] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, C. S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.
- [6] E. Chow, H. Anzt, J. Scott, and J. Dongarra. Using Jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning. *Journal of Parallel and Distributed Computing*, 119:219–230, 2018.
- [7] CUDA Toolkit v8.0. 2017.
- [8] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.
- [9] T. J. Dekker, W. Hoffmann, and K. Potma. Stability of the Gauss-Huard algorithm with partial pivoting. *Computing*, 58(3):225–244, 1997.
- [10] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.
- [11] P. Huard. La méthode simplex sans inverse explicite. *E.D.F. Bull. de la Direction des Études et des Recherches*, 2(2):79–98, 1979.
- [12] MAGMA 2.0.0. <http://icl.cs.utk.edu/magma/>, 2016.
- [13] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.

Part IV

Towards Adaptive Precision Methods

Leveraging Adaptive Precision in Block-Jacobi Preconditioning

Published as: H. Anzt, J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí. Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers. *Concurrency and Computation: Practice and Experience*, 31(6):e4460, 2019

We propose an adaptive scheme to reduce communication overhead caused by data movement by selectively storing the diagonal blocks of a block Jacobi preconditioner in different precision formats (half, single, or double). This specialized preconditioner can then be combined with any Krylov subspace method for the solution of sparse linear systems to perform all arithmetic in double precision. We assess the effects of the adaptive precision preconditioner on the iteration count and data transfer cost of a preconditioned conjugate gradient solver. A preconditioned conjugate gradient method is, in general, a memory bandwidth-bound algorithm, and therefore its execution time and energy consumption are largely dominated by the costs of accessing the problem's data in memory. Given this observation, we propose a model that quantifies the time and energy savings of our approach based on the assumption that these two costs depend linearly on the bit length of a floating point number. Furthermore, we use a number of test problems from the SuiteSparse matrix collection to estimate the potential benefits of the adaptive block-Jacobi preconditioning scheme.

8.1 Introduction

Krylov subspace-based iterative methods for the solution of sparse linear systems typically benefit from the integration of a preconditioner that improves the conditioning of the linear system and, consequently, accelerates the convergence process [20].

A popular preconditioner is the Jacobi preconditioner and its block-Jacobi variants. Preconditioners of this class are based on simple (block-)diagonal scaling, which makes them highly parallel schemes suitable for fine-grained parallelism, and they have proven to provide a fair acceleration for many applications. For example, block-Jacobi preconditioners can efficiently exploit the massive hardware concurrency of graphics processing units (GPUs) [1, 2].

For virtually all current hardware technologies, the computational performance of preconditioned Krylov methods is limited by the memory bandwidth and depends heavily on the cost of memory access. Furthermore, for current architectures, data movement is not just a performance constraint but also a major source of energy consumption. Therefore, with highly parallel, high-performance computing (HPC) systems moving in the direction of an increasing floating point operations (FLOP) per byte ratio, innovative techniques to reduce communication and data transfers are critical for future applications [10, 12, 17, 18].

When a block-Jacobi preconditioner is combined with a simple Krylov iterative method—like the preconditioned conjugate gradient (PCG) method, which is suitable for the solution of sparse linear systems with a symmetric positive-definite (SPD) coefficient matrix [20]—a significant portion of the accesses to main memory is caused by the application of the preconditioner at each iteration. To decrease the costs of this stage, we analyze a version of the block-Jacobi preconditioner that selectively stores part of its data in low precision. This strategy reduces the data access volume during the application of the block-Jacobi preconditioner. We emphasize that, for a memory bandwidth-bound operation such as the PCG method, the time and energy savings

of operating with reduced precision mostly come from the reduction of the volume of data being transferred, not from the increase in the single instruction, multiple data (SIMD) capacity associated with using reduced precision arithmetic. Therefore, our solution aims to reduce the cost of main memory data transfers due to the preconditioner application only. All other data (including the sparse matrix entries) as well as all arithmetic occurs in the conventional double precision. In more detail, our work makes the following contributions:

- We propose an adaptive preconditioner that *stores* the diagonal Jacobi blocks in the preconditioner using half, single, or double precision, depending on the conditioning and data range. In our scheme, the preconditioner blocks are retrieved from memory in the corresponding format and transformed into double precision once in the processor registers; all arithmetic operations are then performed at double precision level. As stated earlier, the entries for the sparse matrix and recurrence vectors for the conjugate gradient (CG) method (or any other Krylov subspace method) are maintained and retrieved in main memory using standard double precision.
- We investigate the impact that storing a block-Jacobi preconditioner in low precision exerts on the PCG convergence rate and the effectiveness of the adaptive precision block-Jacobi at maintaining the reference convergence rate.
- We develop a model that quantifies the runtime and energy savings based on the assumption that these costs depend linearly on the bit length of a floating point number.
- We use a set of test problems from the SuiteSparse matrix collection [9] to analyze the robustness of the adaptive preconditioning in a CG method, and to estimate the potential energy savings.

The use of mixed precision in preconditioned iterative solvers was previously explored with a primary focus on reducing the cost of arithmetic operations. In [4], Arioli and Duff show that, when using a lower-upper (LU) preconditioner computed in single precision within a flexible generalized minimal residual method (GMRES) based iterative solver (which enables one to use a non-constant preconditioning operator), backward stability at double precision can be preserved even for ill-conditioned systems. In [6], Carson and Higham provide a detailed error analysis of LU-based mixed refinement approaches for ill-conditioned systems. In [7], the same authors go as far as using half precision for computing an LU preconditioner that is used in the solution process of a GMRES solver that is part of a mixed precision iterative refinement process.

Our approach is fundamentally different. We do not aim to employ reduced precision in the generation or application of the preconditioner nor in any other arithmetical computation. Instead, we preserve full precision in all computations but store part of the preconditioner at a reduced precision. After reading the preconditioner stored at reduced precision, all data is converted to full precision before proceeding with the arithmetic operations in the actual preconditioner application. We argue that this approach has significantly higher potential for runtime- and energy savings than the previously proposed strategies for three reasons: (1) since the performance of sparse linear algebra algorithms is typically memory bound, the performance benefit obtained by reducing the data access volume is greater than the benefit obtained by reducing the cost of FLOPs; (2) since the energy cost of data access is more than an order of magnitude greater than that of arithmetic operations [21], more resources can be saved by reducing data accesses; and (3) running the preconditioner application at reduced precision results in a preconditioning operator not preserving orthogonality in double precision, implying that previously orthogonal Krylov vectors may not be orthogonal after the preconditioner application. To account for this situation, flexible variants that introduce an additional orthogonalization step are required to preserve convergence [14]. Performing the arithmetic operations in the distinct preconditioner applications in full precision (even though the preconditioner data is stored at reduced precision) preserves the orthogonality of the Krylov subspace and removes the burden of expensive reorthogonalization. Hence, in our approach we do not need to employ a flexible Krylov solver.

Section 8.2 provides a some background on the need for reorthogonalization when using non-constant preconditioning. We also discuss how our strategy using full precision in the arithmetic operations results in a constant preconditioner, which avoids the need for a flexible Krylov method. A brief recap/overview of block-Jacobi preconditioning is provided in Section 8.3. In Section 8.4, we introduce the concept of adaptive precision preconditioning, and we introduce the evaluation criteria for selecting the storage format of the distinct diagonal blocks. Rounding error analysis to support the criteria is given in Section 8.5. We report the experimental results

```

 $A \rightarrow M$ 
Initialize  $x_0, p_0, r_0 := b - Ax_0, \tau_0 := \|r_0\|_2, \gamma_0$ 
 $k := 0$ 
while ( $\tau_k > \tau_{\max}$ )
     $q_{k+1} := Ap_k$ 
     $\eta_k := p_k^T q_{k+1}$ 
     $\alpha_k := \gamma_k / \eta_k$ 
     $x_{k+1} := x_k + \alpha_k p_k$ 
     $r_{k+1} := r_k - \alpha_k q_{k+1}$ 
     $\tau_{k+1} := \|r_{k+1}\|_2$ 
     $z_{k+1} := M^{-1} r_{k+1}$ 
     $\gamma_{k+1} := r_{k+1}^T z_{k+1}$ 
     $\beta_{k+1} := \gamma_{k+1} / \gamma_k$ 
     $p_{k+1} := z_{k+1} + \beta_{k+1} p_k$ 
     $k := k + 1$ 
endwhile

```

Figure 8.1: Mathematical formulation of the PCG method. Here, $\tau_{\max}$ is the relative residual stopping criterion.

in Section 8.6, which includes an analysis of “reckless” precision reduction in block-Jacobi preconditioning, the assessment of the evaluation criteria, and an energy consumption model that quantifies the savings owed to adaptive precision preconditioning. We summarize our findings in Section 8.7 and provide details about the path forward for this research.

8.2 Reduced Precision Preconditioning in the PCG Method

8.2.1 Brief review

Figure 8.1 shows the PCG method for the solution of the linear system $Ax = b$; where the coefficient matrix, $A \in \mathbb{R}^{n \times n}$, is SPD and sparse with n_z nonzero entries; $b \in \mathbb{R}^n$ is the right-hand side; and $x \in \mathbb{R}^n$ is the sought-after solution. The most challenging operations in this algorithm are the computation of the preconditioner (before the iteration commences), the computation of the sparse matrix-vector product (SpMV) (at each iteration), and the preconditioner application (at each iteration). The remaining operations are scalar computations or simple vector kernels like the dot product (dot) and axpy-type vector updates [5].

In the PCG method, the dot operations present a one-to-one ratio of FLOPs to memory accesses (MEMOPS), and the axpy-type operations present a two-to-three ratio of FLOPs to MEMOPS, which clearly identifies these operations as memory bandwidth-bound kernels. For simplicity, moving forward we make no distinction between the cost of reading a number and the cost of writing a number. Assuming the sparse matrix is stored in compressed sparse row (CSR) format [20]—and is using 64 bits for double precision numbers/values (fp64) and 32 bits for integers/indices (int32)—the ratio of FLOPs:MEMOPS for SpMV is $2n_z / ((n + n_z) \cdot \text{fp64} + (n + n_z) \cdot \text{int32})$. As a consequence, this operation is also memory bounded. An analysis of the operations using the preconditioner is provided later in this section.

8.2.2 Orthogonality-Preserving Mixed Precision Preconditioning

In general, using a reduced precision preconditioner (i.e., 32-bit or 16-bit arithmetic) instead of “full” 64-bit, double precision arithmetic requires a careful consideration of the numerical effects. In this section we discuss how our preconditioning strategy results in a constant preconditioning operator. This preserves the orthogonality of the generated Krylov search directions, and therefore allows us to employ the standard CG solver based on the Fletcher-Reeves orthogonalization instead of the flexible CG based on the Polak-Ribière formula.

The PCG method presented in Figure 8.1 assumes that the preconditioner is a constant operator acting on the input vector, $r = r_{k+1}$, as $z = z_{k+1} := M^{-1} r$ [20]. In this case, $r_k^T z_{k+1} = 0$, that is to say, the orthogonality with respect to the previous residual is preserved. Strictly speaking, even when using double precision, the preconditioner application introduces some rounding error so that the computed operator satisfies $z = M^{-1} r +$

```

1 % PCG method
2 % Compute preconditioner A -> M
3 % Initialize x, p, r = b - A * x, tau = r' * r, gamma_old
4 while (tau > tau_max)
5     z = M \ r; % = M^{-1} r, apply preconditioner
6
7     gamma_new = r' * z; % DOT product
8
9     beta = gamma_new / gamma_old; % scalar operation
10    p = z + beta * p; % AXPY-type
11    q = A * p; % SpMV
12    eta = p' * q; % DOT product
13    alpha = gamma_new / eta; % scalar operation
14    gamma_old = gamma_new; % scalar operation
15    x = x + alpha * p; % AXPY
16    r = r - alpha * q; % AXPY
17    tau = r' * r; % = ||r||_2, DOT product
18 end

```

Figure 8.2: Algorithmic formulation (in MATLAB) of the PCG method. For a problem of size n containing n_z nonzero elements in the system matrix stored in CSR format, ignoring the preconditioner application, each PCG iteration requires $(14n + n_z) \cdot \text{fp64} + (n_z + n) \cdot \text{int32}$ memory transactions.

$\mathcal{O}(\varepsilon_d)$, where ε_d stands for fp64 machine precision. Hence, a preconditioner in double precision can also have an impact on the orthogonality. However, as the effects are in the order of the approximation accuracy, the non-consistency of the preconditioning operator is typically disregarded in practice.

In contrast, when applying a preconditioner in less than double precision, this issue becomes more relevant, because the rounding error now grows to $z = M^{-1}r + \mathcal{O}(\varepsilon_r)$, where ε_r is the machine precision of the reduced format. As a result, the orthogonality error increases to ε_r , which becomes relevant if convergence beyond ε_r is desired.

A straightforward workaround is to introduce an additional orthogonalization step to account for the loss in orthogonality. Concretely, replacing the Fletcher-Reeves formula from Figure 8.1,

$$\beta_k := \gamma_{k+1} / \gamma_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}, \quad (8.1)$$

with the Polak-Ribière formula,

$$\beta_k := \frac{(r_{k+1} - r_k)^T z_{k+1}}{r_k^T z_k}, \quad (8.2)$$

naturally accounts for z_{k+1} losing orthogonality with respect to the previous residual, r_k [14]. Compared with the original formulation of the CG method, this orthogonality-preserving “flexible CG” (FCG) [14] incurs an overhead that corresponds to keeping the last residual vector in memory and computing an additional vector operation and dot product. The benefits are that the iterative method can handle a flexible (non-constant) preconditioner [19], which is needed when applying a preconditioner in reduced precision.

Obviously, with a constant preconditioner, $r_k^T z_{k+1} = 0$, i.e. both formulas (8.1) and (8.2) are identical. For $r_k^T z_{k+1} \neq 0$, the Polak-Ribière formula specifies a locally optimal search direction, which means that the convergence rate of this method will not be slower than that of a locally optimal steepest descent method [16]. We complement the preconditioned CG method, based on the Fletcher-Reeves formula shown in Figure 8.2, with the flexible conjugate gradient (FCG) method based on the Polak-Ribière formula in Figure 8.3. The two codes differ only in lines 6–8 (computation of `gamma_new` and additional recurrence for vector `t`), which results in $7n$ additional memory accesses. A faster preconditioner application (i.e., using reduced precision arithmetic operations in the actual preconditioner application) could barely compensate for this overhead.

In our approach, we store the preconditioner at reduced precision, but we convert the data to double precision right after reading it from memory and before invoking the arithmetic computations of the preconditioner application. Hence, although stored at a reduced precision, the preconditioner itself remains constant across

```

1 % Flexible CG method
2 % Compute preconditioner A -> M
3 % Initialize x, p, r = b - A * x, r_old, tau = r' * r, gamma_old
4 while (tau > tau_max)
5     z = M \ r; % = M^{-1} r, apply preconditioner
6     gamma_new = r' * z; % DOT product
7     t = r - r_old; % AXPY-type
8     gamma_t = t' * z; % DOT product
9     r_old = r; % COPY
10    beta = gamma_t / gamma_old; % scalar operation
11    p = z + beta * p; % AXPY-type
12    q = A * p; % SpMV
13    eta = p' * q; % DOT product
14    alpha = gamma_new / eta; % scalar operation
15    gamma_old = gamma_new; % scalar operation
16    x = x + alpha * p; % AXPY
17    r = r - alpha * q; % AXPY
18    tau = r' * r; % = ||r||_2, DOT product
19 end

```

Figure 8.3: Algorithmic formulation (in MATLAB) of the FCG method. For a problem of size n containing n_z nonzero elements in the system matrix stored in CSR format, ignoring the preconditioner application, each FCG iteration requires $(21n + n_z) \cdot \text{fp64} + (n_z + n) \cdot \text{int32}$ memory transactions.

all iterations. This strategy does introduce some overhead in terms of converting the preconditioner data to double precision and using double precision in all arithmetic operations, but it comes with the benefit of using the Fletcher-Reeves formula (8.1) for the orthogonalization step, which results in the more attractive (in terms of memory) standard PCG solver.

8.3 Block-Jacobi Preconditioning

The Jacobi method splits the coefficient matrix as $A = L + D + U$, with a diagonal matrix $D = (\{a_{ii}\})$, a lower triangular factor $L = (\{a_{ij} : i > j\})$, and an upper triangular factor $U = (\{a_{ij} : i < j\})$. The block-Jacobi variant is an extension that gathers the diagonal blocks of A into $D = (D_1, D_2, \dots, D_N)$, with $D_i \in \mathbb{R}^{m_i \times m_i}$, $i = 1, 2, \dots, N$, and $n = \sum_{i=1}^N m_i$. The remaining elements of A are then partitioned into matrices L and U such that L contains the elements below the diagonal blocks in D , while U contains those above them [1]. The block-Jacobi method is well defined if all diagonal blocks are nonsingular. The resulting preconditioner, $M = D$, is particularly effective if the blocks succeed in reflecting the nonzero structure of the coefficient matrix, A . Fortunately, this is the case for many linear systems that, for example, exhibit some inherent block structure because they arise from a finite element discretization of a partial differential equation (PDE) [1].

There are several strategies to integrate a block-Jacobi preconditioner into an iterative solver like CG. In this work, we adopt an approach that explicitly computes the block-inverse matrix, $D^{-1} = (D_1^{-1}, D_2^{-1}, \dots, D_N^{-1}) = (E_1, E_2, \dots, E_N)$, before the iterative solution process commences, and then applies the preconditioner in terms of a dense matrix-vector multiplication (GeMV) per inverse block E_i [3]. Note that GeMV is still a memory bandwidth-bound kernel, independent of the block size. In practice, this strategy shows numerical stability similar to the conventional alternative that computes the LU factorization (with partial pivoting) [13] of each block ($D_i = L_i U_i$) and then applies the preconditioner using two triangular solvers (per factorized block). By comparison, the GeMV kernel is highly parallel, while the triangular solves offer only limited parallelism.

8.4 Adaptive Precision Block-Jacobi Preconditioning

The main goal of this work is to assess the potential benefits of a specialized version of a block-Jacobi preconditioner that selectively stores part of its data at low precision—a technique that reduces the memory access volume during the application of a block-Jacobi preconditioner. Concretely, we employ three precision formats: (1) 16-bit, half precision arithmetic (fp16); (2) 32-bit, single precision arithmetic (fp32); and (3) 64-bit,

(full) double precision arithmetic (fp64). The fp32 and fp64 roughly correspond to the two IEEE standards that are currently supported by practically all commodity processors used in everything from desktop systems to high-performance servers. On the other hand, fp16 has only recently received considerable attention because of its usefulness in deep learning applications, and hardware support for this format is now included in the most recent many-core architectures from NVIDIA.

For our experiments, we use a PCG Krylov solver to expose the effects of storing parts of a block-inverse preconditioner at a reduced precision. Before we introduce our preconditioning scheme and the strategy for selecting the appropriate storage format, we note that, for the type of systems that can be tackled using a CG method, the diagonal blocks of A in the preconditioner D are all symmetric. Therefore, a significant amount of storage (and data transfer cost) can already be saved by explicitly storing only the lower or upper triangular part of each block. We also recognize that some computational cost can be saved by exploiting the symmetry and positive definiteness information of these diagonal blocks. However, as these two cost-saving techniques are orthogonal to those we propose, we refrain from mixing the distinct strategies.

In general, the design of a block-Jacobi preconditioner with adaptive precision is based on the following observations.

1. In the preconditioner matrix, D , each one of the blocks, D_i , is independent.
2. Except for cases where the iterative solver converges quickly, the overhead incurred by determining an appropriate storage format for the preconditioner (before the iteration commences) is irrelevant.
3. The application of each block, D_i , (i.e., multiplication with the inverse block, E_i) should be done with care to guarantee “enough” precision in the result. As we will show in Section 8.5, the accuracy of this application is largely determined by the condition number of D_i with respect to inversion, denoted hereafter as $\kappa_1(D_i) = \|D_i\|_1 \|D_i^{-1}\|_1 = \|D_i\|_1 \|E_i\|_1$, [13].

Armed with these observations, we propose the following adaptive precision block-Jacobi preconditioner:

1. Before the iteration commences, the inverse of each block, D_i , is computed explicitly using fp64: $D_i \rightarrow E_i$. We note that even if D_i is sparse, its inverse, E_i , is likely a dense matrix. For this reason, we store the inverse, E_i , following the conventional column-major order using $m_i \times m_i$ real numbers.
2. At the same stage (i.e., before the iteration), we compute $\kappa_1(D_i) = \kappa_1(E_i) = \|D_i\|_1 \|E_i\|_1$ and we note that, given E_i is explicitly available, computing $\kappa_1(D_i)$ is straightforward and inexpensive compared with the inversion of the block.
3. In principle, we store E_i , which was computed in fp64, in the format determined by its condition number—truncating the entries of the block if necessary—as:

$$\begin{cases} \text{fp16} & \text{if } \tau_h^L < \kappa_1(D_i) \leq \tau_h^U, \\ \text{fp32} & \text{if } \tau_s^L < \kappa_1(D_i) \leq \tau_s^U, \text{ and} \\ \text{fp64} & \text{otherwise,} \end{cases} \quad (8.3)$$

with $\tau_h^L = 0$ and $\tau_h^U = \tau_s^L$. As we will discuss in Section 8.5, the values for the bounds τ_h^U and τ_s^U are selected by taking into account the unit roundoff for each format: $u_h \approx 4.88e-04$ for half precision, $u_s \approx 5.96e-08$ for single precision, and $u_d \approx 1.11e-16$ for double precision.

4. During the iteration, we recover the block E_i stored in the corresponding format in memory, transform its entries to fp64 once in the processor registers, and apply it in terms of a fp64 GeMV to the appropriate entries of r_{k+1} to produce those of z_{k+1} . This is a memory bandwidth-bound operation, and, therefore, its cost is dominated by the overhead of recovering the data for the preconditioner matrix and the vectors from memory (i.e., MEMOPS). Thus, we can expect that in practice the FLOPs will be completely “amortized” (i.e., overlapped) with the data transfers.

The truncation procedure for converting fp64 data to a reduced precision format requires some care to deal with overflows/underflows and their consequences, as described below.

Figure 8.4: Control flow for deciding whether or not to select a reduced format.

```

1  function [Ei, success] = truncate_format(Ei, Di_cond_num1, ...
2                                tau_r_L, tau_r_U, tau_k)
3  %
4  % Inputs : mi x mi dense inverse block Ei;
5  %          condition number of Di (and Ei) in Di_cond_num1; and
6  %          thresholds to determine use of reduced format:
7  %          - tau_r_L and tau_r_U (with tau_r=tau_h or tau_s), and
8  %          - tau_k
9  % Output : mi x mi dense inverse block Ei
10 %          overwritten with the reduced format if applicable
11 %
12 success = 0;                                % FALSE
13 if (tau_r_L < Di_cond_num1) & (Di_cond_num1 <= tau_r_U)
14     Ei_reduced = force_reduction(Ei); % Truncate to reduced format
15     Ei_reduced_norm1 = norm(Ei_reduced, 1);
16     if (Ei_reduced_norm1 > 0.0)                % Ei contains nonzero entries
17         % Compute the condition number of truncated block via explicit
18         % inverse: easier to implement on GPU than SVD
19         Ei_reduced_cond_num1 = Ei_reduced_norm1 * norm(inv(Ei_reduced), 1);
20         if (Ei_reduced_cond_num1 < tau_k)        % Ei is not (close to) singular
21             Ei = Ei_reduced;
22             success = 1;                        %TRUE
23     end
24 end
25 end
26 %
27 return;

```

Figure 8.5: Details of the procedure for deciding whether or not to select a reduced format.

- The truncation of a “large” (in magnitude) value in E_i , represented in fp64, can produce an overflow because the number is too large to be represented in the reduced format, resulting in an “Inf” value in that format. In those cases, we can either discard the use of the reduced format for the complete block E_i or replace the truncated value with the largest number (in magnitude) representable in that format (e.g., for positive values, 65,504 in fp16 and about $3.40e + 38$ in fp32).
- Conversely, the truncation of a “small” (in magnitude) value, in fp64, may yield an underflow that returns a value that is zero. This can turn a nonsingular matrix E_i into a singular matrix. For example, if all entries of E_i are below the minimum representable number in the reduced format, the result of truncation will produce a block that comprises only zeros, and the preconditioned solver will not converge. This could be mitigated to some extent by scaling all the values of the block. Furthermore, even if some of the entries are nonzero the truncated representation of E_i may still become ill-conditioned, thereby causing numerical difficulties for the convergence. In order to avoid this issue, we propose checking the condition number of the truncated representation and not using the corresponding reduced precision if it is above the relevant threshold, τ_k .

Figure 8.4 summarizes the global precision selection process, and the pseudocode in Figure 8.5 provides a practical implementation of the truncation procedure and the various thresholds—taking E_i and $\kappa_1(E_i)$ as inputs. The routine given in the pseudocode, `force_reduction`, simply truncates the fp64 block to a reduced

format. The rest of the code uses several metrics to determine whether the use of the reduced format is safe.

8.5 Rounding Error Analysis

As previously elaborated, we invert the diagonal blocks explicitly using double precision, e.g. via (batched) Gauss-Jordan Elimination [1, 3]. Let $E_i = D_i^{-1}$ be the inverse of block i computed in double precision arithmetic with unit roundoff u_d . By storing the inverse in reduced precision ($\widehat{E}_i$) with unit roundoff u , we introduce the error ΔE_i and get [11], [15, secs. 14.3, 14.4]

$$\widehat{E}_i = E_i + \Delta E_i, \quad \|\Delta E_i\|_1 \leq c_{m_i} \kappa_1(D_i) \|\widehat{E}_i\|_1 u_d + u \|\widehat{E}_i\|_1, \quad (8.4)$$

for some constant c_{m_i} . For the vector segments in z_i and r_i corresponding to the diagonal block i , the subsequent multiplication in double precision results in [15, sec. 3.5]

$$\widehat{z}_i = fl(\widehat{E}_i r_i) = \widehat{E}_i r_i + \Delta z_i, \quad \|\Delta z_i\|_1 \leq c'_{m_i} u_d \|\widehat{E}_i\|_1 \|r_i\|_1. \quad (8.5)$$

Hence

$$\widehat{z}_i = (E_i + \Delta E_i) r_i + \Delta z_i = E_i r_i + \overline{\Delta z}_i, \quad (8.6)$$

where combining (8.4) and (8.5) gives

$$\begin{aligned} \|\overline{\Delta z}_i\|_1 &= \|\Delta E_i r_i + \Delta z_i\|_1 \\ &\leq c_{m_i} \kappa_1(D_i) \|\widehat{E}_i\|_1 \|r_i\|_1 u_d + u \|\widehat{E}_i\|_1 \|r_i\|_1 + c'_{m_i} u_d \|\widehat{E}_i\|_1 \|r_i\|_1 \\ &= (c_{m_i} \kappa_1(D_i) u_d + u + c'_{m_i} u_d) \|\widehat{E}_i\|_1 \|r_i\|_1. \end{aligned} \quad (8.7)$$

We may assume that the constant term $c'_{m_i} u_d$ becomes negligible when storing the diagonal block in the reduced precision format with unit roundoff $u \gg u_d$. With this assumption,

$$\|\overline{\Delta z}_i\|_1 \leq c_{m_i} (\kappa_1(D_i) u_d + u) \|\widehat{E}_i\|_1 \|r_i\|_1. \quad (8.8)$$

Noting that $r_i = E_i^{-1} z_i = D_i z_i$, this bound yields

$$\begin{aligned} \|\overline{\Delta z}_i\|_1 &\leq c_{m_i} (\kappa_1(D_i) u_d + u) \|\widehat{E}_i\|_1 \|D_i\|_1 \|z_i\|_1 \\ &\approx c_{m_i} (\kappa_1(D_i) u_d + u) \kappa_1(D_i) \|z_i\|_1, \end{aligned} \quad (8.9)$$

so that

$$\frac{\|\overline{\Delta z}_i\|_1}{\|z_i\|_1} \leq c_{m_i} (\kappa_1(D_i) u_d + u) \kappa_1(D_i). \quad (8.10)$$

As expected, the relative error depends on the conditioning of the diagonal block D_i . With the unit roundoff being a format-specific constant ($u_h \approx 4.88e-04$ for half precision, $u_s \approx 5.96e-08$ for single precision, and $u_d \approx 1.11e-16$ for double precision), (8.10) provides bounds for the relative error.

Recalling that we are within a preconditioner framework, by ignoring all entries outside the block-diagonal in the inversion process we may have already introduced a significant error. In fact, experiments reveal that preconditioners based on block-Jacobi often come with an error as large as $1.0e-2$ to $1.0e-1$. This makes it reasonable to allow for similar errors in (8.10), which yields the bounds for the condition numbers that are allowed in the respective formats. In the experimental section we use the bounds $\tau_h^U = \tau_s^L := 1.0e+2$, and $\tau_s^U := 1.0e+6$.

Figure 8.6: Boxplot for the distribution of the condition numbers of the diagonal blocks ($\kappa_1(D_i)$) using super-variable agglomeration with the block size set to 24. For each matrix, the blue central box shows where most of the condition numbers are located, the red crosses indicate outliers.

8.6 Experimental Analysis

8.6.1 Experimental framework

In this section, we assess the potential benefits of the adaptive precision block–Jacobi preconditioner with a collection of experiments performed in GNU Octave version 3.8.1. We implement the PCG method according to [14] (Figure 8.2) with an integrated block–Jacobi preconditioner that performs an explicit inversion of the diagonal blocks. We apply supervariable agglomeration to optimize the block diagonal structure of the block–Jacobi preconditioner for the specific problems used here [8]. This procedure aims to identify and capture the block structure of the matrix in the Jacobi blocks of the preconditioner, thereby accumulating multiple blocks into a larger superstructure with the upper bound of the blocksize set to 24.

For the evaluation, we consider a subset comprised of 63 SPD test problems of small to moderate dimension from the SuiteSparse matrix collection [9]. We list the matrices along with some key characteristics in Table 8.1.

In the adaptive precision preconditioner, we use the evaluation strategy shown in Figures 8.4 and 8.5 to determine the precision at which the individual diagonal blocks should be stored. According to the heuristics presented in Section 8.5, we set $\tau_h^L := 0$, $\tau_h^U = \tau_s^L := 1.0e + 2$, $\tau_s^U := 1.0e + 6$; and $\tau_\kappa := 1.0e - 3/u_d$; see also (8.3). Specifying an upper block size bound of 24 in the supervariable agglomeration, we show in Figure 8.6 the condition number distribution of the blocks for each test matrix. These condition numbers are one of the metrics considered when selecting the storage format in the adaptive precision block–Jacobi preconditioner.

Using Octave, we emulate the procedure for truncation of fp64 values to reduced precision formats (`force_reduction` shown in Figure 8.5) as follows. First, we transform the full precision value to a text string and then truncate that string to keep only the two most significant decimal digits for fp16 and the seven most significant decimal digits for fp32. This is a rough approximation of the precision level that can be maintained with the bits dedicated to the mantissa in the IEEE standards for fp16/fp32. To emulate overflow, we set values exceeding the data range of the low precision format to the largest representable number in the target format, $R_{\max}$, which is $R_{\max} = 65,504$ for fp16 and $R_{\max} = 3.40e + 38$ for fp32. We preserve the sign in this truncation process. To emulate underflow, values that are smaller than the minimum value that can be represented in the low precision format, $R_{\min}$, are set to zero, which is $R_{\min} = 6.10e - 5$ for fp16 and $R_{\min} = 1.17e - 38$ for fp32. We stop the PCG iterations once the relative residual has dropped below the threshold $\tau_{\max} := 1.0e - 9$. We allow for, at most, 5,000 PCG iterations.

8.6.2 Reduced precision preconditioning

In the first experiment, we investigate how a reckless/adaptive reduction of the precision for the representation of the block–Jacobi preconditioner impacts the convergence rate of the PCG iterative solver. By recklessly reducing the precision format used for storing the block–diagonal inverse, essential information may be lost, which slows down the convergence of the iterative solver. In the worst case, the diagonal blocks may become singular, or the entries may fall outside of the data range that can be represented in the chosen precision; both

ID	Matrix	# rows	# nonzeros	cond. number	#PCG iterations			
					double	single	half	adaptive
1	1138_bus	1,138	4,054	1.2100e+07	784	778	–	782
2	494_bus	494	1,666	3.8900e+06	269	269	271	269
3	662_bus	662	2,474	8.2100e+05	179	179	179	179
4	685_bus	685	3,249	4.0500e+05	171	171	–	172
5	bcsstk01	48	400	1.6000e+06	35	34	–	34
6	bcsstk03	112	640	6.2700e+06	46	48	–	48
7	bcsstk04	132	3,648	5.5500e+06	72	72	–	72
8	bcsstk05	153	2,423	3.5300e+04	95	95	–	95
9	bcsstk06	420	7,860	1.1900e+07	255	254	–	254
10	bcsstk07	420	7,860	1.1900e+07	255	254	–	254
11	bcsstk08	1,074	12,960	2.3200e+06	231	231	–	231
12	bcsstk09	1,083	18,437	3.6000e+03	325	325	–	325
13	bcsstk10	1,086	22,070	1.3200e+06	517	517	–	517
14	bcsstk11	1,473	34,241	4.2100e+06	768	764	–	764
15	bcsstk12	1,473	34,241	2.9000e+06	768	764	–	764
16	bcsstk13	2,003	83,883	5.6400e+08	1,639	1,631	–	1,444
17	bcsstk14	1,806	63,454	1.3100e+10	276	276	–	276
18	bcsstk15	3,948	117,816	1.9800e+07	585	584	–	583
19	bcsstk16	4,884	290,378	7.0100e+09	263	261	–	263
20	bcsstk19	817	6,853	5.8600e+10	1,775	1,773	–	1,768
21	bcsstk20	485	3,135	7.4800e+12	2,125	2,113	–	2,114
22	bcsstk21	3,600	26,600	2.6000e+06	565	565	–	565
23	bcsstk22	138	696	2.7600e+04	75	75	–	75
24	bcsstk24	3,562	159,910	7.1800e+10	2,505	2,630	–	2,336
25	bcsstk26	1,922	30,336	8.0800e+06	1,979	1,957	–	1,957
26	bcsstk27	1,224	56,126	1.4900e+04	213	213	–	213
27	bcsstk28	4,410	219,024	6.2800e+09	2,182	2,115	–	2,115
28	bcsstm07	420	7,252	1.3400e+04	46	46	46	46
29	bcsstm12	1,473	19,659	8.8800e+05	26	26	1,220	26
30	lund_a	147	2,449	9.8900e+05	89	90	–	90
31	lund_b	147	2,441	6.0300e+04	47	47	48	47
32	nos1	237	1,017	7.5900e+06	157	165	–	165
33	nos2	957	4,137	1.8300e+09	2,418	2,409	–	2,409
34	nos3	960	15,844	7.3500e+04	137	137	137	137
35	nos4	100	594	2.7000e+03	46	46	47	47
36	nos5	468	5,172	3.5900e+03	235	235	–	235
37	nos6	675	3,255	8.0000e+06	77	77	–	77
38	nos7	729	4,617	4.0000e+09	68	68	–	68
39	plat1919	1,919	32,399	2.2200e+18	4,117	4,049	3,772	4,081
40	plat362	362	5,786	7.0800e+11	982	1,112	1,115	1,095
41	mhdb416	416	2,312	5.0500e+09	19	19	–	19
42	bcsstk34	588	21,418	2.6700e+04	185	185	–	185
43	msc00726	726	34,518	8.5500e+05	160	160	–	160
44	msc01050	1,050	26,198	9.0000e+15	1,594	1,593	–	1,593
45	msc01440	1,440	44,998	7.0000e+06	929	928	–	928
46	msc04515	4,515	97,707	4.7800e+05	2,348	2,349	–	2,349
47	ex5	27	279	1.3200e+08	10	25	–	10
48	nasa1824	1,824	39,208	2.3100e+05	896	896	–	896
49	nasa2146	2,146	72,250	2.8100e+03	352	353	–	353
50	nasa2910	2,910	174,296	1.3000e+06	1,369	1,369	–	1,369
51	nasa4704	4,704	104,756	6.4500e+06	4,171	4,123	–	4,123
52	mesh1e1	48	306	8.2000e+00	14	14	14	14
53	mesh1em1	48	306	3.4000e+01	23	23	23	23
54	mesh1em6	48	306	8.8500e+00	14	14	15	15
55	mesh2e1	306	2,018	4.0700e+02	79	79	83	83
56	mesh2em5	306	2,018	2.7900e+02	77	77	81	75
57	mesh3e1	289	1,377	9.0000e+00	18	18	18	18
58	mesh3em5	289	1,377	5.0000e+00	17	17	17	17
59	sts4098	4,098	72,356	3.5600e+07	342	342	–	340
60	Chem97ZtZ	2,541	7,361	3.2900e+02	30	30	30	30
61	mhd3200b	3,200	18,316	2.0200e+13	17	17	–	17
62	mhd4800b	4,800	27,520	1.0300e+14	16	16	–	16
63	plbuckle	1,282	30,644	2.9200e+05	260	260	–	260

Table 8.1: Left: Test matrices along with key properties. Right: Iteration count of the PCG method with the preconditioner stored in double, single, half, or adaptive precision. The “–” symbol indicates cases where the iterative solver did not reach the relative residual threshold $\tau_{\max} = 1.0e - 9$ after 5,000 iterations.

cases would likely result in the algorithm's breakdown. We emphasize that the distinct preconditioners only differ in the format that is leveraged to store the block inverse. Conversely, the problem-specific diagonal block pattern is not affected, and all computations are realized in fp64.

The three leftmost columns in the right part of Table 8.1 report the iterations required for convergence of the PCG method when storing the block-inverse preconditioner in fp64, fp32, or fp16. We observe that storing the preconditioner in fp32 usually has only a mild impact on the preconditioner quality. In most cases, the PCG iteration count matches the one where the preconditioner is stored in fp64. In a few cases, the PCG converges even faster when storing the preconditioner in fp32. Conversely, if the preconditioner is stored in fp16, the PCG does not converge in most cases. Therefore, fp16 storage cannot be recommended as the default choice. In the right-most column of Table 8.1, we report the iteration count for the PCG method preconditioned with adaptive precision block-Jacobi. We observe that, except for some noise, the adaptive precision block-Jacobi preserves the quality of the preconditioner and the convergence rate of the fp64 solver. Figure 8.7 shows that most of the time the adaptively chosen precision is single or half precision, with relatively few instances of double.

8.6.3 Energy model

Having validated that the adaptive precision block-Jacobi preconditioner preserves the convergence rate of the iterative solver, we next quantify the advantage of the adaptive precision block-Jacobi over a standard block-Jacobi using double precision. For this purpose, we specifically focus on the energy efficiency, as this has been identified as an important metric (on par with performance) for future exascale systems.

In terms of energy consumption, the accesses to main memory (MEMOPS) are at least an order of magnitude more expensive than FLOPs, and this gap is expected to increase in future systems [21]. For this reason, in the energy model, we ignore the arithmetic operations (including the access to the data in the processor registers as well as caches) and consider the data transfers from main memory only. Our energy model for estimating the global energy consumption of the solver builds on the premise that the energy cost of memory accesses is linearly dependent on the bit length of the data.

Furthermore, as we only aim to estimate the energy efficiency of the adaptive precision block-Jacobi preconditioner *relative* to the standard fp64 block-Jacobi preconditioner, we will set the (normalized) energy cost of accessing a single bit of data as 1 (energy-unit). The precision formats we consider employ 64, 32, and 16 bits.

For a problem of size n with n_z nonzero elements, the PCG method presented in Section 8.2 and preconditioned with a block-Jacobi preconditioner (consisting of N diagonal blocks of dimensions $m_1 \times m_1, \dots, m_N \times m_N$) performs:

$$\underbrace{14n \cdot \text{fp64}}_{\text{vector memory transfers}} + \underbrace{(2n + n_z) \cdot \text{fp64} + (n + n_z) \cdot \text{int32}}_{\text{CSR-SpMV memory transfers}} + \underbrace{2n \cdot \text{fp64} + \sum_{i=1}^N m_i^2 \cdot \text{fp}x_i}_{\text{preconditioner memory transfers}} \quad (8.11)$$

data transfers (from memory) per iteration, where $\text{fp}x_i$ denotes the precision format selected for the i -th diagonal block of the preconditioner. The data transfer volume of the block-Jacobi preconditioner thus depends on the format employed to store the block inverse. For example, with the PCG running in fp64, the approach also employs fp64 to maintain the block-Jacobi preconditioner. Further, we also consider variants that store the preconditioner entirely in fp32 or fp16 and a more sophisticated strategy that adapts the format of the distinct preconditioner blocks to the data.

For the adaptive precision block-Jacobi approach, we visualize the use of fp64, fp32, and fp16 for storing the diagonal blocks (Figure 8.7). Comparing this information with the data in Figure 8.6, we can identify a relationship between the conditioning of the blocks and the storage precision format: fp64 is primarily employed for those cases comprising very ill-conditioned blocks. Furthermore, the information in Figure 8.7 also shows the savings that can be attained in terms of (1) memory usage to store the preconditioner and (2) data transfers per iteration to retrieve data from main memory. However, note that these savings do not take into account the total cost of the PCG method but only those costs strictly associated with the preconditioner application. Furthermore, the data in Figure 8.7 does not reflect the potentially slower convergence caused by using reduced precision storage.

Figure 8.7: Details on the adaptive precision block-Jacobi. Breakdown of the diagonal blocks stored in fp64, fp32, or fp16.

Figure 8.8: Energy efficiency analysis of the PCG with block-Jacobi preconditioning using different floating point formats for storing the preconditioner. The energy cost of all methods is normalized to the energy cost of the standard implementation using fp64 for storing the block-Jacobi preconditioner.

To avoid the previous two pitfalls, in our final experiment we compute the total data transfers of a single iteration of the PCG method with the block-Jacobi preconditioner stored in fp64, fp32, fp16, or adaptive precision, see equation (8.11). To obtain an estimated total data transfer volume, we then combine the data transfer volume per iteration with the number of iterations needed to reach convergence in each case—ignoring those cases for which half precision does not converge.

In Figure 8.8, we show the total energy balance *relative* to the standard approach that maintains the block-Jacobi preconditioner in double precision.

Some key observations from this last experiment are listed below.

- Storing the block-Jacobi preconditioner in fp32 often reduces the total energy cost. However, for those cases where the information loss increases the PCG iteration count, storing the preconditioner in fp32 can have a negative impact on the energy balance.
- For the (few) cases where the block-inverse matrix can be stored in fp16 without the PCG losing convergence, the total energy cost can be decreased by up to 50%.
- Using the adaptive precision block-Jacobi preconditioner never increases the total energy consumption.
- In most cases, the adaptive precision block-Jacobi preconditioner matches or outperforms the efficiency of storing the preconditioner in fp32. If the problem characteristics allow for it, the adaptive precision block-Jacobi preconditioner employs fp16 to match the half precision efficiency while maintaining convergence for the other cases.
- The adaptive precision block-Jacobi preconditioner “automatically” detects the need to store a diagonal block in fp64 to avoid convergence degradation.

Finally, we note that, for memory bandwidth-bound operations like the block-Jacobi preconditioned CG considered here, the performance is largely determined by the data transfer volume. Therefore, the results shown in Figure 8.8 and the insights gained from that experiment carry over to the runtime performance of the adaptive precision block-Jacobi preconditioner. In summary, these experiments prove that the adaptive precision block-Jacobi preconditioner is an efficient strategy for improving the resource usage, energy consumption, and runtime performance of iterative solvers for sparse linear systems.

8.7 Concluding Remarks and Future Work

We proposed and validated a strategy to reduce the data transfer volume in a block-Jacobi preconditioner. Concretely, our technique individually selects an appropriate precision format to store the distinct blocks of the preconditioner based on their characteristics but performs all arithmetic (including the generation of the preconditioner) in fp64. We note that the condition numbers can be obtained cheaply as our preconditioner is based on explicit inversion of the diagonal blocks. Furthermore, the overhead from selecting the appropriate storage format in the preconditioner setup can easily be amortized by the reduced cost of the preconditioner application in the solver iterations.

Our experimental simulation using Octave on an Intel architecture shows that, in most cases, storing a block-Jacobi preconditioner in fp32 has only a mild impact on the preconditioner quality. On the other hand, the reckless use of fp16 to store a block-Jacobi preconditioner fails in most cases and is therefore not recommended. The adaptive precision block-Jacobi preconditioner basically matches the convergence rate of the conventional double precision preconditioner in all cases and automatically adapts the precision to be used on an individual basis. As a result, the adaptive precision preconditioner can decide to store some of the blocks at precisions even less than fp32, thereby outperforming a fixed precision strategy that relies on only a single precision in terms of data transfers and, consequently, energy consumption.

As part of our future work, we plan to investigate the effect of using other, non IEEE-compliant data formats in the adaptive block-Jacobi preconditioner, prioritizing the exponent range at the cost of reducing the bits dedicated to the mantissa. In this endeavor, we expect to reduce the problems with underflows/overflows while maintaining the “balancing” properties of the preconditioner. Furthermore, we will also develop a practical implementation of the adaptive precision block-Jacobi using IEEE formats with 16, 32, and 64 bits for modern GPUs.

Acknowledgements

We thank Matthias Bollhöfer for fruitful discussions on flexible variants of Krylov solvers allowing for non-constant preconditioning operators and for pointing us to the flexible version of CG in [19]. H. Anzt was supported by the “Impuls und Vernetzungsfond of the Helmholtz Association” under grant VH-NG-1241. G. Flegar and E. S. Quintana-Ortí were supported by project TIN2014-53495-R of the MINECO and FEDER and the H2020 EU FETHPC Project 732631 “OPRECOMP”. N. Higham was supported by MathWorks and by Engineering and Physical Sciences Research Council grant EP/P020720/1. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Bibliography

- [1] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM’17, pages 1–10. ACM, 2017.
- [2] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. Variable-size batched LU for small matrices and its integration into block-Jacobi preconditioning. In *Proceedings of the 46th International Conference on Parallel Processing*, ICPP 2017, pages 91–100. IEEE, 2017.

- [3] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. Variable-size batched Gauss-Jordan elimination for block-Jacobi preconditioning on graphics processors. *Parallel Computing*, 81:131–146, 2019.
- [4] M. Arioli and I. S. Duff. Using FGMRES to obtain backward stability in mixed precision. *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, 33:31–44, 2008.
- [5] R. Barrett, M. Berry, T. F. 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*. SIAM, 2nd edition, 1994.
- [6] E. Carson and N. J. Higham. A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM Journal on Scientific Computing*, 39(6):A2834–A2856, 2017.
- [7] E. Carson and N. J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing*, 40:A817–A847, 2018.
- [8] E. Chow, H. Anzt, J. Scott, and J. Dongarra. Using Jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning. *Journal of Parallel and Distributed Computing*, 119:219–230, 2018.
- [9] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.
- [10] J. Dongarra, J. Hittinger, J. Bell, L. Chacon, R. Falgout, M. Heroux, P. Hovland, E. Ng, C. Webster, and S. Wild. Applied mathematics research for exascale computing. Technical report, U.S. Department of Energy, 2014.
- [11] J. J. Du Croz and N. J. Higham. Stability of methods for matrix inversion. *IMA Journal of Numerical Analysis*, 12(1):1–19, 1992.
- [12] M. Duranton, K. De Bosschere, A. Cohen, J. Maebe, and H. Munk. HiPEAC vision 2015. Technical report, HiPEAC, 2015.
- [13] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.
- [14] G. H. Golub and Q. Ya. Inexact preconditioned conjugate gradient method with inner-outer iteration. *SIAM Journal on Scientific Computing*, 21(4):1305–1320, 1999.
- [15] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002.
- [16] A. V. Knyazev and I. Lashuk. Steepest descent and conjugate gradient methods with variable preconditioning. *SIAM Journal on Matrix Analysis and Applications*, 29(4):1267–1280, 2008.
- [17] J. F. Lavignon, D. Lecomber, I. Phillips, F. Subirada, F. Bodin, J. Gonnord, S. Bassini, G. Tecchiolli, G. Lonsdale, A. Pflieger, B. Andrietti, T. Lippert, A. Bode, H. Falter, P. Blouet, and M. M. ETP4HPC strategic research agenda achieving HPC leadership in Europe. Technical report, European Technology Platform for High Performance Computing, 2013.
- [18] R. Lucas, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, A. Geist, G. Grider, R. Haring, H. J., A. Hoisie, D. Klein, P. Kogge, R. Lethin, V. Sarkar, R. Schreiber, J. Shalf, T. Sterling, and R. Stevens. Top ten exascale research challenges. Technical report, U.S. Department of Energy, 2014.
- [19] Y. Notay. Flexible conjugate gradients. *SIAM Journal on Scientific Computing*, 22(4):1444–1460, 2000.
- [20] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.
- [21] J. Shalf. The evolution of programming models in response to energy efficiency constraints. http://www.oscer.ou.edu/Symposium2013/oksupercompsymp2013_talk_shalf_20131002.pdf, 2013. Slides presented at Oklahoma Supercomputing Symposium 2013.

Part V

Epilogue

9.1 Designing Scientific Software for Sparse Computations

With the previous chapters of this dissertation focusing on the components for the solution of linear systems, the natural next consideration is the integration of these components into a holistic software ecosystem for the solution of linear systems. A central challenge in this context is the manifoldness that arises from combining different components. Each matrix format can be used with any Krylov method. On top of that, each combination can be enhanced with any preconditioner, which could be available directly as a matrix provided by the user (stored in one of the formats supported by the software, or even in a custom format), generated from the system matrix as one of the standard preconditioners (*e.g.*, relaxation or factorization based), or even implemented as a coarse (Krylov) solver. The system matrix may not even be stored explicitly, but available as a specialized implementation provided by the user. The preconditioners themselves may also be complex methods for the solution of linear systems, as even the most common ILU-based preconditioners can be constructed via selecting components from a pool of factorization methods that generate a preconditioner, and a pool of linear system solution methods that are used to solve systems with the upper and lower triangular factors arising from the incomplete factorization. Another aspect increasing the complexity of software design is the heterogeneity of modern hardware. If the goal of the software is to support computation on various devices, or even collaborative computation on a heterogeneous platform, its design has to incorporate abstractions that allow reasoning about the underlying hardware.

While there are various high performance libraries that try to address these aspects [23, 24, 28], they usually struggle with at least one of them. This section proposes one possible theoretical design that aims at solving these issues and briefly describes a new library that is based on it.

9.1.1 Matrices

A central concept for reducing complexity in both mathematics and software engineering is abstraction. By hiding all the unnecessary details of a mathematical or software object, and only representing it as a concept with certain operations that behave according to some rules, every concrete object that defines these operations and follows the same rules can be handled in the exact same way. In the context of Krylov methods, matrices are used only as part of the matrix-vector product operation. In rare cases, the conjugate transpose matrix (*adjoint matrix*) is also needed as part of the conjugate transpose matrix-vector product operation. For a fixed matrix A , the key properties of the matrix-vector product are *additivity* $A(x+y) = Ax + Ay$ and *homogeneity* $A(\alpha x) = \alpha Ax$ — which are often written combined into the equivalent *linearity* property $A(\alpha x + \beta y) = \alpha Ax + \beta Ay$. Focusing only on these properties allows us to define an abstraction that can hide the details about the coefficients and the storage scheme of the matrix.

Theorem 9.1. *For any matrix $A \in \mathbb{F}^{m \times n}$, let $L_A : \mathbb{F}^n \rightarrow \mathbb{F}^m$ be an operator that satisfies $L_A x = Ax$, $x \in \mathbb{F}^n$. The following holds:*

1. L_A is a linear operator, i.e. $L_A(\alpha x + \beta y) = \alpha L_A x + \beta L_A y$, for any $\alpha, \beta \in \mathbb{F}$ and $x, y \in \mathbb{F}^n$.
2. L_A is unique, i.e., if $L_A \neq L_B$ then $A \neq B$, for any $A, B \in \mathbb{F}^{m \times n}$.

3. The mapping $A \mapsto L_A$ is injective, i.e., if $L_A = L_B$ then $A = B$, for any $A, B \in \mathbb{F}^{m \times n}$.

4. The mapping $A \mapsto L_A$ is surjective onto the set of linear operators

$$\mathcal{L}(\mathbb{F}^n, \mathbb{F}^m) := \{L : \mathbb{F}^n \rightarrow \mathbb{F}^m \mid L(\alpha x + \beta y) = \alpha Lx + \beta Ly, \forall \alpha, \beta \in \mathbb{F}, \forall x, y \in \mathbb{F}^n\}, \quad (9.1)$$

i.e., for every $L \in \mathcal{L}(\mathbb{F}^n, \mathbb{F}^m)$ there is a matrix $A \in \mathbb{F}^{m \times n}$ such that $L = L_A$.

Thus, $A \mapsto L_A$ defines a one-to-one correspondence between the set of matrices $\mathbb{F}^{m \times n}$ and linear operators $\mathcal{L}(\mathbb{F}^n, \mathbb{F}^m)$.

Proof.

1. This is a trivial consequence of the definition of L_A : $L_A(\alpha x + \beta y) = A(\alpha x + \beta y) = \alpha Ax + \beta Ay = \alpha L_A x + \beta L_A y$.
2. Assume $L_A \neq L_B$. Then, there exists a vector x such that $Ax = L_A x \neq L_B x = Bx$. In consequence, $(A - B)x \neq 0$. This is only possible if $A - B \neq 0$, i.e. $A \neq B$.
3. For $i = 1, 2, \dots, n$, let a_i and b_i denote the i -th column of matrices A and B , respectively, and let $e_i \in \mathbb{F}^n$ be the i -th vector of the standard basis for $\mathbb{F}^n$, i.e., the i -th component of e_i is 1, while its every other component is 0. Then, $a_i = Ae_i = L_A e_i = L_B e_i = Be_i = b_i$, for every column i , which implies $A = B$.
4. For any linear operator $L \in \mathcal{L}(\mathbb{F}^n, \mathbb{F}^m)$ we construct the matrix $A \in \mathbb{F}^{m \times n}$ by defining its i -th column as $a_i := Le_i$. Then, for any vector $x = [x_1, x_2, \dots, x_n]^T \in \mathbb{F}^n$, the following holds:

$$\begin{aligned} L_A x &= Ax = A\left(\sum_{i=1}^n x_i e_i\right) = \sum_{i=1}^n x_i A e_i \\ &= \sum_{i=1}^n x_i a_i = \sum_{i=1}^n x_i L e_i = L\left(\sum_{i=1}^n x_i e_i\right) \\ &= Lx, \end{aligned} \quad (9.2)$$

generating the equality $L_A = L$.

□

Definition 9.2. For any matrix $A \in \mathbb{F}^{m \times n}$, the unique linear operator $L_A \in \mathcal{L}(\mathbb{F}^n, \mathbb{F}^m)$ defined by $L_A x := Ax$ for $x \in \mathbb{F}^n$ is called the *matrix product operator* with respect to A .

Theorem 9.1 shows that every matrix can be uniquely represented by its matrix product operator, and that a specific matrix can be reconstructed given its linear operator representation. However, to effectively replace matrices with linear operators, the structure of algebraic operations defined on matrices has to be mirrored by linear operators.

Definition 9.3. Let $L \in \mathcal{L}(\mathbb{F}^n, \mathbb{F}^m)$ be a linear operator and let $(\cdot, \cdot)_m : \mathbb{F}^m \times \mathbb{F}^m \rightarrow \mathbb{F}$ and $(\cdot, \cdot)_n : \mathbb{F}^n \times \mathbb{F}^n \rightarrow \mathbb{F}$ be any two inner products [21] on vector spaces $\mathbb{F}^m$ and $\mathbb{F}^n$, respectively. A linear operator $K \in \mathcal{L}(\mathbb{F}^m, \mathbb{F}^n)$ that satisfies

$$(Lx, y)_m = (x, Ky)_n \quad (9.3)$$

for every $x \in \mathbb{F}^n$ and $y \in \mathbb{F}^m$ is called an *adjoint operator* of L with respect to inner products $(\cdot, \cdot)_m$ and $(\cdot, \cdot)_n$, and is denoted by L^* .

Theorem 9.4. For any matrix $A \in \mathbb{F}^{m \times n}$, the adjoint operator L_A^* of the matrix product operator L_A with respect to inner products $(x, y)_m := x^* y$, $\forall x, y \in \mathbb{F}^m$ and $(w, z)_n := w^* z$, $\forall w, z \in \mathbb{F}^n$ is unique, and $L_A^* = L_{A^*}$.

Proof. Let K be any adjoint operator for L_A . Then, applying Equation (9.3) on $e_i \in \mathbb{F}^n$ and $e_j \in \mathbb{F}^m$ yields:

$$\begin{aligned} (Ke_j)_i &= e_i^* Ke_j = (e_i, Ke_j)_n = (L_A e_i, e_j)_m \\ &= (Ae_i, e_j)_m = (a_i, e_j)_m = a_i^* e_j = \overline{a_{ji}} \\ &= (A^*)_{ij} \end{aligned} \quad (9.4)$$

The equality $K = L_{A^*}$ is obtained by applying Equation (9.2) to L_{A^*} and K .

□

Corollary 9.5. *The mapping $A \mapsto L_A$ is an isomorphism between the set of matrices $F^{m \times n}$ and linear operators $\mathcal{L}(\mathbb{F}^n, \mathbb{F}^m)$ that preserves the algebraic structure of the matrix-vector multiplication and conjugate transpose operations, that is:*

1. *The matrix-vector multiplication is equivalent to the application of the matrix product operator: $Ax = L_A x$, for $x \in \mathbb{F}^n$.*
2. *The conjugate transpose matrix-vector multiplication is equivalent to the application of the matrix product operator's adjoint: $A^*x = L_A^*x$, for $x \in \mathbb{F}^n$.*

Proof. Relation 1 is true by the definition of operator L_A . Relation 2 is a direct consequence of Relation 1 and Theorem 9.4. $\square$

9.1.2 Linear Systems

Another central operation that a library focused on Krylov solvers has to provide is, obviously, the solution of a linear system. Interestingly, this operation satisfies the same additivity and homogeneity properties as the matrix-vector multiplication.

Definition 9.6. For any nonsingular matrix $A \in GL_n(\mathbb{F}) := \{M \in \mathbb{F}^{n \times n} \mid \det(M) \neq 0\}$, let $S_A : \mathbb{F}^n \rightarrow \mathbb{F}^n$ be an operator that satisfies $A(S_A b) = b$, for $b \in \mathbb{F}^n$. That is, the operator S_A maps any (right-hand side) vector b to the solution x of the linear system $Ax = b$ induced by the system matrix A and the right-hand side b . Such an operator is called a *solver operator* with respect to A .

Theorem 9.7. *For any $A \in GL_n(\mathbb{F})$:*

1. *S_A is a linear operator, i.e. $S_A(\alpha x + \beta y) = \alpha S_A x + \beta S_A y$, for any $\alpha, \beta \in \mathbb{F}$ and $x, y \in \mathbb{F}^n$.*
2. *S_A is invertible and $S_A^{-1} = L_A$.*
3. *S_A is unique, i.e., if $S_A \neq S_B$ then $A \neq B$, for any $A, B \in GL_n(\mathbb{F})$.*
4. *The mapping $A \mapsto S_A$ is injective, i.e., if $S_A = S_B$ then $A = B$, for any $A, B \in GL_n(\mathbb{F})$.*
5. *The mapping $A \mapsto S_A$ is surjective onto the set*

$$\text{Aut}(\mathbb{F}^n) := \{L \in \mathcal{L}(\mathbb{F}^n, \mathbb{F}^n) \mid L \text{ is invertible}\}, \quad (9.5)$$

i.e., for every $L \in \text{Aut}(\mathbb{F}^n)$ there is a matrix $A \in GL_n(\mathbb{F})$ such that $L = S_A$.

6. *The mapping $\Sigma : \text{Aut}(\mathbb{F}^n) \rightarrow \text{Aut}(\mathbb{F}^n)$, $\Sigma(L_A) := S_A$ is a one-to-one correspondence on the set of invertible linear operators.*

Proof.

1. Due to the equality $A(\alpha S_A x + \beta S_A y) = \alpha A S_A x + \beta A S_A y = \alpha x + \beta y$, $\alpha S_A x + \beta S_A y$ is the unique solution of the system $Aw = \alpha x + \beta y$. From Definition 9.6 it follows that $S_A(\alpha x + \beta y) = \alpha S_A x + \beta S_A y$.
2. Since $L_A S_A x = A S_A x = x$ for $x \in \mathbb{F}^n$, L_A is the inverse for S_A .
3. If $S_A \neq S_B$ then there exists $x \in \mathbb{F}^n$ such that $S_A x \neq S_B x$. Since A is nonsingular, multiplying the inequality with A results in $A S_B x \neq x$. This means that $A \neq B$, as otherwise $x = A S_A x = A S_B x \neq x$.
4. $S_A = S_B$ implies that for each column a_j of A , $A S_B a_j = A S_A a_j = a_j$. Since e_j is the unique solution of $Ax = a_j$ this means that $S_B a_j = e_j$. Multiplying the final equality by B yields $b_j = B e_j = B S_B a_j = a_j$, resulting in $A = B$.
5. Every $L \in \text{Aut}(\mathbb{F}^n)$ has an inverse linear operator L^{-1} . For L and L^{-1} denote by B and A the matrices for which $L = L_B$ and $L^{-1} = L_A$. Since $ABx = L_A L_B x = L^{-1} L x = x$ for every $x \in \mathbb{F}^n$, it follows that $AB = I$ and $\det(A) \det(B) = \det(AB) = \det(I) = 1$. Thus, $\det(A) \neq 0$ and $A \in GL_n(\mathbb{F})$. Since $A(Lx) = L_A(Lx) = L^{-1}(Lx) = x$ for $x \in \mathbb{F}^n$, L is the solver operator for A , i.e. $L = S_A$.

6. The matrix B in (5) is also nonsingular since $\det(A)\det(B) = 1$ implies $\det(B) \neq 0$. This means that the restriction of the injective mapping $A \mapsto L_A$ to the set $GL_n(\mathbb{F})$ is also surjective onto the set $\text{Aut}(\mathbb{F}^n)$. Thus, it establishes a one-to-one correspondence between the sets $GL_n(\mathbb{F})$ and $\text{Aut}(\mathbb{F}^n)$. The mapping Σ is a composition of two bijective mappings $L_A \mapsto A$ and $A \mapsto S_A$, so it is itself a one-to-one correspondence.

□

Theorem 9.7 shows that there is a correspondence between a set of linear systems with a fixed nonsingular matrix (which can be represented as a class of linear systems generated by that matrix) and invertible operators. In addition, since Theorem 9.1 established a link between matrices and linear operators, the matrix can be replaced by its corresponding linear operator, resulting in a well-defined permutation Σ on linear operators, which maps a linear operator L to a linear operator that represents the solution of a system generated by L .

Definition 9.8. In the context of linear algebra software design, we call the higher-order permutation $\Sigma : \text{Aut}(\mathbb{F}^n) \rightarrow \text{Aut}(\mathbb{F}^n)$ defined with $\Sigma(L_A) := S_A$, for $L_A \in \text{Aut}(\mathbb{F}^n)$ the *solver factory* on $\text{Aut}(\mathbb{F}^n)$.

The final piece of the puzzle required to replace the concept of linear systems with linear operators when designing the library is showing that the algebraic structure of operations on linear systems is preserved when transforming them into linear operators.

Corollary 9.9. Let $\Psi : (Ax = \cdot) \mapsto L_A$ be the mapping between linear systems with a fixed matrix A and matrix product operators with respect to A (this is a bijection due to Theorem 9.1). Then, the mapping $\Sigma \circ \Psi$ is an isomorphism between the set of linear systems with a fixed matrix $A \in GL_n(\mathbb{F})$ and invertible linear operators $\text{Aut}(\mathbb{F}^n)$ that preserves the algebraic structure of the conjugate transpose operation and the operation of solving a linear system, that is:

1. Solving a linear system with a fixed matrix A is equivalent to the application of the solver operator S_A ; i.e., $Ax = b$ if and only if $x = S_A b$, for $x, b \in \mathbb{F}^n$.
2. Solving a linear system with the conjugate transpose of a matrix A is equivalent to the application of the solver operator's adjoint S_A^* , i.e.; $A^*x = b$ if and only if $x = S_A^* b$, for $x, b \in \mathbb{F}^n$.

Proof. (1) follows from the construction of S_A . From Theorem 9.7, $S_A = (S_A^{-1})^{-1} = (L_A)^{-1} = L_{A^{-1}}$, which implies $S_A^* = L_{A^{-1}}^* = L_{(A^{-1})^*} = L_{(A^*)^{-1}} = L_{A^*}^{-1} = (S_{A^*}^{-1})^{-1} = S_{A^*}$. (2) is then obtained by substituting A with A^* in (1) and using $S_{A^*} = S_A^*$. □

9.1.3 Preconditioners

Preconditioning (i.e., replacing the original linear system $Ax = b$ with a preconditioned system, e.g. $M^{-1}Ax = M^{-1}b$ for right preconditioning, where M is some approximation of A) is another central concept in the iterative solution of linear systems. While the preconditioner M is often represented by a matrix, the only operations where the preconditioner appears are its application to a vector $y = M^{-1}x$, and sometimes the application of its conjugate transpose to a vector $y = (M^{-1})^*x$. The preconditioner matrix M is associated to the original system matrix A via the method Π that was used to derive the preconditioner from the system matrix A . A preconditioner needs to be nonsingular to preserve the existence and uniqueness of the solution to the linear system. These considerations motivate the following definition:

Definition 9.10. Any mapping $\Pi : \text{Aut}(\mathbb{F}^n) \rightarrow \text{Aut}(\mathbb{F}^n)$ is called a *preconditioner factory*. For any operator $L_A \in \text{Aut}(\mathbb{F}^n)$ (or equivalently, matrix $A \in GL_n(\mathbb{F})$), the linear operator $P_A^\Pi := \Pi(L_A)$ is called the *preconditioner operator* with respect to the operator L_A (or equivalently, A) and the preconditioner factory Π . The unique matrix $M \in GL_n(\mathbb{F})$ such that $P_A^\Pi = L_{M^{-1}}$ is called the *preconditioner matrix* with respect to A and Π .

An isomorphism $M \mapsto P_A^\Pi$ between preconditioner matrices and preconditioner operators that preserves the algebraic structure of preconditioner application and conjugate transpose preconditioner application is established via Corollary 9.5. Thus, preconditioners easily join the family of linear algebra operations that can be represented as linear operators.

9.1.4 Linear Operators — Towards a Generic Interface for Sparse Computations

The previous sections elaborated on how to represent the major components of an iterative solution framework — matrices and matrix-vector product, preconditioners and preconditioner application, and linear systems and their solution — as linear operators. Using this abstraction in the design of a library effectively solves the problem of various component combinations for the solution of linear systems. Every component is implemented in terms of the linear operator abstraction (*e.g.*, a Krylov method uses the abstract interface of the linear operator to realize matrix-vector products and preconditioner applications), and the decision of which concrete linear operators should be used for the components is left to the top-level application. The ability to substitute an abstract linear operator by a concrete implementation allows users to express any of the complex component combinations presented at the beginning of this chapter. Another advantage is that support for application-specific matrix formats, preconditioners, or methods for the solution of linear systems can be enabled by simply leaving the concept of the linear operator open for extension. This allows users to provide their own implementations of components that interoperate with the rest of the library in the same way as built-in components do.

While the linear operator abstraction is perfectly suited for software that computes linear algebra operations symbolically, numerical libraries need to adopt an approximation for these abstractions to work. The usual argument against the linear operator abstraction is the fact that a fixed number of iterations of a Krylov method does not represent a linear operator. Indeed, for any Krylov method, the linear combination $\alpha x_k + \beta y_k$ of k -th approximations x_k and y_k for the systems $Ax = b$ and $Ay = c$ is generally not equal to the k -th approximation z_k for the system $Az = \alpha b + \beta c$. This is not the only “flaw” that can be found in a concrete implementation of the linear operator abstraction. In fact, in a numerical setting with limited precision, none of the linear operators can be realized accurately. A natural implementation of the matrix product operator is a dense or sparse matrix-vector multiplication, using floating point arithmetic. Thus, these computations will contain rounding errors and the computed outputs $\tilde{x}$, $\tilde{y}$ and $\tilde{z}$ for operations $x = Ab$, $y = Ac$ and $z = A(\alpha b + \beta c)$, respectively, will not satisfy $\tilde{z} = \alpha \tilde{x} + \beta \tilde{y}$. The same aspects arise for the preconditioner application. In consequence, the only way to obtain accurate matrix product and preconditioner operators is to use infinite precision arithmetic, which is usually too expensive and not necessary. In the case of Krylov methods, which are used to compute the application of the solver operator S_A , the accurate solution that satisfies the additivity and homogeneity properties could be obtained by using infinite precision arithmetic and running the method until full convergence is reached.

The short answer to the observation about Krylov methods is that the method is not intended to be used as a linear operator S_A , but as a way of approximating the effect of applying the linear operator S_A to a vector. In more detail, the main distinction between the solver operator (which is linear) and any of the Krylov methods (which are strictly speaking not linear operators) is that the former is used to describe what is being computed, and the latter how is it being computed. Assuming there is an implicit consensus that every operation in the software library comes with some approximation error — which is a common concept in computational numerical linear algebra — the linear operator abstraction continues to be valid.

Definition 9.11. Let $L \in \mathcal{L}(\mathbb{F}^n, \mathbb{F}^m)$ be any linear operator and $E : \mathbb{F}^n \rightarrow \mathbb{F}^m$ be any (non-linear) function. The function $\tilde{L} : \mathbb{F}^n \rightarrow \mathbb{F}^m$, $\tilde{L}(x) := Lx + E(x)$ is called an *approximate linear operator* for L with approximation error E .

Strictly speaking, the approximate linear operator does not satisfy the linearity property. However, it does satisfy a relaxed version of linearity: $\tilde{L}(\alpha x + \beta y) = \alpha \tilde{L}(x) + \beta \tilde{L}(y) + (E(\alpha x + \beta y) - \alpha E(x) - \beta E(y))$. Thus, the smaller the approximation error, the closer is the approximate linear operator to fulfilling the defining characteristics of a linear operator. From the numerical linear algebra software design perspective, all concrete implementations of linear operators are approximate linear operators. However, this fact is only revealed during the construction of the approximate linear operator, where the user can sometimes influence the magnitude of the approximation error E . Once the approximate linear operator is constructed, it is treated as a linear operator, since exposing the unnecessary details about the approximation error would only complicate the design without offering any practical advantages.

9.1.5 Ginkgo — A High Performance Linear Operator Library

An example of a software ecosystem building upon the concept of linear operators is the Ginkgo open source library [17]. The library was developed to overcome the problems with composability, heterogeneous hardware

```

1  #include <ginkgo/ginkgo.hpp>
2  #include <iostream>
3
4  int main() {
5      auto gpu = gko::CudaExecutor::create(0, gko::OmpExecutor::create());
6      auto A = gko::read<gko::matrix::Csr<>>(std::cin, gpu);
7      auto b = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
8      auto x = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
9      auto solver =
10         gko::solver::Cg<>::build()
11         .with_preconditioner(gko::preconditioner::Jacobi<>::build().on(gpu))
12         .with_criteria(
13             gko::stop::Iteration::build().with_max_iters(1000u).on(gpu),
14             gko::stop::ResidualNormReduction<>::build()
15                 .with_reduction_factor(1e-15)
16                 .on(gpu))
17         .on(gpu);
18     solver->generate(give(A))->apply(lend(b), lend(x));
19     write(std::cout, lend(x));
20 }

```

Figure 9.1: A simple example of using Ginkgo to solve a linear system.

support and extensibility in existing solutions. In addition to linear operators, Ginkgo adds orthogonal abstractions representing hardware implementation aspects. Precisely, Ginkgo employs “executors” that link the data with the appropriate memory spaces and the operations with computational resources. It implements several matrix formats, which represent matrix product operators, Krylov methods, which approximate the application of the solver operator, as well as preconditioners, which enhance the convergence rate of the Krylov methods. The linear operator abstraction is left open to extension to allow user-defined implementations of matrix formats, Krylov methods, and preconditioners. To simplify the adaptation of existing methods to specific needs, several other extensible abstractions are provided. These include the ability to modify the stopping criteria used to check convergence of iterative methods, as well as logging hooks, which enable to interactively track the progress of the distinct operations.

Figure 9.1 shows an example where Ginkgo is used to iteratively solve a linear system. Line 5 defines the CUDA device with id 0 as the execution space where the operations will be performed. The linear system data, including the input matrix, the right-hand side, and the initial guess is read from the standard input and stored in linear operators A, b and x in Lines 6–8. Since the specified execution space for these operators is a CUDA device, the data will be automatically copied to the memory space attached to it. Then, Lines 9–17 define a specific instantiation of the solver factory Σ (denoted by the `solver` variable in the code) which will be used to generate the solver operator. The solver operators generated by this specific instantiation of the factory will use the conjugate gradient (CG) Krylov method to approximate its application (Line 10). The method is executed for at most 1000 iterations (Line 13), or until the original residual norm is reduced by 15 orders of magnitude (Lines 14–16), whichever happens first. The Krylov method will be enhanced by a preconditioner generated from the system matrix using the block-Jacobi preconditioner factory Π (Line 11). This part of the example shows that, while there is a unique solver factory Σ , libraries may allow multiple “instantiations” of it to incorporate the fact that the same mathematical operation may be computed in multiple ways, depending on the desired accuracy and problem characteristics. The complete computation is done in Line 18, which generates the solver operator and solves the system. The `generate` method uses the matrix product operator A to create the corresponding solver operator. Since the specified Krylov method is enhanced with preconditioners generated by the block-Jacobi preconditioner factory, this factory’s `generate` method will be used to generate a block-Jacobi preconditioner operator from the matrix product operator A. Then, once the `apply` method is called, the CG Krylov solver is employed to approximate the result of the solver operator application, using the matrix product operator A and the previously generated block-Jacobi preconditioner operator for A. Finally, the approximate solution is written to standard output in Line 19.

9.2 Conclusions and Open Research Directions

This work focused on GPU acceleration of components for the iterative solution of linear systems, and showed that significant performance improvements can be obtained on modern hardware even with basic, well-studied building blocks.

Sparse matrix-vector product. Part II started with a discussion on the most time consuming and difficult to parallelize operation: the sparse matrix-vector product. Even though a variety of advanced sparse matrix formats and accompanying matrix-vector product algorithms were recently proposed, Chapter 2 demonstrated that most of them exhibit corner cases where the formats either consume significantly more memory than the standard formats, or achieve far lower matrix-vector product performance than standard implementations. Furthermore, since the majority of existing software packages provides high performance implementations of other operations, they are often restricted to one of the standard formats as a means of managing software complexity and developer burden. Thus, while application-specific formats can undoubtedly outperform the conventional alternatives, they should only be developed in case the potential improvements over the formats provided by the underlying general-purpose library (accounting for the necessary format conversions required for interfacing with other parts of the library) outweigh their development cost.

Chapter 2 focused on reducing the improvement potential of specialized formats — and the need to invest resources in the development of application-specific formats — by optimizing relevant corner cases of the most widely used CSR format. Enabled by advancements in accelerator technology, which recently started offering full support for atomic operations, the proposed optimizations effectively deal with the issue of imbalanced sparsity patterns. While the classical approach still offers superior performance for regular sparsity patterns, the ultimate matrix-vector product algorithm can be obtained by coupling the two realizations with a simple heuristic that predicts the winner, and selects the superior approach automatically.

Chapter 3 continued the development of the sparse matrix-vector product by identifying the COO format as an alternative general-purpose format for GPUs. Similarly to the improved CSR algorithm, the new kernel developed for the COO format is highly efficient and does not suffer from extremely unfavorable sparsity patterns. Furthermore, its higher minimum and average performance make it a better default choice than CSR. Ultimately, reasonable use-cases can be found for both options: the improved CSR algorithm can be used in conjunction with software that relies on CSR historically or in case the efficiency of other operations depends on it. In contrast, COO can be adopted as the default choice for new software whose performance does not depend on CSR.

While most sparse matrix-vector product algorithms are focused on large problems that utilize the entire GPU (processor group), Chapter 4 explored the underdeveloped case of smaller problems suitable for individual streaming multiprocessors (single processors). It demonstrated that this case can be implemented more efficiently by slightly modifying standard algorithms to make better use of the available memory hierarchy.

New findings presented in this part can be used in the development of more specialized matrix formats. For example, the ideas from the COO algorithm are currently being used for the development of an improved hybrid matrix format [6]. The success of the synchronization-free load-balancing algorithms on a single GPU should also prove to be useful for the development of sparse matrix-vector product algorithms that utilize the computational resources of the entire node, or even multiple nodes in unison as synchronization and load-balancing penalties become more pronounced on higher levels of the *hardware hierarchy* (vector unit, processor, processor group, node, cluster). While this task is far from straightforward when targeting a general matrix-vector product due to the unpredictable memory access and lack of atomic operations on a distributed memory architecture, previous research on the topic suggests that it may be possible to design effective algorithms if problem-specific details are exploited in algorithm design [1]. Finally, the simple heuristic used to select between the two CSR algorithms is also a small contribution to research in automatic sparse matrix format selection. This research area focuses on selecting the best format based on the properties of the matrix, by using certain policies (often based on machine learning algorithms) to decide among several available formats [20, 25], or even assemble a matrix-vector product implementation from a pool of potential optimizations [14].

Preconditioning. Part III explored the potential of block-Jacobi preconditioning on highly parallel GPU hardware. Even though this relatively simple preconditioner usually features lower convergence improvement than the popular ILU-based preconditioners, this part of the thesis showed that problems with inherent block structure can greatly benefit from block-Jacobi preconditioning. The parallel performance of block-Jacobi can be attributed to its inherent parallelism, as each block can be processed independently. The first step towards a

high performance implementation consists of mapping the blocks to the appropriate level of the hardware hierarchy. By assigning each block to a single vector unit, taking advantage of increased register counts and warp shuffle instructions available on recent hardware, and replacing conventional pivoting strategies with implicit pivoting, the resulting algorithms are able to outperform equivalent functionality available in vendor libraries. The first such algorithm, the block-Jacobi preconditioner based on Gauss-Jordan elimination, was presented in Chapter 5. The algorithm inverts the diagonal blocks during preconditioner generation, which means that the application stage can be realized as a sequence of highly parallel dense used matrix-vector products.

However, as discussed in Section 1.1, the solution of a linear system via matrix inversion can result in numerical instability. Chapter 6 addressed these concerns by showing that, in practice, there is no difference in preconditioner quality when using the explicit inversion-based scheme, as opposed to a factorization-based approach. It also revisited the unconventional Gauss-Huard method for the solution of linear systems, and revealed that this method can be superior to the inversion-based algorithm if only few iterations of the Krylov solver are needed. Finally, Chapter 7 compared the Gauss-Huard solver with the standard LU factorization algorithm, and showed that, provided the conventional “lazy” triangular solve algorithm is replaced with the “eager” variant, the LU factorization can outperform the alternate Gauss-Huard decomposition.

The contributions presented in this part constitute only a small sample of recent developments in preconditioning techniques. Some of these developments include new highly parallel methods for solving triangular systems [3, 8, 22], and parallel generation of threshold ILU preconditioners [5, 9]. As a direct extension of the block-Jacobi preconditioner presented here, future research can explore the effectiveness of other preconditioners based on relaxation methods. An orthogonal research direction would include scaling up the block-Jacobi preconditioner to distributed memory systems. Unlike the sparse matrix-vector product, the regular structure of the block-Jacobi preconditioner allows for relatively straightforward distributed algorithm, as each block can be moved to the memory space that holds the corresponding segments of the input and output vectors, and processed on the processor with direct access to that memory space.

All algorithms presented in this part are also related to a broader topic of “batched” routines, which apply the same operation on a sequence of small problems. While there is a recent proposal for a standardized batched BLAS interface [13], it is still unclear whether this effort will result in wide adoption, as there are major issues with the proposed interface. One such issue concerns the data format used to store the blocks. Since distinct architectures and applications require specific storage schemes (*e.g.*, one parameter batch is shared, another stored as a contiguous sequence, or scattered throughout the memory), covering all the options leads to an exponential growth of interface functions with an unmanageable number of parameters. Another issue arises from the implicit synchronization between two consecutive batched routine calls. While there are no dependencies between the distinct problems forming a batch, the entire batch is still synchronized, as each batched call is essentially a (parallel) loop over all problem instances. While this can be partially alleviated by implementing the routines in terms of a set of jobs submitted to a job scheduling system, relying on the existence of such a system is not always an option. For example, this is the case for block-Jacobi preconditioning presented in this work, where the “batch” of problems is distributed on the GPU (where scheduling systems are not commonly used), and there is additional code needed for preprocessing and postprocessing of the problem data as part of the same GPU kernel. Ultimately, the pragmatic solution might be to depart from the idea of “batched” routines, and instead build libraries that provide BLAS-like functionality for various levels of the hardware hierarchy. Essentially, the responsibility of building the inherently parallel outer loop would be left to the user, removing the need for complicated parameter lists, greatly reducing the number of interface variations, and avoiding implicit synchronizations of unrelated problems. While such libraries are still uncommon, NVIDIA has recently taken a step in this direction with its CUTLASS [12] library, which provides matrix-matrix multiply implementations (GEMMs) for various levels of the GPU hardware hierarchy.

Adaptive precision. As a result of recent trends in HPC and the emergence of hardware support for low precision arithmetic, Part IV evaluated the potential of employing low precision techniques in combination with preconditioning. Chapter 8 analyzed the theoretical aspects and the effect on convergence rate when lower precision storage is used in the block-Jacobi preconditioner. The success of the approach is based on several observations: 1) since the preconditioner application is bounded by the memory bandwidth, performance improvements are possible by reducing the precision of data stored in memory, and not the precision of computations; 2) a preconditioner is already an approximation of the original system matrix, so the storage accuracy does not need to be higher than the accuracy of the approximation; and 3) storage accuracy cannot be reduced

blindly, but has to be carefully tuned to the numerical properties of the problem. Practical experiments showed that the combination of these techniques successfully reduces total storage while preserving preconditioner quality, and, according to a theoretical energy model, can reduce energy consumption of the complete Krylov solver. While a theoretical analysis alone does not guarantee the feasibility of practical implementations, new research has demonstrated the effectiveness of adaptive block-Jacobi preconditioning in practice [15], and produced a first GPU implementation of this preconditioner, which is now available in the Ginkgo library [17].

Since the adaptive precision block-Jacobi represents a pioneering idea in preconditioner design, it may be possible to enhance other preconditioners using similar techniques. Adaptive block-Jacobi preconditioning itself may also be improved further by using non-conventional storage formats instead of the standard IEEE floating point types [15]. Exploiting support for low-precision computing hardware, such as the tensor cores available on the latest generations of NVIDIA GPUs is another possible research direction. However, one would first have to escape the memory-bandwidth-bound nature of preconditioner application (most likely through the use of block Krylov methods, or simply by solving a problem with multiple right-hand sides) before gaining benefits from these hardware features. In the bigger picture of iterative methods, adaptive block-Jacobi preconditioning is only one example of recent research in low-precision methods. In addition to the well-known mixed precision iterative refinement [11], other examples include the adaptive versions of the Jacobi and PageRank relaxation methods [4, 18, 19] and the development of the modular precision storage format, which enables efficient access to the same array of floating point values in multiple precisions [7, 18, 19]. It is also worth mentioning that, orthogonal to the developments in unconventional storage formats, the potential of performing computations in such formats is also being explored. While this research rarely results in performance improvements on conventional hardware, it has the potential to influence the design of future hardware by demonstrating its effectiveness on simulated systems [16, 26].

Scientific software. As a parting note, the final paragraph of this work focuses on the role of scientific software in high performance computing. Leaving aside the question of whether or not “scientific software engineering” should be considered an academic field, it is undeniable that highly efficient software is one of the most important pillars in modern science based on computer simulations, data analytics, and machine learning. Thus, it is not surprising that significant efforts (including this work) are dedicated to the development of novel algorithms and their efficient realization on the latest hardware technology. Paradoxically, even though software is the most valuable contribution of such efforts, researchers’ efficiency is still the most valuable contribution to such efforts, a scientist’s reputation is still mostly determined by the traditional metrics like the number of publications or the Hirsch index. Since the vast majority of journals and conferences does not have any software quality assurance policies in place, most scientific software produced today is developed as a prototype “throwaway” implementation, intended only to support the publication of a scientific paper, instead of providing benefits to the broader community. As such, this software is usually of low quality, poorly documented, and lacks a community which would maintain it. Furthermore, using such software as a basis for performance evaluation in a scientific paper is highly questionable, as it usually does not have to implement edge-case handling, nor takes into account the tradeoffs between performance gains and the manpower cost of its development, maintenance and integration into larger projects. The aspect of reproducibility — one of the central pillars of science — is in many cases also ignored. To avoid these issues, most of the code presented in this work (specifically, the CSR and COO SpMV algorithms, and the inversion based block-Jacobi and adaptive block-Jacobi algorithms) are integrated into the Ginkgo software library [17], ensuring continuing community support and maintenance in the future. On a brighter note, as hardware and software becomes more complex, and the available manpower scarcer, recent years have witnessed the emergence of community efforts with the goal of increasing the quality of academic software [2, 10, 27, 29]. Hopefully, this trend will continue, and ultimately lead to a change of metrics that define the success of a researcher in a direction favourable for scientific software and the high performance community.

9.3 Conclusiones y Líneas abiertas de Investigación

Este trabajo se ha centrado en la aceleración de componentes para la resolución de sistemas lineales dispersos sobre GPUs, y ha mostrado que es posible obtener mejoras de rendimiento significativas en hardware reciente, incluso en el caso de núcleos computacionales básicos muy trabajados.

Producto matriz-vector disperso. El Bloque II se abrió con una discusión de la operación más costosa y más

difícil de paralelizar: el producto matriz-vector disperso. Aunque recientemente se han propuesto una gran variedad de formatos de almacenamiento avanzados, junto con sus correspondientes algoritmos para el producto matriz-vector, el Capítulo 2 demuestra que la mayor parte de estos presentan casos particulares donde, o bien consumen mucha más memoria, o bien ofrecen un rendimiento bastante más reducido que las implementaciones estándar. Además, la mayoría de bibliotecas de programas existentes proporcionan implementaciones optimizadas de otras operaciones, que están restringidas en cuanto a los formatos de almacenamiento que pueden usar como medio para gestionar la cantidad y la complejidad del trabajo del desarrollador. Así, mientras que los formatos específicamente diseñados para algunas aplicaciones pueden batir a las alternativas estándar, los primeros solo tienen sentido en caso de que las mejoras potenciales, respecto a los formatos en bibliotecas de propósito general, compensen el coste de su desarrollo.

El Capítulo 2 se centró en reducir la diferencia de rendimiento frente a los formatos especializados — y la necesidad de destinar recursos al desarrollo de formatos específicos para aplicaciones — a través de la optimización de casos particulares revelantes para el formato CSR. Gracias a los recientes avances en tecnología de aceleradores, que recientemente empezaron a proporcionar pleno soporte para operaciones atómicas, las optimizaciones propuestas resuelven de manera efectiva los problemas asociados con patrones irregulares de dispersión. Si bien es cierto que la aproximación clásica todavía ofrece un rendimiento superior para patrones de dispersión regulares, un algoritmo globalmente óptimo para el producto matriz-vector definitivo puede obtenerse combinando ambas soluciones con un heurístico simple que predice el mejor esquema, y que selecciona la aproximación correspondiente de manera automática.

El Capítulo 3 continuó con el desarrollo del producto matriz-vector disperso identificando el formato COO como una alternativa de propósito general para GPUs. De manera análoga al caso mejorado del algoritmo CSR, el nuevo núcleo desarrollado para el formato COO es altamente eficiente y no sufre ante patrones de dispersión con propiedades extremadamente desfavorables. Además, sus tasas de rendimiento menor y medio más elevadas hacen que sea una opción por defecto más favorable que CSR. En global, es posible obtener casos de uso mejores para cada una de las opciones: el algoritmo CSR mejorado puede utilizarse en combinación con programas que históricamente han explotado el formato CSR, o en aquellos casos en los que la eficiencia de otras operaciones dependa del uso de este formato. Por su parte, el formato COO puede escogerse para nuevos programas cuyo rendimiento no dependa del uso de CSR.

Si bien la mayor parte de algoritmos para el producto matriz-vector se centran en operaciones de cálculo con problemas enormes, que utilizan la GPU al completo, el Capítulo 4 explora el caso menos desarrollado de problemas más pequeños, apropiados para su ejecución en multiprocesadores individuales de la GPU. En particular, el capítulo demuestra que este caso se puede implementar de manera más eficiente modificando los algoritmos estándar para que estos hagan un mejor uso de la jerarquía de memoria.

Los avances presentados en esta parte pueden utilizarse en el desarrollo de formatos más especializados. Por ejemplo, las ideas del algoritmo COO se están usando actualmente para el desarrollo de un formato híbrido [6]. El éxito de los formatos de equilibrado de carga libres de sincronizaciones en una única GPU también pueden ser útiles para el desarrollo de algoritmos para el producto matriz-vector que aprovechen los recursos computacionales del nodo completo, o incluso de múltiples nodos ya que las penalizaciones debidas a sincronizaciones y desequilibrios de carga son más elevadas en los niveles superiores de la jerarquía hardware (unidad vectorial, procesador, grupo de procesadores, nodo y cluster). Si bien esta tarea no es trivial cuando se aplica a un producto matriz-vector general, debido al patrón irregular de acceso a la memoria y la falta de operaciones atómicas en una arquitectura de memoria distribuida, las últimas investigaciones sobre el tema sugieren que es posible diseñar algoritmos efectivos si se explotan los detalles específicos del problema durante el diseño del algoritmo [1]. Finalmente, el heurístico simple para seleccionar entre los dos algoritmos CSR es también una pequeña contribución en materia de investigación hacia la selección automática de formatos para matrices dispersas. Esta área se centra en seleccionar el mejor formato basándose en propiedades de la matriz, utilizando ciertas políticas (a menudo basadas en algoritmos de aprendizaje automático) para decidir entre varios formatos [20, 25], o incluso en ensamblar una implementación de posibles productos matriz-vector a partir de un conjunto de optimizaciones potenciales [14].

Precondicionado. El Bloque III exploró las ventajas del preconditionador de Jacobi por bloques implementado sobre hardware altamente paralelo de tipo GPU. Aunque este tipo de preconditionador, relativamente simple, presenta una menor aceleración de la convergencia frente a los preconditionadores de tipo ILU, esta parte de la tesis doctoral mostró que, para problemas con una estructura por bloques intrínseca, los preconditionadores

de Jacobi por bloques son especialmente eficientes. El rendimiento paralelo de estos preconditionadores puede atribuirse a su paralelismo intrínseco, puesto que cada bloque puede procesarse independientemente. El primer paso en pos del desarrollo de una implementación de alto rendimiento consiste en mapear los bloques sobre el nivel apropiado de la jerarquía hardware. La asignación de un bloque a cada unidad vectorial, aprovechando el incremento en el número de registros y las instrucciones *warp shuffle* disponibles en hardware reciente, y el reemplazo de las estrategias convencionales de pivotamiento con una técnica implícita, ofrece una serie de algoritmos que son capaces de batir en cuanto a rendimiento a las implementaciones de funcionalidad equivalente en las bibliotecas de los vendedores. El primero de estos algoritmos, el preconditionador de Jacobi por bloques basado en la eliminación de Gauss-Jordan, se presentó en el Capítulo 5. Este algoritmo particular invierte los bloques diagonales durante la generación del preconditionador, lo que significa que la etapa de aplicación se puede implementar como una sencilla secuencia de productos matriz-vector densos altamente paralela.

Sin embargo, tal y como se discute en la Sección 1.1, la resolución de un sistema lineal mediante la inversión explícita de matrices en general puede introducir inestabilidades numéricas. El Capítulo 6 aborda esta problemática demostrando que, en la práctica, no existe diferencia en la calidad de un preconditionador que usa el esquema basado en la inversión explícita frente a otro clásico que utiliza la aproximación basada en factorizaciones. Así mismo, también se revisita el método de Gauss-Huard para la resolución de sistemas lineales, y se demuestra que este último puede superar al método de Gauss-Jordan cuando se necesitan solo unas pocas iteraciones para la convergencia. Finalmente, el Capítulo 7 comparó el resolutor Gauss-Huard con el algoritmo estándar de factorización LU, y mostró que, si la versión tradicional “perezosa” (*lazy*) del algoritmo de resolución triangular se reemplaza con una variante “voraz” (*eager*), la factorización LU puede mejorar el rendimiento de la alternativa basada en la factorización Gauss-Huard.

Las contribuciones de este bloque constituyen solo una pequeña parte de los desarrollos recientes en técnicas de preconditionado. Algunos de estos desarrollos incluyen nuevos métodos altamente paralelos para resolver sistemas triangulares [3, 8, 22], y la generación paralela de preconditionadores ILU con umbral [5, 9]. Como extensión directa de los preconditionadores de Jacobi por bloques presentados aquí, la investigación futura puede explorar la efectividad de otros preconditionadores basados en métodos de relajación. Una dirección de investigación ortogonal podría incluir el escalado de los preconditionadores de Jacobi por bloques en sistemas de memoria distribuidos. Al contrario que en el producto matriz-vector disperso, la estructura regular del preconditionador por bloques de Jacobi permite obtener un algoritmo distribuido de manera relativamente directa, puesto que cada bloque se puede alojar en el espacio de memoria que almacena el segmento correspondiente de los vectores de entrada y salida, y además se puede tratar en el procesador correspondiente que tiene acceso a ese espacio de memoria.

Todos los algoritmos presentados en este bloque están asimismo relacionados con el campo más amplio de las rutinas “por lotes” (*batch*), que aplican la misma operación a una secuencia enorme de problemas pequeños e independientes. Si bien existe una propuesta reciente para estandarizar una interfaz para BLAS por lotes [13], en estos momentos todavía no está claro si este esfuerzo resultará en una adopción general de la propuesta, puesto que hay problemas significativos con el diseño de la interfaz. Dado que arquitecturas distintas requieren formatos de almacenamiento específicos (por ejemplo, uno de los parámetros del lote es compartido, otro se almacena como una secuencia contigua en memoria, o un tercero se distribuye sobre la memoria), cubrir todas las posibles opciones conlleva un crecimiento exponencial en el número de funciones en la interfaz, con un volumen inmanejable en el número de parámetros. Otro problema surge de la sincronización implícita entre dos llamadas a rutinas de procesamiento por lotes. Aunque no existen dependencias entre los distintos problemas que conforman un lote, el lote entero todavía funciona de manera síncrona, puesto que cada rutina por lotes equivale, básicamente, a un bucle que recorre todas las instancias del problema. Este inconveniente puede resolverse parcialmente mediante la implementación de las rutinas como un conjunto de trabajos enviados a un sistema de planificación de trabajos, pero confiar en que tal sistema exista siempre puede no ser una buena opción. Por ejemplo, este es el caso del preconditionador de Jacobi por bloques presentado en este trabajo, en el que el lote de problemas se distribuye en la GPU (donde no es común disponer de gestor de planificación), y existe un sobre coste debido a la necesidad de preprocesar y posprocesar los datos del problema como parte el mismo núcleo GPU. En último extremo, la solución práctica puede requerir abandonar la idea de rutinas por bloques, y en su lugar construir bibliotecas que proporcionan una funcionalidad similar a la del BLAS para varios niveles de la jerarquía hardware. En esencia, con esta aproximación, la tarea de construir el bucle externo paralelo se dejaría en manos del usuario, eliminando la necesidad de una compleja y larga lista de parámetros,

reduciendo enormemente las posibles variantes en la interfaz, y eliminando la imposición de sincronizaciones implícitas sobre problemas no relacionados. Aunque este tipo de bibliotecas todavía son poco comunes, recientemente NVIDIA ha mostrado su posición favorable a esta línea, con su biblioteca CUTLASS [12], que proporciona implementaciones del producto general de matrices (GEMM) para varios niveles de la jerarquía hardware de la GPU.

Precisión adaptativa. Como resultado de algunas tendencias actuales en computación de altas prestaciones (CAP) y el soporte nativo en circuitería a la aritmética de precisión reducida, el Bloque IV evaluó las ventajas de introducir técnicas de precisión reducida en preconditionadores. En concreto, el Capítulo 8 analizó los aspectos teóricos y el efecto sobre la velocidad de convergencia de los métodos iterativos de Krylov cuando se emplea un esquema basado en precisión reducida en un preconditionador de Jacobi por bloques. El éxito de esta aproximación se basa en varias cuestiones: 1) dado que la operación de aplicación del preconditionador está limitada por el acceso a la memoria, es posible mejorar el rendimiento reduciendo la precisión de los datos almacenados en memoria, pero manteniendo la precisión de las operaciones aritméticas; 2) un preconditionador es una aproximación de la matriz original del sistema, así que la precisión del almacenamiento no tiene por qué ser superior a la precisión de la aproximación; 3) la precisión del almacenamiento no puede reducirse a ciegas, sino que tiene que ajustarse cuidadosamente según las propiedades numéricas del problema. Los experimentos prácticos muestran que la combinación de estas técnicas consigue reducir el volumen de almacenamiento total mientras que preserva la calidad de preconditionador y, según muestra un modelo de consumo de energía teórico, puede reducir el consumo del resolutor de Krylov global. Aunque un análisis teórico por sí solo no garantiza la viabilidad del esquema, un esfuerzo reciente de investigación ha demostrado la efectividad de los preconditionadores de Jacobi por bloques con precisión adaptativa en la práctica [15], y ha producido una primera implementación para GPUs de este preconditionador, que ya se ha integrado con la biblioteca Ginkgo [17].

Los preconditionadores de Jacobi por bloques con precisión adaptativa representan una idea pionera, y es posible técnicas similares se puedan aplicar en el caso de otros preconditionadores. También es posible mejorar el esquema de preconditionado de Jacobi por bloques adaptativo en sí mismo, mediante el uso de formatos de almacenamiento no convencionales, alternativos a los formatos estándar de IEEE [15]. Explotar el hardware de precisión reducida, como los cores tensores presentes en las últimas generaciones de procesadores de NVIDIA, es otra línea de investigación abierta. Sin embargo, primero debe evitarse la naturaleza limitada por la memoria de la aplicación de los preconditionadores (muy probablemente, a través del uso de métodos de Krylov por bloques, o simplemente resolviendo varios sistemas de ecuaciones, con la misma matriz de coeficientes, de manera simultánea) antes de poder aprovechar las tecnología ofrecida por este tipo de hardware. En la visión en conjunto de los métodos iterativos, el preconditionado de Jacobi por bloques adaptativo es solo una de las líneas de investigación sobre métodos de precisión reducida para la resolución de sistemas lineales. Además del conocido método de refinamiento iterativo con precisión mixta [11], otros ejemplos incluyen las versiones adaptativas de los métodos de relajación de Jacobi y PageRank [4, 18, 19] y el desarrollo de un formato de almacenamiento de precisión segmentada, que posibilite un acceso eficiente en diferente precisión a un conjunto de valores en coma flotante almacenados sobre un vector [7, 18, 19]. También es interesante mencionar que, de manera ortogonal al desarrollo de formatos de almacenamiento no convencionales, se están explorando las ventajas de desarrollar los cálculos también en esos formatos alternativos. Si bien esta aproximación raramente producirá una mejora del rendimiento si se efectúa sobre hardware convencional, sí tiene la capacidad potencial de influir en el diseño de hardware futuro a través de la demostración de su efectividad a través de simulación [16, 26].

Software científico. Como nota final, la parte final de este trabajo se centra en el papel del software científico en CAP. Dejando al margen la cuestión de si la “ingeniería de software científico” debería considerarse como un campo académico, es innegable que el software altamente eficiente es uno de los pilares más importantes de la ciencia contemporánea, basada en simulaciones por computador, análisis de datos, y aprendizaje automático. Así, no resulta sorprendente que se hayan dedicado esfuerzos significativos, incluyendo esta tesis doctoral, al desarrollo de nuevos algoritmos y su implementación eficiente sobre las últimas tecnologías hardware. Paradójicamente, aunque el software resultante es la contribución más valiosa de estos esfuerzos y la eficiencia de los investigadores es el recurso máspreciado, la reputación de un científico todavía se mide en número de publicaciones o su índice Hirsch. Puesto que la mayor parte de revistas y conferencias no tiene políticas establecidas sobre la garantía del software, el software científico producido hoy en día se desarrolla,

en gran medida, como un prototipo de “publicar-y-tirar”, con el único propósito de soportar la publicación de un artículo científico, en lugar de ayudar a la comunidad científica. Bajo estas premisas, habitualmente estos programas presentan una baja calidad, están pobremente documentado, y carecen de una comunidad que esté dispuesta a mantenerlos. Además, usar estos programas como base para el análisis del rendimiento en un artículo científico es altamente cuestionable, puesto que no implementan los casos extremos, ni tienen en cuenta el equilibrio entre las ganancias observadas en el rendimiento y sus costes de desarrollo, mantenimiento e integración en proyectos más grandes. La cuestión de la reproducibilidad también se ignora en muchos casos. Para evitar estos problemas, la mayor parte del código presentado en esta tesis (específicamente, los algoritmos para el producto matriz-vector disperso basados en CSR y COO, y el preconditionador de Jacobi por bloques con inversión explícita y precisión adaptativa) se han integrado en la biblioteca Ginkgo [17], con el propósito de asegurar en el futuro un esfuerzo continuo de desarrollo y mantenimiento por parte de la comunidad. Como nota positiva, a medida que el hardware y los programas se han vuelto más complejos, y los recursos humanos más escasos, en los últimos años hemos sido testigos de esfuerzos por parte de la comunidad científica para incrementar la calidad del software académico [2, 10, 27, 29]. Confiamos en que esta tendencia perdurará, y eventualmente pueda llevar a un cambio de las métricas que definen el éxito de un investigador en una dirección que sea beneficiosa para el desarrollo de software científico de calidad y la comunidad que trabaja en CAP.

Bibliography

- [1] A. Abdelfattah, H. Ltaief, D. Keyes, and J. Dongarra. Performance optimization of sparse matrix-vector multiplication for multi-component pde-based applications using gpus. *Concurrency and Computation: Practice and Experience*, 28(12):3447–3465, 2016.
- [2] H. Anzt and G. Flegar. Are we doing the right thing? — a critical analysis of the academic HPC community. In *Proceedings of the 20th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing*, page submitted. IEEE, 2019.
- [3] H. Anzt, E. Chow, and J. Dongarra. Iterative sparse triangular solves for preconditioning. In *Proceedings of the 21st International European Conference on Parallel and Distributed Computing*, Euro-Par 2015, pages 650–661. Springer, 2015.
- [4] H. Anzt, J. Dongarra, and E. S. Quintana-Ortí. Adaptive precision solvers for sparse linear systems. In *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing*, E2SC’15, page 2. ACM, 2015.
- [5] H. Anzt, E. Chow, and J. Donagarra. ParILUT—a new parallel threshold ILU factorization. *SIAM Journal on Scientific Computing*, 40(4):C503–C519, 2018.
- [6] H. Anzt, T. Cojean, Y.-C. Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang. Load-balancing sparse matrix vector product kernels on gpus. *ACM Transactions on Parallel Computing*, submitted, 2018.
- [7] H. Anzt, G. Flegar, V. Novaković, E. S. Quintana-Ortí, and A. E. Tomás. Residual replacement in mixed-precision iterative refinement for sparse linear systems. In *High Performance Computing*, pages 554–561. Springer, 2018.
- [8] H. Anzt, T. K. Huckle, J. Bräckle, and J. Dongarra. Incomplete sparse approximate inverses for parallel preconditioning. *Parallel Computing*, 71:1–22, 2018.
- [9] H. Anzt, T. Ribizel, G. Flegar, E. Chow, and J. Dongarra. A parallel threshold ILU for GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IPDPS 2019. IEEE, 2019. accepted.
- [10] Better Scientific Software (BSSw). <https://bssw.io/>, 2019.
- [11] E. Carson and N. J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing*, 40:A817–A847, 2018.

- [12] CUTLASS. <https://github.com/NVIDIA/cutlass>, 2019.
- [13] J. Dongarra, I. S. Duff, M. Gates, A. Haidar, S. Hammerling, J. Higham, J. Hogg, P. Valero-Lara, D. Rellon, S. Tomov, and M. Zounon. A proposed API for batched basic linear algebra subprograms. Technical report, The University of Manchester, 2016.
- [14] A. Elafrou, G. Goumas, and N. Koziris. Performance analysis and optimization of sparse matrix-vector multiplication on modern multi- and many-core processors. In *Proceedings of the 46th International Conference on Parallel Processing*, ICPP 2017, pages 292–301. IEEE, 2017.
- [15] G. Flegar, T. Cojean, H. Anzt, and E. S. Quintana-Ortí. Customized-precision block-Jacobi preconditioning for Krylov iterative solvers on data-parallel manycore processors. *ACM Transactions on Mathematical Software (TOMS)*, submitted.
- [16] G. Flegar, F. Scheidegger, V. Novaković, G. Mariani, A. E. Tomás, A. C. I. Malossi, and E. S. Quintana-Ortí. FloatX: A C++ library for customized floating-point arithmetic. *ACM Transactions on Mathematical Software (TOMS)*, submitted.
- [17] Ginkgo. <https://ginkgo-project.github.io>, 2019.
- [18] T. Grützmacher and H. Anzt. A modular precision format for decoupling arithmetic format and storage format. In *Proceedings of the 24th International European Conference on Parallel and Distributed Computing*, Euro-Par 2018, pages 434–443. Springer, 2018.
- [19] T. Grützmacher, H. Anzt, F. Scheidegger, and E. S. Quintana-Ortí. High-performance GPU implementation of PageRank with reduced precision based on mantissa segmentation. In *Proceedings of the 2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms*, IA3, pages 61–68. IEEE, 2018.
- [20] P. Guo, L. Wang, and P. Chen. A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1112–1123, 2014.
- [21] K. Hoffman and R. A. Kunze. *Linear Algebra*. Pearson India Education Services, 2nd edition, 2015.
- [22] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter. A synchronization-free algorithm for parallel sparse triangular solves. In *Proceedings of the 22nd International European Conference on Parallel and Distributed Computing*, Euro-Par 2016, pages 617–630. Springer, 2016.
- [23] MAGMA 2.5.0. <http://icl.cs.utk.edu/magma/>, 2019.
- [24] PARALUTION. <http://www.paralution.com/>, 2015.
- [25] B.-Y. Su and K. Keutzer. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS’12, pages 353–364. ACM, 2012.
- [26] G. Tagliavini, A. Marongiu, and L. Benini. FlexFloat: A software library for transprecision computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Early Access, 2018.
- [27] The TOMS Initiative and Policies for Replicated Computational Results (RCR). <https://toms.acm.org/replicated-computational-results.cfm>, 2019.
- [28] ViennaCl. <http://viennacl.sourceforge.net/>, 2015.
- [29] xSDK: Extreme-scale Scientific Software Development Kit. <https://xsdk.info/>, 2019.

List of Figures

1.1	A pseudocode of the Conjugate Gradient Krylov method	6
1.2	Derivation of common sparse matrix storage formats	8
2.1	Data layouts for sparse matrices	18
2.2	Sparsity pattern and memory consumption of FREESCALE/TRANSIENT	19
2.3	A (sequential) C implementation of the CSR-I algorithm	20
2.4	Storage consumption of different sparse matrix formats and overhead compared to CSR	22
2.5	GFLOPS distribution of SpMV implementations	22
2.6	Comparison of SpMV implementations	23
2.7	Comparison between CSR and CSR-I implementations	24
2.8	Relationship between $s[n_{zr}]/E[n_{zr}]$ and speed-up/slow-down of CSR-I over CSR	25
3.1	Different storage formats for a sparse matrix	28
3.2	C00 SpMV kernel design	29
3.3	Nonzero count vs. size for the considered test matrices	31
3.4	Histogram for the (standard deviation / avg) of the nonzero-per-row metric	32
3.5	Evaluating the effect of the parameter ω on the performance of the C00 kernel	33
3.6	Performance of the distinct SpMV kernels for all problems included in the test suite	33
3.7	Fastest kernel comparison	34
3.8	Performance statistics for the distinct kernels over large test cases	34
3.9	Runtime overhead of the distinct SpMV kernels	35
4.1	Basic storage formats and memory consumption for sparse matrices	40
4.2	Sequential C implementations of basic SpMV algorithms	42
4.3	Performance of SpMV routines for homogeneous batches	44
4.4	Performance of CSR-based SpMV routines for a homogeneous batch	45
4.5	Performance of the flexible batched SpMV routines for all matrices in the test benchmark	46
4.6	Performance and memory bandwidth of SpMV for a heterogeneous batch using selected matrices	47
4.7	Performance and memory bandwidth of SpMV for a heterogeneous batch using all matrices	47
5.1	Basic GJE implementation in Matlab notation using standard pivoting	53
5.2	Basic GJE implementation in Matlab notation using implicit pivoting	54
5.3	Generation of the block-Jacobi preconditioner	55
5.4	Illustration of the memory requests for the cached extraction and shared extraction	57
5.5	Performance comparison of batched matrix inversion routines for various batch sizes	59
5.6	Performance comparison of batched matrix inversion routines for various matrix sizes	60
5.7	Performance improvement from reducing the shared memory size in the generation step	61
5.8	Sparsity plots of test matrices used to evaluate the diagonal block extraction	61
5.9	Block-Jacobi generation time for increasing block sizes and various nonzero distributions	62

5.10	Block-Jacobi generation time for matrices taken from the SuiteSparse matrix collection	63
5.11	Performance comparison of the generic and the GEMV-based preconditioner application	64
5.12	IDR(4) convergence and performance comparison for different preconditioner block sizes . . .	64
5.13	Detailed comparison of IDR(4) enhanced with block-Jacobi using different block sizes	66
6.1	Basic GH implementation in Matlab notation	71
6.2	Speedup of BGJE and BGH / BGHT over BLU	73
6.3	Ppreconditioner application runtime and BiCGSTAB convergence for block-Jacobi variants . .	74
6.4	Total execution time for BiCGSTAB enhanced with block-Jacobi	75
7.1	Basic LU factorization in Matlab notation using explicit and implicit pivoting	82
7.2	Batched factorization and triangular solve (TRSV) in a block-Jacobi preconditioner setting . .	83
7.3	Illustration of the “lazy” and “eager” TRSV algorithms variants	84
7.4	Loop-body of the “lazy” and “eager” TRSV algorithms variants	84
7.5	Illustration of the memory requests for the shared memory extraction	85
7.6	Performance of batched factorization routines depending on the batch size	87
7.7	Performance of batched factorization routines depending on the size of the matrices	88
7.8	Performance of batched triangular solve routines depending on the batch size	89
7.9	Performance of batched triangular solve routines depending on the size of the matrices	90
7.10	IDR(4) convergence using block-Jacobi preconditioning based on LU or GH	90
7.11	Execution time for IDR(4) enhanced with block-Jacobi preconditioning based on LU or GH . .	91
8.1	Mathematical formulation of the PCG method	99
8.2	Algorithmic formulation (in MATLAB) of the PCG method	100
8.3	Algorithmic formulation (in MATLAB) of the FCG method	101
8.4	Control flow for deciding whether or not to select a reduced format	103
8.5	Details of the procedure for deciding whether or not to select a reduced format	103
8.6	Boxplot for the distribution of the condition numbers	105
8.7	Breakdown of the diagonal blocks stored in fp64, fp32, or fp16.	108
8.8	Energy efficiency of PCG with block-Jacobi using various precisions	108
9.1	A simple example of using Ginkgo to solve a linear system	118

List of Tables

1.1	Common direct methods for the solution of linear systems	4
1.2	Common relaxation methods for the solution of linear systems	5
3.1	Statistical information on the GFLOPs metric of the SpMV kernel	35
5.1	Iterations and execution time of IDR(4) enhanced with GJE-based block-Jacobi	67
7.1	Iterations and execution time of IDR(4) enhanced with LU-based block-Jacobi	92
8.1	Matrix properties and iteration counts of PCG with the preconditioner stored in various precisions	106