Multicore Acceleration of Sparse Electromagnetics Computations

David Moisés Fernández Becerra

Doctor of Philosophy

Department of Electrical & Computer Engineering

McGill University

Montreal, Quebec

July 2011

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

DEDICATION

This thesis is dedicated to my loving family Adriana, Diana, Alejandro, my parents José and Margarita a continuous source of inspiration, and my special angel Blanca.

ABSTRACT

Multicore processors have become the dominant industry trend to increase computer systems performance, driving electromagnetics (EM) practitioners to redesign their applications using parallel programming paradigms. This is especially true for computations involving complex data structures such as sparse matrix computations that often arise in EM simulations with the finite element method (FEM). These computations require pointer manipulation that render useless many compiler optimizations and parallel shared memory frameworks (e.g. OpenMP). This work presents new sparse data structures and techniques to efficiently exploit multicore parallelism and short-vector units (the last of which has not been exploited by state of the art sparse matrix libraries) for recurrent computationally intensive kernels in EM simulations, such as the sparse matrix-vector multiplication (SMVM) and the conjugate gradient (CG) algorithms. Up to 14 times performance speedups are demonstrated for the accelerated SMVM kernel and 5.8x for the CG kernel using the proposed methods over conventional approaches for two different multicore architectures.

Finally, a new method to solve the FEM for parallel processing is presented and an optimized implementation is realized on two different generations of NVIDIA GPUs (manycore) accelerators with performance increases of up to 27.53 times compared to compiler optimized CPU results.

ABRÉGÉ

Les processeurs multicœurs sont devenus la tendance dominante de l'industrie pour accroître la performance des systèmes informatiques, forçant les concepteurs de systèmes électromagnétiques (EM) à reconcevoir leurs applications en utilisant des paradigmes de programmation parallèle. Cela est particulièrement vrai pour les calculs impliquant des structures de données complexes comme les calculs de matrices creuses qui surviennent souvent dans des simulations électromagnétiques (EM) avec la méthode d'analyse par éléments finis (FÉM). Ces calculs nécessitent de manipulation de pointeurs qui rendent inutiles de nombreuses optimisations du compilateur et les bibliothèques de mémoire partagée parallèle (OpenMP, par exemple). Ce travail présente de nouvelles structures de données rares et de nouvelles techniques afin d'exploiter efficacement le parallélisme multicœur et les unités de vecteur court (dont le dernier n'a pas été exploité par des bibliothèques de matrices creuses à la fine pointe de la technologie) pour les noyaux de calcul intensif récurrents dans les simulations EM, tels que les multiplications matrice-vecteur rares (SMVM) et des algorithmes à gradient conjugué (CG). Des performances d'accélérations jusqu'à 14 fois supérieures sont démontrées pour le noyau accéléré par SMVM et jusqu'à 5,8 fois supérieures pour le noyau CG en utilisant les méthodes

proposées par rapport aux approches conventionnelles pour deux architectures multicœurs différentes.

Enfin, une nouvelle méthode pour résoudre la FÉM pour le traitement parallèle est présentée et une implantation optimisée est réalisée sur deux générations d’accélérateurs de GPU NVIDIA (multicœur) avec des augmentations de performances allant jusqu'à 27,53 fois par rapport aux résultats du CPU optimisé par compilateur.

ACKNOWLEDGMENTS

I would like express my deepest respect and gratitude to my supervisors, Professors Dennis D. Giannacopoulos and Warren J. Gross, who have continuously provided their keen insight and guidance, key to the success of this doctoral thesis. Their advice has always kept me focused on the big picture, the take home message and the special sauce. Special thanks to Professor Dennis Giannacopoulos for his financial support in the most stringent times and our frequent talks; they were always a cheerful ending to our meetings. I would also like to thank my sponsor, la Universidad del Zulia, for offering me the opportunity and financial support to realize my doctoral studies in this prestigious university.

To my wife Adriana and children Diana and Alejandro whose love and support continuously inspires me. To my parents that always support me unconditionally and continue to be my role models, to my brothers and sisters Daniel, Mónica, Shirley, Reinier, and my mother-in-law that can always find ways to cheer me.

Special thanks to Maryam Mehri Dehnavi a great colleague with whom I did great research. Also to Kevin Guangran, a good friend through this long road. Last but not least, I like to thank my colleagues in the Computational Electromagnetics Lab with whom I spent many endearing moments and that made my stay a pleasant experience.

TABLE OF CONTENTS

DEDICATION ii
TABLE OF CONTENTS vii
LIST OF FIGURES ix
LIST OF TABLES xii
LIST OF ACRONYMS xiii
Chapter 1 Introduction 1
1.1 Motivation 1
1.2 Literature Review 5
1.2.1 Modern Technological Trends 6
1.2.2 Previous Work in SMVM and CG for Multicore 9
1.3 Main Objective of this Work 15
1.4 Thesis Organization 15
Chapter 2 Review of Sparse Matrix Concepts, the Sparse Matrix-Vector Multiplication and Conjugate Gradient Algorithm 17
2.1 What are Sparse Matrices? 18
2.2 Structure in Sparse Matrices 19
2.3 Sparse Matrix Formats 21
2.3.1 General Sparse Formats 21
2.3.2 Sparse Matrix Formats that Exploit Matrix Structure 23
2.3.3 Sparse Matrix Formats for Vector Processing 24
2.4 Sparse Matrix Repositories 26
2.5 Overview of Sparse Matrix History and Software 26
2.6 A Closer Look at the Sparse Matrix-Vector Multiplication (SMVM) Operation 27
2.7 The Conjugate Gradient (CG) Algorithm 31
2.8 Summary of Bottlenecks for SMVM and CG 35
Chapter 3 Accelerating the SMVM Algorithm for Multicore Processors 37
3.1 A Naïve Approach to Parallelizing the SMVM Kernel on Multicore Processor 37
3.2 Overview of Multicore Architecture and their Programming Challenges 40
3.2.1 Architectural Characteristics of the Two Hardware Platforms Used 40
3.2.2 Programming Challenges for Exploiting Architectural Features 43
3.3 Putting it All Together: A New Sparse Matrix Format and SMVM Kernel for Parallel Multicore Computing 47
3.3.1 The New Pipeline-Matched Sparse (PMS) Matrix Format 48
3.3.2 Vectorizing the SMVM Algorithm with PMS 53
3.3.3 Scheduling Multiple Cores with the Vectorized SMVM 55

3.4 Experimental Results 59
3.4.1 Experimental Setup 60
3.4.2 Test Results 61
3.5 Concluding Remarks 69
Chapter 4 Blocked PMS Format 72
4.1 Blocking PMS for the SMVM Kernel 72
4.1.1 The Blocked-Pipeline-Matched (BPMS) Sparse Matrix Format 72
4.1.2 Enhancing Block Structure in BPMS 77
4.1.3 Vectorized SMVM and Multicore Parallelization for the BPMS Format 78
4.2 Experimental Results for SMVM 84
4.3 Conjugate Gradient Results 90
4.4 Concluding Remarks 94
Chapter 5 A Parallel Approach to Solving the Finite Element Method 97
5.1 New FEM Single Element Solution (FEM-SES) Method: Alternate *Fine-Grained* Parallelism in the Solution of FEM 98
5.1.1 FEM-SES Mathematical Formulation 101
5.1.2 The 2-Step Iterative Relaxation Method 102
5.2 Proof of Concept Results 104
5.3 Parallelizing Results for the FEM-SES Method 107
5.3.1 Multicore Results for the Intel Platform 108
5.3.2 Manycore Adaptation of the FEM-SES Method 110
5.4 Related Work 125
5.5 Concluding Remarks 128
Chapter 6 Conclusions and Future Work 130
6.1 Original Contributions 131
6.2 Future Work 133
Appendix A: Compilation of Modern Matrix Libraries 136
Appendix B: A Flexible and Portable Multithreaded Library for Sparse Matrix Computations (FPMSparseLib) 140
B.1 Description of the Library and the Sparse Matrix Data Structure 142
B.2 Description of the Sparse Matrix Data Structure Used in the Library 142
Bibliography 145

LIST OF FIGURES

Figure 1: Dominant computational kernels in electromagnetic simulations. 5
Figure 2: Sparse matrix non-zero pattern representation (can___24 from Matrix Market [42]). 19
Figure 3: Example of structured and unstructured sparse matrices. 20
Figure 4: Representation of a general sparse matrix in the COO, CSR, CSR and MCSR formats. 23
Figure 5: Sparse matrix-vector multiply kernel using the CSR format. The algorithm to the left uses an inner (dot) product approach with stride-1 access to matrix data; whereas, the algorithm to the right uses a *saxpy* approach with non-sequential access to matrix data. 29
Figure 6: Conjugate gradient algorithm. Where ε is the tolerance used, α and β are the constants used to update the x vector of unknowns and the new search direction d, r is the residual vector, and ∂ and q are temporary variables. 33
Figure 7: Two parallel approaches to the sparse matrix-vector multiplication. The example presented here assumes a multicore processor with 4 cores. The color in each block refers to the processor in charge of the computations for that block. The white vector in subfigure (a) is broadcasted across all processors. The blue vectors in subfigure (b) are summed by the same processor, thus are the same color. 38
Figure 8: Block diagrams of the architectural features of modern multicore processors. Subfigure (a) shows the diagram for the homogeneous multicore Intel Core 2 Quad processor family [65], and subfigure (b) shows the diagram for the Cell BE heterogeneous multicore processor family [12]. 42
Figure 9: Representation of a sparse matrix in CSR format and the new Pipeline-Matched Sparse (PMS) representation assuming a vector pipeline with of two floating point numbers. 50
Figure 10: Strip-mining or loop-unrolling of the inner loop in the dot-product version of the SMVM algorithm for a strip-size of 4 using the PMS format configure for 4-SPFP vector units (128-bit wide SIMD units). 55
Figure 11: Two-level partitioning scheme of a matrix. Coarse grained partitions generate row blocks, and fine grained partitions create smaller data sets to transfer in a block fashion. 58
Figure 12: Double-buffering implementation for the Cell-SPEs. The fine grained partitions are defined to be the size of a single buffer in the SPEs. 58
Figure 13: Speedup results presented as the time ratio of SMVM-CSR/SMVM-PMS for each hardware platform. 63
Figure 14: Performance in MFlops/s of the SMVM kernel for the CSR and PMS formats. 65
Figure 15: Performance scaling results of the SMVM kernel for the Cell BE and the Intel processor using the CSR and PMS sparse formats. 67

Figure 16: Representation of a sparse matrix in BCSR format and the new block pipeline-matched sparse (BPMS) representation 75
Figure 17: Bandwidth reduction examples when applying the reverse Cuthill-Mckee algorithm. . 79
Figure 18: Partitioning example for the "can___24" matrix after reordering using the level-set vector generated in RCMK. The number along the brackets to the right of the matrix identifies the worst case scenario for the size of the blocks that could be generated by such partitioning..... 80
Figure 19: Blocked sparse matrix-vector kernel using the BPMS format. 80
Figure 20: Steps to compute the 1st-level partitions of the 2-level partitioning scheme and to statically load balance a matrix in BPMS format across several processors. 83
Figure 21: Speedup results for the SMVM kernel with the PMS, BPMS and BCSR with respect to the CSR format for one Intel-core..... 87
Figure 22: Multicore performance scaling in GFlops/s for the SMVM kernel using the CSR, BCSR, PMS and BPMS formats on the Intel Core 2 Quad processor. 90
Figure 23: Speedup scaling for the conjugate gradient method using BPMS with 1 to 4 Intel CPU cores. 92
Figure 24: Conjugate gradient performance results in GFlops/s using 4 Intel cores. 92
Figure 25: Scaling of the CG performance in GFlops/s for the modified PMS (MPMS) format. ... 94
Figure 26: Complete workflow used to compute the CG algorithm with the vectorized SMVM kernel and the 2-level partitioning scheme. 96
Figure 27: Comparison of classic finite element method (FEM) workflow (top figure) with respect to the proposed single element solution (FEM-SES) method (bottom Figure). 100
Figure 28: The 2-step iterative relaxation method. Step 1 shows examples of how to compute the solutions for elements with 1, 2, and 0 BCs. The examples here correspond to triangular elements with first order basis functions. 103
Figure 29: 2D section of electrostatic coaxial cable. The inner conductor is fixed at 10V and outer conductor at 0V. The space between the conductors is considered free space. The insert shows a 3D representation of the cable being model..... 105
Figure 30: Energy results for the FEM and FEM-SES. 106
Figure 31: Iterations scaling of the FEM-SES method for increasing number of unknowns..... 106
Figure 32: Profiling results for the sequential FEM-SES method. 108
Figure 33: Multicore implementation of the FEM-SES method for the Intel processor. 109
Figure 34: Performance comparison of the proposed FEM-SES method running on four Intel cores compared to a single core vectorized FEM using CG algorithm. The blue and red lines indicate the time of the two versions of FEM, and the green dotted line represents the speedup of the parallel FEM-SES method with respect to the single core optimized FEM-CG. 110
Figure 35: Basic architecture of NVIDIA's first generation G80 CUDA enabled graphic cards. . 112
Figure 36: Basic architecture of NVIDIA's third generation FERMI graphic cards. 114

Figure 37: Workflow to parallelize the 2-step iterative relaxation method on NVIDIA GPUs. 117
Figure 38: Kernel 1-CUDA kernel used to compute the single element solutions in FEM-SES. 118
Figure 39: Speedup of the Fermi GPU FEM-SES implementation versus the classic FEM sequential implementation on the Intel CPU using a CG solver. 122
Figure 40: Speedup of the GPU solution times with respect to the CPU times. 122
Figure 41: Time distribution of the kernels used to implement the 2-step iterative relaxation method. 125
Figure 42: Dense matrix libraries classification. 137
Figure 43: Sparse matrix libraries classification. 138
Figure 44: Sparse direct and iterative solver libraries. 139
Figure 45: Library organization by functionality. 141
Figure 46: Sparse matrix data structure. 143

LIST OF TABLES

Table 1: Classification of important sparse matrix libraries based on the concurrent model implemented. 28
Table 2: Main bottlenecks for the sparse matrix-vector multiplication (SMVM) and the conjugate gradient (CG) algorithm. 36
Table 3: Programming challenges for implementing scientific kernels (e.g. SMVM) on modern multicore processors. 48
Table 4: Finite element (FE) test matrices from the Matrix Market repository [42]. The matrices are square with number of rows and columns equal to the matrix rank (in column 3). The number of nonzeros (NZ) are shown in column 4, column 5 has the percentage fill of the matrix with respect to the dense case, the total nonzeros with padding are shown in column 6, and column 7 contains the percentage of added nonzeros. 62
Table 5: SMVM speedup scaling using different formats for the "s3dkq4m2" matrix (5). 68
Table 6: Finite element (FE) test matrices from the Matrix Market repository [42]. 85
Table 7: SMVM performance comparison results for the BPMS, PMS and BCSR SMVM kernels for the Intel Core 2 Quad and Cell BE processors. 89
Table 8: Comparison of architectural features for the GeForce 8800 GT and the GeForce GTX 480 GPUs. In the Fermi GPU, shared memory and L1 Cache share a common space of 64KB. 112
Table 9: Finite element mesh dimensions for the 2D coaxial cable test case and solution times in seconds (last three columns) for the 2-step iterative relaxation method. 121
Table 10: Limiting performance factors for the GPU kernels. 123
Table 11: Performance results for the FEM-SES method of the hand optimized CPU code versus the GPU implementations. 125
Table 12: Classification of sparse matrix libraries according to the concurrent paradigm implemented. 136
Table 13: Library description and file dependency. 144

LIST OF ACRONYMS

CPU: central processing units.

GPU: graphic processing units.

CMP: chip multiprocessor.

SMP: symmetric multi-processor.

CE: computational electromagnetics.

EM: electromagnetics.

FEM: finite element method.

SIMD: single instruction multiple data processing (also short vector processing).

SIMT: single-instruction multiple-thread (model used by NVIDIA GPUs).

CSR: compressed sparse row, also called compressed row storage (CRS)

BCSR: blocked compressed sparse storage.

PMS: pipeline-matched sparse matrix format.

BPMS: blocked pipeline-matched sparse matrix format.

SPD: symmetric positive definite matrices.

SMVM: sparse matrix-vector multiply.

saxpy: scalar a x plus y (x, y : vectors) following the LAPACK convention.

(P)CG: (preconditioned) conjugate gradient algorithm.

Chapter 1
Introduction

1.1 Motivation

Computational electromagnetics (CE) is increasingly an area of active research whose applications are not only important in electrical engineering areas (e.g. communications, circuit design, optics, electromagnetic compatibility, etc.) but also influence importantly other areas of knowledge such as medicine [1], biology [2], and geophysics [3] to mention a few. Regardless of the particular electromagnetic (EM) application, there is a continuous demand for more detailed simulations obtained in reasonable times, using evermore complex models, both to better understand the particular problem and to create more efficient solutions without confronting the high costs associated with design prototyping, testing and refinement.

The solution of such increasingly complex computational electromagnetic (and generally challenging scientific) problems has relied, in part, on the continual advances in microprocessor technology, namely increasing clock speeds and instruction level parallelism, during the past three decades which came to be known as the *free ride* [4]. However, technological limitations (frequency, power, radiation, cross-talk and other) have dictated the need to explore new alternatives. The partial solution to some of these problems has

been sought with some degree of success by integrating multiple central processing units (CPUs) cores in a single die, namely *multicore*[1] processors. The independent computing cores in multicore processor are clocked at lower frequencies than their single core predecessors, which make imperative the use of parallel programming to benefit from such newer architectures. Even though multicore processors increase system performance by providing modest parallel resources, they also require greater programming effort which might not compensate for the gain in performance as argued in a recent study from a group at Berkley [5]. Moreover, this group points out that the strategy followed by manycore[2] processors can provide a longer-term solution to current technological limitations capable of higher performance. Although manycore processors offer higher performance through increased number of simpler processing cores with similar programming effort as that required by multicore ones, they are only efficient on compute intensive data-parallel applications and have yet to overcome several other limitations to become a viable solution for these specific

[1] The term *multicore* processors refers to the new technology trend that integrates several general purpose processor cores into a single die, sharing part of the cache hierarchy usually at level 2 or 3 depending on the actual implementation.

[2] The term *manycore* processors refers to chips with hundreds of simple processing cores inside a single die. These cores are often grouped into small cluster that may share a local memory hierarchy and are all connected though the global memory system and specialized interconnects fabrics.

applications [6]. Modern graphic processing units (GPUs) are examples of these manycore processors.

The trend towards multicore/manycore microprocessors clearly established the need to address parallel programming paradigms early in application development to benefit from the computing potential of emerging mainstream architectures, creating new programming opportunities and challenges. In particular for the EM community it portrays the need to redefine numerical methods, specifically their dominant computing kernels, in parallel terms.

Consequently, the constant demand for more detailed simulation can only be satisfied with new clever numerical methods implementations that efficiently use modern computational resources. The main objective of this work is to present new techniques to efficiently exploit the different hardware architectural features found in modern multicore/manycore processor to accelerate EM computations.

Three numerical methods stand out in EM simulations, the finite difference time domain (FDTD) method, the method of moments (MOM, also known as boundary integral equation method BIE), and the finite element method (FEM). Among these, the FEM has enjoyed great popularity mainly due to its ability to provide continuous solutions throughout the modeled space, model irregular geometries (a limiting factor for FDTD), and generate sparse linear systems (as

opposed to the dense systems of MOM) that can be efficiently exploited computationally.

The solution of the sparse linear systems obtained from the FEM is frequently the most time consuming operation in these simulations. The sparse matrix-vector multiplication (SMVM) $Ax = b$, where a sparse matrix A usually multiplies a dense vector x to yield a dense vector b, is one of the dominant computing kernels in popular iterative solvers such as the conjugate gradient (CG) method that often dominates computational time. In fact it has been recently classified as the second of the “Seven Dwarfs” (most frequently found kernels in scientific computations) in [5], and it represents a central topic of this work, as shown in Figure 1, among other contributions. Emphasis is made on accelerating the conjugate gradient method with the accelerated sparse matrix vector multiplication kernels on homogeneous and heterogeneous multicore processors; although a specific contribution for manycore processors is also presented to accelerate FEM.

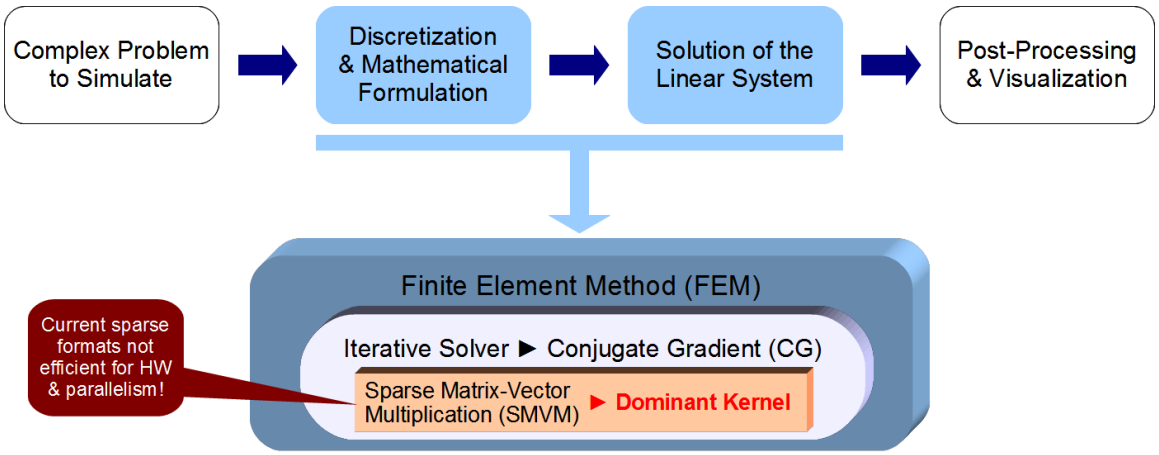


Figure 1: Dominant computational kernels in electromagnetic simulations.

1.2 Literature Review

A wealth of knowledge has been developed in sparse computations for over five decades starting from the 1960's. Most of the work done up to recent years had been concerned with deploying sparse algorithms on single core processors, symmetric multi-processors (SMP), mainframes and more recently on clustered processors. The new trend towards multiple core processors and heterogeneous systems has radically redefined hardware platform for these algorithms, thus requiring similar changes in sparse algorithms and data structures. This subsection presents a brief overview of the modern technological trends that have motivated the changes in modern sparse computations, followed by relevant contributions in the context of the sparse matrix vector multiplication and the conjugate gradient algorithm to accelerate these operations mainly related to multicore processors. Although this section provides an overview of the previous

work done related to this work and identifies the caveats in knowledge, a set of comprehensive references are provided in the following chapters as required to support the design decisions taken. In particular, Chapter 2 studies in detail the concepts related to sparse matrices, SMVM and CG providing also citing landmark references.

1.2.1 Modern Technological Trends

For over four decades since 1965 the evolution of computer systems advancements was marked by Moore's Law. Computer architects chose to favour increasing single thread performance of general purpose processors over more expensive vector and symmetric multiprocessor alternatives. Greater advancements were then evidenced in these general purpose single core CPUs with increasing transistor counts that enabled larger and more complex cache designs, superscalar pipelines, and increased clock frequency. The appearance of cluster computing contributed to the trend of relegating parallel computing to clusters or grids of computers which constitute today's supercomputing environments.

These advancements in turn created several technology challenges among which three are of particular importance: increased power leakage and dissipation constraints represents an important concern in modern processors, mainly referred to as the *power wall*; frequency increases also lead to more

power demands and transmission problems such as jitter, crosstalk interference and others, which are referred to as the *frequency wall*; and the third called *memory wall* refers to the gap in performance between the main system memory and the computing speeds. As this gap continues to grow, feeding sufficient data to the processor so that it is always kept busy becomes unattainable. Integrating multiple cores into modern processors has become an alternative to provide the increasing processing power that users have come to expect.

Multicore processors originally appeared in 2001 with the introduction of the Power4 processor by IBM and in 2004 with Sun's UltraSPARK IV. They were soon followed by the first x86 compatible multicore processor (dual core Opteron, April 21st) by AMD in 2005, followed in 2006 with the Pentium Dual-Core processor from Intel and the Niagara processor from SUN Microsystems, all which established the most important landmark of the decade in computer architecture advances. On the other hand, general purpose computing on manycore processors (GPGPU) started to become popular approximately since 2003 [7] (although earlier references exist), however this was restricted to a reduced community familiar with the specialized graphic development environments available.

Programming general purpose applications using graphic application programming interfaces (APIs) required great effort since the applications had to

be written with specialized graphic instructions not related to the actual operations being done. After the introduction of the Compute Unified Device Architecture (CUDA) [8] software/hardware architecture in 2006 for general purpose computing using standard high level programming languages (e.g. C/C++ and Fortran), manycore processors became an important parallel computing resource to accelerate applications for the broader scientific community. This trend has continued to strengthen with other important initiatives such as AMD's accelerated parallel processing (APP) software development kit (SDK), formerly ATI Stream [9, 10] used to program ATI GPUs and the OpenCL [11] standard for programming parallel of heterogeneous systems. In spite of the high integer and floating point performance offered by manycore processors compared to multicore processors, it is important to note that manycore processors are not a standalone solution by themselves. They are designed for compute intensive data-parallel applications, performing very poorly for sequential applications, applications that require more input/output (from here on I/O-I/O bounded[3]) operations, and for applications with many low computational load parallel tasks.

Hybrid multicore processors have also emerged as possible alternatives to the technological limitations mentioned before. For example the Cell broadband

[3] I/O bounded applications are also termed bandwidth bound or limited applications.

engine [12] is a multicore chip with nine processors, one IBM Power cores called power processor elements (PPEs) and eight vector processors called synergetic processing elements (SSEs) that deliver high computing power with considerably lower performance/power ratio than conventional homogeneous multicore systems. Other types of hybrid chips will appear in the near future which combine multicore with manycore processors. An example of one such system is the emerging AMD family of accelerated processing units (APUs) called AMD fusion, which combines in a single die a general purpose multicore CPU, a discrete GPU, and memory/IO controller hub (also referred to as Northbridge), substantially reducing the power consumption of the system [13]. Multicore and manycore technologies, as well as their future hybrid combinations, constitute the current trend that is expected to dominate modern microprocessor advances in the foreseeable future, being implemented in almost all modern processors offered by the most important industry leaders (e.g., Intel, AMD, IBM, and NVIDIA).

1.2.2 Previous Work in SMVM and CG for Multicore

An in-depth study to characterize the limiting performance factors for the sparse matrix-vector multiplication kernel for multicore systems is carried out in [14] by Goumas *et al.*, using a large set of matrices from different problem domains. The main contribution of this work is to report its findings as a set of

guidelines to efficiently implement optimizations on the SMVM kernel in modern processors, but does not introduce new techniques to enhance its performance nor to leverage vector processing in these new processors. Goumas *et al.* identify the inner product as one of the main limiting factors in the SMVM operations for row schemes, and the *saxpy* operations for column schemes.

The work by Eun-Jin Im and Katherin Yelick in [15] presents one of the first attempts to create an optimized toolbox called Sparsity [16, 17] to accelerate the sparse matrix vector operation. Im *et al.* use techniques such as register and cache blocking, loop transformations, multiple vectors, exploiting symmetry, special diagonal data structures and matrix reordering mainly aimed at superscalar single core processors. This optimized framework [17] was shown to obtain up to 4X performance enhancements over conventional sparse implementations on single core and symmetric multi processor systems. The most beneficial optimization performance-wise in this work was the register blocking technique. The same group from Berkley led by Richard Vuduc, James Demmel, and Kathy Yelick have expanded the work to provide support for general sparse matrix operations in an autotuning toolbox called Optimized Sparse Kernel Interface (OSKI) [18], but have yet to offer support for shared/distributed memory and vector processing.

Similar performance enhancements of up to 2.5x speedup (from 40MFlops/s to 100MFlops/s) where demonstrated by Toledo [19] by employing similar techniques (that improve instruction level parallelism and increase locality) to those in [17] on a RISC POWER2 processor. Pinar and Heath [20] expand on Toledo's work by defining variable blocks with a new ordering algorithm to reduce cache misses due to memory indirections, showing performance increases of up to 33%.

A later paper by Rajesh Nishtala *et al.* [21] studies in detail the effect of *cache blocking* on modern processors, demonstrating performance of up to 2.93x for single core processors. Nevertheless, this technique is shown to provide performance gains under very strict matrix and vector circumstances. Performance gains are shown only when the input vector is large and the output vector is small compared to the cache size, and the non-zero pattern of the sparse matrix exhibit a nearly random distribution. This is commonly not the case for matrix systems derived from FEM and FDTD methods thus little performance gain can be expected from cache blocking in these applications.

In [22, 23] Samuels *et al.* optimize the SMVM kernel using the techniques in OSKI in addition to vectorization for multicore systems. They show that the most important gain when computing the SMVM kernel on multicore systems comes from the parallelization across different processing cores instead of the single

core optimizations of [17, 18]. In particular, this work states that vector processing using single-instruction multiple-data (SIMD) processing units of modern homogeneous multicore processors resulted in little or no performance gain compared to straight C code when compiler optimization flags are used. The limited performance gains obtained for the SIMD processing in the aforementioned work can be attributed to the fact that it only uses standard sparse matrix formats (namely CSR, BCSR and BCOO) that help to implement blocking techniques, but are not the best suited for vector operations.

Specialized sparse matrix formats for vector computing have been devised in the past and shown to exploit efficiently vector processors as shown in [24] and the references therein, even though they require certain structure in the matrix to be efficient (e.g. non-zeros distributed across diagonals, or similar number of non-zeros per row). Some specialized sparse formats for vector processing are ELLPACK-ITPACK [25-28], diagonal format (DIA) [26, 28, 29], and jagged diagonal storage (JDS or JAD) [30]. Another important approach to exploit vector processors, less sensitive to the matrix structure, is the segmented scan operations proposed in [31]; however it was mainly designed for older vector computers.

Considering that SIMD vector processing has the potential to accelerate floating point performance from two to four times (for double and single precision

respectively in current processors), the question of enhancing the performance of the sparse matrix-vector multiplication and that of sparse iterative solvers on modern multicore/manycore processors while exploiting SIMD vector processing still remains an open problem.

Relative modern compilations of state-of-the-art knowledge for iterative solvers have also been presented. Among the most important are the work presented by Saad in [26, 30] that treats different iterative methods and eigenproblems with special considerations for parallel processing, as well as the work of Barrett *et al.* [32]. Demmel *et al.* also present a modern compilation of current state of the art knowledge for eigenproblems, direct methods and iterative methods in [33] specific to parallel environments. Across these works, three main bottlenecks for parallelizing iterative solvers are identified: inner products, the matrix-vector products, and the preconditioning operations. However, most of the work for iterative solvers has been developed in the context of distributed memory systems (e.g. clustered systems) with very few references to shared memory environments. In particular, research in multicore systems for iterative linear systems is quite an active area considering that these systems have been available for less than a decade.

One of the earliest works done to accelerate CG for multicore processors is presented by Wiggers *et al.* [34]. The acceleration is done on the SMVM

operation using the compressed sparse row (CSR) matrix format, OpenMP to exploit the parallel cores, and the Intel MKL library for the linear algebra optimizations. It was determined that the overhead from OpenMP increases with the number of cores used, which limited the performance of this implementation. Furthermore, the use of the CSR format also limits the attainable performance of the optimizations done with the Intel MKL library which accounts for the relative low performance presented. GPU results are also presented for the 8800 GTX NVIDIA graphics card that outperformed the multicore results by 2.56X, which are low performance results for this GPU.

In [35] an approximate inverse preconditioner is used with the BiCG-Stab method and parallelized using a pool of threads to avoid thread creation and deletion on a multicore system to overcome standard thread library overheads. Lee I. [36] presents an adaptation of the general parallel PCG scheme used in distributed memory systems and adapts it for multicore processing. Lee proposes a classic threaded barrier scheme to synchronize the execution of threads for the different operations in the PCG algorithm, and presents performance models for both the SMVM and preconditioner solve operations that include the effect of cache hierarchy in multicore processors. However, this work only presents the theoretical analysis lacking practical result to compare with the performance models presented.

It also apparent that most of the work done related to the conjugate gradient solver (linear solvers in general) is mainly concerned with parallelizing the applications using a shared memory approach, disregarding other possible optimizations such as vector processing using specialized matrix formats, that have the potential to enhance the general parallel performance of these solvers.

1.3 Main Objective of this Work

This work aims at accelerating the sparse matrix vector multiplication operation and the conjugate gradient algorithm exploiting the parallel cores and vector units of modern multicore processors in the context of finite element electromagnetics, which are the main voids identified in the literature. To achieve this goal new sparse matrix formats and algorithms must be devised. The implementation of such matrix formats and algorithms requires advance data structures and other basic functionality that can be assembled into a new sparse matrix library, which is presented as an additional contribution of this work. An alternate approach to exploiting parallelism in the finite element method is also proposed as an alternative to accelerating its dominant computing kernels.

1.4 Thesis Organization

The remainder of this dissertation is organized as follows; Chapter 2 reviews basic concepts, formats and algorithms for sparse matrices. This chapter also analyses the main bottlenecks of the sparse matrix-vector multiplication and

conjugate gradient algorithms. Next, Chapter 3 presents a new technique to accelerate the sparse matrix-vector multiplication for multicore processors with user controlled local memories, showing results to demonstrate its benefits. Based on this new technique, an alternative approach is proposed for cache based multicore processors in Chapter 4.

Chapter 5 presents an alternative approach to solving the finite element method for multicore and manycore environments that does not use traditional direct or iterative solver approaches. Finally, Chapter 6 presents the conclusions and future directions of this work.

Chapter 2
Review of Sparse Matrix Concepts, the Sparse Matrix-Vector Multiplication and Conjugate Gradient Algorithm

Sparse matrices[5] where initially used in the 1960s by Electrical Engineers to solve linear systems derived from electric networks according to Saad [30], although the term “sparse matrix” is attributed in [37, 38] to Harry M. Markowitz for his work in economics in 1957. The motivation then was to alleviate the memory (counted in Kilobytes at the time) and computational demands of the ever-growing linear systems in spite of using more complex data structures and algorithms; moreover, in those days some problems where simply not feasible to solve using dense representations. These more complex sparse representations resulted in substantial memory savings in the orders of $O(n^2)$ locations, and computational savings of order of $O(n^3)$ operations [14] for compute intensive kernels such as the GEneral Matrix Multiply, also known as GEMM following the LAPACK [39] notation. Since then sparse matrix computations have gained much popularity and are now the standard approach to solve increasingly complex systems whenever sparse matrices are available. Deciding when to represent a matrix as sparse or dense is somewhat of an art and it commonly depends on the definition used of sparse matrix. This chapter presents the basic definitions

[5] The term “system” will be used interchangeably with that of “matrix” or “matrices” throughout this work.

related to sparse matrices that will be used throughout this dissertation and analyses the bottlenecks in the sparse matrix-vector multiplication and conjugate gradient algorithm.

2.1 What are Sparse Matrices?

Sparse matrices are usually defined in terms of the relationship among the zeros and non-zeros entries in the system. In [40] Duff defines sparse matrices in terms of “the ratio of the zero to non-zero entries in the matrix”, implying that the matrix is mainly populated with zero entries (i.e. making the matrix aspect ratio zero/nonzero greater than one). An example of a sparse matrix is presented in Figure 2, showing only the distribution pattern of the matrix nonzeros. The sparse matrix illustrated in this figure is a square matrix of rank 24 with 160 nonzeros, which results in a zero/nonzero ratio of 2.6. This is actually a small value that often occurs in small matrices, whereas bigger matrices usually have a much greater aspect ratio.

A more practical definition (that will be used throughout this work) based on those provided by Duff [40] and more recently Stathis [41] defines a sparse as:

> A system where the number of non-zeros and/or their distribution provides advantage in performance or resource wise when the matrix is represented and operated in compressed form (only using its non-zero entries).

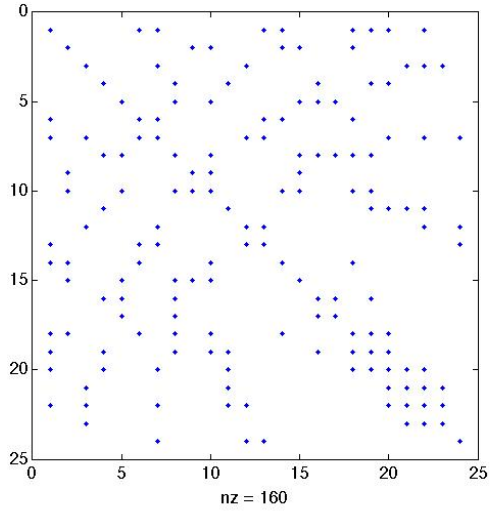
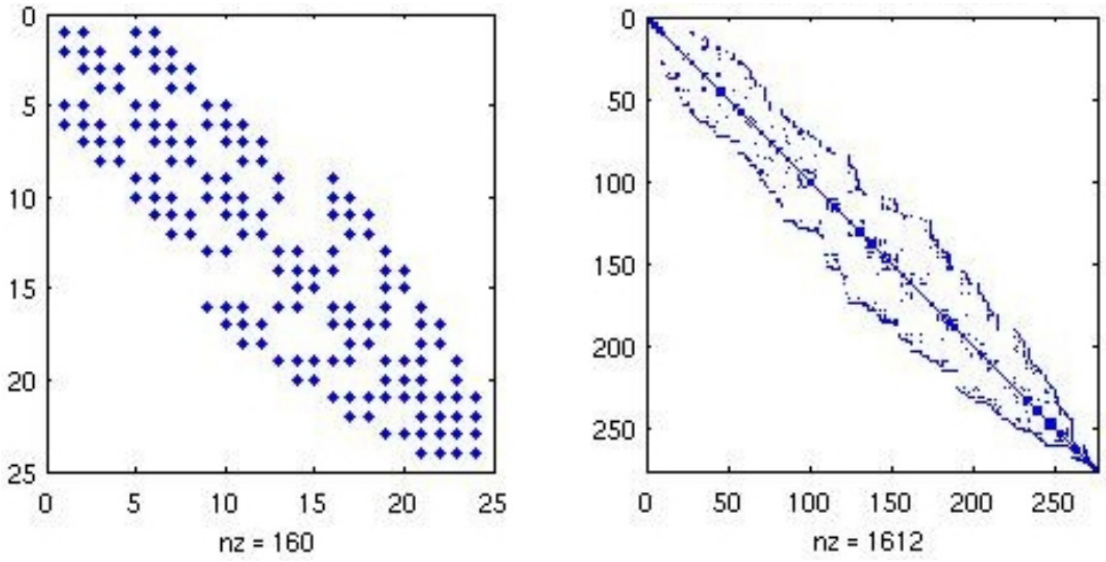


Figure 2: Sparse matrix non-zero pattern representation (can___24 from Matrix Market [42])

2.2 Structure in Sparse Matrices

The non-zero entries distribution in sparse matrices will vary significantly depending on several factors such as the numerical method used, the problem dimensionality and its geometry, and the meshing method among others. Depending on the combinations of these factors, a matrix may be classified as structured or unstructured (or irregularly structured) [30]. *Structured* matrices refer to those where the non-zeros are distributed in a regular pattern inside the matrix usually along diagonals. The non-zero distribution may also be composed of small dense blocks laid out in a block diagonal pattern. On the other hand, *unstructured* matrices are those that do not exhibit a regular pattern in their non-zero entry distribution. The matrix structure is important when defining efficient ways of representing sparse matrices both for storage and computations.



(a) Structured (b) Unstructured

Figure 3: Example of structured and unstructured sparse matrices.

For example a totally random sparse matrix will have no regularity in its nonzero distribution so it can be classified as irregularly structured or unstructured. Matrices derived from rectangular grids using the Finite Difference Method (FDM) will normally give rise to regularly-structured matrices, whereas matrices derived from complex mesh geometries using the Finite Element (FEM) or Finite Volume Methods (FVM or method of moments-MoM) may lead to unstructured matrices. An example of structured and unstructured matrices is presented in Figure 3, where the x-axis and y-axis represents the column and row indices respectively. The next section presents some important sparse matrix formats identifying those that exploit structure in sparse matrices.

2.3 Sparse Matrix Formats

A sparse matrix may be represented in several different ways using specialized formats. Different formats have been created to exploit special characteristics of the matrix structure, algorithm or machine-architecture targeted, but all share the common goal of storing and operating on the non-zero entries of the sparse matrix. This subsection presents some of the most important sparse matrix formats.

2.3.1 General Sparse Formats

Amongst the many different formats that exist, there are four commonly used to represent sparse matrices that make no assumption on the matrix nonzeros structure, the COO, CSR, CSC and MSR. The simplest format is called the coordinate format (COO) or triplet format, since it stores each entry of the sparse matrix in a triplet structure containing the matrix value, its row index and column index. If the nonzero values of the matrix are stored in a specific order (e.g. by rows or columns) then a more efficient representations of the matrix indices may be done, which is the approach used in the other three general formats.

The second format is called compressed sparse row (CSR) format, also referred to as compressed row storage (CRS), AJI (from A, JA-column indices, IA-row indices), or YALE format. In CSR three vectors represent the sparse matrix; the first vector usually called A stores the matrix nonzero (nz) values in

row order; the second called *JA* (or *AJ*), stores the column indices of the nonzero values; and the third vector, *IA* (or *AI*), contains the index to the first element of each row, including an additional index with the total number of nonzeros. Many efficient libraries used to solve linear systems implement this format or some variation of it, as is the case with PETSc [43]. The third format is the column counterpart of CSR termed compressed sparse column (CSC) format, where the nonzeros are stored by columns, while the second vector stores the row indices of each nonzero element, and the third contains the indices of the elements that begin a new column in the data and row index vectors.

The fourth of the general formats is the modified compressed sparse row (MSR) that contains only two vectors. The first vector stores the nonzero elements of the main diagonal first, then skips the n+1 position and then stores the remainder nonzero elements of the sparse matrix. The second vector stores the index values that point to the beginning of each row for the off-diagonal elements in the first vector, then skips the n+1 position and stores the column index of the corresponding nonzero elements in the first vector. Figure 4 shows a general sparse matrix and its representations in each of the four general formats described. It is obvious from this figure that the COO format is the most inefficient as far as storage is concerned which has relegated its use to mainly storing sparse matrices, especially since the other formats can be readily derived from it.

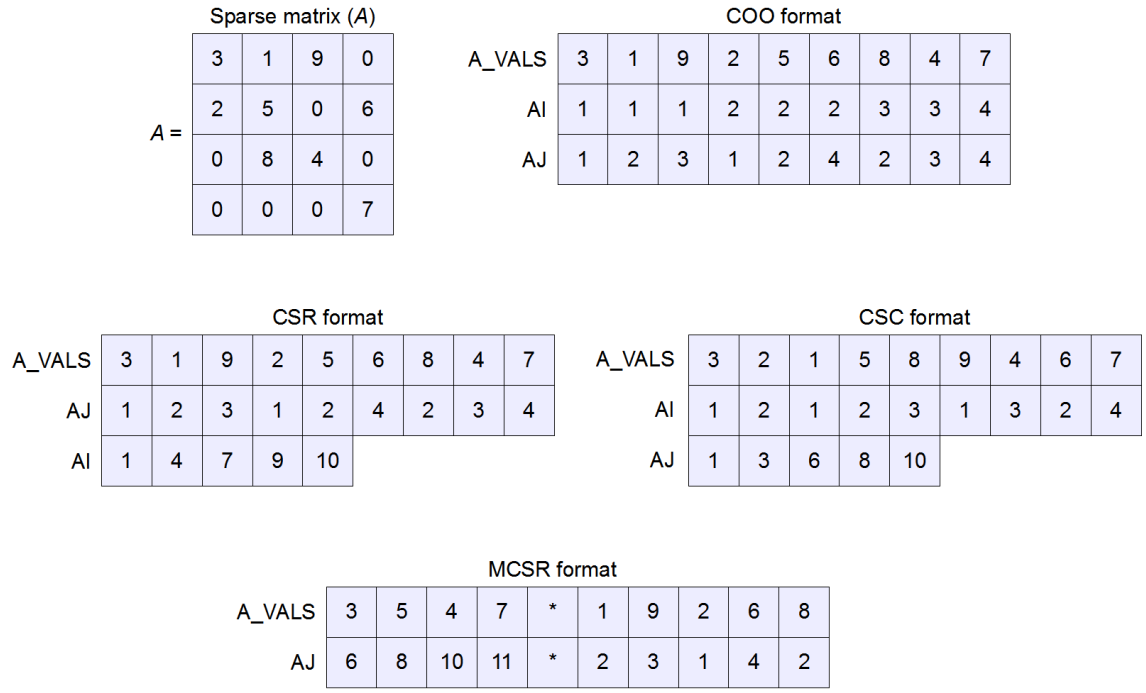


Figure 4: Representation of a general sparse matrix in the COO, CSR, CSR and MCSR formats.

2.3.2 Sparse Matrix Formats that Exploit Matrix Structure

Other formats take advantage of the regularity in the nonzero distribution; these formats and many others are described in [30, 44], but only the most popular ones are described here. For structured matrices with nonzero diagonal patterns the compressed diagonal storage (CDS) format or diagonal (DIAG or DIA) format can be efficiently used. This format stores the matrix by diagonals in a *nd*n* array where *nd* is the number of the matrix diagonals and *n* is the number of matrix columns. An auxiliary vector of size *nd* containing the offsets of each diagonal stored from the main diagonal. This format is also well suited for vector processing due to its long diagonal vector structures.

If the matrix nonzeros are grouped into small dense clusters regularly across the matrix (not all in diagonal patterns) then a more efficient pattern may be used called the blocked compressed sparse row (BCSR or BSR) format. The BSR format is also useful to implement cache and register blocking techniques to exploit specific architectural features in a target processor. BSR is a blocked version of the CSR format that stores the matrix row-wise in three arrays. The first vector stores the matrix values row-wise in small dense matrices of the same size; the vector size is *bdim*num_blocks*, where *bdim* is the non-zeros per block and *num_blocks* is the number of blocks in the matrix. Next, the column index of the first element in each block is stored in a column index vector; finally, the indices to the elements of each block starting a new row-block are stored in a row index vector. This format compresses the row and column index information providing important memory savings.

2.3.3 Sparse Matrix Formats for Vector Processing

The two most popular formats for vector processors documented in [30, 44] are the ELLPACK-ITPACK (ELL) format and the jagged diagonal sparse (JDS) format, which provide long vectors of the same size (for ELLPACK-ITPACK) or mostly the same size (JDS) well suited for vector processing and loop transformation techniques (e.g. loop unrolling). Both formats assume that the number of nonzeros per row is nearly the same, otherwise they are not efficient.

Sparse matrices in ELL are stored in two dense matrices. The first, stores the matrix values by rows padding with zeros each row to match the size of the largest row. The second matrix stores the index column of each element in the first matrix. If most of the matrix rows contain the same number of nonzeros, regardless of their distribution, then this format will provide an efficient way to store and process the matrix. If the matrix has several groups of rows of the same size then the JDS format is more efficient. JDS requires the matrix rows to be ordered from the largest to the shortest one. Once the rows are ordered the matrix values are stored column-wise in a vector as follows: first, the first element in each row is stored in the first column; next, the second element of each row is stored in the second columns, and this procedure continues until all matrix values are stored. A second vector is used to store the column indices of the nonzeros in the same order as they were stored; and finally, a third vector is used store the pointers to the matrix values that begin a new column, with an extra index to determine the size of the last column.

Many other special purpose sparse matrix formats exist, a summary of such formats can be found in the work by Saad [28, 30], Barret *et al.* [32], Stathis [41], Vuduc [16] and the references therein. This dissertation adopts the three characters naming scheme proposed in [28] for its compact referencing of different sparse matrix formats.

2.4 Sparse Matrix Repositories

The matrices used in this work were obtained from the Matrix Market repository [42]. These matrices can also be obtained from the University of Florida Sparse Matrix Collection [45]. These repositories contain the entire Harwell-Boeing Sparse Matrix Collection (Release I), Yousef Saad's SPARSKIT collection, the Nonsymmetric Eigenvalue Problem (NEP) collection of Bai, Day, Demmel and Dongarra, and matrices generated from other sources.

2.5 Overview of Sparse Matrix History and Software

Direct methods were the first to be implemented for solving sparse systems; this was the main topic of the first “Sparse Matrix” symposium [46] held in 1968. These classic methods, namely Gauss elimination, LU and Cholesky decompositions including matrix pivoting, reordering and partitioning techniques, have been documented in a comprehensive survey by Duff [40] in 1977 and latter in a book by Duff, Erisman and Reid [47] in 1986. A more updated reference on sparse direct methods can be found in the book by Timothy Davis [48] from 2006, which revisits LU and Cholesky decomposition and introduces QR decompositions with state of the art algorithms most of which are currently being used in MatLab with some variations.

Early work on iterative solvers for sparse systems has been compiled in the book by Richard Varga [49], and the work by Young and Hageman [50, 51].

Fixed-points methods where used to solve sparse systems in the 1960's, a little after direct solvers for sparse systems and then projection methods came to dominate the solution of sparse systems to date. A modern treatment on iterative methods for sparse linear systems can be found in the book by Saad [30]. Both direct and iterative methods are also studied in the classic reference book by Golub and Van Loan [52].

The works referenced here and in Chapter 1 have given rise to numerous sparse matrix libraries mainly targeted at shared or distributed memory systems. Table 1 shows some of the most important sparse matrix libraries known today (refer to Appendix A for a more extensive reference of other important dense and sparse matrix libraries) characterized by the concurrent model they implement. The libraries shown here are concerned either with exploiting local memory hierarchies or optimizing distributed memory executions, but none exploits SIMD processing which is a central theme in this work.

2.6 A Closer Look at the Sparse Matrix-Vector Multiplication (SMVM) Operation

The sparse matrix-vector multiplication operation (also called SpMV or SpMxV) is one of the most recurrent and time consuming kernels in scientific computing, where a sparse matrix A multiplies a dense vector x to generate a dense vector b as shown in (1).

$$Ax = b \tag{1}$$

Table 1: Classification of important sparse matrix libraries based on the concurrent model implemented.

Concurrent model	Library
Sequential	**Sparselib++ [53]**: iterative solvers. **C(X)sparse [48]**: direct solvers. **ITSOL(SparseKit) [44]**: iterative solvers. **Sparsity [17]**: sparse matrix vector multiply. **Oski [18]**: sparse BLAS.
Distributed memory (MPI)	**PSBLAS [54]**: sparse BLAS and direct and iterative solvers. **pARMS [55]**: sparse iterative solvers. **PETSc [43]**: PDE sparse iterative solvers.

The main objective of the SMVM kernel is to limit the number of computations and storage to the matrix non-zero entries only, taking advantage of the sparsity nature in the matrix. This apparently simple operation has been and continues to be the subject of much research to optimize its performance as presented in section 1.2.2. A classic SMVM algorithm using the CSR format is presented in Figure 5, where the matrix nonzeros are stored in *A_VAL*, *AJ* stores column indices and the *AI* contains the row pointers.

Dot-product SMVM

```
1: for i=1 to number of rows do,
2:    b[i]=0.0
3:    for j=AI[i] to AI[i+1]-1 do,
4:       b[i]=b[i]+A_VAL[j]*x[AJ[j]]
5:    end for
6: end for
```

Saxpy SMVM

```
1: for i=1 to number of rows do,
2:    b[i]=0.0
3: end for
4: for j=1 to number of columns do,
5:    for each element k in col=j and row=i do,
6:       b[i]=b[i]+A_VAL[k]*x[AJ[k]]
7:    end for
8: end for
```

Figure 5: Sparse matrix-vector multiply kernel using the CSR format. The algorithm to the left uses an inner (dot) product approach with stride-1 access to matrix data; whereas, the algorithm to the right uses a *saxpy* approach with non-sequential access to matrix data.

There are three main performance bottlenecks in the sparse matrix-vector multiplication kernel as follows:

- The matrix entries have no data (temporal) reuse and little *spatial locality*[6].
- Depending on the sparse matrix format used, access to the multiplying *x-vector* or the results *b-vector* is indirect and irregular; i.e. the vector elements are fetched using the matrix index information thus usually little spatial and temporal locality is available.
- There is a great deal of instruction overhead in the SMVM kernel required to identify the proper range of non-zeros to compute on (e.g. per row, column, diagonal etc.) and fetch the vector data using the indirect indexing mentioned before.

[6] Temporal locality for data refers to the reuse of the same memory location in at least two distinct instructions at different times. Spatial locality refers to the use of nearby memory locations [51].

Depending on the choice of sparse format and algorithmic implementation (e.g. how the matrix is traversed, by rows, columns, diagonals, etc.) the effect of these three limiting factors may be minimized or maximized. Nevertheless, these choices are often influenced by other subtle factors such as the architecture of the target processor and the programming language used. In general, one aims at selecting a sparse format that exploits the matrix structure and an algorithm that accesses matrix data sequentially (i.e. in a streaming fashion) with a *stride-1* access pattern, also referred to as *sequential locality* [56]. An example of this is provided by Petersen and Arbenz in [57] page 148, where they show results for different implementations of the sparse matrix-vector multiplication kernel where the algorithm is formulated in terms of dot-products or *saxpy* operations for a shared memory system. These results show that on any of the two approaches the outer loop is the most beneficial to parallelize. Even though the performance scalability of the two approaches is similar as stated by the authors, their results also consistently show that as the processor number grows the dot-product approach is better than the *saxpy* version, which can be mainly attributed to less synchronization points in the former approach.

Now considering the efficiency of the SMVM kernel with respect to computations and data transfers, one immediately observes that it is *bandwidth*

bound[7] (or I/O bound). In SMVM, each nonzero (*nz*) matrix entry will be multiplied by an entry of the *x* vector and accumulated per row to generate a single entry of the results vector *b*, which amounts to computing *2*nz* floating point operations (flops[8]) per SMVM execution.

On the other hand, each multiplication will require loading 2 floating point operands (a matrix value and a value from the *x* vector) and an extra floating point operand is required for the accumulation, finally the results must be stored requiring an additional data transfer. This means that theoretically, for each nonzero entry in the sparse matrix, SMVM requires a total of 4 floating point data transfers. Now relating the useful work done per amount of data transfer, one obtains a ratio of 2/4 or one half of *flops/nz* which explains the low percentage of processor peak performance ~10% that is usually attained when computing the SMVM kernel [18].

2.7 The Conjugate Gradient (CG) Algorithm

Among modern sparse iterative solvers the conjugate gradient (CG) algorithm is one of the most popular for solving symmetric positive definite (SPD) systems

---

[7] Computational kernels can be limited either by the number of operations that can be executed concurrently called *compute bound*, or by the number of data transfers that are required to load and store the results called *bandwidth bound* (or input/output-*I/O bound*), see Kung [50] page 198.

[8] In this work the acronym *flops* will be used to refer the total number of floating point operations to compute, while *flops/s* will be used for the performance metric number of floating point operations per second.

due to its convergence properties, efficient computations and low storage requirements. The conjugate gradient method approximates the solution by constructing a Krylov subspace based on orthogonal residuals and the previous search directions which are made to be *A-conjugate*. The use of the orthogonal residuals and the *A-conjugate* search directions simplify importantly the algorithm implementation, which only requires storing a few vectors to compute successive iterates, reducing the operation count from $O(n^2)$ to $O(nz)$ where n represents the rank of a square matrix. Since the original CG algorithm proposed in the seminal paper by Hestenes and Stiefel in 1952 [58] many version of the conjugate gradient algorithm have been developed. This work uses an efficient version of the CG algorithm presented by Shewchuk in [59] as shown in Figure 6. A short explanation of the CG algorithm is presented next.

The main loop of the conjugate gradient algorithm is organized in three basic steps; the first step computes the new iterate using a line search procedure; the second step determines the new residual based on the previous residual and the projection of the previous search direction; and finally, the third step computes the next search direction making it *A-conjugate* to the previous search directions (current Krylov subspace). These are the general steps of the conjugate gradient algorithm and while they may be organized in different ways they have been presented here following the order of the algorithm in Figure 6.

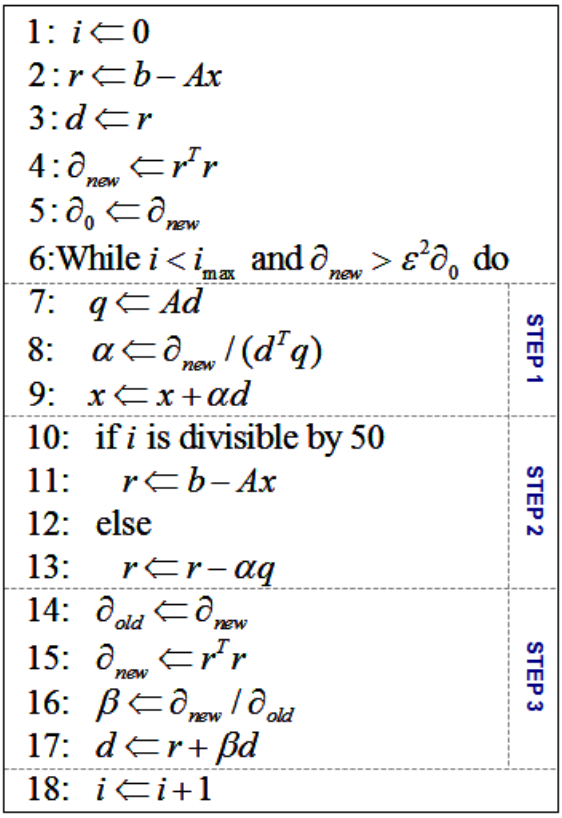


Figure 6: Conjugate gradient algorithm. Where ε is the tolerance used, α and β are the constants used to update the x vector of unknowns and the new search direction d, r is the residual vector, and ∂ and q are temporary variables.

The remainder of this section briefly analyses the main computing kernels in the conjugate gradient algorithm and its bottlenecks. Only the linear algebra operations in the *for-loop* of CG are analyzed, disregarding all scalar operations, since they make up the bulk of the computations. Three types of basic linear algebra operations are used in the main loop of the CG algorithm, particularly for the algorithm in Figure 6 these operations are: three vector updates or *saxpy* operations (in lines 9, 13, and 17) of $O(n)$ complexity, two *dot-products* (in lines 8 and 15) also of $O(n)$ complexity, and one *SMVM* operation (in line 7) of $O(nz)$

complexity. Among these BLAS operations the dominant computing kernel is the SMVM as it was pointed out in section 1.2.2 (making CG $O(n^{3/2})$ [59]); thus, the main bottleneck in CG are the same ones that were analyzed in the previous subsection for SMVM. This also justifies that most of the previous work focused on accelerating the SMVM kernel as means to accelerating CG performance.

Although the SMVM is the dominant computing kernel in CG, the other operations that represent a challenge, especially from the parallel processing point of view, are the two dot-products. The dot-products require reduction summations and become synchronization points in the CG algorithms. On the other hand, the vector updates operations can be computed very fast in an embarrassingly parallel fashion. A final observation of importance regarding CG performance bottlenecks is that CG is an intrinsically sequential algorithm. Most of the operations in CG depend on the results of the previous operation, except for current solution approximation (i.e. iterate) and residual updates (*saxpy* operations) that can occur concurrently.

Chronopoulos and Gear present a variant of CG [60] with increased data locality since the vectors are loaded only once per iterations, and only one synchronization point because the two dot-products are located in the same point in the algorithm. Moreover, the two dot-products are independent and can be computed concurrently. However, this increased data locality and reduced

synchronization point is attained at the expense of *2n* extra flops [33]. Chronopoulos and Gear further increase locality and parallelism by creating an “s” size Krylov subspace per iteration, but increases the operation count requiring an additional SMVM operation and incurring in possible instability. This instability may lead convergence towards the dominant eigenvector instead of the true solution vector as stated in [33].

2.8 Summary of Bottlenecks for SMVM and CG

Table 2 presents a summary of the main bottlenecks identified for the SMVM and the CG algorithms that have been identified in this chapter, some of which will be solved for general sparse matrices in the following chapters.

The following two chapters present new ways to accelerating the sparse matrix-vector multiplication kernel for different types of modern multicore processors using both the multiple cores available and exploiting the vector units found in them.

Table 2: Main bottlenecks for the sparse matrix-vector multiplication (SMVM) and the conjugate gradient (CG) algorithm.

SMVM
• The matrix entries have no data (temporal) reuse and little spatial locality. • Vector access is indirect and irregular. • Large instruction overhead compared to useful floating point operations. • Low flops/Data-access ratio (less than 1, thus being I/O-bound). • Performance dependence on matrix format and algorithm combination.
CG
• Intrinsically sequential algorithm. • Dot-products become synchronization points and must be parallelized carefully since they require reduction sum. • Those of SMVM.

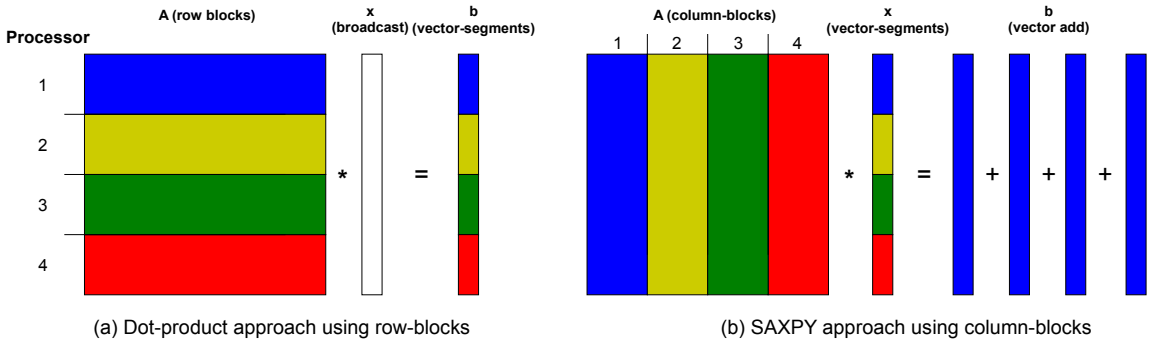
Chapter 3
Accelerating the SMVM Algorithm for Multicore Processors

This chapter presents a novel way of accelerating the sparse matrix-vector multiplication algorithm on multicore processors using short vector units (SIMD units) and multiple cores.

3.1 A Naïve Approach to Parallelizing the SMVM Kernel on Multicore Processor

Parallelism for the sparse matrix-vector multiplication operation can be implemented in different ways. Starting from the two SMVM algorithms in Figure 5 (see section 2.5) that show the traditional implementations of the SMVM kernel using the dot-product (inner-product) approach and the *saxpy* approach, we can define naïve ways to parallelize them as explained next. Two simple schemes can be used to parallelize the dot-product version (see Figure 7.a), either parallelizing the outer *for-loop* (in line 1 of Figure 5.a), which assigns a set of dot-products to each processing-core/processor-node[9]; or the inner *for-loop* (in line 3 of Figure 5.a), which would imply computing every dot-product using all processors resulting in increased amount of communication among processors leading to very poor performance.

[9] Term “processor” will be used to refer to processing-cores in a multicore system. Without loss of generality or ambiguity, this term will also be used to refer to different compute-nodes in a distributed memory system. The loose use of this term is appropriate since most of the algorithms presented can be implemented for both shared and distributed memory systems.



(a) Dot-product approach using row-blocks

(b) SAXPY approach using column-blocks

Figure 7: Two parallel approaches to the sparse matrix-vector multiplication. The example presented here assumes a multicore processor with 4 cores. The color in each block refers to the processor in charge of the computations for that block. The white vector in subfigure (a) is broadcasted across all processors. The blue vectors in subfigure (b) are summed by the same processor, thus are the same color.

Even though a choice exist, it only makes sense to parallelize the outer *for-loop* of the *dot-product* approach, since it leads to independent computations among the coloured row-blocks and yields independent segments of the results vector as shown in Figure 7(a).

An analogous situation occurs when parallelizing the *saxpy* approach. Also here the sensible choice is to parallelize the outer *for-loop* (in line 2 of Figure 5.b), assigning different column-blocks and segments of the multiplying vector to each processor as shown in Figure 7.b. The main drawback would be the amount of processor synchronization required to sum the contributions of each column-block product as they become available. Even though this parallel approach requires greater amount of synchronization than the dot-product one, it was widely used in older vector processor mainly because of the long vector

computations in the *saxpy* operations and the column data layout of matrices in the Fortran programming language.

Up until now the discussion has purposely omitted implementation details in order to keep the descriptions general, but it is now time to introduce two important considerations required for an efficient parallel implementation. The first is the choice of the sparse matrix format; although it was somewhat intuitive that the *dot-product* approach would benefit from row storage (e.g. CSR) while the *saxpy* approach would benefits from column storage (e.g. CSC). It is important to stress that the performance of the sparse matrix-vector multiplication (and other important linear algebra kernels) is directly dependent on the sparse matrix format used as mentioned in section 2.5.1 and commented repeatedly in the literature [16, 24, 28, 29, 57, 61-64] and others. Moreover, the amount of parallelism that can be efficiently exploited in SMVM varies significantly depending on the sparse matrix format used, which has lead to the various sparse matrix formats as commented in section 2.3.3. The second relevant subject is the hardware architecture used. The hardware architecture imposes additional challenges to the parallelization process of the SMVM and the sparse matrix format and algorithm used. The limiting factors of the SMVM kernel were already identified in section 2.5.1 and will be used in this section to enhance SMVM performance; however, the hardware architecture features that impose

additional performance constraints have not been reviewed, this is the subject of the next section before proposing the new optimized SMVM kernel.

3.2 Overview of Multicore Architecture and their Programming Challenges

Modern multicore processors have various architectural features that maybe exploited to enhance their performance. This section briefly describes the most important architectural features found in multicore processors and the algorithmic requirements to exploit them. The discussion will be based on two types of multicore processors (see Figure 8): homogeneous (where all processing cores have the same architectures) and heterogeneous (where there are different types of processing cores embedded in a single chip).

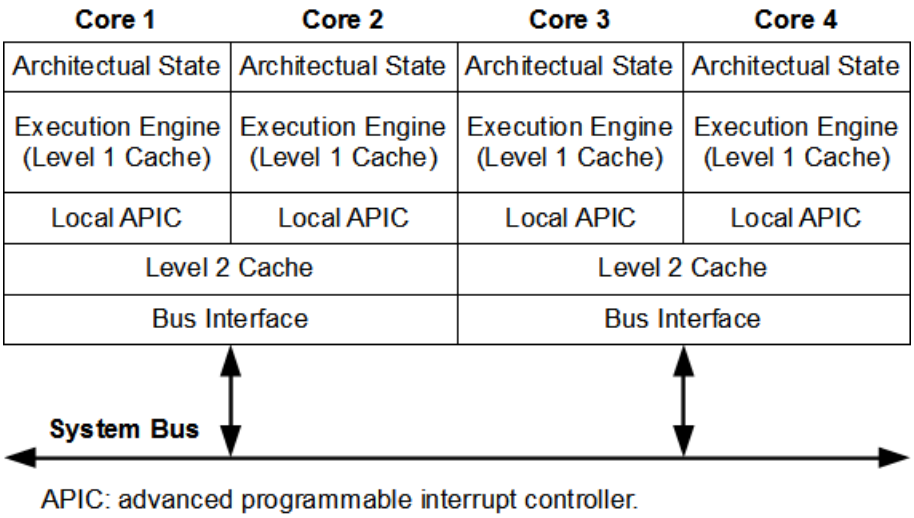
3.2.1 Architectural Characteristics of the Two Hardware Platforms Used

The first processor used in this work is the Intel Core 2 Quad (Q6600, code name Kentsfield) that is a traditional cache based architecture representative of homogeneous multicore processor. This processor contains four cores clocked at 2.40GHz with 64KB of L1 cache[10] (32KB-data/32KB-instructions), 4MB of L2 cache per core-pair, and 4GB of global DDR2 (double data rate DRAM 2) shown in Figure 8.a. The Intel Core 2 Quad processor family supports the Intel streaming SIMD extensions SSE, SSE2 and SSE3 [65]. The processor has 128-bits (16Bytes) wide SIMD units that can compute 4-way single precision floating

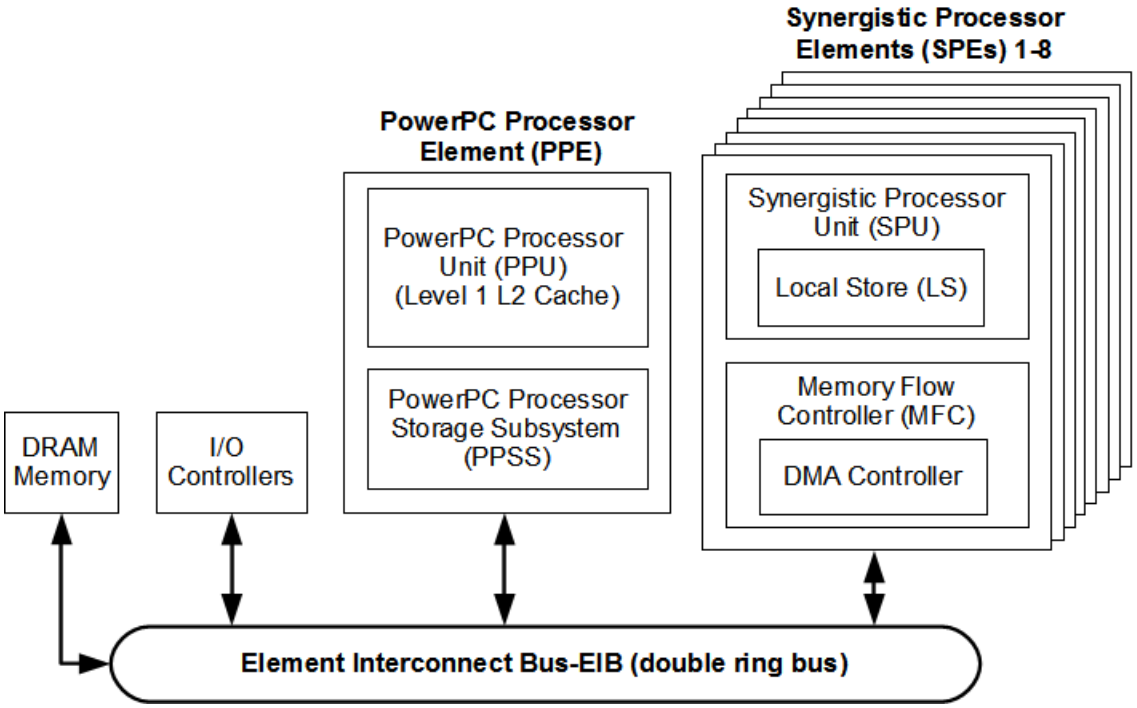
[10] Both L1 and L2 have 64B cache lines.

point (SPFP) operations or 2-way double precision floating point (DPFP) operations per SIMD instruction.

The second processor used is a simplified version of the Cell BE processor (found in the PS3) representative of heterogeneous multicore processors. This simplified Cell BE contains two distinct type of cores [12]: one PowerPC (named PPE) general purpose processor (GPP) and six SIMD processors (called SPEs) both clocked at 3.2GHz. The PPE has a traditional two level cache hierarchy (L1:32KB-data/32KB-instructions and L2:512KB both with 128-bit cache lines) whereas the SPEs have a 256KB user controlled memory (scratch-pad type memory), and both have access to a 256MB Rambus extreme data rate (XDR) DRAM global memory. In the Cell processor the PPE is commonly used for administrative and control tasks while the SPE cores are used as the main computing resource. SPEs are high performance SIMD cores with software controlled memory hierarchy, a 4-way SPFP SIMD and 2-way DPFP SIMD pipeline (128-bit wide), and limited hardware support for branch prediction. They have a large register file (128-128b registers), and a 256KB on-core software managed memory called Local-Store (LS). A distinct characteristic of the SPEs from other processors is that transfers to/from LS and main system memory must be explicitly programmed by the user, which requires some extra effort but yields more efficient memory management for predictable access patterns.



(a) Intel Core 2 Quad Processor



(b) Cell BE Processor

Figure 8: Block diagrams of the architectural features of modern multicore processors. Subfigure (a) shows the diagram for the homogeneous multicore Intel Core 2 Quad processor family [65], and subfigure (b) shows the diagram for the Cell BE heterogeneous multicore processor family [12].

3.2.2 Programming Challenges for Exploiting Architectural Features

To take advantage of the hardware features described in the previous section it is necessary to satisfy certain conditions described next.

a) Short-Vector Units or SIMD Units

SIMD units are the first architectural feature of interest. Most of modern day multicore processors have SIMD units that enable vector processing, as is the case for both of the architectures presented in Figure 8. In order to use these vector units one generally requires conforming to certain data layout and size constraints, while using specialized SIMD instructions to direct the execution to the vector units instead of the scalar units.

In the case of Intel processors, the data layout and size constraints are very relaxed thus allowing the programmer great flexibility to choose the type of data and layout that is better suited for their applications purpose while using the SIMD units. Although this is possible, the guidelines provided in chapter 4 of *Optimization Reference Manual* [66] for the IA32 architectures require data to be aligned in 16-byte (16B) memory boundaries to make efficient use of SIMD units. This can be achieved with special data structures and zero-padding techniques. For the Cell BE processor all data requires to be aligned to natural memory boundaries (1, 2, 4, 8 and 16B boundaries), but 16B memory boundaries are suggested (see chapter 19 of [12]) to maximize performance. Moreover, memory

transfers strictly require the data to be multiples of 16B (unless smaller data sets are required), which implies that all declared data must have a size multiple of 16B.

On the other hand, to enable vector execution one requires using special vector instructions. There are three ways to use these special instructions [66], the first is to explicitly program assembly code for each type of processor with the required vector instructions; the second way, is to use special high-level vector intrinsic[11] [67, 68] for the particular high level language employed; and the third, is to rely on the compiler optimizations to auto-vectorize the desired code. This last alternative is not viable for sparse matrix operations, since the compiler cannot make assumptions on data that is managed by pointers and complex data structures, which is the case for sparse matrix operations. Support for vector intrinsics is available in all modern C/C++ and Fortran compilers, and it is the one used in this work for both architectures. Intrinsics are also a portable abstraction

[11] High-level vector intrinsics (or just vector intrinsics) refer to a set of high level languages (e.g. C, C++, Fortran, etc.) instructions that are inlined to one or more assembly language instructions of a given hardware by the compiler to provide access to vector operations. In a more broader sense the term “intrinsic” or “built in functions” refers to special high-level language functions handled by the compiler in a programming language that provide access optimized code for a given operation or low-level hardware functionality that is otherwise not available. Intrinsics are commonly used to access vector operations and for parallel directives in some parallel frameworks such as OpenMP.

used by the compilers, which can select the best set of assembly instructions depending on the underlying hardware.

A final considerations regarding SIMD vectorization is that it should only be used in compute intensive sections of the code. These are found in long running code-loops that are then manipulated using loop transformation techniques [69] to vectorize the code. Common loop transformations used to vectorize scientific codes are loop-unrolling, strip-mining and loop-blocking. The first two will be explained in some detail when the SMVM vectorized kernel is presented.

Conclusion 1.a: take the time to properly design data structures considering size and alignment for SIMD processing.

Conclusion 1.b: use intrinsics to exploit vector processing whenever the compiler provides support for it.

Conclusion 1.c: use loop transformations techniques to enable vector processing.

b) Memory hierarchy

It was already pointed out in the previous section that memory alignment is key to efficient execution, but one must also consider enhancing temporal and spatial memory locality to make efficient use of data caches in cache based architecture and the local store memories in the Cell-SPEs. Little can be done with temporal locality for the matrix entries since they are only used once per

SMVM, however spatial locality can be enhanced with proper data structures to allow stride-1 access to data thus enhancing data locality. Following this line of thought, data structures should be designed as structure of arrays (SoA) instead of array of structures (AoS) as suggested in section 4.5 in [66] and chapter 22 in [12].

Conclusion 2: favour data structures that implement SoA instead of AoS to increase locality.

c) Parallel Cores

The last important hardware feature to cater for is the parallel cores in multicore processors. Since multicore processors use a similar shared memory model as the one used in older symmetric multi processors (SMPs), a straight forward approach would be to launch as many working threads as parallel cores in the target processor and to schedule the workload to different threads. The main concerns here are:

- How to partition the data and schedule it across different cores.
- How to balance the workload.
- Programming efficient parallel algorithms to manage the parallel work keeping to a minimum the parallel book keeping.

Programmers often rely on parallel shared memory frameworks such as OpenMP [70] to take care of these issues. Such approaches usually work well

when dealing with dense matrix systems, but for sparse matrix computations where complex sparse formats and pointers are required this is not feasible, hence one must directly use the threaded libraries available in each system (e.g. Posix threads [71], Intel Threading Building Blocks (Intel TBB) library [72], and AMD x86 Open64 Compiler Suite [73]).

Conclusion 3: need to use lower level threaded libraries to reduce parallel overhead and deal with complex data structures.

A summary of the most important challenges for parallelizing scientific kernels (e.g. SMVM, CG, etc.) on multicore processors are presented in Table 3.

3.3 Putting it All Together: A New Sparse Matrix Format and SMVM Kernel for Parallel Multicore Computing

Implementing an efficient sparse matrix-vector multiplication kernel requires designing a sparse matrix format and an algorithm that takes into account the bottlenecks of the SMVM operation summarized in Table 2, and the programming challenges imposed by the hardware summarized in Table 3. This section first presents the new sparse matrix format designed for this purpose (namely the pipeline-matched sparse matrix format, or PMS, the first contribution of this work) and then explains the new algorithm implemented.

Table 3: Programming challenges for implementing scientific kernels (e.g. SMVM) on modern multicore processors.

Architectural features	Programming challenges
Vector processing (SIMD)	• Memory alignment and sizes (multiples of 16B). • Use of vector intrinsics whenever available. • Employ loop transformations.
Memory hierarchy	• Use SoA data structures.
Parallel cores	• Explicit threading using thread libraries or frameworks.

3.3.1 The New Pipeline-Matched Sparse (PMS) Matrix Format

The main objectives of designing a new sparse matrix format was to exploit short vector (SIMD) units found in modern processors, while offering opportunities to easily partition and distribute data across the multiple cores. The design of the new format was also motivated by the fact that traditional sparse matrix formats do not take into account the programming challenges imposed by the hardware architecture (e.g. data partitioning on SIMD boundaries, and memory alignment requirements). Moreover, traditional vector formats such as ELL [30, 64] (Ellpack/Itpack) and JDS [30] are inefficient for a wide variety of sparse matrices where the nonzeros per row may vary significantly as mentioned in Chapter 2, which usually occurs in the finite element method when doing mesh (*h*) refinement, interpolation function (*p*) refinement or hybrid refinement (that combines both *h* and *p* refinements) commonly referred to as *hp-refinement*. Thus an additional objective of designing a new sparse matrix format was to

make it tolerant to large variations in non-zeros per row for general sparse matrices independent of their structure. The final idea that was kept in mind to design the sparse format is that it is easier and more efficient to compute on a regular kernel (e.g. dense matrix-vector multiplication) than on a sparse kernel, thus it would be desirable to produce a sparse format that can be treated as a dense kernel.

Following these directives a new format called pipeline-matched sparse (PMS) format was created. PMS is based on the Compressed Sparse Row (CSR) format, and it comprises four vectors (see Figure 9) as follows:

(i) A_VAL: the first vector stores the nonzero elements of the sparse matrix, with zero padding by rows to match the SIMD pipeline-width of the target processor (i.e. 16Bytes for both the Intel and the Cell BE processors).

(ii) AJ: the second vector contains the column indices of the nonzero elements.

(iii) SUB_ROWS: the third vector stores the number of sub-rows of size equal to the size of the SIMD pipeline-width per matrix row.

(iv) X_VAL: the last vector contains the elements of *x-vector* indexed by column indices in AJ.

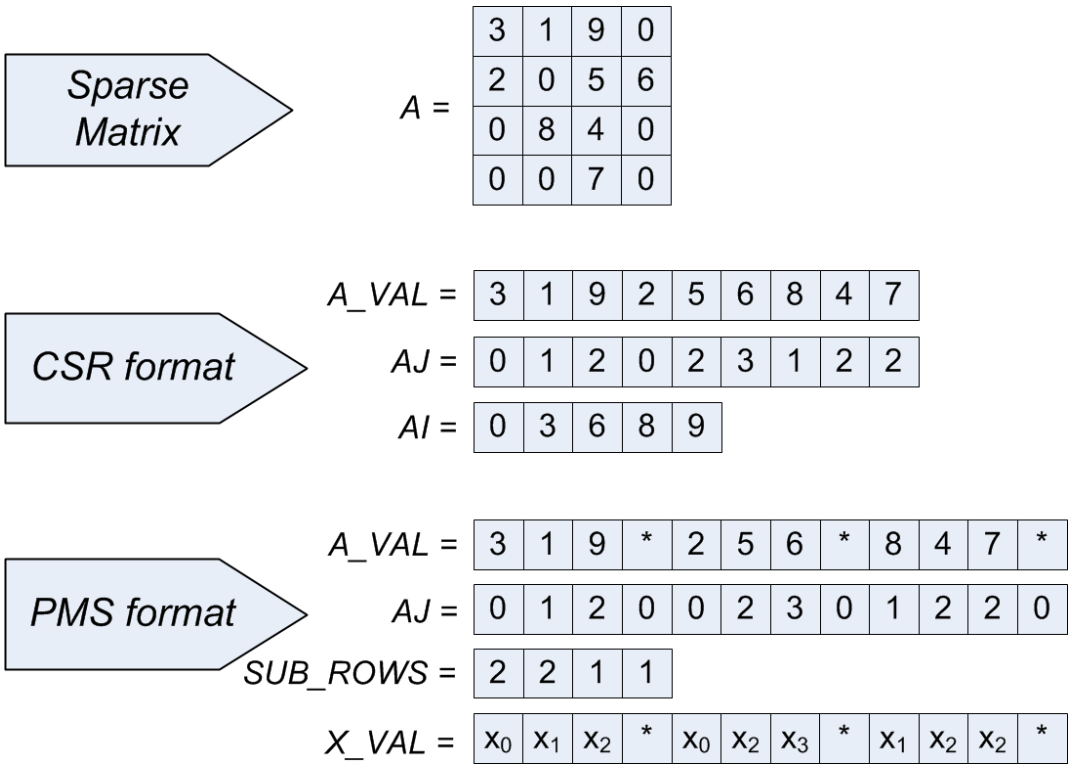


Figure 9: Representation of a sparse matrix in CSR format and the new Pipeline-Matched Sparse (PMS) representation assuming a vector pipeline with of two floating point numbers.

Once this representation is built, only the fourth vector (*X_VAL*) need be modified to solve for different *x-vectors*. The mapping of the *x-vector* elements into the new format (called vector-spreading operation) involves extra processing, but ultimately this work has to be done in the SMVM kernel regardless of the sparse format used. By doing this work in advance memory access patterns become regular (unit-stride access to the *x-vector* elements is achieved), offering better spatial locality and reduced cache misses on GPPs.

The proposed pipeline-matched sparse representation renders three important benefits:

- **Enables flexible configuration to exploit SIMD units (low-level parallelism):** by using zero padding the PMS format can be customized for different SIMD pipeline-widths (e.g. 4 single precision floating point values per SIMD register, or 4-way pipeline in the Cell processor, or the new 256-bit wide AVX extension in the Intel Core i5, i7 family – see section 5.13 in [65])) and efficiently exploit available processor parallelism. In the rare case where no SIMD units are available, PMS can be configured with a 1-way SIMD size resulting in no zero padding.
- **Enhances spatial locality and regularity in data access patterns:** the irregular-indirect access to the *x-vector* is solved by mapping the *x-vector* into the PMS format *apriori* regularizing the data access pattern. This allows a regular stride-1 access to all data required in the SMVM kernel, and thus creating a “dense-type” kernel with a more efficient execution even when considering the extra operations required by the zero-padding.
- **Provides natural boundaries for data partitions (high-level parallelism)**: the zero padding used to generate SIMD vectors in each of the matrix rows also serves the purpose of defining natural partition boundaries on the

vector boundaries defined. These boundaries can be used to exploit parallelism across CPU cores, and assure that matrix data is kept aligned.

Compared to the CSR format, PMS requires the storage of an extra floating point vector of size equal to the number of nonzeros, in addition to the zeros used to match the pipeline-width of the target architecture. But this extra memory usage yields benefits in terms of easier data (matrix and vector) partitioning and subsequent data communication to the parallel computing cores, as well as a regular computation. Also, because this format already contains the *x-vector* there is no need to transfer it separately; in fact the amount of data transferred to the processing cores is similar to the amount required by the CSR format. Only three of the four vectors need be transferred to compute the SMVM kernel using PMS: A_VAL, SUB_ROWS, and X_VAL; whereas CSR requires an additional vector (the column indices).

PMS can be thought of as a compressed vector storage of the sparse matrix, with vectors-sizes of the target architecture pipeline-width. Whilst this format was designed within the scope of FE applications, it can be used to represent other sparse matrix types regardless of their sparsity pattern, density, symmetry, or target application since the amount of zeros added is kept to a minimum by doing it row-wise. This new representation could also be used in non-conventional multicore architectures or reconfigurable hardware providing similar benefits.

A special data structure called `c_array_t` was defined to represent all the sparse matrices and sparse matrix formats used in this work, which is described in Appendix B. This data structure was designed to accommodate many different sparse (and dense) matrix formats and to facilitate aligning data to natural memory boundaries (power of two memory addresses).

3.3.2 Vectorizing the SMVM Algorithm with PMS

The dot-product version of the SMVM algorithm for the CSR format (presented in section 2.5 and reformatted in Figure 10.a for the PMS format) is used here as the basis for vectorization. This version was selected since it assumes a row ordered matrix format (as is the PMS format presented in the previous section), and because the parallelization of this version will not require any synchronization for parallel processing as shown in section 3.1.

The first step to vectorize the SMVM code is to apply a loop transformation technique called *strip-mining* [63, 66] (also called “loop sectioning”) to the inner loop (dot-product loop) of the algorithm (see Figure 10.b). This technique creates strips or segments within a loop, where the strip size usually matches the desired vector (SIMD) size or smaller so that it can be vectorized. The control variable of the loop is then incremented by the size of the strips. This technique also enhances data locality and reduces the conditional evaluation (branching) overhead. The way *strip-mining* is applied in this work is effectively the same as

another loop transformation technique called loop unrolling[12] (or unwinding), as referred in [74, 75]; thus, the two terms will be used interchangeably even though in other contexts these techniques might differ slightly.

Next, the instructions in the stripped loop are vectorized using compiler intrinsics as shown in Figure 10.c for the Intel processor. The intrinsics for the Cell BE processor vary but the implementation procedure is the same. The remaining computation is to reduce the four elements accumulated from the vectorized dot-product into a single scalar result, which is done in Figure 10.d using a tree reduction procedure. The vectorization procedure presented in Figure 10 was easily done because the PMS format was configured to generate 4 SPFP sub-rows for the sparse matrix, thus conforming to the 128-bits SIMD units assumed. These optimizations and the forthcoming ones (related to multicore parallelism) have been implemented in a new sparse matrix library that is briefly described in Appendix B.

[12] Loop unrolling is a loop transformation technique used to reduce the branch overhead in loops, increase the locality and instruction level parallelism inside the loop, thereof improving the instruction scheduling and overall performance for long running loops (see section 2.2 in [66]).

```
1:for(i=0;i<rows;i++) {
2:   b[i]=0.0;
3:   start=stop;
4:   stop+=SUB_ROWS[i+1]*vector_size;
5:   for(j=start;j<stop;j++) {
6:     b[i]+=A_VAL[j]*X_VAL[j];
7:   }
8:}
```

Note: "vector_size" is 4 in this example.

(a) Dot-product SMVM version using PMS

```
1:for(i=0;i<rows;i++) {
2:   start=stop;
3:   stop+=SUB_ROWS[i+1]*vector_size;
4:   segment[0:vector_size]=0.0;
5:   for(j=start;j<stop;j+=vector_size) {
6:     for(k=0;k<vector_size;k++) {
7:       segment[k]+=A_VAL[j+k]*X_VAL[j+k];
8:     }
9:     b[i]=segment[0]+segment[1]+
           segment[2]+segment[3];
10:}
```

(b) Step 1: Strip-mining or loop unrolling.

```
1:for(i=0;i<rows;i++) {
2:   start=stop;
3:   stop+=SUB_ROWS[i+1]*vector_size;
4:   res_acc_V= _mm_setzero_ps();
5:   for(j=start;j<stop;j++) {
6:     mat_data_V= _mm_load_ps(A_VAL[j]);
7:     vector_data_V= _mm_load_ps(X_VAL[j]);
8:     temp_mul_V= _mm_mul_ps(mat_data_V,
                              vector_data_V);
9:     res_acc_V= _mm_add_ps(temp_mul_V,
                              res_acc_V);
10: }
11: b[i]=res_acc_V[0]+res_acc_V[1]+
         res_acc_V[2]+res_acc_V[3];
12:}
```

(c) Step 2: Vector intrinsics (example for Intel processor).

```
1:for(i=0;i<rows;i++) {
2:   start = stop;
3:   stop += SUB_ROWS[i+1]*vector_size;
4:   res_acc=_mm_setzero_ps();
5:   for(j=start;j<stop;j++) {
6:     mat_data_V = _mm_load_ps(A_VAL[j]);
7:     vector_data_V = _mm_load_ps(X_VAL[j]);
8:     temp_mul_V = _mm_mul_ps(mat_data_V,
                               vector_data_V);
9:     res_acc_V = _mm_add_ps(temp_mul_V,
                               res_acc_V);
10: }
11: res_acc_V = _mm_hand_ps(res_acc_V,
                            res_acc_V);
12: res_acc_V = _mm_hadd_ps(res_acc_V,
                            res_acc_V);
13: _mm_store_ss(b+i, res_acc_V);
14:}
```

(d) Step 3: Tree reduction.

Figure 10: Strip-mining or loop-unrolling of the inner loop in the dot-product version of the SMVM algorithm for a strip-size of 4 using the PMS format configure for 4-SPFP vector units (128-bit wide SIMD units).

3.3.3 Scheduling Multiple Cores with the Vectorized SMVM

For homogeneous multicore processors (such as the Intel Core families) dividing the workload of SMVM for multicore processing can be done easily by

partitioning the matrix into row-blocks and assigning them to different processing cores as illustrated in section 3.1. The main concern while doing this is to properly load balance the work of each core. The row-block nonzeros can be balanced by using the SUB_ROWS vector in the PMS format to compute the number on matrix elements in each row-block. Balancing the load in this manner requires a simple but efficient algorithm with $O(n)$ complexity. Once the row-blocks partitions are defined, indices for the matrix entries, x-vector entries (in the PMS format) and SUB_ROW fields are sent to each processing cores and the SMVM computation may begin.

The multicore parallelism used for the heterogeneous processor such as the Cell BE requires a slightly different approach, since the small memories in the SPEs (its main processing cores) require explicit memory transfers between main memory and SPE's local-store (LS). A two-level partitioning scheme was designed to distribute data in shared memory multicore architectures taking into account the limited memory space available in the SPEs. The data partitioning scheme was developed to generate *coarse-grained* (1st level – row-block partitions) and *fine-grained* (2nd level – buffer partitions) partitions on the sparse matrix as shown in Figure 11. The objective of the coarse-grained partitions is to schedule and load-balance row-blocks across parallel cores. The number of rows assigned to these coarse partitions (row-blocks) is set to have a uniform

distribution of matrix nonzeros per row-block; whereas, fine-grained partitions are used to determine the number of matrix chunks to stream within each processing core.

The fine grained partitioning is used to cope with the limited memory in the parallel processing cores (cache for GPP cores or LS in the SPEs) and can be viewed as the cache blocking techniques used to reduce the effect of cache misses in GPPs. This second partitioning is also key to applying streaming techniques that enable overlapping communication with computations on the Cell-SPE cores as illustrated in Figure 12. The technique used to stream data to the SPUs is called multi-buffering. The multi-buffering implementation done for the Cell-SPE uses two input and two output buffers (also called double-buffering), where the main idea is to transfer data to one of the input buffers while useful work is being done on the other one. Similarly, results are written to an output buffer, while the previously computed results in the alternate output buffer are sent back to global memory. As an added benefit of the PMS format, there is no need to partition the *x-vector* separately since it is already contained in the proposed format and thus uses the same partition boundaries as the matrix nonzeros.

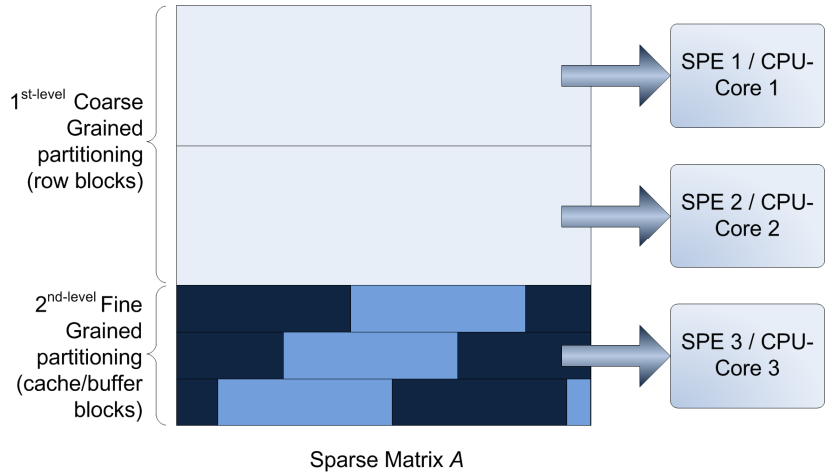


Figure 11: Two-level partitioning scheme of a matrix. Coarse grained partitions generate row blocks, and fine grained partitions create smaller data sets to transfer in a block fashion.

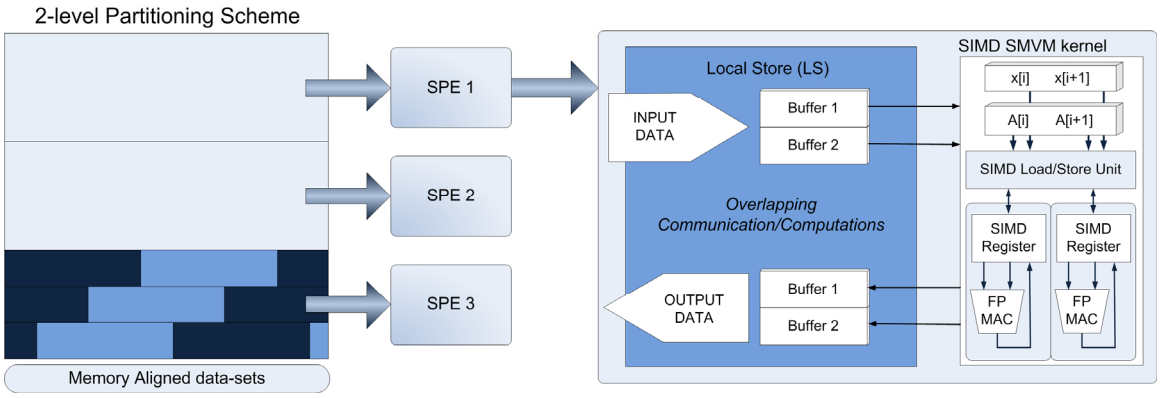


Figure 12: Double-buffering implementation for the Cell-SPEs. The fine grained partitions are defined to be the size of a single buffer in the SPEs.

It is worth noting that the 2-level partitioning scheme proposed here is flexible, thus different partitioning schemes can be used for the two partition levels. Moreover, once the coarse-grained partitions are defined the fine-grained

partitions in different row-blocks can be configured independently with different schemes if so desired.

Overall, this partitioning scheme provides good load balancing for shared memory architectures. However, when clusters of these multicore processors are considered a more sophisticated coarse grained partitioning scheme might become necessary to minimize communications between multicore chips. Important studies on sparse matrix partitioning based on graph and hypergraph methods with precise estimation of communication volume are presented in [76-79]. The study of these methods will be important when implementing efficient SMVM operations on massively parallel multicore clustered systems which will be the subject of future work.

3.4 Experimental Results

To examine the performance of the new PMS representation and partitioning scheme the SMVM kernel was implemented using the two processors described in section 3.2.1. The Cell processor heterogeneous multicore processor was installed with a 64-bit Fedora Core 6 Linux operating system and was programmed using the Cell SDK version 3.1. The Intel homogeneous multicore processor was installed with a 64-bit Fedora Core 7 operating system. Both a reference CSR SMVM (referred to as SMVM-CSR) and the PMS version (called SMVM-PMS) were implemented for validation and comparison purposes.

3.4.1 Experimental Setup

The SMVM algorithms developed for the two architectures where compiled using GCC 4.1.2 using “-O2” and “-O3” compiler flag. The Cell-SPE accelerated version of the SMVM kernel was implemented using specific vector intrinsics [12] for the SPEs. The Intel compiler collection version 11.0 was also used for the Intel processor to take advantage of high performance vector intrinsics available for this processor. All times were taken using the Linux `gettimeofday` function, and only the best performance results are shown in each case for the different combination of optimization flags and compilers used.

The vectorized algorithm for the PMS format on the Intel processor was already shown in Figure 10.d; the implementation for the Cell BE processor required a similar process in addition to the memory transfers from global memory to the local memory of each SPE. This mainly involved intrinsics to control the asynchronous DMA transfers between SPE LS and the Cell main memory; and specific intrinsics to perform SIMD multiplications and additions on 4 SPFP elements simultaneously, thus capitalizing the 4-way SIMD pipeline in the SPEs. The PPE was used to create the SPE threads and schedule the work to be done.

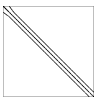
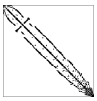
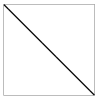
For both architectures the PMS format was configured to generate 4-SPFP sub-rows (per matrix row) matching the SPE and SSE3 pipeline width. To

minimize the overhead effect of the SMVM's control statements in the Cell-SPEs, simple conditional instructions were substituted with bit-selection intrinsics, thus eliminating the corresponding branch occurrences in the code. Whenever this was not possible, branch hint instructions were used to reduce the impact of misprediction latency.

3.4.2 Test Results

A set of finite element matrices with varying sizes and different sparsity patterns taken from the Matrix Market repository [42] are used to study the performance and scalability of the new approach (format and algorithm) presented. These matrices are shown in Table 4 ordered by increasing number of nonzeros. The first set of results (see Figure 13) present the speedup (SU) of the SMVM-PMS kernel with respect to a reference SMVM-CSR implementation, and serve as proof-of-concept to show the effectiveness of the proposed approach. The comparison is done for each hardware platform independently using a single computing core with either the CSR format and the classic algorithm (presented in Figure 5) with compiler optimizations, or the PMS format with the vectorized kernel shown in Figure 10.d. The speedup is computed as the wall-clock time ratio of the SMVM-CSR to SMVM-PMS execution times, using average times from 1000 runs for each kernel.

Table 4: Finite element (FE) test matrices from the Matrix Market repository [42]. The matrices are square with number of rows and columns equal to the matrix rank (in column 3). The number of nonzeros (NZ) are shown in column 4, column 5 has the percentage fill of the matrix with respect to the dense case, the total nonzeros with padding are shown in column 6, and column 7 contains the percentage of added nonzeros.

#	Name	Rank	NZ	Sparsity %	NZ with padding	% added zeros	Sparsity pattern
1	can___24	24	160	27.78%	208	30.00%	
2	cavity26	4562	138187	0.66%	144148	4.31%	
3	e40r5000	17281	553956	0.19%	578312	4.40%	
4	fidapm37	9152	765944	0.91%	781100	1.98%	
5	s3dkq4m2	90449	4820891	0.06%	5001068	3.74%	

The speedup results shown in Figure 13 demonstrate that the proposed vectorization using the PMS format outperforms the automatic vectorization possible with compiler options using the classic CSR. It is important to observe that these techniques are useful for moderately small to large matrices, whereas for smaller matrices little or no performance benefit can be achieved mainly due

to the effective caching of the matrices and vectors when using the CSR format, as evidenced with matrix (1) for the Intel processor.

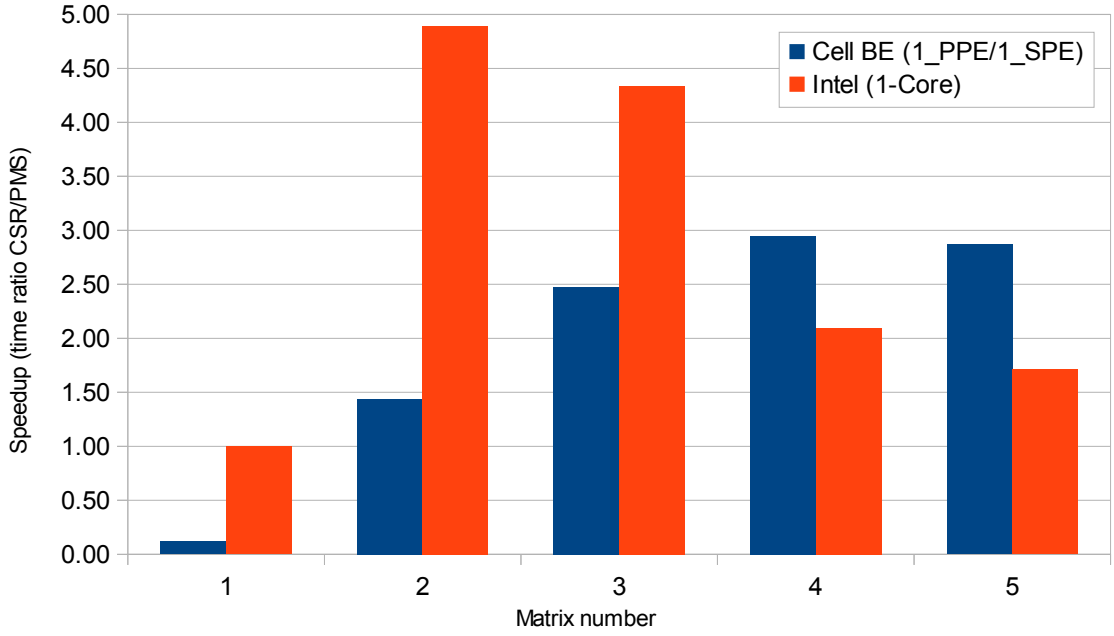


Figure 13: Speedup results presented as the time ratio of SMVM-CSR/SMVM-PMS for each hardware platform.

On the other hand, the results for matrix (1) using the SMVM-PMS in the Cell BE processor actually slow down computations. This is attributed to the fact that the CSR version runs in the PPE and only access the Cell global memory; while the PMS version that runs on an SPE, requires to explicitly transfer all matrix and vector data from global memory to SPE local-store (LS) using the double-buffering technique described in the previous subsection. When the data to transfer is small (either fits in one or a few buffers) there is not sufficient time to

overlap communication with computations and the double-buffering technique is rendered useless.

Performance results in MFlops/s (million of floating point operations per second) for single core implementations of the SMVM-CSR and SMVM-PMS kernels are shown in Figure 14. In this figure it is also evidenced that the vectorized SMVM-PMS kernel has better performance than the SMVM-CSR for a single core in each processor, except for matrix (1) for in both processors for the reasons mentioned earlier. It is also interesting to observe that as the matrix size (nonzeros) grows the performance of the SMVM-PMS for the Cell-SPE grows (referred to as CELL-PMS in Figure 14), which is explained by the better overlapping of computations and communications with the double-buffering technique used. On the contrary, cache based architectures like the Intel processor suffer from more cache misses from the bigger data sets, which is reflected as a decrease in performance for the SMVM-PMS kernel on the Intel processor (referred to as Intel-PMS in Figure 14). Of course, performance will also vary depending on the zero padding done, which will be presented next.

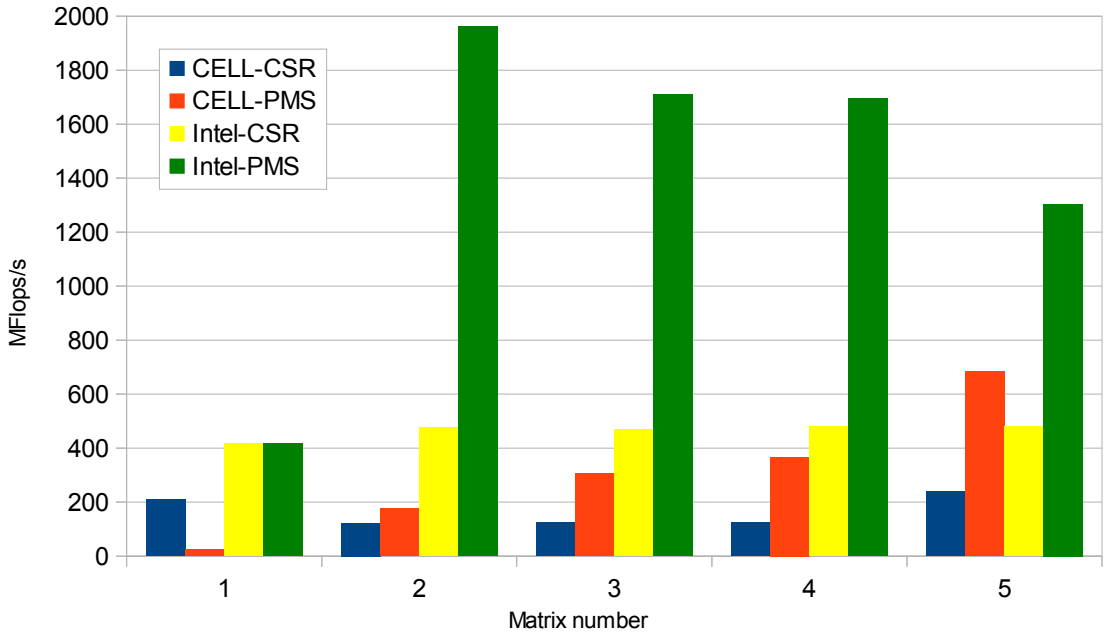
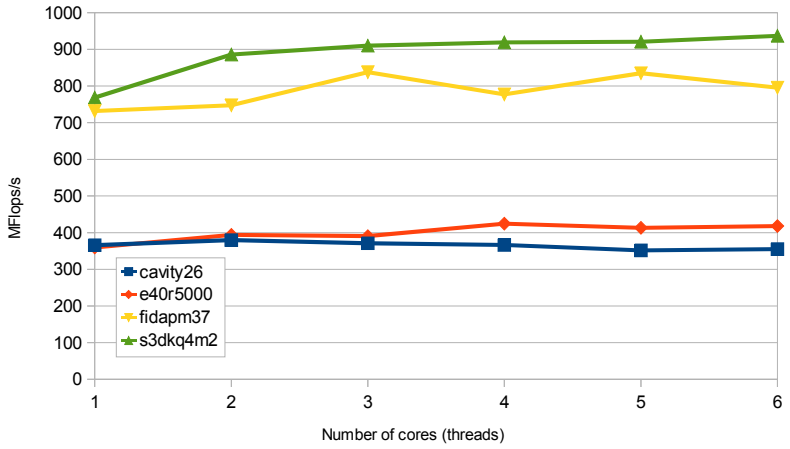


Figure 14: Performance in MFlops/s of the SMVM kernel for the CSR and PMS formats.

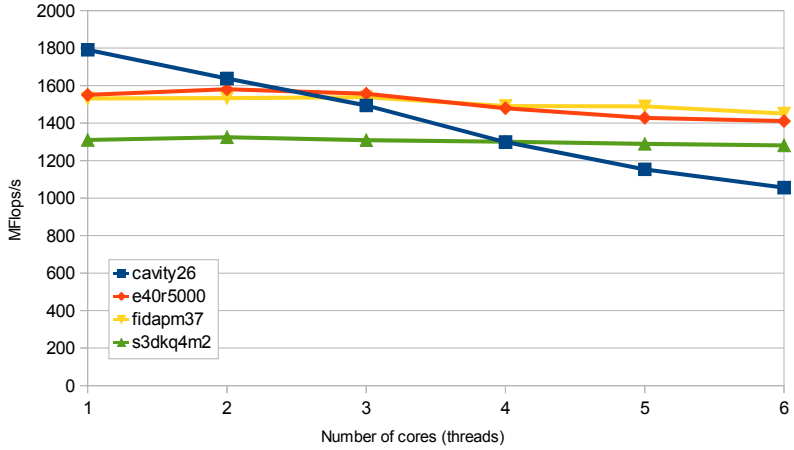
The scaling performance results in MFlops/s for the best performing CSR kernel and the PMS format for the Intel and Cell BE processors are presented in Figure 15. These results are only for the matrices (2-5) since matrix (1) is too small to do any efficient parallel work. The performance of matrix (2) does not scale for any of the formats and platforms used because of its size. Considering that the SMVM kernel is a I/O-bound algorithm (bandwidth-bound kernel as mentioned in Chapter 2), the performance scaling from the CSR implementations and PMS on the Intel processor are expected to be very poor, see Figure 15.a and Figure 15.b. Only marginal performance benefits are obtained for these cases up to 2-cores, since the Intel Core 2 Quad processor shared the two L2 caches per core-pair; thus, after increasing the thread count over 2, more cache

misses occur and no further performance gain is possible. Even so, it is also clear that the SMVM-PMS outperforms the SMVM-CSR for all cases ranging for 4.9X speedups for the smallest test matrix to 1.6X for the largest test matrix.

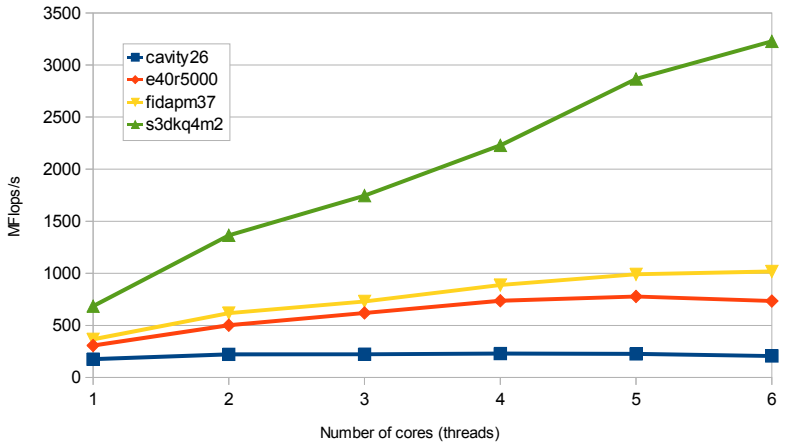
A better behaviour is observed for the Cell-SPEs, where the performance increases with the matrix size obtaining up to 3.2 GFlops/s for the largest test matrix outperforming that of the Intel CPU. This is mainly attributed to the multi-buffering technique used, which effectively overlaps computations with communications as shown in Figure 15.c. For the largest test case (~4.8 million nonzeros) the Cell-SPE SMVM kernel is 3.5X faster than the Intel SMVM-CSR, 2.6X over Intel SMVM-PMS, and nearly 14X faster than the SMVM-CSR implementation in the Cell-PPE (see speedup results for the biggest test matrix in Table 5), exhibiting a superlinear speedup compared to the Cell-PPE version. Although this type of performance is in agreement with the results presented in [23], the main benefit from the results presented in this dissertation come from the vector processing and not from the cache-blocking or register-blocking techniques used in [23].



(a) CSR



(b) PMS-Intel



(c) PMS-SPE

Figure 15: Performance scaling results of the SMVM kernel for the Cell BE and the Intel processor using the CSR and PMS sparse formats.

Table 5: SMVM speedup scaling using different formats for the "s3dkq4m2" matrix (5).

Cores (threads)	PMS-Intel / CSR-Intel	PMS-SPE / CSR-Intel	PMS-SPE / PMS-Intel	PMS-SPE / CSR-PPE
1	1.66	0.89	0.54	2.87
2	1.44	1.54	1.07	5.72
3	1.40	1.92	1.37	7.32
4	1.38	2.42	1.76	9.34
5	1.37	3.11	2.28	12.01
6	1.33	3.45	2.58	13.53

The last analysis done is related to the scalability of the PMS format itself. The question here is how the zero padding scales with the matrix size? The answer to this question depends on two factors: the size of the vector-pipeline being matched, which affects the amount of zero-padding performed per row; and the distribution on nonzeros in the matrix that determines the number of nonzeros per row and thus the padding required.

For example, in the test cases presented in this chapter the PMS format was configured to match a 4-SPFP pipeline found in both the Cell BE and Intel CPU was shown to scale gracefully with the matrix size. This is evidenced in Table 4 where the effect of the zero-padded is less significant for bigger matrices (that was always less than 5% of the real nonzeros in the matrix) than for small matrices where it can be significantly higher. Another way to estimate the impact of the zero-padding for a specific matrix would be given by the following formula:

$$\%pad = (rows * (vectorSize - (avgRowSize \bmod vectorSize))) * 100 / NZ \quad (2)$$

Here the *%pad* refers to the percentage of padded zeros, the *vectorSize* is the size of the vector-pipeline to match, *avgRowSize* is the average row size of the matrix, and *NZ* represents the real nonzeros in the matrix. If the *avgRowSize* of the matrix is not known, an upper-bound for the added nonzeros can also be estimated as follows:

$$MaxPad = numRows * (vectorSize - 1) \quad (3)$$

It is worth noting that for modern multicore processors and even parallel manycore GPUs the *vectorSize* will always be small (usually 4 for SPFP, 2 for DPFP, or 8 for SPFP and DPFP in GPUs), thus matrices with average nonzeros per row *equal-to* or *bigger-than* these values will not suffer from large zero padding. Nonetheless, the results presented in this section show that the zero padding done for the PMS format will render better performance for large matrices than traditional CSR formats.

3.5 Concluding Remarks

This chapter presents the first two contributions of this work, a new sparse matrix format called pipelined-matched sparse representations or PMS (using the three character convention discussed in chapter 2) and a 2-level partitioning scheme with a modified SMVM algorithm (first presented in CEFC 2008 [80] and

published in 2009 [81]) that render the following benefits for SIMD processing of the SMVM kernel in multicore environments:

- Enable exploiting short-vector (SIMD) processing units regardless of the sparse matrix nonzero pattern, and adaptable to different vector (SIMD) pipeline sizes depending on the target architecture.
- Enhance spatial locality in the matrix entries.
- Alleviate the indirect and irregular vector access of the SMVM kernel by creating a “dense-type” approach to solving a dense problem.
- Facilitate data partitioning, distribution, and load balancing.
- Limits the amount of data transfers and instruction overhead.
- Good scaling behaviour of the PMS format with the matrix size. The percentage of padded zeros tends to reduce as the matrix size grows.
- The SMVM Cell-SPE accelerated kernel was on average 3.5X faster than the Intel SMVM-CSR, 2.6X over Intel SMVM-PMS, and nearly 14X faster than the SMVM-CSR implementation in the Cell-PPE for the largest test matrix.

The PMS format exhibits excellent scaling behaviour for the Cell BE platform (streaming type processors) where the multi-buffering technique was applied. On the other hand, even though the performance of the PMS-SMVM kernel in the cache based architecture (Intel CPU) always outperformed that of the SMVM-

CSR, the scalability suffered for bigger matrices because of the increased cache misses. All of the optimizations presented in this chapter and the forthcoming ones have been included in a sparse matrix library described in Appendix B. The next chapter presents a solution to the scaling difficulties that PMS presents for cache based architectures.

Chapter 4
Blocked PMS Format

The scaling behaviour of sparse matrix formats is an important concern for parallel processing. The previous chapter introduced a new sparse matrix format called *pipeline-matched sparse* (PMS) representation that enabled efficient vector processing for SMVM while demonstrating good performance scalability for the memory controlled Cell-SPEs, but that did not scale well for the cache based Intel processor. This chapter introduces a new sparse matrix format better suited for cache based architectures and shows the performance benefits of the new sparse formats for the conjugate gradient method.

4.1 Blocking PMS for the SMVM Kernel

In this section, first the new blocked sparse matrix format is introduced and then the corresponding blocked algorithm is presented.

4.1.1 The Blocked-Pipeline-Matched (BPMS) Sparse Matrix Format

The new format called *blocked-pipeline-matched sparse* (BPMS or BPS) representation is the third main contribution if this work. As in PMS [81], BPMS defines clear data boundaries for partitions, nonetheless it also offers better opportunities to exploit fine grained parallelism and it does not require the vector-spreading operation. In BPMS the matrix is stored in small dense matrix-blocks, which are enforced to be a multiple of the vector-registers size on the target

architecture (e.g. 128-bit register that can store 4 single precision floating point values or 2 double precision in modern Intel Core2 CPUs) as in PMS; thus allowing to easily exploit short-vector (SIMD) units in multicore processors. Furthermore, when the block size is a multiple greater than one of the vector registers, other loop transformations can be implemented to enhance performance (not possible on PMS). BPMS stores the matrix data in four linear arrays in the following way:

(i) A_VAL: stores the nonzero elements of the matrix in dense square/rectangular blocks (elements in blocks are stored row-wise), with zero padding to match the pipeline width of the target processor per row (as in PMS).

(ii) AJ: contains the column indices of the first element in each block (as in BCSR).

(iii) AI: has the index of the first matrix element that starts a new row-block (as in BCSR).

(iv) Blocks per row-block: the number of dense blocks preceding the block pointed to by each of the row indices.

Because of the increased data locality gained by creating the blocks within the matrix format there is no longer the need to include the *x-vector* in the BPMS format; instead, BPMS can now rely on an efficient access of the *x-vector* data

from the underlying cache hierarchy in cache-based processors, while avoiding the time required to spread the x-vector into the format. However, the drawback is that the SMVM-BPMS kernel depends on the matrix structure to be efficient.

Even though BPMS format does not include the *x-vector* it can easily be included in the format using a fifth vector. The memory impact of such addition would be minimal relative to PMS since each block would use the same *x-vector* segment for all of its rows, increasing not only spatial but also temporal locality for the access to this vector. Moreover, the cost of the vector spreading operation would also be minimized since each block only requires to map the x-vector for the first row and all other nonzero elements (in the other rows inside the block) would reuse the vector-segment as mentioned above. The negative side is that depending on the structure of the sparse matrix there might still be a lot of *x-vector* data repeated in the BPMS format; therefore in this work it will not be included in the BPMS format. The converse would also apply to the PMS format, so PMS could be represented without embedding the *x-vector* into it, which would yield similar benefits to the ones presented for BPMS.

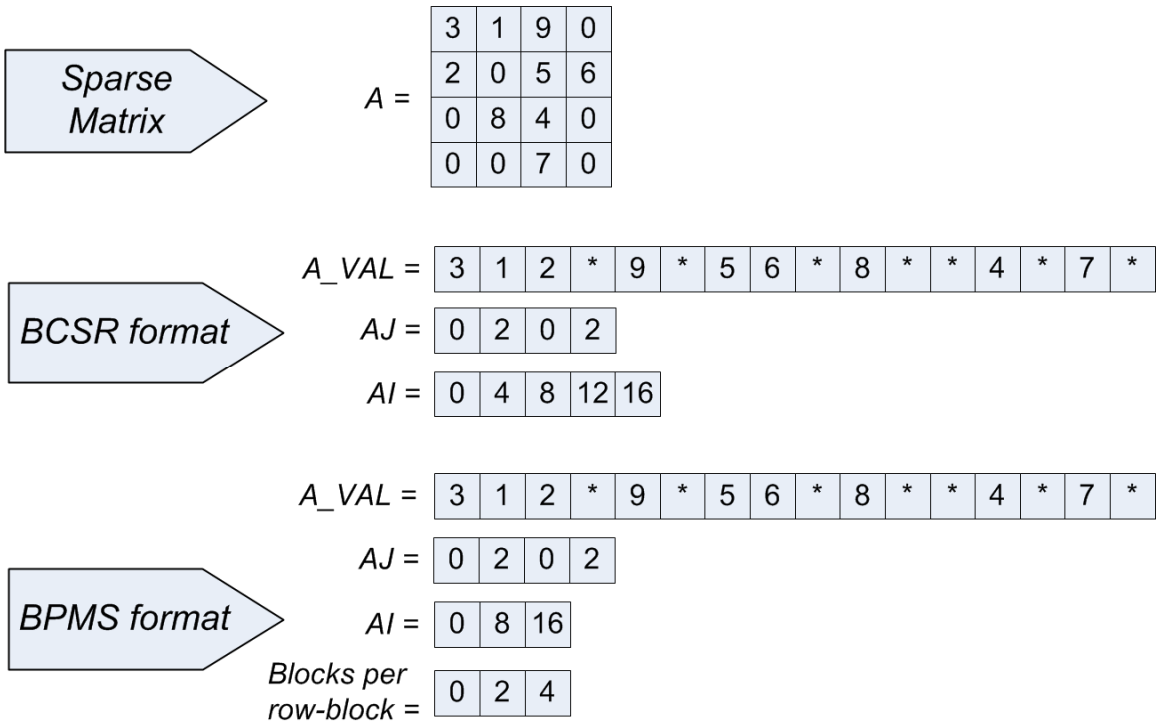


Figure 16: Representation of a sparse matrix in BCSR format and the new block pipeline-matched sparse (BPMS) representation

Recalling the example matrix presented in section 3.3.1, Figure 16 shows this same matrix in the *blocked pipeline-matched sparse* (BPMS) representation and the *block CSR* (BCSR) representation for comparison. Both BPMS and BCSR formats have been configured with the same block sizes for this example for simplicity. In reality the block size for BCSR is determined based on the desired cache blocking or register blocking techniques, which does not consider the size of the vector-register in the SIMD units in the target processor. Common benefits of BPMS with respect to other blocked formats for the SMVM operation include:

- Increased spatial data locality for the vector access.
- Enabling efficient loop transformation techniques that decrease the loop iteration count and thus reduce the instruction overhead from the constant evaluation of the conditional statements (less branching).
- Compressing the row and column index information.

The blocked pipeline-matched sparse format has similar advantages to those of the PMS format except for the direct access to the x-vector that is not included in BPMS. The main advantage of BPMS over other blocked formats such as BCSR is that it exploits the vector units in modern processors. The pipeline-matching in BPMS blocks creates vector-data sets aligned to natural vector boundaries in memory, which ease implementing vector operations. Traditional blocked formats only exploit data-locality by creating small blocks that match the size of the register file or the cache lines (also achieved by BPMS), but do not align data to vector boundaries nor do they assure data sizes to fit within vector-registers as mentioned earlier. The row index vector (AI) can be used to determine high-level boundaries for data-partitions to spread across processing cores, whereas the newly introduced fourth vector (**Blocks per row-block**) aids in load balancing by providing information on the amount of data to compute on for each coarse data-partition defined by AI. The information provided by the fourth vector also provides useful information for low level loop optimizations.

4.1.2 Enhancing Block Structure in BPMS

Block structures occur naturally in finite element matrices [82, 83] as the effect of adding the element stiffness matrices to the global coefficient matrix. However, the overall matrix structure may require many zero fill-ins when creating the blocked formats. The non-desired effect of such zero fill-ins may be considerably reduced by grouping the nonzero entries in the matrix. One way of achieving this is by using a bandwidth reduction algorithm. This work implements one such algorithm called the reverse Cuthill-Mckee [84] (RCMK) algorithm, a well known and time efficient [85] bandwidth reduction algorithm, before creating the block formats. The RCMK algorithm compresses the nonzero elements along the main diagonal, increasing spatial locality and normally incurring less zero fill-ins. Also, access to the *x-vector* will now be almost sequential for matrices that allow for a good compression.

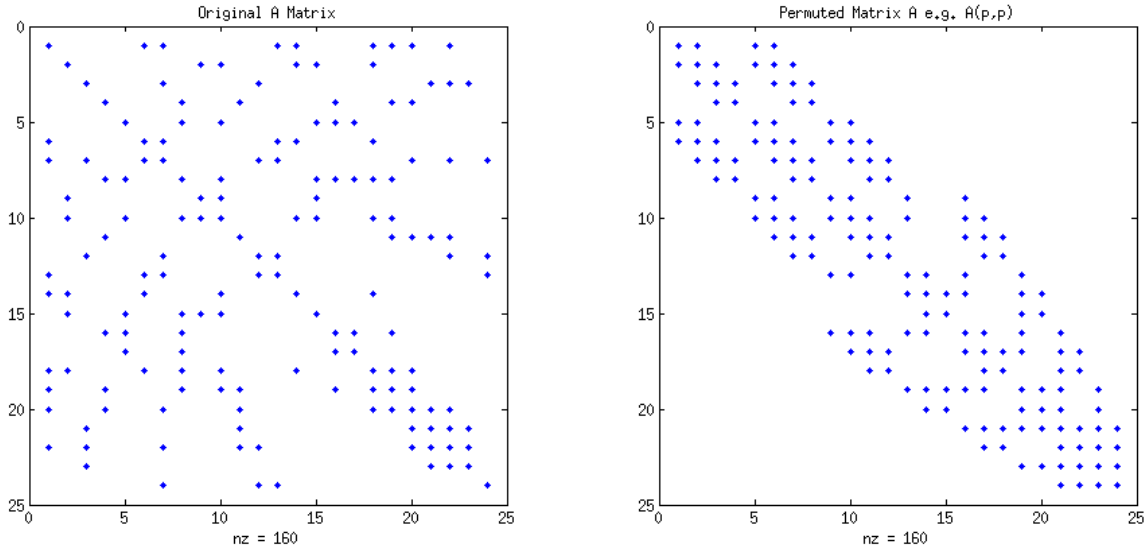
The enhanced locality property in the access of the *x-vector* when reordering will also be available for the non-blocked formats, effectively reducing cache misses and hiding better memory latencies, thus the test results in this section will be for the reordered matrices in all formats. Figure 17.a shows how the bandwidth of the “can___24” matrix is reduced from 24 to 13, and reduces its average bandwidth from 8.3 to 3.8 when applying the RCMK algorithm. A similar reduction is experienced for “fidapm37” where the bandwidth reduces from 1364

to 903, and the average bandwidth goes from 442 to 290 (see Figure 17.b). On the other hand, the ordering from the matrices 2, 3, and 5 results in an increase of the bandwidth since they were already diagonal matrices (even though the average BW remains almost the same as the unordered one), thus for such matrices there is no gain in reordering.

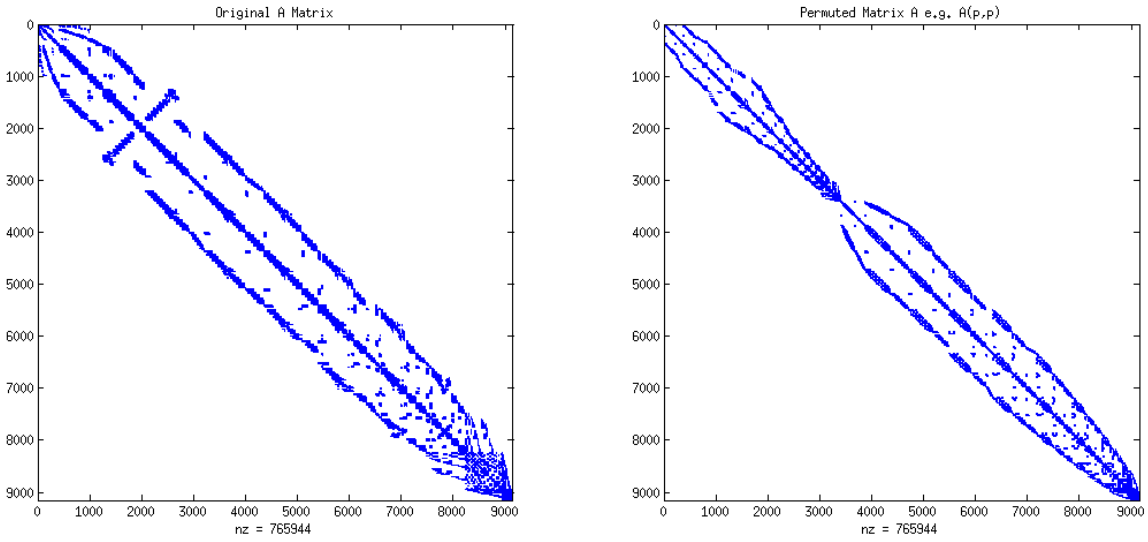
Finally, the insight provided by the reordering process can also be used to define coarse data-partition boundaries within the matrix. When a matrix is reordered using RCMK two auxiliary vectors are generated, a vector called level-set (which aids in the decision on how to reorder the matrix) and a permutation vector that determines reordering to do on the matrix. The level-set vector can be used to define partitions in the reordered/permuted matrix as shown in Figure 18. The main drawback of this approach is that the partitions defined in this fashion are too coarse and thus difficult to balance, which may lead to large zero fill-ins in the blocks generated.

4.1.3 Vectorized SMVM and Multicore Parallelization for the BPMS Format

The vectorization of the SMVM kernel for the blocked formats is similar as the one done in Chapter 3, where the inner loop is unrolled by the vector-register size (4-SPFP or 128-bit vector registers) as shown in Figure 19.



(a) can___24 matrix.



(b) fidapm37 matrix.

Figure 17: Bandwidth reduction examples when applying the reverse Cuthill-Mckee algorithm.

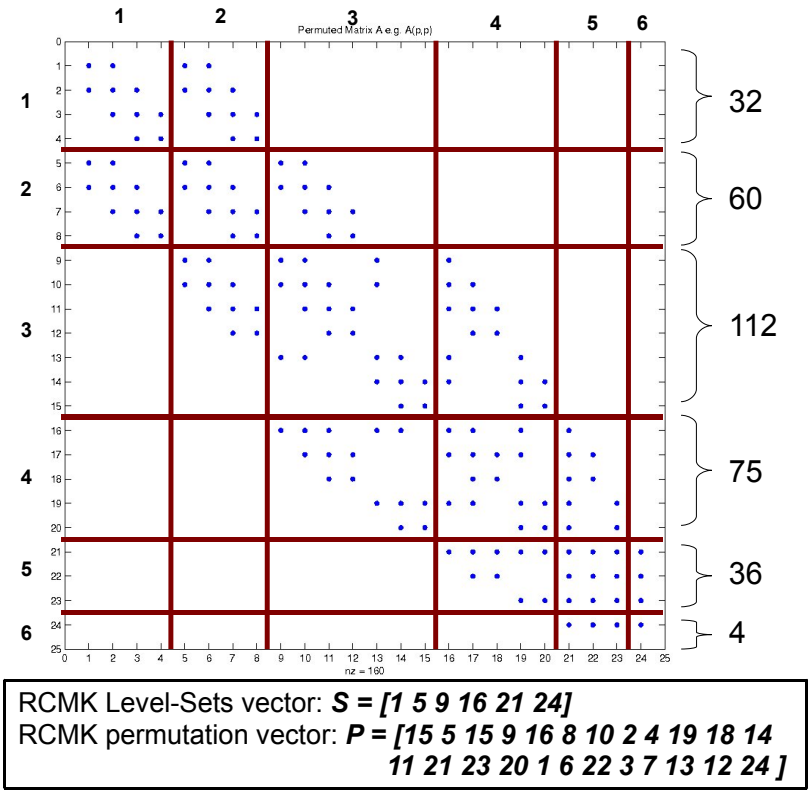


Figure 18: Partitioning example for the "can___24" matrix after reordering using the level-set vector generated in RCMK. The number along the brackets to the right of the matrix identifies the worst case scenario for the size of the blocks that could be generated by such partitioning.

```
1:for(i=0; i<rows/blockRank; ++i){
2:  start = stop;
3:  stop = start+(A->blocks[i+1]-A->blocks[i]);
4:  res_acc = _mm_setzero_ps();
5:  for(j=start;j<stop;++j){
6:   mat_data = _mm_load_ps(mat_ptr);
7:   vector_data = _mm_set_ps(x_ptr[col_idx+j+1],x_ptr[col_idx+j],
                              x_ptr[col_idx+j+1],x_ptr[col_idx+j]);
8:   temp_mul = _mm_mul_ps(mat_data, vector_data);
9:   res_acc = _mm_add_ps(res_acc, temp_mul);
10:  mat_ptr += 4;
11: }
12: res_acc2 = _mm_hadd_ps(res_acc, res_acc);
13: _mm_storel_pi(res_ptr,res_acc2);
14: res_ptr += 2;
15:}
```

Figure 19: Blocked sparse matrix-vector kernel using the BPMS format.

However, in this case the four partial results obtained for each inner-iteration correspond to two[13] consecutive rows of the inner loop, so a simple 1-level tree reduction can be applied at the end of each *row-of-blocks*[14] (see line 12 in Figure 19). The other difference in this kernel is the trip count (number of iterations) in the outer-loop, which is reduced by the rank of the blocks defined in the sparse format as shown in line 1 of Figure 19. This also means that there is less instruction overhead in the outer loop. The SMVM computation is done in lines 6 to 9 (highlighted in Figure 19) as follows.

- Line 6: load a matrix block into a vector-register.
- Line 7: load the corresponding *x-vector* elements into a vector-register.
- Line 8: multiply the matrix & vector elements (*blocked dot-product*).
- Line 9: accumulate the partial results for the *row-of-blocks*.

Multicore processing is achieved in the same manner as for PMS using PThreads. Matrix data is partitioned in run-time using the *2-level partitioning scheme* described in Chapter 3, and then *row-blocks* (1st level partitions) are assigned to each of the processing cores. Each core computes the SMVM kernel on their partition and results are synchronized and gathered. The number of rows

---

[13] In general, the number of partial results obtained for different rows will correspond to the number of rows in the blocks defined while constructing the blocked format.

[14] The term *row-of-blocks* refers to a set of consecutive blocks in the same set of consecutive rows that are formed in the blocked formats. Note that this concept differs from that of *row-block* which refers to the coarse (Level 1) data partition proposed in Chapter 3.

grouped into such *row-block* sets may vary to statically balance the non-zeros before being assigned to each core, this depends on the input matrix and the number of core. Figure 20 shows the steps followed to partition and load balance the BPMS matrix.

Partitioning and load balancing BPMS matrix data with the *2-level partitioning scheme* is a fast operation of $O(n / blockRank)$ complexity (where *n* is the number of matrix rows and *blockRank* is the number of rows in each small dense block defined in the matrix). An even cheaper approach is to obtain a similar *row-block* partitioning using the *set-vector* generated in the RCMK reordering. This vector contains row indices which define bounds for the matrix data. Such bounds can be directly used as the partition indices. Nonetheless, the success of such an approach depends on the compression of the matrix bandwidth (BW). If the BW is not evenly compressed the load will be unbalanced and a more sophisticated load balancing approach must be used. In this work we only use the 2-level partitioning scheme explained in Chapter 3. The next section presents the results for the SMVM kernel with the new sparse format, focussing on its scalability for the cache based architecture used.

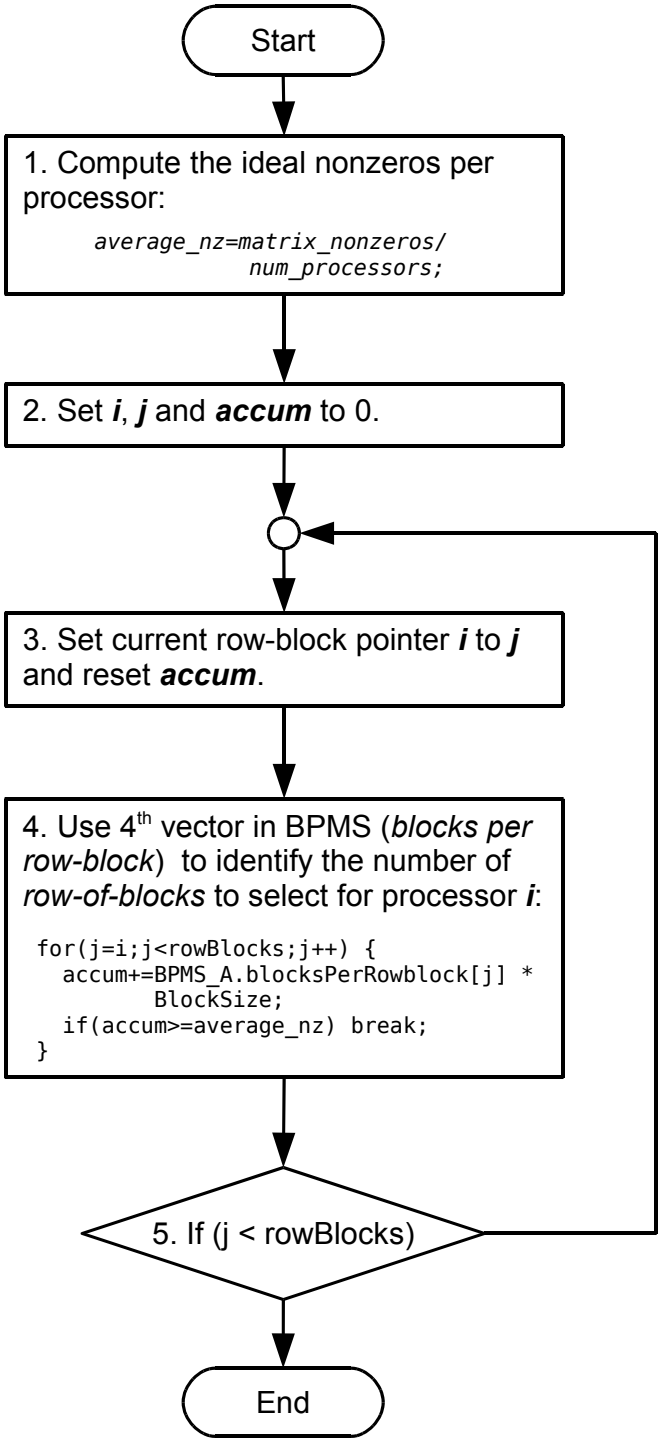


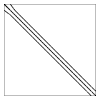
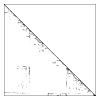
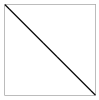
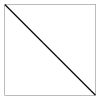
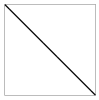
Figure 20: Steps to compute the 1st-level partitions of the 2-level partitioning scheme and to statically load balance a matrix in BPMS format across several processors.

4.2 Experimental Results for SMVM

The experimental setup for the results presented in this section is the same as the one used in section 3.4, thus comments will only be made for the variations in the setup or new considerations. Both of the blocked formats presented in this section (the BCSR and the new BPMS formats) have been configured in the same way so that a fair comparison can be assessed. The two formats were configured with 2x2 blocks (which minimizes the memory footprint of zero fill-ins). The BCSR format was also unrolled and optimized using the compiler optimizations. Non-square block configurations can also be tested, but were not considered for the results in this dissertation.

In addition to the matrices used in Chapter 3, two new matrices where added from Matrix Market repository [42] (bcsstk32 and s3dkt3m2) and a set of artificially generated matrices (with nonzeros ranging from 6 million to 10 million) to better study the performance scaling behaviour of the new sparse format and algorithm (see Table 6). The artificial matrices generated have a band structure, increasing very little the amount of zero padding done for the PMS and BPMS formats. Table 6 shows the information for all the matrices, but results for this chapter will focus on the bigger matrices (4-8).

Table 6: Finite element (FE) test matrices from the Matrix Market repository [42].

#	Name	Rank	NZ	Sparsity %	PMS NZ	PMS % added zeros	BPMS NZ	BPMS % added zeros	Sparsity pattern
1	can___24	24	160	27.78%	208	30.00%	256	60%	
2	cavity26	4562	138187	0.66%	144148	4.31%	185428	34.18%	
3	e40r5000	17281	553956	0.19%	578312	4.40%	736336	32.92%	
4	fidapm37	9152	765944	0.91%	781100	1.98%	931648	21.63%	
5	bcsstk32	44609	2014701	0.10%	2082628	3.37%	2626404	30.36%	
6	s3dkt3m2	90449	3453461	0.05%	3931224	4.74%	4467300	19.02%	
7	s3dkq4m2	90449	4820891	0.06%	5001068	3.74%	5366604	11.32%	
8	SP10	10000	10006318	10.01%	10016832	0.11%	10034776	0.28%	

The first results evaluate the performance of the new BPMS format compared to PMS and BCSR (the reference block format). The executions Speedup[15] (SU) results for the SMVM kernel using the PMS, BCSR, and the new BPMS formats with respect to the CSR format are shown in Figure 21 for increasing matrix sizes

[15] Absolute time ratio of CSR with respect to PMS, BCSR and BPMS execution times.

on a single Intel core. The CSR format is used as a reference providing the base computing time for comparison (since it contains no zero padding), whereas the other formats have extra computational overhead. The PMS results are shown mainly for comparison with the BPMS, since its performance was already studied in Chapter 3. The SU curves increase as the matrices grow and stabilize around 2.5x for BCSR, 2.9x for PMS, and 4.4x for BPMS using the optimized kernels as described in section 4.1.2. These SU results clearly show that BPMS outperforms the other formats, a trend that stabilizes for the bigger matrices as the cache misses become regular. It is also interesting to observe that even for the worst case in Figure 21 (corresponding to the bcsstk32 matrix with 2 million nonzeros, when the matrix does not fit in the cache) BPMS is 3.4x faster than CSR. These results are even more impressive considering the high zero fill-in percentage (30.36%) in this matrix.

BPMS also demonstrates good scaling for increasing matrix sizes, requiring less padded-zeros (see Table 6). This is true in general, but zero padding may slightly increase for very irregularly-structured matrices or regularly-structured matrices with numerous cavities between its entries, e.g., the three matrices in the valley of Figure 21. Compared to the PMS format the newly introduced format requires more zero fill-in, which can be observed from the percentage of zero fill-ins for each of these formats in Table 6.

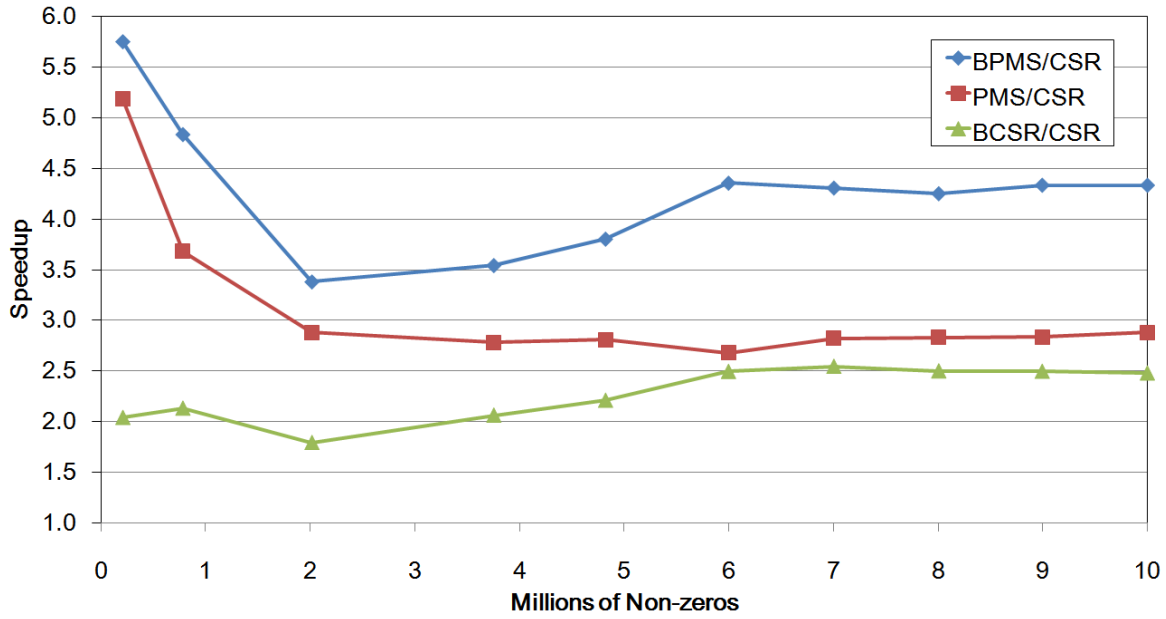


Figure 21: Speedup results for the SMVM kernel with the PMS, BPMS and BCSR with respect to the CSR format for one Intel-core.

The next set of results describes the effect of the SIMD vectorization running on several processing cores for all matrix formats with and without optimizations. The first section of Table 7 shows SU[16] results obtained for the optimized (vectorized) SMVM kernels (BCSR, PMS, and BPMS) with respect to the non-optimized (non-vectorized) versions respectively for 1 and 4 cores. These results demonstrate that the vectorization of the SMVM kernel increases considerably the performance for all specialized formats, achieving up to 17x speedups for matrix (4) using the new BPMS format. The high SU obtained for this matrix is mainly due to the fact that it fits in the Intel-CPU cache. Overall, the amount of

[16] The speedup results obtained in this section were computed as the time ratio of non-optimized to optimized kernels for a given number of computing cores.

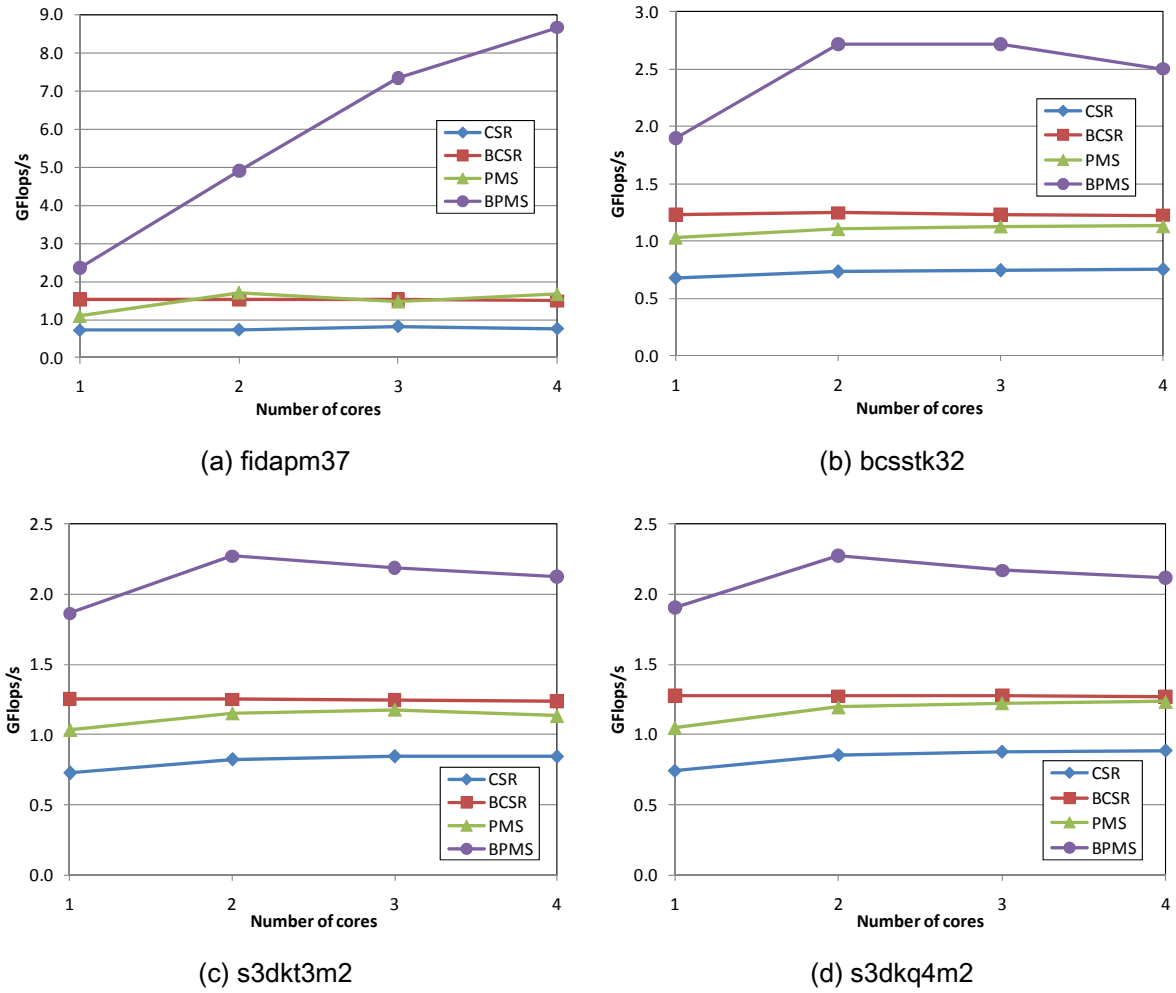
performance gains with the vectorization of the SMVM kernel is reduced when more cores are used. On the other hand, for the larger matrices that do not fit in the cache the performance increased with the optimizations and the number of cores but was limited by the achievable memory bandwidth (BW) for each test matrix.

The second part of Table 7 presents performance results for the vectorized kernels in GFlops/s using 4 Intel-cores and 6 Cell-SPE cores. Overall, a sustained performance of up to 8.24 GFlops/s for the SMVM-BPMS was observed with an average of 3.4 GFlops/s, the fastest SMVM kernel. Overall time results show that BPMS is 3.6x faster than CSR, 2.2x faster than the PMS format and 2.5x faster than BCSR for the matrices in Table 7 using four Intel cores. For the Cell BE the PMS format performs better than the BPMS format. The reason this happens is that PMS allows to efficiently stream both matrix and *x-vector* data to the SPE's local memory using the double-buffering technique; whereas BPMS fetches the *x-vector* segment required to SPE local memory upon a miss in the *x-vector* segment already contained in SPE local memory.

Table 7: SMVM performance comparison results for the BPMS, PMS and BCSR SMVM kernels for the Intel Core 2 Quad and Cell BE processors.

Intel SMVM optimized/non-optimized performance speedup					
Matrix (#)	(4)	(5)	(6)	(7)	(8)
1C-PMS	3.33	2.71	2.68	2.7	2.78
4C-PMS	2.87	2.19	2.03	1.04	1.73
1C-BCSR	2.53	2.42	2.42	2.44	2.62
4C-BCSR	3.28	2.11	1.88	1.86	1.81
1C-BMPS	5.49	4.42	4.35	4.45	4.8
4C-BMPS	17.14	4.58	3.55	1.84	2.85
4-Intel cores and 6-SPE cores SMVM performance results in GFlops/s					
Intel-PMS	1.53	1.2	1.26	1.3	1.38
Intel-BCSR	1.57	1.14	1.12	1.27	1.48
Intel-BPMS	8.24	2.47	2.13	2.11	2.21
SPE-PMS	0.54	0.65	0.67	0.69	0.66
SPE-BPMS	0.3	0.22	0.29	0.29	0.31

Multicore performance scaling results for the cache-based architecture (Intel processor) deserves special attention since they motivated the creation of a second sparse matrix format. Figure 22 shows performance scaling results of four matrices varying the number of compute cores from 1 to 4. Again the almost linear scalability for SMVM-BPMS in Figure 22.a is achieved because the matrix fits in cache. The other matrices exhibit a reduced scalability for BPMS, but that is still much better compared to CSR, PMS and BCSR which was the original objective.



(a) fidapm37

(b) bcsstk32

(c) s3dkt3m2

(d) s3dkq4m2

Figure 22: Multicore performance scaling in GFlops/s for the SMVM kernel using the CSR, BCSR, PMS and BPMS formats on the Intel Core 2 Quad processor.

The next section presents results for the conjugate gradient method accelerated method accelerated with vectorized SMVM algorithms.

4.3 Conjugate Gradient Results

In many modern EM simulations the solution of the linear system derived from the discretization of the problem domain (or operators) consumes a great deal of computational resources, often being the most time consuming operation.

Thereof, this section discusses the acceleration of the conjugate gradient (CG, discussed in Chapter 2) method using the newly introduced sparse formats and vectorized SMVM kernels. Since the parallel SMVM kernel is called in every iteration of the CG algorithm there will be considerable time invested in creating and destroying PThreads. To avoid this, a set of persistent threads (also called thread-pool) was created. These threads are set to sleep until an SMVM operation is required; at this point they are started. When the SMVM operation is completed the threads are synchronized and set to sleep again. Threads are only terminated when the CG algorithm ends.

The non-vectorized parallel SMVM kernel is used in a parallel CG algorithm for the new BPMS format as *proof-of-concept*. The scaling performance with respect to 1-core BPMS-CG is shown in Figure 23 using four of the biggest test matrices. Near 3-times increase in performance (measured in GFlops/s) is observed when running the non-vectorized BPMS-CG from 1 to 4-cores confirming good scaling performance expected from the BPMS accelerated CG.

Performance results in GFlops/s are shown in Figure 24 for the CG accelerated versions using 4 Intel-cores for different sparse matrix formats. The average performance in GFlops/s for each format is as follows: 1.31 for BCSR, 0.58 for PMS and 2.83 for BPMS. Thus, BPMS is approximately 2.1x faster than BCSR, almost 4.7x faster than PMS, and 5.8x faster than CSR on average.

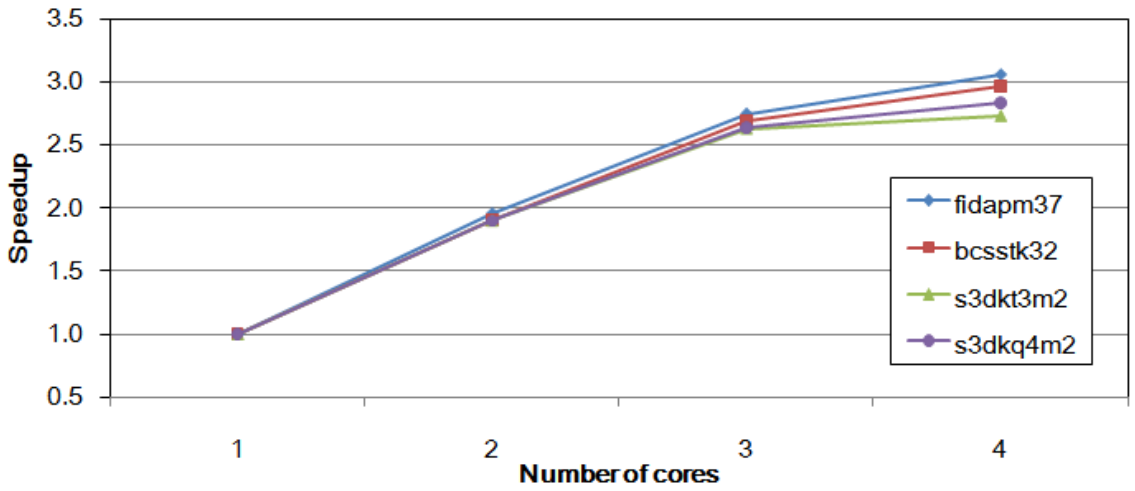


Figure 23: Speedup scaling for the conjugate gradient method using BPMS with 1 to 4 Intel CPU cores.

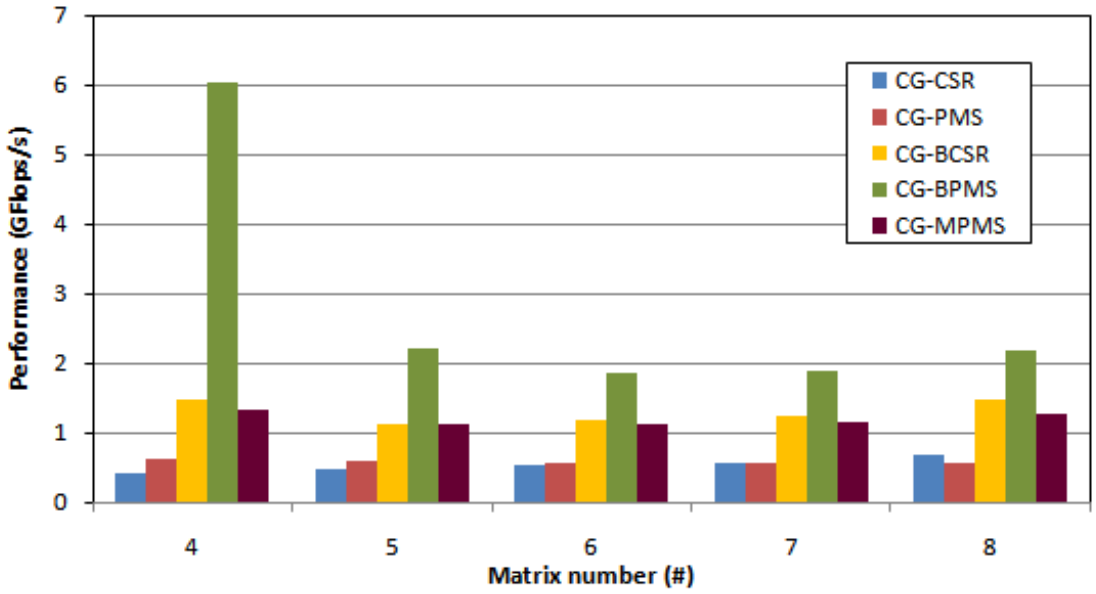


Figure 24: Conjugate gradient performance results in GFlops/s using 4 Intel cores.

These results demonstrate the performance benefits of BPMS over all other formats for CG algorithm in the Intel cache-based architecture. A different situation is found for PMS, which is not the best suited format for CG since there

are insufficient instructions to hide the *vector-spreading* computation time as shown by these results. Nevertheless, the PMS format can be used in other applications where sufficient instructions parallelism exists to hide the cost of the *vector-spreading* operations or when multiple vectors exist so that the spreading time can be overlapped with other instructions.

An alternative to recover the performance lost in PMS by the spreading operation of the *x-vector* is to entirely avoid the spreading operation. The performance of the SMVM-PMS will be reduced compared to that shown in Chapter 3 due to loss of regular access patterns to the x-vector data; this is because the SMVM-PMS kernel can no longer be treated as a dense kernel. For cache based architectures this approach takes advantage of the cache hierarchy to access *x-vector* data, but performance will depend on the sparsity pattern of the matrix. The results for this approach are presented in Figure 24 as CG-MPMS (CG using the Modified-PMS, with no vector spreading operation), showing how a great deal of performance is recovered by dynamically accessing the x-vector in the PMS format, where BPMS is only 2.3x faster than MPMS (a similar performance difference to the corresponding SMVM results). Even though the performance is recovered for the MPMS its scalability is still poor as shown by Figure 25.

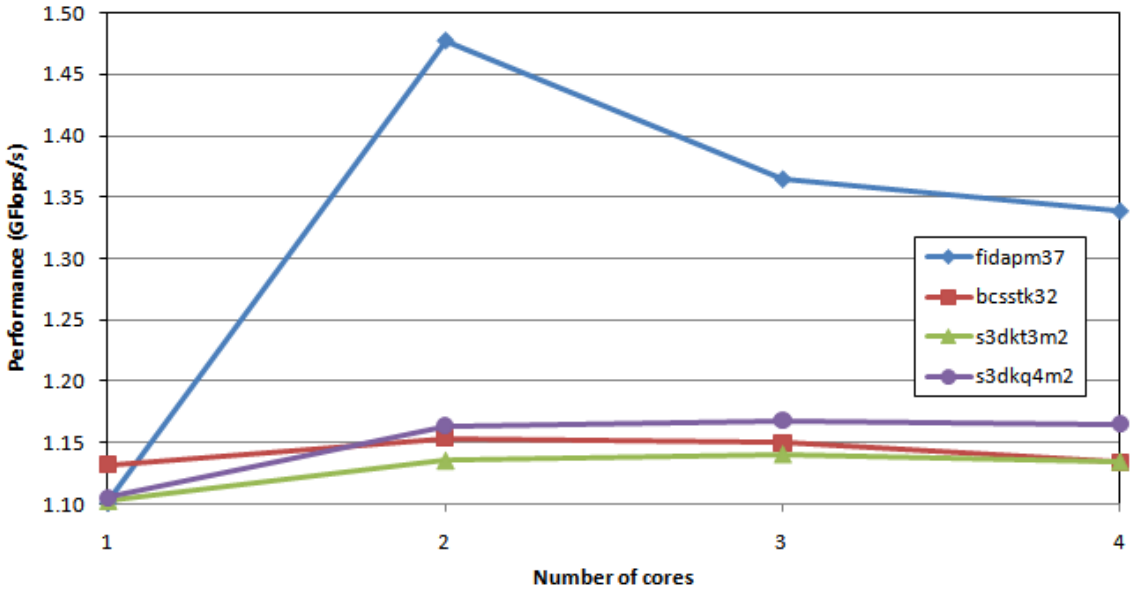


Figure 25: Scaling of the CG performance in GFlops/s for the modified PMS (MPMS) format.

4.4 Concluding Remarks

This Chapter presents the third contribution of this work, a new blocked format called blocked pipeline-matched sparse (BPMS) representation that solves the performance scalability problems of the PMS format for cache-based architectures. The BPMS format compresses the index information, enhances data locality, and avoids overhead related to the vector-spreading operation required in PMS. Even though BPMS requires more zero fill-ins than PMS it demonstrates superior performance for cache-based architectures, while PMS performs better for the streaming architectures such as the Cell BE. The BPMS accelerated SMVM kernel on the Intel processor using 4 Intel cores demonstrated an average performance benefit of 2.5x over BCSR and 2.2x

faster than PMS. On the other hand, BPMS was 2.1x faster than BCSR and almost 5x faster than PMS on average for the CG algorithm. Figure 26 shows the complete workflow used to compute the CG algorithm. The vector spreading operation and the sequential nature of the CG algorithm are the main factors that reduce the performance of the PMS format for the CG algorithm.

An alternate configuration of PMS is proposed that does not include the *x-vector* elements and that offers better performance for cache-based processors. The blocked format introduced in this chapter was originally presented in [86] and published in [87]. The next chapter introduces the last major contribution of this work, an alternate formulation of the FEM solution for massively parallel computing.

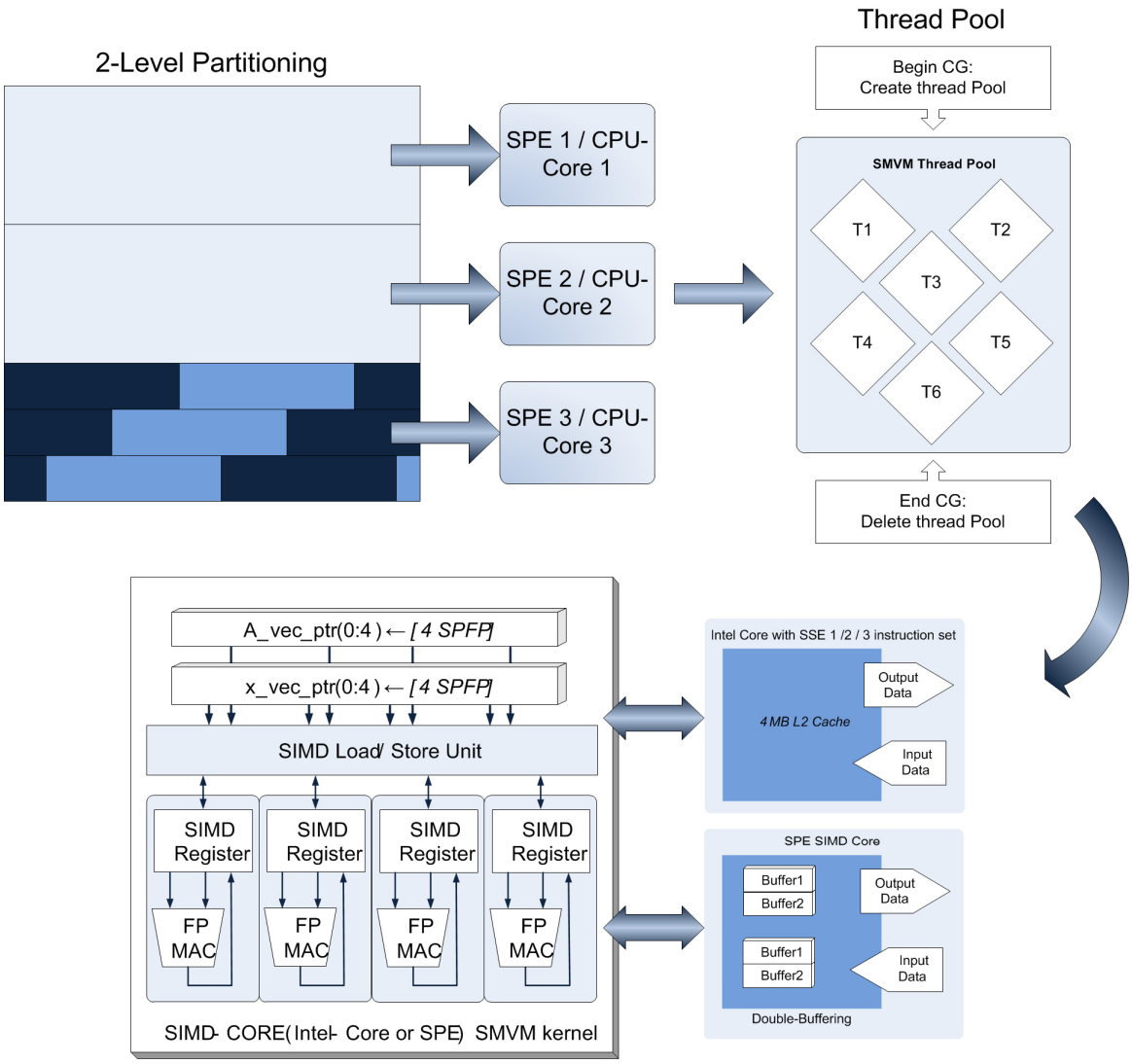


Figure 26: Complete workflow used to compute the CG algorithm with the vectorized SMVM kernel and the 2-level partitioning scheme.

Chapter 5
A Parallel Approach to Solving the Finite Element Method

Parallel implementations of the finite element method (FEM) in large clustered or symmetric multi-processor (SMP) systems has traditionally been done exploiting *coarse-grained* parallelism in the dominant computing kernels (the SMVM and preconditioner operations in iterative solvers), while it has relied in traditional technological advances (Moore's Law) to increase performance in local nodes (single processors). Chapters 2 and 3 have introduced new techniques to exploit *fine-grained* parallelism (SIMD-vectorization and parallel multicore processing) found in modern multicore processors that can be used to accelerate computations in these traditional approaches to parallel the FEM.

The first two sections of this chapter introduce a new approach to solve the FEM by exploiting the low level parallelism available in the formulation of the method itself, thus well suited for parallel computing. The second part of the chapter develops a technique to implement the proposed approach on graphic processing units (GPUs) that makes efficient use of the massive parallel resources found in these processors. The new approach and the technique to implement it on GPUs constitute the last two major contribution of this work. The

chapter ends with an overview of specific work related to the two contributions introduced here.

5.1 New FEM Single Element Solution (FEM-SES) Method: Alternate *Fine-Grained* Parallelism in the Solution of FEM

The classic FEM formulation can be presented as a seven step procedure [82] as follows (see Figure 27.a).

(i) Discretization of the problem domain.

(ii) Definition of boundary conditions (BCs).

(iii) Construction of the element stiffness matrices.

(iv) Assembly of the global coefficient matrix imposing corresponding BCs.

(v) Solution of the algebraic system.

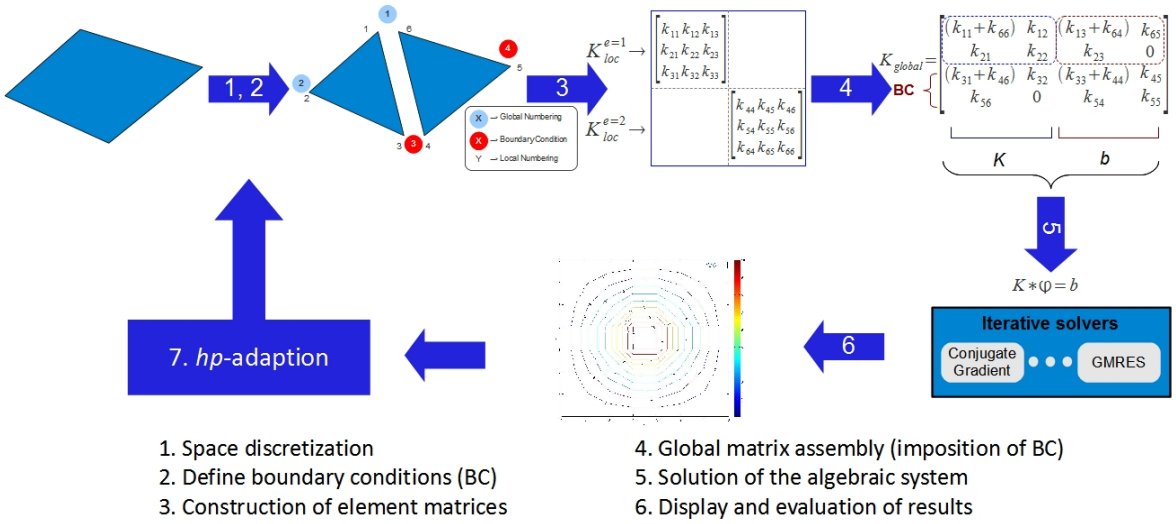
(vi) Post-processing of the results. If the results meet the required accuracy then the method ends, otherwise an additional step is taken.

(vii) Refine the mesh and/or change the basis functions and restart the process from the first step.

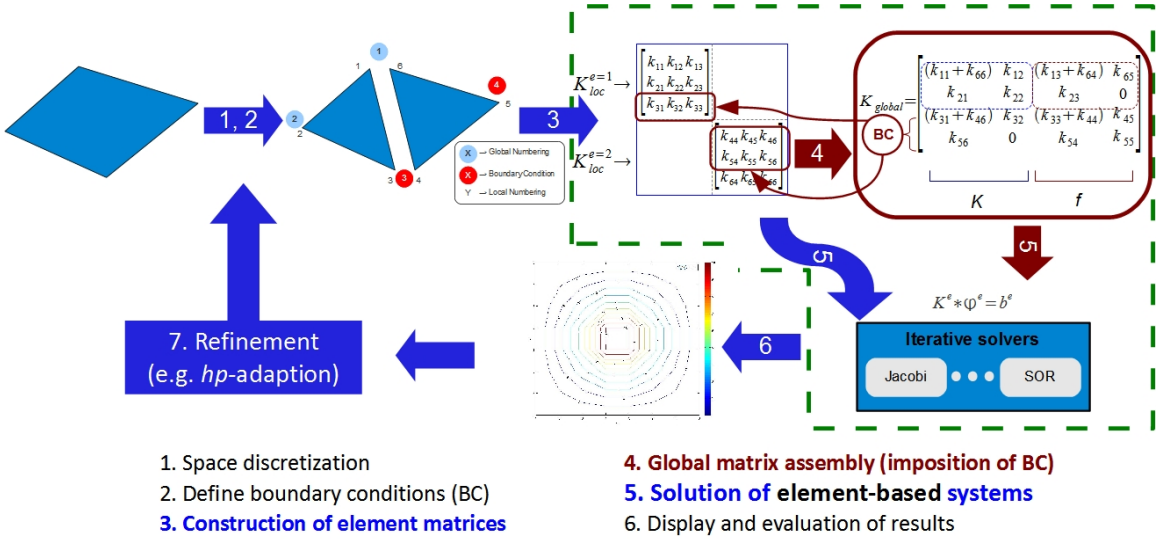
Often the most time consuming and attractive candidate for parallelization in the finite element method is solving the algebraic system derived from step (v). The three most common approaches to parallelize the solution of FEM are: a) partitioning and solving in parallel the derived algebraic system [87-89]; b) employing domain decomposition techniques [90-93]; and c) using multigrid

techniques [91, 93]. However, a greater amount of parallelism is sought to take advantage of parallelism in multicore/manycore processors. This work proposes a new approach called single element solution or FEM-SES, in which the solution of each finite element is decoupled from that of the whole mesh by computing element stiffness matrices (subject to boundary conditions) concurrently.

Figure 27.b illustrates the proposed change with a blue arrow (labelled 5) that connects step (iii) to step (v) directly. The disconnected solutions are then averaged node-wise using a weighted sum over all concurrent nodes and iterated until convergence is achieved. By skipping step (iv) of the classic FEM procedure described before, the proposed approach does not require building a global coefficient matrix. Thus the new FEM-SES method uses the same steps (i), (ii), (vi) and (vii) from the classic FEM workflow, skips step (iv), and modifies steps (iii) and (v). The remainder of this chapter concentrates on explaining the changes in steps (iv) and (vi) (encircled inside the green dashed line in Figure 27.b), proving the validity of the new method and its benefits for parallel computing. The mathematical formulation for the proposed decoupled finite element method-single element solution (FEM-SES) is presented next.



(a) Classic finite element method (fem) workflow.



(b) Proposed single element solution (FEM-SES) workflow.

Figure 27: Comparison of classic finite element method (FEM) workflow (top figure) with respect to the proposed single element solution (FEM-SES) method (bottom Figure).

5.1.1 FEM-SES Mathematical Formulation

Equations (4-6) present the classic FEM variational formulation for a simple 2D electrostatic boundary value problem, which will be used throughout this chapter without loss of generality. In the aforementioned equations $F(\varphi)$ is the functional to minimize, and the unknowns and boundary conditions are represented using φ and p respectively.

$$\partial F(\varphi) = 0 \tag{4}$$

$$\varphi = p, \text{ on the boundary } \Gamma \tag{5}$$

$$F(\varphi) = \iint_{\Omega} \left[\left(\frac{\partial \varphi}{\partial x} \right)^2 + \left(\frac{\partial \varphi}{\partial y} \right)^2 \right] d\Omega \tag{6}$$

The functional is then applied to each element in the discretized domain as shown in Equations (7-8) where the superscript e refers to the element index.

$$F(\varphi) = \sum_{e=1}^{n} F(\varphi^e) \tag{7}$$

$$F^e(\varphi^e) = \iint_{\Omega_e} \left[\left(\frac{\partial \varphi^e}{\partial x} \right)^2 + \left(\frac{\partial \varphi^e}{\partial y} \right)^2 \right] d\Omega \tag{8}$$

Next, the local functionals are minimized and boundary conditions are enforced on each element independently, see Equation (9).

$$\left\{ \frac{\partial F^e}{\partial \varphi^e} \right\}_{BC\_reduced} = \left\{ K^e \right\} \left\{ \varphi^e \right\} - \left\{ b^e \right\} = \left\{ \varnothing \right\} \tag{9}$$

This is where the new method departs from the classic FEM.

5.1.2 The 2-Step Iterative Relaxation Method

To obtain the global solution from Equation (9) a *2-step iterative relaxation* approach is proposed (see Figure 28). The first step applies a relaxation technique using the previously obtained iterate-φ and the local system derived using a matrix modification process from Equation (9) to compute the local element solutions concurrently. Figure 28 shows an example for a Jacobi type update on first order triangular elements with one, two or no boundary conditions and first order basis functions. In the second step the local solutions from overlapping nodes are summed using a weighted average to compute the global solution.

The weights are computed using the main diagonal values of each of the elements matrices. Finally, a convergence check is performed to either exit or repeat the process. The next subsections briefly describe the sources of parallelism, advantages and disadvantage of the proposed *2-step iterative relaxation* method.

a) Sources of Parallelism

Various sources of parallelism exist in the new approach. The most significant are listed in the following:

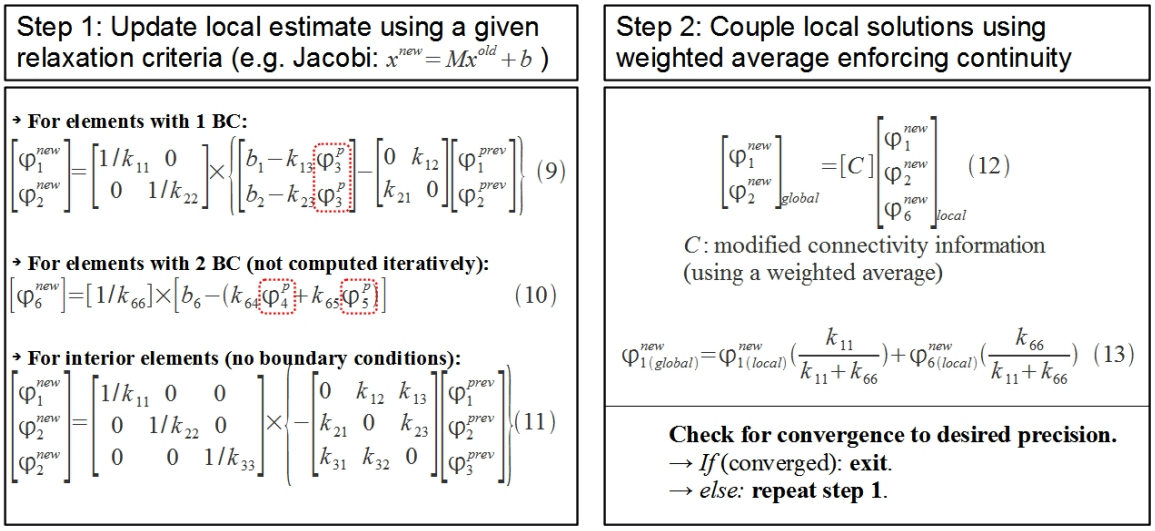


Figure 28: The 2-step iterative relaxation method. Step 1 shows examples of how to compute the solutions for elements with 1, 2, and 0 BCs. The examples here correspond to triangular elements with first order basis functions.

- Element stiffness matrices can be built in parallel and preserved in distributed CPU/cores to be computed later.
- The solution of each element can be computed in parallel independently.
- The weighted average can be performed in parallel across different nodes taking into account the element connectivity.

b) Advantages and Disadvantages

The main disadvantage of the *2-step iterative relaxation* is that it will converge slowly, similar to the Jacobi iterative method; although, the amount of parallelism per iteration is increased considerably. On the other hand, by exploiting parallelism in each iteration, the new approach reduces the total execution time of the finite element method as demonstrated in the results section. Among other advantages, the proposed FEM-SES method does not require special numbering

(local and global numberings) to build the global coefficient matrix and later obtain the final results requiring less time and effort in housekeeping procedures. There is no need to assemble a global coefficient matrix which might also become a time consuming step. The proposed method uses the same information as the classic FEM, and good scaling is expected since the element connectivity (the number of surrounding elements connected to a given element) of the FEM mesh will be almost constant as the mesh is further refined to better represent the geometry of the problem.

5.2 Proof of Concept Results

This section presents results to validate the new proposed FEM-SES method as well as performance results using a classic 2D electrostatic coaxial cable problem as shown in Figure 29. In this test case the outer square conductor is connected to ground while the inner square conductor is held at a constant voltage of 10V, and the two conductors are considered to be separated by air. A program was developed to generate regular meshing of this particular problem in MatLab in order to have better control over the number of elements being generated. The meshing program adds more elements to the mesh by creating equally spaced partitions between the inner and outer conductors and then discretizing these partitions using triangular elements. We use first order basis functions for all the elements in the finite element mesh.

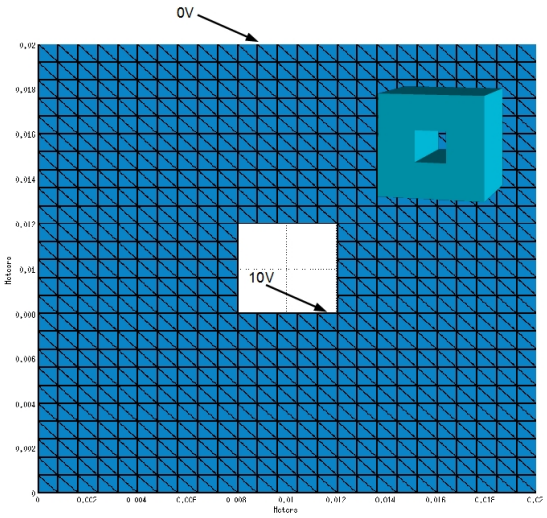


Figure 29: 2D section of electrostatic coaxial cable. The inner conductor is fixed at 10V and outer conductor at 0V. The space between the conductors is considered free space. The insert shows a 3D representation of the cable being model.

All tests are conducted on the Intel processor described in Chapter 3. Sequential implementations for both the traditional FEM and the new FEM-SES methods are done to validate the results of the new method. The systems energy results for the two methods are presented in Figure 30 for increasing number of unknowns. As shown in Figure 30 there is good agreement between the two energy results, validating the new method.

To empirically evaluate the convergence scaling of the FEM-SES method the original mesh is refined to increase the number of unknowns. The method is evaluated for various mesh discretizations and the resulting iterations are plotted versus the number of unknowns in a log-log plot in Figure 31. The reference line (dotted line) in Figure 31 represents a linear scaling with slope 1, while the solid lines correspond to the actual iteration count for each run.

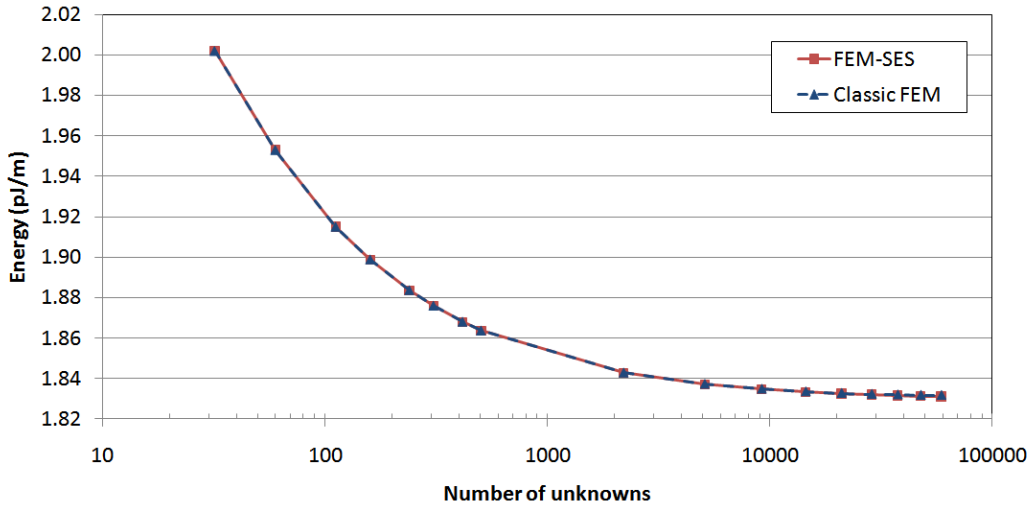


Figure 30: Energy results for the FEM and FEM-SES.

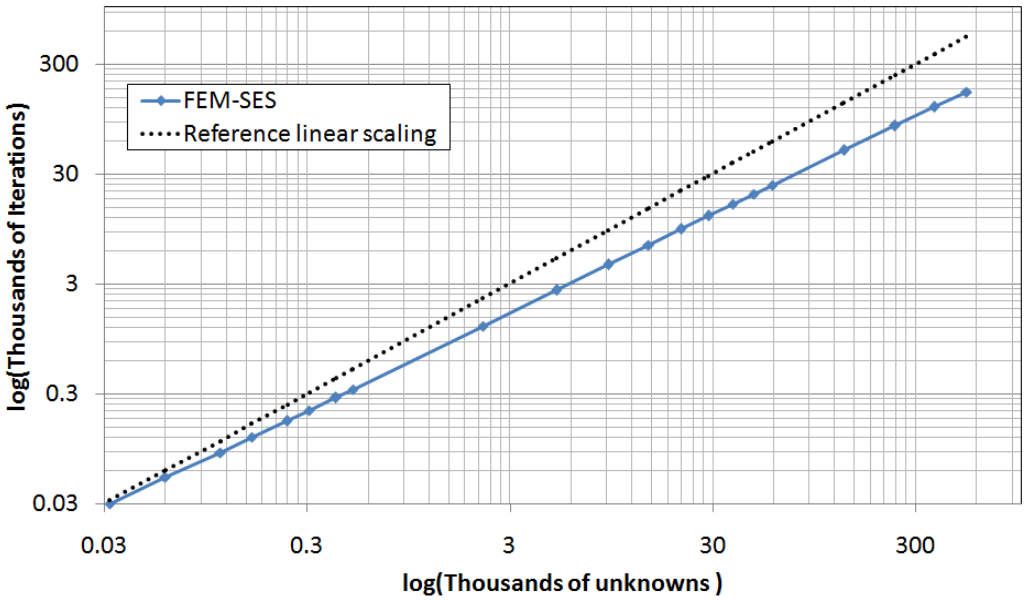


Figure 31: Iterations scaling of the FEM-SES method for increasing number of unknowns.

These results show a sub-linear iteration scaling of the proposed FEM-SES method as the number of unknowns increase, which is a desirable scaling property for iterative methods. Such sub-linear scaling is obtained even though the condition number increases considerably from 89.8 to 9672.7 as the mesh size (or unknowns) grows for the smallest to largest mesh sizes respectively.

5.3 Parallelizing Results for the FEM-SES Method

In this section, techniques to parallelize FEM-SES are presented. A sequential implementation of the FEM-SES method is profiled first to determine the amount of time spent in the main sections of the algorithm. The two most important operations in the algorithm are the actual assembly of the element matrices and the *2-step iterative relaxation* method itself. All other operations in the algorithm are considered as pre- and post-processing steps. The results for the profiling are shown in Figure 32. It is clear that as the number of unknowns grows the FEM-SES solution (i.e. *2-step iterative relaxation* method) dominates all other operations (assembly, pre and post processing). Consequently, we concentrate on parallelizing *the 2-step iterative relaxation* method only, that correspond to the last two sources of parallelism identified in Section 5.1.2.a.

Even though the methods proposed up to now have taken advantage of multicore processors with a few (2 to 4, or 8 in future generation multicore chips) cores, the proposed FEM-SES method offers a great amount of parallelism (each element solution can be computed concurrently) which is better suited for a processor with hundreds or thousands of cores, such as those found in graphic processing units (GPUs). Thus, this section will concentrate in implementing the FEM-SES method to exploit the parallelism found in modern GPUs, while a reference multicore implementation will also be presented for comparison.

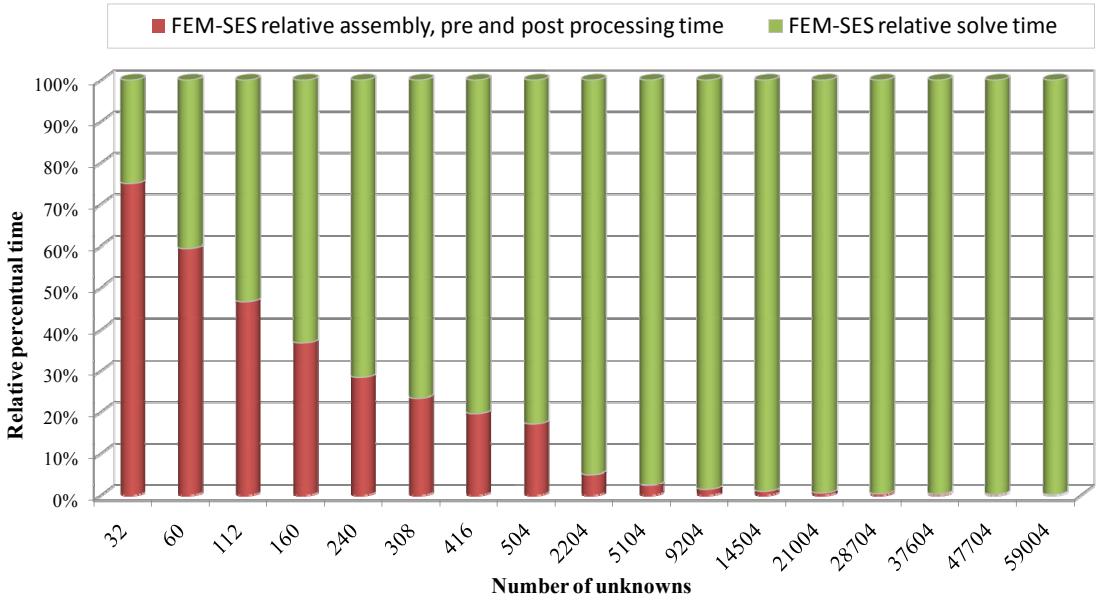


Figure 32: Profiling results for the sequential FEM-SES method.

5.3.1 Multicore Results for the Intel Platform

A parallel implementation of the FEM-SES method was done for the Intel multicore processor (vectorizing the element solutions) to asses to potential performance gains in this type of multicore architecture. The vectorization is done on the *2-step iterative relaxation* method. A graphical representation of the three steps used to parallelize the proposed methods for the Intel processor is presented in Figure 33. Here, the idea is to first divide the workload by elements in order of appearance in the mesh file (to minimize the amount of processor synchronization) across multiple cores. The second step is to use a thread pool to compute iteratively the *2-step iterative relaxation* method while reducing the overhead of thread management. Finally, the results from different threads are summed per iteration, and a convergence check is preformed.

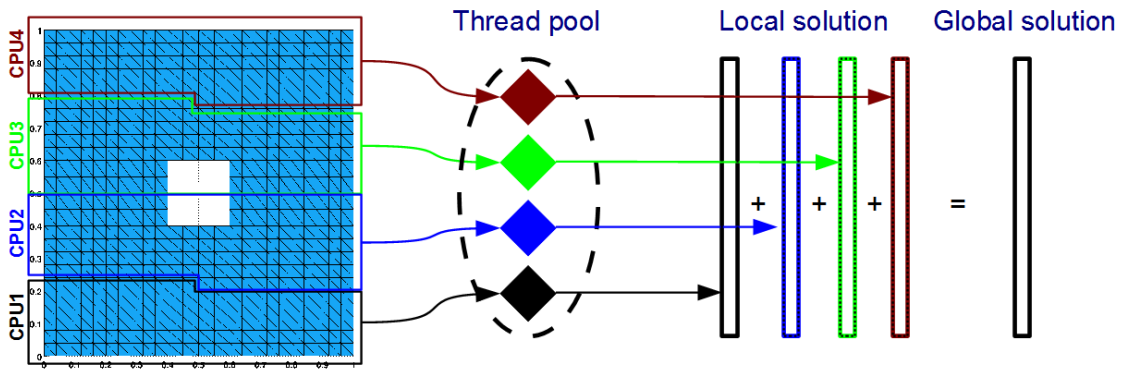


Figure 33: Multicore implementation of the FEM-SES method for the Intel processor.

Timing results for the parallel version of the proposed FEM-SES method running in four Intel cores are shown in Figure 34. For the smaller problem cases no performance benefit is obtained because there is not sufficient parallel work to do to overcome the overhead of threading and synchronizations. However, with the increase of the mesh size an average speedup of 3x is obtained with respect to the sequential FEM-CG implementation. Although, these results are encouraging the FEM-SES method provides a greater amount of parallelism than what can be actually exploited with multicore processors.

As was described earlier each element in the FEM can be computed independently, after which the individual node results are averaged. Thus, it is expected that manycore processors are a hardware architecture better suited to exploit the parallelism available in the FEM-SES method. The next section describes a GPU adaptation of the FEM-SES method and results for two different generations of NVIDIA graphic processors.

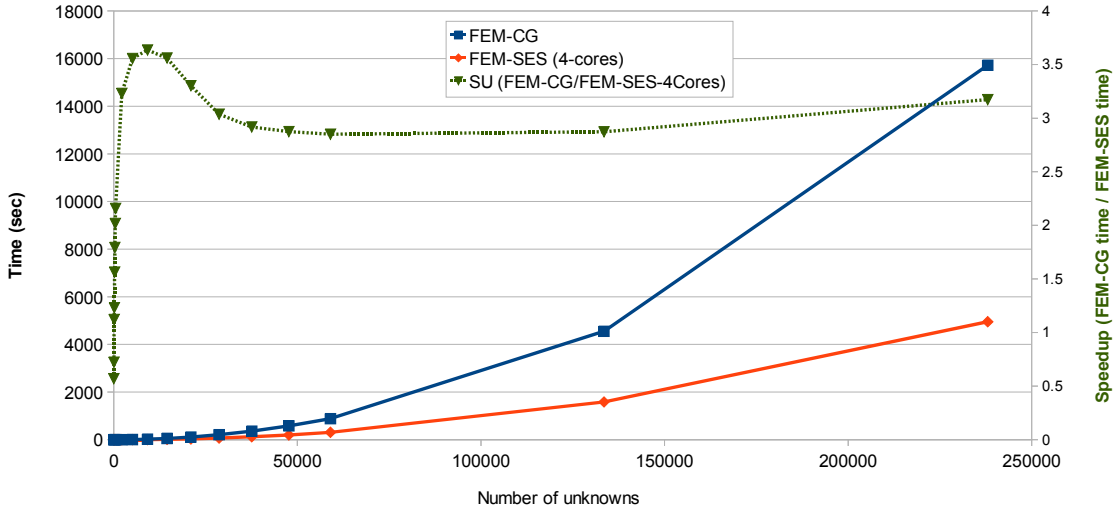


Figure 34: Performance comparison of the proposed FEM-SES method running on four Intel cores compared to a single core vectorized FEM using CG algorithm. The blue and red lines indicate the time of the two versions of FEM, and the green dotted line represents the speedup of the parallel FEM-SES method with respect to the single core optimized FEM-CG.

5.3.2 Manycore Adaptation of the FEM-SES Method

Before presenting the methodology used to implement the FEM-SES method on GPUs and the results obtained, an overview of the architecture and general GPU programming approach is presented.

a) Overview of GPU Architecture

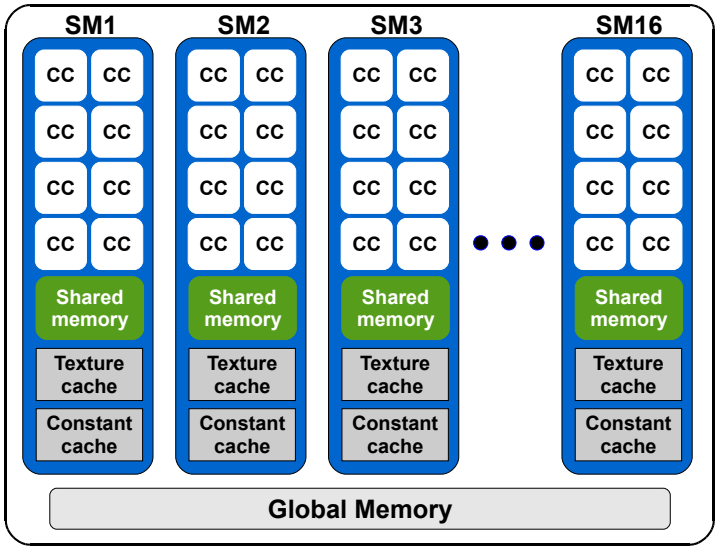
In this work two NVIDIA graphic cards are used to accelerate the FEM-SES method. The first GPU is the GT8800 a first generation CUDA enabled GPU; and the second, is the GTX485 classified as the third generation CUDA enabled GPU. An overview of their architecture features is presented next. NVIDIA GPUs are composed of hundreds of scalar processors called CUDA cores that execute

the device kernels. CUDA cores are arranged into clusters of 8 to 32 cores (depending on the GPU generation, see Table 8) called streaming-multiprocessors (SMs). Threads are scheduled to run on the CUDA cores in groups of 32 called warps.

These clusters of CUDA cores have access to varied memory hierarchy ranging from registers, local memory, shared memory, constant/texture memories, and global memory. On the top of the hierarchy we have large registers files per SM varying from 8KB to 32KB, which along with the shared memory determine the number of simultaneous threads that an SM can allocate. Each thread in the SM has a private local memory used for local variables, register spills and function calls. Shared memory is a small multibank low latency memory controlled by the programmer (i.e. scratchpad memories, similar to local store in the Cell SPEs), that allows fast access to commonly accessed data shared among threads of the same block. Texture and constant memories are special types of shared memories with different types of memory addressing modes that are cached. These memories are designed to provide fast access to immutable data. Finally, global memory is a large long access latency memory used to store all data required by a kernel execution. This basic architecture was originally implemented in the NVIDIA G80 series (see Figure 35).

Table 8: Comparison of architectural features for the GeForce 8800 GT and the GeForce GTX 480 GPUs. In the Fermi GPU, shared memory and L1 Cache share a common space of 64KB.

GeForce 8800GT (G80)	GeForce GTX480 (Fermi)
CUDA Capability: 1.1	CUDA Capability: 2.0
Clock frequency: 1.5GHz	Clock frequency: 1.4GHz
Number of SMs: 14	Number of SMs: 15
CUDA *cores*: 112 (8 per SM)	CUDA *cores*: 480 (32 per SM)
Register file: 8KB	Register file: 32KB
Shared memory: 16KB	Shared memory: 16KB/48KB
No L1 Cache	L1 Cache: 16KB/48KB
No L2 Cache	L2 Cache: 768KB
Global memory: 512MB	Global memory: 1.5GB
Single *warp* issue	Dual *warp* issue
Single-precision floating-point (SPFP)	Double and single precision FP
Peak SPFP performance: 128 multiply add ops/clock	Peak SPFP performance: 512 fused multiply add ops/clock



CC: CUDA *cores*

Figure 35: Basic architecture of NVIDIA's first generation G80 CUDA enabled graphic cards.

The latest generation of CUDA enabled graphic cards called Fermi introduced several technological advancements [94] compared to the previous generation of NVIDIA GPUs. Full data cache support for the memory hierarchy, with a configurable Level 1 (L1) cache and a unified L2 cache and a dual warp scheduler per SM, which allows issuing two instructions to two separate warps at the same time. This feature is supported by an increase in the number of CUDA cores from 8 to 32 cores (divided in two groups of 16-cores, one per warp), and increasing the shared memory space up to 48KB. Also the different memory types now have a unified address space which provides full support for C/C++ pointers and full 64-bit addressing. Fermi has faster single and double precision processing with full IEEE 745-2008 support for both representations (as opposed to IEEE 754-1985 for previous generations of GT200). Among other architectural advances, atomic operations are faster, full memory support or error correction codes for critical applications has been included, as well as the ability to execute multiple kernels concurrently. Figure 36 shows the block diagram depicting the architecture of NVIDIA Fermi GPUs, and Table 8 compares some of the most important architectural features of GTX480 vs. GT800 GPUs from NVIDIA.

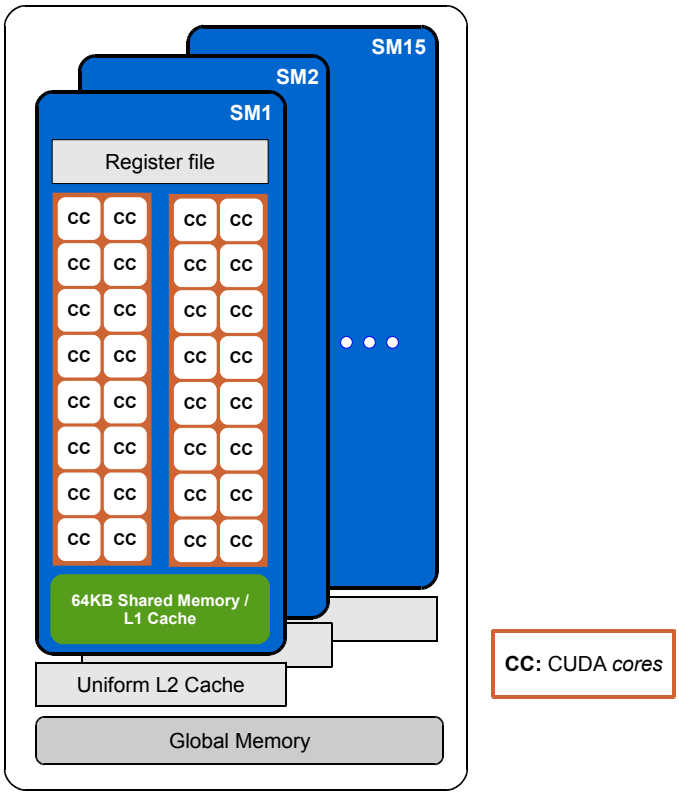


Figure 36: Basic architecture of NVIDIA's third generation FERMI graphic cards.

A CUDA kernel launches thousands of threads which are organized into groups called *blocks* that compute the same code on independent data-sets, a model that is referred to as Single-Instruction, Multiple-Thread (SIMT). Blocks are independent and can execute in any order, thus having independent data-sets is important. Threads inside a block communicate using shared memory and can be synchronized, but threads across blocks can only communicate through global GPU memory using atomic operations. In both cases, judicious use of synchronizations and atomic operations should be exercised since they may become performance bottlenecks.

Programming a CUDA kernel involves identifying the data-parallel compute intensive sections of the application to be offloaded onto the GPU (also called device). The programmer must then specify how many threads per blocks and how many blocks per kernel should be used. Executing a GPU kernel involves sending the required data to the GPU, launching the actual CUDA kernel, and transferring results back to the host CPU. Once a CUDA kernel is launched, each SM is allotted several thread blocks that are executed sequentially in an arbitrary order. The SMs then schedule warps from a block to be executed. In particular, warps have been designed to execute the same code in a lock-step manner (similar to vector processors), but they provide flexibility to allow different execution paths (also called thread divergence) at the expense of performance reductions.

b) GPU Parallel Implementation

The CUDA 3.2 SDK [8] was used to implement the *2-step iterative relaxation* algorithm on the GPU. The host function first defines the global device memory required to store the element matrices K_e (including their right hand sides b_e), the global unknown vector φ, and the pre-computed weight factors w_e [17]. These values are then transferred to the GPU global memory. Next, the host function loops over the three device kernels (GPU kernels) that parallelize different

[17] The subindex “e” is used to refer to the matrix or vector of a particular element. Whenever this index is not used then it is assumed that the matrix or vector correspond to the whole mesh.

sections of the method until the convergence criteria are satisfied (see Figure 37). The shaded process boxes in Figure 37 correspond to each of the CUDA kernels. Once convergence is achieved the host function exits the loop and transfers the global solution φ back to host memory. Large data transfers only occur outside the loop minimizing the effects of global memory access latencies. Only single scalar values are transferred inside the loop (illustrated in Figure 37).

Kernel 1 computes the solutions of each element in parallel as described in step 1 of Figure 28. Each thread in the kernel computes the solution of one element and stores it in global memory. Each block in this kernel consists of 256 threads and the number of blocks in the grid is computed dynamically at runtime depending on the problem size which equals the number of elements in the FEM mesh divided by the block size. Most of the memory accesses are coalesced due to sequential addressing of the K_e, b_e and w_e data sets but non-uniform and indirect access will still be required for the unknown φ-vector. The indirect accesses are one of the main performance limiting factors in this kernel. To minimize the effects of accessing the φ-vector (which is used several times in a thread), it is stored into shared memory. Techniques such as loop unrolling and variable reuse are also used to enhance performance. The above mentioned kernel is the most time consuming of the three, Figure 38 presents the kernel code.

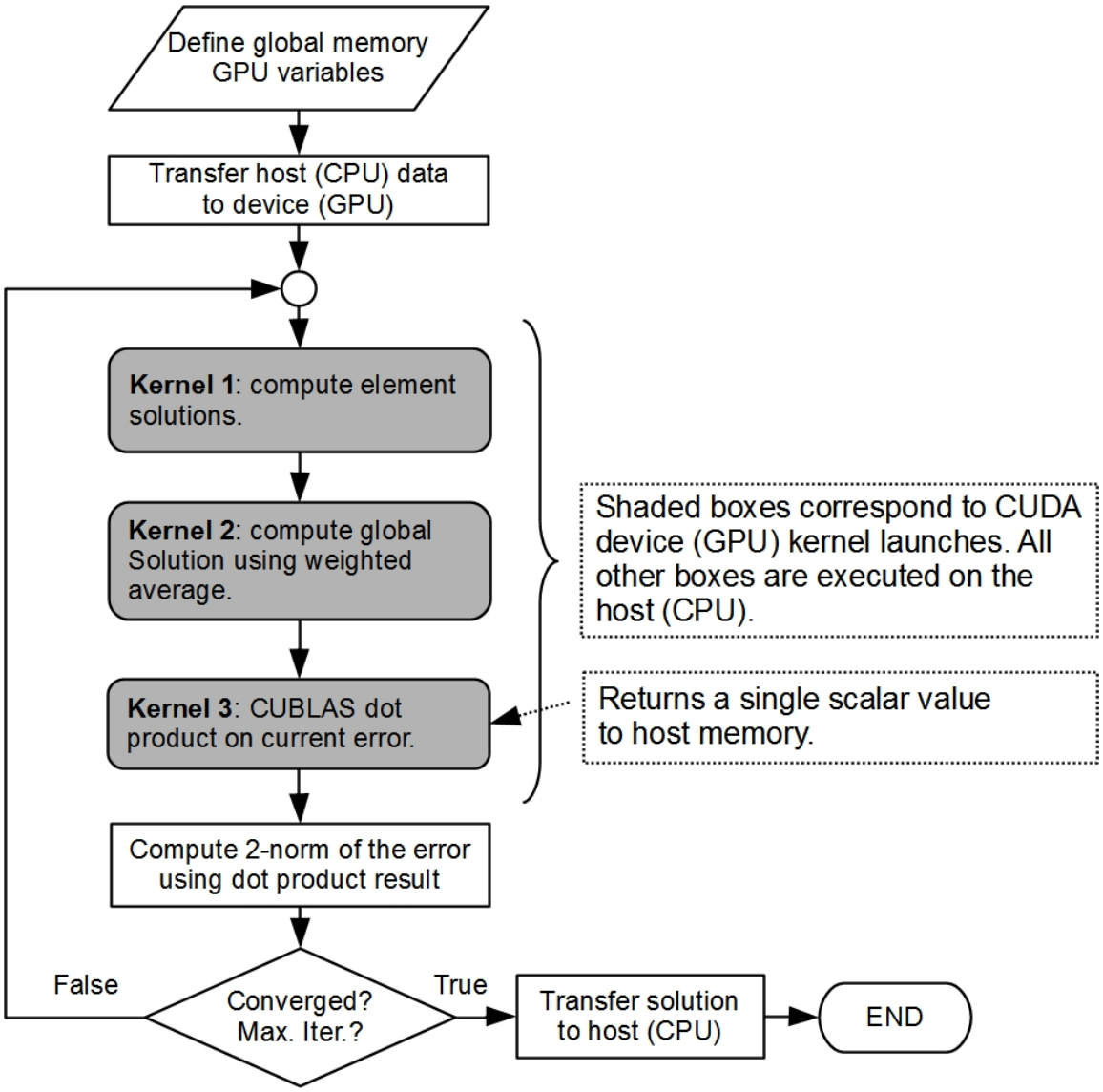


Figure 37: Workflow to parallelize the 2-step iterative relaxation method on NVIDIA GPUs.

```
1: __global__ void FEM_coalesed_solver_kernel(
       int num_elements, float *K_dis, float *b_dis, int
       *fem_nodes, float *w, float *phi, float *d_odata) {
2: // Local memory.
3:    float temp1;
4:    int data_disp, int data_disp3;
5: // Shared memory.
6:    __shared__ float temp2[256*3];
7: // Use block and thread IDs to select the element.
8:    data_disp = blockDim.x*blockIdx.x + threadIdx.x;
9:    if (data_disp < num_elements) {
10:      data_disp3 = data_disp * 6;
11:      data_disp = data_disp * 3;
12: // Load x to shared memory (partially coalesced).
13:      temp2[threadIdx.x*3] = phi[fem_nodes[data_disp]];
14:      temp2[threadIdx.x*3+1] = phi[fem_nodes[data_disp+1]];
15:      temp2[threadIdx.x*3+2] = phi[fem_nodes[data_disp+2]];
16: // Coalesced access to all variables.
17: // Node 1
18:      temp1 = b_dis[data_disp];
19:      temp1 -= K_dis[data_disp3]*temp2[threadIdx.x*3+1];
20:      temp1 -= K_dis[data_disp3+1]*temp2[threadIdx.x*3+2];
21:      d_odata[data_disp] = temp1 * w[data_disp];
22: // Node 2
23:      temp1 = b_dis[data_disp+1];
24:      temp1 -= K_dis[data_disp3+2]*temp2[threadIdx.x*3];
25:      temp1 -= K_dis[data_disp3+3]*temp2[threadIdx.x*3+2];
26:      d_odata[data_disp+1] = temp1 * w[data_disp+1];
27: // Node 3
28:      temp1 = b_dis[data_disp+2];
29:      temp1 -= K_dis[data_disp3+4]*temp2[threadIdx.x*3];
30:      temp1 -= K_dis[data_disp3+5]*temp2[threadIdx.x*3+1];
31:      d_odata[data_disp+2] = temp1 * w[data_disp+2];
32: }}
```

Figure 38: Kernel 1-CUDA kernel used to compute the single element solutions in FEM-SES.

After the local element solutions are obtained, Kernel 2 is called to compute the global node solutions using an average sum. Again, the host function launches a 1 dimensional grid with 256 threads per block. The number of blocks launched is also computed in runtime and depends on the connectivity among elements. In this kernel, each thread is assigned the task of gathering results for one node. Once the new results estimate is computed, the error between the

new estimate and the previous one is determined and stored to device global memory.

The third kernel partially computes the *2-norm* of the error. This is done by calling the `cublasSdot` function from NVIDIA the CUBLAS [95] library which returns the dot-product of this vector to the host function. The host function then computes the square root to obtain the final value of the 2-norm, and finally convergence is assessed. Kernels are designed to avoid using synchronizations primitives to minimize execution bottlenecks.

c) GPU Results

Al results presented in this subsection were obtained for single precision floating point (SPFP) since the 8800GT graphics card only supports this precision, and in order to have a common basis for comparisons for all platforms. Future implementations will evaluate double precision performance. The timing in the GPU code was done using CUDA events that were found to be more accurate than the CUTIL timers [8]. Optimizations were used to compile the GPU codes (as well as the CPU code as described in Chapter 3), while the "–arch=sm_20" flag was used with the NVCC compiler to enable FERMI advanced architecture features. The best timing results are shown for each version (CPU and for each GPU), and the algorithm is made to stop for a precision of 1e-3 with respect to the error in the unknown vector for two previous iterations. The coaxial

cable mesh was refined several times to increase the problem size. Table 9 shows the data corresponding to the different mesh sizes used as well as execution times for a vectorized single core Intel implementation and the two GPUs. Here the number of unknowns is represented by the column labelled Nodes.

Timing results for the reference sequential implementation of the classic FEM (with an efficient conjugate gradient iterative solver) used in section 5.2 were compared to the time results for the best performing GPU implementation. The speedup results for the FEM-SES Fermi implementation over the classic FEM implementations are shown in Figure 39, demonstrating that FEM-SES outperforms the classic FEM implementation with a better scaling behaviour. Next, the solution to the FEM was parallelized using an efficient parallel matrix-vector multiplication in the conjugate gradient solver referred, using up to four processing cores on the Intel CPU with no significant time benefits mainly because of the test problem size and insufficient amount of parallel resources in the CPU.

The time ratio results for the solution of the *2-step iterative relaxation* method on the CPU relative to the two GPU times are shown in Figure 40.

Table 9: Finite element mesh dimensions for the 2D coaxial cable test case and solution times in seconds (last three columns) for the 2-step iterative relaxation method.

Elements	Nodes	Iterations	CPU	8800GT	GTX480
1120	504	327	0.012	0.316	0.793
4640	2204	1227	0.196	0.461	0.878
10560	5104	2626	0.975	0.929	1.137
18880	9204	4475	3.013	2.095	1.475
29600	14504	6755	7.744	4.349	1.864
42720	21004	9433	15.268	8.243	2.322
58240	28704	12503	28.659	14.473	3.067
76160	37604	15938	48.508	23.716	4.028
96480	47704	19739	77.197	36.817	5.501
119200	59004	23877	116.545	54.703	7.201
268800	133504	49486	554.643	250.002	23.273
478400	238004	82632	1663.66	744.573	60.415

As expected, these results show that for smaller problems there is not sufficient parallel work to benefit from the GPU implementations; this in addition to the large caches in modern microprocessors allow the CPU version to outperform the GPU ones for the first two meshes tested (see Table 9). On the other hand, as the problem size grows, even for relatively small problems, the GPU speedups over the CPU times become apparent. A speedup of up to **2.23**x times over the CPU solution times is obtained for the 8800GT and up to **27.53x** times for the GTX480. It is important to point out that the times reported in Table 9 include all data transfers from the CPU to the GPUs and vice versa.

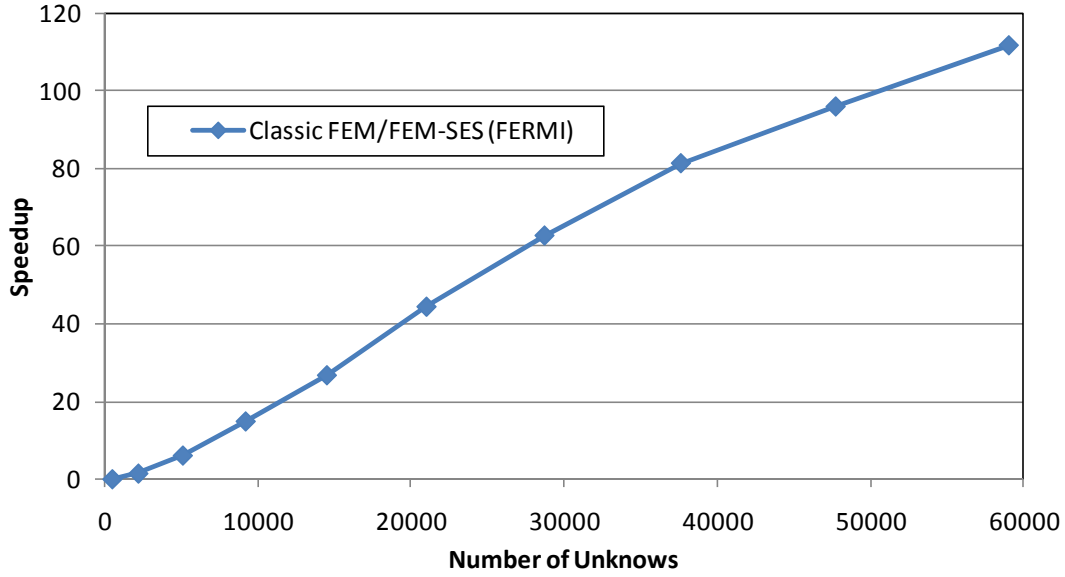


Figure 39: Speedup of the Fermi GPU FEM-SES implementation versus the classic FEM sequential implementation on the Intel CPU using a CG solver.

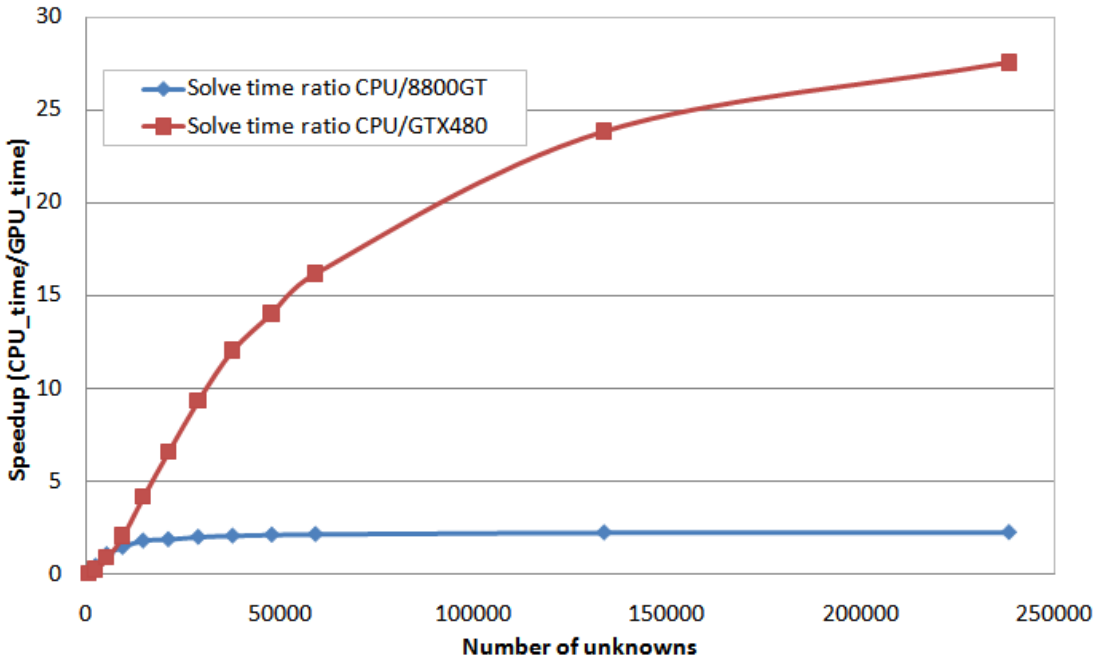


Figure 40: Speedup of the GPU solution times with respect to the CPU times.

Table 10: Limiting performance factors for the GPU kernels.

Kernel 1	Occupancy	Limiting Factor	Resources
8800GT	0.666	Registers	7168 out of 8192
GTX480	0.833	Registers	30720 out of 32768
Kernel 2	**Occupancy**	**Limiting Factor**	**Resources**
8800GT	0.666	Registers	5632 out of 8192
GTX480	1.000	None	32768 out of 32768
Kernel 3	**Occupancy**	**Limiting Factor**	**Resources**
8800GT	0.666	Registers	7168 out of 8192
GTX480	0.666	Block-size	blocks of 128 threads

Table 10 shows the limiting factors for each of the three CUDA kernels developed for each graphic card. The main limiting factor for the three kernels that ran on the first generation 8800GT graphic card is the number of registers available per SM. Only 66.6% of the available computing resources are used due to the small register file per SM on this graphic card, limiting the performance which accounts for the plateau of the speedup in Figure 40.

This is not the case for the newer generation GTX480. For the GTX480 kernel 1 was limited by the number of registers but a higher utilization factor of 83.3% of the resources is attained. Kernel 2 exhibits no limiting performance factors achieving a 100% use of parallel resources, while for kernel 3 only 66.6% of resources are used. In this last kernel the limiting factor is no longer the number of registers available but the actual block size used (128 threads/block) as reported by the CUDA profiler. Kernel 3 uses the `cublasSdot` function from

CUBLAS library to compute the dot-product of the error-vector in our implementation, which provide the best timing results but does not allow us to control the resources used in this kernel.

The average time consumed by each of the three kernels for the biggest mesh sizes used is presented in Figure 41. These times show that kernels 1 and 3 dominate overall execution. This behaviour is in agreement with the resource utilization explained before. Although kernel 1 implements several code optimization techniques (coalesced memory accesses, shared memory usage and loop unrolling), it is still limited by the amount of register (hardware resources) available. Kernel 3 computes a dot-product which requires a reduction operation (a common bottleneck in scientific kernels); this kernel is implemented using the built in functions in the CUBLAS [95] library.

Finally, an optimized CPU code using loop unrolling, data alignments to natural memory boundaries and compiler vectorization (as explained in the previous two chapters) was compared with the two GPU runs. Table 11 presents the performance results of the FEM-SES method for the CPU and GPUs using the biggest mesh. These results clearly show that the code running on GTX480 can be efficiently parallelized outperforming modern CPUs even with aggressive hand coded optimizations.

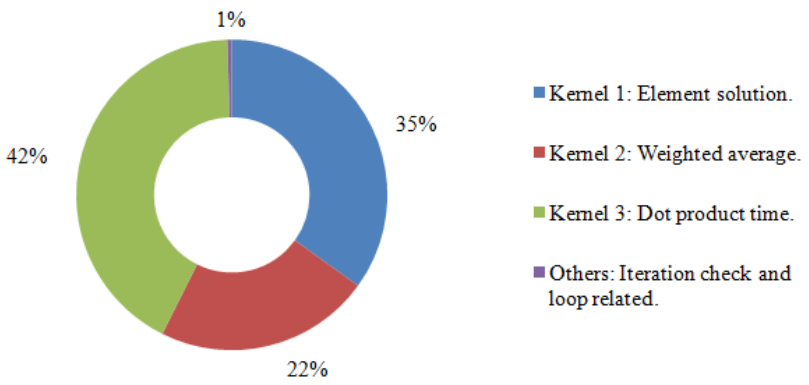


Figure 41: Time distribution of the kernels used to implement the 2-step iterative relaxation method.

Table 11: Performance results for the FEM-SES method of the hand optimized CPU code versus the GPU implementations.

	CPU	8800GT	GTX480
2-step iterative relaxation	828.255	744.573	60.415
GTX480 Speedup	13.71X	12.32X	1X

5.4 Related Work

To the best of our knowledge the closest effort to implement a finite element method based on a decoupled solution is presented by Bastos *et al.* in [96] called *N-scheme*. The *N-scheme* computes the solution of each mesh nodes independently based on the current value of the neighbour nodes connected to it without building a global coefficient matrix. FEM-SES is fundamentally different since it computes the solution based on solving the element stiffness matrices, and the resulting solutions are coupled to obtain the global solution. In [97], Eyng *et al.* implement a parallel version of the *N-scheme* method using 4 cores (out of

168 cores) of a Mirynet 2G cluster. This second work presents only preliminary results for a small test case (2362 nodes and 4153 elements) that yielded similar performance as that of the sequential implementation. The results obtained here for the parallel FEM-SES (see section 5.4) show that indeed for small problems there is not enough work for parallel systems to exhibit any performance benefits. On the other hand, this work demonstrates how the proposed method exhibits good parallel performance scaling behaviour for bigger test cases.

Other previous efforts have successfully implemented accelerated versions of the finite element method on graphic processors. Some of the most important ones are presented next and compared to our proposed method. An early work by Göddeke et.al [98] uses a mixed precision defect correction algorithm to solve the linear systems derived from FEM obtaining up to 2.3 times faster performance than the CPU version alone. The referred work uses single precision computations on the early graphic processors (with no double precision support) to obtain the solution of the linear system, and then an outer loop executed by the CPU in double precision corrects the solution. Modern graphic cards now support native double precision arithmetic, thus no longer requiring this type of indirect computations to achieve double precision accuracy. Nevertheless, further studies could be done to identify the potential performance gain of using these methods considering the faster single precision operations. A

double precision of the FEM-SES method will be developed on GPU to assess its performance compared to both double and single precision CPU and GPU code.

In [99] two approaches to accelerate FEM are presented by assembly of the global coefficient matrix in the GPU, demonstrating up to 15 times gains in performance. The FEM-SES method does not require building the global coefficient matrix thus importantly reducing the computation cost associated to it. Moreover, our approach allows building the elements coefficient matrices in the parallel compute cores where they will be used which will benefit distributed and multicore implementations of the FEM-SES method. Another interesting work that accelerating the assembly process of FEM is presented by Cecka et al. [100]. Here several methods to parallelizing the assembly process are presented and evaluated on two different GPUs. Up to 65 times speedup is shown for the assembly of unstructured finite element meshes with first order basis functions, and the performance degrades as the order of the basis functions increase.

Multi-GPU efforts to accelerate the FEM have also been addressed in [101-103]. The first work implements the discontinuous Garlekin FEM on a GPU cluster using asynchronous concurrent executions which obtained up to 18 times the performance of an 8-node cluster of quad core CPUs. In the second work, a commodity cluster with heterogeneous hardware resources (nodes with CPU and GPU) is used to enhance the performance of a finite element framework,

showing significant performance increases even when using older generation graphic cards. The third work proposes a parallelization approach with a dynamic load balancing that combines task partitioning and stealing method to efficiently exploit both CPU and GPU processors. Multi-GPU implementations are one of the main subjects of our future work

5.5 Concluding Remarks

This chapter present two major contributions. The first contribution is a new element-based solution technique for solving the finite element method called FEM-SES well suited for parallel processing. This idea was first presented in [104] as a new approach for computing the finite element method in parallel processors, but only sequential proof-of-concept results were shown at that time.

The second contribution is a methodology for implementing the proposed FEM-SES method to exploit the parallel computing power of modern graphic processors. The goal of designing the FEM-SES method is to expose more parallelism in the finite element method compared to traditional approaches. The method was implemented in two different generations of NVIDIA GPUs obtaining up to 27.53 times speedup on the GTX480 compared to compiler optimized CPU results. Even for a hand optimized CPU code with optimizations such as loop unrolling and vectorization, our GPU implementation still achieves 14 times faster performance when using the GTX480 graphic card.

Although a simple test case is used to validate the FEM-SES method in this work, we believe that it will converge to correct results for a broader type of problems where the Jacobi method is applicable. Such statement is made based on the fact that a Jacobi update scheme is used in the *2-step iterative relaxation* method to obtain the decoupled element solutions.

Chapter 6
Conclusions and Future Work

The constant demand for more detailed models and greater precision in electromagnetics (EM) simulations has defined a clear trend towards solving increasingly complex computational EM problems that has relied, in part, on continual CPU improvements; however, current technological limitations in hardware construction (in particular power dissipation problems, and frequency related problems) have dictated the need to explore new alternatives.

Lead hardware architects and manufacturers have defined an irreversible shift in paradigm towards parallel multicore processors as the means to overcome many of today's hardware limitations. This parallel trend has been accelerated even more with the recent popularity gained by massively parallel-manycore GPUs for general purpose computing (GPGPU) that also require parallel programming models. This problem is also patent in clustered systems that have traditionally relied in the increasing performance of previous single core processors and communications frameworks to provide an aggregate high performance clustered system. Moreover, existing frameworks that are useful in parallelizing dense linear systems usually cannot handle complex data structures required for sparse linear system that often arise in EM simulations. This means

that EM practitioners must explicitly consider parallel algorithms and techniques in order to efficiently exploit the full potential of emerging computing processors.

This thesis focuses on accelerating the sparse matrix-vector multiplications (a dominant compute intense kernel in EM computations) and shows that by rethinking the way sequential sparse algorithms are implemented and transforming them into dense-like algorithms, memory access patterns are enhanced, computations become regular (enabling short-vectorization of the kernel), also facilitating the partitioning load balancing of the SMVM kernel, leading to greater performance. Similar ideas are applied to the finite element method which also leads to exposing the intrinsic parallelism in the algorithm and the expected performance enhancements.

6.1 Original Contributions

In this work five original contributions are presented, three to accelerate the SMVM algorithm and two for the finite element method:

- A new sparse matrix format called *pipelined-matched sparse* (PMS) representations well suited for streaming architectures (e.g. Cell BE, FPGAs, etc.) that enables efficient vector processing for the SMVM algorithm, implementing the SMVM kernel as a dense kernel (opposed to the traditional sparse implementation), and facilitates matrix partition and load balancing.

- A second sparse matrix format called *blocked pipeline-matched sparse* (BPMS) representation that solves the scalability problems of the PMS format for cache-based architectures, but retaining similar benefits as PMS.
- A *2-level partitioning scheme* and modified SMVM kernels that vectorize and exploit parallel cores in the two multicore architectures used. The optimized PMS kernel on the Cell-SPE processor demonstrated average performance benefits of 3.5X faster than the Intel SMVM-CSR, 2.6X over Intel SMVM-PMS, and nearly 14X faster than the SMVM-CSR implementation in the Cell-PPE for the largest test matrix. Similarly, the optimized SMVM-BPMS kernel running on the Intel processor demonstrated 2.5x performance enhancements with respect to BCSR and 2.2x faster than PMS on average.
- A new element-based solution technique for solving the finite element method called FEM *single element solution* (FEM-SES) well suited for parallel processing, that decouples the element solution of the FEM mesh so that they can be computed in parallel.
- A methodology for implementing the proposed FEM-SES method to exploit the parallel computing power of modern manycore processors (GPUs) with performance increases from 27.53 times speedup on the

GTX480 compared to compiler optimized CPU results to 14 times on an aggressively hand optimized CPU code.

The optimizations techniques presented throughout this work have been applied to specific hardware architectures to demonstrate their performance benefits; however, they are general in nature, thus holding greater theoretical value and broad applicability to different parallel architectures.

All the software developed to implement and test the contributions presented above has been collected and organized in a modular, flexible and easily extensible C library that can be viewed as an additional contribution to this work. Libraries such as the one proposed in this work (among other tools, e.g. parallel compilers and programming languages) have been identified [105] as one of the key tools that will be required to take advantage of the current multicore trend that is expected to dominate computer architectures for the foreseeable future. This library will be made available freely through a website at McGill.

6.2 Future Work

The first three contributions presented are related to accelerating the SMVM algorithm on different types of multicore processors. The computations were mainly done in single precision since the SPE cores in the Cell BE processor used are not efficient in double precision arithmetic and to keep the comparisons fair with the Intel architectures all computations were done in single precision. So

a natural extension of this work is to implement double precision versions of the algorithms proposed, some of which have already been done and included in the C library produced.

The next extension that is currently being studied is the multicore acceleration of the preconditioned iterative methods. The effectiveness of the proposed techniques for multicore processors was demonstrated in this work for the unpreconditioned conjugate gradient method; however, most iterative methods are used with a preconditioner to accelerate their convergence rate. Accelerating the preconditioning step in iterative solvers is crucial, as commented in Chapter 2, since along with the SMVM kernel they represent the two dominant kernels in iterative solvers. Of particular interest, as an extension to this work, are polynomial preconditioners that are commonly applied in the iterative method as successive SMVM operations that can be done efficiently in multicore with the techniques proposed here. The C library provided contains several CG implementations as well as some preconditioned CG algorithms and provides support for sequential application of Jacobi, Choleski and a simple polynomial preconditioner.

A third direction proposed is related to the last two contributions. The proposed FEM-SES proved to effectively accelerate the overall execution time of the finite element method even though the method has slow convergence. Thus,

the next step in this research will be to evaluate different ways to accelerating the *2-step iterative relaxation* method by either finding better update scheme or modifying the method itself.

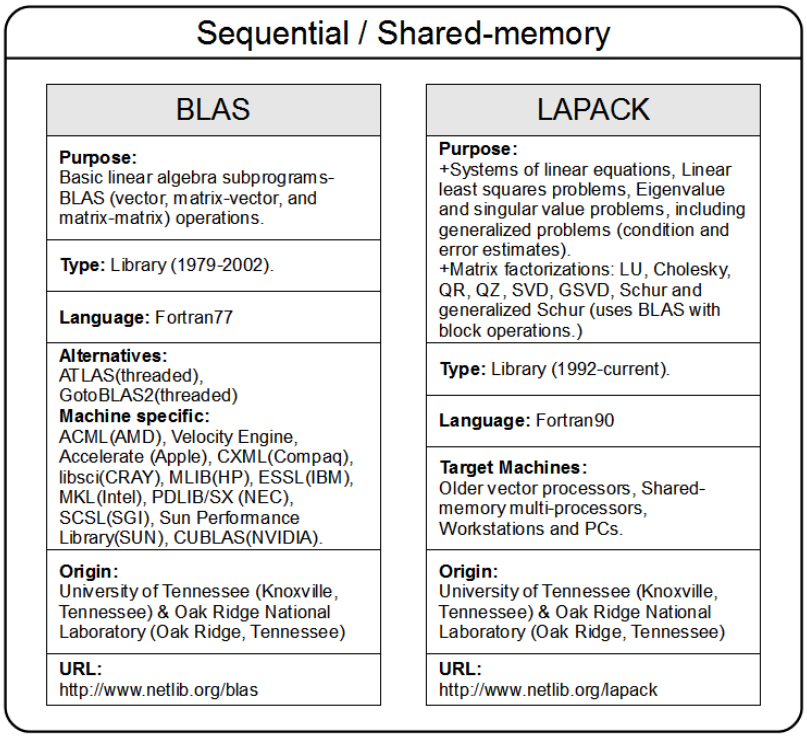
The final future work direction suggested in this thesis is the continued development of the matrix library proposed in this work to optimize the functions that have not yet been optimized and enhance its functionality. The intention is to provide an in-house software product that will offer current and future developers a base platform to run sparse algorithms with different optimizations, and test new research ideas while extending its functionality.

Appendix A:
Compilation of Modern Matrix Libraries

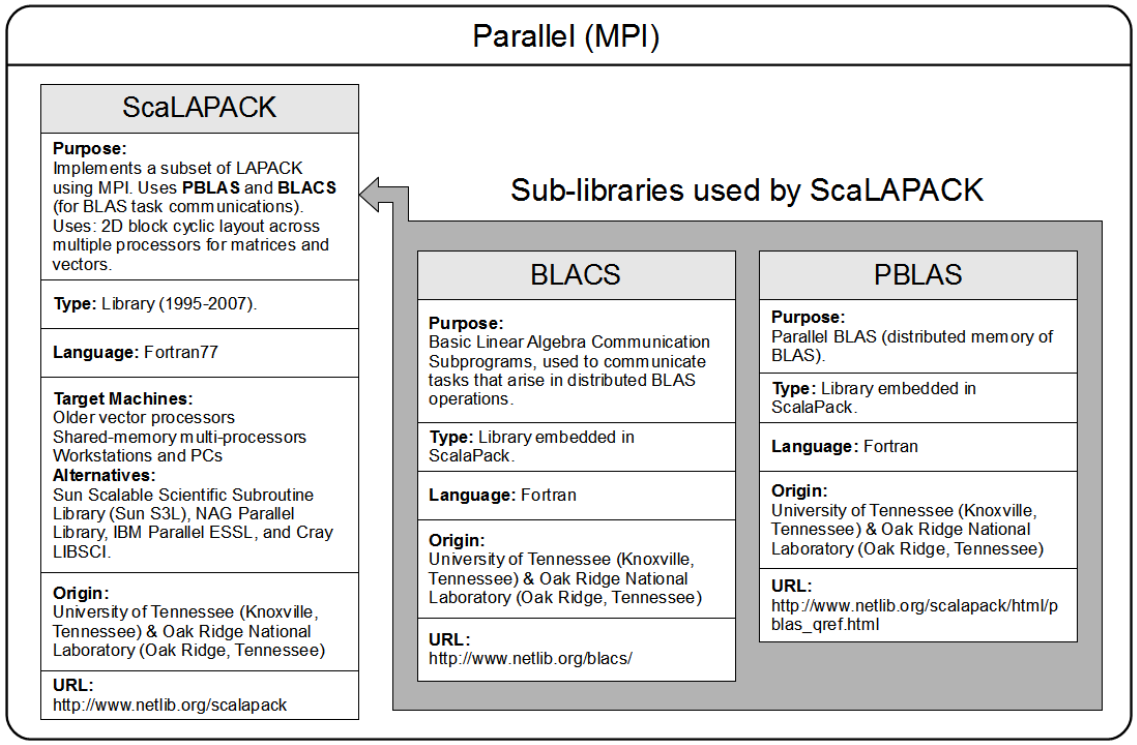
This appendix presents a brief overview of the most widely adopted matrix libraries classified according to the concurrent model implemented. Table 12 shows the classification of these libraries and Figure 42 and Figure 43 present summary information for each of these libraries. It is important to note that many of these libraries do not provide multithreading support for multicore processors and none exploit short-vector (SIMD) processing. The references made to vector processing in some of the libraries refer to older vector processing machines as described in [24], and not for the type of short-vector units found in modern processors.

Table 12: Classification of sparse matrix libraries according to the concurrent paradigm implemented.

Matrix Type / Sequential / Parallel	Dense	Sparse
Sequential (Seq.) Shared memory (SM)	BLAS (Seq.) LAPACK (Seq.) GotoBLAS (SM) Atlas (SM)	Sparsity Oski Sparselib++ Csparse ITSOL(SparseKit)
Distributed memory (MPI)	SCALAPACK (PBLAS, BLACS)	PSBLAS pARMS

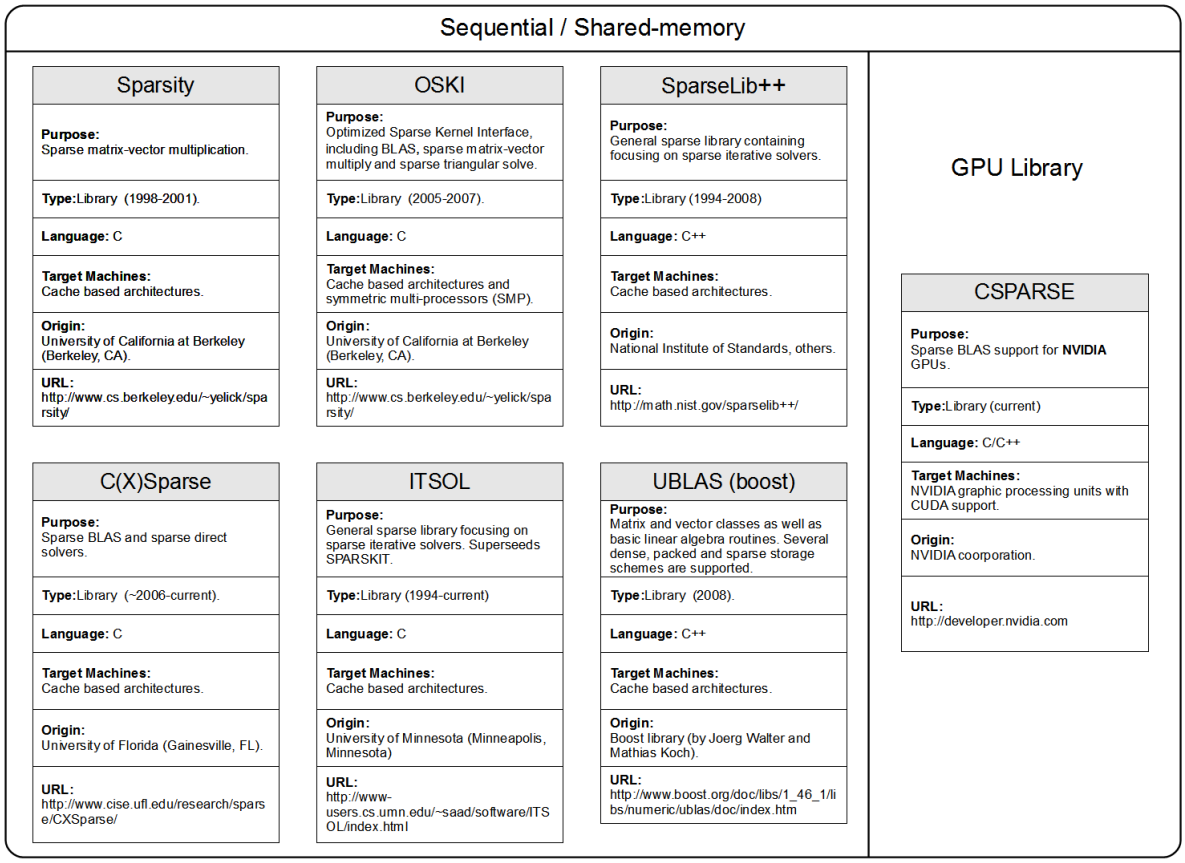


(a) Sequential and shared memory libraries.

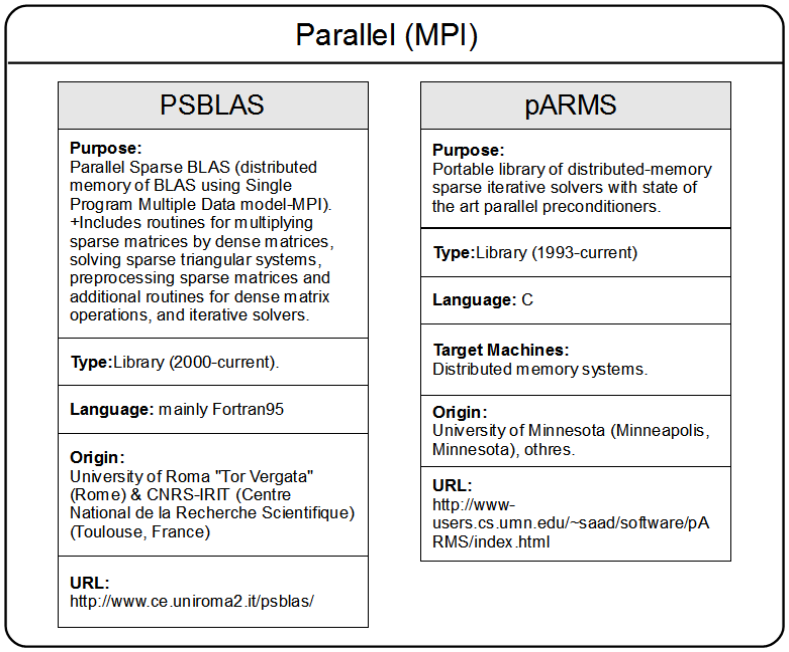


(b) Parallel distributed memory (MPI) libraries.

Figure 42: Dense matrix libraries classification.



(a) Sequential and shared memory libraries.



(b) Parallel distributed memory (MPI) libraries.

Figure 43: Sparse matrix libraries classification.

A compilation of some of the most important iterative and direct solver libraries for sparse systems is also presented for reference in Figure 44 for completeness. No classification has been done in this case because of the broad spectrum of targeted systems and methods implemented.

SuiteSparse
Purpose: Collection of sparse matrix libraries containing BLAS, direct solvers, multifrontal methods, and others.
Type: Library collection (current)
Language: mainly C
Target Machines: Cache based architectures.
Origin: University of Florida (Gainesville, FL).
URL: http://www.cise.ufl.edu/research/sparse/SuiteSparse/

PETSc
Purpose: Parallel solution of scientific applications modeled by partial differential equations (MPI-Sparse)
Type: Library and PDE application solvers (1995-current).
Language: C
Target Machines: Clustered systems with GPU support.
Origin: Argone national laboratory (Chicago).
URL: http://www.mcs.anl.gov/petsc/petsc-as/

SPOOLES
Purpose: General sparse library containing direct and iterative solvers. Some multithreaded and MPI support.
Type: Library (1999)
Language: C
Target Machines: Various.
Origin: Boeing Phantom Works.
URL: http://www.netlib.org/linalg/spooles/spooles.2.2.html

TAUCS
Purpose: Sparse direct and iterative linear solvers.
Type: Library (up to 2003)
Language: C
Target Machines: Support for threaded.
Origin: Tel-Aviv University (Tel-Aviv, Israel)
URL: http://www.tau.ac.il/~stoledo/taucs/

HIPS
Purpose: Efficient parallel iterative solver for very large sparse linear systems (based on a Schur complement approach and they systematically exploit blocking).
Type: Library (2004-current)
Language: C
Target Machines: Distributed memory systems.
Origin: INRIA Bordeaux-Sud-Ouest (France) and University of Minnesota (Minneapolis, Minnesota), others.
URL: http://hips.gforge.inria.fr/index.html

Cusp
Purpose: Sparse linear algebra and graph computations on CUDA, also has conjugate gradient.
Type: Library (2010-current)
Language: C/C++
Target Machines: NVIDIA graphic processing units with CUDA support.
Origin: NVIDIA (Nathan Bell and Michael Garland)
URL: http://code.google.com/p/cusp-library/

Figure 44: Sparse direct and iterative solver libraries.

Appendix B:
A Flexible and Portable Multithreaded Library for Sparse Matrix Computations (FPMSparseLib)

A library was organized with the implementations of all the algorithms presented in this thesis as well as complementary algorithms and general functions required for the testing and benchmarking done. All the programming was done using standard C language (chosen for its efficiency and portability), and the library organization was done so that it could be used in a modular and flexible way and easily extensible. A `makefile` is provided so the library can be easily recompiled with different compilers, compiler optimizations flags, and for different software platforms. Currently the `makefile` provided only supports cache based machines and NVIDIA GPUs, since the SPEs in the Cell BE processor require a different software setup not included in this library.

The proposed library is self contained, thus no software dependencies are required to install it in a new computing system with the exception of the CUDA SDK, if one desired to test the GPU enabled functions. The only other external libraries required not developed by the author is a small library (`mmio`) from the Matrix Market [42] repository used to read and write Matrix Market files. These files are included as part of the library built, but are only used for benchmarking the algorithms with Matrix Market matrices. Matrices that are obtained from a real

application or built by the user need not use this library, instead they should use the libraries built to read and write matrices to files provided in this library. Moreover, reading a Matrix Market matrix using the `mmio` library has been encapsulated in a function that automatically converts the matrix to CSR representation, the *de facto* representation used by almost all sparse matrix software. Figure 45 presents the organization of the sparse library by functionality and an overview of the functionality for each of the modules and files is given in the next subsection.

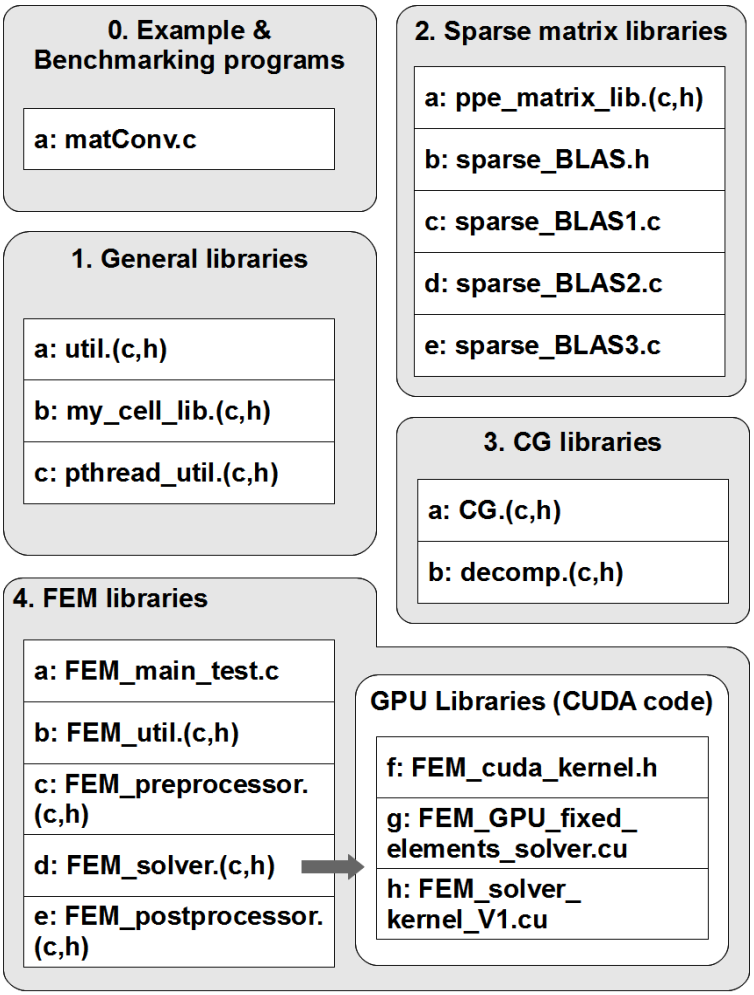


Figure 45: Library organization by functionality.

A brief overview of the functionality offered by each of the modules in the libraries is presented next, followed by a description of the sparse matrix data structure created, which is a central component in the proposed library.

B.1 Description of the Library and the Sparse Matrix Data Structure

The description of the library along with the file dependency relations are presented in Table 13 by modules, which are defined in the context of this work as groups of files with a common functionality.

B.2 Description of the Sparse Matrix Data Structure Used in the Library

Figure 46 shows the compressed sparse matrix data structure created for this work, which is a central component in the library. All of the functions in the library use this general data structure regardless of the sparse matrix format used, keeping a standard interface. The first three fields (lines 2-3) are used for all matrix formats, whereas the fields in lines 6 and 7 are used for blocked formats, and the field in line 5 is used to store the vector elements in the PMS format along with field in line 8 (that stores the number of vectors per matrix row). All of the fields described (from line 2 to 8) are pointers where any kind of primitive data type can be stored, and that are accompanied by corresponding pointer fields (ending in the word “free”) which are used by the memory allocation functions provided to align all data to the specified memory boundary (memory boundaries must be a power of two value) .

Fields in lines 26 and 27, store the matrix name and properties if some are desired. Future enhancements to the library may standardize the content of such pointers to use matrix properties in the code itself and not just as matrix information. The remainder of the fields in the data structure are self explanatory, and basically provide information on the matrix dimensions, total number of nonzeros with and without padding, block size used, total number of blocks, the type of data stored in the matrix (*integer*, *float*, *double*, *complex*) and the sparse format used (*compressed_format*).

```
1: typedef struct _compressed_array_t {
2:   data_type_ptr data;
3:   data_type_ptr col_idx;
4:   data_type_ptr row_idx;
5:   data_type_ptr vec_data;
6:   data_type_ptr block_ptr;
7:   data_type_ptr row_block_size;
8:   data_type_ptr subrows;
9:   data_type_ptr data_free;
10:  data_type_ptr col_idx_free;
11:  data_type_ptr row_idx_free;
12:  data_type_ptr vec_data_free;
13:  data_type_ptr block_ptr_free;
14:  data_type_ptr row_block_size_free;
15:  data_type_ptr subrows_free;
16:  unsigned int rows;
17:  unsigned int cols;
18:  long int nnz;
19:  unsigned int padded_nnz;
20:  unsigned int padding_factor;
21:  unsigned int padded_sub_rows;
22:  int block_size;
23:  int num_blocks;
24:  int data_format;
25:  int compressed_format;
26:  char *name;
27:  char *properties;
28: } c_array_t;
```

Figure 46: Sparse matrix data structure.

Table 13: Library description and file dependency.

Program files and structure	Description	Dependencies
0. Examples & Benchmarking programs		
0.a: matConv.c	Has examples on how to run the different types of SMVM and CG operations. Also contains a conversion function that reads from a Matrix Market file and generates the CSR, BCSR, PMS, and BPMS sparse representations.	1.a, 1.c, 2.a, 2.b, 3.a, 3.b, 4.b, 4.f
1. General libraries		
1.a: util.(c,h)	Provides support to create, copy, destroy and identify basic *C* data types through a single interface.	---
1.b: my_cell_lib.(c,h)	Contains complex data structures (general sparse matrix, vector), SIMD vector sizes, aligned-portable memory allocation and free functions, support data structure and functions for partitioning.	1.a
1.c: pthread_util.(c,h)	Functions and data structures to control (start, stop and join) a pool of threads	---
2. Sparse matrix libraries		
2.a: ppe_matrix_lib.(c,h)	Support data structures and functions for various matrix operations. Includes read/write matrix/vector files, obtain errors reports, create 2-level partitions, create/delete sparse matrix memory space, initialize matrix/vector data to predefined values, conversion between matrix formats, printing matrix/vectors to screen or files, utility functions related to matrix operations.	1.b, 1.c
2.b: sparse_BLAS.h	Function interfaces (prototypes) for all BLAS libraries.	---
2.c: sparse_BLAS1.c	Limited implementation of BLAS 1 functions (saxpy, daxpy, dot-product, 2-norm, others) – Non optimized	2.a
2.d: sparse_BLAS2.c	Limited implementation of BLAS 2 functions (SMVM for each format, threaded and non-threaded) – Optimized and Non-optimized (requires Intel ICC headers for the Intel implementations).	1.b, 1.c, 2.a
2.e: sparse_BLAS3.c	Limited implementation of BLAS 3 functions (multiply and transpose both precisions) – Non optimized	1.b, 2.a
3. CG libraries		
3.a: CG.(c,h)	CG (for CSR both precisions, BCSR, PMS and BPMS with single precision), various PCG implementations (CSR and PMS for single and double precisions), create/solve for Jacobi and polynomial preconditioners (single precision), and solver for incomplete Cholesky preconditioner (single precision).	1.b, 1.c, 2.a, 2.b
3.b: decomp.(c,h)	Create incomplete Cholesky factorization for single and double precision.	1.b, 2.a
4. FEM libraries		
4.a: FEM_main_test.c	Implementations of classic and proposed approaches for the FEM. For the proposed approach sequential and parallel version are provided including the GPU version for single precision. In each case a FEM mesh is read and the matrix system is built (using the FEM_preprocessor), then the appropiate solver is called (in the FEM_solver file), and finally the absolute error of the solution and the system energy are computed using the FEM_postprocessor.	1.b, 2.a, 2.b, 3.a, 4.b, 4.c, 4.d, 4.e
4.b: FEM_util.(c,h)	Data structures and support functions to read from file mesh information and create/destroy basic constructs (nodes, mesh elements, BC, etc) related to FEM. Also provides special functions that support multicore parallelism using a pool of threads.	1.a, 1.b, 1.c
4.c: FEM_preprocessor.(c,h)	Functions to read FEM mesh information from file and to assemble the matrix system. Also provides functions to create the decoupled system used in FEM-SES.	2.a, 2.b, 4.b
4.d: FEM_solver.(c,h)	Function for sequential and parallel solution of the FEM system using the FEM-SES method.	2.a, 2.b, 4.b
4.e: FEM_postprocessor.(c,h)	Functions to compute the energy of the system.	2.a, 4.b
GPU Libraries (CUDA code compiled with NVCC)		
4.f: FEM_cuda_kernel.h	Function interfaces (prototypes) to be used by the host to call functions that launch GPU code for the new FEM-SES method.	---
4.g: FEM_GPU_fixed_elements_solver.cu	Function definitions for the FEM-SES GPU kernels.	1.b, 2.a, 2.b, 4.c, 4.d, 4.h, and CUBLAS
4.h: FEM_solver_kernel_V1.cu	GPU kernel implementations for the FEM-SES method.	1.b, 4.b

Bibliography

[1] J. Yuanwei, J. Yi, and J. M. F. Moura, “Time Reversal Beamforming for Microwave Breast Cancer Detection,” in *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, pp. V - 13-V - 16, 2007.

[2] H. Berg, B. Gunther, I. Hilger *et al.*, “Bioelectromagnetic field effects on cancer cells and mice tumors,” *Electromagn Biol Med,* vol. 29, no. 4, pp. 132-43, Dec, 2010.

[3] R. Mittet, “High-order finite-difference simulations of marine CSEM surveys using a correspondence principle for wave and diffusion fields,” *Geophysics,* vol. 75, no. 1, pp. F33-F50, Jan-Feb, 2010.

[4] G. K. Konstadinidis, “Challenges in microprocessor physical and power management design,” in *VLSI Design, Automation and Test, 2009. VLSI-DAT '09. International Symposium on*, pp. 9-12, 2009.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro *et al.*, *The Landscape of Parallel Computing Research: A View from Berkeley,* UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18, 2006.

[6] D. A. Patterson, "The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges," Parallel Computing Research Laboratory (Par Lab), U.C. Berkeley, 2009, http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf, [January, 2010].

[7] N. Goodnight, C. Woolley, G. Lewin *et al.*, “A multigrid solver for boundary value problems using programmable graphics hardware,” in *HWWS'03. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, (San Diego, California), pp. 102-111, 2003.

[8] *CUDA Programming Guide for CUDA Toolkit 3.2*: NVIDIA Corporation, 2010, http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf, [April, 2011].

[9] "ATI Stream Software Development Kit," [January, 2011]. http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx.

[10] "AMD Accelerated Parallel Processing (APP) SDK (formerly ATI Stream)," [January, 2011]. http://developer.amd.com/gpu/AMDAPPSDK/Pages/default.aspx.

[11] "OpenCL," [January, 2011]. http://www.khronos.org/opencl.

[12] *Cell Broadband Engine Programming Handbook*, version 1.1, New York: IBM, 2007, http://www.ibm.com/developerworks/power/cell/documents.html, [April, 2008].

[13] N. Brookwood, "AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience," 2010, http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf, [March, 2011].

[14] G. Goumas, K. Kourtis, N. Anastopoulos *et al.*, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *Journal of Supercomputing,* vol. 50, no. 1, pp. 36-77, Oct, 2009.

[15] E. J. Im, and K. A. Yelick, "Optimizing sparse matrix vector multiplication on SMPs," in *Proceedings of the SIAM Conference on Parallel Processing for Scientic Computing*, (San Antonio, TX, USA), 1999.

[16] R. W. Vuduc, "Automatic Performance Tuning of Sparse Matrix Kernels," PhD Thesis, University of California, Berkeley, 2003.

[17] E. J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *International Journal of High Performance Computing Applications,* vol. 18, no. 1, pp. 135-158, Spr, 2004.

[18] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *SciDAC 2005. Scientific Discovery Through Advanced Computing*, (San Francisco, CA), pp. 521-530, 2005.

[19] S. Toledo, “Improving the memory-system performance of sparse-matrix vector multiplication,” *IBM Journal of Research and Development,* vol. 41, no. 6, pp. 711-725, Nov, 1997.

[20] A. Pinar, and M. T. Heath, “Improving performance of sparse matrix-vector multiplication,” in *Supercomputing'99. Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, (Portland, Oregon, United States), pp. 30, 1999.

[21] R. Nishtala, R. W. Vuduc, J. W. Demmel *et al.*, “When cache blocking of sparse matrix vector multiply works and why,” *Applicable Algebra in Engineering Communication and Computing,* vol. 18, no. 3, pp. 297-311, May, 2007.

[22] S. W. Williams, J. Shalf, L. Oliker *et al.*, “The potential of the cell processor for scientific computing,” in *Proceedings of the 3rd conference on Computing frontiers*, (Ischia, Italy), pp. 9-20, 2006.

[23] S. Williams, L. Oliker, R. Vuduc *et al.*, “Optimization of sparse matrix-vector multiplication on emerging multicore platforms,” in *SC '07. Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, (Reno, Nevada), pp. 38:1--38:12, 2007.

[24] J. J. Dongarra, I. S. Duff, D. C. Sorensen *et al.*, *Solving Linear Systems on Vector and Shared Memory Computers*, Philadelphia, PA, USA: Society for Industrial and Applied Mathematics (SIAM), 1991.

[25] D. R. Kincaid, T. C. Oppe, and D. M. Young, *Adapting ITPACK routines for use on a vector computer,* CNA-177, Center for Numerical Analysis, Univ. Texas, Austin, TX, 1982.

[26] Y. Saad, “Krylov subspace methods on supercomputers,” *SIAM J. Sci. Stat. Comput.,* vol. 10, no. 6, pp. 1200-1232, November, 1989.

[27] T. C. Oppe, and D. R. Kincaid, “The performance of ITPACK on vector computers for solving large sparse linear systems arising in sample oil reseervoir simulation problems,” *Communications in Applied Numerical Methods,* vol. 3, no. 1, pp. 23-29, 1987.

[28] Y. Saad, *SPARSKIT : A basic tool kit for sparse matrix computations,* RIACS-90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990.

[29] N. K. Madsen, G. H. Rodrigue, and J. I. Karush, “Matrix Multiplication by Diagonals on a Vector/Parallel Processor,” *Information Processing Letters,* vol. 5, no. 2, pp. 41-45, June, 1976.

[30] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., Philadelphia, PA, USA: Society for Industrial and Applied Mathematics (SIAM), 2003.

[31] G. E. Blelloch, *Vector Models for Data-Parallel Computing*, Cambridge, Mass., USA: MIT Press, 1990.

[32] R. Barrett, M. Berry, T. F. Chan *et al.*, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, Philadelphia, PA: SIAM, 1994.

[33] J. W. Demmel, M. T. Heath, and H. A. van der Vorst, *Parallel Numerical Linear Algebra,* CSD-92-703, UC Berkeley, EECS technical reports, October 6, 1992.

[34] W. A. Wiggers, V. Bakker, A. B. J. Kokkeler *et al.*, “Implementing the conjugate gradient algorithm on multi-core systems,” *2007 International Symposium on System-on-Chip Proceedings*, pp. 11-14, 2007.

[35] G. A. Gravvanis, P. I. Matskanidis, K. M. Giannoutakis *et al.*, “Finite element approximate inverse preconditioning using POSIX threads on multicore systems,” in *IMCSIT'10. Proceedings of the International Multiconference on Computer Science and Information Technology*, pp. 297-302, 2010.

[36] I. Lee, “Analyzing Multithreaded Preconditioned Conjugate Gradient (mtPCG) Algorithm on Multicore Architecture,” in *Proceedings of the Ieee Southeastcon 2009, Technical Proceedings*, pp. 317-322, 2009.

[37] H. M. Markowitz, “The Elimination Form of the Inverse and Its Application to Linear-Programming,” *Management Science,* vol. 3, no. 3, pp. 255-269, April, 1957.

[38] J. R. Yost. "Harry M. Markowitz, OH 333. Oral history interview by Jeffrey R. Yost," Charles Babbage Institute, University of Minnesota, Minneapolis, [Dicember, 2010]. http://www.cbi.umn.edu/oh/display.phtml?id=322.

[39] "LAPACK—Linear Algebra PACKage, Version 3.3.0 ", [November 14, 2010]. http://www.netlib.org/lapack/index.html.

[40] I. S. Duff, “Survey of Sparse-Matrix Research,” *Proceedings of the IEEE,* vol. 65, no. 4, pp. 500-535, April, 1977.

[41] P. T. Stathis, “Sparse Matrix Vector Processing Formats,” PhD Thesis, Delft University of Technology, 2004.

[42] R. Boisvert, R. Pozo, K. Remington *et al.* "Matrix market," National Institute of Standards and Technology (NIST), Gaithersburg (Maryland), [January, 2011]. http://math.nist.gov/MatrixMarket/.

[43] S. Balay, J. Brown, K. Buschelman *et al.*, *PETSc Users Manual,* ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010.

[44] Y. Saad. "ITSOL," University of Minnesota, [January, 2011]. http://www-users.cs.umn.edu/~saad/software/ITSOL/index.html.

[45] T. A. Davis, and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software (to appear)*, http://www.cise.ufl.edu/research/sparse/matrices, [January, 2009].

[46] "Symposium on Sparse Matrices and Their Applications," IBM T.J. Watson Research Center, September 9-10, 1968.

[47] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*, Oxford Oxfordshire: Oxford University Press Inc., 1986.

[48] T. A. Davis, *Direct methods for sparse linear systems*, Philadelphia: Society for Industrial and Applied Mathematics, 2006.

[49] R. S. Varga, *Matrix iterative analysis*, 2nd rev. and expanded ed., Berlin ; New York: Springer Verlag, 2000.

[50] L. A. Hageman, and D. M. Young, *Applied iterative methods*, New York: Academic Press, 1981.

[51] D. M. Young, “Convergence Properties of the Symmetric and Unsymmetric Over-Relaxation Methods,” *Math. Comp.,* vol. 24, pp. 793-807, 1970.

[52] G. H. Golub, and C. F. Van Loan, *Matrix computations*, 3rd ed., Baltimore: Johns Hopkins University Press, 1996.

[53] R. Pozo, K. Remington, and A. Lumsdaine, *SparseLib++ Sparse Matrix Class Library, User's Guide*: National Institute of Standards and Technology, 1996, http://math.nist.gov/sparselib++/sparselib-userguide.pdf,

[54] S. Filippone, and M. Colajanni, “PSBLAS: a library for parallel linear algebra computation on sparse matrices,” *ACM Trans. Math. Softw.,* vol. 26, no. 4, pp. 527-550, December, 2000.

[55] Z. Li, M. S. Masha, D. O. Kuffuor *et al.* "pARMS," University of Minnesota, [January, 2011]. http://www-users.cs.umn.edu/~saad/software/pARMS/.

[56] S. Y. Kung, *VLSI array processors*, Englewood Cliffs, N.J.: Prentice Hall, 1988.

[57] W. P. Petersen, and P. Arbenz, *Introduction to parallel computing A Practical Guide with Examples in C*, Oxford: Oxford University Press, 2004.

[58] M. R. Hestenes, and E. Stiefel, “Methods of Conjugate Gradients for Solving Linear Systems,” *Journal of Research of the National Bureau of Standards,* vol. 49, no. 6, pp. 409-436, December, 1952.

[59] J. R. Shewchuk, *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[60] A. T. Chronopoulos, and C. W. Gear, “S-Step Iterative Methods for Symmetric Linear-Systems,” *Journal of Computational and Applied Mathematics,* vol. 25, no. 2, pp. 153-168, Feb, 1989.

[61] R. C. Agarwal, F. G. Gustavson, and M. Zubair, “A high performance algorithm using pre-processing for the sparse matrix-vector multiplication,” in *Supercomputing '92, Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, (Minneapolis, Minnesota, United States), pp. 32-41, 1992.

[62] J. J. Dongarra, F. G. Gustavson, and A. Karp, “Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine,” *SIAM Review,* vol. 26, no. 1, pp. 91-112, January, 1984.

[63] V. Eijkhout, *LAPACK Working Note 50: Distributed Sparse Data Structures for Linear Algebra Operations*, University of Tennessee, Knoxville, TN, USA, 1992.

[64] P. Fernandes, and P. Girdinio, “A New Storage Scheme for an Efficient Implementation of the Sparse Matrix-Vector Product,” *Parallel Computing,* vol. 12, no. 3, pp. 327-333, Dec, 1989.

[65] *Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 1: Basic Architecture, Order Number: 253665-037US*, 2011, [January, 2011].

[66] *Intel® 64 and IA-32 Architectures Optimization Reference Manual Number: 248966-023a*: Intel Corporation, 2011, http://www.intel.com/products/processor/manuals/index.htm, [January, 2011].

[67] "Compiler Intrinsics-Visual Studio 2010," [January, 2011]. http://msdn.microsoft.com/en-us/library/26td21ds.aspx?ppud=4.

[68] *Intrinsics Reference” in Intel® C++ Compiler for Linux* - 9.x manuals*: Intel Corporation, 2008, http://software.intel.com/en-us/articles/intel-c-compiler-for-linux-9x-manuals/, [December, 2008].

[69] M. J. Wolfe, *High performance compilers for parallel computing*, Redwood City, Calif.: Addison-Wesley, 1996.

[70] *OpenMP Application Program Interface*, version 3.0, 2008, http://www.openmp.org/mp-documents/spec30.pdf,

[71] D. Buttlar, J. Farrell, and B. Nichols, *PThreads Programming A POSIX Standard for Better Multiprocessing*: O'Reilly & Associates, 1996.

[72] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*, first ed.: O'Reilly Media, 2007.

[73] *Using the x86 Open64 Compiler Suite*: Advanced Micro Devices Inc., 2010, http://developer.amd.com/cpu/open64/onlinehelp/pages/x86_open64_help.htm, [January, 2011].

[74] J. L. Hennessy, and D. A. Patterson, *Computer Architecture A Quantative Approach*, fourth ed., San Francisco, USA: Morgan Kaufmann, 2007.

[75] W. P. Cockshott, and K. Renfrew, *SIMD programming manual for Linux and Windows*, London New York: Springer, 2004.

[76] U. V. Catalyurek, and C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” *Ieee Transactions on Parallel and Distributed Systems,* vol. 10, no. 7, pp. 673-693, Jul, 1999.

[77] K. D. Devine, E. G. Boman, R. T. Heaphy *et al.*, “Parallel hypergraph partitioning for scientific computing,” in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10 pp., 2006.

[78] B. Hendrickson, R. Leland, and S. Plimpton, “An Efficient Parallel Algorithm for Matrix-Vector Multiplication,” *International Journal of High Speed Computing,* vol. 7, no. 1, pp. 73-88, Mar, 1995.

[79] B. Hendrickson, Tamara, and G. Kolda, “Partitioning Rectangular And Structurally Nonsymmetric Sparse Matrices For Parallel Processing,” *SIAM J. Sci. Comput,* vol. 21, no. 6, pp. 2048-2072, 1998.

[80] D. M. Fernández, D. Giannacopoulos, and W. J. Gross, “Efficient Multicore Sparse Matrix-Vector Multiplication for Finite-Element Electromagnetics,” in *CEFC'08. Proceedings of the 13th Biennial IEEE Conference of Electromagnetic Field Computation*, (Athens, Greece), pp. 469, 2008.

[81] D. M. Fernández, D. Giannacopoulos, and W. J. Gross, “Efficient Multicore Sparse Matrix-Vector Multiplication for FE Electromagnetics,” *Magnetics, IEEE Transactions on,* vol. 45, no. 3, pp. 1392-1395, 2009.

[82] J.-M. Jin, *The finite element method in electromagnetics*, 2nd ed., New York: John Wiley & Sons, 2002.

[83] P. P. Silvester, and R. L. Ferrari, *Finite elements for electrical engineers*, 3rd ed., New York: Cambridge University Press, 1996.

[84] E. Cuthill, and J. McKee, “Reducing the bandwidth of sparse symmetric matrices,” in *ACM'69. Proceedings of the 1969 24th national conference*, pp. 157-172, 1969.

[85] "MATLAB - Sparse Matrices Demo," MathWorks, [January, 2011]. http://www.mathworks.com/products/matlab/demos.html?file=/products/demos/shipping/matlab/sparsity.html.

[86] D. M. Fernández, D. Giannacopoulos, and W. J. Gross, “Multicore Acceleration of CG Algorithms Using Blocked-Pipeline-Matching Techniques,” in *Proceedings of the Seventeenth Conference on the Computation of Electromagnetic Fields*, (Florianópolis, Brazil), pp. 827-828, 2009.

[87] D. M. Fernández, D. Giannacopoulos, and W. J. Gross, “Multicore Acceleration of CG Algorithms Using Blocked-Pipeline-Matching Techniques,” *IEEE Transactions on Magnetics,* vol. 46, no. 8, pp. 3057-3060, Aug, 2010.

[88] M. M. Dehnavi, D. M. Fernández, and D. Giannacopoulos, “Finite-Element Sparse Matrix Vector Multiplication on Graphic Processing Units,” *Magnetics, IEEE Transactions on,* vol. 46, no. 8, pp. 2982-2985, August, 2010.

[89] M. Mehri Dehnavi, D. M. Fernández, and D. Giannacopoulos, “Enhancing the performance of conjugate gradient solvers on graphic processing units,” *To appear in IEEE Transactions on Magnetics*, 2011.

[90] A. Toselli, and O. B. Widlund, *Domain decomposition methods--algorithms and theory*, Berlin: Springer, 2005.

[91] Y. Q. Liu, and J. S. Yuan, “A finite element domain decomposition combined with algebraic multigrid method for large-scale electromagnetic field computation,” *IEEE Transactions on Magnetics,* vol. 42, no. 4, pp. 655-658, Apr, 2006.

[92] T. Itoh, G. Pelosi, and P. P. Silvester, *Finite element software for microwave engineering*, New York: Wiley, 1996.

[93] A. Takei, S. I. Sugimoto, M. Ogino *et al.*, “Full Wave Analyses of Electromagnetic Fields With an Iterative Domain Decomposition Method,” *IEEE Transactions on Magnetics,* vol. 46, no. 8, pp. 2860-2863, Aug, 2010.

[94] "NVIDIA’s Next Generation CUDA Compute Architecture: Fermi," *Whitepaper*, NVIDIA Corporation, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf, [January, 2011].

[95] *CUDA CUBLAS Library*, version PG-05326-032_V02: NVIDIA Corporation, 2010, http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf, [January, 2011].

[96] J. P. A. Bastos, and N. Sadowski, “A New Method to Solve 3-D Magnetodynamic Problems Without Assembling an Ax=b System,” *Magnetics, IEEE Transactions on,* vol. 46, no. 8, pp. 3365-3368, 2010.

[97] J. Eyng, J. P. A. Bastos, N. Sadowski *et al.*, “Parallel programming applied to the N Scheme for solving FE cases without assembling an A x = b system,” in *Electromagnetic Field Computation (CEFC'10), 2010 14th Biennial IEEE Conference on*, pp. 1-1, 2010.

[98] D. Göddeke, R. Strzodka, and S. Turek, “Accelerating Double Precision FEM Simulations with GPUs,” in *ASIM'05. 18th Symposium Simulations technique, Frontiers in Simulation*, pp. 139–144, 2005.

[99] J. Filipovic, I. Peterlik, and J. Fousek, “GPU Acceleration of Equations Assembly in Finite Elements Method - Preliminary Results,” in *SAAHPC’09. Symposium on Application Accelerators in HPC*, 2009.

[100] C. Cristopher, A. Lew, and E. Darve, “Introduction to assembly of finite element methods on graphics processors,” *IOP Conference Series: Materials Science and Engineering,* vol. 10, no. 1, pp. 012009, 2010.

[101] D. Goddeke, R. Strzodka, J. Mohd-Yusof *et al.*, “Exploring weak scalability for FEM calculations on a GPU-enhanced cluster,” *Parallel Computing,* vol. 33, no. 10-11, pp. 685-699, Nov, 2007.

[102] E. Hermann, B. Raffin, F. Faure *et al.*, “Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations,” *Euro-Par 2010 - Parallel Processing, Part II,* vol. 6272, pp. 235-246, 2010.

[103] N. Godel, N. Nunn, T. Warburton *et al.*, “Scalability of Higher-Order Discontinuous Galerkin FEM Computations for Solving Electromagnetic Wave Propagation Problems on GPU Clusters,” *IEEE Transactions on Magnetics,* vol. 46, no. 8, pp. 3469-3472, Aug, 2010.

[104] D. M. Fernández, D. Giannacopoulos, and W. J. Gross. (2010). Alternate approach to FEM for parallel processing. *The 10th International Workshop on Finite Elements for Microwave Engineering*, p. 52, Meredith, New Hampshire, USA.

[105] S. H. Fuller, and L. I. Millett, “Computing Performance: Game Over or Next Level?,” *Computer,* vol. 44, no. 1, pp. 31-38, January, 2011.