

To the Graduate Council:

I am submitting herewith a thesis written by Rajib Kumar Nath entitled “Accelerating Dense Linear Algebra for GPUs, Multicores and Hybrid Architectures: an Autotuned and Algorithmic Approach.” I have examined the final paper copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this thesis
and recommend its acceptance:

Stanimire Z. Tomov

Lynne E. Parker

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

To the Graduate Council:

I am submitting herewith a thesis written by Rajib Kumar Nath entitled “Accelerating Dense Linear Algebra for GPUs, Multicores and Hybrid Architectures: an Autotuned and Algorithmic Approach.” I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this thesis
and recommend its acceptance:

Stanimire Z. Tomov

Lynne E. Parker

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Accelerating Dense Linear Algebra for GPUs, Multicores and Hybrid Architectures: an Autotuned and Algorithmic Approach

A Thesis Presented for
The Master of Science
Degree

The University of Tennessee, Knoxville

Rajib Kumar Nath

August 2010

© by Rajib Kumar Nath, 2010
All Rights Reserved.

This dissertation is dedicated to my father, Surjo Nath, and to my mother Nilima Das, who has supported and encouraged me to pursue education throughout my whole life.

Acknowledgements

I would like to thank my supervisor Stanimire Tomov and my adviser Jack Dongarra for their guidance for last 2 years. I also like to thank all the members at ICL with whom I have got the opportunity to work with. I would like to mention the name of Jakub Kurzak, Dan Terpstra, and Emmanuel Agullo for their guidance during my work period at Innovative Computing Laboratory in University of Tennessee, Knoxville.

If you want to do it just go for it.

Abstract

Dense linear algebra(DLA) is one of the most important softwares in high performance computing. It is also important for it's wide usage in other application domains like machine learning, gaming, speech processing, image processing, etc. The introduction of new machines from vendor provides us opportunities to optimize DLA libraries for the new machines and thus exploit their power. Unfortunately the optimization phase is not straightforward all the time. The most important part of DLA libraries are it's basic linear algebra subprograms(BLAS) kernels. The optimum code of a certain BLAS kernel in two different machines with different semiconductor process can be different even if they share the same features in terms of instruction set architecture, memory hierarchy and clock speed. It has become an tradition to optimize BLAS for upcoming machines. Vendors like Intel, AMD and IBM maintain highly optimized BLAS libraries targeting their own CPUs. In the GPU sector, NVIDIA is also providing CUBLAS for it's accelerator cards like GTX280, Tesla2050. There has been few research in academia to optimize BLAS for GPUs. But the area is still new and presents numerous cases/opportunities for improvements. The existing BLAS for GPUs are not highly optimized for DLA algorithms. For example, vendors don't have highly optimized BLAS for rectangular shaped problem size. Level 2 BLAS e.g. symmetric matrix matrix multiplication, which are very important for memory bound operations like tridiagonalization, performs poorly. In certain GPUs like GTX280 BLAS kernels have performance dips due to partition camping phenomenon in global memory modules. More importantly the existing BLASs are not optimized for generic

problem size. In my research I have provided new algorithms for several important BLAS kernels for different generation of GPUs and introduced a pointer redirecting approach to make BLAS run faster in generic problem size. I have also presented an auto-tuning approach to parameterize the developed BLAS algorithms and select the best set of parameters for a given card.

The hardware trends have also brought up the need for updates on existing legacy DLA software packages, such as the sequential LAPACK. To take advantage of the new computational environment, successors of LAPACK must incorporate algorithms of three main characteristics: high parallelism, reduced communication, and heterogeneity-awareness. In all cases though, the development can be streamlined if the new algorithms are designed at a high level, using just a few, highly optimized low level kernels. In the dense linear algebra community, several projects have addressed this challenge on different hardware architectures. On multicore architectures, Parallel Linear Algebra Software for Multicore Architectures (PLASMA) has been developed to meet the challenges in multicore. On the other extreme, Matrix Algebra on GPU and Multicore Architectures (MAGMA) library demonstrated a hybridization approach that indeed streamlined the development of high performance DLA for multicores with GPU accelerators. The performance of these two libraries depend upon right choice of parameters for a given problem size and given number of cores and/or GPUs. In this work, the issue of automatically tuning these two libraries is presented. For a matter of conciseness, the focus is on one particular operation, the QR factorization, which is representative of all three one-sided factorizations (QR, LU, Cholesky) currently available in PLASMA. A prune based auto-tuning method has been proposed for tuning PLASMA. Part of the tuning method for PLASMA was considered to tune hybrid MAGMA library.

Contents

List of Tables	ix
List of Figures	xi
1 Introduction	1
2 BLAS Kernels Developement for GPUs: Algorithmic Perspective	6
2.1 Level 1 BLAS	9
2.2 Level 2 BLAS	10
2.2.1 xGEMV	11
2.2.2 xSYMV	14
2.3 Level 3 BLAS	19
2.3.1 xGEMM	19
2.3.2 xSYRK	23
2.3.3 xSYR2K	25
2.3.4 xTRSM	26
3 Generic BLAS Kernels Developement for GPUs: Pointer Redirect-	
ing	28
3.1 Pointer Redirecting	29
3.2 Performance	33
4 Autotuning BLAS Kernels for GPUs: MAGMABLAS	37

4.1	Auto-tuning GEMM	38
4.2	Performance results	41
5	Tuning Dense Linear Algebra for Multicore Architecture: PLASMA	47
5.1	Tunable parameters	48
5.2	Motivation for an empirical approach	51
5.3	Outline of the method	54
5.4	Experimental environments	55
5.5	Step 1: Benchmarking the most compute-intensive serial kernels	58
5.6	Step 2: Benchmarking at-scale executions	63
5.6.1	Discretization	63
5.6.2	Impact of the heuristics on the time required for tuning	63
5.6.3	Prune As You Go (PSPAYG)	64
5.6.4	Accuracy of the tuning	64
6	Tuning Dense Linear Algebra for Hybrid Architecture: MAGMA	72
7	Conclusion	77
	Bibliography	80
	Vita	87

List of Tables

2.1	Key parameters of a sample of GPU GEMM kernels.	21
3.1	Performance comparison between MAGMA BLAS with pointer redirecting and CUBLAS for the QR factorization in single precision arithmetic	36
4.1	Different kernel configurations.	41
5.1	Elapsed time (hh:mm:ss) for Step 1 and Step 2	57
5.2	Average performance achieved with a “pre-selection” (PS) method or a “pre-selection and prune as you go” (PSPAYG) method, based on different heuristics (H) applied at step 1. The performance is presented as a proportion of the exhaustive search (ES) or of the prunes search (PS). The column “optimum” indicates the number of times the optimum combination (with respect to the reference method) was found among the number of tests performed.	69
5.3	Performance of ES on the AMD Istanbul Machine	70
5.4	Performance of Heuristic 0 on the AMD Istanbul machine.	70
5.5	Performance of Heuristic 1 on the AMD Istanbul machine	71
5.6	Performance of Heuristic 2 on the AMD Istanbul machine	71
6.1	Performance of MAGMA’s LU Factorization on GTX 280 for different panel size	74

6.2	Performance of MAGMA's LU Factorization on TESLA for different panel size	75
-----	--	----

List of Figures

2.1	Algorithmic view of Level 1 and Level 2 BLAS.	10
2.2	Performance of xGEMV (non-transpose) on a GTX 280.	12
2.3	Two memory access implementations of xGEMV (transpose).	12
2.4	Performance of xGEMV (transpose) on a GTX 280.	13
2.5	Three cases of TB computations in xSYMV.	14
2.6	Performance of xSYMV on a GTX 280.	14
2.7	Data access pattern in new xSYMV algorithm.	16
2.8	Results produced by each thread block in new xSYMV algorithm. . .	16
2.9	Recursive blocking in new xSYMV algorithm.	16
2.10	xSYMV in single precision with new algorithm on GTX280, RB^+ means recursive blocking was used	18
2.11	The GPU GEMM ($C = AB$) of a single TB.	20
2.12	Performance of GEMM ($C = \alpha AB^T + \beta C$) on a GTX 280.	21
2.13	The GPU GEMM ($C = AB$) of a single TB in Fermi.	23
2.14	Performance of dGEMM on a Fermi.	24
2.15	Performance of dGEMM on a Fermi.	24
2.16	Performance of xSYRK on a GTX 280.	25
2.17	Performance of SSYR2K on GTX280	26
2.18	Performance of xTRSM on a GTX 280.	27
3.1	GEMM Performance on Square Matrices.	29
3.2	The algorithmic view of GEMM for GPUs.	30

3.3	GEMM Implementation with Conditional Statement in Inner Loop. . .	31
3.4	Possible Illegal Memory Reference in Matrix Multiply.	31
3.5	(Left) Last Valid Access (Middle) Pointer Redirecting (Right) Mirroring	32
3.6	Algorithmic view of GEMM for GPUs with Pointer Redirecting. . . .	33
3.7	Flops overhead in xGEMM	34
3.8	Performance dGEMM	34
3.9	Performance sGEMM	35
3.10	Performance xGEMM with Padding (Data In/Out in CPU Memory). .	35
4.1	Performance of auto-tuned DGEMM kernel ($Op(A) = A^T$, $Op(B) = B$) on a GTX 280.	41
4.2	Performance of the auto-tuned SGEMM ($Op(A) = A$, $Op(B) = B^T$) kernel for square matrices on a GTX 280.	42
4.3	Performance comparison of the auto-tuned (solid line) <i>vs</i> CUBLAS 2.3 DGEMMs occurring in the block LU factorization (for block sizes $BS = 64$ on the left and 128 on the right) of a matrix of size 6144×6144 . The two kernels shown are for multiplying $N \times BS$ and $BS \times N - BS$ matrices (denoted by $N \times N - BS \times BS$), and $N \times BS$ and $BS \times BS$ matrices (denoted by $N \times BS \times BS$). K6 was used when $BS = 64$ and K7 was used when $BS = 128$	44
4.4	Solvers in GPU NVIDIA GTX 280.	45
4.5	Two-sided factorization in single precision on GPU NVIDIA GTX 280.	46
5.1	Panel factorization and corresponding updates.	48
5.2	DAG of the tile QR factorization. The matrix is split in 5×5 tiles. .	49
5.3	Performance of the sequential PLASMA QR factorization on an Intel Core Tigerton machine.	50
5.4	Performance of the PLASMA QR factorization on an Intel Core Tigerton machine using 16 cores.	50

5.5	Performance of the PLASMA QR factorization on an IBM Power6 machine using 32 cores.	51
5.6	Performance (in Gflop/s) of a sequential matrix multiplication $c \leftarrow c + a \times b$ on the Intel Core Tigerton machine as a standard call to the vendor BLAS library. With the <i>No Flush</i> strategy, data (a , b and c) is <i>not</i> flushed from the cache. With the <i>MultCallFlushLRU</i> strategy (29), a and b (but not c) are flushed from the cache. The values corresponding to a matrix order $NB = 60$ are circled.	52
5.7	Performance (in Gflop/s) of the tile matrix multiplication on the Intel Core Tigerton machine using 1 core. The tile size is $NB = 60$	53
5.8	Step 1-a: Performance of the DSSRFB serial kernel depending on the (NB-IB) parameters. Note that two (NB-IB) pairs with a common NB value have the same abscisse.	59
5.9	Step 1-b: Picking up the optimum IB for each NB.	60
5.10	Performance of the pre-selected search (PS) against the exhaustive search (ES) on the Intel Core Tigerton machine. The graphs are almost superimposed.	61
5.11	Step 1-c: Extracting the convex hull (Heuristic 0)	61
5.12	Step 2 - Heuristic 1: maximum steepness	62
5.13	Step 2 - Heuristic 2: even distribution	62
5.14	Intel Core Tigerton machine - $N = 6000$	67
5.15	Intel Core Tigerton machine - $N = 2000$	67
5.16	Intel Core Tigerton machine - $N = 1000$	68
5.17	IBM Power6 machine - $N = 2000$	68
6.1	Algorithms as collection of BLAS-based tasks and dependencies among them (DAGs) for hybrid GPU-based computing	73
6.2	MAGMA's LU performance for different panel size.	76

Chapter 1

Introduction

Recent activities of major chip manufacturers, such as Intel, AMD, IBM and NVIDIA, make it more evident than ever that future designs of microprocessors and large HPC systems will be hybrid/heterogeneous in nature, relying on the integration (in varying proportions) of two major types of components:

1. Multi/many-cores CPU technology, where the number of cores will continue to escalate while avoiding the power wall, instruction level parallelism wall, and the memory wall (13); and
2. Special purpose hardware and accelerators, especially GPUs, which are in commodity production, have outpaced standard CPUs in performance, and have become as easy, if not easier to program than multicore CPUs.

The relative balance between these component types in future designs is not clear, and will likely vary over time, but there seems to be no doubt that future generations of computer systems, ranging from laptops to supercomputers, will consist of a composition of heterogeneous components.

These hardware trends have inevitably brought up the need for updates on existing legacy software packages, such as the sequential LAPACK (14), from the area of dense linear algebra (DLA). To take advantage of the new computational environment,

successors of LAPACK must incorporate algorithms of three main characteristics: high parallelism, reduced communication, and heterogeneity-awareness. In all cases though, the development can be streamlined if the new algorithms are designed at a high level, using just a few, highly optimized low level kernels. In the dense linear algebra community, several projects have addressed this challenge on different hardware architectures. On graphic processing units (GPUs), among others, (26) and (10) have proposed efficient approaches. On multicore architectures, Parallel Linear Algebra Software for Multicore Architectures (PLASMA) (27; 36) has been developed.

PLASMA is a redesign of LAPACK (14) and ScaLAPACK (37) for shared memory systems based on multi-core processor architectures. All of the traditional multicore vendors maintain efficient BLAS library for their machines, e.g. MKL (4) from Intel, ESL (6) from IBM, ACML (5) from AMD. So PLASMA does not need to worry too much about efficient BLAS kernels. To achieve high performance on this type of architecture, PLASMA relies on tile algorithms and high performance BLAS directly provided by the vendors. PLASMA aims at providing a fine granularity and high asynchronicity to fit multicore constraints. One of the vital requirement of PLASMA's approach is that it needs intensive tuning to fully benefit from the potential of the hardware.

On the other extreme, MAGMA library (10) demonstrated a hybridization approach that indeed streamlined the development of high performance DLA for multicores with GPU accelerators. The new algorithms, covering core DLA routines, are now part of the MAGMA library (10), a successor to LAPACK for the new heterogeneous/hybrid architectures. Similarly to LAPACK, MAGMA relies on the efficient implementation of a set of low level linear algebra kernels. In the context of GPU-based hybrid computing, a subset of BLAS (15) for GPUs is needed. Although there have been several recent success in developing highly optimized BLAS for GPUs (2; 26; 11), the area is still new and presents numerous cases/opportunities for improvements. The GPU BLAS provided by the vendors e.g. CUBLAS from

NVIDIA isn't highly optimized for all the BLASs that are needed for DLA. Even if some of the required BLASs are optimized e.g. Matrix-Matrix multiplication, it is optimized for few problem sizes (sizes divisible by 64 on GTX 280). Many blas routines have performance oscillations because of the constraint from implementation (inner block or algorithm dependent parameter in kernel) and GPU global memory layout. This work addresses an algorithmic approach to optimize BLAS routines that are needed for DLA. In some of the cases existing algorithms (e.g. Matrix-Matrix multiplication (26)) are revisited. In some of the cases new algorithms are developed (e.g. symmetric matrix-vector multiplication) to enhance the performance. This work also addresses the issues of problem size constraint in data parallel architecture like GPUs and presents the *Pointer Redirecting* approach as a feasible solution as opposed to *Padding*. The complex architecture of GPUs introduces many tunable parameters in the BLAS algorithms.

Tuning consists of finding the parameters that maximize a certain metric (most of the time the performance) on a given environment. In general, the term *parameter* has to be considered in its broad meaning, possibly including a variant of an algorithm. The *search space*, corresponding to the possible set of values of the *tunable parameters* can be very large in practice. Depending on the context, on the purpose and on the complexity of the search space, different approaches may be employed. Vendors can afford dedicated machines for delivering highly tuned libraries (4; 6; 5) and have thus limited constraints in terms of time spent in exploring the search space. Some of the vendors e.g. NVIDIA have not yet provided highly optimized BLAS for their platform (e.g. GPUs). Section 4 describes a framework for parameterizing and auto-tuning BLAS algorithms described in Section 2. As BLAS are very critical for hybrid algorithms and GPUs are new, an exhaustive or user supervised approach is incorporated to tune GPU BLAS kernels.

At higher level, libraries that are on top of efficient BLAS kernels provided by vendors or aim at being portable and efficient on a wider range of architectures cannot afford a virtually unlimited time for tuning. For instance, the Automatically Tuned

Linear Algebra Software (ATLAS) library (8) aims at achieving high performance on a large range of platforms. To do so, empirical tuning is performed at installation time. There is thus a trade-off between the time the user accepts to afford to install the library and the quality of the tuning. In that case, the main difficulty consists in efficiently pruning the search space. Of course, once a platform has been tuned, the information can be shared with the community so that it is not necessary to tune again the library, but this is an orthogonal problem that is not addressed here. Not to mention that the increasing importance of tuning goes beyond the field of dense linear algebra. Among many on-going efforts, the PetaBricks (38) library is a general purpose tuning method providing a language to describe the problem to tune. It has several applications ranging from efficient sorting (38) to multigrid optimization (39). Finally it is important to note that pruning the search is possible, thanks to model-driven considerations. However, in practice, the robustness of the assumptions on the model strongly depends both on the algorithm to be tuned and on the target architecture. There is no clearly identified trend yet, but several model-driven approaches have been successfully led on GPU architectures, such as the matrix vector product (40) or dense linear algebra kernels (26; 10). On the other hand, even on a single-core CPU, basic linear algebra algorithms tend to need more empirical search (8). Indeed, on CPU-based architectures, there are many parameters that are not under user control and difficult to model (different levels of cache, different cache policies at each level, possible memory contention, impact of translation lookaside buffers (TLB) misses, ...).

In this work, the issue of automatically tuning dense linear algebra libraries for multicore and hybrid architectures are presented. In multicore area, PLASMA library was selected. For a matter of conciseness, the focus is on one particular operation, the QR factorization, which is representative of all three one-sided factorizations (QR, LU, Cholesky) currently available in PLASMA. A prune based auto-tuning method has been proposed for tuning PLASMA. Part of the tuning method for PLASMA was considered to tune hybrid MAGMA library.

The report is organized as follows. Section 2 points out state of the art and new algorithmic contributions for different GPU BLAS routines that are crucial for DLA algorithms. Section 3 presents the *Pointer Redirecting* approach for generic GPU kernel developement. An auto-tuning framework for GPU BLAS kernels is described in Section 4. Auto-tuning of PLASMA and MAGMA is presented in Section 5 and Section 6 respectively. Finally I conclude and present future work directions in Section 7.

Chapter 2

BLAS Kernels Developement for GPUs: Algorithmic Perspective

Implementations of the BLAS interface are major building block of dense linear algebra libraries, and therefore have to be highly optimized. This is true for GPU computing as well, especially after the introduction of shared memory in modern GPUs. This is important because it enabled fast Level 3 BLAS implementations for GPUs (2; 26; 11), which in turn made possible the development of DLA for GPUs to be based on BLAS for GPUs (26; 10). Earlier attempts (before the introductions of shared memory) could not rely on memory reuse, only on the GPU's high bandwidth, and as a result were slower than the corresponding CPU implementations. The results of this work are included in the recently released and freely available *Matrix Algebra for GPU and Multicore Architectures* (MAGMA) version 0.2 BLAS Library (10).

Despite the current success in developing highly optimized BLAS for GPUs (2; 26; 11), the area is still new and presents numerous cases/opportunities for improvements. The part of my work addresses several very important kernels, namely the matrix-matrix multiplication that are crucial for the performance throughout

DLA, and matrix-vector multiplication that are crucial for the performance of one-sided factorization, linear solvers, two-sided matrix factorizations (and hence eigensolvers), and iterative refinement procedures. An efficient BLAS routines can be achieved by following seven steps:

- i. We need to understand the numerical problem.
- ii. We have to study the underlying architecture.
- iii. We have to select an existing algorithm that seem to be promising for the underlying problem in the given architecture. If there is not any efficient algorithm we have to device a new one.
- iv. We have to parameterize the selected algorithm.
- v. We have to tune the parameters and selct the best kernel.
- vi. We need to compute the ratio between achieved performance and theoretical peak performance for the implemented kernel on the particular machine. If the ration is reasonable, we can stop here. Otherwise we have to go to step 7.
- vii. We have to start over again. But it's not always clear from where we have to start. The problem might be with the algorithm selected in step 3 that fails to exploit all the architectural features. It could be poor understanding of the architecture in step 2. It could be the problem itself. For example due to low compute to data ratio, the performance of Level 2 BLAS routines are limited by the memory wall in current architecture.

More or less it is an iterative procedure. If tuning part is not automated the procedure is painful, and often referred as *Hand Tuning* which involves human hour, frustration, and more frustration. My contributions are better algorithmic solution for a subset of BLAS routines in GPUs and an autuning framework for tuning those algorithms.

This section describes some of the basic principles on how to write high performance kernels for GPUs. Along with the specifics on developing each of the

BLAS considered, the stress is on two important issues for achieving high performance. Namely, these are:

Blocking Blocking is a DLA optimization technique where a computation is organized to operate on blocks/submatrices of the original matrix. The idea is that blocks are of small enough size to fit into a particular level of the CPU’s memory hierarchy so that once loaded, to reuse the blocks’ data to perform the arithmetic operations that they are involved in. This idea can be applied for GPUs, using GPUs’ shared memory. As demonstrated below, the application of blocking is crucial for the performance of numerous GPU kernels.

Coalesced Memory Access GPU global memory accesses are costly and not cached, making it crucial for the performance to have the right access pattern to get maximum memory bandwidth. There are two access requirements (16). The first is to organize global memory accesses in terms of parallel consecutive memory accesses – 16 consecutive elements at a time by the threads of a half-warp (16 threads) – so that memory accesses (to 16 elements at a time) be **coalesced** into a single memory access. This is demonstrated in the kernels’ design throughout the section. Second, the data should be properly aligned. In particular, the data to be accessed by half-warp should be aligned at $16 * \text{sizeof}(\text{element})$, e.g., 64 for single precision elements.

Clearly, fulfilling the above requirements will involve partitioning the computation into blocks of fixed sizes (e.g., multiple of 16) and designing memory accesses that are coalescent (properly aligned and multiple of 16 consecutive elements). This is demonstrated in the kernels’ design throughout the section. The problem of selecting best performing partitioning sizes/parameters for the various algorithms as well as the cases where (1) the input data is not aligned to fulfill coalescent memory accesses and (2) the problem sizes are not divisible by the partitioning sizes required for achieving high performance, need special treatment and are considered in Section 3. The main

ideas in this section are demonstrated on general and symmetric matrices, in both the transpose and non-transpose cases.

The BLAS considered are not exhaustive; only subroutines that are critical for the performance of MAGMA are discussed. Moreover, these would often be DLA-specific cases that can be accelerated compared to CUBLAS (2), an implementation of the BLAS standard provided by NVIDIA.

Further down a *thread block* will be denoted by TB, its size by N_{TB} (or $N_{TBX} \times N_{TBY}$ in 2D), the number of threads in a TB by N_T (or $N_{TX} \times N_{TY}$ in 2D), and the size associated with blocking (as described above) by nb .

2.1 Level 1 BLAS

Implementing Level 1 BLAS, especially reduce-type operations like dot-product, isamax, etc., is of general interest for parallel computing, but not in the area of DLA. The reason is that Level 1 BLAS are of very low computational intensity (flops *vs* data required) and are avoided at first place (at algorithm design level) in DLA. Even when they can not be avoided algorithmically, e.g., the use of isamax in LU for pivoting, their computation on the GPU is avoided by scheduling their execution on the CPU (10). One operation that fits very well the GPU architecture, and therefore can be efficiently executed on GPUs, is xAXPY:

$$y := \alpha x + y,$$

where x and y are vectors of size N , and α is a scalar. An example of its use is the mixed-precision iterative refinement solvers in MAGMA (17).

The implementation is straightforward – one dimensional TB of size N_{TB} computes N_{TB} consecutive elements of the resulting vector y (a thread per element; also illustrated in Figure 2.1(a)). Important for achieving high performance in this case, as discussed at the beginning of this section, is coalesced memory accesses, tuning N_{TB}

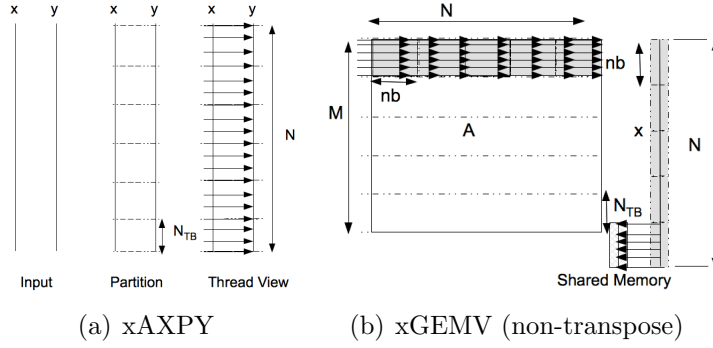


Figure 2.1: Algorithmic view of Level 1 and Level 2 BLAS.

and properly handling the case when N is not divisible by N_{TB} (i.e., $N \% N_{TB} \neq 0$). These are recurring issues for obtaining high-performance BLAS and will be further discussed in the context of other BLAS kernels and GPU optimization techniques like auto-tuning (in Section 4) and pointer redirecting (in Section 3).

Tunable Parameters N_{TB}

Note that the algorithm described satisfies the first requirement for coalescent memory access – to organize global GPU memory accesses in terms of parallel consecutive memory accesses. The pointer redirecting technique in Section 3.1 deals with the second requirement for coalescent memory access, namely cases where the starting address of x is not multiple of $16 * \text{sizeof}(\text{element})$ and/or $N \% N_{TB} \neq 0$. The same applies for the other BLAS kernels in the section and will not be explicitly mentioned again.

2.2 Level 2 BLAS

Level 2 BLAS routines, similar to Level 1 BLAS, are of low computational intensity and ideally, DLA algorithms must be designed to avoid them. An example from the area of DLA is the *delayed update* approach where the application of a sequence of Level 2 BLAS is delayed and accumulated in order to be applied at once as a more efficient single matrix-matrix multiplication (14). In many cases, like MAGMA's

mixed-precision iterative refinement solvers (17) or two-sided matrix factorizations (18), this is not possible, and efficient implementations are crucial for the performance. This section considers the GPU implementations of two fundamental Level 2 BLAS operations, namely the matrix-vector multiplication routines for correspondingly general (xGEMV) and symmetric matrices (xSYMV).

2.2.1 xGEMV

The xGEMV matrix-vector multiplication routine performs one of:

$$y := \alpha Ax + \beta y \quad \text{or} \quad y := \alpha A^T x + \beta y,$$

where A is an M by N matrix, x and y are vectors, and α and β are scalars. The two cases are considered separately as follows:

Non-Transposed Matrix: The computation in this case can be organized in one dimensional grid of TBs of size N_{TB} where each block has $N_T = N_{TB}$ threads, as shown in Figure 2.1(b). Thus, each thread computes one element of the resulting vector y .

GEMV is the first of the kernels considered to which *blocking* can be applied. Although matrix A can not be reused in any blocking, vector x can be reused by the threads in a TB. Specifically, the computation is blocked by loading nb consecutive elements of x at a time into the shared memory (using all N_T threads). This part of x is then used by all N_T threads in a TB to multiply it by the corresponding $N_{TB} \times nb$ submatrix of A . The process is repeated $\frac{N}{N_{TB}}$ times.

Tunable Parameters N_{TB} and nb

Note that the algorithm as described depends on two parameters – N_{TB} and nb . Figures 6.2(a), 6.2(b) compare the performance for cases $N_{TB} = nb = 16, 32, 64$ with that of CUBLAS-2.3. The performances are for matrix sizes $M = N$ that are

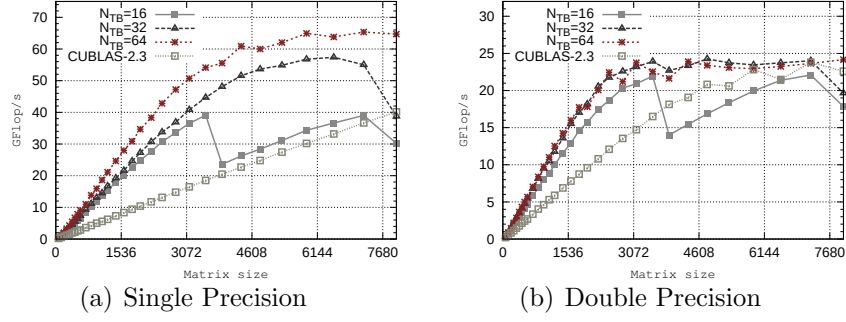


Figure 2.2: Performance of xGEMV (non-transpose) on a GTX 280.

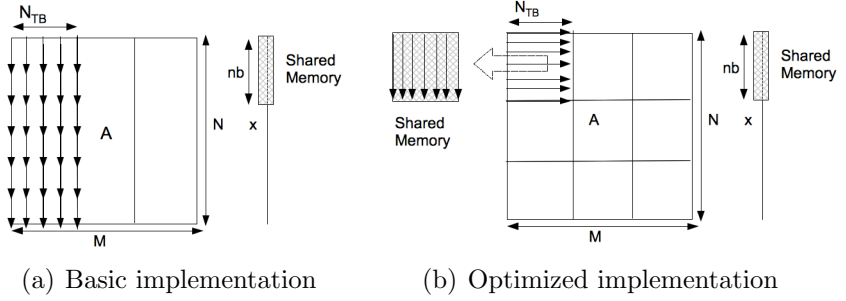


Figure 2.3: Two memory access implementations of xGEMV (transpose).

divisible by the corresponding blocking sizes. Also, the starting addresses of A , x , and y are taken to be divisible by $16 * \text{sizeof}(\text{element})$ and the leading dimension of A is divisible by 16. This guarantees that all memory accesses in the algorithm are coalescent.

Transposed Matrix: Following the non-transposed version approach leads to poor performance because the memory accesses are not going to be coalesced (see Figure 2.3(a)).

To improve the speed on accessing the data, blocks of the matrix A can be first loaded into the shared memory using coalesced memory accesses, and second, data only from the shared memory can be used to do all the necessary computations (see Figure 2.3(b)).

Although the new version significantly improves the performance, experiments that increase the design space of the algorithm, show that further improvements

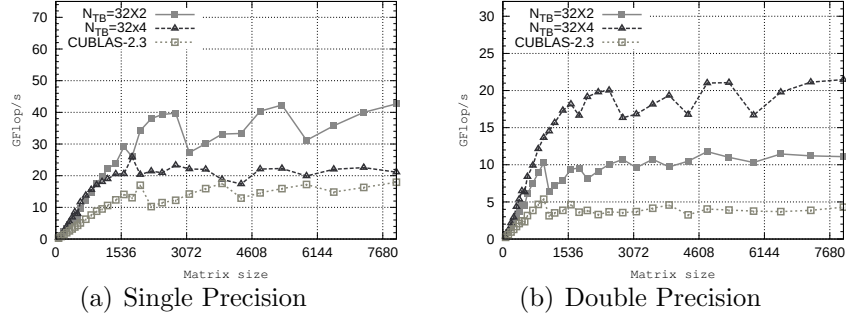


Figure 2.4: Performance of xGEMV (transpose) on a GTX 280.

are possible. In particular, one exploration direction is the use of higher number of threads in a TB, e.g. 64, as high performance DLA kernels are associated with the use of 64 threads (and occasionally more). Using 64 threads directly does not improve performance though because the amount of shared memory used (a 64×64 matrix) gets to be excessive, prohibiting the effective scheduling of that amount of threads (16). Decreasing the use of shared memory, e.g., to a 32×32 matrix, while having higher level of thread parallelism, e.g., a grid of 32×2 threads, is possible in the following way – (1) two groups of 32×1 threads, e.g., denoted by 32_j where $j = 0/1$, load correspondingly the two 32×16 submatrices of the shared memory matrix using coalesced memory accesses, (2) each group performs the computation from the second GEMV version but constrained to the 16×32 submatrix of the shared memory matrix, accumulating their independent y_j results. The final result $y := y_0 + y_1$ can be accumulated by one of the $j = 0/1$ threads.

The same idea can be used with more threads, e.g., 32×4 , while using the same amount of shared memory. Performance results are shown in Figure 2.4 along with a comparison to the performance from CUBLAS 2.3.

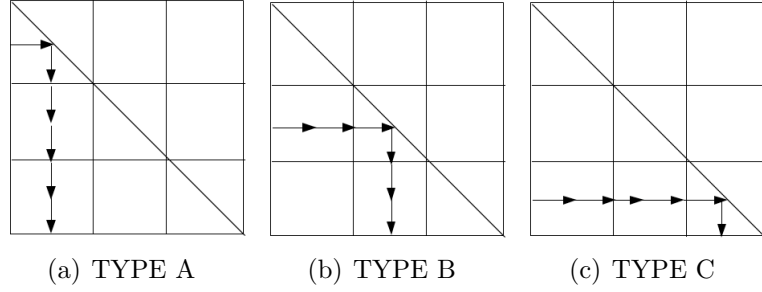


Figure 2.5: Three cases of TB computations in xSYMV.

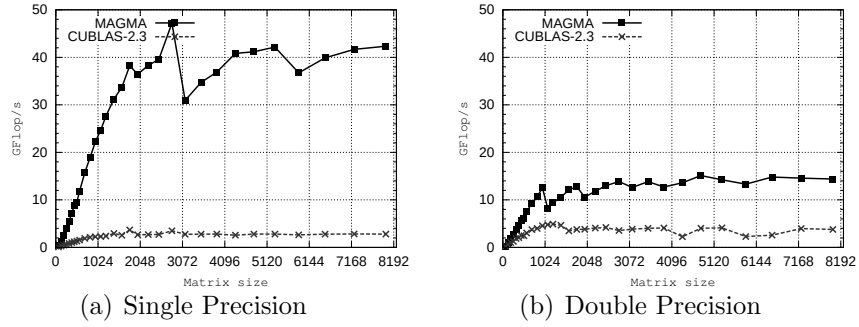


Figure 2.6: Performance of xSYMV on a GTX 280.

2.2.2 xSYMV

The xSYMV matrix-vector multiplication routine performs:

$$y := \alpha Ax + \beta y,$$

where α and β are scalars, x and y are vectors of size N , and A is an N by N symmetric matrix, stored in the upper or lower triangular part of a two-dimensional array of size $N \times N$. The difficulty of designing a high performance SYMV kernel stems from the triangular data storage, which is more challenging to organize a data parallel computation with coalescent memory accesses. Indeed, if A is given as an $N \times N$ array, storing both the upper and lower triangular parts of the symmetric matrix A , the SYMV kernel can be implemented using GEMV. Similar to GEMV,

the computation is organized in one dimensional grid of TBs of size N_{TB} , where each block has $N_T = N_{TB}$ threads.

A TB computation can be classified as one of three cases (see the illustration in Figure 2.5):

- Type A – TB threads do SYMV followed by GEMV (transpose);
- Type B – threads do GEMV (non-transpose) followed by SYMV and GEMV (transpose);
- Type C – threads do GEMV (non-transpose) followed by SYMV.

This way the computation within a TB is converted into one/two GEMVs (to reuse the GEMV kernels) and a SYMV involving a matrix of size $N_{TB} \times N_{TB}$. The remaining SYMV is also converted into a GEMV by loading the $N_{TB} \times N_{TB}$ matrix into the GPU's shared memory and generating the missing symmetric part in the shared memory (a process defined as *mirroring*). Figure 2.6 compares the performance for kernel with parameters $N_{TB} = nb = 32$, $N_T = 32 \times 4$ with that of CUBLAS-2.3.

Although the algorithm described above yields better performance comparing to CUBLAS-2.3 in GTX280, the observed performance is far away from the theoretical peak performance that relates to the bandwidth of the GPU. SGEMV in GTX 280 gets upto 66 GFlops/s. As the bandwidth is 70GBytes/s, one might expect that the performance of SSYMV will be in the vicinity of 99 GFlops/s. The previous algorithm does not take the structure of the symmetric matrix into consideration. It loads the full A matrix whereas loading half of the symmetric matrix would have been sufficient. This insight provides the motivation for finding a better algorithms for xSYMV that runs efficiently on GPUs by taking advantage of the data storage formate of symmetric matrix.

In the new algorithm for xSYMV, the computation is also organized in one dimensional grid of TBs of size N_{TB} as it was done for previous algorithm, where each block has $N_T = N_{TB}$ threads. The layout of the thread block is irrelevant

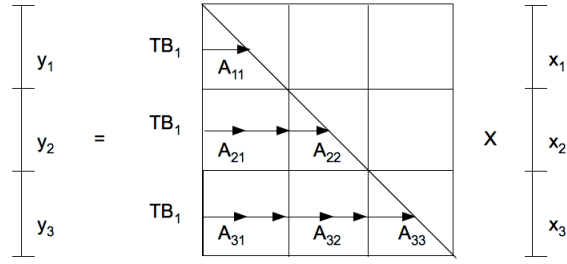


Figure 2.7: Data access pattern in new xSYMV algorithm.

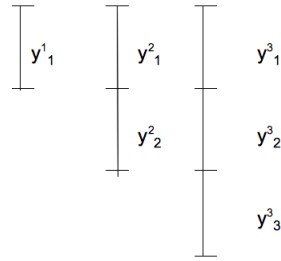


Figure 2.8: Results produced by each thread block in new xSYMV algorithm.

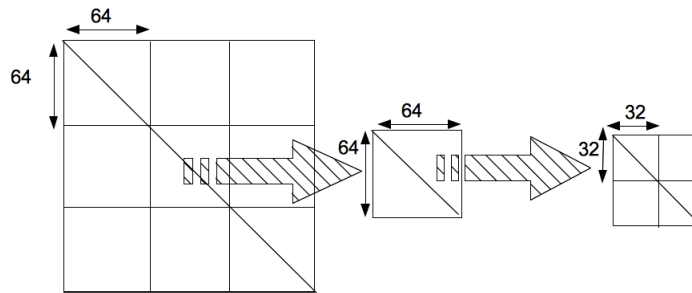


Figure 2.9: Recursive blocking in new xSYMV algorithm.

as inside a single kernel the threads can rearrange themselves on the fly to match the required computation or memory access pattern. Thread block TB_i will access blocks $\{A_{i,j} : 1 \leq j \leq i\}$ from matrix A . as shown in Figure 2.7. Some blocks $\{A_{i,j} : i \neq j\}$ can be used twice to compute partial results of resultant vectors y_i and y_j . So instead of computing a single final vector y_i , TB_i will be computing partial results of vectors $\{y_j : 1 \leq j \leq i\}$. These partial result vectors produced by TB_i are named as $\{y_j^i : 1 \leq j \leq i\}$ as shown in Figure 2.7. The computation by TB_i will be as follows:

$$y_j^i := A_{i,j}^T x_i \text{ for } j = 1 \text{ to } i - 1$$

$$y_i^i := \sum_{j=1}^{j=i} A_{i,j} x_j$$

As described in the first algorithm, the missing symmetric part in the diagonal blocks $A_{i,i}$ are produced using mirroring. This completes the first phase of new xSYMV algorithm. Finally another kernel in the same one dimensional grid formate is launched to compute the final y_i 's as follows:

$$y_i := \sum_{j=i}^{j=|TB|} y_i^j$$

Here $|TB|$ is the number of required blocks for a matrix size N , $|TB| = \lceil \frac{N}{N_{TB}} \rceil$. However the algorithm described above has some overhead in terms of time and space. It launches an extra kernel to add up the partial results y_j^i . It requires some extra memory to store the partial results. The extra memory requirement is:

$$\frac{N_{TB} \times |TB| \times (|TB| + 1)}{2}$$

There are two tunable parameters in the above algorithm: N_{TB} and N_T . Usually bigger values of N_{TB} brings greater performance. With $N_{TB} = 64$, we will need

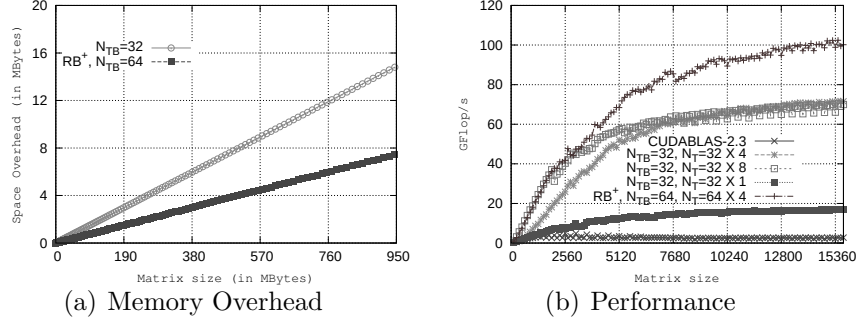


Figure 2.10: xSYMV in single precision with new algorithm on GTX280, RB^+ means recursive blocking was used

64×64 dimension of shared memory for the on the fly mirroring operation in the diagonal computations, $A_{i,i} \times x_i$. Due to the limited amount of shared memory in GPUs, the above algorithm fails to work with $N_{TB} = 64$. But this limitation can be overcome by using recursive blocking as shown in Figure 2.9. With $N_{TB} = 64$ and $N_T = 256$, a 64×16 dimension of matrix is allocated in shared memory. In the off-diagonal computations, $A_{i,j}^T \times x_i$ where $i \neq j$ or $A_{i,j} \times x_j$ where $i \neq j$, the layout of the thread block is $N_T = 256 = 64 \times 4$. The mechanism for these off-diagonal computations are straight forward. The diagonal computations, $A_{i,i}x_i$, are performed in a recursive way using the same kernel with block size $T_{NB} = 32$. As we can see from Figure 2.9, there will be two such blocks. These two blocks are processed sequentially by the same 256 threads. During recursive part of the kernel, 256 threads inside a thread block rearrange themselves as 32×8 threads to meet the computation and data access pattern.

All the intermediate results are stored in the register instead of global memory. Figures 2.10(b) compares the performance for cases $N_{TB} = 32$ with $N_T = 32 \times 1$, $N_{TB} = 32$ with $N_T = 32 \times 4$, $N_{TB} = 32$ with $N_T = 32 \times 8$, recursive $N_{TB} = 64$ with $N_T = 64 \times 4$ with that of CUBLAS-2.3 on GTX280. Figure 2.10(a) shows the memory overhead for different values of N_{TB} . With $N_{TB} = 32$, the space overhead is 1.56% of the matrix size and with $N_{TB} = 64$ the space overhead is 0.78% of the matrix size. Not only $N_{TB} = 64$ with recursive blocking offers better performance, it also reduces

the space overhead by a factor of two comparing to the kernels with $N_{TB} = 32$. The only problem with this algorithm is that if there is not enough memory available on the GPU, the code will not be able to execute.

2.3 Level 3 BLAS

Level 3 BLAS routines are of high computational intensity, enabling their implementations (and that of high level DLA algorithms based on Level 3 BLAS) to get close within the computational peak of ever evolving architectures, despite that architectures are evolving with an exponentially growing gap between their compute and communication speeds. The shared memory of GPUs, similar to memory hierarchy in standard CPUs, can be used to develop highly efficient Level 3 BLAS kernels. This section describes the GPU implementations of three primary Level 3 BLAS operations – the matrix-matrix multiplication (xGEMM), the symmetric rank-k update (xSYRK), and the triangular matrix solver (xTRSM).

2.3.1 xGEMM

The xGEMM matrix-matrix multiplication routine performs one of:

$$C := \alpha \text{op}(A)\text{op}(B) + \beta C,$$

where $\text{op}(X)$ is X or X^T , α and β are scalars, and A , B and C are matrices. , with $\text{op}(A)$ an M by K matrix, $\text{op}(B)$ a K by N matrix and C an M by N matrix. Crucial for the performance is the application of blocking – schematically represented in Figure 3.2(a) for the case of $C := \alpha AB + \beta C$ and described as follows (26). The computation is done on a two-dimensional grid of TBs of size $N_{TBX} \times N_{TBY}$ and each TB is assigned to $N_T = N_{TX} \times N_{TY}$ threads.

For simplicity, take $N_T = N_{TBX}$. Then, each thread is coded to compute a row of the sub-matrix assigned to the TB. Each thread accesses its corresponding row of A ,

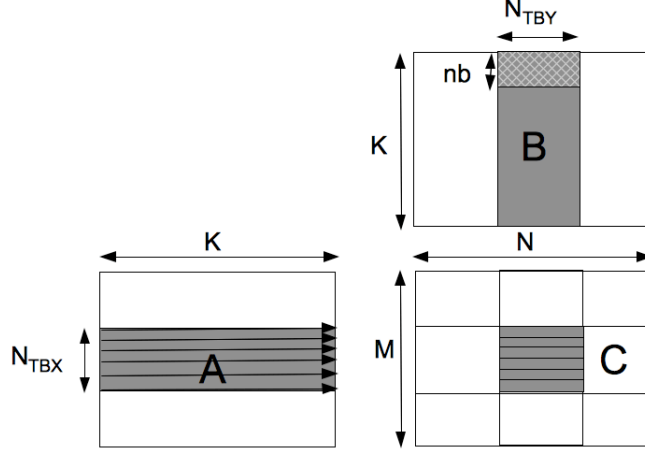


Figure 2.11: The GPU GEMM ($C = AB$) of a single TB.

as shown by an arrow, and uses the $K \times N_{TBY}$ sub-matrix of B for computing the final result. This TB computation can be blocked, which is crucial for obtaining high performance. In particular, sub-matrices of B of size $nb \times N_{TBY}$ are loaded into shared memory and multiplied nb times by the corresponding $N_{TBX} \times 1$ sub-matrices of A . The $N_{TBX} \times 1$ elements are loaded and kept in registers while multiplying them with the $nb \times N_{TBY}$ part of B . The result is accumulated to the resulting $N_{TBX} \times N_{TBY}$ sub-matrix of C , which is kept in registers throughout the TB computation (a row per thread, as already mentioned). This process is repeated until the computation is over. All memory accesses are coalesced.

Kernels for various N_{TBX} , N_{TBY} , N_{TX} , N_{TY} , and nb can be automatically generated (see Section 4) to select the best performing for particular architecture and GEMM parameters.

A sample choice of these kernels is shown in Table 2.1. Figure 2.12 compares their performances with that of CUBLAS-2.3 on square matrices. K1 performs well for small matrices (e.g., of dimension ≤ 512) as it provides more parallelism compared to the other kernels in Table 2.1. The performance deteriorations experienced by some of the kernels are due to the GPUs global memory layout and memory access patterns of

Kernel	N_{TBX}	N_{TBY}	nb	N_{TX}	N_{TY}
K1	32	8	4	8	4
K2	64	16	4	16	4
K3	128	16	8	16	8
K4	256	16	16	16	16

Table 2.1: Key parameters of a sample of GPU GEMM kernels.

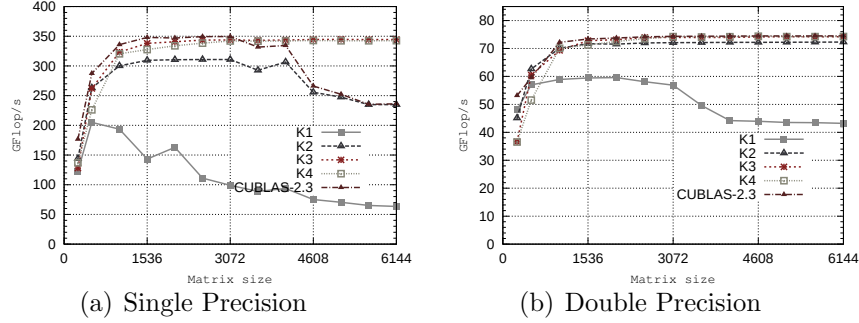


Figure 2.12: Performance of GEMM ($C = \alpha AB^T + \beta C$) on a GTX 280.

hitting a particular memory module (a phenomena referred to by NVIDIA as *partition camping*).

This particular configuration works well when $Op(A) = A$, $Op(B) = B$. The $Op(A) = A^T$, $Op(B) = B^T$ case is similar – only the arguments order and the update location of C at the end of the kernel have to be changed, as:

$$C := \alpha A^T B^T + \beta C \quad \text{or} \quad C^T := \alpha BA + \beta C^T.$$

The $Op(A) = A^T$, $Op(B) = B$ kernel can be analogously developed except that both A and B must be stored into shared memory.

NVIDIA's new architecture *Fermi* has brought the prospect of incredible performance for DLA algorithms as well as for a large domain of scientific computing applications. Although the basic architecture of *Fermi* and it's predecessor GPUs, e.g. GTX280, have a wide range of architectural features in common, there are subtle differences. Those changes in architecture has necessitates the need for upgrading

most of the BLAS for DLA algorithm. A highly optimized kernel for previous GPUs such as Tesla C1060, GTX280, fails to achieve reasonable performance in GPUs with Fermi architecture, e.g. Tesla C2060. Note to mention that the latency to access register and shared memory were comparable in GTX280 or Tesla C1060. But in the new architecture Fermi, accessing data from shared memory is several magnitude slower than accessing data from registers. Moreover the number of memory banks has increased from 16 in GTX280 to 32 in Fermi. This gives us the motivation for redesigning all the BLASs in particular xGEMM for Fermi to get most of the theoretical peak.

The algorithmic view of xGEMM for Fermi is shown in Figure 2.3.1. Similarly as in xGEMM kernel for GTX, the computation is divided into two-dimensional grid of TBs of size $N_{TBX} \times N_{TBY}$ and each TB is assigned to $N_T = N_{TX} \times N_{TY}$ threads. In case of Fermi, it has been observed that loading both matrix A and matrix B into shared memory brings good performance. It is beneficial because it leads to better use of register blocking technique with square shape. For simplicity of description, a set of values for the parameters are selected, $N_{TBX} = N_{TBY} = 64$ and $N_{TX} = N_{TY} = 16$. With this parameter values, 16×16 threads will be computing 64×64 elements of matrix C . Hence each thread will compute 16 elements. The 64×64 block of matrix C is divided into 16 sub-blocks of dimension 16×16 as shown in then mentioned Figure. Each 16×16 sub-block is computed by a TB of dimension 16×16 . Hence one element is computed by one thread. Element (x, y) represented by green diamond will be computed by thread (x, y) represented by black diamond, for $0 \leq x, y \leq 15$. All the 16 elements computed by thread $(0, 0)$ are shown by black diamonds in the figure. In summary, each thread will be computing a 4×4 matrix with stride 16. This distribution leads to coalesced write of final results from registers to matrix C in global memory.

Before starting each phase of computation, all the threads inside a TB bring 64×16 elements of matrix A and 16×64 elements of matrix B to shared memory in a coalesced way. Depending upon $Op(A)$ and $Op(B)$, 256 threads choose one of the

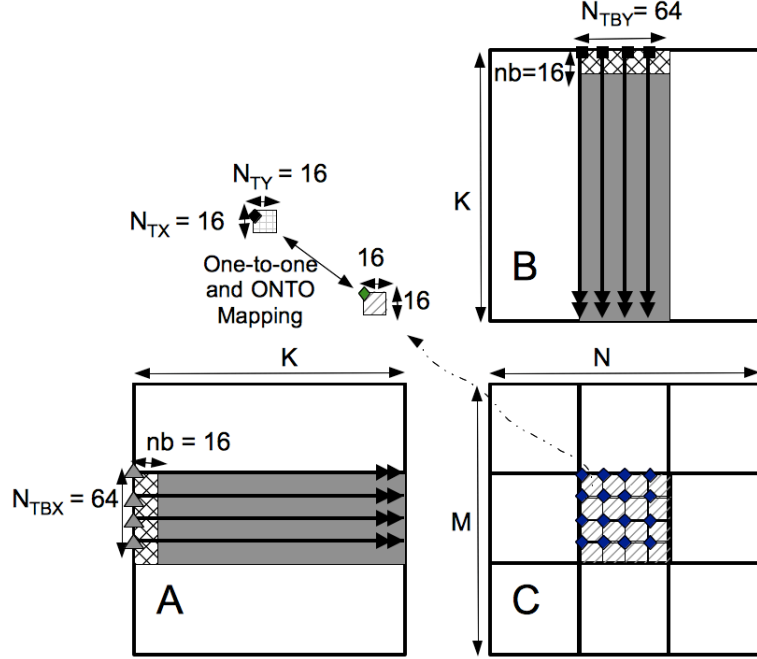


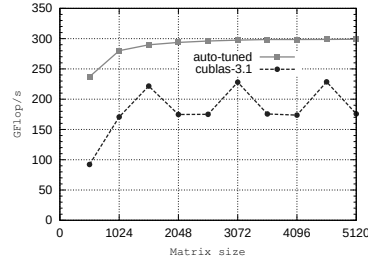
Figure 2.13: The GPU GEMM ($C = AB$) of a single TB in Fermi.

following shapes: 16×16 or 64×4 . This reshaping helps coalesced memory access from global memory. The elements from matrix A and B needed by thread (0,0) is shown by arrows. But these elements are accessed through shared memory. First four elements from shared A (shown by grey triangle) and four elements from shared B (shown by black rectangle) are loaded into registers. Then these 8 elements are used to do 16 FMAD operations. With this register blocking scheme the performance is increased. Note to mention that *Fermi* has level 1 and level 2 caches. In order to be benefited from the cache architecture, all the accesses for matrix A and B are done through texture memory. The performance of xGEMM in Fermi using this algorithm is shown in Figure 2.14.

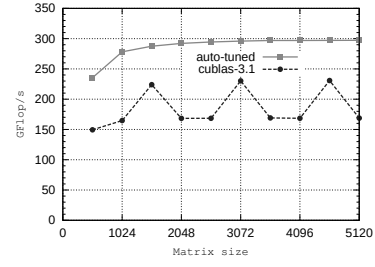
2.3.2 xSYRK

The xSYRK routine performs one of the symmetric rank-k updates:

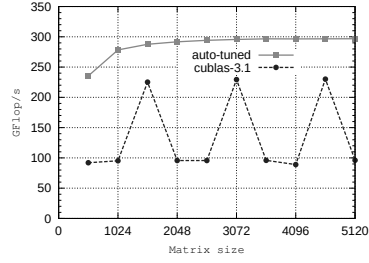
$$C := \alpha AA^T + \beta C \quad \text{or} \quad C := \alpha A^T A + \beta C,$$



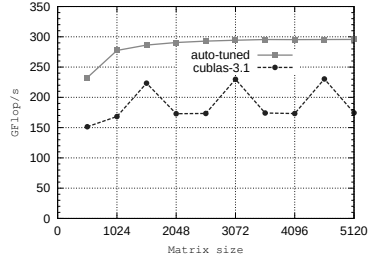
(a) $\text{Op}(A)=N$ and $\text{Op}(B)=N$



(b) $\text{Op}(A)=N$ and $\text{Op}(B)=T$

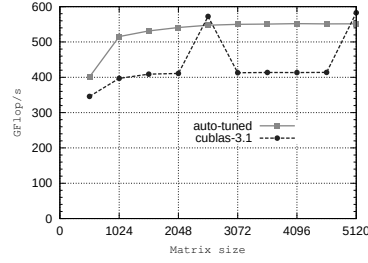


(c) $\text{Op}(A)=T$ and $\text{Op}(B)=N$

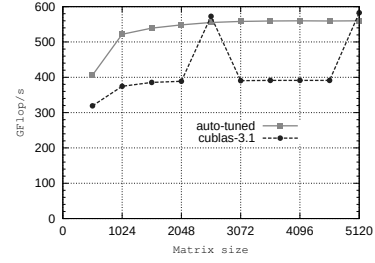


(d) $\text{Op}(A)=T$ and $\text{Op}(B)=T$

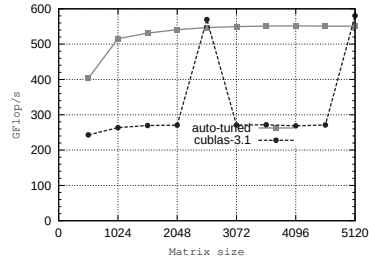
Figure 2.14: Performance of dGEMM on a Fermi.



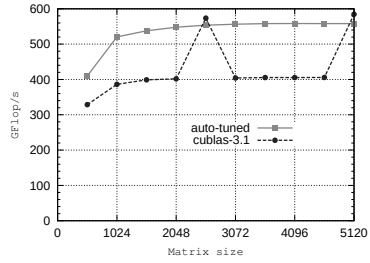
(a) $\text{Op}(A)=N$ and $\text{Op}(B)=N$



(b) $\text{Op}(A)=N$ and $\text{Op}(B)=T$



(c) $\text{Op}(A)=T$ and $\text{Op}(B)=N$



(d) $\text{Op}(A)=T$ and $\text{Op}(B)=T$

Figure 2.15: Performance of dGEMM on a Fermi.

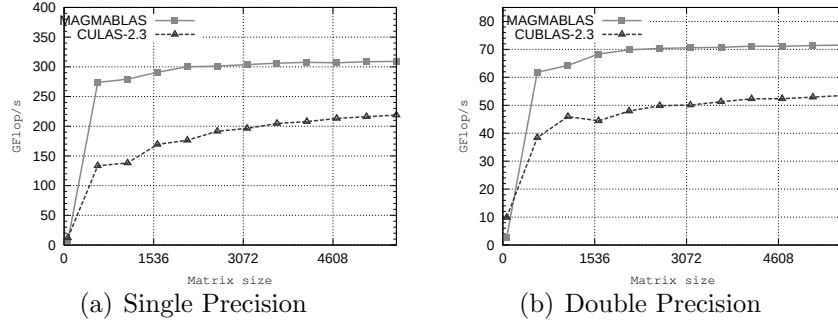


Figure 2.16: Performance of xSYRK on a GTX 280.

where α and β are scalars, C is an $N \times N$ symmetric matrix and A is an $N \times K$ matrix in the first case and a $K \times N$ matrix in the second case. A TB index reordering technique can be used to initiate and limit the computation only to TBs that are on the diagonal or in the lower (correspondingly upper) triangular part of the matrix. In addition, all the threads in a diagonal TB compute redundantly half of the block in a data parallel fashion in order to avoid expensive conditional statements that would have been necessary otherwise. Some threads also load unnecessary data to ensure coalescent global memory accesses. At the end, the results from the redundant computations (in the diagonal TBs) are discarded and the data tile is correctly updated.

2.3.3 xSYR2K

The xSR2YK routine performs one of the symmetric rank-k updates:

$$C := \alpha AB^T + \alpha BA^T + \beta C \quad \text{or} \quad C := \alpha A^T B + \alpha B^T A + \beta C,$$

where α and β are scalars, C is an $N \times N$ symmetric matrix and A is an $N \times K$ matrix in the first case and a $K \times N$ matrix in the second case. This kernel can be implemented by incorporating the TB index reordering technique that was used in xSYRK. The concatenation of two matrix multiplication operations yields the kernel.

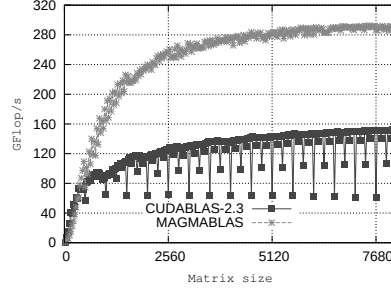


Figure 2.17: Performance of SSYR2K on GTX280

Two tunable parameters are N_{TB} and N_T . The auto-tuner described in Section 4 found a highly optimized kernel by tuning these parameters and applying a state of the art loop optimization technique in particular circular loop skewing. Circular loop optimization reorders the computation (GPUs internal TB scheduling) such a way that the overwall bandwidth from the global memory is maximized. More details can be found in Section 4. The performance shown in figure 2.17 shows the effect of circular loop skewing. The auto-tuned kernel doesn't have any performance oscillation which is acute in CUBLAS-2.3's kernel.

2.3.4 xTRSM

The xTRSM routine solves one of the matrix equations:

$$op(A)X = \alpha B \quad \text{or} \quad Xop(A) = \alpha B,$$

where α is a scalar, X and B are M by N matrices, A is upper/lower triangular matrix and $op(A)$ is A or A^T . Matrix B is overwritten by X .

Trading off parallelism and numerical stability, especially in algorithms related to triangular solvers, has been known and studied before (19; 20). Some of these TRSM algorithms

are getting extremely relevant with the emerging highly parallel architectures, especially GPUs. In particular, the MAGMA library includes implementations that

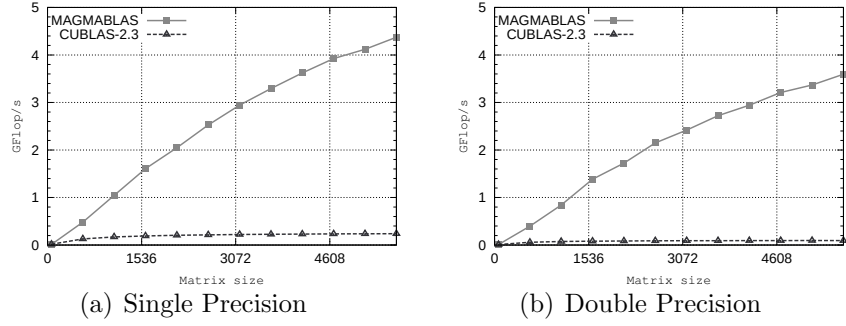


Figure 2.18: Performance of xTRSM on a GTX 280.

explicitly invert blocks of size 32×32 on the diagonal of the matrix and use them in blocked xTRSM algorithms. The inverses are computed simultaneously, using one GPU kernel, so that the critical path of the blocked xTRSM can be greatly reduced by doing it in parallel (as a matrix-matrix multiplication). Variations are possible, e.g., the inverses to be computed on the CPU, to use various block sizes, including recursively increasing it from 32, etc. Similarly to xSYRK, extra flops can be performed to reach better performance – the empty halves of the diagonal triangular matrices can be set to zeros and the multiplications with them done with GEMMs instead of with TRMMs. This avoids diverting warp threads and ensures efficient parallel execution. The algorithm and performance result in Figure 2.18 is due to Peng Du. However a auto-tuned xGEMM was used inside his xTRSM kernels to increase the performance.

Chapter 3

Generic BLAS Kernels

Developement for GPUs: Pointer Redirecting

One current BLAS library for GPUs is NVIDIA's CUBLAS (2). Figure 3.1(a) shows the performance of the single precision matrix-matrix multiplication routine (SGEMM) for a discrete set of matrix dimensions. Figure 3.1(b) shows similar data but for double precision arithmetic. Note that at some dimensions the performance is much higher than at other dimensions, e.g. taken at odd numbers like 65, 129, etc. These performance dips, that actually happen in the majority of matrix dimensions are one of our acceleration targets. The reason for these dips is very likely related to an implementation that has even inner-blocking size to match various hardware parameters and considerations to get high performance. The performance graphs illustrate a quite high performance loss for the cases when the matrix dimension is obviously not a multiple of the inner blocking size. In particular, the performance gap is more than 24 GFlops/s in double precision (around 34% of the peak performance), and is worse for single precision.

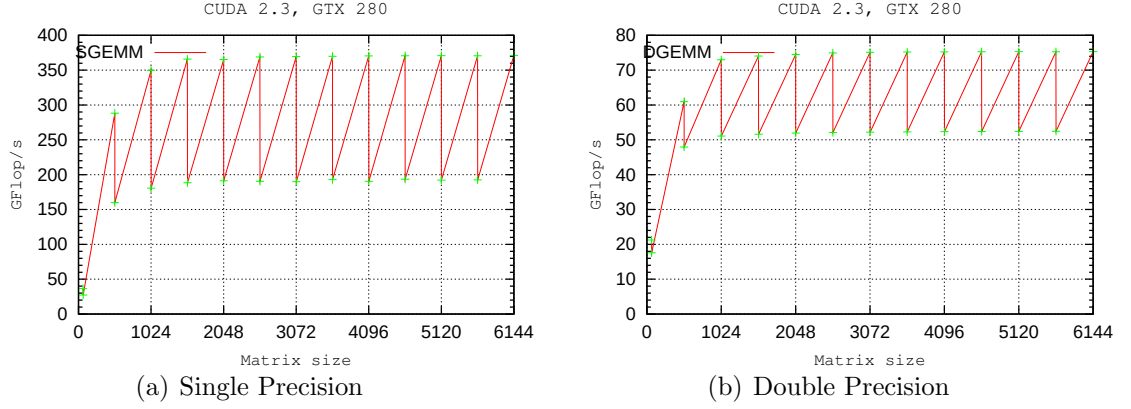


Figure 3.1: GEMM Performance on Square Matrices.

There are ways around to work with these BLAS routines and still get high performance in high level algorithms. One possible solution is to force the user to allocate and work with matrices multiple of the blocking size. This though leads to memory waste. Sometimes it is a burden to the user if the application is already written, and in general is obviously not a good solution. Another solution is padding with 0s to fit the blocking factor, do the computation and keep this transparent to the user. This approach has the overhead of copying data back and forth, and possibly some extra computation. A third approach is to rewrite the kernels in such a way that there are no extra computations, no data movement or any other overheads. This rewriting though is difficult and time consuming, especially taken into account different GPU specifics as related to data coalescing, data parallel computation, computation symmetry, and memory bank layout.

3.1 Pointer Redirecting

The matrix-matrix multiplication (xGEMM; e.g. $C = AB$) algorithm for GPUs is schematically represented in Figure 3.2(a). Matrix C is divided into blocks of size $blk_M \times blk_N$ and each block is assigned to a block of $nthd_x \times nthd_y$ threads. Each thread inside a thread block computes a row of sub matrix $blk_M \times blk_N$. Each thread

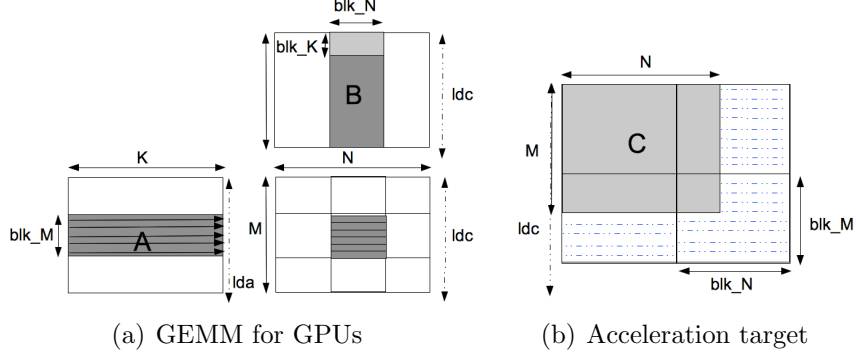


Figure 3.2: The algorithmic view of GEMM for GPUs.

accesses corresponding row of matrix A as shown by an arrow and uses the sub-matrix $K \times blk_N$ of matrix B for computing the final result. As the portion of matrix B needed by each thread inside a thread block is the same, they load a sub-matrix of matrix B of size $blk_N \times blk_K$ from global memory to shared memory in a coalesced way, synchronize themselves, do the computation and repeat until the computation is over. All these happen in a series of synchronized steps. With an optimal selection of $blk_M, blk_N, blk_K, nthd_X, nthd_Y$, we can get the best kernel for the matrix sizes that are divisible by blocking factors, i.e. $M \% blk_M = 0$, $N \% blk_N = 0$, $K \% blk_K = 0$.

The question is how to deal with matrix dimensions that are not divisible by the blocking factor. Whatever solution we choose, we have to keep it transparent to the user while maintaining highest flexibility. The goal is to allow reasonable overhead (if needed) and to achieve high performance in general cases. We show in Figure 3.2(b) matrix C of a xGEMM operation ($C = \alpha C + \beta Op(A)Op(B)$) where dimensions M and N are not divisible by the blocking factor. The matrix has only one full block. We can do the computation for the full block and do the other partial blocks by loading data and doing computation selectively. This will introduce several if-else statements in the kernel which will prevent the threads inside a thread-block to run in parallel. Figure 3.3 shows the performance of one such implementation. Note that GPUs run all the threads inside a thread block in parallel as long as they execute the same instruction on different data. If the threads ever execute different instruction,

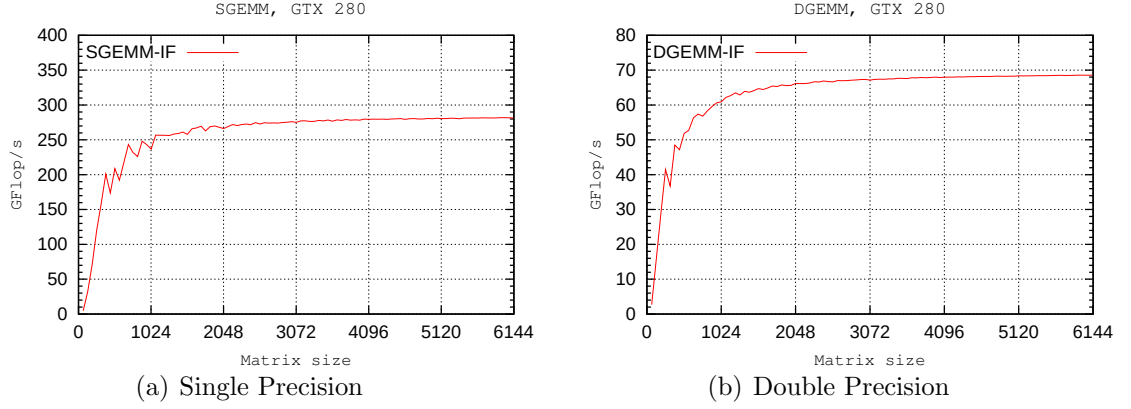


Figure 3.3: GEMM Implementation with Conditional Statement in Inner Loop.

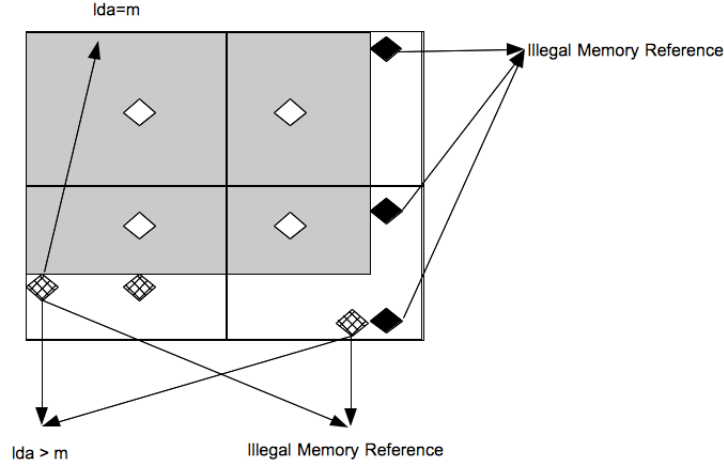


Figure 3.4: Possible Illegal Memory Reference in Matrix Multiply.

their processing would become temporary sequential until they start executing the same instructions again.

Another approach is to let the unnecessary threads do similar work so that the whole thread block can run in data parallel mode. In Figure 3.2(b) the dashed blue lines correspond to unnecessary flops that are done by respective thread. It is not clear yet which data they will operate on, but it also does not matter because the computation will be discarded. Lets take a look at the scenario where all the threads assume that the matrix fits into the block and do the work in a natural way until updating matrix C . In Figure 3.4, the shaded region corresponds to original matrix

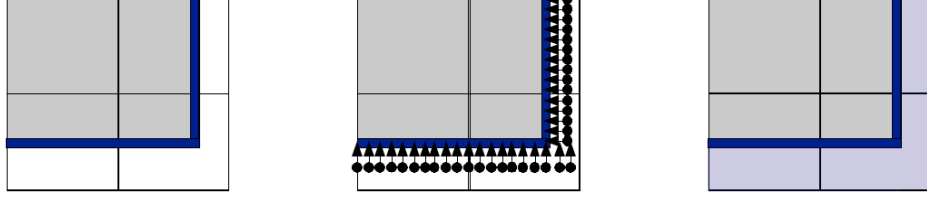


Figure 3.5: (Left) Last Valid Access (Middle) Pointer Redirecting (Right) Mirroring

and the outmost rectangle corresponds to the largest matrix that best fits in terms of blocking factor. We are going to make $\lceil \frac{M}{dim_M} \rceil \times \lceil \frac{N}{dim_N} \rceil$ number of grids and allow threads at the partial block to compute the same way as it is done in a full block. It is evident that memory accesses inside the shaded region in Figure 3.4, denoted by white diamond, are always valid. Memory accesses denoted by red diamonds are always invalid. Memory accesses represented by green diamond could be valid or illegal. As we can see in the Figure 3.4, the leftmost green diamond could be an element from the next column, e.g. when $lda \leq blk_M \times \lceil \frac{M}{blk_M} \rceil$. It could be an element in the same column when $lda > blk_M \times \lceil \frac{M}{blk_M} \rceil$, or it could be invalid memory reference.

In Figure 3.5(Left), the blue lines in last row and last column are last valid memory reference irrespective of any values of lda , M , N , K , blk_M , blk_N , $nthd_X$, $nthd_Y$. If some thread needs to access some memory location beyond this last row/column, we are going to force him reference to this last row/column by adjusting the pointer. These threads will be doing unnecessary computation, we don't care from where this data is coming from. All we care is that together they make best use of memory bandwidth and layout, access data in a coalesced manner. Figure 3.5(Middle) depicts the complete scenario how the memory is referenced. As a result the matrix will have some virtual row where rows beyond the last row are replication of last row and columns beyond the last column are replication of last column. It is shown in Figure 3.5.

Let's see how it fits into xGEMM's (Op(A) = Op(B) = Non-Transposed) context in terms of accessing matrix A. As in Figure 3.6(a), thread t1, t2, t3, t4 will be

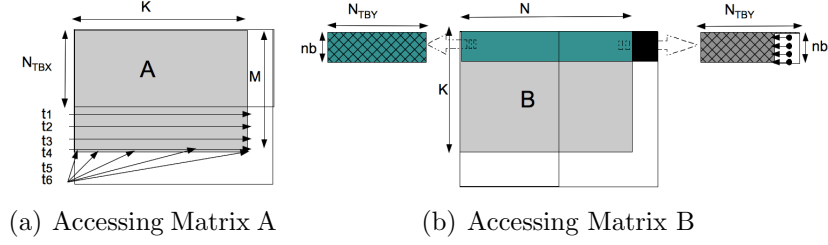
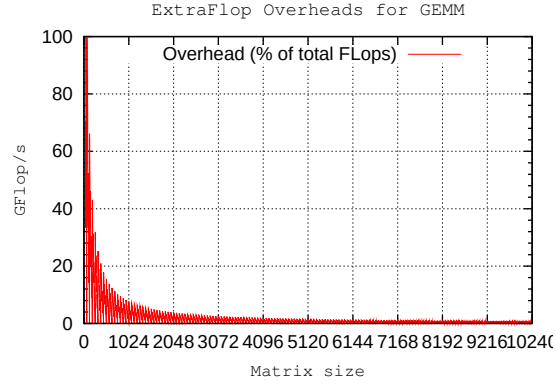


Figure 3.6: Algorithmic view of GEMM for GPUs with Pointer Redirecting.

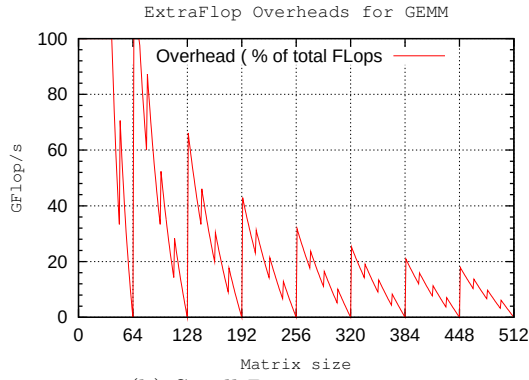
accessing valid memory location. And all the threads beyond thread t4, e.g. thread t5, t6 will be accessing same memory thread t4 is accessing. As a result no separate memory read operation will be issued and no latency will be experienced for this extra load. If we look at Figure 3.6(b), $blk_K \times blk_N$ data of matrix B are brought into shared memory by $nthd_X \times nthd_Y$ threads in a coalesced manner. The left $blk_K \times blk_N$ block is necessary as we can see. But the right $blk_K \times blk_N$ is partially needed. The black portions are unnecessary memory access. As discussed before, it will access the last row or column that is needed instead of accessing invalid memory. This will still be done in a coalesced way and it is accessing less memory now. Some memory are accessed more than once, which doesn't hamper performance. This a simple solution to the problem with little overhead that doesn't break the pattern of coalesced memory access. Note that we will not be doing any extra computation in K dimension, so we don't need to zeroing out values to keep the computation valid.

3.2 Performance

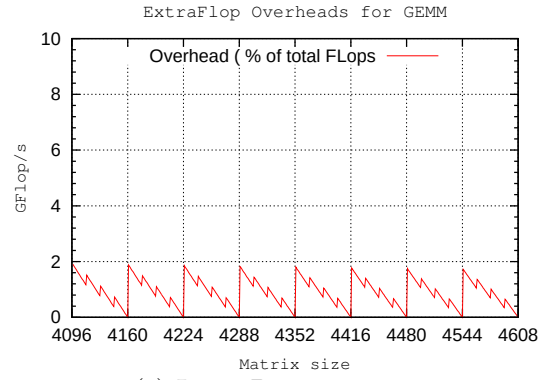
For the unnecessary computation there will be some overhead. Figure 3.7 shows the percentage of extra flops needed for different dimensions of matrix with parameters $blk_M = 64, blk_N = 16, blk_K = 16, nthd_X = 16, nthd_Y = 4$ for different matrix sizes. The overhead is scaled to 100 for visibility. Figure 3.9 and Figure 3.8 shows the performance results for GEMM in single and double precision respectively. In double precision we are seeing an improvement of 24 GFlops/s and in single precision it is



(a) All Dimension

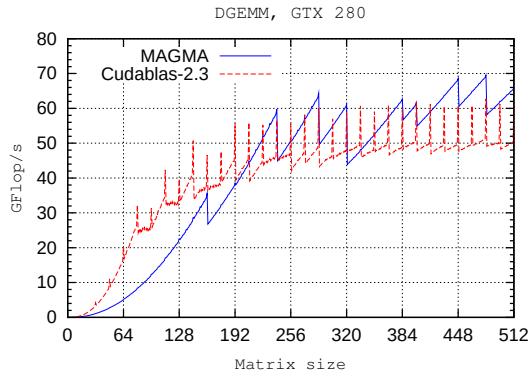


(b) Small Dimension

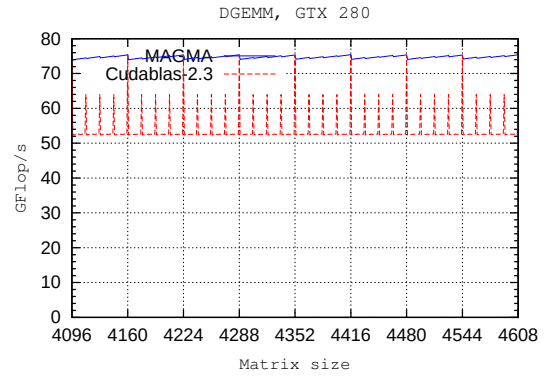


(c) Large Dimension

Figure 3.7: Flops overhead in xGEMM



(a) Small Dimension



(b) Large Dimension

Figure 3.8: Performance dGEMM

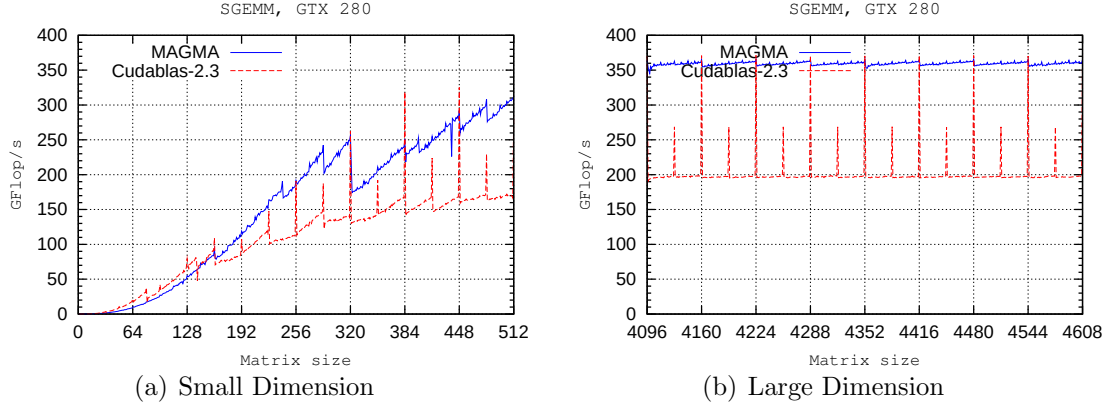


Figure 3.9: Performance sGEMM

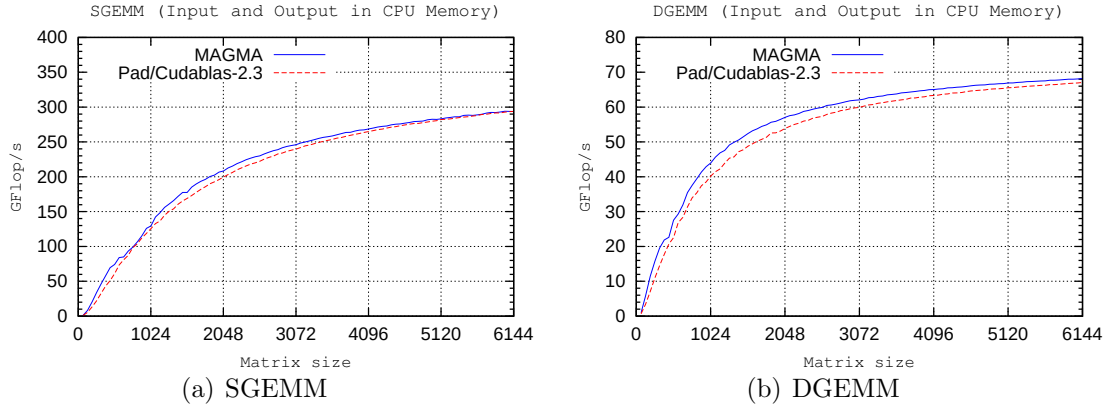


Figure 3.10: Performance xGEMM with Padding (Data In/Out in CPU Memory).

like 170 GFlops/s. As we have discussed before other than small dimensions the improvement is significant. The zig-zag patterns in performance graph resembles the blocking factor of the kernel.

As we have discussed before, if the matrices are in CPU memory one can use padding, e.g., as in (12). We have to allocate a bigger dimension of matrix in GPU memory, put zeroes in the extra elements, then transfer the data from CPU to GPU and then call the Kernel. Figure 3.10 shows the performance comparison when data is in CPU memory. It is evident that for small matrix size our implementation is better and for higher dimension they are very identical. We note that the pointer redirecting

Matrix Size	CUBLAS	MAGMA BLAS
1001	47.65	46.01
2001	109.69	110.11
3001	142.15	172.66
4001	154.88	206.34
5001	166.79	226.43
6001	169.03	224.23
7001	175.45	246.75
8001	177.13	251.73
9001	179.11	269.99
10001	180.45	262.90

Table 3.1: Performance comparison between MAGMA BLAS with pointer redirecting and CUBLAS for the QR factorization in single precision arithmetic

approach does not use extra memory, does not require a memory copy if non padded matrix is given on the GPU memory, and finally does not require initialization of the padded elements.

Table 3.1 shows the performance of the one-sided QR factorization using CUBLAS and MAGMA BLAS for matrix sizes not divisible by the kernel’s block size. The pointer redirecting approach brings 20% to 50% performance improvement over CUBLAS in this case. This approach is extendable to other BLAS routines such as xGEMV, xSYRK, xSYR2K, xSYMV, etc.

Chapter 4

Autotuning BLAS Kernels for GPUs: MAGMABLAS

Automatic performance tuning (optimization), or auto-tuning in short, is a technique that has been used intensively on CPUs to automatically generate near-optimal numerical libraries. For example, ATLAS (8; 21) and PHiPAC (22) are used to generate highly optimized BLAS. In addition, FFTW (23) is successfully used to generate optimized libraries for FFT, which is one of the most important techniques for digital signal processing.

With the success of auto-tuning techniques on generating highly optimized DLA kernels on CPUs, it is interesting to see how the idea can be used to generate near-optimal DLA kernels on modern high-performance GPUs.

Indeed, work in the area (24) has already shown that auto-tuning for GPUs is very practical solution to easily port existing algorithmic solutions on quickly evolving GPU architectures and to substantially speed up even highly hand-tuned kernels.

There are two core components in a complete auto-tuning system:

Code generator The code generator produces code variants according to a set of pre-defined, parametrized templates/algorithms. The code generator also applies certain state of the art optimization techniques.

Heuristic search engine The heuristic search engine runs the variants produced by the code generator and finds out the best one using a feedback loop, e.g., the performance results of previously evaluated variants are used as a guidance for the search on currently unevaluated variants.

Below is a review of certain techniques and choice of parameters that significantly impact the performance of the GEMM kernel. Therefore, these techniques and parameters must be (and have been) incorporated into the code generator of an auto-tuning GEMM system. The ultimate goal is to develop similar auto-tuning for all of the BLAS of interest.

4.1 Auto-tuning GEMM

Figure 3.2 depicts the algorithmic view of a GEMM code template. It was already mentioned that five parameters can critically impact performance (see Table 2.1 for a sample choice), and therefore are incorporated in a GEMM code generator. This choice though can be extended and enhanced with various optimization techniques:

Number of threads computing a row: Section 2.3.1 imposed the constraint $N_{TX} \times$

$N_{TY} = N_{TBX}$ so that each thread in a TB is computing an entire row of the submatrix of C computed by the TB (denoted further as BC). This constraint can be lifted to introduce an additional template parameter. Depending upon the value of N_T each thread will compute either an entire row or part of a row. For example, suppose $N_{TBY} = 16$ and $N_{TBX} = 64$, and the TB has 16×4 threads, then each thread will compute exactly one row of BC . If the thread block has 16×8 threads, then each thread will compute half of a row.

A/B being in shared memory: As described in Section 2.3.1, whether A or B is put into shared memory plays a crucial factor in the kernel’s performance. Different versions of GEMM ($\text{Op}(\mathbf{X})$ is $\mathbf{X}$ or $\mathbf{X}^T$) require putting A and/or B into shared memory. This parameter of the auto-tuner is denoted by sh_{AB} . When

only (part of) A is in shared memory each thread per TB computes an entire column or part of a column of BC . When both A and B are in shared memory the computation can be splitted in terms of rows or columns of the resulting submatrix of C .

Submatrix layout in shared memory: This parameter determines the layout of each $N_{TBX} \times nb$ submatrix of the matrix A (referred as BA from now on) or $N_{TBY} \times nb$ submatrix of the matrix B (referred as BB from now on) in the shared memory, i.e., whether the copy of each block BA or BB in the shared memory is transposed or not. Since the shared memory is divided into banks and two or more simultaneous accesses to the same bank cause bank conflicts, transposing the layout in the shared memory may help reduce the possibility of bank conflicts, thus potentially improving the performance.

Amount of allocated shared memory: Two parameters, $offset_{BA}$ and $offset_{BB}$, relate to the actual allocation size of BA or BB in shared memory. When $N_{TBY} = 16$ and $nb = 16$, it requires 16×16 2D-array for BB in shared memory. Dependig upon the computation sometimes it is better to allocate some extra memory so that the threads avoid bank conflict while accessing operands from shared memory. It means allocating 16×17 array instead of 16×16 . So there is a offset of 1. It could be 0, 2 or 3 depending upon other parameters and the nature of computation. The auto-tuner handles this offset as a tunable parameter in internal optimization.

Prefetching into registers: As in CPU kernels, GPU kernels can benefit by prefetching into registers. For the access of matrices A and B , the auto-tuner inserts prefetch instruction for the data needed in next iteration and checks the effect. Insertion of prefetch instruction leads to usage of registers which might limit the parallelism of the whole code. The auto-tuner investigtes this with various combinations of prefetches: no prefetch, prefetch A only, prefetch B only, and prefetch both A and B , to finally pick the best combination.

Loop optimization techniques: Different state of the art loop optimization techniques such as strip mining and loop unrolling are incorporated in order to extract parallelism and achieve performance. Another interesting loop optimization technique, namely *circular loop skewing* was incorporated in the auto-tuner to deal with GPU global memory layout. Circular loop skewing is based upon a very simple idea of reordering the computation in inner loop. In the context of GPUs, inner loop are considered the data parallel tasks that make up a kernel. These tasks are scheduled by CUDA (controlling the outer loop) on the available multiprocessors and the order of scheduling sometimes is crucial for the performance. Circular loop skewing techniques are incorporated to explore benefits of modified scheduling. Their most important use is in removing performance deteriorations related to *partition camping* (described above).

Precision: The code generators also takes precision as a parameter.

The code generator takes all these parameters as input, and generates the kernel, the timing utilities, the header file, and the Makefile to build the kernel. The code generator first checks the validity of the input parameters before actually generating the files. By validity it means 1) the input parameters confirm to hardware constraints, e.g., the maximum number of threads per thread block $N_{TX} \times N_{TY} \leq 512$ in GTX 280, and 2) the input parameters are mutually compatible, e.g., $(N_{TBX} \times N_{TBY}) \% (N_{TX} \times N_{TY}) = 0$, i.e., the load of BA 's data into share memory can be evenly distributed among all the threads in a thread block, etc. By varying the input parameters, the auto-tuner can generate different versions of the kernel, and evaluate their performance, in order to identify the best one. Along the way the auto-tuner tries to optimize the code by using different optimization techniques such as prefetching, circular loop skewing, adjusting offset in shared memory allocation as described above. One way to implement auto-tuning is to generate a small number of variants for some matrices with typical sizes during installation time, and choose

Kls	Prec	N_{tbx}	N_{tby}	nb	N_{tx}	N_{ty}	sh_{AB}	$Trns$	$op(A)$	$op(B)$	$skewing$
K1	S/DP	32	8	4	8	4	B	No	N	T	No
K2	S/DP	64	16	4	16	4	B	No	N	T	No
K3	S/DP	128	16	8	16	8	B	No	N	T	No
K4	S/DP	256	16	16	16	16	B	No	N	T	No
K5	DP	32	32	8	8	8	AB	No	T	N	No
K6	DP	64	16	16	16	4	B	Yes	N	N	No
K7	DP	128	16	8	16	8	B	Yes	N	N	No
K8	SP	64	16	4	16	4	B	No	N	T	All
K9	SP	64	16	4	16	4	B	No	N	T	Selective

Table 4.1: Different kernel configurations.

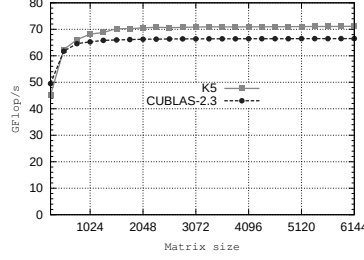


Figure 4.1: Performance of auto-tuned DGEMM kernel ($Op(A) = A^T$, $Op(B) = B$) on a GTX 280.

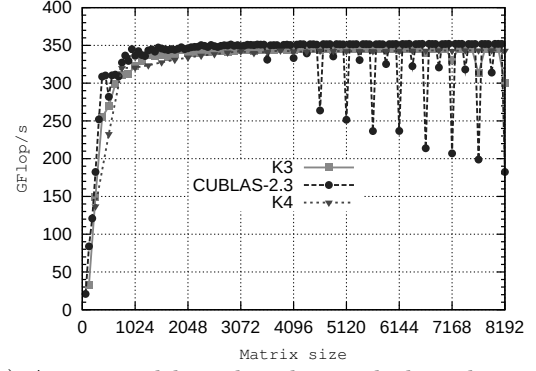
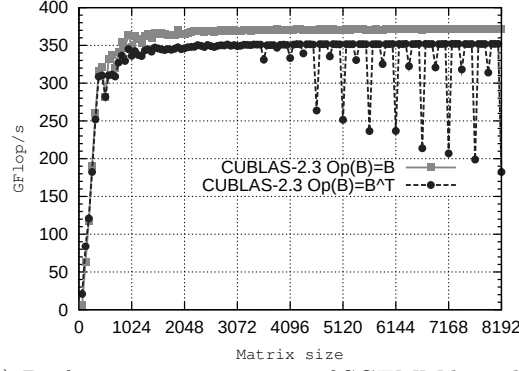
the best variant during run time, depending on the input matrix size and high level DLA algorithm.

4.2 Performance results

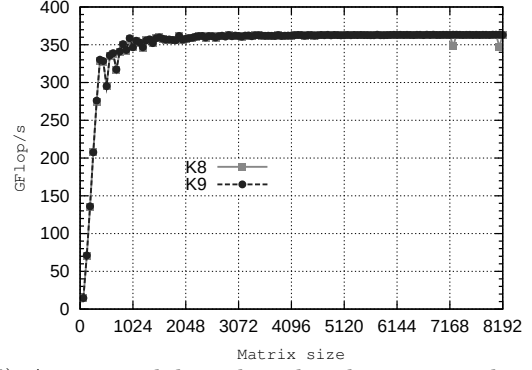
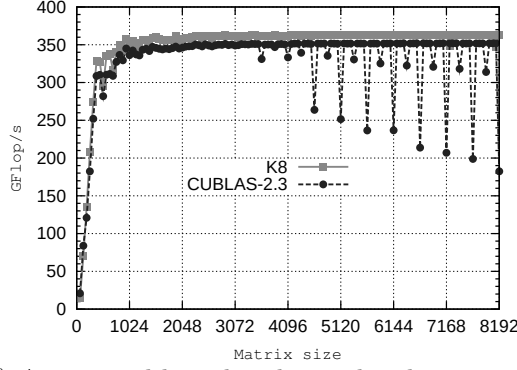
Table 4.1 gives the parameters of different xGEMM kernels used in this section. The table also provides parameters for all the kernels used in section 2.3.1. The $Trns$ parameter denotes if the kernel was implemented by taking tranpose operation in both side of the equation of the original operation, as:

$$C := \alpha A^T B^T + \beta C \quad \text{or} \quad C^T := \alpha B A + \beta C^T.$$

Figure 4.1 compares the performance of the xGEMM auto-tuner in double precision with the CUBLAS 2.3 for multiplying square matrices where $Op(A) = A^T$ and $Op(B) = B$. It can be seen that the performance of the auto-tuner is apparently 15% better than the CUBLAS 2.3 DGEMM. The fact that the two performances are



(a) Performance comparison of SGEMM kernel between $Op(B) = B$ and $Op(B) = B^T$ with parameter $Op(A) = A$



(c) Auto-tuned kernel with circular skewing in all dimension

(d) Auto-tuned kernel with selective circular skewing

Figure 4.2: Performance of the auto-tuned SGEMM ($Op(A) = A$, $Op(B) = B^T$) kernel for square matrices on a GTX 280.

so close is not surprising because the auto-tuned code and CUBLAS 2.3’s code are based on the same kernel, and this kernel was designed and tuned for current GPUs (and in particular the GTX 280), targeting high performance for large matrices.

The global memory layout of current GPUs presents challenges as well as opportunities for auto-tuners. As shown in Figure 4.2(a), CUBLAS-2.3 SGEMM has performance deteriorations for certain problem sizes when $Op(A) = A$ and $Op(B) = B^T$. Interestingly, when $Op(A) = A$ and $Op(B) = B$, the performance is very smooth. The reason for this is that GPU global memory is interleaved into a number of memory modules and the memory requests from all the concurrently running thread blocks may not be evenly distributed among the GPU memory modules. As a result the memory requests are sequentially processed and all the threads experience huge memory latency. This phenomenon is referred to as *partition camping* in NVIDIA terms. The auto-tuner found two kernels ($K3$, $K4$), as shown in Figure 4.2(b), that work significantly better in this situation. $K3$ and $K4$ work better because as partition size N_{TBX} is increased, the total number of accesses to global memory for matrix B’s data is correspondingly 1/2 and 1/4th compared to that for kernel $K2$ (besides, TLP is increased). Kernels $K3$ and $K4$ perform fair compared to CUBLAS-2.3 in any dimension, and remarkably well for the problem sizes where CUBLAS-2.3 has performance deteriorations. Interestingly, the auto-tuner was successful in finding a better kernel by applying circular loop skew optimization in kernel $K2$. The performance is shown in Figure 4.2(c). Note that there are no performance deteriorations and performance is better than CUBLAS-2.3 for all matrix sizes. However, this technique does not work in all cases and may have to be applied selectively. The performance of such kernel ($K9$) is shown in Figure 4.2(d).

Finally, in the area of DLA, it is very important to have high performance GEMMs on rectangular matrices, where one size is large, and the other is fixed within a certain block size (BS), e.g. $BS = 64, 128$, up to about 256 on current architectures. For example, in an LU factorization (with look-ahead) it requires two types of GEMM, namely one for multiplying matrices of size $N \times BS$ and $BS \times N - BS$, and another for

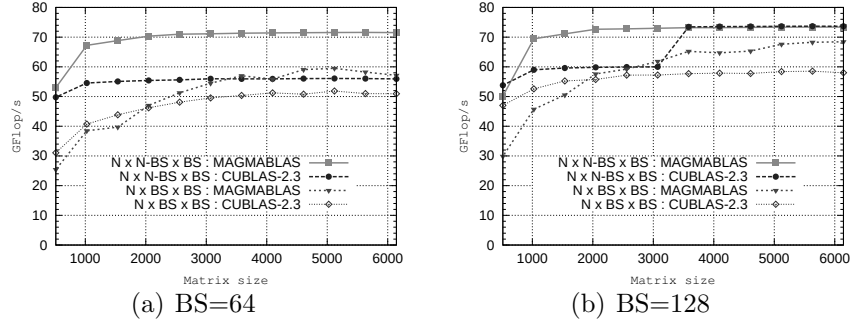


Figure 4.3: Performance comparison of the auto-tuned (solid line) *vs* CUBLAS 2.3 DGEMMs occurring in the block LU factorization (for block sizes $BS = 64$ on the left and 128 on the right) of a matrix of size 6144×6144 . The two kernels shown are for multiplying $N \times BS$ and $BS \times N - BS$ matrices (denoted by $N \times N - BS \times BS$), and $N \times BS$ and $BS \times BS$ matrices (denoted by $N \times BS \times BS$). K6 was used when $BS = 64$ and K7 was used when $BS = 128$.

multiplying $N \times BS$ and $BS \times BS$ matrices. This situation is illustrated on Figure 4.3, where the performances of the CUBLAS 2.3 *vs* auto-tuned DGEMMs occurring in the block LU factorization of a matrix of size 6144×6144 is compared. The graphs show that the auto-tuned code significantly outperforms (up to 27%) the DGEMM from CUBLAS 2.0.

The impacts of auto-tuned kernels of higher level DLA routines are remarkable. In MAGMA, some of the auto-tuned kernels are used for mixed precision iterative refinement solvers, tridiagonal reduction, and hessenberg reduction. To take advantage of the fact that GPU's single precision is currently of much higher performance than the double precision (theoretically $\approx 10\times$), MAGMA version 0.2 provides a second set of solvers, based on the mixed precision iterative refinement technique. Many auto-tuner kernels, e.g. DSYMV, DGEMV, DLANGE, SLAG2D, DLAG2S, DLACPY, DAXPY, were used as a building block for these iterative refinement techniques. The solvers are based again on correspondingly the LU, QR, and Cholesky factorizations, and are designed to solve linear problems in double precision accuracy but at a speed that is characteristic for the much faster single precision computations. The idea is to use single precision for the bulk of the computation,

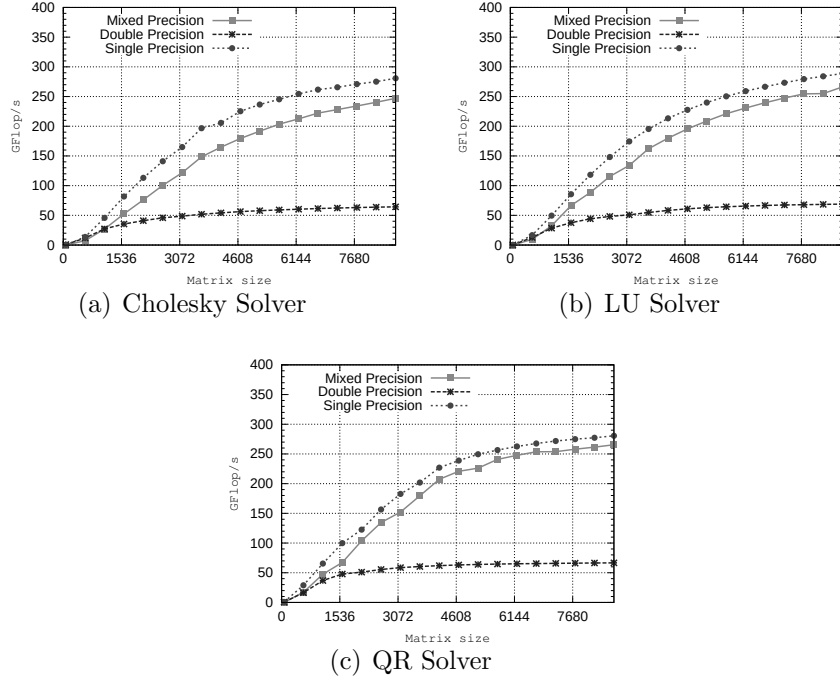
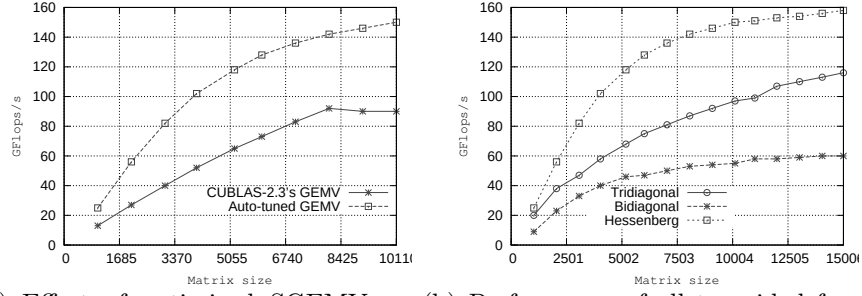


Figure 4.4: Solvers in GPU NVIDIA GTX 280.

namely the factorization step, and then use that factorization as a preconditioner in a simple iterative refinement process in double precision arithmetic. This often results in the desired high performance and high accuracy solvers. The performance of solvers with mixed precision iterative refinement is presented in Figure 4.4 with NRHS=1.

Figure 4.5(a) shows the effect of auto-tuned SGEMV kernel on hessenberg reduction with a comparison to CUBLAS SGEMV. The performance for all the three two-sided factorization with auto-tuned kernel is shown in Figure 4.5(b). The comparison with CUBLAS isn't provided here because for some of the routines, e.g. SSYMV, CUBLAS is very slow, 2 GFlops/s in CUBLAS vs 102 GFlops/s in the auto-tuned kernel.

The results on GPU BLAS auto-tuning support experiences and observations by others on “how sensitive the performance of GPU is to the formulations of your



(a) Effect of optimized SGEMV on the Hessenberg reduction. (b) Performance of all two-sided factorization

Figure 4.5: Two-sided factorization in single precision on GPU NVIDIA GTX 280.

kernel” (25) and that an enormous amount of well thought experimentation and benchmarking (26; 25) is needed in order to optimize performance.

Chapter 5

Tuning Dense Linear Algebra for Multicore Architecture: PLASMA

The development of programming models that enforce asynchronous, out of order scheduling of operations is the concept used as the basis for the definition of a scalable yet highly efficient software framework for computational linear algebra applications. In PLASMA, parallelism is no longer hidden inside Basic Linear Algebra Subprograms (BLAS) (3) but is brought to the fore to yield much better performance. The details of the tile algorithms is not presented here, only basic principles are addressed. The basic idea is to split the initial matrix of order N into $NT \times NT$ smaller square pieces of order NB , called *tiles*. Assuming that NB divides N , the equality $N = NT \times NB$ stands. The algorithms are then represented as a Directed Acyclic Graph (DAG) (28) where nodes represent tasks performed on tiles, either panel factorization or update of a block-column, and edges represent data dependencies among them. More details on tile algorithms can be found (27). PLASMA currently implements three one-sided (QR, LU, Cholesky) tile factorizations. The DAG of the Cholesky factorization is the least difficult to schedule since there is relatively little work required on the critical path. LU and QR factorizations have exactly the same dependency pattern between the nodes of the DAG, exhibiting much more severe

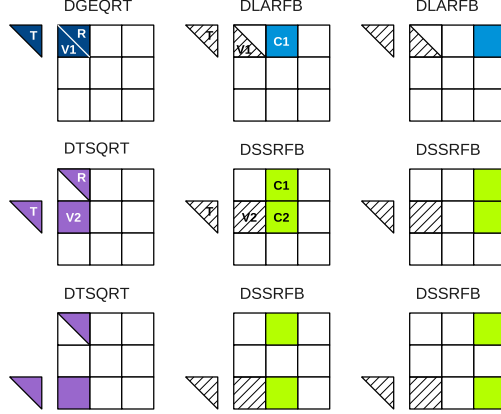


Figure 5.1: Panel factorization and corresponding updates.

scheduling and numerical (only for LU) constraints than the Cholesky factorization. Therefore, tuning the QR factorization is somehow representative of the work to be done for tuning the whole library. In the following, the QR factorization of square matrices in double precision is investigated. Note that the version (2.1) of PLASMA that have been studied is scheduled statically with a trade off between load balancing and data reuse.

Similarly to LAPACK which was built using a set of basic subroutines (BLAS), PLASMA QR factorization is built on top of four serial kernels. Each kernel indeed aims at being executed sequentially (by a single core) and corresponds to an operation performed on one or a few tiles. For instance, assuming a 3×3 tile matrix, Figure 5.1 represents the first panel factorization (DGEQRT and DTSQRT serial kernels (27)) and its corresponding updates (DLARFB and DSSRFB serial kernels (27)). The corresponding DAG (assuming this time that the matrix is split in 5×5 tiles) is presented in Figure 5.2.

5.1 Tunable parameters

The shape of the DAG depends on the number of tiles ($NT \times NT$). For a given matrix of order N , choosing the tile size NB is equivalent to choose the number of

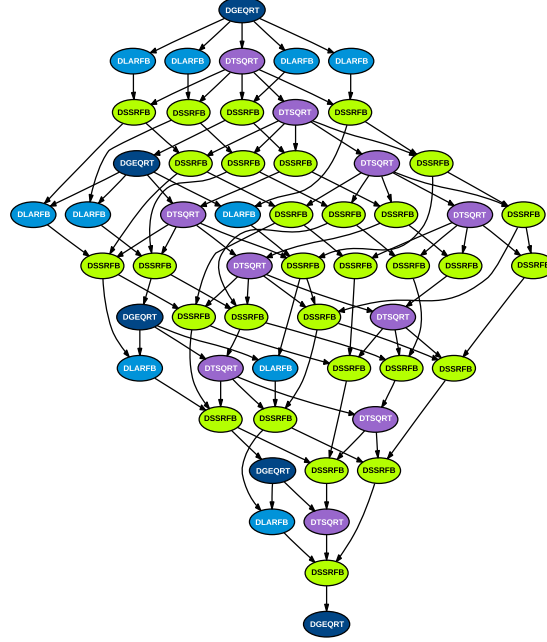


Figure 5.2: DAG of the tile QR factorization. The matrix is split in 5×5 tiles.

tiles (since $N = NB \times NT$). Therefore, NB is a first tunable parameter. A small value of NB induces a large number of tasks in the DAG and subsequently enables the parallel processing of many tasks. On the other hand, the serial kernel applied to the tiles needs a large enough granularity in order to achieve a decent performance. The choice of NB thus trades off the degree of parallelism with the efficiency of the serial kernels applied to the tiles. There is a second tunable parameter, called inner block size (IB). It trades off memory load with extra-flops due to redundant calculations. If no inner blocking occurs, the resulting extra-flops overhead may represent 25% of the whole QR factorization. More details is available (27). The general objective of the paper is to address the following problem.

Problem 5.1.1. *Given a matrix size N and a number of cores $ncores$, which tile size and internal blocking size (NB, IB) do maximize the performance of the tile QR factorization?*

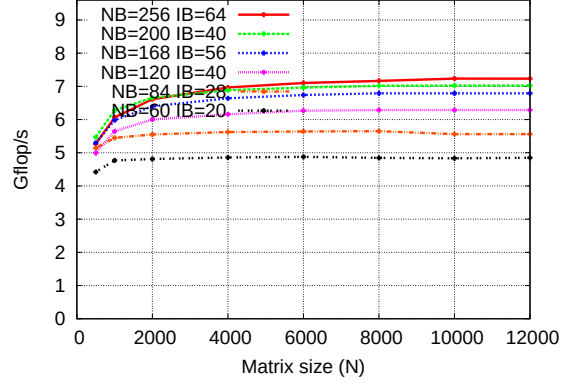


Figure 5.3: Performance of the sequential PLASMA QR factorization on an Intel Core Tigerton machine.

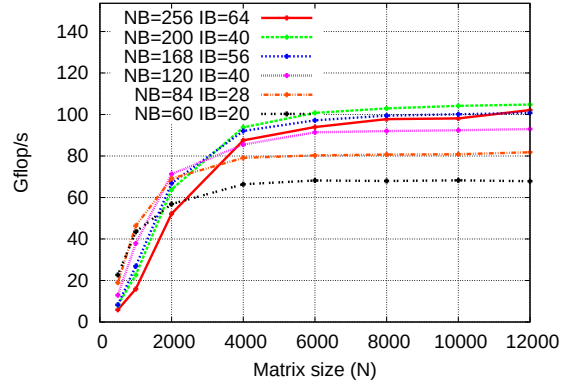


Figure 5.4: Performance of the PLASMA QR factorization on an Intel Core Tigerton machine using 16 cores.

The decision should be instantaneous when the user requests to factorize a matrix. So we need to tune the library during installation time.

In a sequential execution of PLASMA, parallelism cannot be exploited. In that case, PLASMA's performance is only related to the performance of the serial kernel which increases with the tile size. Figure 5.3 illustrates this property on an Intel Core Tigerton machine that will be described in details in Section 5.4.

In a parallel execution of PLASMA, the optimum tile size depends on the matrix size as shown on a 16 cores execution in Figure 5.4. Indeed, if the matrix is small, it needs to be cut in even smaller pieces to feed all the 16 cores even if this induces

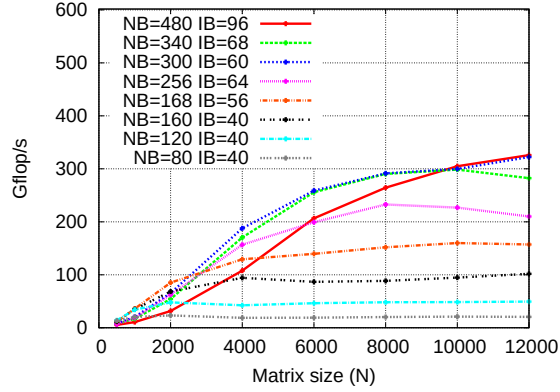


Figure 5.5: Performance of the PLASMA QR factorization on an IBM Power6 machine using 32 cores.

that the serial kernels individually achieve a lower performance. When the matrix size increases, all the cores may evenly share the work using a larger tile size and thus achieving a higher performance. In a nutshell, the optimum tile size both depends on the number of cores and the matrix size, and its choice is critical for performance. Figure 5.5 shows that the impact is even stronger on a 32 cores IBM Power6 machine, also described in details in Section 5.4. The (NB,IB) choice equal to (80,40) is optimum on a matrix of order 500 but leads to a performance which is only 6.3% of the optimum performance (20.6 Gflop/s against 325.9 Gflop/s) on a matrix of order 12,000.

5.2 Motivation for an empirical approach

In literature, the two main classes of tuning methods are the model-driven and empirical approaches. Previously it has been mentioned that DLA algorithms are difficult to model on CPU-based architectures, and in particular on multicore architectures. Let us illustrate this claim now. Before coming back to the tile QR factorization, let us temporarily consider a simpler tile algorithm: the tile matrix multiplication: $C \leftarrow C + A \times B$. Matrices A , B and C are split into tiles a_{ij} , b_{ij} and c_{ij} , respectively. The tile matrix multiplication is then the standard nested

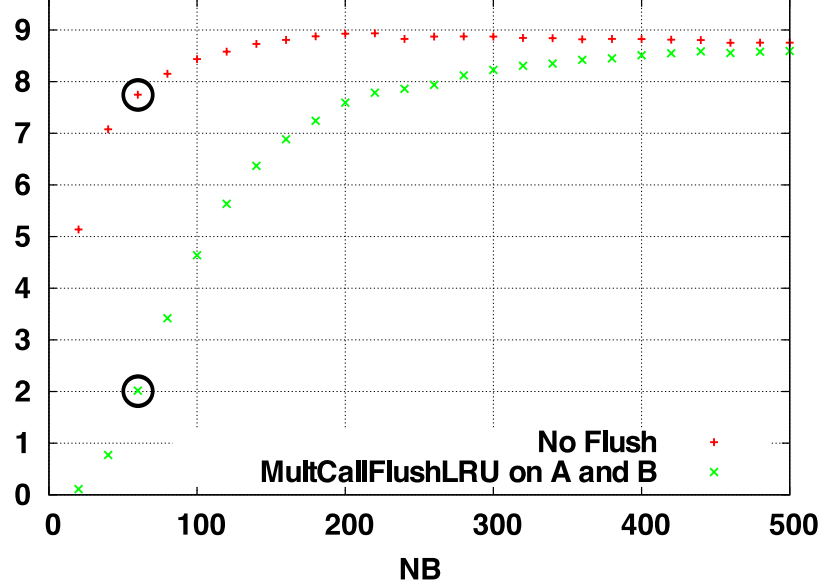


Figure 5.6: Performance (in Gflop/s) of a sequential matrix multiplication $c \leftarrow c + a \times b$ on the Intel Core Tigerton machine as a standard call to the vendor BLAS library. With the *No Flush* strategy, data (a , b and c) is *not* flushed from the cache. With the *MultCallFlushLRU* strategy (29), a and b (but not c) are flushed from the cache. The values corresponding to a matrix order $NB = 60$ are circled.

loop on sub-arrays i , j and k whose single instruction is a DGEMM BLAS call on the corresponding tiles: $c_{ij} \leftarrow c_{ij} + a_{ik} \times b_{kj}$. Given the simplicity of this algorithm (simple DAG, only one kernel, ...) one may expect that extrapolating the performance of the whole tile algorithm $C \leftarrow C + A \times B$ from the performance of the BLAS kernel $c_{ij} \leftarrow c_{ij} + a_{ik} \times b_{kj}$ is trivial.

However, the first difficulty is to correctly model how data are accessed during the execution of the tile algorithms. Indeed, before performing the BLAS call, some tiles may be in cache while others are partially or fully out of cache. Figure 5.6 presents the impact of the initial state of the tiles on the performance of a sequential matrix multiplication $c \leftarrow c + a \times b$ on the Intel Core Tigerton machine as a DGEMM call to the vendor BLAS library. In the *No Flush* strategy, all the tiles are initially in cache (if they can fit). On the other hand, in the *MultCallFlushLRU* (29) strategy, a and b (but not c) are flushed from the cache between two successive calls. To achieve accurate timing, the DGEMM kernel for each matrix order (NB) is called several

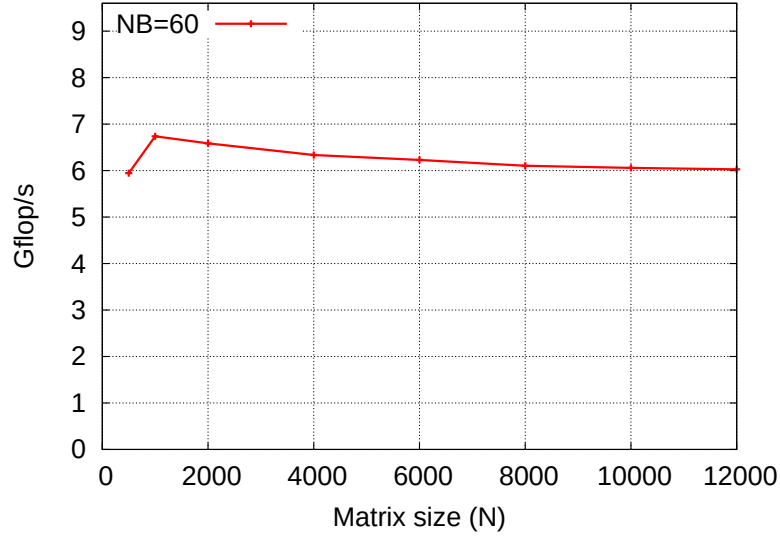


Figure 5.7: Performance (in Gflop/s) of the tile matrix multiplication on the Intel Core Tigerton machine using 1 core. The tile size is $NB = 60$.

times (50). The 50 calls are timed all at once; the average value finally computed is then more accurate than in the case of timing a single call (29). To simulate the case where data is not flushed, all 50 executions are performed on the same data (29). To simulate the case where a and b are flushed, two large arrays A and B are allocated, and the pointers a and b are moved along these arrays between two successive calls. This self-flushing strategy was introduced in (29). Figure 5.6 shows that the impact of the initial state is very important. For instance, for a tile of order $NB = 60$, the performance is four times higher (8 Gflop/s against 2 Gflop/s) in the *No Flush* case. In practice, none of these cases is a correct model for the kernel, since the sequential tile multiplication based on a tile size $NB = 60$ is neither 8 nor 2 Gflop/s but 6 Gflop/s as shown in Figure 5.7.

One may argue that the model could be improved to enable a better extrapolation. This is true. But the purpose of this experiment showed that modeling tile algorithms on CPU-based architectures is not trivial, even in the sequential case and even in the case of a simple algorithm such as the matrix multiplication. Complementary experiments showed (not presented explicitly here) that parallel

execution performance is even more difficult to forecast. For instance, frequent concurrent accesses to the memory bus can slow down the memory controller (as observed for small tile sizes on large matrices in Figure 5.5). The behavior of shared caches is also difficult to anticipate. On top of that, other algorithmic factors would add up to this complexity in the case of a more complex operation such as a one-sided factorization. For instance, load balancing issues and scheduling strategies must be taken into account when modeling a tile QR factorization.

As a consequence, it is decided to base the tuning approach on an extensive empirical search coupled with only few but strongly reliable properties to prune that search space.

5.3 Outline of the method

Given the above considerations, a method based on at-scale benchmarking of the tile QR factorization seems to be very promising. However, an *exhaustive search* is cumbersome since the search space is huge. As noted in (9), there are more than 1000 possible combinations for (NB,IB) even if we constrain NB to be an even integer between 40 and 512 and if we constrain IB to divide NB. For instance, exploring this search space on a matrix of order $N = 10,000$ with 8 cores on the Intel Core Tigerton machine would take several days. Hence the need to prune the search space. In Section 5.5, it is shown that preliminary pruning can be performed thanks to considerations on the most compute-intensive serial kernel and several heuristics for performing that preliminary pruning is presented. Section 5.6 then shows that further pruning can be done based on the results of previous at-scale experiments. Since the adopted approach is highly empirical, before that, let us present the set of machines that is used to conduct the experiments.

5.4 Experimental environments

The Top 500 supercomputers list of November 2009 (30) is dominated by the Intel EM64T processors family (79.2%), followed by IBM Power (10.4%) and AMD x86_64 (8.4%). The experiments are conducted on a distribution of machines that approximately follows these hardware trends with a bias to shared memory multicore machines. Below is the list of machines used in our experiments conducted with PLASMA 2.1.

Intel Core Tigerton. This 16 cores machine is a quad-socket quad-core Xeon E7340 (codename Tigerton) processor, an Intel Core micro-architecture. The processor operates at 2.39 GHz. The theoretical peak is equal to 9.6 Gflop/s per core or 153.2 Gflop/s for the whole node, composed of 16 cores. There are two levels of cache. The level-1 cache, local to the core, is divided into 32 kB of instruction cache and 32 kB of data cache. Each quad-core processor being actually composed of two dual-core Core2 architectures, the level-2 cache has 2×4 MB per socket (each dual-core shares 4 MB). The effective bus speed is 1066 MHz per socket leading to a bandwidth of 8.5 GB/s (per socket). The machine is running Linux 2.6.30 and provides Intel Compilers 11.0 together with the MKL 10.1 vendor library.

Intel Core Clovertown. This 8 cores server is another machine based on an Intel Core micro-architecture. The machine is composed of two quad-core Xeon X5355 (codename Clovertown) processors, operating at 2.66 GHz. The theoretical peak is equal to 10.64 Gflop/s per core and thus 85.12 Gflop/s for the whole machine. The machine comes with Linux 2.6.28, Intel Compilers 11.0 and MKL 10.1.

Intel Core Yorkfield. This 4 cores desktop is also based on an Intel Core micro-architecture. The machine is composed of one Core 2 Quad Q9300 (codename Yorkfield) processor, operating at 2.5 GHz. The theoretical peak is equal to 10.0 Gflop/s per core and thus 40.00 Gflop/s for the whole machine with a shared 3 MB level-2 cache per core pair. Each core has 64 KB of level-1 cache. The machine comes with Linux 2.6.33, Intel Compilers 11.0 and MKL 10.1.

Intel Core Conroe. This 2 cores desktop is based on an Intel Core micro-architecture too. The machine is composed of one Core 2 Duo E6550 (codename Conroe) processors, operating at 2.33 GHz. The theoretical peak is equal to 9.32 Gflop/s per core and thus 18.64 Gflop/s for the whole machine with a shared 4 MB level-2 cache. Each core has 128 KB of level-1 cache. The machine comes with Linux 2.6.30.3, Intel Compilers 11.1 and MKL 10.2.

Intel Nehalem. This 8 cores machine is based on an Intel Nehalem micro-architecture. Instead of having one bank of memory for all processors as in the case of the Intel Core's architecture, each Nehalem processor has its own memory. Nehalem is thus a Non Uniform Memory Access (NUMA) architecture. Our machine is a dual-socket quad-core Xeon X5570 (codename Gainestown) running at 2.93GHz and up to 3.33 GHz in certain conditions (Intel Turbo Boost technology). The Turbo Boost was activated during our experiments, allowing for a theoretical peak of 13.32 Gflop/s per core, i.e., 106.56 Gflop/s for the machine. Each socket has 8 MB of level-3 cache (that was missing from most Intel Core-based microprocessors such as Tigerton and Clovertown). Each core has 32 KB of level-1 instruction cache and 32 KB of level-1 data cache, as well as 256 KB of level-2 cache. The machine comes with Linux 2.6.28, Intel Compilers 11.1 and MKL 10.2.

AMD Istanbul. This 48 cores machine is composed of eight hexa-core Opteron 8439 SE (codename Istanbul) processors running at 2.8 GHz. Each core has a theoretical peak of 11.2 Gflop/s and the whole machine 537.6 Gflop/s. Like the IBM Nehalem, the Istanbul micro-architecture is a NUMA architecture. Each socket has 6 MB of level-3 cache. Each processor has a 512 KB level-2 cache and a 128 KB level-1 cache. After having benchmarked the AMD ACML and Intel MKL BLAS libraries, MKL (10.2) is selected as it appeared to be slightly faster in our experimental context. Linux 2.6.32 and Intel Compilers 11.1 were also used.

IBM Power6. This 32 cores machine is composed of sixteen dual-core IBM Power6 processors running at 4.7 GHz. The theoretical peak is equal to 18.8 Gflop/s per core and 601.6 Gflop/s for the whole symmetric multiprocessing (SMP) node.

There are three levels of cache. The level-1 cache, local to the core, can contain 64 kB of data and 64 kB of instructions; the level-2 cache is composed of 4 MB per core, accessible by the other core; and the level-3 cache is composed of 32 MB common to both cores of a processor with one controller per core (80 GB/s). The memory bus (75 GB/s) is shared by the 32 cores of the node. The machine runs AIX 5.3 and provides the xlf 12.1 and xlc 10.1 compilers together with the Engineering Scientific Subroutine Library (ESSL) (6) 4.3 vendor library.

Table 5.1: Elapsed time (hh:mm:ss) for Step 1 and Step 2

Machine		Step 1	Step 2		
Architecture	# cores		Heuristic	PS	PSPAYG
Conroe	2	00:24:33	0	14:46:37	03:05:41
			1	09:01:08	00:01:58
			2	07:30:53	00:34:47
Yorkfield	4	00:20:57	0	17:40:00	04:48:13
			1	09:30:30	00:05:10
			2	08:01:05	02:58:37
Clovertown	8	00:21:44	0	20:08:43	02:56:25
			1	11:06:18	00:13:09
			2	08:52:24	01:10:53
Nehalem	8	00:16:29	0	06:20:16	01:51:30
			1	06:20:16	01:51:30
			2	06:20:16	01:51:30
Tigerton	16	00:34:18	0	23:29:35	03:15:41
			1	12:22:06	00:08:57
			2	09:54:59	01:01:06
Istanbul	48	00:24:23	0	21:09:27	02:53:38
			1	12:25:30	00:11:01
			2	10:04:46	00:54:51
Power6	32	00:15:23	0	03:06:05	00:25:07
			1	03:06:05	00:25:07
			2	03:06:05	00:25:07

5.5 Step 1: Benchmarking the most compute-intensive serial kernels

As explained before, the tile QR factorization consists of four serial kernels. However, the number of calls to DSSRFB is proportional to NT^3 while the number of calls to the other kernels is only proportional to NT (DGEQRT) or to NT^2 (DTSQRT and DLARFB). Even on small DAGS (see Figure 5.2), calls to DSSRFB are predominant. Therefore, the performance of this compute-intensive kernel is crucial. DSSRFB's performance also depends on (NB, IB) . It is thus natural to pre-select (NB, IB) pairs that allow a good performance of DSSRFB before doing at-scale experiments. The practical advantage is that a kernel is applied at the granularity of a tile, which is assumed to be bounded by 512 ($NB \leq 512$). Consequently, preliminary benchmarking this serial kernel can be done exhaustively in a reasonable time. This is *step 1*. To achieve accurate timing, the guidelines of (29) as presented in Section 5.2 is followed. In particular, DSSRFB is called 50 times for each (NB, IB) pair. Both *No Flush* and *MultCallFlushLRU* strategies are implemented. In this report, the results related to the *No Flush* approach is presented. The reason is that it runs faster and provides satisfactory results as it will be shown. A comparison of both approaches is left as future work. Column “Step 1” of Table 5.1 shows that the total elapsed time for step 1 is acceptable on all the considered architecture (between 16 and 35 minutes).

Figure 5.8 shows the resulting set of empirical data collected after step 1 on the Intel Core Tigerton machine. Contrary to NB which trades off parallelism for kernel performance, IB only affects the kernel performance. The following property can be deduced. [theorem]Property [theorem]Problem

Property 5.5.1. *For a given NB value, we can safely pre-select the value of IB that maximizes the kernel performance.*

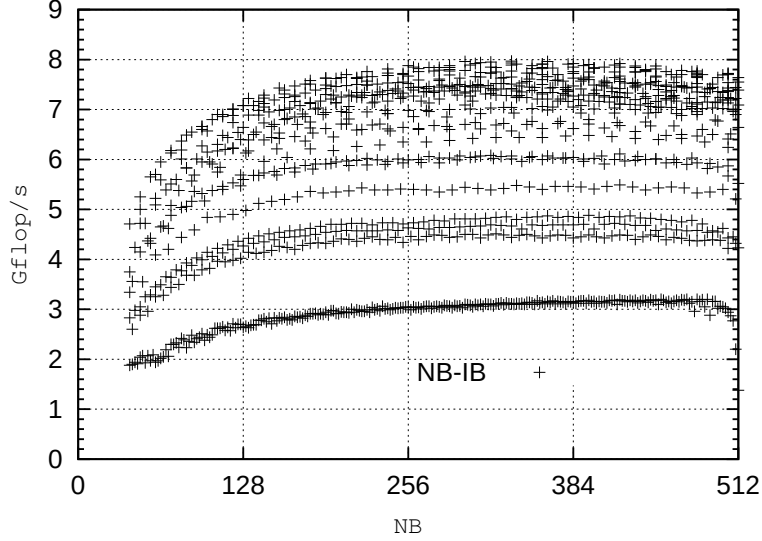


Figure 5.8: Step 1-a: Performance of the DSSRFB serial kernel depending on the (NB-IB) parameters. Note that two (NB-IB) pairs with a common NB value have the same abscisse.

Figure 5.9 shows how Property 5.5.1 can be used to perform a first pre-selection of (NB-IB) pairs that will be tested at scale. One can further more claim the following assumption.

Property 5.5.2. *A search performed with a well chosen subset of a limited number – say 8 - of (NB,IB) pairs is enough to consistently achieve a maximum performance for any matrix size N or number of cores $ncores$.*

The process consisting in choosing these limited number of pairs is termed as *pre-selection* (PS). To validate Property 5.5.2, 8 points from the convex hull of Figure 5.9 were chosen manually. Then the maximum performance (PS) obtained with one of these pre-selected points on at-scale executions was compared to an exhaustive search (ES). As illustrated in Figure 5.10, PS performance is almost superimposed with ES. In the above experiment, the pre-selection was done manually. If a subset of the convex hull includes (quasi-)optimum pairs, a fortiori, the convex hull will also include (quasi-)optimum pairs. In the following, a search on the whole convex hull will thus been considered as an exhaustive search. Given an empirical set such as the

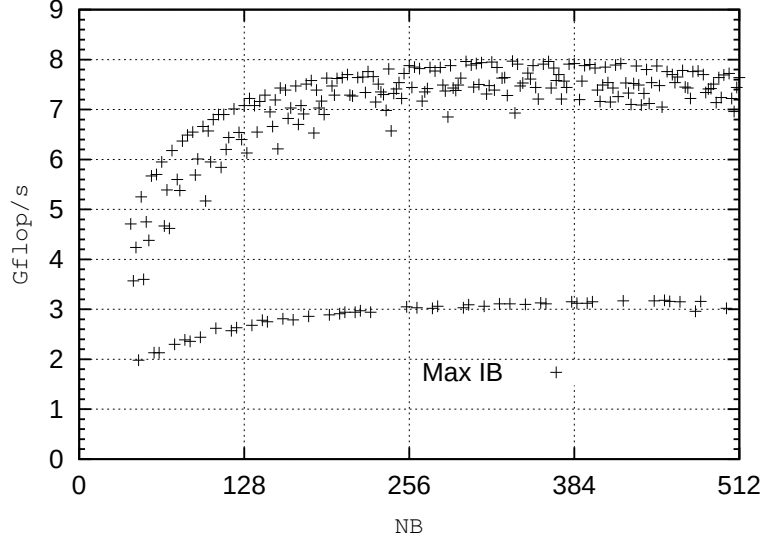


Figure 5.9: Step 1-b: Picking up the optimum IB for each NB.

one from Figure 5.8 the convex hull is automatically extracted. The resulting data set is shown in Figure 5.11. The data set of the points constituting the convex hull can be used to perform at-scale experiments in the second step. As a consequence, the extraction of the convex hull can be considered as a heuristic (Heuristic 0) to perform the pre-selection (PS). But, in general, this approach may provide too many pairs. Therefore, it is necessary to prune further the data set. To do so, two simple heuristics are introduced. Since NB trades off kernel efficiency with parallelism, it is natural to select the points with a high steepness (or more accurately a point after a segment with a high steepness). Heuristic 1 finds the 8 points with maximum steepness among the points of the convex hull. The drawback is that all these points tend to be located in the same area as shown in Figure 5.12. To correct this deficiency, a variant of that heuristic is formed which is called as Heuristic 2. Heuristic 2 consists of dividing the x-axis into iso-segments and picking up the point of maximum steepness on each of these segments. Figure 5.13 shows the resulting pre-selection.

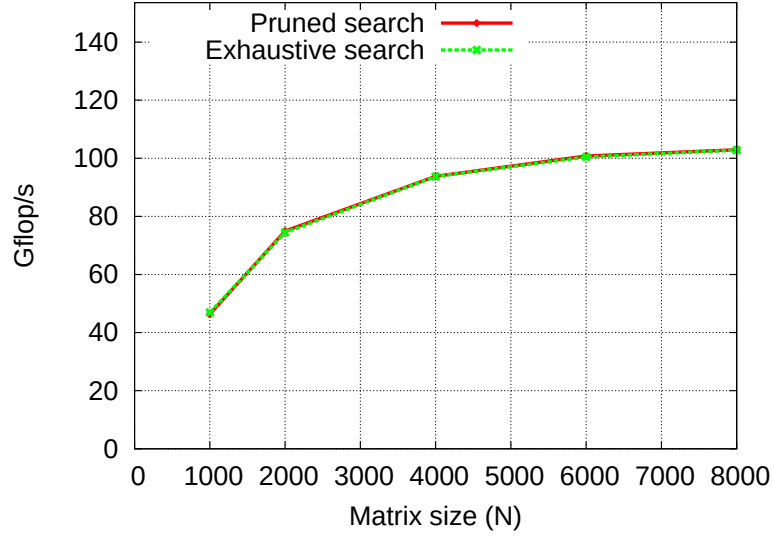


Figure 5.10: Performance of the pre-selected search (PS) against the exhaustive search (ES) on the Intel Core Tigerton machine. The graphs are almost superimposed.

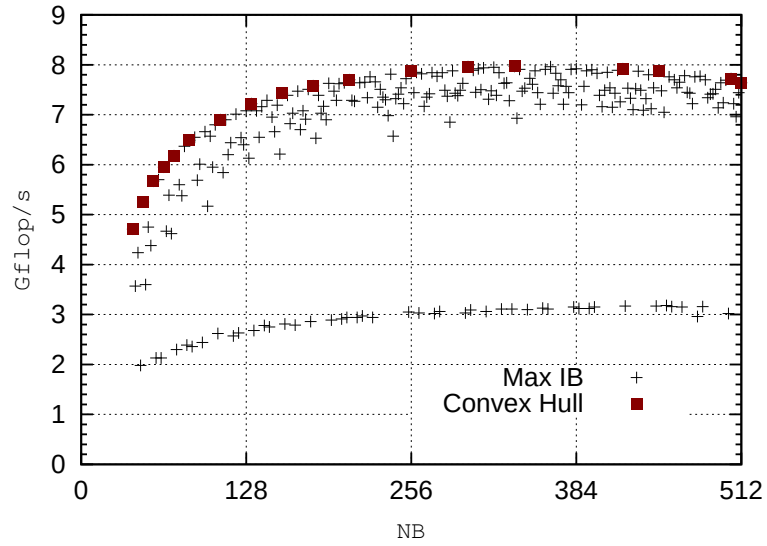


Figure 5.11: Step 1-c: Extracting the convex hull (Heuristic 0)

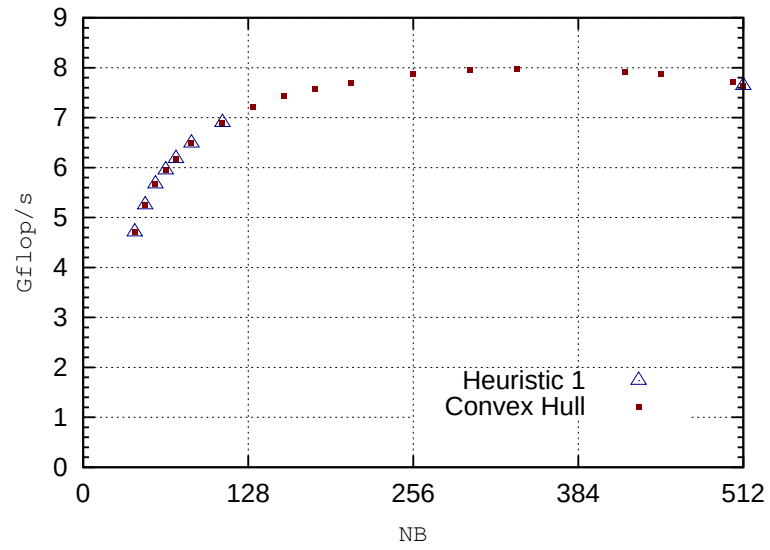


Figure 5.12: Step 2 - Heuristic 1: maximum steepness

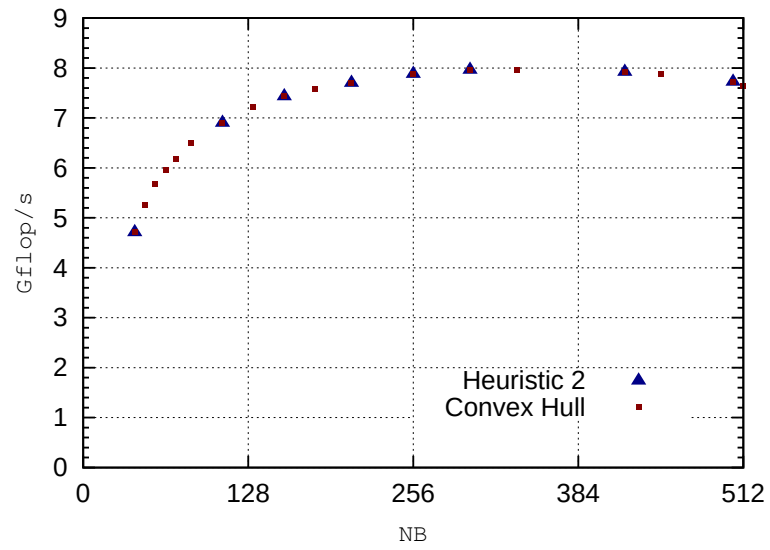


Figure 5.13: Step 2 - Heuristic 2: even distribution

5.6 Step 2: Benchmarking at-scale executions

This step consists of running at-scale PLASMA QR factorizations. The (NB,IB) pairs tested correspond to the ones pre-selected at step 1. From now on the convex hull will be considered as a reference. In other words, exploring the pre-selected set of pairs obtained through Heuristic 0 (H0-PS) is equivalent to performing an Exhaustive Search (ES). Therefore, to assess the accuracy and efficiency of the devised methods and heuristics, everything will be compared to ES.

5.6.1 Discretization

In this step, it is not feasible to explore all the N and $ncores$ values. The space has thus to be discretized. It's decided to benchmark all the power of two cores (1, 2, 4, 8, ...) plus the maximum number of cores in case it is not a power of two such as on the AMD Istanbul machine. The motivation comes from empirical observation. Indeed, Figures 5.14, 5.15, 5.16 and 5.17 show that the optimum (NB,IB) can be finely interpolated with such a distribution. The space on N is discretized more regularly because the choice of the optimum pair is much more sensible to that dimension (see figures 5.4 and 5.5). The following set of values for N was benchmarked {500, 1000, 2000, 4000, 6000, 8000, 10000}*. Each run is performed 6 times to attenuate potential perturbations. When the user requests the factorization of parameters that have not been tuned (for instance $N=1800$ and $ncores=5$) the parameters found for the closest configuration are chosen (the ones of $N=2000$ and $ncores=4$ in that case).

5.6.2 Impact of the heuristics on the time required for tuning

Column PS (pre-selected) in Table 5.1 shows the impact of the heuristics on the time required for benchmarking step 2. Clearly Heuristic 0 induces a very long step 2 (up

*Except on the IBM Power6 machine where $N=10000$ was not benchmarked.

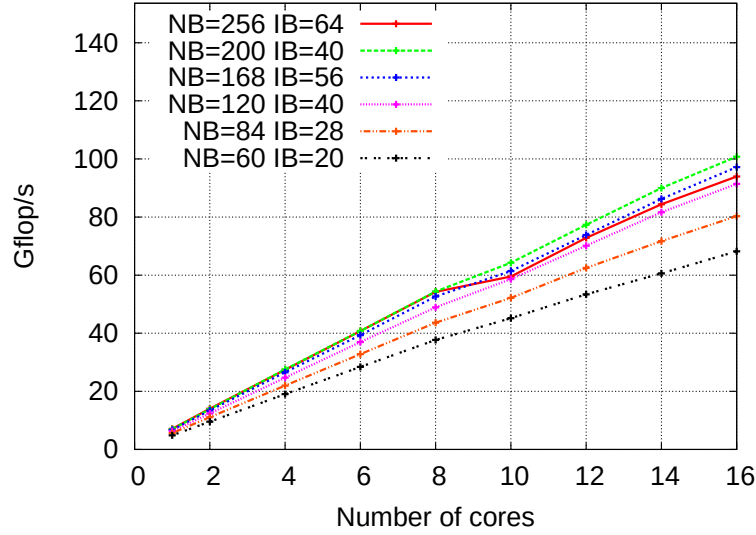


Figure 5.14: Intel Core Tigerton machine - $N = 6000$.

to 1 day). Heuristic 1 and 2 induce a lower time for step 2 (about 10 hours) but that may be not acceptable for many users.

5.6.3 Prune As You Go (PSPAYG)

To further reduced the time taken in step 2, a complementary pruning on the fly is proposed. Indeed, Figures 5.4 and 5.5 show the following property.

Property 5.6.1. *Let us denote by $P(NB_1, N)$ and $P(NB_2, N)$ the performances obtained on a matrix of order N with tile sizes NB_1 and NB_2 , respectively. If $P(NB_1, N) > P(NB_2, N)$ and $NB_1 > NB_2$, then $P(NB_1, N') > P(NB_2, N')$ for any $N' > N$.*

This property is used to prune as we go. Step 2 was performed in increasing order of N . After having benchmarked the current set of (NB, IB) pairs on a matrix of order N , all the couples (NB_1, NB_2) that satisfy Property 5.6.1 are identified and removed from the current subset the (NB, IB) pair in which NB_2 is involved. Indeed, according to Property 5.6.1, it would lead to a lower performance than NB_1 on larger values of N which are going to be explored next. This pruning strategy is denoted

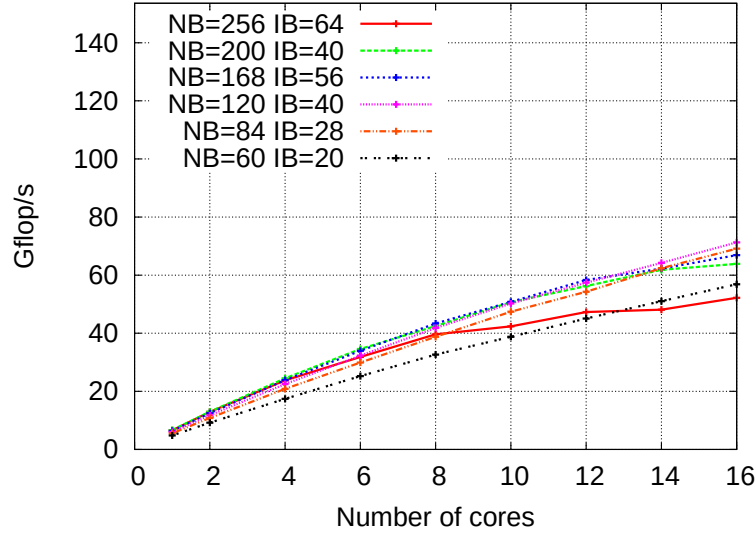


Figure 5.15: Intel Core Tigerton machine - $N = 2000$.

by “PSPAYG” (pre-selection and prune as you go). Column PSPAYG in Table 5.1 shows that the time for step 2 is dramatically improved with this technique. Indeed, the number of pairs to explore decreases when N increases, that is, when benchmark is costly. For heuristic 2 (values in bold in Table 5.1), the time required for step 2 is reduced by a factor greater than 10 in two cases (Intel Core Conroe and AMD Istanbul machines).

5.6.4 Accuracy of the tuning

Table 5.2 shows that heuristic 2 coupled with the PSPAYG approach is very efficient since it achieves a high proportion of the performance that would be obtained with an exhaustive search (values in bold). The worst case occurs on the Intel Core Tigerton machine, with an average relative performance of 97.9%. However, even on that platform, the optimum (NB,IB) pair was found in seven cases out of sixteen tests.

The last two columns allow to specifically assess the impact of the “prune as you go” method since they compare the average performance obtained with PSPAYG (where pairs can be discarded during step 2 according to Property 5.6.1) compared to

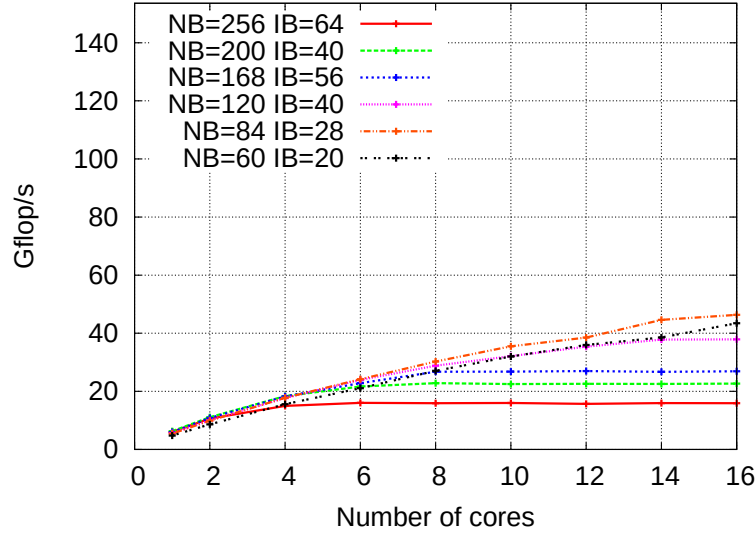


Figure 5.16: Intel Core Tigerton machine - $N = 1000$.

PS (where no pair is discarded during step 2). The result is clear: pruning during step 2 according to Property 5.6.1 does not hurt performance, showing that Property 5.6.1 is strongly reliable.

More detailed performance results is presented now to explain more accurately how the synthetic results of Table 5.2 were obtained. The whole mechanism will be discussed with performance results of the AMD Istanbul machine (tables 5.3, 5.4, 5.5 and 5.6). To assess the efficiency of the different methods presented in the paper, 8 to 16 tests on each machine have been performed. Each test is an evaluation of the method for a given number of cores $ncores$ and a matrix size N . On the AMD Istanbul machine, the 16 possible combinations of $N = 2000, 2700, 4200$ or 6000 and $ncores = 4, 7, 40$ or 48 have been tested. An exhaustive search (ES) is first performed for all these 16 combinations to be used as a reference (Table 5.3). Then it is checked which (NB,IB) would have been chosen by the autotuner depending on the method it is built on (tables 5.4, 5.5 and 5.6).

The results obtained for Heuristic 2 will be explained more (Figure 5.6) since it is the heuristic that is planned to set as a default in PLASMA. The first four rows show results related to experimental conditions in which both the matrix order and the

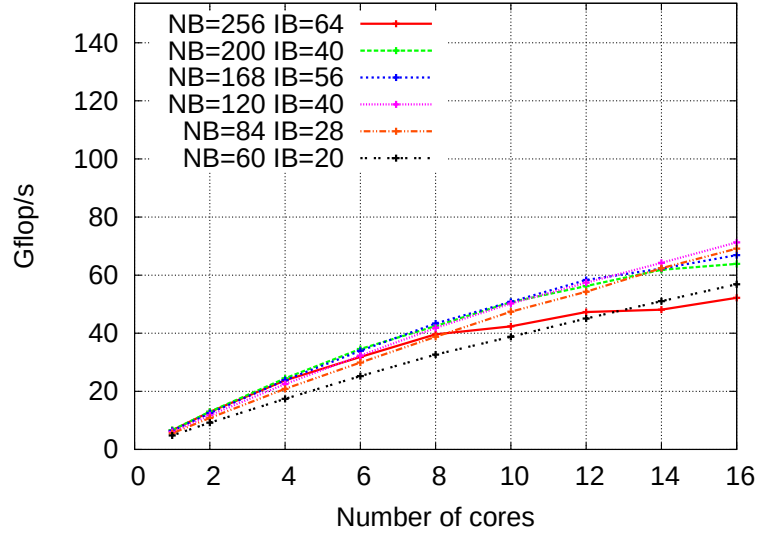


Figure 5.17: IBM Power6 machine - $N = 2000$.

number of cores are part of the values that were explicitly benchmarked during the tuning process ($N=2000$ or 6000 and $ncores=4$ or 48). No interpolation is needed. In three cases, the optimum configuration is found both by PS and PSPAYG. In the case where it was not found ($N=6000$ and $ncores=4$) the optimum configuration was actually not part of the initial pre-selected points by Heuristic 2 ($Y=0$). The four next rows ($N=2700$ or 4200 and $ncores=4$ or 48) require to interpolate the matrix order (but not the number of cores). For $N=2700$, the selection is based on the benchmarking realized on $N_0=2000$ while $N_0=4000$ is chosen when $N = 4200$. The achieved performance is not ideal since it is 8% lower than the exhaustive search. As expected, the interpolation on $ncores$ is much less critical (four next rows). This observation confirms the validity of a discretization coarser on the $ncores$ dimension. Finally (last four rows), the quality of the tuning for the interpolation in both dimensions is comparable to the one related to the interpolation on N .

Table 5.2: Average performance achieved with a “pre-selection” (PS) method or a “pre-selection and prune as you go” (PSPAYG) method, based on different heuristics (H) applied at step 1. The performance is presented as a proportion of the exhaustive search (ES) or of the prunes search (PS). The column “optimum” indicates the number of times the optimum combination (with respect to the reference method) was found among the number of tests performed.

Machine	H	$\frac{PS}{ES}(\%)$		$\frac{PSPAYG}{ES}(\%)$		$\frac{PSPAYG}{PS}(\%)$	
		avg optimum		avg optimum		avg optimum	
Conroe	0	99.67	6/8	99.67	6/8	100	8/8
	1	95.28	0/8	95.28	0/8	100	8/8
	2	99.54	5/8	99.54	5/8	100	8/8
Yorkfield	0	98.63	6/12	98.63	6/12	100	12/12
	1	91.53	0/12	91.59	0/12	100.07	10/12
	2	98.63	6/12	98.63	6/12	100	12/12
Clovertown	0	98.59	8/16	98.35	7/16	99.76	15/16
	1	91.83	0/16	91.83	0/16	100	16/16
	2	98.49	9/16	98.25	8/16	99.76	15/16
Nehalem	0	98.6	8/16	98.9	8/16	100.33	16/16
	1	98.6	8/16	98.9	8/16	100.33	16/16
	2	98.6	8/16	98.9	8/16	100.33	16/16
Tigerton	0	97.36	8/16	97.54	5/16	100.21	12/16
	1	91.61	0/16	91.61	0/16	100	16/16
	2	97.51	8/16	97.79	7/16	100.31	15/16
Istanbul	0	97.17	7/16	97.17	7/16	100	16/16
	1	94.12	2/16	94.12	2/16	100	16/16
	2	97.23	7/16	97.1	7/16	99.87	15/16
Power 6	0	100	16/16	100	16/16	100	16/16
	1	100	16/16	100	16/16	100	16/16
	2	100	16/16	100	16/16	100	16/16

Table 5.3: Performance of ES on the AMD Istanbul Machine

N ncore		Perf (Gflop/s)	NB	IB
2000	4	24.81	168	28
2000	48	140.1	96	32
6000	4	30.36	504	56
6000	48	272.55	168	28
2700	4	26.35	300	60
2700	48	176.7	108	36
4200	4	28.65	480	60
4200	48	239.93	128	32
2000	7	40.31	168	28
2000	40	135.72	96	32
6000	7	50.41	300	60
6000	40	236.8	168	28
2700	7	44.13	180	36
2700	40	168.79	108	36
4200	7	48.44	300	60
4200	40	213.27	168	28

Table 5.4: Performance of Heuristic 0 on the AMD Istanbul machine.

N ncore Y			PS	$\frac{PS}{ES}\%$	PSPAYG	$\frac{PSPAYG}{ES}\%$	$\frac{PSPAYG}{PS}\%$
2000	4	1	24.81	100	24.81	100	100
2000	48	1	140.1	100	140.1	100	100
6000	4	1	30.36	100	30.36	100	100
6000	48	1	272.55	100	272.55	100	100
2700	4	1	24.24	92	24.24	92	100
2700	48	1	169.32	95.83	169.32	95.83	100
4200	4	1	26.8	93.52	26.8	93.52	100
4200	48	1	237.19	98.86	237.19	98.86	100
2000	7	1	40.31	100	40.31	100	100
2000	40	1	126.66	93.32	126.66	93.32	100
6000	7	1	50.36	99.9	50.36	99.9	100
6000	40	1	236.8	100	236.8	100	100
2700	7	1	40.4	91.56	40.4	91.56	100
2700	40	1	164.76	97.61	164.76	97.61	100
4200	7	1	44.64	92.16	44.64	92.16	100
4200	40	1	213.27	100	213.27	100	100

Table 5.5: Performance of Heuristic 1 on the AMD Istanbul machine

N	n	core	Y	PS	$\frac{PS}{ES}\%$	PSPAYG	$\frac{PSPAYG}{ES}\%$	$\frac{PSPAYG}{PS}\%$
2000	4	0		22.84	92.06	22.84	92.06	100
2000	48	1		140.1	100	140.1	100	100
6000	4	0		29.47	97.07	29.47	97.07	100
6000	48	0		256.42	94.08	256.42	94.08	100
2700	4	0		22.9	86.92	22.9	86.92	100
2700	48	1		169.32	95.83	169.32	95.83	100
4200	4	0		25.87	90.28	25.87	90.28	100
4200	48	1		239.93	100	239.93	100	100
2000	7	0		36.92	91.57	36.92	91.57	100
2000	40	1		126.66	93.32	126.66	93.32	100
6000	7	0		49.07	97.35	49.07	97.35	100
6000	40	0		224.13	94.65	224.13	94.65	100
2700	7	0		38.83	88	38.83	88	100
2700	40	1		164.76	97.61	164.76	97.61	100
4200	7	0		43.04	88.85	43.04	88.85	100
4200	40	0		209.74	98.34	209.74	98.34	100

Table 5.6: Performance of Heuristic 2 on the AMD Istanbul machine

N	n	core	Y	PS	$\frac{PS}{ES}\%$	PSPAYG	$\frac{PSPAYG}{ES}\%$	$\frac{PSPAYG}{PS}\%$
2000	4	1		24.81	100	24.81	100	100
2000	48	1		140.1	100	140.1	100	100
6000	4	0		29.98	98.75	29.35	96.66	97.89
6000	48	1		272.55	100	272.55	100	100
2700	4	1		24.24	92	24.24	92	100
2700	48	0		169.32	95.83	169.32	95.83	100
4200	4	1		26.8	93.52	26.8	93.52	100
4200	48	0		237.19	98.86	237.19	98.86	100
2000	7	1		40.31	100	40.31	100	100
2000	40	1		135.72	100	135.72	100	100
6000	7	1		50.36	99.9	50.36	99.9	100
6000	40	1		236.8	100	236.8	100	100
2700	7	0		40.4	91.56	40.4	91.56	100
2700	40	0		157.06	93.05	157.06	93.05	100
4200	7	1		44.64	92.16	44.64	92.16	100
4200	40	1		213.27	100	213.27	100	100

Chapter 6

Tuning Dense Linear Algebra for Hybrid Architecture: MAGMA

The Matrix Algebra on GPU and Multicore Architectures (MAGMA) project (10) is a demonstration of algorithmic techniques and their effect on performance and portability across hybrid systems. Designed to be similar to LAPACK in functionality, data storage, and interface, the MAGMA libraries allows scientists to effortlessly port their LAPACK-relying software components and to take advantage of each component of the new hybrid architectures. Current work of MAGMA spans in GPU-based systems. MAGMA efficiently deals with the complex challenges stemming from the heterogeneity of the current GPU-based systems.

MAGMA represents DLA algorithms as a collection of BLAS-based tasks and dependencies among them (see Figure 6.1). It uses parametrized task granularity to facilitate auto-tuning frameworks and performance models to facilitate the task splitting/mapping. The execution of the BLAS-based tasks are scheduled over the multicore and the GPU: Usually small, non-parallelizable tasks are scheduled on the CPU and large, parallelizable (in particular data parallel tasks) are off-loaded to the GPU. MAGMA hard-coded the algorithm's *critical path* and prioritize its execution/scheduling.

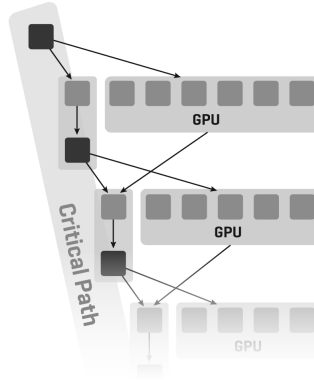


Figure 6.1: Algorithms as collection of BLAS-based tasks and dependencies among them (DAGs) for hybrid GPU-based computing

The splitting of the algorithms into tasks is in general easy as it is based on the splitting of large BLAS into smaller ones. More challenging is choosing the granularity and shape of the splitting and the subsequent scheduling of the sub-tasks. There are two main guiding directions on how to design the splitting and scheduling of tasks. **First**, the splitting and scheduling should allow for asynchronous execution and load balance among the hybrid components. **Second**, it should harness the strengths of the components of a hybrid architecture by properly matching them to algorithmic/task requirements.

Scheduling is very important for the efficient execution of MAGMA's algorithm. In general, the execution of the critical path of an algorithm should be scheduled as soon as possible. This often remedies the problem of synchronizations introduced by small non-parallelizable tasks (often on the critical path; scheduled on the CPU) by overlapping their execution with the execution of larger more parallelizable ones (often Level 3 BLAS; scheduled on the GPU).

Choosing the task granularity, can be done by parametrizing the tasks' sizes in the implementations and tuning them empirically (11). Currently MAGMA provides an interface to the user to manually set the panel size parameter, NB . More precisely

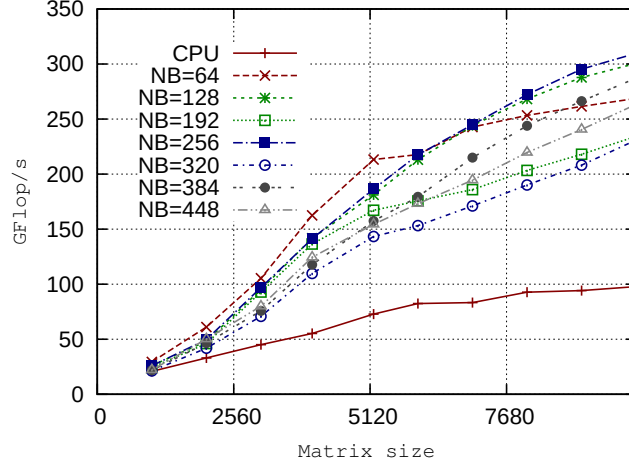
NB is the number of columns in a panel. The goal of this section is to investigate if the process can be automated (8). Auto-tuning is crucial for the performance and the maintenance of modern numerical libraries, especially for algorithms designed for hybrid architectures. MAGMA needs to use different values of NB s in different hosts. Figure 6.2 shows the performance of MAGMA's LU factorization with different values of NB on two different platforms. GPUs on both platform are pretty similar and the performance of GPU BLAS on both platform are very comparable. But due to different host characteristics, one NB which is optimal in one system might not be optimal in another system. This gives us the motivation for tuning NB in MAGMA. The data for Figure 6.2 is presented in terms of numerical values in Table 6.1, 6.2. Currently I am investigating if the tuning procedure of NB can be automated by following the same procedure as it was done in PLASMA's QR factorization. I am also investigating the idea of using different values of NB for a single matrix factorization.

Table 6.1: Performance of MAGMA's LU Factorization on GTX 280 for different panel size

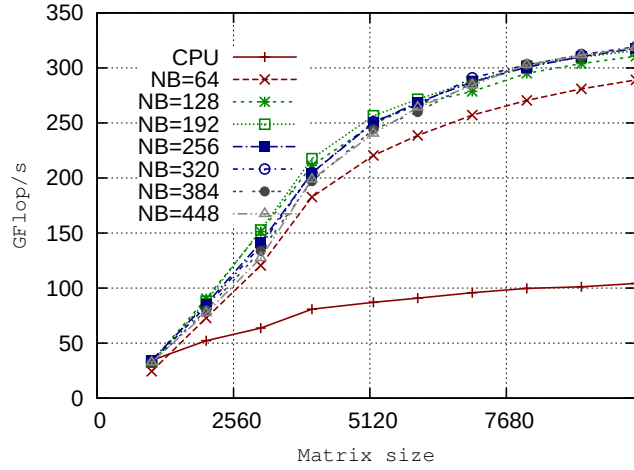
Size	NB=64	NB=128	NB=192	NB=256	NB=320	NB=384	NB=448
1024	24.47	33.2	32.66	34.4	32.76	31.98	32.59
2048	72.76	90.44	88.12	85.5	83.41	80.01	77.39
3072	120.44	151.31	152.81	141.48	138.32	133.79	127.34
4032	182.64	211.3	217.62	204.5	204.6	197.05	198.9
5184	220.56	249.03	256.53	250.12	251.17	244.14	240.35
6016	238.68	267.11	271.51	268.55	266.51	259.94	264.07
7040	257.14	278.6	287.85	287.16	291.02	286.33	284.36
8064	270.53	295.03	301.59	300.64	302.75	302.68	302.9
9088	280.98	303.91	309.69	310.28	312.07	309.58	311.79
10112	289.05	310.6	315.87	317.76	319.12	319.36	318.52

Table 6.2: Performance of MAGMA’s LU Factorization on TESLA for different panel size

Size	NB=64	NB=128	NB=192	NB=256	NB=320	NB=384	NB=448
1024	29.3	26.04	24.33	25.83	21.16	21.83	22.08
2048	60.98	48.09	46.12	49.73	41.91	46.33	50.21
3072	105.4	95.21	92.81	97	70.77	75.67	79.94
4032	162.44	141.73	136.53	141.34	109.77	117.68	124.36
5184	213.18	181.12	167.14	186.55	143.33	157.43	154.33
6016	217.66	213.02	175.58	217.45	153.27	179.31	173.16
7040	242.76	244.04	185.92	244.91	171.08	215	194.53
8064	253.4	268.16	203.4	272.25	190.11	243.9	219.66
9088	261.41	287.85	218.07	295.31	208.17	266.35	240.58
10112	268.81	300.55	234.37	309.73	230.94	287.18	263.34



(a) **GPU:** Tesla T10 Processor, 1440.0 MHz clock, 4095.8 MB Global Memory, **HOST:** Quad Core AMD Opetron(4 socket $\times$ 4 = 16 cores, 2.4 GHz, 2MB L3 Cache per socket, 512KB Shared L2 Cache, 64 KB L1 Cache) 8358SE



(b) **GPU:** GeForce GTX 280, 1296.0 MHz clock, 1023.8 MB Global memory, **HOST:** Intel(R) Xeon(R) CPU (8 Core, E5410 @ 2.33GHz)

Figure 6.2: MAGMA's LU performance for different panel size.

Chapter 7

Conclusion

In this work, I have presented some new algorithms for some of the important GPU BLASs that are building blocks of many hybrid DLA routines. New algorithms bring unprecedented performance comparing to BLAS provided by the vendors, e.g. auto-tuned sSYMV with recursive blocking gets up to 102 GFlops/s in GTX 280 *VS* 2 GFlops/s with CUBLAS. I have also revisited existing optimized algorithms for some of the BLAS (e.g. xTRSM, xGEMM) and pointed out different tunable parameters for those algorithms. After that, I have addressed different state of the art loop optimization techniques and emphasized their prospect in the context of GPU BLASs, e.g. circular loop skewing removes performance oscillation (the magnitude of the oscillation is 100 to 150 GFlops/s in single precision) in sSYR2K and some version of xGEMM (AB^T). Then I have presented a new pointer redirecting method that helps getting comparable performance in problem sizes (e.g. 1001, 2135) that are not multiple of certain hardware parameters. Removing these performance oscillations from GPU BLASs are important for two reasons. First, it removes performance oscillation in higher level DLA algorithm (e.g. 20 to 50% performance improvement in MAGMA's QR factorization). Second, this will make the auto-tuning easier from higher level.

In this work, I have presented an auto-tuning framework to tune GPU BLAS routines for the existing algorithms and new algorithms. I have also shared some of the success stories of auto-tuner in GPU BLAS context and their effect on higher level DLA algorithms. Future work involves finding optimized BLAS algorithms for NVIDIA's new architecture FERMI. Some preliminary work has shown that DGEMM gets upto 270 GFlops/s using CUDA *VS* 320 GFlops/s with NVIDIA's assembly kernel. Perhaps availability of new compiler will improve the performance of the same code and make the development of BLAS routines a lot easier.

After that, a new autotuned method for dense linear algebra libraries on multicore architectures has been presented. The approach has been validated, thanks to the PLASMA library on a wide range of architectures representative of today's HPC CPU trends. The framework was illustrated with the QR factorization, which is representative of the difficulty of tuning any of the three one-sided factorizations (QR, LU, Cholesky) present in PLASMA. The experimental validation has shown that the whole autotuning process can in general be brought to completion in a decent time (less than two hours in most cases) though allowing to achieve a very high performance (often finding the optimum tunable parameters and consistently achieving at least 90% of the optimum performance in the case of our best heuristic) on a wide range of architectures.

In PLASMA's autotuning, the factorization of square matrices was investigated. The factorization of non square matrices has to be studied too. In particular, the case of tall and skinny matrices (which have a larger number of rows than columns) often arises in several important applications (33). In (32), the authors have shown that communication-avoiding algorithms (33) are well adapted for processing such matrices in a multicore context. They consist of splitting further the matrix in multiple block rows (called domains) to enhance parallelism. The number p of domains is another tunable parameter that combines with NB and IB and should thus be integrated in the empirical search method. This is a possible future work.

Then some insights about the tuning in hybrid context were presented. Currently, I am investigating if the auto-tuning framework in PLASMA can be reused to tune MAGMA's parameters.

Bibliography

Bibliography

- [26] V. Volkov and J. Demmel. Benchmarking gpus to tune dense linear algebra. In Proc. of *SC '08*, pages 1–11, Piscataway, NJ, USA, 2008. 2, 3, 4, 6, 19, 46
- [2] CUDA CUBLAS Library. <http://developer.download.nvidia.com>. 2, 6, 9, 28
- [3] BLAS: Basic linear algebra subprograms. <http://www.netlib.org/blas/>. 47
- [4] Math Kernel Library (MKL), Intel(R) <http://www.intel.com/cd/software/products/asm-na/eng.347757.htm>. 2, 3
- [5] AMD Core Math Library (ACML). 2, 3
- [6] IBM, Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL. <http://www-03.ibm.com/systems/p/software/essl/>. 2, 3, 57
- [7] GoToBLAS, Texas Advanced Computing Center <http://www.tacc.utexas.edu/>
- [8] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing* **27** (2001), no. 1-2, 3–35. 4, 37, 74
- [9] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. LAPACK Working Note 217, Accepted for publication at SC '09. 54
- [10] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 Users' Guide. <http://icl.cs.utk.edu/magma>, 11/2009. 2, 4, 6, 9, 72

- [11] Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In Proc. of *ICCS '09*, pages 884–892, Baton Rouge, LA, 2009. 2, 6, 73
- [12] S. Barrachina, M. Castillo, F. Igual, R. Mayo, and E. Quintana-Orti Evaluation and Tuning of the Level 3 CUBLAS for Graphics Processors In PDSEC '08. 35
- [13] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences Department, University of California at Berkeley, 2006. 1
- [14] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992. 1, 2, 10
- [15] BLAS: Basic linear algebra subprograms. <http://www.netlib.org/blas/>. 2
- [16] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide. <http://developer.download.nvidia.com>, 2007.
- [17] S. Tomov, R. Nath, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. UTK EECS Technical Report ut-cs-09-649, December 2009. 8, 13
- [18] S. Tomov and J. Dongarra. Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing. Technical Report 219, LAPACK Working Note, May 2009. 9, 11
- [19] James W. Demmel. Trading Off Parallelism and Numerical Stability, EECS Department, University of California, Berkeley, UCB/CSD-92-702, September 1992. 11

- [20] Nicholas J. Higham. Stability of parallel triangular system solvers SIAM J. Sci. Comput., vol. 16, no. 2, 1995, pp. 400–413. [26](#)
- [21] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. Proceedings of the IEEE 93 (2005), no. 2, special issue on “Program Generation, Optimization, and Adaptation”. [37](#)
- [22] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. International Conference on Supercomputing, 1997, pp. 340–347. [37](#)
- [23] Matteo Frigo and Steven G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, vol. 3, IEEE, 1998, pp. 1381–1384. [37](#)
- [24] Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In ICCS’09, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag. [37](#)
- [25] Michael Wolfe, Compilers and More: Optimizing GPU Kernels. HPC Wire, <http://www.hpcwire.com/features/33607434.html>, 10/2008. [46](#)
- [26] V. Volkov and J. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC ’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press. [2](#), [3](#), [4](#), [6](#), [19](#), [46](#)
- [27] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing, 1, pages, 38–53, volume 35, year 2009. [2](#), [47](#), [48](#), [49](#)
- [28] N. Christofides. Graph Theory: An algorithmic Approach. New York: Academic Press, 1975. [47](#)

- [29] C. Whaley, and M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. In *Software: Practice and Experience*, volume 38, number 15, 2008, pages, 1621–1642. [xiii](#), [52](#), [53](#), [58](#)
- [30] H. Meuer and E. Strohmaier and J. Dongarra and H. Simon. TOP500 Supercomputer Sites. Edition 32nd, November, 2009. <http://www.netlib.org/benchmark/top500.html>. [55](#)
- [31] IBM LoadLeveler for AIX 5L. First Edition, December 2001.
- [32] B. Hadri and H. Ltaief and E. Agullo and J. Dongarra. Enhancing Parallelism of Tile QR Factorization for Multicore Architectures. Innovative Computing Laboratory, University of Tennessee, Submitted to *Transaction on Parallel and Distributed Systems*. [78](#)
- [33] J. W. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. LAPACK Working Note, 204, UTK, August, 2008. [78](#)
- [34] H. Ltaief and S. Tomov and R. Nath and J. Dongarra. Hybrid Multicore Cholesky Factorization with Multiple GPU Accelerators. Innovative Computing Laboratory, University of Tennessee, *Transaction on Parallel and Distributed Systems*, submitted.
- [35] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. “DAGuE: A generic distributed DAG engine for high performance computing,”. ICL-UT-10-01, UTK, April, 2010.
- [36] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The Impact of Multicore on Math Software. PARA 2006, Umea, Sweden, June 2006 [2](#)

- [37] L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK Users’ Guide”. SIAM, 1997. [2](#)
- [38] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. ACM SIGPLAN Conference on Programming Language Design and Implementation. June, 2009, Dublin, Ireland. [4](#)
- [39] C Chan, J. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman. Autotuning Multigrid with PetaBricks. ACM/IEEE Conference on Supercomputing, Nov, 2009, Portland, OR. [4](#)
- [40] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP). [4](#)

Appendix

Vita

Rajib holds a bachelors degree in Computer Science and Engineering from Bangladesh University of Engineering and Technology, Bangladesh. Following his graduation, Rajib worked as a quantitative software developer at Stochastic Logic Ltd, Bangladesh where he was involved in cutting-edge research in Computational Finance and helped developing a software aimed to predict financial market behavior. Rajib had also worked as a lecturer at the School of Computer Science and Engineering, United International University, Bangladesh. In summer of 2008, Rajib worked as a research intern at the French National Institute for Research in Computer Science and Control (INRIA). During his employment with INRIA, Rajib worked with the fault tolerance of VIGNE, a Grid Middleware, under the supervision of Christine Morin and Thomas Ropars. In summer 2009, Rajib worked with Greg Henry at the Math Kernel Library (MKL) department of Intel (R). His work at Intel centered on Super Compilier, a Generic Automatic Code Optimizer tool. He started working at Innovative Computing Laboratory(ICL) in University of Tennessee, Knoxville in August, 2008. During his employment at ICL, Rajib work was actively involved in two flagship projects of ICL: Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) and Matrix Algebra on GPU and Multicore Architectures (MAGMA).