Autotuning GEMM Kernels for the Fermi GPU

Jakub Kurzak, *Member*, *IEEE*, Stanimire Tomov, *Member*, *IEEE*, and Jack Dongarra, *Life Fellow*, *IEEE*

Abstract—In recent years, the use of graphics chips has been recognized as a viable way of accelerating scientific and engineering applications, even more so since the introduction of the Fermi architecture by NVIDIA, with features essential to numerical computing, such as fast double precision arithmetic and memory protected with error correction codes. Being the crucial component of numerical software packages, such as LAPACK and ScaLAPACK, the general dense matrix multiplication routine is one of the more important workloads to be implemented on these devices. This paper presents a methodology for producing matrix multiplication kernels tuned for a specific architecture, through a canonical process of heuristic autotuning, based on generation of multiple code variants and selecting the fastest ones through benchmarking. The key contribution of this work is in the method for generating the search space; specifically, pruning it to a manageable size. Performance numbers match or exceed other available implementations.

Index Terms—Graphics processing unit, matrix multiplication, code generation, automatic tuning, GEMM, BLAS, CUDA

1 Introduction

Graphics Processing Units (GPUs) maintain a strong lead over more traditional multicore CPUs in peak floating-point performance and memory bandwidth [31], which also translates to higher power efficiency. Hybrid accelerator-based systems have also been identified as likely candidates to deliver Exascale performance in the future [12], [22], [39]. Today, many key scientific and engineering applications rely on GPUs to deliver performance in excess of what standard multicores are capable of providing [24]. Due to its computational intensity and algorithmic regularity, dense linear algebra is a perfect candidate for GPU acceleration, and matrix multiplication is the canonical GPU programming example [31].

The hardware target of this paper is the NVIDIA Fermi GPU (GF100 architecture) GPUs [33], [35], [32], the first line of GPUs with essential high-performance computing features, such as high performance in double precision arithmetic and memory with *Error Correction Code* (ECC) protection. This device is usually programmed with NVIDIA's *Compute Unified Device Architecture* (CUDA) [31]. The OpenCL standard [21] could be used as an alternative, but currently its available implementations are known to lag behind CUDA in performance [14].

The workload implemented here is general matrix multiplication, referred to as *GEMM*, following the *Basic Linear Algebra Subroutines* (BLAS) standard [6]. The GEMM routine is a building block of software packages such as LAPACK [3] and ScaLAPACK [8], absolutely essential to their performance, and can also be used as the basis for implementing all other Level-3 BLAS routines [20].

• *The authors are with the Electrical Engineering and Computer Science Department, University of Tennessee, 1122 Volunteer Blvd, Ste 413 Claxton, Knoxville, TN 37996-3450.*
E-mail: {kurzak, tomov, dongarra}@eecs.utk.edu.

Manuscript received 26 June 2011; revised 23 Nov. 2011; accepted 22 Dec. 2011; published online 30 Dec. 2011.
Recommended for acceptance by S. Ranka.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-06-0427.
Digital Object Identifier no. 10.1109/TPDS.2011.311.

Not without significance is the fact that GEMM is also critical to the performance of the *High Performance Linpack Benchmark* (HPL) [13], used to rate the systems on the Top500 list of the fastest (disclosed) computers in the world. In June 2011, the top spot was captured by the Tianhe-1A supercomputer in China, a hybrid system based on Intel Xeon processors and NVIDIA Fermi GPUs. However, in November 2011, the Tianhe-1A supercomputer was pushed to the second place by the K computer in Japan, which does not use GPUs.

This work addresses the development of BLAS-compliant GEMM, with support for all parameters specified by the standard. Different variants of GEMM, with respect to the floating-point precision (single/double) and the type of arithmetic (real/complex), are referred to by their BLAS names (*SGEMM, DGEMM, CGEMM, ZGEMM*). Column-major matrix layout is used here, following the "legacy" BLAS interface and the convention of LAPACK and ScaLAPACK.

The software is being developed as a component of the *Matrix Algebra for GPUs and Multicore Architectures* (MAGMA) project [2]. It is the authors' intention to use it as an initial proof-of-concept prototype for a framework, *Automatic Stencil TuneR for Accelerators* (ASTRA).

2 CUDA Basics

In November 2006, NVIDIA introduced the *Compute Unified Device Architecture*, a general purpose parallel computing architecture, with a new parallel programming model and instruction set architecture, that leverages the parallel compute engine in NVIDIA GPUs to solve complex computational problems.

At its core are three key abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization, which are exposed to the programmer as a set of language extensions. They guide the programmer to partition the problem into coarse subproblems that can be solved independently in parallel by blocks of threads, and each subproblem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads.

The CUDA architecture is built around a scalable array of multithreaded *Streaming Multiprocessors* (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called *Single-Instruction, Multiple-Thread* (SIMT). The instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism extensively with simultaneous hardware multithreading. However, unlike CPU cores, they are issued in order and there is no branch prediction and no speculative execution.

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. The term warp originates from weaving, the first parallel thread technology. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution.

3 Motivation

Initially, this work was motivated by the observation that, while CUBLAS and MAGMA single precision GEMM achieved much higher performance in complex arithmetic than in real arithmetic, double precision did not. The higher performance was due to the higher computational intensity of complex arithmetic and should have manifested itself equally in both single precision and double precision. Since this was not the case, a clear performance improvement opportunity presented itself. At the same time, there are important applications where complex double precision GEMM is essential [5].

The main motivation for this work, however, was the delivery of optimized GEMM GPU kernels, produced automatically through a robust process of code generation and autotuning. Until now, the GEMM kernels in MAGMA were produced through exhaustive experimentation, rather than a systematic autotuning process. With the new Kepler and Maxwell architectures planned for 2011 and 2013, respectively, as disclosed in NVIDIA's roadmap, a much more sustainable process is in high demand.

It is of significance that the high-level abstraction of CUDA maps very well to the hardware architectures of NVIDIA GPUs. Programmers do not have to resort to lower level abstractions, such as the *Parallel Thread Execution* (PTX) [34] (the pseudoassembly of CUDA), for the development of fast GEMM kernels for NVIDIA cards. At the same time, CUDA GEMM codes proved not to be "performance-portable," as was shown by efforts of porting kernels for the GT200 (Tesla) architecture to the GF100 (Fermi) architecture. This combination of factors makes NVIDIA GPUs attractive targets for autotuning efforts.

Autotuning is an attractive option for GPU code development. First of all, many important architectural details are proprietary knowledge, undisclosed to the public, such as the mechanism for scheduling of blocks within the device and scheduling of warps within the multiprocessor. Second, low-level programming constructs are inaccessible, i.e., there is no publicly available assembler for NVIDIA GPUs. The lowest accessible level is the one of PTX. Yet the strongest motivation for autotuning on GPUs is probably the complexity of these massively parallel, massively hardware-multithreaded devices.

For conventional architectures it has been shown that hand-tuning has the capability to outperform autotuning [17], [23], and also, tuning parameters can be determined analytically [44], [23]. Nonetheless, autotuned libraries are in wide use due to their ability to quickly adapt to new platforms. The autotuning approach presented here is envisioned to have similar benefits as new generations of GPUs are developed.

4 Related Work

The list of prominent autotuning software projects includes: *Automatically Tuned Linear Algebra Software* (ATLAS) [43], and its predecessor *Portable High Performance ANSI C* (PHiPAC) [7], *Optimized Sparse Kernel Interface* (OSKI) [42], *Fastest Fourier Transform in the West* (FFTW) [16], and *SPIRAL* [37] (code generation for digital signal processing transforms). All these projects address autotuning for standard processors (not accelerators).

Jiang and Snir [19] developed an ATLAS-like autotuning system for matrix multiplication on Nvidia GPUs before CUDA was available. As such, pre-CUDA efforts are often referred to as *General Purpose GPU* (GPGPU) programming. The BrookGPU language was used to express a parameterized kernel with 458,752 possible instantiations. The authors used ad hoc pruning based on arbitrary problem-specific choices, and exploited orthogonality of parameters during the actual search.

Barrachina et al. [4] carried out a preliminary study of Level 3 CUDA BLAS (CUBLAS) using the GeForce 8800 Ultra card (G80 architecture). Performance of SGEMM, SSYRK, and STRSM routines was reported. Subsequently, three optimization techniques were applied, which did not involve any modifications to the CUDA source code: padding of input matrices, implementation of SSYRK, and STRSM on top of SGEMM, and splitting the work between the GPU and a CPU. Altogether, the application of these techniques produced substantial performance improvements with minimal programming effort and no coding in CUDA.

Early work on tuning GEMMs in CUDA for NVIDIA GPUs targeted the previous generation of GPUs, of the GT200 architecture, such as the popular GTX 280. Pioneering work was done by Volkov and Demmel [41]. Similar efforts followed in the MAGMA project [25]. The introduction of the NVIDIA Fermi architecture triggered the development of MAGMA GEMM kernels tuned for that

architecture [29], [28]. Although tuning was an important part of this work, it was accomplished through exhaustive experimentation rather than a systematic autotuning effort.

One important development in MAGMA was the implementation of complex GEMM routines by expressing the complex matrix multiplication through three real matrix multiplications and five real matrix additions [15], which results in up to 25 percent decrease in the number of floating-point operations [29]. However, Higham observes that this method has a fundamental numerical weakness, since the "imaginary part may be contaminated by relative errors much larger than those for conventional multiplication" [18]. Although, Higham also notes that "if the errors are measured relative to $\|A\|\,\|B\|$ [...], then they are just as small as for conventional multiplication" [18]. The method simply employs the SGEMM and DGEMM routines for an implementation of the CGEMM and ZGEMM routines with a reduced number of floating-point operations and different numerical properties. Since it does not involve implementation of any new kernels, it will not be further discussed here.

An important approach to the development of optimized GEMM routines is code generation through compiler transformations. Rudy et al. [38] presented the *CUDA-CHiLL* source-to-source compiler transformation and code generation framework, which transforms sequential loop nests to high-performance GPU code, based on a polyhedral transformation system CHiLL [9]. Autotuning was used to explore a small parameter space (tiling in multiples of 16, up to 128). Fermi SGEMM $A \times B$ kernel was produced with performance slightly lower than CUBLAS, due to not using texture caches (which has been remedied since then, according to the authors).

Cui et al. [11] presented a similar system built using the Open64 compiler [36] and the WRaP-IT/URUK/URGenT polyhedral toolchain [10]. Here the authors started with optimized MAGMA/CUBLAS Fermi SGEMM kernels ($A \times B, A^T \times B, A \times B^T, A^T \times B^T$) and used automatic code transformations to extrapolate the SGEMM performance to the other three Level 3 BLAS kernels (STRMM, STRSM, SSYMM) with all combinations of inputs covered (left/right, lower/upper). Indeed, performance very close to SGEMM was reported for all the other kernels, greatly outperforming CUBLAS.

Two recent articles describe low-level development of GPU matrix multiplication, without any use of autotuning methodology. Tan et al. [40] describe a joint work by the Institute of Computing Technology, Chinese Academy of Science and NVIDIA on the development of the double precision (DGEMM) $A \times B$ kernel to be used in the *High Performance Linpack* (HPL) benchmark for the Tianhe-1A and Nebulae Chinese supercomputers. The process is based on meticulous analysis of the hardware and instruction scheduling in assembly using NVIDIA's assembler for Fermi (not publicly available). Impressive performance of 362 Gflop/s is reported, which corresponds to 70 percent efficiency.

Similar work has been done by Nakasato [27] who developed GEMM kernels for the Cypress GPU from ATI. Single and double precision $A \times B$ and $A^T \times B$ kernels were developed in real arithmetic (using row-major layout). An astounding performance of 2 Tflop/s in single precision and 470 Gflop/s in double precision was shown. The kernels were coded using AMD *Intermediate Language* (IL), an assembly-like language for the AMD IL virtual instruction set architecture [1].

5 Original Contribution

One contribution of this work is the introduction of a universal code stencil for producing all variants of the GEMM routine included in the BLAS standard. This universal code supports: real and complex arithmetic, single and double precision, transposed, nontransposed, and conjugate transposed layout of input matrices. The code also supports memory access with and without using texture caches, with texture reads implemented as both 1D texture reads and 2D texture reads.

The main contribution of this work is in the search space generator, specifically in the mechanism for pruning the search space. Especially important is the fact that the size of the search space can easily be controlled and adjusted to a smaller size for quicker searches or to a bigger size for more exhaustive searches. At the same time, the parameters controlling the size of the search space are intuitive to anyone with basic understanding of the *Single Instruction Multiple Threads* GPU programming model.

Finally, the desired products of this work are GEMM kernels for the NVIDIA Fermi architecture that match or exceed existing CUBLAS kernels and previous MAGMA kernels in all cases, with significant improvement in the case of the complex double precision kernel (ZGEMM).

6 Solution

6.1 Hardware Target

A number of articles are available with the details of the Fermi architecture [33], [35], [32]. Here, the most important differences from the previous generation of NVIDIA GPUs are briefly discussed. The crucial new features include: fast double precision, L2 and L1 caches, and ECC protection.

The most important feature, from the standpoint of numerical computing, is double precision performance on a par with single precision performance. Double precision operations consume twice the storage of single precision operations (two 32-bit registers per element) and execute at half the throughput of single precision operations (16 operations per multiprocessor per cycle), which is the desired behavior. The *Fused Multiply-Add* (FMA) operation is available, which offers extra precision over the *Multiply-Add* (MADD) operation. Also, the floating-point hardware supports *denormalized* numbers and all four IEEE 754-2008 rounding modes (nearest, zero, positive infinity, negative infinity).

Fermi contains a 768 KB L2 cache shared by all multiprocessors and a 64 KB L1 cache per multiprocessor. The L1 can be configured as 16 KB of (hardware controlled) cache and 48 KB of (software controlled) *shared memory* or the other way around. Shared memory is more useful with more regular (more predictable) memory access patterns, while hardware cache is more useful with less regular (less predictable) access patterns. Since matrix multiplication is a

very regular and predictable workload, the first option is always used in this work, with 48 KB of shared memory and 16 KB of L1 cache. The L1 cache still plays a vital role in achieving performance by caching register spill, which would go to DRAM without the cache hierarchy.

Finally, Fermi is the first GPU to support ECC protection against bit flips caused by cosmic rays [30]. Fermi' s register files, shared memory, L1 cache, L2 cache, and DRAM are all ECC protected. One exception to the rule is Fermi' s texture cache. The texture cache has two parts: a 12 KB L1 cache in each SM and a larger L2 cache. While the L2 is ECC protected, the L1 is not. However, since the L1 is quite small, and the lifetime of data in the L1 is very low, silent errors are very unlikely in all but the largest installations. Also, the issue is expected to be fixed in the Kepler architecture [26].

6.2 GEMM Basics

The BLAS standard defines the general matrix multiplication operation as $C = \alpha A \times B + \beta C$, where C, A, and B are matrices of sizes $m \times n, m \times k$, and $k \times n$, respectively, and α and β are scalars. In canonical form, matrix multiplication is represented by three nested loops (Algorithm 1).[1]

Algorithm 1. Canonical form of matrix multiplication

for $m = 0$ **to** M **do**
  for $n = 0$ **to** N **do**
    for $k = 0$ **to** K **do**
      $C_{m,n} += \alpha A_{m,k} \times B_{k,n}$
    end for
    $C_{m,n} = \beta C_{m,n}$
  end for
end for

The primary tool in optimizing matrix multiplication is the technique of loop tiling. Tiling replaces one loop with two loops: the inner loop incrementing the loop counter with the step of one and the outer loop incrementing the loop counter with the step equal to the tiling factor. In the case of matrix multiplication, tiling replaces the three loops of Algorithm 1 with the six loops of Algorithm 2. Tiling improves locality of reference by exploiting the fact that matrix multiplication involves $O(n^3)$ floating-point operations over $O(n^2)$ data items, which is referred to as the *surface to volume effect*.

Algorithm 2. Tiling of matrix multiplication

for $\tilde{m} = 0$ **to** M **step** M_{tile} **do**
  for $\tilde{n} = 0$ **to** N **step** N_{tile} **do**
    for $\tilde{k} = 0$ **to** K **step** K_{tile} **do**
      for $m = 0$ **to** M_{tile} **do**
        for $n = 0$ **to** N_{tile} **do**
          for $k = 0$ **to** K_{tile} **do**
            $C_{\tilde{m}+m,\tilde{n}+n} += \alpha A_{\tilde{m}+m,\tilde{k}+k} \times B_{\tilde{k}+k,\tilde{n}+n}$
          end for
        end for
      end for
    end for
  end for
end for

Scaling C by β is skipped for clarity.

Finally, the technique of loop unrolling is applied, which replaces the three innermost loops with a single block of straight-line code (a single *basic block*), as shown by Algoritm 3. The purpose of unrolling is twofold: to reduce the penalty of looping (the overhead of incrementing loop counters, advancing data pointers and branching), and to increase instruction-level parallelism by creating sequences of independent instructions, which can fill out the processor' s pipeline.

Algorithm 3. Unrolling of matrix multiplication

for $\tilde{m} = 0$ **to** M **step** M_{tile} **do**
  for $\tilde{n} = 0$ **to** N **step** N_{tile} **do**
    for $\tilde{k} = 0$ **to** K **step** K_{tile} **do**
      instruction
      instruction
      instruction
      ...
    end for
  end for
end for

This optimization scheme is universal for almost any computer architecture, including "standard" superscalar processors with cache memories, GPU accelerators, and other unconventional architectures, such as the IBM Cell B. E. processor [23]. Tiling, also referred to as blocking, is often applied at multiple levels, e.g., L2 cache, L1 cache, registers file, etc. Unrolling an outer loop, and then fusing together copies of the inner loop, sometimes called unroll and jam, is usually applied until the point where the register file is exhausted. Unrolling the inner loop beyond that point, but without reordering instructions, is used to further reduce the overhead of looping.

6.3 Universal GEMM Stencil

6.3.1 General Structure

A GPU is a *data-parallel* device with the *barrier* being the only mechanism for synchronization. Therefore, parallelization relies on identifying independent work. Parallelization at the device level is shown on Fig. 1. Matrix multiplication of the general form $C = C + A \times B$ of size $M_{dev} \times N_{dev} \times K_{dev}$ is parallelized by spanning matrix C with a 2D grid of tiles. Each tile is processed by one thread block. Each thread block passes through a $M_{blk} \times K_{dev}$ stripe of A and a $K_{dev} \times N_{blk}$ stripe of B and produces the final result for a $M_{blk} \times N_{blk}$ tile of C. In one iteration of the outermost loop a thread block produces the partial result of a $M_{blk} \times N_{blk} \times K_{blk}$ matrix multiplication. (While K_{dev} is the loop boundary for the outermost loop, K_{blk} is the tiling factor for that loop.) The tile of C is read from the device memory and kept in registers throughout the duration of the thread block' s operation. $M_{blk} \times K_{blk}$ stripe of A and $K_{blk} \times N_{blk}$ stripe of B are placed in shared memory for each iteration of the outermost loop.

1. In this paper, "for" loops are understood to go from the starting value up to, but not including, the ending value. For example, "for i = A to B" is understood to be the same as the C code "for (i = A; i < B; i++)".

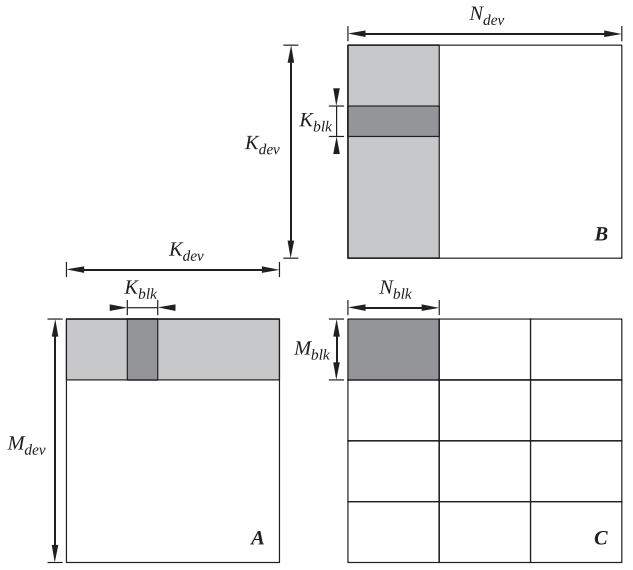

Fig. 1. GEMM at the device level.

Fig. 2 shows how a $M_{blk} \times N_{blk} \times K_{blk}$ partial result is produced in one iteration of the outermost loop of the thread block's code. The figure shows parallelization at the thread level. The light shade shows the shape of the thread grid and the dark shade shows the elements involved in the operations of a single thread.

Fig. 3 shows the operation from the perspective of one thread. Each thread streams in elements of A and B from the shared memory to the registers and accumulates the matrix multiplication results in C, residing in registers. This is the ideal situation. Whether it actually is the case depends on the actual tiling factors at each level. Whenever the compiler runs out of registers, register spills to memory will occur, which on Fermi is mitigated to some extent by the existence of the L1 cache.

Two comments about the use of shared memory are in place here. One important detail is that the shared memory is allocated in a skewed fashion, i.e., an array of size $M \times N$ is declared as $M \times N + 1$, which is the usual "trick" to eliminate bank conflicts whether a warp accesses the matrix

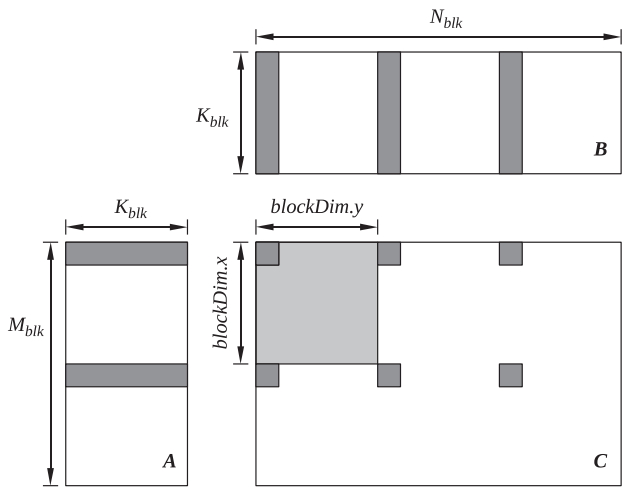

Fig. 2. GEMM at the block level.

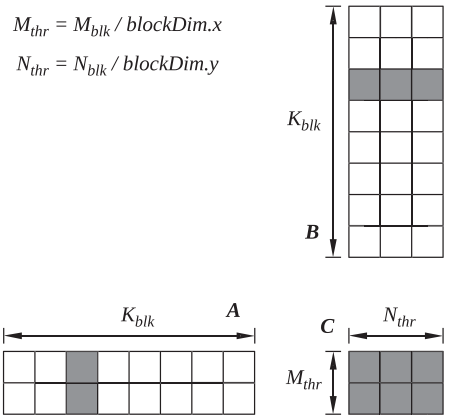

Fig. 3. GEMM at the thread level.

by rows or by columns. Skewing is required for matrix A if it is transposed, and always required for matrix B. Here, for simplicity, skewing is always applied to both matrices.

Another comment concerns the use of the shared memory in general. For some GPU architectures, the shared memory can be bypassed altogether for one of the input matrices [41], [27]. This turns out not to be the case on the Fermi, where the use of shared memory is required to mitigate strided access to the device memory and redundant reads from the device memory by multiple threads in the same block. Here, matrices A and B are always placed in the shared memory, which also simplifies the coding of different *transposed/nontransposed* scenarios.

6.3.2 Pipelined Loop

The last important concept in optimizing matrix multiplication is the classic technique of software pipelining (Fig. 4), also referred to as *double buffering*. The objective of software pipelining is to increase the instruction level parallelism by overlapping arithmetic operations from

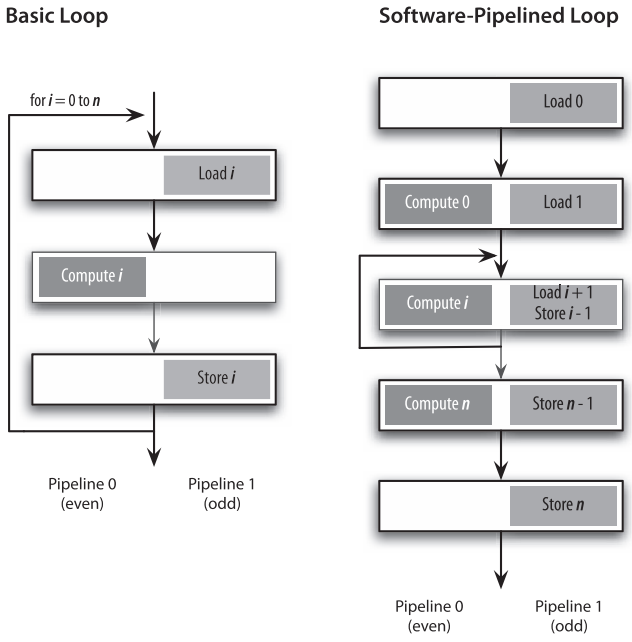

Fig. 4. Software pipelining.

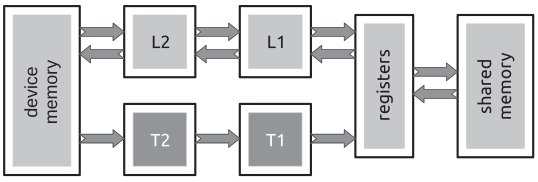

Fig. 5. Data paths in the Fermi GPU. L1 and L2 are caches, T1 and T2 are texture caches.

even iterations with memory operations from odd iterations. The concept is especially applicable to *dual-issue* architectures with two pipelines, one devoted to arithmetic and the other to memory accesses, which can issue instructions in the same cycle. An iconic dual-issue architecture was the IBM Cell B. E. processor.

The Fermi chip is also a dual-issue architecture and double buffering is applicable to the GEMM loops. One difference from the canonical structure of Fig. 4 stems from the data-parallel nature of the device. Since all the work is partitioned to completely independent sets, there is no need for storing the results until the very end. Because of that, computations of even iterations are only overlapped with reads of odd iterations, but not stores, which are done at the very end.

Fig. 5 shows the flow of data through the memory system of the Fermi GPU. Normally, data are read to registers and written back to the memory through L1 and L2 caches. Alternatively, read-only data can be read through texture caches. Texture caches do not have any fundamental performance advantage over L1 and L2 caches. However, passing read-only data through texture caches leaves more space in L1 and L2 for read/write data, register spills, etc. For GEMMs it always pays off to use texture caches to access the A and B input matrices. An added advantage is the use of texture clamping to avoid *cleanup codes*.

Data can also be transferred between the registers and the shared memory. The purpose of the shared memory is to enable communication between threads. It is a fallacy to think of the shared memory as a cache/scratchpad memory/local store. The GPU philosophy is to hide memory latency through massive SIMT, not through the use of a scratchpad memory. If one treats the GPU as a vector device, where a warp is the vector, then the shared memory can be thought of as a vector register file, enabling shuffles/permutations of vector elements. In the context of matrix multiplication, the shared memory allows one to deal efficiently with transposed access.

Algorithm 4 shows the pseudocode for the generic GEMM stencil. The code follows the classic pipelined loop scheme with a prologue and epilogue. In the steady state, the loop body loads elements of A and B from shared memory to registers and computes their product, while at the same time loading another batch of elements of A and B to a separate set of registers. The loading from the device memory to the "odd" registers happens in lines 6 and 7. The loading from the shared memory to the "even" registers and the computation happens in the loop between lines 8 and 12. The values in the "odd" registers are passed to the shared memory in lines 14 and 15, to be used in the upcoming iteration of the loop. This last part has to be protected with the `__syncthreads()` barriers, to avoid the data hazard on access to the shared memory. One of these barriers can be eliminated at the cost of doubling the shared memory usage. This, however, decreases occupancy, which inevitably decreases performance. The factors $alpha$ and $beta$ are applied when storing the results in the device memory (line 23). This pipelining scheme, developed for the MAGMA project by Nath et al. [29] has shown itself to be fastest in practice.

Algorithm 4. Generic GEMM stencil pseudocode

1: $C_{regs} \Leftarrow 0$
2: $A_0\ dev \Rightarrow shmem$
3: $B_0\ dev \Rightarrow shmem$
4: __syncthreads();
5: **for** $K = 0$ **to** $K_{dev} - K_{blk}$ **step** K_{blk} **do**
6: $\quad A_{odd}\ dev \Rightarrow regs$
7: $\quad B_{odd}\ dev \Rightarrow regs$
8: $\quad$ **for** $k = 0$ **to** K_{blk} **step** 1 **do**
9: $\quad\quad A_{even}[k]\ shmem \Rightarrow regs$
10: $\quad\quad B_{even}[k]\ shmem \Rightarrow regs$
11: $\quad\quad C = C + A_{even}[k] \times B_{even}[k]$
12: $\quad$ **end for**
13: $\quad$ __syncthreads();
14: $\quad A_{odd}\ regs \Rightarrow shmem$
15: $\quad B_{odd}\ regs \Rightarrow shmem$
16: $\quad$ __syncthreads();
17: **end for**
18: **for** $k = 0$ **to** K_{blk} **step** 1 **do**
19: $\quad A_{odd}[k]\ shmem \Rightarrow regs$
20: $\quad B_{odd}[k]\ shmem \Rightarrow regs$
21: $C = C + A_{even}[k] \times B_{even}[k]$
22: **end for**
23: $C_{dev} = alpha \times C_{regs} + beta \times C_{dev}$

regs—registers, *shmem*—shared memory, *dev*—device memory
even—even iteration, *odd*—odd iteration.

6.3.3 Parameterization

The code is generalized to handle: double and single precision, real and complex arithmetic, and transposition (and conjugation) of A and B. It also allows for reading the device memory with or without the use of texture caches. If texture caches are used, matrices A and B can be accessed either as 1D textures or 2D textures. Further on, only the use of 1D textures is discussed, since this is the fastest performing scenario. All options are controlled using the C preprocessor's macro definitions (`#define`) and the C language type definitions (`typedef`). Altogether the code can be compiled into 78 different variants. This does not result in a code bloat, because for the most part, different options are orthogonal. The entire stencil is roughly 500 lines long. Such small size is also due to the fact that unrolling is left entirely to the compiler. Only `#pragma unroll` directives are used. All loops are unrolled, except for the outermost loop (line 5). Such aggressive unrolling is a common practice on GPUs. The volume of resulting code very rarely prevents the compiler from unrolling it.

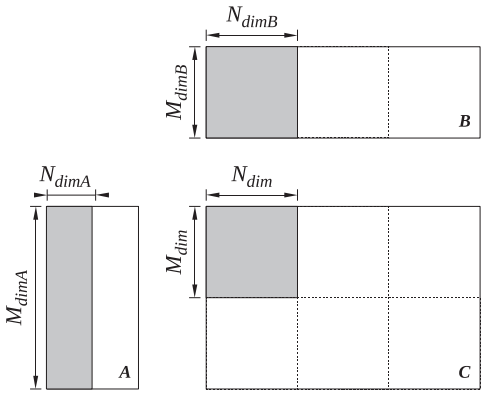

Fig. 6. Reshaping the thread block for reading A and B.

Different precisions (single/double) are handled by macros with type definitions. Complex arithmetic is handled by inline functions defined in the CUBLAS library (`cuCadd()`, `cuCmul()`, `cuCfma()`, etc.), which are cast to additions and multiplications for real arithmetic. So is conjugation of an input matrix if it is *conjugate-transposed*. Different ways of accessing the device memory (texture caches or no texture caches) are implemented through conditional compilation of the address translation blocks. Transposition of A and B is handled when loading from device memory to registers and shared memory (lines 2, 3; 6, 7; and 14, 15). The innermost loops performing the actual computation (lines 8-12) are oblivious to the layout of the input matrices.

Finally, the size of work for a thread block is parameterized ($M_{blk}, N_{blk}, K_{blk}$), as well as the shape of the thread grid ($blockDim.x, blockDim.y$). It can also be observed that the thread grid can be reshaped for reading of A and B as long as each of the three shapes perfectly overlay the corresponding matrix (Fig. 6). Therefore, the values $M_{dimA}, N_{dimA}, M_{dimB}, N_{dimB}$ are also the stencil's parameters (subject to preprocessor correctness checks). Also, for consistency, $blockDim.x$ and $blockDim.y$ will be referred to, from now on, as M_{dim} and N_{dim}.

6.4 Search Space Generator

The search space generator is a *brute-force* machinery that runs through all possible values of parameters $M_{blk}, N_{blk}, K_{blk}, M_{dim}, N_{dim}, M_{dimA}, N_{dimA}, M_{dimB}, N_{dimB}$ and rejects the combinations that produce invalid code and the combinations that do not meet certain performance guidelines, e.g., minimum occupancy requirement. To start with, the 9D parameter space is enormous. Constraints come from a few different sources. Here, the following categories of constraints are identified: *queryable hardware constraints, nonqueryable hardware constraints, hard implementation constraints, and soft implementation constraints.*

6.4.1 Hardware and Implementation Constraints

Queryable hardware constraints are hardware constraints which can be queried at runtime using calls to the CUDA runtime library (specifically the `cuDeviceGetAttribute()` function). Nonqueryable hardware constraints are hardware constraints which cannot be queried like that, but are tied to the GPU *compute capability* and defined in CUDA documentation (e.g., *"NVIDIA CUDA C Programming Guide"* [31, Appendix G]). Hard implementation constraints are constraints that would make the implementation invalid if violated, and soft implementation constraints are constraints that would make the implementation perform poorly if violated, but not make it invalid.

The following device parameters are queried:

- $WARP\_SIZE$,
- $MAX\_THREADS\_PER\_BLOCK$,
- $MAX\_REGISTERS\_PER\_BLOCK$,
- $MAX\_SHARED\_MEMORY\_PER\_BLOCK$.

Then the compute capability is checked using the `cuDeviceComputeCapability()` function and the following parameters are set using a table lookup:

- $MAX\_WARPS\_PER\_SM$,
- $MAX\_BLOCKS\_PER\_SM$.

Here, a few simplifying assumptions are made, that seem to hold for all compute capabilities so far. It is assumed that the maximum number of threads per multiprocessor is defined by the maximum number of warps

$$MAX\_THREADS\_PER\_SM = MAX\_WARPS\_PER\_SM \times WARP\_SIZE,$$

the number of 32-bit registers per multiprocessor equals the maximum number of registers per block, i.e.,

$$MAX\_REGS\_PER\_SM = MAX\_REGISTERS\_PER\_BLOCK,$$

and the amount of shared memory per multiprocessor equals the maximum amount of shared memory per block, i.e.,

$$MAX\_SHMEM\_PER\_SM = MAX\_SHARED\_MEMORY\_PER\_BLOCK.$$

6.4.2 Performance Guidelines

Performance guidelines are provided to the generator as input and allow for adjusting the amount of generated combinations. Three such guidelines are used here: minimum occupancy, minimum number of blocks per multiprocessor, and minimum register reuse. Minimum occupancy defines the minimum number of threads, per multiprocessor, that the kernel is required to launch. The SIMT computation model of the GPU relies on a massive number of simultaneously active threads to deliver performance, so it is reasonable to specify that a kernel should allow for, e.g., the minimum of 512 threads (out of 1,536) to be active at the same time in a multiprocessor. This constraint eliminates kernels that consume resources, such as register and shared memory, too aggressively. Similarly, the minimum number of blocks per multiprocessor requirement eliminates kernels that consume resources too heavily to allow for at least the given number of blocks to reside in one multiprocessor. Finally, the register reuse requirement forces a given number of floating-point operations to be performed per single memory operation. This constraint eliminates kernels that move data too much and do not compute enough. To be

precise, the value is a floating-point ratio of FMAs to loads in the innermost loop of the kernel (Algorith 4, lines 8-12).

6.4.3 Generation and Pruning

Algorithm 5 shows the nested loops of the search space generator. The two outermost loops iterate over the sizes of the thread grid. The three innermost loops iterate over the sizes of the block's working space. The steps for M_{blk} and N_{blk} are M_{dim} and N_{dim}, respectively (the thread grid has to overlay a tile of C). K_{blk} does not have to be constrained in any way. (The step is 1.)

Algorithm 5. Search space generator pseudocode

for $N_{dim} = 1$ **to** N_{dimMAX} **step** 1 **do**
for $M_{dim} = 1$ **to** M_{dimMAX} **step** 1 **do**

for $K_{blk} = 1$ **to** K_{blkMAX} **step** 1 **do**
for $N_{blk} = N_{dim}$ **to** N_{blkMAX} **step** N_{dim} **do**
for $M_{blk} = M_{dim}$ **to** M_{blkMAX} **step** M_{dim} **do**

if parameters meet constraints **then**
generate all variants
$(M_{dimA}, N_{dimA}, M_{dimB}, N_{dimB})$
such that:
$M_{dimA} \times N_{dimA} == M_{dim} \times N_{dim}$
$M_{dimA} \% M_{blk} == 0$
$N_{dimA} \% K_{blk} == 0$
$M_{dimB} \times N_{dimB} == M_{dim} \times N_{dim}$
$M_{dimB} \% K_{blk} == 0$
$N_{dimB} \% N_{blk} == 0$
end if
end for
end for
end for
end for
end for
end for

In principle, the loops upper boundaries could be set to some device parameters, e.g., the upper boundaries for the two outermost loops could be set to $MAX\_BLOCK\_DIM\_X$ and $MAX\_BLOCK\_DIM\_Y$. Here the boundaries are set to 256 for $M_{dimMAX}, N_{dimMAX}, M_{blkMAX}$, and N_{blkMAX}, and to 64 for K_{blkMAX}. The choice was made experimentally, such that no combinations are missed because of the loop boundaries being too low, i.e., increasing the boundaries does not produce any more valid combinations. (All such combinations are eliminated by the constraints discussed further.) At the same time, the running time of the generator is kept short (on the order of seconds).

Algorithm 6 shows the set of constraints enforced inside the nested loops of Algorithm 5. It is divided into four sections. The first section enforces a mixed set of hardware and implementation (hard and soft) constraints. The second section enforces the minimum occupancy performance guideline, based on the amount of available shared memory. The third section enforces the minimum occupancy performance guideline, based on the number of available registers. Finally, the fourth section enforces the minimum register reuse performance guideline. Next, each block is discussed in detail.

Algorithm 6. Search space generator constraints

Require: $M_{dim} \times N_{dim} \leq MAX\_THREADS\_PER\_BLOCK$
Require: $(M_{dim} \times N_{dim}) \% WARP\_SIZE == 0$
Require: $(M_{blk} \times K_{blk}) \% (M_{dim} \times N_{dim}) == 0$
Require: $(K_{blk} \times N_{blk}) \% (M_{dim} \times N_{dim}) == 0$

$shmem\_per\_block = ((M_{blk} + 1) \times K + (K + 1) \times N) \times sizeof(type)$
$blocks\_per\_sm = min(MAX\_SHMEM\_PER\_SM / shmem\_per\_block, MAX\_BLOCKS\_PER\_SM)$
$warps\_per\_block = (M_{dim} \times N_{dim}) / WARP\_SIZE$
$warps\_per\_sm = min(blocks\_per\_sm \times warps\_per\_block, MAX\_WARPS\_PER\_SM)$
$blocks\_per\_sm = warps\_per\_sm / warps\_per\_block$
Require: $blocks\_per\_sm \geq MIN\_BLOCKS\_PER\_SM$
$threads\_per\_sm = M_{dim} \times N_{dim} \times blocks\_per\_sm$
Require: $threads\_per\_sm \geq MIN\_THREADS\_PER\_SM$

$regs\_per\_thread = (M_{thr} \times N_{thr}) + (M_{thr} + N_{thr})$
$regs\_per\_block = regs\_per\_thread \times (M_{dim} \times N_{dim})$
$regs\_per\_block \mathrel{+}= M_{blk} \times K_{blk} + K_{blk} \times N_{blk}$
$regs\_per\_block \times= sizeof(type) / sizeof(float)$
$blocks\_per\_sm = min(MAX\_REGS\_PER\_SM / regs\_per\_block, MAX\_BLOCKS\_PER\_SM)$
$warps\_per\_block = (M_{dim} \times N_{dim}) / WARP\_SIZE$
$warps\_per\_sm = min(blocks\_per\_sm \times warps\_per\_block, MAX\_WARPS\_PER\_SM)$
$blocks\_per\_sm = warps\_per\_sm / warps\_per\_block$
Require: $blocks\_per\_sm \geq MIN\_BLOCKS\_PER\_SM$
$threads\_per\_sm = M_{dim} \times N_{dim} \times blocks\_per\_sm$
Require: $threads\_per\_sm \geq MIN\_THREADS\_PER\_SM$

if real arithmetic **then**
$regs\_reuse = (M_{thr} \times N_{thr}) / (M_{thr} + N_{thr})$
else {complex arithmetic}
$regs\_reuse = (4 \times M_{thr} \times N_{thr}) / (2 \times (M_{thr} + N_{thr}))$
end if
Require: $regs\_reuse \geq MIN\_REGS\_REUSE$

The first block applies a set of straightforward checks. Line one verifies that the thread grid does not exceed the maximum number of threads. Line two checks if the thread grid is divisible into warps. Line three checks if the thread grid can be used (regardless of its shape) to read a stripe of A, without any threads being idle. Similarly, line 4 checks if the thread grid can be used (regardless of its shape) to read a stripe of B, without any threads being idle.

The second block enforces the minimum occupancy based on shared memory consumption. First, the amount of shared memory required by the kernel is calculated. This equals the amount of shared memory required to store a stripe of A and a stripe of B (Algorithm 4, lines 2, 3, and 14, 15). Then, the number of possible thread blocks per multiprocessor is calculated and filtered though the hardware maximum. Next, the number of warps is calculated and also filtered through the hardware maximum. Finally, the number of possible blocks per multiprocessor and the number of possible threads per multiprocessor are recalculated and checked against the corresponding performance guidelines. Admittedly, some checks are redundant.

TABLE 1
Search Space Generator Constraints for Each Kernel Type

kernel type	min. occupancy[a]	min. reg. reuse[b]	min. no. blocks[c]
SGEMM	512	3.0	2
CGEMM	512	5.0	2
DGEMM	512	2.0	2
ZGEMM	512	2.0	2

[a] *minimum number of threads.*
[b] *minimum ratio of load instructions to fused multiply-add instructions in the innermost loop.*
[c] *minimum number of thread blocks per multiprocessor.*

The third block performs similar checks with respect to the register consumption. First, the number of registers required by the kernel is calculated. This equals the number of registers to "prefetch" odd iteration A and B (Algorithm 4, lines 6, 7), stream in even iteration A and B (lines 9, 10), and accumulate the results in C (line 11). What follows closely resembles the preceding block. The number of possible thread blocks per multiprocessor is calculated and filtered through the hardware maximum. Next, the number of warps is calculated and also filtered through the hardware maximum. Finally, the number of possible blocks per multiprocessor and the number of possible threads per multiprocessor are recalculated and checked against the corresponding performance guidelines. It has to be pointed out that the approach is heuristic. When compiled, the code will use more registers. (Registers will be used for local variables, loop counters, etc.)

It has to be pointed out that formulas for resource usage are only approximations. Specifically, the formula for register usage specifies only the minimum number of registers necessary for the double-buffered loop to function efficiently. It is almost guaranteed to be the lower bound on the actual register usage of the compiled kernel. This is in perfect alignment with the autotuning and pruning philosophy. The idea is to only eliminate kernels, which are certain to perform poorly. Some kernels with the actual register usage beyond what was anticipated will make it to the benchmarking phase, but will perform poorly, and not be picked as optimum. The point is that the elimination process is permissive rather than restrictive.

The last block simply calculates the ratio of loads to FMAs in the innermost loops (Algorithm 4, lines 9-11) and checks it against the performance guideline. The conditional takes into account the different ratio of memory operations to computation for real arithmetic and complex arithmetic. (Complex arithmetic is twice as compute intensive as real arithmetic.)

7 Results and Discussion

7.1 Generation Results

Table 1 shows the performance guidelines applied. The values were chosen experimentally to produce the number of combinations for each of the 16 versions of the GEMM to be on the order of hundreds. Notably, the minimum occupancy of 512 threads per multiprocessor (0.33) was always used and the minimum number of blocks per multiprocessor of two. The only parameter that varied was the register reuse, ranging from two to five. (Integer values were used, although in principle, the ratio is a floating-point number.)

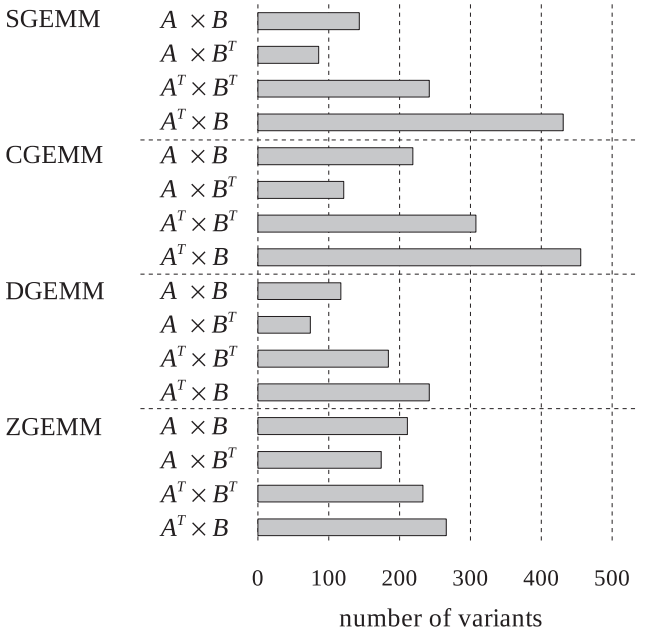

Fig. 7. Number of variants generated for each kernel type under the constraints listed in Table 1.

Fig. 7 shows the number of combinations produced for each of the 16 versions of the GEMM when the guidelines from Table 1 are applied. The number varies from slightly below 100 to slightly above 400. It should be noted that the choice of performance guidelines and the number of combinations produced is an arbitrary decision, which trades off the range of the search sweep with the time required to perform the sweep.

The pruning of the search space is a powerful and necessary mechanism here. For instance, with the performance guidelines taken away (and only hardware and implementation constraints applied) the generator will produce slightly more than one million combinations of SGEMM $A \times B$. Take another example, using the guidelines from Table 1 for CGEMM $A \times B$, but changing the minimum register reuse from 5.0 to 6.0 will create only six combinations, which do not include the fastest performing one. As a general observation, the generator produces many combinations with very good characteristics, in terms of occupancy and register reuse, which turn out not to perform the fastest. This strengthens the hypothesis that autotuning is a necessary component of GPU code development.

7.2 Selection Results

With all combinations generated, the next steps are runs, performance measurements, and selection of the fastest kernels. A few words about the hardware/software setup are in place here. The process of autotuning was conducted and the final performance results were produced on an NVIDIA Tesla S2050 system, with the Fermi GPU containing 14 multiprocessors and clocked at 1.147 GHz. CUDA SDK 4.0 release candidate 11 was used, the newest version at the time of the experiments. Square matrices A, B, and C were used in all cases and problem sizes were chosen such

TABLE 2
Autotuning Summary: Parameters and Performance of the Fastest Kernels

kernel	type	tiling	thread arrangement			performance	previously[a]
			compute C	load A	load B	[Gflop/s]	[Gflop/s]
SGEMM	$A \times B$	$96 \times 96 \times 16$	16×16	32×8	8×32	654	650
	$A \times B^T$	$96 \times 96 \times 16$	16×16	32×8	32×8	662	641
	$A^T \times B^T$	$96 \times 96 \times 16$	16×16	16×16	32×8	657	650
	$A^T \times B$	$96 \times 96 \times 16$	16×16	16×16	16×16	650	650
CGEMM	$A \times B$	$64 \times 64 \times 16$	16×16	32×8	16×16	804	778
	$A \times B^T$	$64 \times 64 \times 16$	16×16	16×16	16×16	804	783
	$A^T \times B^T$	$64 \times 64 \times 16$	16×16	16×16	32×8	805	792
	$A^T \times B$	$64 \times 64 \times 16$	16×16	16×16	16×16	804	782
DGEMM	$A \times B$	$64 \times 64 \times 16$	16×16	16×16	16×16	300	303
	$A \times B^T$	$64 \times 64 \times 16$	16×16	16×16	16×16	301	303
	$A^T \times B^T$	$64 \times 64 \times 16$	16×16	16×16	16×16	300	303
	$A^T \times B$	$64 \times 64 \times 16$	16×16	16×16	16×16	300	302
ZGEMM	$A \times B$	$24 \times 16 \times 8$	8×8	8×8	8×8	340	306
	$A \times B^T$	$16 \times 24 \times 8$	8×8	8×8	8×8	340	308
	$A^T \times B^T$	$16 \times 24 \times 8$	8×8	4×16	8×8	341	306
	$A^T \times B$	$24 \times 16 \times 8$	8×8	8×8	8×8	340	304

[a]MAGMA or CUBLAS (whichever is faster)

that all data would occupy 1 GB of the GPU memory. This results in the dimensions of 10,000 for SGEMM, 8,000 for CGEMM and DGEMM, and 6,000 for ZGEMM. Three runs were made for each case and the maximum performance taken. This was more of a precaution than an actual need, since performance fluctuation was virtually inexistent. With roughly 3,000 cases to run, the process takes 1 day on a single GPU.

The timing runs confirm that the generator with the pruning mechanism could not be a selection tool on its own. As was already mentioned, using strict performance guidelines does not result in converging on the fastest case. Although, under the constraints used, all tested kernels are good candidates for fast kernels, their performance can vary wildly. Here, ZGEMM showed the smallest performance variation. The slowest of all ZGEMM kernels ran at 180 Gflop/s, which is slightly more than half of the speed of the fastest, running at 340 Gflop/s. At the same time, the slowest SGEMM kernel ran at 64 Gflop/s which is less than 10 percent of the speed of the fastest one, running at 662 Gflop/s.

Table 2 shows the final selection of the fastest kernels. It needs to be pointed out that for each case there was a large number of kernels with performance very close to the fastest one (sometimes a couple of kernels within one percent). Here, simply the fastest one in each case is reported. The table shows a comparison against CUBLAS and MAGMA (whichever was faster for each case). Small improvements can be seen in almost all cases. Significant improvement can be observed for ZGEMM. While for CUBLAS and MAGMA ZGEMM runs only as fast as DGEMM, the new ZGEMM runs substantially faster. The autotuning process revealed ZGEMM kernels that successfully take advantage of its higher computational intensity versus DGEMM. One distinct feature of the ZGEMM kernels is that, unlike for all other cases, the tiles of the C matrix are not square. This could be one reason that someone coding the kernels by hand would not explore the case.

Fig. 8 shows the autotuning sweep for the $A \times B$ ZGEMM (NoTrans, NoTrans). The flat line at 306 Gflop/s shows the performance of the equivalent CUBLAS kernel. The autotuning run identifies 25 variants with higher performance. Some of them tile the C matrix in square tiles ($16 \times 16, 24 \times 24, 32 \times 32$). Most of them, however, tile the C matrix in rectangular tiles with dimensions taking values of: 8, 16, 24, 32, 48, and 64. The two spikes on the left side of the chart identify the fastest variants, one corresponding to tiling of 24×16, the other corresponding to tiling of 16×24. The former is faster by a fraction of a Gflop/s.

One reason for concern could be the fact that the timing part of the process was performed for specific (large) sizes. One could speculate that the kernels are tuned specifically for these sizes and perform suboptimally for other sizes. Just to make sure that this is not the case, the performance for the chosen kernels was measured across all matrix sizes. Fig. 9 shows the results. Square matrices are used with sizes corresponding to the tiling factor, 96 for SGEMM, 64 for CGEMM and DGEMM, and 48 for ZGEMM. Here it was considered pointless to time cases with partially filled border tiles. Generally, the impact on performance is

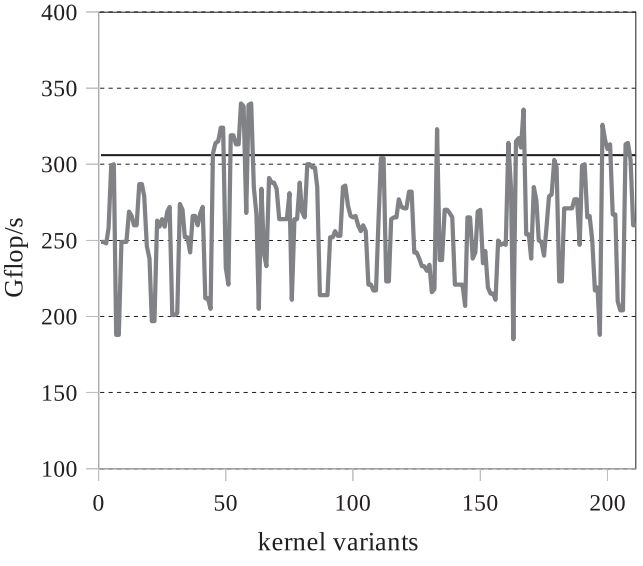

Fig. 8. ZGEMM autotuning sweep.

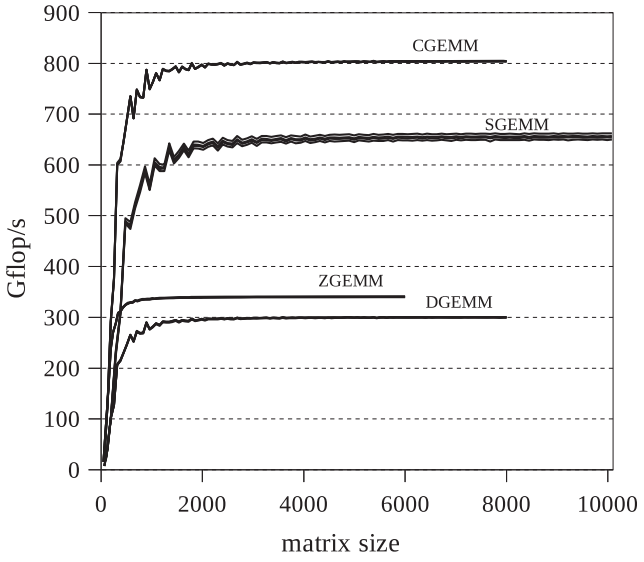

Fig. 9. GEMM Performance on a 1.147 GHz Fermi GPU.

negative, but negligible. (Very efficient method of dealing with such scenarios was designed by Nath et al. [29].) Fig. 9 shows clearly that the kernels perform consistently across all problem sizes, rising quickly to asymptotic performance, with the usual jitter at the beginning (more prominent for the more bandwidth-limited cases of single precision SGEMM and CGEMM).

8 Conclusions

It is the author's belief that this work provides strong evidence that autotuning is a crucial component in GPU code development. The essential component in this process is the capability of generating and effectively pruning the search space. For matrix multiplication, pruning turned out to be straightforward with the use of hardware and implementation constraints and constraints referred to as performance guidelines, such as minimum required occupancy. The choice of constraints allows for trading the thoroughness of the search with its duration.

The process can be generalized to other types of workloads, including more complex kernels and more bandwidth-bound kernels. In principle, this should be the case as long as the code can be parameterized and its properties, such as demand for registers and shared memory, are expressed as functions of the parameters.

It came as a surprise that, this late into the process of BLAS development for the Fermi architecture, an autotuning process managed to prove superior to hand-tuned codes. Although significant, the performance improvements were not dramatic. Hopefully, the system will show its power with the appearance of new architectures and also as a platform for the development of more complex kernels.

9 Future Plans

There are areas where the usefulness of the system is immediately applicable. The generation process revealed a huge number of kernels with much smaller tiling factors, performing nearly as good as kernels with larger tiling factors. The kernels with smaller tiles can readily replace the other kernels for smaller matrix sizes, where the use of large tiles limits the amount of parallelism, preventing the device from achieving good performance. For instance, for the DGEMM $A \times B$ operation, the fastest kernel uses tiles of 64×64 and asymptotically achieves the performance of 300 Gflop/s. The autotuning process revealed a kernel that uses tiles of 32×32 and asymptotically achieves the performance of 286 Gflop/s. For small matrix sizes the use of the latter kernel will quadruple parallelism at the loss of 5 percent of asymptotic kernel performance. Depending on the problem size, this can result in a huge performance gain.

Another opportunity presents itself where one dimension of the operation is significantly smaller than the other. MAGMA is a great example here. For instance, in the right-looking LU factorization, the GEMM is called with the dimension $K = 64$, which is much smaller than M and N. This causes the default GEMM kernel to only achieve 253 Gflop/s instead of the asymptotic 300 Gflop/s. When tuned for this shape, a kernel was found by the autotuner that delivers 268 Gflop/s. If K is further reduced to 32, the default kernel's performance drops to 204 Gflop/s, while the autotuner is capable of finding a kernel that delivers 242 Gflop/s for that case.

10 Software

Ultimately the software will be distributed as part of the MAGMA project (http://icl.cs.utk.edu/magma/). Initial prototype snapshots will be posted on the authors' websites (http://icl.cs.utk.edu/people/). All code will be released under the modified BSD license.

Acknowledgments

This work was supported by DOE grant #DE-SC0003852, "Architecture-aware Algorithms for Scalable Performance and Resilience on Heterogeneous Architectures," DOE grant #DE-SC0004983, "Matrix Algebra for GPU and Multicore Architectures (MAGMA) for Large Petascale Systems," Georgia Institute of Technology subcontract #RA241-G1 funded by NSF grant #OCI-0910735, "Keeneland: National Institute for Experimental Computing." The authors would like to thank David Luebke, Steven Parker, and Massimiliano Fatica for their insightful comments about the Fermi architecture.

References

[1] "Advanced Micro Devices, Inc," *AMD Intermediate Language, Version 2.0e,* http://developer.amd.com/gpu/AMDAPPSDK/assets/AMD_Intermediate_Language _(IL)_Specification_v2.pdf, 2010.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects," *J. Physics: Conf. Series,* vol. 180, no. 1, 2009, doi: 10.1088/1742-65 96/180/1/012037.

[3] 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, http://www.netlib.org/lapack/lug/, 1992.

[4] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, and E.S. Quintana-Ortí, "Evaluation and Tuning of the Level 3 CUBLAS for Graphics Processors," *Proc. Int'l Workshop Parallel and Distributed Scientific and Eng. Computing (PDSEC '08),* Apr. 2008, http://www.dicc-cid.uji.es/InfTec/reports/TR-01-01-2008.pdf.

[5] R.F. Barrett, T.H.F. Chan, E.F. D'Azevedo, E.F. Jaeger, K. Wong, and R.Y. Wong, "Complex Version of High Performance Computing LINPACK Benchmark (HPL)," *Concurrency and Computation: Practice and Experiences,* vol. 22, no. 5, pp. 573-587, 2009, doi: 10.1002/cpe.1476.

[6] "Basic Linear Algebra Technical Forum," *Basic Linear Algebra Technical Forum Standard,* http://www.netlib.org/blas/blast-forum/blas-report.pdf, Aug. 2001.

[7] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C.-W. Chin, "LAPACK Working Note 111: Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology," Technical Report UT-CS-96-326, Computer Science Department, Univ. of Tennessee, http://www.netlib.org/lapack/lawnspdf/lawn111.pdf, 1996.

[8] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J.J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, *ScaLAPACK Users' Guide,* SIAM, http://www.netlib.org/scalapack/slug/, 1997.

[9] C. Chen, J. Chame, and M. Hall, "CHiLL: A Framework for Composing High-Level Loop Transformations," Technical Report 08-897, Computer Science Department, Univ. of Southern California, 2008. http://www.cs.usc.edu/research/08-897.pdf.

[10] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache, "Facilitating the Search for Compositions of Program Transformations," *Proc. Int'l Conf. Supercomputing (ICS '05),* pp. 151-160, June 2005, doi: 10.1145/1088149.108816 9.

[11] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng, "Automatic Library Generation for BLAS3 on GPUs," *Proc. IEEE 25th Int'l Parallel and Distributed Processing Symp.,* May 2011.

[12] J. Dongarra et al., "The Int'l Exascale Software Roadmap," *Int'l J. High Performance Computer Applications,* vol. 25, no. 1, 2011.

[13] J.J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: Past, Present and Future," *Concurrency Computation: Practice and Experience,* vol. 15, no. 9, pp. 803-820, 2003.

[14] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-Platform GPU Programming," Technical Report CS-10-656, Electrical Eng. and Computer Science Department, Univ. of Tennessee, LAPACK Working Note 228, 2010.

[15] A.T. Fam, "Efficient Complex Matrix Multiplication," *IEEE Trans. Computers,* vol. 37, no. 7, pp. 877-879, July 1988, doi: 10.1109/12.2236.

[16] M. Frigo and S. Johnson, "The Design and Implementation of FFTW3," *Proc. IEEE,* vol. 93, no. 2, pp. 216-231, Feb. 2005, doi: 10.1109/JPROC.2004.8 40301.

[17] K. Goto and R.A. van de Geijn, "Anatomy of High-Performance Matrix Multiplication," *ACM Trans. Math. Software,* vol. 34, no. 3, pp. 1-25, 2008, doi: 10.1145/1356052.1356053.

[18] N.J. Higham, "Stability of a Method for Multiplying Complex Matrices with Three Real Matrix Multiplications," *SIAM. J. Matrix Analysis and Applications,* vol. 13, no. 3 pp. 681-687, 1992.

[19] C. Jiang and M. Snir, "Automatic Tuning Matrix Multiplication Performance on Graphics Hardware," *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '05),* pp. 185-194, Sept. 2005, doi: 10.1109/PACT.2005.10.

[20] B. Kågström, P. Ling, and C. van Loan, "GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark," *ACM Trans. Math. Software,* vol. 24, no. 3, pp. 268-302, Sept. 1998.

[21] Khronos Group, *The OpenCL Specification, Version 1.1,* http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf. 2010.

[22] P. Kogge, "Exascale Computing Study: Technology Challenges in Achieving Exascale Systems," Technical Report 278, DARPA Information Processing Techniques Office, http://www.er.doe.gov/ascr/Research/CS/DARPA exascale-hardware.pdf, 2008.

[23] J. Kurzak, W. Alvaro, and J. Dongarra, "Optimizing Matrix Multiplication for a Short-Vector SIMD Architecture CELL Processor," *Parallel Computing,* vol. 35, no. 3, pp. 138-150, 2009, doi: 10.1016/j.parco.20 08.12.010.

[24] J. Kurzak, D.A. Bader, and J. Dongarra, *Scientific Computing with Multicore and Accelerators,* Chapman & Hall/CRC Computational Science Series. CRC Press, 2010.

[25] Y. Li, J. Dongarra, and S. Tomov, "A Note on Auto-Tuning GEMM for GPUs," *Proc. Int'l Conf. Computational Science (ICCS '09),* May 2009, doi: 10.1007/978-3-64 2-01970-8_89.

[26] D. Luebke and S. Parker, email communication.

[27] N. Nakasato, "A Fast GEMM Implementation on a Cypress GPU," *Proc. First Int'l Workshop Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS '10),* Nov. 2010, http://www.dcs.warwick.ac.uk/sdh/pmbs10/pmbs10/Workshop_Programme_file s/fastgemm.pdf.

[28] R. Nath, S. Tomov, and J. Dongarra, "Accelerating GPU Kernels for Dense Linear Algebra," *Proc. Int'l Meeting on High Performance Computing for Computational Science (VECPAR '10),* June 2010, doi: 10.1007/978-3-64 2-19328-6_10.

[29] R. Nath, S. Tomov, and J. Dongarra, "An Improved MAGMA GEMM for Fermi Graphics Processing Units," *Int'l J. High Performance Computing Application,* vol. 24, no. 4, pp. 511-515, 2010, doi: 10.1177/1094342010385 729.

[30] E. Normand, "Single Event Upset at Ground Level," *IEEE Trans. Nuclear Science,* vol. 43, no. 6, pp. 2742-2750, Dec. 1996, doi: 10.1109/23.5568 61.

[31] Nvidia, *NVIDIA CUDA C Programming Guide, Version 3.2,* http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf. 2010.

[32] NVIDIA Corporation, *NVIDIA's Next Generation CUDA Compute Architecture: Fermi, Version 1.1,* http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[33] NVIDIA Corporation, *NVIDIA GF100, World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism, Version 1.5,* http://www.nvidia.com/object/IO_89569.html, 2010.

[34] NVIDIA Corporation, *PTX: Parallel Thread Execution ISA, Version 2.1,* http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/ptx_ isa_2.1.pdf, 2010.

[35] NVIDIA Corporation, *Tuning CUDA Applications for Fermi, Version 1.0,* http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVID IA_FermiTuningGuide.pdf, 2010.

[36] Open64, http://www.open64.net/, 2012.

[37] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *Proc. IEEE,* vol. 93, no. 2, pp. 232-275, Feb. 2005, doi: 10.1109/JPROC.2004.8 40306.

[38] G. Rudy, M.M. Khan, M. Hall, C. Chen, and J. Chame, "A Programming Language Interface to Describe Transformations and Code Generation," *Proc. 23rd Int'l Workshop Languages and Compilers for Parallel Computing (LCPC '10),* Oct. 2010, doi: 10.1007/978-3-64 2-19595-2_10.

[39] V. Sarkar (Editor & Study Lead), "Exascale Software Study: Software Challenges in Extreme Scale Systems," Technical Report 159, DARPA Information Processing Techniques Office, 2008, http://users.ece.gatech.edu/mrichard/ExascaleComputingStudy Reports/ECSS report 101909.pdf.

[40] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast Implementation of DGEMM on Fermi GPU," *Proc. IEEE/ACM Supercomputing Conference (SC '11),* Nov. 2011.

[41] V. Volkov and J.W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," *Proc. ACM/IEEE Conf. Supercomputing (SC '08),* Nov. 2008, doi: 10.1145/1413370.141340 2.

[42] R. Vuduc, J.W. Demmel, and K.A. Yelick, "OSKI: A Library of Automatically Tuned Sparse Matrix Kernels," *J. Physics: Conf. Series,* vol. 16, no. 16, pp. 521-530, 2005, doi: 10.1088/1742-6596/16/1/071.

[43] R.C. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," *Parallel Computing System Applications,* vol. 27, nos. 1/2, pp. 3-35, 2001, doi: 10.1016/S0167-81 91(00)00087-9.

[44] K. Yotov, X. Li, G. Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill, "Is Search Really Necessary to Generate High-Performance BLAS?" *Proc. IEEE,* vol. 93, no. 2, pp. 358-386, Feb. 2005, doi: 10.1109/JPROC.2004.8 40444.

Jakub Kurzak received the MSc degree in electrical and computer engineering from Wroclaw University of Technology, Poland, and the PhD degree in computer science from the University of Houston. He is a research director in the Innovative Computing Laboratory in the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville. His research interests include parallel algorithms, specifically in the area of numerical linear algebra, and also parallel programming models and performance optimization for parallel architectures multicore processors and GPU accelerators. He is a member of the IEEE.

Stanimire Tomov received the MS degree in computer science from Sofia University, Bulgaria, and the PhD degree in mathematics from Texas A&M University. He is a research director in ICL and adjunct assistant professor in the EECS at UTK. His research interests include parallel algorithms, numerical analysis, and high-performance scientific computing (HPC). Currently, his work is concentrated on the development of numerical linear algebra software for emerging architectures for HPC. He is a member of the IEEE.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high-performance computers using innovative approaches; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's Award for Career Achievement; and in 2011 he was the recipient of the IEEE IPDPS 2011 Charles Babbage Award. He is a fellow of the AAAS, ACM, and SIAM; a member of the National Academy of Engineering; and a life fellow of the IEEE.