Autotuning GEMMs for Fermi *

Jakub Kurzak
Electrical Engineering
and Computer Science
University of Tennessee
Knoxville, TN, USA
kurzak@eecs.utk.edu

Stanimire Tomov
Electrical Engineering
and Computer Science
University of Tennessee
Knoxville, TN, USA
tomov@eecs.utk.edu

Jack Dongarra
Electrical Engineering
and Computer Science
University of Tennessee
Knoxville, TN, USA
dongarra@eecs.utk.edu

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 article 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.

Categories and Subject Descriptors

G.4 [**MATHEMATICAL SOFTWARE**]: Parallel and vector implementations; F.2.1 [**Numerical Algorithms and Problems**]: Computations on matrices; C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms

Performance

Keywords

automatic generation, tuning, matrix multiplication, accelerator, GPU, CUDA, BLAS

*This work was supported by the U. S. Department of Energy, the National Science Foundation, NVIDIA and Microsoft.

1. INTRODUCTION

Graphics Processing Units (GPUs) maintain a strong lead over more traditional multicore CPUs in peak floating-point performance and memory bandwidth [27], 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 [11, 19, 35]. Today, many key scientific and engineering applications rely on GPUs to deliver performance in excess of what standard multicores are capable of [20].

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 [27]. Although a large body of scientific and engineering workloads deal with sparse systems, such as those produced by unstructured irregular meshes of finite element simulations, dense systems are also used in these key areas: nuclear fusion, material science and radar cross-section, just to name a few.

The hardware target of this article are the NVIDIA GF100 (Fermi) architecture GPUs [28, 29, 31], 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. NVIDIA's *Compute Unified Device Architecture* (CUDA) [27] is the programming environment of choice here. The OpenCL standard [18] could be used as an alternative, but currently its available implementations are known to lag behind CUDA in performance [13].

The workload implemented here is general matrix multiplication, referred to as *GEMM*, following the *Basic Linear Algebra Subroutines* (BLAS) standard [5]. The GEMM routine is a building block of software packages such as LAPACK [3] and ScaLAPACK [7], absolutely essential to their performance, and can also be used as the basis for implementing all other Level-3 BLAS routines [17].

Not without significance is the fact that GEMM is also critical to the performance of the *High Performance Linpack Benchmark* (HPL) [12], used to rate the systems on the Top500 list of the fastest (disclosed) computers in the world. Currently, the top spot is occupied by the Tianhe-1A supercomputer in China, a hybrid system based on Intel Xeon processors and NVIDIA Fermi 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]. The system is further referred to as *BACUGen*, standing for *A×B (BA) CUDA Generator*, with intentional similarity to the popular *Bakugan Battle Brawlers* anime television series.

2. 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 [4].

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 was in high demand.

It is of big significance that the high level abstraction of CUDA maps very well to the hardware architectures of NVIDIA GPUs. It is remarkable that programmers never resorted to lower level abstractions, such as the *Parallel Thread Execution* (PTX) [30] (the pseudo-assembly 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 ideal targets for autotuning efforts.

3. RELATED WORK

The list of prominent autotuning software projects includes packages such as: *Automatically Tuned Linear Algebra Software* (ATLAS) [38], and its predecessor *Portable High Performance ANSI C* (PHiPAC) [6], *Optimized Sparse Kernel Interface* (OSKI) [37], *Fastest Fourier Transform in the West* (FFTW) [15] and *SPIRAL* [33] (code generation for digital signal processing transforms). All these projects address autotuning for standard processors (not accelerators).

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 [36]. Similar efforts followed in the MAGMA project [21]. The introduction of the NVIDIA Fermi architecture triggered the development of MAGMA GEMM kernels tuned for that architecture [24, 25]. 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 [14], which results in up to 25 % decrease in the number of floating-point operations [25]. 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" [16]. Although, Higham also notes that "if the errors are measured relative to $\|A\|\,\|B\|$ [...], then they are just as small as for conventional multiplication" [16]. 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.

Recently, Nakasato presented GEMM kernels for the Cypress GPU from ATI [23]. 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].

An important approach to the development of optimized GEMM routines is code generation through compiler transformations. Rudy et al. [34] 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 [8]. 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. [10] presented a similar system built using the Open64 compiler [32] and the WRaP-IT/URUK/URGenT polyhedral toolchain [9]. 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.

4. 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, non-transposed 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* (SIMT) GPU programming model.

Finally, the desired product 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).

5. SOLUTION

5.1 Hardware Target

A number of articles are available with the details of the Fermi architecture [28, 29, 31] 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* (MAD) 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 [26]. 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 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 [22].

5.2 Universal GEMM Stencil

5.2.1 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 Figure 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 two dimensional 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 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.

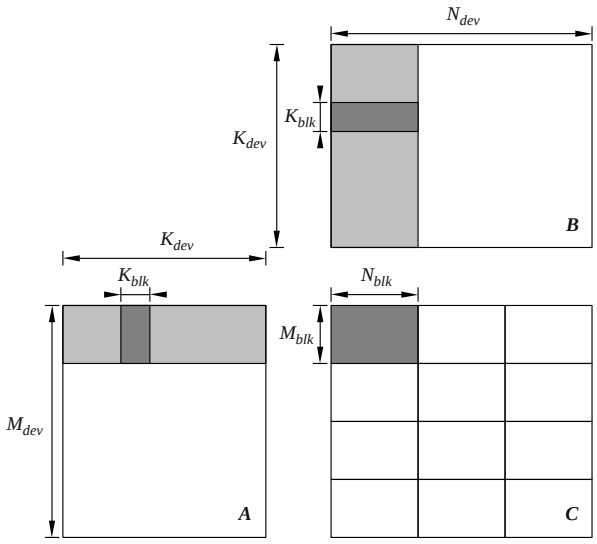

Figure 1: GEMM at the device level.

Figure 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.

Figure 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.)

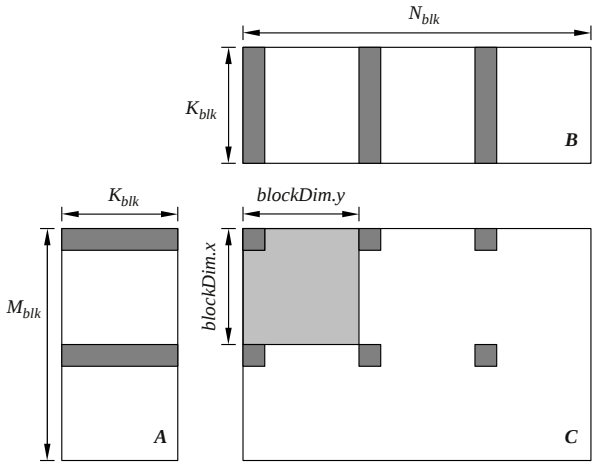

Figure 2: GEMM at the block level.

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 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.

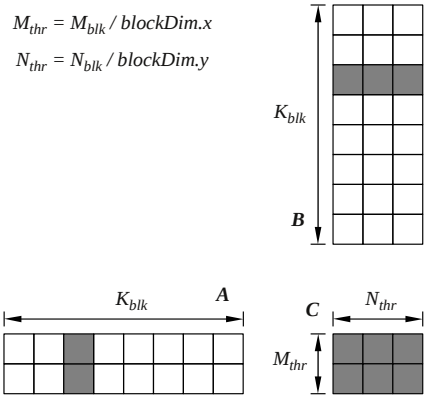

Figure 3: GEMM at the thread level.

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 [23, 36]. 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 immensely simplifies the coding of different *transposed/non-transposed* scenarios).

5.2.2 Pipelined Loop

Algorithm 1 shows the pseudocode for the generic GEMM stencil. (Adherence to any strict notation is traded for clarity here.) The code follows the classic pipelined loop scheme with a prologue and epilogue. The pipelined loop body performs odd iteration loads of A and B from the device memory to registers (lines 6, 7), "consumes" even iteration A and B in shared memory (lines 8-12) and drops odd iteration A and B to shared memory. 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. [25] proves to be the fastest.

Algorithm 1 Generic GEMM stencil pseudocode.

```
 1: C_regs ⇐ 0
 2: A_0 dev ⇒ shmem
 3: B_0 dev ⇒ shmem
 4: __syncthreads();
 5: for K = 0 to K_dev − K_blk step K_blk do
 6:    A_odd dev ⇒ regs
 7:    B_odd dev ⇒ regs
 8:    for k = 0 to K_blk step 1 do
 9:       A_even[k] shmem ⇒ regs
10:       B_even[k] shmem ⇒ regs
11:       C = C + A_even[k] × B_even[k]
12:    end for
13:    __syncthreads();
14:    A_odd regs ⇒ shmem
15:    B_odd regs ⇒ shmem
16:    __syncthreads();
17: end for
18: for k = 0 to K_blk step 1 do
19:    A_odd[k] shmem ⇒ regs
20:    B_odd[k] shmem ⇒ regs
21:    C = C + A_even[k] × B_even[k]
22: end for
23: C_dev = alpha × C_regs + beta × C_dev
```

regs - registers
shmem - shared memory
dev - device memory
odd - even iteration
even - odd iteration

5.2.3 Parametrization

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 to 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.

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 parametrized (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 (Figure 4). 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}.

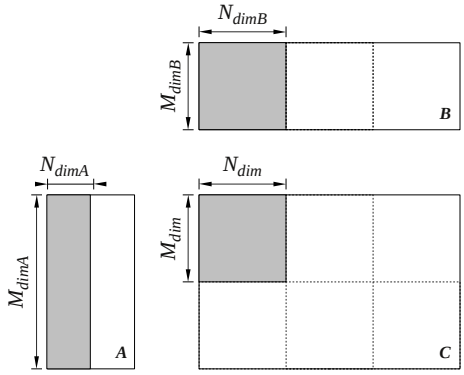

Figure 4: Reshaping the thread block for reading A and B.

5.3 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 9-dimensional parameter space is enormous. Constraints come from a few different sources. Here the following categories of constraints are identified: *queryable hardware constraints, non-queryable hardware constraints, hard implementation constraints and soft implementation constraints.*

5.3.1 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). Non-queryable 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"* [27], 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.

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

5.3.2 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 states the minimum number of threads per multiprocessor that the kernel has to allow for. 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 1536) 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 1, lines 8-12).

5.3.3 Generation and Pruning

Algorithm 2 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 2 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 × N_dimA == M_dim × N_dim
            M_dimA % M_blk == 0
            N_dimA % K_blk == 0
            M_dimB × N_dimB == M_dim × N_dim
            M_dimB % K_blk == 0
            N_dimB % N_blk == 0
          end if

        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 3 shows the set of constraints enforced inside the nested loops of Algorithm 2. 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.

The first block applies a set of straightforward checks. Line one verifies if the thread grid does not exceed the maximum number of threads. Line two checks if the 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 four 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 1, 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.

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 1, 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 heristic. When compiled, the code will use more registers. (Registers will be used for local variables, loop counters, etc.)

The last block simply calculates the ratio of loads to FMAs in the innermost loops (Algorithm 1, 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.)

6. RESULTS AND DISCUSSION

6.1 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

Algorithm 3 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: $threads\_per\_sm \geq MIN\_THREADS\_PER\_SM$

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

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.)

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

Figure 5 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 one hundred to slightly above four hundred. 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.

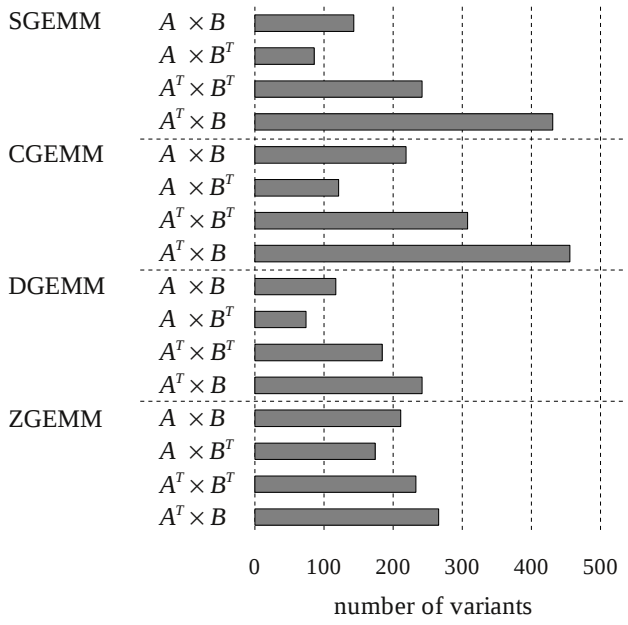

Figure 5: Number of variants generated for each kernel type under the constraints listed in Table 1.

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 6 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.

6.2 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 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 three thousand cases to run, the process takes one 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 % 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 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.

One reason for concern could be the fact that the timing part of the process was performed for specific (large) sizes.

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)

One could speculate that the kernels are tuned specifically for these sizes and performs 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. Figure 6 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 negative, but negligible. (Very efficient method of dealing with such scenarios was designed by Nath et al. [25].) Figure 6 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).

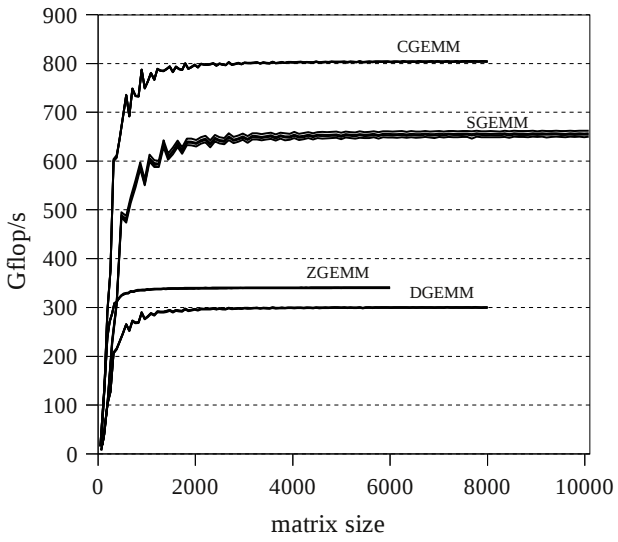

Figure 6: GEMM Performance on a 1.147 GHz Fermi GPU

7. CONCLUSIONS

It is the authors belief that this work proves 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.

It also is the authors belief that the process can be easily 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, 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.

8. 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 % 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.

9. ACKNOWLEDGMENTS

The authors would like to thank David Luebke, Steven Parker and Massimiliano Fatica for their insightful comments about the Fermi architecture.

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.

References

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

[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. Phys.: Conf. Ser.*, 180(1), 2009. DOI: 10.1088/1742-6596/180/1/012037.

[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, Philadelphia, PA, 1992. `http://www.netlib.org/lapack/lug/`.

[4] 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 Computat.: Pract. Exper.*, 22(5):573–587, 2009. DOI: 10.1002/cpe.1476.

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

[6] 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, University of Tennessee, 1996. `http://www.netlib.org/lapack/lawnspdf/lawn111.pdf`.

[7] 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, Philadelphia, PA, 1997. `http://www.netlib.org/scalapack/slug/`.

[8] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, Computer Science Department, University of Southern California, 2008. `http://www.cs.usc.edu/research/08-897.pdf`.

[9] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache. Facilitating the search for compositions of program transformations. In *Proceedings of the International Conference on Supercomputing 2005, ICS'05*, pages 151–160, Cambridge, MA, June 18-21 2005. ACM. DOI: 10.1145/1088149.1088169.

[10] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng. Automatic library generation for BLAS3 on GPUs. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium*, Anchorage, AK, May 16-20 2011. IEEE.

[11] J. Dongarra, P. Beckman, et al. The international exascale software roadmap. *Int. J. High Perf. Comput. Applic.*, 25(1), 2011. ISSN: 1094-3420 (to appear).

[12] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present and Future. *Concurrency Computat.: Pract. Exper.*, 15(9):803–820, 2003.

[13] 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 Engineering and Computer Science Department, University of Tennessee, 2010. LAPACK Working Note 228.

[14] A. T. Fam. Efficient complex matrix multiplication. *IEEE Trans. Comput.*, 37(7):877–879, 1988. DOI: 10.1109/12.2236.

[15] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. DOI: 10.1109/JPROC.2004.840301.

[16] N. J. Higham. Stability of a method for multiplying complex matrices with three real matrix multiplications. *SIAM. J. Matrix Anal. Appl.*, 13(3):681–687, 1992.

[17] B. Kågström, P. Ling, and C. van Loan. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

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

[19] P. Kogge (Editor & Study Lead). Exascale computing study: Technology challenges in achieving exascale systems. Technical Report 278, DARPA Information Processing Techniques Office, 2008. http://www.er.doe.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf.

[20] J. Kurzak, D. A. Bader, and J. Dongarra, editors. *Scientific Computing with Multicore and Accelerators*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2010. ISBN: 1439825365.

[21] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 2009 International Conference on Computational Science, ICCS'09*, Baton Roube, LA, May 25-27 2009. Lecture Notes in Computer Science 5544:884-892. DOI: 10.1007/978-3-642-01970-8_89.

[22] D. Luebke and S. Parker. email communication.

[23] N. Nakasato. A fast GEMM implementation on a Cypress GPU. In *1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems, PMBS'10 (held as part of SC'10)*, New Orleans, LA, November 13-19 2010. http://www.dcs.warwick.ac.uk/~sdh/pmbs10/pmbs10/Workshop_Programme_files/fastgemm.pdf.

[24] R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU kernels for dense linear algebra. In *Proceedings of the 2010 International Meeting on High Performance Computing for Computational Science, VECPAR'10*, Berkeley, CA, June 22-25 2010. Lecture Notes in Computer Science 6449:83-92. DOI: 10.1007/978-3-642-19328-6_10.

[25] R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi graphics processing units. *Int. J. High Perf. Comput. Applic.*, 24(4):511–515, 2010. DOI: 10.1177/1094342010385729.

[26] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, 1996. DOI: 10.1109/23.556861.

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

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

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

[30] NVIDIA Corporation. *PTX: Parallel Thread Execution ISA, Version 2.1*, 2010. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/ptx_isa_2.1.pdf.

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

[32] Open64. http://www.open64.net/.

[33] 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. *Proceedings of the IEEE*, 93(2):232–275, 2005. DOI: 10.1109/JPROC.2004.840306.

[34] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing, LCPC'10*, Houston, TX, October 7-9 2010. Lecture Notes in Computer Science 6548:136-150. DOI: 10.1007/978-3-642-19595-2_10.

[35] 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/ExascaleComputingStudyReports/ECSSreport101909.pdf.

[36] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC'08*, Austin, TX, November 15-21 2008. IEEE Press. DOI: 10.1145/1413370.1413402.

[37] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *J. Phys.: Conf. Ser.*, 16(16):521–530, 2005. DOI: 10.1088/1742-6596/16/1/071.

[38] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parellel Comput. Syst. Appl.*, 27(1-2):3–35, 2001. DOI: 10.1016/S0167-8191(00)00087-9.