CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Auto-Tuning of Level 1 and Level 2 BLAS for GPUs

Hans Henrik Brandenborg Sørensen

Informatics and Mathematical Modelling, Technical University of Denmark, Bldg. 321, DK-2800 Lyngby, Denmark

SUMMARY

The use of high performance libraries for dense linear algebra operations is of great importance in many numerical scientific applications. The most common operations form the backbone of the Basic Linear Algebra Subroutines (BLAS) library. In this paper, we consider the performance and auto-tuning of level 1 and level 2 BLAS routines on GPUs. As examples, we develop single-precision CUDA kernels for three of the most popular operations, the Euclidian norm (SNRM2), the matrix-vector multiplication (SGEMV), and the triangular solve (STRSV). The target hardware is the most recent Nvidia Tesla 20-series (Fermi architecture), which is designed from the ground up for high performance computing. We show that it is essentially a matter of fully utilizing the fine-grained parallelism of the many-core GPU in order to achieve high performance for level 1 and level 2 BLAS operations. We show that auto-tuning can be successfully employed to kernels for these operations so that they perform well for all input sizes. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: GPU, BLAS, Dense linear algebra, Parallel algorithms

1. INTRODUCTION

Advances in the architecture of graphical processing units (GPUs) have improved the performance and programmability of this hardware in such a way that it is commonly used for accelerating the execution of scientific applications. One of the areas of successful application is dense linear algebra. In particular, those applications that rely on the performance delivered by matrix-matrix multiplication are experiencing large speed-ups on GPUs because this operation makes it possible to utilize much of the hardware's computing capabilities. The most basic linear algebra operations, such as vector and matrix-vector operations, are limited in their performance by data-movement and therefore less well suited for acceleration on a GPU. However, since many scientific applications and numerical methods are based on these operations, it is still of great importance for developers and scientists to have high performance GPU implementations available.

Currently, in order to exploit the computing potential of GPUs, the programmer has to use either Nvidia's Compute Unified Device Architecture (CUDA) [1] or the Open Compute Language (OpenCL) [2]. Recent years have shown that many applications written in these languages, which fully utilize the hardware of the target GPU, show impressive performance speed-ups compared to the CPU. At the same time programmers working in these languages are also facing serious challenges. Most importantly, the time and effort required to program an optimized routine or application has also increased tremendously.

An essential prerequisite for many scientific applications is therefore the availability of high performance numerical libraries for linear algebra on dense matrices, such as the BLAS library [3]

*Correspondence to: Journals Production Department, John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK.

and the Linear Algebra Package (LAPACK) [4]. Several such libraries targeting Nvidia's GPUs have emerged over the past few years. Apart from Nvidia's own CUBLAS, which is part of the CUDA Toolkit [1], other prominent libraries are the open source Matrix Algebra on GPU and Multicore Architectures (MAGMA) library [5] and the commercial CUDA GPU-accelerated linear algebra (CULA) library [6]. These provide subsets of the functionality of BLAS/LAPACK for GPUs but are not mature in their current versions.

Automatic performance tuning, or auto-tuning, has been widely used to automatically create near-optimal numerical libraries for CPUs [7], e.g., in the famous the ATLAS package [8]. Modern GPUs offer the same complex and diverse architectural features as CPUs, which require nontrivial optimization strategies that often change from one chip generation to the next. Therefore, as already demonstrated in MAGMA [9], auto-tuning is also very compelling on GPUs.

In this paper, our goal is to design GPU kernels for level 1 and level 2 BLAS based on template parameters and auto-tune them for given problem sizes. To this end, we have implemented an auto-tuning framework that can automate the performance tuning process by running a large set of empirical evaluations to configure applications and libraries on the targeted computing platform. All the kernels that are candidates as the best kernel for a particular problem size, are thereby generated automatically by the compiler as templates, and do not need to be hand-coded by the programmer.

In this work, we target Nvidia GPUs, specifically the Tesla C2050 (Fermi architecture), which is designed from the outset for scientific computations. All kernels are made in the CUDA programming model (Toolkit 4.0), where the hardware is controlled by using blocks (3D objects of sizes $1024 \times 1024 \times 64$ containing up to 1024 threads) and grids (2D objects containing up to 65535×65535 blocks). Groups of 32 threads are called warps. We consider only the single-precision case and present the double-precision results in another work.

This paper is organized as follows. Sect. 2 states the performance considerations for our kernels and Sect. 3 discusses performance estimation. Next we outline the kernel designs in Sect. 4. In Sect. 5 we describe the auto-tuning process. The experimental results are presented in Sect. 6.

2. PERFORMANCE IN THE CUDA PROGRAMMING MODEL

When implementing a GPU kernel in CUDA, the path to high performance follows mainly two directions. The first is to maximize the instructions throughput and the second is to optimize the memory access patterns. Often one needs to focus attention only on one of these directions, depending on whether the maximum performance is bounded by the first or the latter.

2.1. *Memory Bound Kernels*

For our target GPU the single-precision peak performance is 1.03 Tflops/s and the theoretical memory bandwidth is 144 GB/s. On such hardware, the kernels we develop for vector and matrix-vector operations will consistently fall under the memory bound category. For example, a matrix-vector multiplication requires $N^2 + 2N$ memory accesses and $2N^2$ floating point operations. Since the resulting arithmetic intensity is much less than the perfect balance (~ 4.5 flops per byte) for the target GPU [10], the corresponding kernel is inherently memory bound for all N. This means that the arithmetic operations are well hidden by the latency of memory accesses, and we will concentrate on optimizing the memory access pattern in order to reach the maximum memory bandwidth.

2.2. *Coalesced Memory Access*

In general, two requirements must be fulfilled for a kernel's memory access to be efficient. First, the access must be contiguous so that consecutive threads within a warp (32 threads) always read contiguous memory locations. Second, the memory must be properly aligned so that the first data element accessed by any warp is always aligned on 128 bytes segments. This allows the kernel to read 32 memory locations in parallel as a single 128 byte memory access, a so-called coalesced access. If by design an algorithm is required to read data in a non-coalesced fashion, one can use the

DOI: 10.1002/cpe

shared memory available on the graphics card to circumvent such access patterns. Shared memory should also be used if data is to be re-used or communicated between threads within a block.

2.3. Registers

A key performance parameter in CUDA is the number of registers used per thread. The fewer registers a kernel uses, the more threads and blocks are likely to reside on a multiprocessor, which can lead to higher occupancy (the ratio of the resident warps on the hardware to the maximum possible number of resident warps). A large number of warps is often required to hide the memory access latencies well. However, since registers represents the major part of the per-multiprocessor memory and have the shortest access latency, it is advantageous to implement high-performance kernels for exhaustive register usage, if possible.

2.4. Loop Unrolling

It is important to design memory bound kernels with enough fine-grained thread-level parallelism (TLP) to allow for the occupancy to be high and latencies well hidden. Alternatively, having many independent instructions, i.e. high instruction level parallelism (ILP), can also hide the latencies [11]. In our kernels we allow the compiler to unroll inner loops (using the keyword `#pragma unroll`), which will increase the instruction level parallelism of the kernels and facilitate some degree of latency hiding. Unfortunately, this may also increase the register usage which may lower the occupancy and subsequently performance. The key is to find the best compromise between ILP, TLP, the number of threads per block, the register usage, and the shared memory usage to achieve the best performance of a given kernel. To this end, we will employ auto-tuning.

3. PERFORMANCE ESTIMATES OF MEMORY BOUND KERNELS

In this work we only consider the level 1 and level 2 BLAS operations, which are inherently memory bound with the maximum performance limited by the effective bandwidth achieved during execution. Therefore, the measurement of the effective bandwidth for a given input size is also the best metric for estimating the maximum performance that can be reached by the GPU kernels.

3.1. Theoretical Bandwidth

The theoretical limit to the maximum global memory bandwidth is always available as one of the key specifications of a graphics card or by calculating it using the memory clock rate and interface width [14]. For the C2050 card it is 144 GB/s. However, this specification is only relevant for ideal situations and for large memory chunk sizes. In general, we are interested in the effective bandwidth for all possible chunk sizes in order to be able to estimate the maximum performance for any input.

3.2. Effective Read and Write Bandwidths

The level 1 and level 2 BLAS routines have different requirements for reading and writing memory. For example the Euclidian norm (SNRM2) requires N elements to be read and one element to be written. We will therefore need to determine the read and the write effective bandwidths separately. In order to measure these values we make two simple kernels that either only reads data (SREAD) or only writes data (SMSET, i.e., setting memory to a constant value). Since there is no output in the read-only kernel, it has to be fitted with a conditional statement in such a way that the compiler does not optimize by removing all the kernel code.

We allow the SREAD kernel to access memory as doubles because on some GPUs this is slightly faster than floats. In addition, we will use an auto-tuning framework to make sure that all tunable performance parameters for the read and write kernels are optimal for a given input size. The auto-tuning framework and the tuning parameters used are described further in Sect. 5.

DOI: 10.1002/cpe

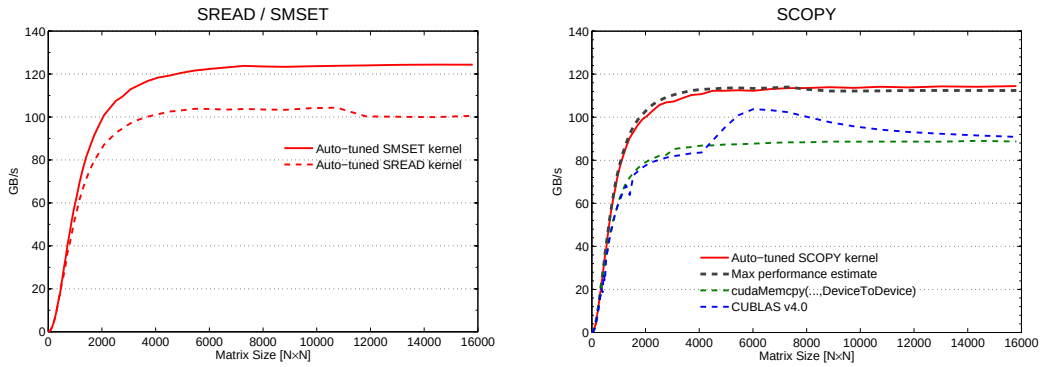

Figure 1. Left; the read (SREAD) and write (SMSET, i.e., memory set) effective bandwidth measured in GB/s using auto-tuned kernels for different size square matrices. Right; the measured performance and estimated maximum performance of the level 1 BLAS routine SCOPY using an auto-tuned kernel together with the performance of equivalent calls to `cudaMemcpy(...,DeviceToDevice)` and CUBLAS 4.0.

In the left part of Fig. 1, we show the measured effective bandwidth for reading and writing data on the Tesla C2050 card using auto-tuned kernels. The measurements show that the bandwidth increases as the input matrix size increases and that the maximum bandwidth in practice requires matrix sizes larger the 5000×5000. Also, we see that the effective bandwidth differs significantly between reading and writing, i.e., it is in fact about 20 percent faster to write than to read.

3.3. Performance Estimates

We are able to estimate the maximum performance of all the level 1 and level 2 BLAS operations using the measurements of the read and write bandwidths. We need only determine the number of elements read and written for the different operations and scale these by the measured bandwidths.

E.g., consider the SCOPY operation, which copies the elements of a vector to another position in memory. Since we are reading and writing exactly the same number of elements, the expression

$$\mathrm{B}_{\mathtt{SCOPY}} = (\mathrm{B}_{\mathtt{SREAD}} + \mathrm{B}_{\mathtt{SMSET}})/2, \tag{1}$$

where $\mathrm{B}_{\mathtt{X}}$ is the bandwidth of `X` in GB/s, is chosen as the performance estimate. In the right part of Fig. 1, we show this estimate together with the measurement of the actual performance in GB/s for the corresponding auto-tuned SCOPY kernel. For comparison, we also show the performance of corresponding calls to the CUDA function `cudaMemcpy(...,DeviceToDevice)` and to the SCOPY in CUBLAS 4.0. We see very good agreement between the estimate and the actual performance for the auto-tuned kernel. Moreover, we see that the performance of `cudaMemcpy()` and the CUBLAS 4.0 SCOPY is significantly lower than the maximum performance estimate.

Correspondingly, we can estimate the bandwidth achieved for other operations and in turn estimate the performance in Gflops/s. For example, the Euclidian norm (SNRM2), which requires 2 flops per vector element (approximately) with 4 bytes per element in single precision, is estimated as

$$\mathrm{P}_{\mathtt{SNRM2}} = (2/4) \times (N\mathrm{B}_{\mathtt{SREAD}} + \mathrm{B}_{\mathtt{SMSET}})/(N+1), \tag{2}$$

where $\mathrm{P}_{\mathtt{X}}$ is the performance of `X` in Gflops/s.

The matrix-vector multiplication (SGEMV) for an $M \times N$ matrix requires $MN + 2N$ elements to be read and M elements to be written, while doing 2 flops per matrix element, leading to

$$\mathrm{P}_{\mathtt{SGEMV}} = (2/4) \times ((MN+M)\mathrm{B}_{\mathtt{SREAD}} + N\mathrm{B}_{\mathtt{SMSET}})/(MN+M+N), \tag{3}$$

as the maximum performance estimate for SGEMV in single precision.

The triangular solve (STRSV) for an $M \times M$ matrix and one right-hand side requires $(M^2 + M)/2 + M$ elements to be read and M elements to be written, and again 2 flops per matrix element,

DOI: 10.1002/cpe

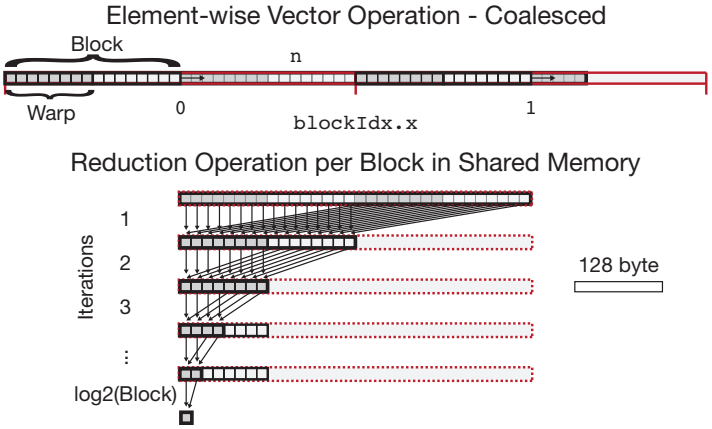

Figure 2. Coalesced access pattern for the element-wise operation on a vector and the subsequent parallel reduction operation per block in shared memory.

which yields

$$\mathrm{P}_{\mathrm{STRSV}} = (2/4) \times \left(\left((M^2 + M)/2 + M \right) \mathrm{B}_{\mathrm{SREAD}} + M \mathrm{B}_{\mathrm{SMSET}} \right) / \left((M^2 + M)/2 + 2M \right), \quad (4)$$

as the maximum performance estimate for STRSV in single precision.

4. LEVEL 1 AND LEVEL 2 BLAS OPERATIONS ON FERMI GPUS

In this section, we describe the design and implementation of level 1 and level 2 BLAS routines for the Tesla C2050 Fermi GPU. The Fermi architecture provides 14 multiprocessors of 32 cores each that can execute threads in parallel [12]. In CUDA, we utilize this parallel hardware by distributing the work of an operation to the individual threads via a grid of blocks. During execution, the blocks are assigned to multiprocessors, which further split each block into sets of 32 threads known as warps, that execute the same instructions on the multiprocessor synchronously.

4.1. Operations on a Vector

In Fig. 2, we illustrate the two main cases of operations on vectors, where the first corresponds to merely reading and writing a vector as required for an operation on the individual elements of the vector, e.g., a vector scale and vector add (SAXPY), and the second corresponds to the subsequent reduction operation in shared memory required for, e.g., a sum or an Euclidian norm (SNRM2).

In the first case, the operation is embarrassingly parallel and the memory access pattern for reading the vector in a CUDA kernel is made fully coalesced by having a block size given by a multiple of the warp size (assuming the vector is stored at an aligned address). Each thread can be assigned to handle one or more elements. Whether the values should be stored in shared memory for optimal usage depends on the operation to be performed on its elements.

In the second case, the illustrated access pattern for the reduction is designed to avoid shared memory bank conflicts [13]. It requires $log_2(\texttt{BLOCKSIZE})$ iterations, where the last 5 are executed synchronously per warp. Although, the reduction operation suggested here suffers from performance inhibiters like the use of explicit synchronizations and leaving threads idle in the last 5 iterations, this technique is currently the optimal for reducing a result on Fermi GPUs.

4.2. Operations on a Matrix

An important access pattern for operations on a matrix is when each thread transverses elements of a given row in order to operate on the individual elements of the row or to reduce a result or part

DOI: 10.1002/cpe

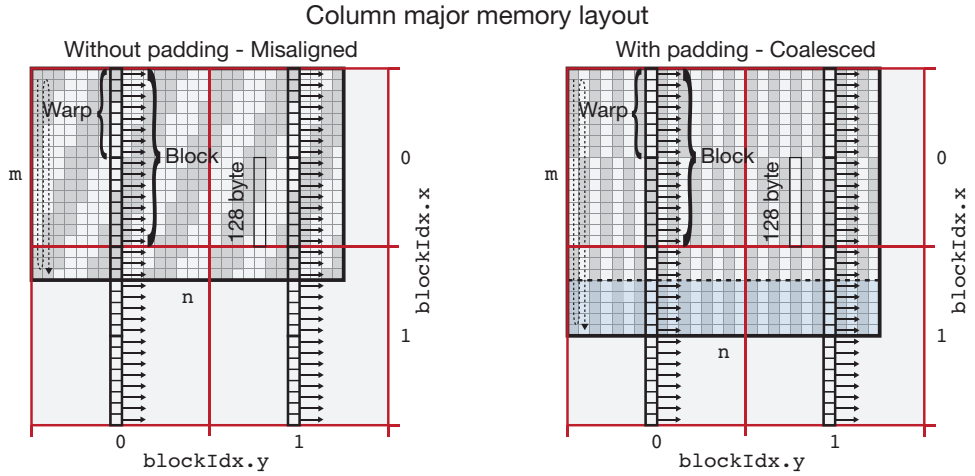

Figure 3. Access pattern for the row-wise reading of matrices having column major memory layout. Left; the misaligned case of arbitrary number of rows. Right; the coalesced case occurring when the number of rows is padded to a multiple of the warp size.

of a result. E.g., the typical parallel implementation of a matrix-vector multiplication (SGEMV), where each thread performs a dot product between one row of $\mathbf{A}$ and $\mathbf{x}$ to produce one element of the result $\mathbf{y}$. In the common case, where the matrix is stored in column major memory layout, this access pattern can be achieved by dividing the matrix into slices of `BLOCKSIZE` rows and launching a block for each of them. Each thread might only take care of part of a row, which is accomplished by dividing the matrix into tiles instead of slices and using a 2D grid of blocks.

As illustrated in the left part of Fig. 3, the described memory access pattern for an arbitrary $m \times n$ matrix is contiguous but misaligned for each warp (except for every 32nd column). In single precision, each warp of 32 threads will request $32 \times 4 = 128$ bytes of memory per memory access in the kernel. On a Fermi GPU with compute capability 2.0, the misalignment breaks the memory access of the required 128 bytes per warp into two aligned 128 byte segment transactions [14].

As illustrated in the right part of Fig. 3, for the matrix memory accesses to be coalesced, both the number of threads per block and the height of the matrix must be a multiple of the warp size. In particular, this means that a matrix whose height is not a multiple of the warp size will be accessed more efficiently if it is actually allocated with a height rounded up to the closest multiple of this size and its columns padded accordingly.

4.3. Operations on a Transposed Matrix

In BLAS, all level 2 routines for matrix-vector multiplication and linear solvers are available for ordinary as well as transposed matrices (by specifying the `TRANS` argument to `'T'`). The operations on transposed matrices can be advantageously implemented without explicit transpositions. Moreover, one can view the operations on transposed matrices stored in column major layout as equivalent to operations on ordinary matrices stored in row major layout.

In Fig. 4, we illustrate an access pattern for an operation that requires the row-wise transversal of matrix elements in row major memory layout, e.g., $\mathbf{y} = \mathbf{A}^{\mathrm{T}}\mathbf{x}$ (SGEMV with `'T'`). The threads in a block are distributed along the rows of the matrix in order for the memory access to be contiguous for each warp. For the case of arbitrary number of columns n (left part of the figure) the access will be misaligned. This can only be avoided if the width of the matrix is padded to a multiple of the warp size (right part of the figure).

In the access pattern shown, we designate one block per row and the threads have to work together if a reduction result is required, i.e., the row-wise transversal might be followed by a reduction in shared memory as discussed above. We allow each thread to take care of more elements on the same

DOI: 10.1002/cpe

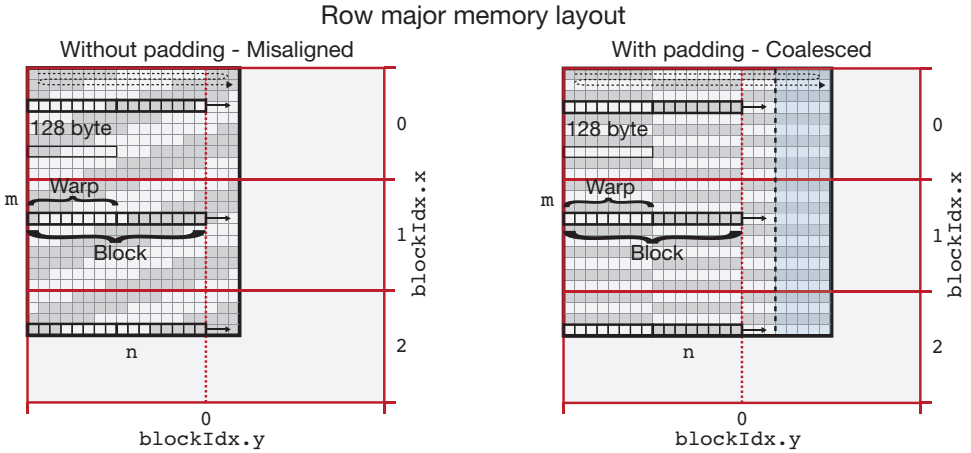

Figure 4. Access pattern for the row-wise reading of matrices having row major memory layout. Left; the misaligned case of arbitrary number of columns. Right; the coalesced case occurring when the number of columns is padded to a multiple of the warp size.

row. In addition, we allow each block to take care of more than one row by dividing the matrix into slices of a given number of rows and launching a block for each of them.

4.4. Forward and Backward Substitution

A common technique for solving systems of linear equations is to determine the LU factorization of the left-hand side matrix followed by forward and backward substitution procedures [15]. In BLAS, the substitution procedures with one right-hand side are implemented as the level 2 routine STRSV.

Since a substitution procedure is sequential in nature, the implementation of the STRSV routine for parallel execution on a GPU requires some effort. Good performance on a GPU demands thread-level parallelism to hide memory latency, i.e., we need to introduce parallelism, which comes at the price of less numerical stability and more work than the minimum M^2 flops.

We adopt the approach of Nath *et al* [16], which is also used for the MAGMA library, and introduce parallelism by computing matrix inverses of blocks on the diagonal of the left-hand side triangular matrix. The matrix inverse of a triangular diagonal block (e.g., lower triangular) is based on the formulae

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}, \quad \mathbf{A}^{-1} = \begin{pmatrix} \mathbf{A}_{11}^{-1} & \mathbf{0} \\ -\mathbf{A}_{22}^{-1}\mathbf{A}_{21}\mathbf{A}_{11}^{-1} & \mathbf{A}_{22}^{-1} \end{pmatrix}, \tag{5}$$

for a square matrix $\mathbf{A}$ with a 2×2 block structure. A similar formulae exists for upper triangular matrices. Note that this is an $O(K^3)$ operation for blocks of size K.

In Fig. 5, we schematically illustrate this technique for doing forward substitution with a lower triangular left-hand side matrix. As shown in part A, the matrix inversions of the diagonal blocks are initially performed in shared memory up to the maximum block size (`INNER_DIAGSIZE`) that fits in the shared memory. This block size might not be sufficient in terms of introducing parallelism, so in part B, the inverted blocks on the diagonal are expanded further in global memory. Although recursively expanding the matrix inverse in global memory is significantly slower than in shared memory, there is a particular diagonal block size (`OUTER_DIAGSIZE`) where the trade off between extra work and introduced parallelism is optimal. In part C, the actual substitution procedure is performed in a blocked form based on parallel matrix-vector multiplication (see SGEMV above).

An important consideration for the approach we use here for STRSV is the numerical stability and loss of accuracy due to the evaluation of the matrix inverses. In this work we do not present such a discussion. Similar techniques have been known for some time, e.g., as the divide and conquer

DOI: 10.1002/cpe

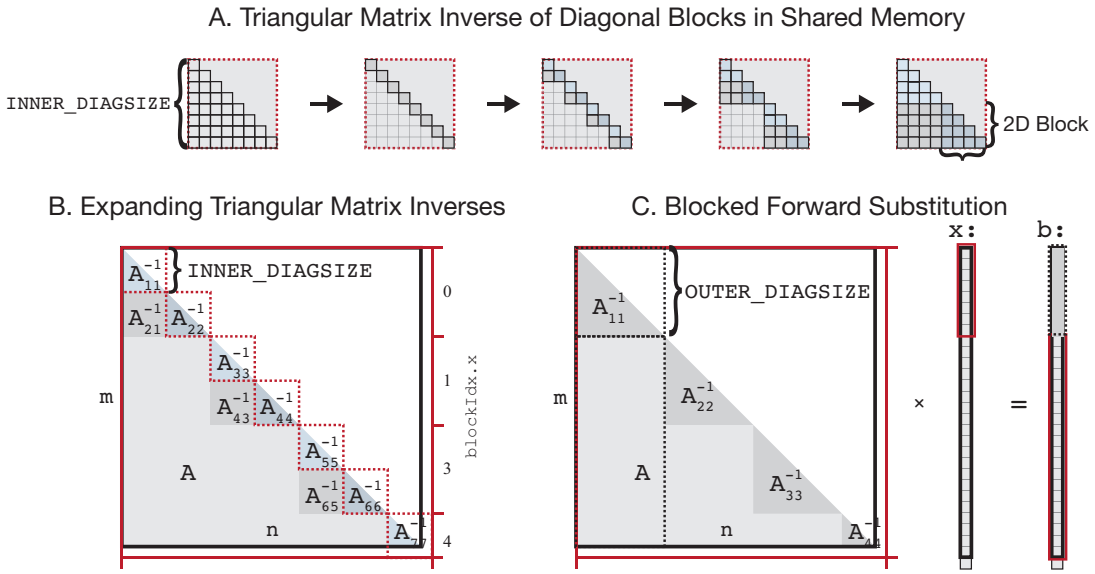

Figure 5. GPU algorithm for forward substitution. Part A; the initial matrix inverse of diagonal blocks in shared memory. Part B; subsequent expansion of the diagonal inverse blocks to sizes that do not fit in shared memory. Part C; final blocked forward substitution based on matrix-vector multiplication.

method for inverting a triangular matrix, which has been described by Heller [17] and later by Higham [18]. We refer the reader to [18] for a thorough discussion of the stability and accuracy.

5. AUTO-TUNING

In order to configure the level 1 and level 2 BLAS kernels with optimal parameters for the targeted GPU we will use auto-tuning of kernel parameters. We have implemented an auto-tuning framework that can automate the performance tuning process by running a large set of empirical benchmarks and store the results. Our search strategy is to consider a sub-set of the possible configurations based on knowledge we have about the GPU hardware and find the optimal among them using brute-force.

5.1. Pruning the Parameter Space

In general an auto-tuning framework is based on the assumption that if one can enumerate all possible implementations of a kernel, a computer can benchmark all of these variants in less time than a programmer would require to optimize the kernel by hand. When the full parameter space of a kernel is small, an exhaustive search implemented as a series of nested loops is effective and easy to implement [19]. The auto-tuner is then guaranteed to find the optimal kernel implementation within all the possible candidates that can be generated from the kernel. For complicated kernels with many parameters, however, the enumeration might be too extensive to meet the auto-tuning assumption, and the key to auto-tuning is then how to select the particular combinations to try.

One strategy is to search only part of the parameter space by doing a line-search for each parameter in turn, while the other parameters are fixed. This is sometimes referred to as a greedy search [20] and is widely used, e.g., in ATLAS [8]. The order in which the parameters are selected for line-search is important since the best parameterization depends on the previous searches. It is up to the programmer or user to provide the best search-order using whatever insight is available.

Another search strategy is to rely on heuristics to prune the search space so that it contains only a sub-set of the possible configurations and determine the best among them using an exhaustive search. We will use this strategy for the level 1 and level 2 BLAS kernels presented in this work.

DOI: 10.1002/cpe

Although a greedy search might reduce the auto-tuning runtime, most kernels can be auto-tuned in a matter of minutes after the parameter space has been pruned, which is sufficient for our purposes.

5.2. Using C++ Templates

GPU auto-tuners can be constructed in a variety of ways to test different kernel implementations, ranging from simply calling the kernel with different function arguments to run-time code generation of kernel candidates [21]. In this work, we implement the auto-tuner based on C++ function templates. The goal is to represent all tuning parameters as template values, which are then evaluated at compile time. This allows inner loops over these parameters to be completely unrolled and conditionals to be evaluated by the compiler at compile time. We are also able to set launch bounds depending on the tuning parameters using the CUDA keyword `__launch_bounds__`. All that is required is the declaration of the kernel as a template via `template <..>` and a large switch statement [13].

5.3. Tuning Parameters

The vector and matrix-vector operations implemented in this work each incorporate three or four tuning parameters, which are built into the design of the kernels.

5.3.1. Parameter 1: Block Size Commonly, the most important tuning parameter in the CUDA model is the number of threads per block. Since the smallest work entity to be scheduled and executed is a warp of 32 threads, we know that having `BLOCKSIZE` a multiple of 32 is the best choice for a high-performance kernel. We also know that to reach an occupancy of 1, at least 192 threads per block are needed [22], while to use the maximum 63 registers per thread at most 64 threads per block are allowed [22]. This trade-off leads us to search the parameter space

$$\texttt{BLOCKSIZE} \in \{32, 64, 96, 128, 160, 192, 224, 256\}, \tag{6}$$

for the optimal value (experiments confirm this to be appropriate for the C2050).

5.3.2. Parameter 2: Work Size per Thread Another tuning parameter addresses the performance trade-off between launching many threads in order to utilize the fine-grained parallelism of the GPU and having each thread perform a reasonable amount of work before it retires. Empirically, we found that the parameter space

$$\texttt{WORKSIZE} \in \{1, 2, 3, 4, 5, 6, 7, 8\} \times \texttt{BLOCKSIZE}, \tag{7}$$

for the number of elements handled per thread, is adequate for the C2050.

5.3.3. Parameter 3: Unroll Level A tuning parameter built into the design of some of our kernels is related to the CUDA compiler's technique for unrolling inner loops, where a particular unroll level `x` can be specified by `#pragma unroll x`. Using a high level gives the smallest loop counter overhead and fewer instructions but it also requires more registers per thread. We found that the unroll levels

$$\texttt{UNROLL\_LEVEL} \in \{\texttt{FULL}, 2, 3, 4, 5, 6, 7, 8\}, \tag{8}$$

can lead to different performances and this space is therefore searched.

5.3.4. Parameter 4: Algorithmic Trade-offs A final type of tuning parameter required in the design of some kernels are the parameters related to algorithmic performance trade-offs. For example in the STRSV kernel, the trade-off between increasing the parallelism and doing extra work, yields

$$\begin{aligned} \texttt{INNER\_DIAGSIZE} &\in \{2, 4\} \times \texttt{BLOCKSIZE}, \\ \texttt{OUTER\_DIAGSIZE} &\in \{1, 2, 4, 8, 16\} \times \texttt{INNER\_DIAGSIZE}, \end{aligned} \tag{9}$$

as the relevant parameter space to be searched in order to find the optimal settings.

We note that this amounts to a total of $8 \times 8 \times 8 = 512$ configurations for the SNRM2 and SGEMV kernels and $8 \times 2 \times 5 = 80$ configurations for the STRSV kernel to be tested.

DOI: 10.1002/cpe

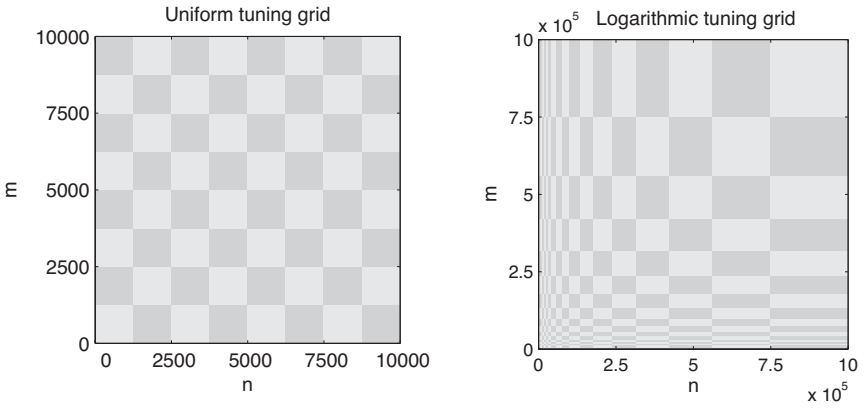

Figure 6. Tuning grids used by the auto-tuner for selecting the best parameters and kernels for different input matrix sizes $m \times n$. The left panel shows a uniform grid and the right panel a logarithmically spaced grid.

5.4. Tuning grid

The task of the auto-tuner is to select the most appropriate parameters for a given input vector or matrix size from the tuning space of the kernel. To this end, we want to store information of which parameters should be used for different sizes in a way that it is easy and cheap to access. The simplest approach is to divide the sizes $m \times n$ of the input that are to be taken into consideration into a regular and uniform grid and obtain tuning information for each tile. If the range of input sizes are limited for a given scientific application, it is most effective to use the auto-tuner for this range alone, in order to make the tuning grid more fine grained.

In order for the auto-tuning process to be reasonable fast, we have to restrict the granularity of the tuning grid to a modest level. Since the kernels are more sensitive to input size changes for small vectors and matrices it is most intuitive to have logarithmically spaced vertices in the grid for large tuning ranges. This would be the case for general purpose libraries, where one would like to have the kernels tuned for use over all possible sizes.

For proof of concept in this work, we select a 1×16 uniform tuning grid for vectors over the range 1 to 1000000 and a 8×8 uniform tuning grid for matrices over the range 1 to 10000, and assume shapes outside this region to be well represented by the "closest" tuned kernel within the region of the grid. We show the matrix grids used for auto-tuning in this work in Fig. 6.

6. RESULTS

Our test platform is a Nvidia Tesla C2050 card having 3 GB device memory on a host with a quad-core Intel® Core™i7 CPU operating at 2.80 GHz. The GPU's peak performance is 1.03 Tflops and the theoretical bandwidth is 144 GB/s. The error correction code (ECC) is on. See Fig. 1 for measurements of the effective global memory bandwidth on this platform. Note that the performance timings shown do not include transfer of data between host and GPU unless stated otherwise.

6.1. Euclidian Norm (SNRM2) on Fermi GPU

We show the result from auto-tuning the SNRM2 kernel in Fig. 7 for sizes up to 1000000. The top panel displays the selected best kernel designated by color and the bottom panel the corresponding performance achieved in Gflops/s. Also the names of the best kernels are listed in the middle in a form where the name of the operation is appended by "b1×{BLOCKSIZE}_w{UNROLL_LEVEL}×{WORKSIZE}".

DOI: 10.1002/cpe

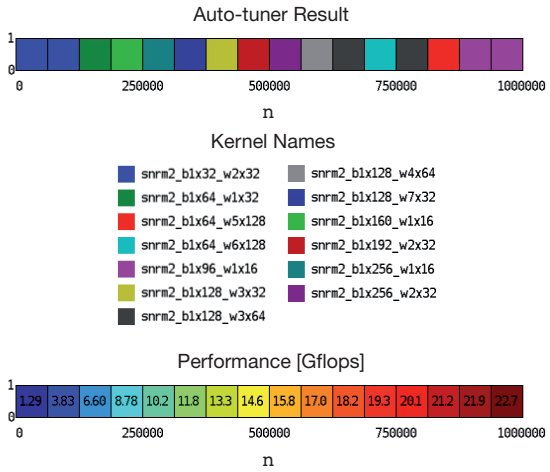

Figure 7. Result of auto-tuning for the SNRM2 kernel on a 1×16 grid for vector sizes up to $n = 1000000$. Top; best kernel designated by color and name. Bottom; performance in Gflops/s.

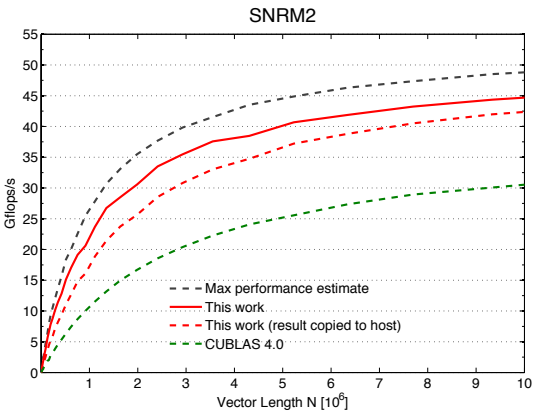

Figure 8. Performance of the auto-tuned SNRM2 kernel on a Nvidia Tesla C2050 card. The curves show the average performance from ten subsequent calls to the kernel.

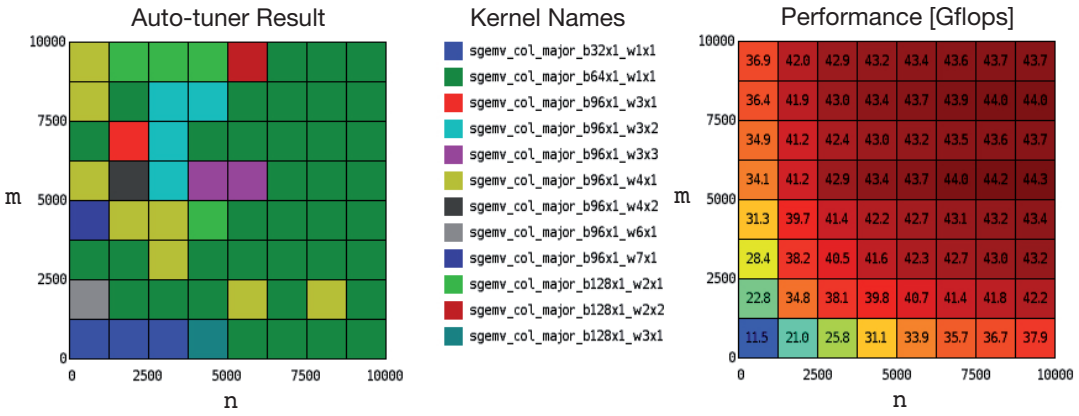

Figure 9. Result of auto-tuning for the SGEMV kernel on a 8×8 grid for matrix sizes $m \times n$ up to 10000×10000 Left; best kernel designated by color. Right; performance in Gflops/s.

In Fig. 8, we show the average performance of the auto-tuned SNRM2 kernel over a span of sizes of up to 10^7 elements. We compare the achieved results with the similar performance measurement for the SNRM2 kernel from CUBLAS 4.0 library. Note that the SNRM2 function in CUBLAS 4.0 gives the result on the host while our SNRM2 function by default gives the result on the GPU. For the sake of comparison, we make a version of our kernel that also copies the result to the host, although the difference in performance because of this is relatively small and becomes smaller for larger sizes of n.

We see from the figure that on average our auto-tuned SNRM2 kernel performs $> 30\%$ better than the current CUBLAS 4.0 kernel. In addition, we see by comparing to the performance estimate that the achieved performance is only about 10% lower than the maximum set by the effective memory bandwidth. Since there is a reduction operation involved in computing the Euclidian norm, this loss is fully acceptable as a decrease in performance cannot be avoided (as discussed in Sect. 4.1).

DOI: 10.1002/cpe

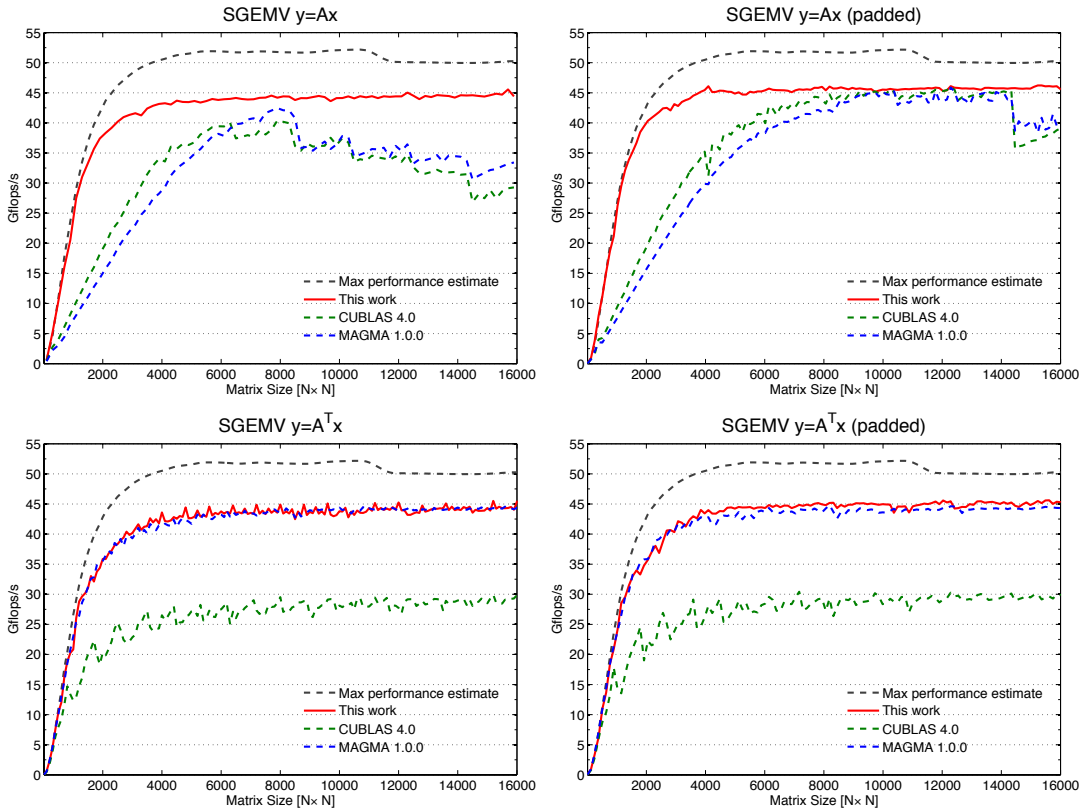

Figure 10. Performance of the auto-tuned SGEMV kernel (with `TRANS` = 'N' and 'T') on a Nvidia Tesla C2050 card. The curves show the average performance from ten calls.

6.2. *Matrix-Vector Multiplication (SGEMV) on Fermi GPU*

We show the result of auto-tuning our SGEMV kernel in Fig. 9, where we have considered matrix sizes up to 10000×10000 on an 8×8 tuning grid. The auto-tuner and performance results shown are obtained as averages from 25 samples within each tuning grid tile. A total of 12 different best kernels, designated by a unique color, are selected. The names of the best kernels have the extension "b{`BLOCKSIZE`}×1_w{`UNROLL_LEVEL`}×{`WORKSIZE`}".

In Fig. 10, we show the achieved performance of the auto-tuned SGEMV kernel in the cases of ordinary ($\mathbf{y} = \mathbf{A}\mathbf{x}$) and transposed ($\mathbf{y} = \mathbf{A}^{\mathrm{T}}\mathbf{x}$) square matrix-vector multiplication and for matrices with and without padding. We see a significant improvement in comparison with the corresponding kernel in the current CUBLAS 4.0 library (up to $\sim 100\%$ in the transposed case).

We also compare with the most recent MAGMA library [5] and see some improvement in the ordinary matrix-vector multiplication case. In the transposed matrix-multiplication case, our auto-tuned kernel confirms that MAGMA's kernel is already highly optimized for square matrices of the sizes considered here.

In addition, the performance results show that the padding of matrices is not necessary in order to achieve high performance on the C2050 card. In our kernel the increase in performance from padding to a multiple of the warp size is only a few percent. We credit this to the L1 and L2 caches available on Fermi GPUs [12].

In all the performance plots in Fig. 10 the estimate of the maximum performance based on the effective memory bandwidth is indicated. In all cases, for matrix sizes larger than 3000×3000, the achieved performance of the auto-tuned kernels is $10 - 20\%$ below the estimate. Again, this can be

DOI: 10.1002/cpe

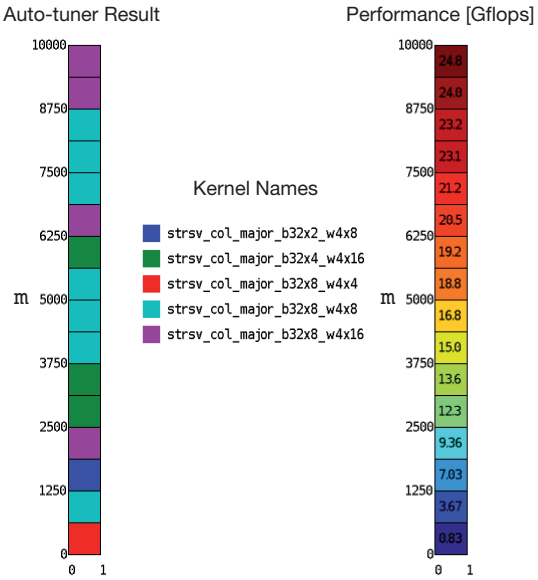

Figure 11. Result of auto-tuning for the STRSV kernel on a 16×1 grid for matrix sizes up to $m \times m = 10000 \times 10000$. Left; best kernel designated by color and name. Right; performance in Gflops/s.

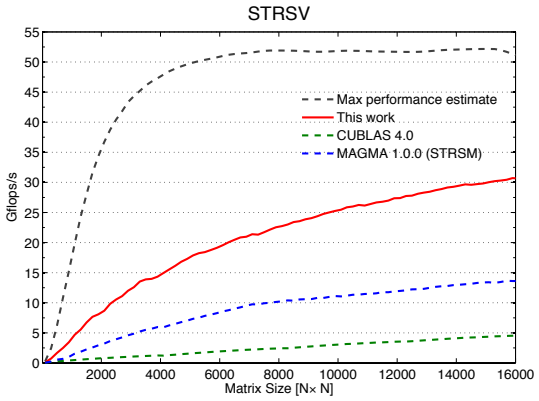

Figure 12. Performance of the auto-tuned STRSV kernel on a Nvidia Tesla C2050 card. The curves show the average performance from ten subsequent calls to the kernel.

partly explained by the reduction operations, which are required in the sequence of inner products that make up the matrix-vector multiplication operation.

6.3. Triangular Solve (STRSV) on Fermi GPU

In Fig. 11 we show the result of auto-tuning our STRSV kernel for lower triangular left-hand side matrices of sizes up to 10000×10000. The tuning grid is uniform an has 16×1 tiles. The auto-tuner and performance results shown are obtained as averages from 5 uniformly distributed samples within each tuning grid tile. The auto-tuning process results in 5 best kernels, which are named with extensions "b{$32\times$BLOCKSIZE}_w{INNER_DIAGSIZE}$\times${OUTER_DIAGSIZE}" and designated by color to the left and the corresponding performance to the right. We note that in all the best kernels we have the tuning parameter INNER_DIAGSIZE $= 4$, which is explained by the fixed maximum size of the shared memory (up to 48 KB per multiprocessor for C2050).

In Fig. 10, we show the achieved performance of the auto-tuned STRSV kernel together with curves for similar calls to the CUBLAS 4.0 library and MAGMA 1.0.0 (since the STRSV routine is not available in MAGMA 1.0.0, the level 3 BLAS routine STRSM with one right-hand side is used instead). We see a significant increase ($> 100\%$) in performance for the auto-tuned kernel in comparison with the existing libraries. In the case of the MAGMA 1.0.0 library, which uses the same algorithm as in this work, the main improvement is due to the use of the already auto-tuned kernel SGEMV for the implicit matrix-vector multiplications in the blocked substitution procedure.

The max performance estimate for the STRSV kernel, which is also plotted in Fig. 10, indicates that the performance is far from the limit set by the global memory bandwidth. This shows, that in order to get better performance it is not sufficient to auto-tune the currently available kernels.

7. CONCLUSION

In this work, we have implemented level 1 and level 2 BLAS operations as high-performance GPU kernels designed for auto-tuning. We used auto-tuning of the kernels in order to select the optimal kernel parameters on the Tesla C2050 card (Fermi GPU). The auto-tuning consisted of an

DOI: 10.1002/cpe

exhaustive search of the tuning parameter space containing key hardware dependent parameters that sets the number of threads per block, the work per thread, the unroll level of the inner-most loop, and algorithmic trade-offs.

We implemented our auto-tuner and kernels based on C++ function templates, which are supported in the CUDA programming model, and allows for meta-programming kernels, where the tuning arguments are evaluated at compile time. The proposed auto-tuning procedure then required at most 512 different configurations for a particular kernel to be compiled and benchmarked for a given input or on a tuning grid. Furthermore, we used empirical tests and knowledge about the hardware to prune the parameter space before the auto-tuning, which effectively reduced the auto-tuning runtime to a matter of minutes for all the level 1 and level 2 BLAS kernels.

We have illustrated the approach for the Level 1 BLAS routines, with the example of the Euclidian norm (SNRM2), and for the Level 2 BLAS routines in the case of the matrix-vector multiplication (SGEMV) and the triangular solve with one right-hand side (STRSV). In all cases we achieved significantly better performance than the similar routines in the CUBLAS 4.0 library.

In order to evaluate the performance of level 1 and level 2 BLAS kernels, which are inherently memory bound, we proposed that the measurement of the effective bandwidth for a given input size is also the best metric for estimating the maximum performance that can be reached. In practice, the estimates showed that the SNRM2 and SGEMV kernels are relatively close to the best we can hope for, whereas the STRSV kernel is far from the maximum performance. In other words, there is still room for improvement in the development of parallel GPU algorithms for triangular solvers.

REFERENCES

1. NVIDIA Corp.. *CUDA Toolkit Version 3.2* 2010.
2. Khronos Group. *OpenCL Specification 1.1* 2010.
3. Dongarra JJ, Du Croz J, Hammarling S, Duff IS. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* March 1990; **16**:1–17.
4. Anderson E, Bai Z, Bischof C, Blackford LS, Demmel J, Dongarra JJ, Du Croz J, Hammarling S, Greenbaum A, McKenney A, *et al.*. *LAPACK Users' guide (third ed.)*. SIAM: Philadelphia, PA, USA, 1999.
5. Tomov S, Nath R, Du P, Dongarra J. *MAGMA v0.2 Users' Guide* 2009.
6. Humphrey JR, Price DK, Spagnoli KE, Paolini AL, Kelmelis EJ. CULA: hybrid GPU accelerated linear algebra routines. *Proc. SPIE* Apr 2010; **7705**.
7. Dongarra J, Moore S. *Empirical Performance Tuning of Dense Linear Algebra Software*, chap. 12. CRC Press, 2010; 255–272.
8. Whaley RC, Petitet A, Clint R, Antoine W, Jack P, Dongarra JJ. *Automated Empirical Optimizations of Software and the ATLAS project* 2000.
9. Li Y, Dongarra J, Tomov S. A note on auto-tuning gemm for gpus 2009.
10. Micikevicius P. Analysis-driven performance opt. GTC, Recorded Session 2010.
11. Volkov V. Better performance at lower occupancy. GTC, Recorded Session 2010.
12. NVIDIA Corp.. *Fermi, Whitepaper* 2009.
13. Harris M. Optimizing parallel reduction in cuda. *NVIDIA Dev. Tech.* 2008; .
14. NVIDIA Corp.. *CUDA C Programming Guide Version 3.2* 2010.
15. Golub GH, Van Loan CF. *Matrix Computations*. The Johns Hopkins University Press, 1996.
16. Nath R, Tomov S, Dongarra J. Blas for gpus. *Scientific Computing with Multicore and Accelerators*, Kurzak J, Bader DA, Dongarra J (eds.). CRC Press, 2010.
17. Heller D. A survey of parallel algorithms in numerical linear algebra. *SIAM Review* 1978; **20**(4):pp. 740–777.
18. Higham NJ. Stability of parallel triangular system solvers. *SIAM J. Sci. Comput.* March 1995; **16**:400–413.
19. Sato K, Takizawa H, Komatsu K, Kobayashi H. Automatic tuning of cuda execution parameters for stencil processing. *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, 2010.
20. Samuel W, Kaushik D, Leonid O, Jonathan C, John S, Katherine Y. *Auto-Tuning Memory-Intensive Kernels for Multicore*. CRC Press, 2010.
21. Klöckner A, Pinto N, Lee Y, Catanzaro B, Ivanov P, Fasih A. PyCUDA: GPU Run-Time Code Generation for High-Performance Computing Nov 2009; .
22. NVIDIA Corp.. *CUDA GPU Occupancy Calculator* 2010.

DOI: 10.1002/cpe