Auto-tuning Dense Vector and Matrix-Vector Operations for Fermi GPUs

Hans Henrik Brandenborg Sørensen

Informatics and Mathematical Modelling,
Technical University of Denmark, Bldg. 321, DK-2800 Lyngby, Denmark
hhs@imm.dtu.dk
http://www.gpulab.imm.dtu.dk

Abstract. In this paper, we consider the automatic performance tuning of dense vector and matrix-vector operations on GPUs. Such operations form the backbone of level 1 and level 2 routines in the Basic Linear Algebra Subroutines (BLAS) library and are therefore of great importance in many scientific applications. As examples, we develop single-precision CUDA kernels for the euclidian norm (SNRM2) and the matrix-vector multiplication (SGEMV). The target hardware is the most recent Nvidia Tesla 20-series (Fermi architecture). We show that auto-tuning can be successfully applied to achieve high performance for dense vector and matrix-vector operations by appropriately utilizing the fine-grained parallelism of the GPU. Our tuned kernels display between 25-100% better performance than the current CUBLAS v.3.2 library.

Keywords: GPU, BLAS, Dense linear algebra, Parallel algorithms

1 Introduction

Graphical processing units (GPUs) have already become an integral part of many high performance computing systems, since they offer dedicated parallel hardware that can potentially accelerate the execution of many scientific applications. 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. However, developers 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] 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 used extensively to automatically generate 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.

In this paper, we will apply auto-tuning to reduce the time and effort required to program high performance vector and matrix-vector operation kernels for GPUs. Our goal is to design kernels based on template parameters and auto-tune them for given problem sizes. All the kernels, which may be 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 3.2), 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. Next we describe the implementations. In Sect. 4 we describe the auto-tuning process. The experimental results are presented in Sect. 5.

2 Performance Considerations

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 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 [9], 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 effective memory bandwidth of the GPU.

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 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 [10]. 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 Vector and Matrix-Vector Operations on Fermi GPUs

The Tesla C2050 Fermi architecture provides 14 multiprocessors of 32 cores each that can execute threads in parallel. 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.

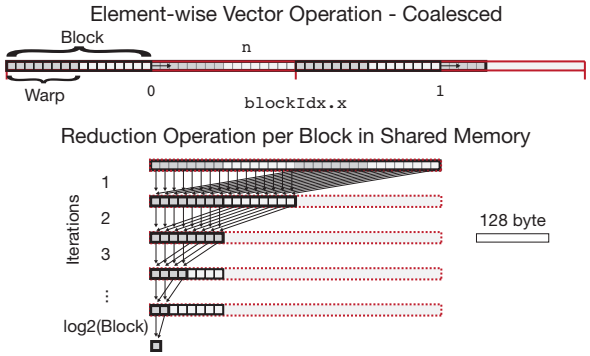

Fig. 1. Coalesced access pattern for the element-wise operation on a vector and the subsequent parallel reduction operation per block in shared memory.

3.1 Operations on a Vector

In Fig. 1, 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 or 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 [11]. 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.

3.2 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 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

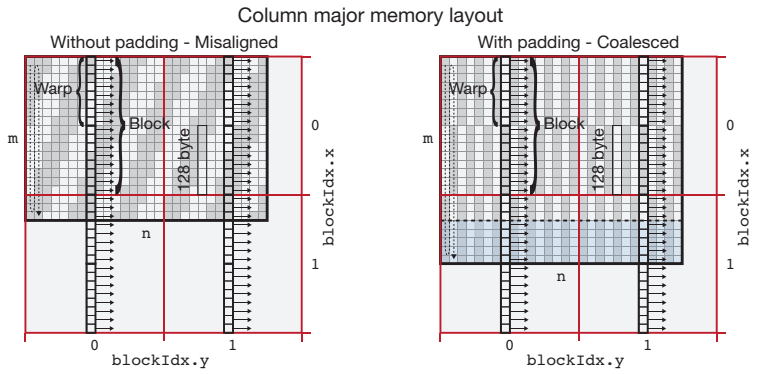

Fig. 2. Access pattern for the row-wise reading of matrices having column major memory layout. Left; the misaligned case of arbitrary number of rows. Right; the coalesced case occurring when the number of rows is padded to a multiple of the warp size.

might only take care of part of a row, which is accomplished by dividing the matrix into tiles instead of slices and using 2D grid of blocks.

As illustrated in the left part of Fig. 2, 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 [12].

As illustrated in the right part of Fig. 2, 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.

3.3 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. 3, 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

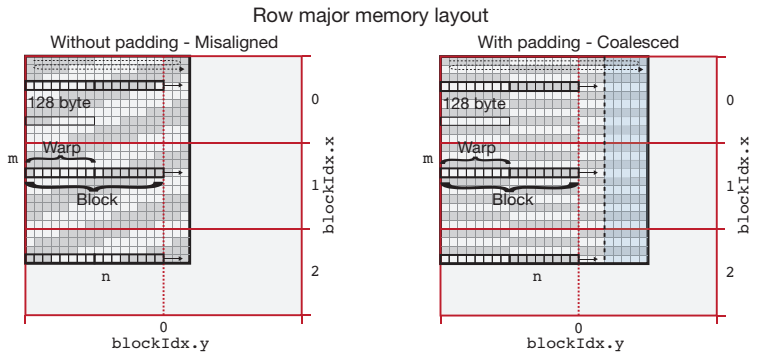

Fig. 3. Access pattern for the row-wise reading of matrices having row major memory layout. Left; the misaligned case of arbitrary number of columns. Right; the coalesced case occurring when the number of columns is padded to a multiple of the warp size.

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 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 Auto-tuning

In order to configure the vector and matrix-vector kernels with optimal parameters for the targeted GPU we will use auto-tuning. We have implemented an auto-tuning framework that can automate the performance tuning process by running a large set of empirical benchmarks. Our search strategy is to rely on heuristics to evaluate only a sub-set of the possible configurations based on knowledge we have about the GPU hardware and find the optimal among them.

4.1 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 [13]. 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 [11].

4.2 Tuning Parameters

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

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 [14], while to use the maximum 63 registers per thread at most 64 threads per block are allowed [14]. This trade-off leads us to search the parameter space

$$\texttt{BLOCKSIZE} \in \{32, 64, 96, 128, 160, 192, 224, 256\}, \tag{1}$$

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

Parameter 2: Work Size per Thread Another tuning parameter adresses 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

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

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

Parameter 3: Unroll Level A final tuning parameter built into the design 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

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

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

We note that this amounts to total of $8 \times 8 \times 8 = 512$ configurations of our vector and matrix-vector kernels to be auto-tuned for a given problem size.

5 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 Gflops and the theoretical bandwidth is 144 GB/s. The error correction code (ECC) is on. Running `bandwidthTest()` from the CUDA Toolkit 3.2 SDK gives 84.4 GB/s. Note that the performance timings shown do not include transfer of data between host and GPU unless stated otherwise.

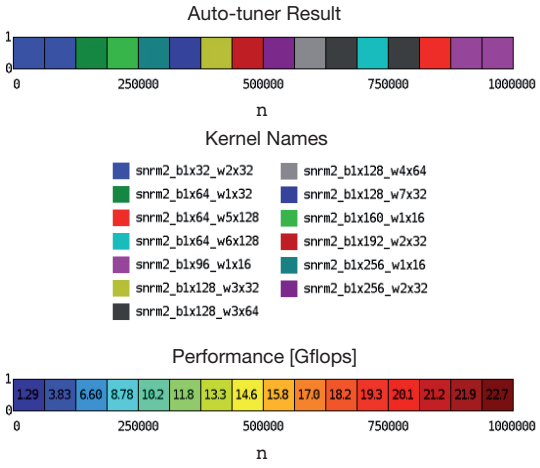

Fig. 4. Result of auto-tuning for the SNRM2 kernel on a 1×16 grid for vector sizes up to $n = 1000000$. Top; best kernel designated by color and name. Bottom; performance in Gflops.

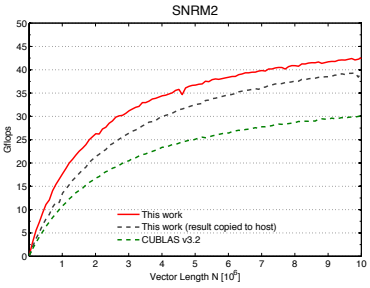

Fig. 5. Performance of the auto-tuned SNRM2 kernel on a Nvidia Tesla C2050 card. The curves show the average performance from ten subsequent calls to the kernel.

5.1 Euclidian Norm (SNRM2) on Fermi GPU

We show the result from auto-tuning the SNRM2 kernel in Fig. 4 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. 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`}".

In Fig. 5, 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 v3.2 library. Note that the SNRM2 function in CUBLAS v3.2 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. As shown, the difference in performance because of this is relatively small and becomes smaller for larger sizes of n. On average, our auto-tuned SNRM2 kernel performs $> 30\%$ better than the current CUBLAS v3.2 kernel.

5.2 Matrix-Vector Multiplication (SGEMV) on Fermi GPU

We show the result of auto-tuning our SGEMV kernel in Fig. 6, where we have considered matrix sizes up to 10000 rows and 10000 columns 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`}".

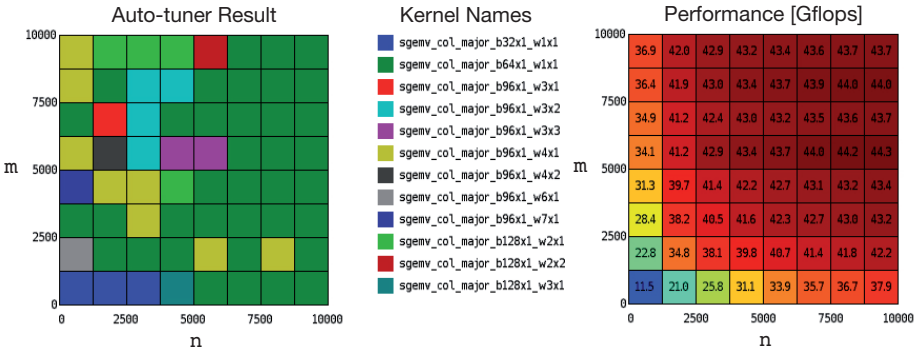

Fig. 6. Result of auto-tuning for the SGEMV kernel on a 8×8 grid for matrix sizes $m \times n$ up to 10000 Left; best kernel designated by color. Right; performance in Gflops.

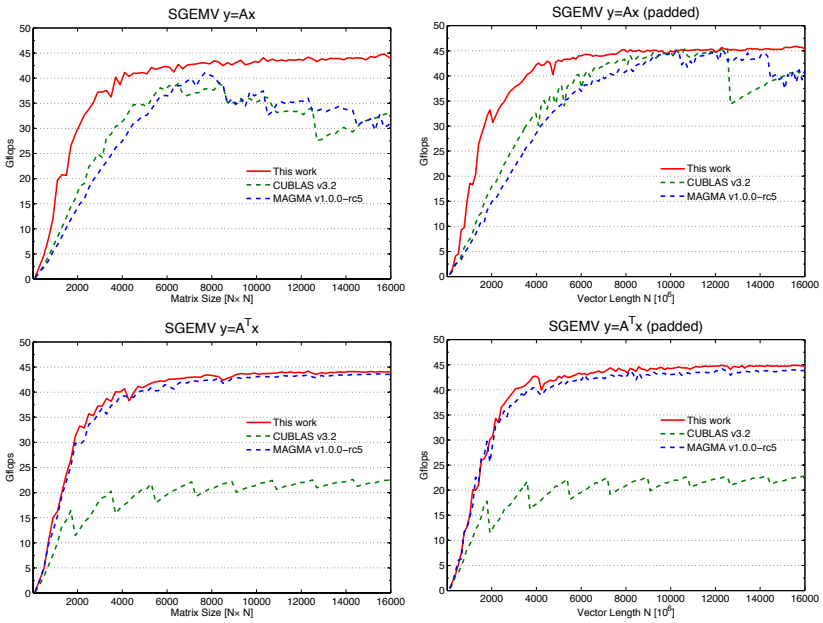

Fig. 7. Performance of the auto-tuned SGEMV kernel (with `TRANS` = 'N' and 'T') on a Nvidia Tesla C2050 card. The curves show the average performance from ten calls.

In Fig. 7, we show the achieved performance of the auto-tuned SGEMV kernel in the cases of ordinary ($\mathbf{y} = \mathbf{Ax}$) 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 v3.2 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.

6 Conclusion

In this work, we have implemented vector and matrix-vector 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 a heuristic search of a tuning space containing key hardware dependent parameters that sets the number of threads per block, the work per thread, and the unroll level of the inner-most loop.

We have illustrated the approach for the Level 1 BLAS routines, with the example of the Euclidian norm, and for the Level 2 BLAS routines in the case of the matrix-vector product operations. We achieve significantly better performance compared to the CUBLAS v3.2 library. Two other basic Level 2 operations, the rank-1 and rank-2 updates and the triangular solve, are left as future work.

References

1. NVIDIA Corp.: CUDA Toolkit Version 3.2. (2010)
2. Khronos Group: OpenCL Specification 1.1. (2010)
3. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. **16** (1990) 1–17
4. Angerson, Bai, Dongarra, Greenbaum, McKenney, Croz, D., Hammarling, Demmel, Bischof, Sorensen: LAPACK: A portable linear algebra library for high-performance computers. SC Conference **0** (1990) 2–11
5. Tomov, S., Nath, R., Du, P., Dongarra, J.: MAGMA v0.2 Users' Guide. (2009)
6. Humphrey, J.R., Price, D.K., Spagnoli, K.E., Paolini, A.L., Kelmelis, E.J.: CULA: hybrid GPU accelerated linear algebra routines. Proc. SPIE **7705** (2010)
7. Jack, D., Shirley, M.: 12. In: Empirical Performance Tuning of Dense Linear Algebra Software. CRC Press (2010) 255–272
8. Whaley, R.C., Petitet, A., Clint, R., Antoine, W., Jack, P., Dongarra, J.J.: Automated Empirical Optimizations of Software and the ATLAS project. (2000)
9. Micikevicius, P.: Analysis-driven performance optimization. GPU Technology Conference, Recorded Session (2010)
10. Volkov, V.: Better performance at lower occupancy. GPU Technology Conference, Recorded Session (2010)
11. Harris, M.: Optimizing parallel reduction in cuda. NVIDIA Dev. Tech. (2008)
12. NVIDIA Corp.: CUDA C Programming Guide Version 3.2. (2010)
13. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: PyCUDA: GPU Run-Time Code Generation for High-Performance Computing. (2009)
14. NVIDIA Corp.: CUDA GPU Occupancy Calculator. (2010)