Auto-Tuning of Thread Assignment for Matrix-Vector Multiplication on GPUs

Jinwei WANG†,††, Student Member, Xirong MA††, Yuanping ZHU††, and Jizhou SUN††, Nonmembers

SUMMARY Modern GPUs have evolved to become a more general processor capable of executing scientific and engineering computations. It provides a highly parallel computing environment due to its large number of computing cores, which are suitable for numerous data parallel arithmetic computations, particularly linear algebra operations. The matrix-vector multiplication is one of the most important dense linear algebraic operations. It is applied to a diverse set of applications in many fields and must therefore be fully optimized to achieve a high-performance. In this paper, we proposed a novel auto-tuning method for matrix-vector multiplication on GPUs, where the number of assigned threads that are used to compute one element of the result vector can be auto-tuned according to the size of matrix. On the Nvidia’s GPU GTX 650 with the most recent Kepler architecture, we developed an auto-tuner that can automatically select the optimal number of assigned threads for calculation. Based on the auto-tuner’s result, we developed a versatile generic matrix-vector multiplication kernel with the CUDA programming model. A series of experiments on different shapes and sizes of matrices were conducted for comparing the performance of our kernel with that of the kernels from CUBLAS 5.0, MAGMA 1.3 and a warp method. The experiments results show that the performance of our matrix-vector multiplication kernel is close to the optimal behavior with increasing of the size of the matrix and has very little dependency on the shape of the matrix, which is a significant improvement compared to the other three kernels that exhibit unstable performance behavior for different shapes of matrices.

key words: GPU, matrix-vector multiplication, performance tuning, dense linear algebra

1. Introduction

The matrix-vector multiplication is an important operation for a diverse set of applications in many fields, such as scientific computing, engineering, economic modeling, information retrieval and etc. Matrix-vector multiplication is also a key routine of level 2 Basic Linear Algebra Subroutines (BLAS) [1]. Therefore, additional effort must be spent to accelerate the computation of matrix-vector multiplications.

Currently, as a general-purpose and high performance parallel hardware, Graphics Processing Units (GPUs) are being developed continuously. GPU is a powerful many-core processor which can handle thousands of threads running concurrently. The highly parallel architecture of a GPU is very suitable for linear algebra operations, e.g., matrix-vector multiplication. In order to exploit the computing potential of GPUs, the programmer has to use either Nvidia’s Compute Unified Device Architecture (CUDA) [2], [3] or the Open Compute Language (OpenCL) [4] that greatly facilitates the development of applications targeted at GPUs.

Several high-performance BLAS libraries on GPUs have been developed with CUDA, e.g., Nvidia’s CUBLAS [5] and the open source MAGMA library [6]. Many developers rely on these libraries to achieve good performance from their applications. However, in the case of matrix-vector multiplication, these libraries are currently unsatisfactory and suffer from low utilization of the GPU’s hardware. Due to the fact that the shape of the matrix can greatly impact the performance of matrix-vector multiplication [7], these libraries cannot maintain good performance for all shapes of matrices.

In this paper, we seek to solve the above problem by auto-tuning matrix-vector multiplication for GPUs. Auto-tuning is a technique that has been used widely on CPUs to automatically generate near optimal numerical libraries [8], e.g., ATLAS [9]. The general-purpose GPUs offer more complex architectures than CPUs, which require nontrivial optimization strategies that often change from one chip generation to the next. Therefore, as already demonstrated in MAGMA [6], auto-tuning is also very compelling on GPUs. As such, auto-tuning can also be a very effective optimization technique on GPUs. In this work, our goal is to design a versatile matrix-vector multiplication kernel by auto-tuning of thread assignment, where threads compute one row of the matrix together within a group, the size of which can be auto-tuned according to the row and column size of the input matrix, that is in order to balance the workload across the GPUs resources and achieve high-performance for all given shapes of matrices. We target Nvidia GPUs, specifically the GTX650 (Kepler architecture), which is designed from the outset for scientific computations.

The remainder of the paper is organized as follows. Section 2 describes GPU architecture and CUDA programming model. Section 3 introduces some existing GPU-based matrix-vector multiplication implementations. Section 4 presents our design of the auto-tunable matrix-vector multiplication kernel. Section 5 presents our experimental results, and Sect. 6 concludes the paper.

2. GPU Architecture and CUDA Model

The GPU architecture is based on a set of multiprocessor units called streaming multiprocessors (SM), each one containing a set of processor cores called streaming proces-
There are several previous works on optimizing matrix-vector multiplication for GPUs. In this section we will review the most relevant work.

Nvidia Corporation implemented matrix-vector multiplication kernel in CUBLAS, which is the vendor implementation for BLAS on GPUs and is often regarded as the reference implementation. The latest version of CUBLAS is 5.0 [5] that delivers 6x to 17x faster performance than the latest MKL BLAS [15].

Fujimoto [16] developed a fast matrix-vector multiplication kernel in CUDA that was specifically tuned for the Nvidia’s graphics card: GeForce 8800GTX. He used the shared memory blocking method in order to maximize data reuse of the x vector in combination with tiling of the matrix A. The performance of his kernel was significantly better than the CUBLAS v1.1 library available at that time.

Tomov et al. presented a matrix-vector multiplication kernel as one of the key ingredients in their MAGMA library [17], which is a dense linear algebra package for heterogeneous CPU-GPU systems with the same functionality as the legacy LAPACK library [18]. Because several generic optimization techniques including pointer redirection [19] and auto-tuning [20] were introduced, the performance of their kernel is a significant improvement over the CUBLAS 2.3. In this work they also presented a kernel for transposed matrix-vector multiplication, which like Fujimoto’s kernel, allows groups of threads within a block to work together followed by a required reduction operation. The maximum performance for the transposed version was more than twice of what CUBLAS 2.3 could deliver.

W. Xu et al. [21] proposed an algorithm to optimize matrix-vector multiplication on Fermi GPU. They use a warp to compute a row of the matrix A. Their kernels perform better than CUBLAS 4.0 or MAGMA for small square and wide matrices. They also proposed an empirical auto-tuning method to choose a best algorithm for an input matrix from their algorithms and CUBLAS.

4. Auto-Tuning Method

In this section, we first conducted an analysis of how thread assignments affects the performance of the matrix-vector multiplication kernel on the basis of results of the experiment. Next, according to this analysis, to achieve high-performance for all shapes of matrices we present a novel auto-tuning method of thread assignment for developing the matrix-vector multiplication kernel, where a group of threads are used to compute one element of the result vector together, and the size of the group can be auto-tuned according to the row and column size of the input matrix. A versatile matrix-vector multiplication kernel and an auto-tuner are developed based on this method.

4.1 Thread Assignment and Performance

In previous matrix-vector multiplication kernels, the typical parallel implementation is that one or several threads (e.g. a warp) are assigned to perform a dot product between one...
Fig. 2 The influence of the number of assigned threads which are used to compute one element of result vector to the performance of matrix-vector multiplication. There are eleven different numbers of assigned threads from $2^0$ to $2^{10}$ and five different shapes (rows $\times$ columns) of matrices.

Table 1 Assigned threads and performance.

| Matrix size (Rows $\times$ Columns) | Multiplier (Highest PRF / Lowest PRF) | Assigned Threads (at Highest PRF) |
|-----------------------------------|--------------------------------------|----------------------------------|
| 32 $\times$ 32768                | 74.2                                 | 128                              |
| 128 $\times$ 8192                | 19.8                                 | 64                               |
| 1024 $\times$ 1024              | 6.1                                  | 16                               |
| 8192 $\times$ 128               | 28.0                                 | 8                                |
| 32768 $\times$ 32              | 106.6                                | 4                                |

The reason for above phenomenon is that when the matrix is very wide (the number of rows is small and the number of columns is large) and a small number of threads are launched, the occupancy of GPU will be very low and each thread needs to conduct a large number of calculations. The fewer threads assigned, the more calculations that each thread needs to conduct. As a result, the performance of matrix-vector multiplication kernel will reduce noticeably. Thus more threads should be launched to participate in calculation. On the contrary, when the matrix is very tall (the number of rows is large and the number of columns is small), assigning too many threads for computing one element of the result vector but will cause decreasing of the registers and shared memory that each thread can use and hence low performance of the matrix-vector multiplication kernel. Therefore, the method where an invariable amount of threads are assigned to compute one element of the result vector obviously is not exactly appropriate for all shapes and sizes of input matrices.

4.2 Auto-Tuning of Thread Assignment

In order to develop a versatile matrix-vector multiplication kernel suitable for all shapes of matrices, we present a novel auto-tuning method, where instead of a fixed number of threads, a thread group with variable size is assigned to compute one element of result vector. The details of our method are illustrated in Fig. 3. We consider only the case of row major memory layout. Assume that the size of thread group is $N$. Threads in the same group read both the elements in the same row of the matrix, and the corresponding elements of vector from the global device memory to perform a multiplication and an addition. For satisfying coalesced memory access, as shown in Fig. 3, each thread in the group reads one element every $\log_2 N$ elements once. Finally, when reaching the end of the row each thread gets a partial result and stores this result in the shared memory. Then a parallel reduction by all the threads in the group is performed on shared memory to get the final result. The access pattern for the reduction is illustrated in Fig. 4, which is designed to avoid shared memory bank conflicts [22]. It requires $\log_2 N$
iterations. In this method, the size of thread group, $N$, can be auto-tuned according to the row and column size of the matrix. It means that when the matrix has many columns the $N$ should be large, and when the matrix has many rows the $N$ should be small.

To implement our method, we choose the GeForce GTX 650 GPU as the development platform which is based on the NVIDIA new architecture: Kepler GK107. This GPU has two new Streaming Multiprocessors, called SMX, and each SMX has 192 Scalar Processors, resulting in a total of 384 processor cores. Also it has compute capabilities 3.0 where the maximum amount of shared memory per SMX is 48KB, the maximum number of threads per block is 1024 and the maximum number of resident threads per SMX up to be 2048. Thus, on GTX 650 the maximum number of total resident threads is 4096.

We selected eleven different numbers from $2^0$ to $2^{10}$ as the size $N$ of assigned thread group to fit all sizes or shapes of matrices, and implemented a matrix-vector multiplication kernel for each $N$ with CUDA.

$$N \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$$

The reason that we selected 1024 as the max number of $N$ is that all assigned threads in the group needs to perform a parallel reduction on shared memory in a block and the maximum number of threads per block is 1024 on the GPU with Kepler architecture. These eleven matrix-vector multiplication kernels have similar structures, thus we just give a representative pseudo-code, described in Algorithm 1, to explain the operation process of these kernels.

Lines 1 and 2 show how the algorithm designs the thread block and grid. In our kernels, the number of threads per block is 1024, the number of blocks per grid is 4, resulting in a total of 4096 threads running on GPU and 100% occupancy. Therefore, our method can simultaneously calculate at most 4096 rows when $N = 1$, and at least 4 rows when $N = 1024$. Line 3 calculates the ID of the row that each thread starts from. Line 4 calculates the step between the rows that need to be computed by each thread. Line 5 allocates an array with 1024 cells, $\text{SubSum}$, in shared memory. Lines 7–24 are the procedure for computing one element of $y$. First, lines 7–13 calculate the sum of elements dispatched to each thread and store the partial sum into a cell of $\text{SubSum}$. Then, after a block synchronization (line 14), on lines 15–21 all threads in the same group conduct a parallel reduction on the $\text{SubSum}$ to get the total sum. Finally, on lines 22–24 the first thread of each group writes the total sum into the global memory.

Specially, on lines 16, 17 and 19, there are two processes of parallel reduction: intra-warp reduction and inter-warp reduction. When $N \leq 32$ (a warp size) it is needed only intra-warp reduction, and when $N > 32$ it is needed inter-warp reduction first and intra-warp reduction next. According to Nvidia’s official documents, the implicit warp-synchronous programming are unsafe and easily broken by evolutionary improvements to the optimization strategies used by the CUDA compiler toolchain [23], therefore, we used a warp shuffle operation, which is a new warp-level intrinsic operation introduced by Kepler architecture, to implement the intra-warp reduction. The shuffle operation allows the threads of a warp to exchange data with each other directly without going through shared memory. The shuffle instruction also has lower latency than shared memory access [23]. The access pattern for the inter-warp reduction is the same as being showed in Fig. 4, in which a synchronization operation across threads in the same group must be performed between every two iterations.

We combined the eleven kernels into a versatile generic matrix-vector multiplication kernel based on C++ function templates. The goal is to represent the number $N$ of assigned threads in a group as a template value, which is evaluated at compile time. All that is required is the declaration of the kernel as a template via template <...> and a large switch statement. In order to automatically find the optimal number of assigned threads for different shapes and sizes of matrices at runtime, based on the versatile matrix-vector multiplication kernel we also implemented an auto-tuner which can automate the performance tuning process by running a large set of benchmarks.

---

Algorithm 1: Our matrix-vector multiplication kernel

```c
// A: matrix with $m$ rows and $n$ columns.
// x: vector with $n$ elements.
// y: result vector with $n$ elements.
// N: the size of the thread group.
// gridSize: the size of the thread grid.
// blockSize: the size of the thread block.
// threadIdxInBlock: the ID of the thread in the block.
// threadIdxInGrid: the ID of the thread in the grid.
// stepOfRows: the step of rows in matrix A.
// i: the row ID in matrix A.
// SubSum: a shared memory array.
1. gridSize $\leftarrow 4$;
2. blockSize $\leftarrow 1024$;
3. $i \leftarrow \text{threadIdxInGrid} / N$;
4. $\text{stepOfRows} \leftarrow \left( \text{gridSize} \times \text{blockSize} \right) / N$;
5. $\text{subSum}[1024]$ in shared memory;
6. while $i < m$
7. total $\leftarrow 0$;
8. $k \leftarrow \text{threadIdxInGroup}$
9. while $k < n$
10. total $\leftarrow$ total $+$ $A[i*n+k]$;
11. $k \leftarrow k + N$;
12. end of while
13. $\text{subSum}[\text{threadIdxInBlock}] \leftarrow$ total;
14. sync threads in block;
15. if $N > 32$
16. inter-warp parallel reduction in $\text{subSum}$;
17. intra-warp parallel reduction by shuffle operation;
18. else
19. intra-warp parallel reduction by shuffle operation;
20. end of if
21. sum $\leftarrow$ reduction result;
22. if threadIdxInGroup $= 0$
23. $y[i] \leftarrow$ sum;
24. end of if
25. $i \leftarrow i + \text{stepOfRows}$;
26. end of while
```
5. Experiment Results

The results shown in this section were obtained on a single Kepler GTX 650 GPU card manufactured by GIGA-BYTE, featuring 384 SPs running at 1058 MHz and 1 GB of DDR5 DRAM. The host machine that runs Windows 7 has a dual-core AMD Athlon II X 255 processor running at 3.1 GHz and equipped with 8 GB of main memory. Our versatile matrix-vector multiplication kernel is implemented for single-precision arithmetic and compiled by CUDA 5.0 compiler with optimization option /O2. The kernel uses eight bytes (a single float) of shared memory per thread. The performance results were measured with NVIDIA CUDA command line profiler 5.0, and do not include transfer of data between host and GPU. Each result is repeated ten times and the average results are reported. The results will be shown in three parts: In 5.1, we present auto-tuning results of thread assignment for the versatile matrix-vector multiplication kernel on a logarithmic tuning mesh. In 5.2 and 5.3, we compared our versatile matrix-vector multiplication kernel with CUBLAS, MAGMA and the warp method on the performance for different shapes and different rows and columns of matrices.

5.1 Auto-Tuner Results

We choose a 48×48 logarithmic tuning mesh which includes various sizes and shapes of matrices to run the auto-tuner on. For each tile of the tuning mesh, the auto-tuner executes our versatile matrix-vector multiplication kernel ten times at each different template value in the eleven numbers of assigned threads (described in 3.2), and compute an average performance value. Then the auto-tuner selected the number with the maximum performance value to be the best number of assigned threads for the particular size of matrix. The execution time of the auto-tuner on our test platform took about 6 hours.

Figure 5 shows the auto-tuner results. The black area represents the sizes of matrices that do not fit into memory on the graphics card. In this figure we can see that all the eleven allowed numbers of assigned threads appear. The best numbers of assigned threads differ distinctively between different shapes and sizes of matrices and present a special pattern, which is that the best number of assigned threads increases with increasing column size of matrix and varying the shape of matrix from tall-thin to wide-short. These results can be seen as a strong advocation for using auto-tuning for matrix-vector multiplication kernels on GPUs. On basis of the auto-tuner results we created a 48×48 lookup table which is used to determine the best number of assigned threads for our versatile matrix-vector multiplication kernel according to the number of rows and columns of the input matrix.

5.2 Performance Comparison of Different Shapes

In Fig. 6, we compared the performance in GFLOPS of our matrix-vector multiplication kernel with that of the SGEMV routine from the current version of the most commonly used numerical libraries for GPUs: CUBLAS 5.0, the SGEMV routine from the open source project: MAGMA 1.3 and the matrix-vector multiplication kernel based on the warp method [21] for different shapes of matrices as a function of memory footprint. The shapes are denoted as tall, square, and wide, and given by sizes 1000M×M, M×M, and M×1000M, respectively, for M = 10, 20, . . . , 30. In Fig. 6(a), the SGEMV function of the CUBLAS 5.0 library performs best for tall matrices, but worst for wide matrices. For square matrices the performance shows great fluctuation when the size of matrix is great than 10^6. In Fig. 6(b), the SGEMV function of the MAGMA 1.3 library performs similar with that of the CUBLAS 5.0 library for these shapes of matrices, but little better for tall matrices and worse for square matrices. In Fig. 6(c), the performance of the matrix-vector multiplication kernel based on the warp method is better than that of the two methods above for square and wide matrices, but worse for tall matrices. For wide matrices the performance greatly fluctuates when the size of matrix is greater than 10^6. In summary, the performance of all these three SGEMV kernels above is not stable for different shapes of matrices, for example a 10^6 matrix, the performance for tall is about 10 times more than that for wide in Fig. 6(a) and 30 times in Fig. 6(b). In the final analysis, it’s a matter of invariable thread assignment. Correspondingly, in Fig. 6(d) regardless of the shape of matrix, we observe that the three curves show almost the same behavior for our kernel, which is a significant improvement over the similar performance plots for the three SGEMV kernels above.
5.3 Performance Comparison of All Sizes

In Fig. 7, we compared the performance of our matrix-vector multiplication kernel with that of the SGEMV routines from the CUBLAS 5.0, the MAGMA 1.3 and the warp method in color coded form over the 48 x 48 logarithmic mesh of matrix sizes. These three kernels above are not auto-tuned, resulting in the several features in the coloring, which indicate lack of performance for certain shapes or sizes of matrices.

The Fig. 7 (a) and Fig. 7 (b) show that both the CUBLAS 5.0 and MAGMA 1.3 matrix-vector multiplication kernels are performing well, but only for some tall and square matrices, which suggests that they are designed by assigning one thread to compute an element of the result vector. In particular, the performance for wide and fat matrices, which is problematic for this type of kernel, does not meet the hardware’s potential for high-performance. The Fig. 7 (c) shows that the matrix vector multiplication kernel that uses one warp of threads to compute an element of the result vector performs well only for the matrices whose rows and columns are both greater than $10^3$, but not well for the matrices that have the other sizes.

As shown in Fig. 7 (d), our versatile generic matrix-vector multiplication kernel performs quite well for all shapes of matrices. Notice that the performance of our kernel depends primarily on the number of elements in the matrix, and performance increases as the size of the matrix is increased. Moreover, there is a sign of close to optimal size dependency behavior, which is that the figure appears to be almost skew-symmetric. The performance for very tall and skinny matrices is worse than that for comparable wide and fat matrices. As there are more rows in tall matrices and also more elements in the result vector, this results in the matrix-vector multiplication kernel making more store accesses to the global device memory on the GPU card, which are very slow.

6. Conclusion

In this paper, in order to achieve high-performance for all shapes and sizes of matrices we proposed a novel auto-
tuning method for high-performance computing GPUs in which we auto-tuned the number of assigned threads that are used to compute one element of the result vector. As a starting point, on the latest generation of Nvidia's GPU architecture Kepler GTX 650, we selected eleven different numbers of assigned threads from 1 to 1024 and correspondingly designed eleven matrix-vector multiplication kernels with the CUDA programming model. Each kernel aimed for the optimal utilization of the fine-grained parallelism of the GPU hardware, but for different matrix sizes and shapes. The eleven kernels were then combined into a single versatile generic matrix-vector multiplication kernel based on the C++ function templates. We also developed an auto-tuner that can automate the performance tuning process among the eleven kernels by running a large set of benchmarks. The proposed auto-tuning procedure benchmarked on a 48 × 48 logarithmic tuning mesh over various sizes of the matrices. Then a lookup table was created based on the auto-tuner’s results to determine the optimal number of assigned threads according to the rows and columns of the input matrix.

We measured the performance of the matrix-vector multiplication kernel in a series of numerical experiments for different shapes and sizes of matrices, and compared it with that of the SGEMV routine from the most commonly used numerical libraries for GPUs: CUBLAS 5.0, the open source project: MAGMA 1.3 and the warp method. The experiment’s results show that our matrix-vector multiplication kernel can well utilize the computing power of the many-core GPU. The measured performance has very little dependency on the shape of the matrix, which is a significant improvement compared to the other three kernels that do not fully utilize the potential of the GPU’s hardware, resulting in unstable performance for different shapes of matrices.

Acknowledgements

This work is supported by the National Natural Science Foundation of China under No. 61203259, No. 60970060; the Natural Science Foundation of Tianjin, China under No. 08JCYBJC13700.

References

[1] L.S. Blackford, A. Petitet, R. Pozo, et al., “An updated set of basic linear algebra subprograms (BLAS),” ACM Trans. Math. Softw.,
[2] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” Queue, vol. 6, no. 2, pp. 40–53, March 2008.

[3] Nvidia, “CUDA C programming guide v5.0,” NVIDIA Corporation, Oct. 2012.

[4] B. Gaster, L. Howes, D.R. Kaeli, P. Mistry, and D. Schaa, Heterogeneous computing with OpenCL: revised OpenCL 1.2 ed., Morgan Kaufmann, 2012.

[5] Nvidia, CUBLAS library v5.0, NVIDIA Corporation, Oct. 2012.

[6] E. Agullo, J. Demmel, J. Dongarra, et al., “Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects,” J. Phys.: Conference Series, vol. 180, no. 1, 012037, 2009.

[7] H. Goto, “Acceleration of computing the Kleene Star in Max-Plus algebra using CUDA GPUs,” IEICE Trans. Inf. & Syst., vol. E94-D, no. 2, pp. 371–374, Feb. 2011.

[8] J.B. Seo and D.W. Kim, “A parallel implementation of the Gustafson-Kessel clustering algorithm with CUDA,” IEICE Trans. Inf. & Syst., vol. E95-D, no. 4, pp. 1162–1165, April 2012.

[9] Q. Miao, G.J. Wang, and X.G. Lin, “Implementation of scale and rotation invariant on-line object tracking based on CUDA,” IEICE Trans. Inf. & Syst., vol. E95-D, no. 12, pp. 2549–2552, Dec. 2012.

[10] R.C. Whaley, A. Petitet, and J.J. Dongarra, “Automated empirical optimizations of software and the ATLAS project,” J. Parallel Computing, vol. 27, no. 1, pp. 3–35, 2001.

[11] S. Tomov, R. Nath, and J. Dongarra, “Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing,” J. Parallel Computing, vol. 36, no. 12, pp. 645–654, 2010.

[12] X. Zhu, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.H. Wu, “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA,” Proc. ACM Conf. PPoPP’08, Salt Lake City, UT, USA, pp. 73–82, Feb. 2008.

[13] Intel, “Math kernel library,” Intel corporation, www.intel.com/software/products/mkl, accessed May 1, 2013.

[14] N. Fujimoto, “Faster matrix-vector multiplication on GeForce 8800 GTX,” Proc. IEEE Conf. IPDPS 2008, pp. 1–8, Miami, Florida, USA, April 2008.

[15] J. Dongarra and S. Moore, “Empirical performance tuning of dense linear algebra software,” in Performance Tuning of Scientific Applications, pp. 255–272, CRC Press, 2010.

[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel computing.” Proc. IEEE Conf. ICPADS 2008, pp. 30–36, Singapore, Dec. 2012.

[17] M. Harris, “Optimizing parallel reduction in CUDA,” Proc. IEEE Conf. SC07, Reno, Nevada, USA, Nov. 2007.

[18] Nvidia, “CUDA C programming guide v5.0,” NVIDIA Corporation, Oct. 2012.