High-Performance Matrix-Vector Multiplication on the GPU

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 develop a high-performance GPU kernel for one of the most popular dense linear algebra operations, the matrix-vector multiplication. The target hardware is the most recent Nvidia Tesla 20-series (Fermi architecture), which is designed from the ground up for scientific 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 dense matrix-vector multiplication. We show that auto-tuning can be successfully employed to the GPU kernel so that it performs well for all matrix shapes and sizes.

Keywords: GPU, Matrix-Vector Multiplication, Dense linear algebra.

1 Introduction

The single-instruction-multiple-data (SIMD) parallel capabilities of Nvidia GPUs have been made accessible to scientists and developers through the CUDA programming model [1]. The most recent Fermi GPU architecture features up to 16 streaming multiprocessors (SM) having 32 single-precision cores each. Execution on this potent parallel hardware is controlled through CUDA keywords; a block is a 3D structure of up to 1024 threads and a grid is a 2D structure of blocks.

For many programmers, the key to good performance of numerical scientific applications is still linked to the availability of high-performance libraries for the most common dense linear algebra operations. Several such libraries have recently become available for GPUs, e.g., Nvidia's CUBLAS [2] and the open source MAGMA library [3]. In the case of matrix-vector multiplication, however, these libraries are currently not satisfactory and suffer from low utilization of the GPU hardware in particular for rectangular shaped problems [4].

In this paper, we seek to remedy this lack of performance for matrix-vector multiplication for all problem shapes and sizes. We will contribute to the present state of the art of GPU matrix-vector multiplication kernels by developing an auto-tunable rigorously parallel and versatile kernel, where threads can work together, not only within a block, but also between blocks. This provides the kernel with an additional layer of parallelism - at the grid level - which is essential in order to achieve high-performance for rectangular matrix-vector multiplication.

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 377–386, 2012.

The motivation from a parallel computing point of view is to maintain a good load balancing across the GPUs resources in all situations.

2 Related Work

Several previous works on matrix-vector multiplication kernels for GPUs exists of which we will mention some of the most recent. In 2008, Fujimoto [5] described a matrix-vector kernel written in CUDA that was specifically tuned for the Nvidia's GeForce 8800GTX graphics card. The performance he achieved was significantly better than the CUBLAS v1.1 library available at that time, reaching a maximum performance of 36 Gflops in single precision for a GPU with a theoretical memory bandwidth of 86 GB/s. The main design motivation for his kernel was an attempt to maximize data reuse of the $\mathbf{x}$ vector in combination with tiling of the matrix $\mathbf{A}$. This led to important optimizations of the naive matrix-vector implementation such as a two-dimensional block structure and simultaneous reduction operations, which are also adopted in this work.

Later in 2009, Tomov and Dongarra et al. developed a fast matrix-vector kernel to be one of the key ingredients in their MAGMA library [6], which is a dense linear algebra package for heterogeneous CPU-GPU systems with the same functionality as the legacy LAPACK library [7]. Several generic optimization techniques were introduced to improve on the matrix-vector kernel performance, including pointer redirection [8] and auto-tuning [9]. For square matrices of sizes that are divisible by 32, they report a performance of up to 66 Gflops in single precision on a graphics card that has a theoretical memory bandwidth of 141 GB/s [6]. This result is a significant improvement over the CUBLAS 2.3 that was available in 2009. 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 43 Gflops, which was more than twice of what CUBLAS 2.3 could deliver.

3 Matrix-Vector Multiplication Kernels

In this section, we describe the matrix-vector multiplication kernels we have developed for the C2050 card. To achieve high-performance for all shapes of matrix $\mathbf{A}$ we implement four different kernels to fit the four cases; very tall, tall and skinny, close to square, and wide and fat. The cases and the names of the kernels are illustrated in Fig. 1. We also combine the four kernels into a versatile generic kernel. We consider only the case of column major memory layout. In the next section we introduce auto-tuning of the versatile kernel in order to automatically select the best performing of the four kernels at runtime.

3.1 One Thread per Row

The typical implementation of a matrix-vector multiplication kernel, as illustrated in Fig. 2 (a), is where each thread performs a dot product between one

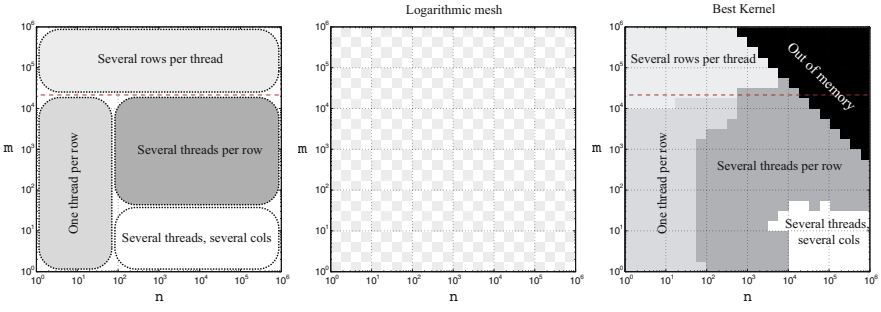

Fig. 1. Left; Four matrix-vector multiplication kernels designed to perform well at different shapes $m \times n$ of $\mathbf{A}$. Middle; Tuning mesh. Right; Best kernel in practice. The dashed line indicates the minimum 21504 rows needed in $\mathbf{A}$ for full occupancy of the Nvidia Tesla C2050 card in a one-thread-per-row kernel. Note the logarithmic axes.

row of $\mathbf{A}$ and $\mathbf{x}$ to produce one element of the result $\mathbf{y}$. The threads are then grouped in 1D blocks along the columns of $\mathbf{A}$. For a given size of $\mathbf{A}$, the only parameter required is the number of threads per block, which we will denote by `BLOCKSIZE`. The size of the grid specified when launching the kernel in CUDA is determined by the `BLOCKSIZE` parameter. Dividing the m rows of $\mathbf{A}$ into slices of size `BLOCKSIZE`, with the last slice possibly containing less than `BLOCKSIZE` rows, we have a one dimensional grid of size

$$\texttt{GRIDSIZE\_m} = (m + \texttt{BLOCKSIZE} - 1)/\texttt{BLOCKSIZE}.$$

Using a grid of this size requires an if conditional inside the kernel to make sure the last block does not access memory outside the m rows of $\mathbf{A}$. In Fig. 2 (a) the kernel is shown for a `GRIDSIZE_m` of 4 as indicated with the red 4×1 mesh.

Since all threads need the same n values of $\mathbf{x}$ for their dot products it is best to read these into shared memory once per block and then let threads access them from there. This allows for maximum reuse of the data. We therefore divide $\mathbf{x}$ into chunks of `BLOCKSIZE` and set up a loop to let the threads collaborate in reading chunks in a coalesced fashion into a shared memory once per block. It requires the allocation of a shared memory array of size `BLOCKSIZE` for each block. The usage of shared memory is illustrated by red-dotted boxes in Fig. 2.

The one-thread-per-row matrix-vector multiplication kernel is appropriate as a high-performance kernel on the C2050 card for tall and skinny $\mathbf{A}$ only. This is because the Fermi GPU with 14 SMs supports 1536 active threads per SM [10], so that full occupancy requires $1536 \times 14 = 21504$ rows in $\mathbf{A}$. If m is less than this, and $\mathbf{A}$ is not skinny, then we are not utilizing the hardware to the maximum. SMs might be idle or running at low occupancy during kernel execution, while the running threads might do a lot of work each. If $\mathbf{A}$ is skinny, e.g., $n < 100$, then dispite the low utilization, the individual threads complete fast enough for this kernel to be the best implementation. In Fig. 1, we indicate the dimensions of $\mathbf{A}$ for which the one-thread-per-row kernel is designed to perform well.

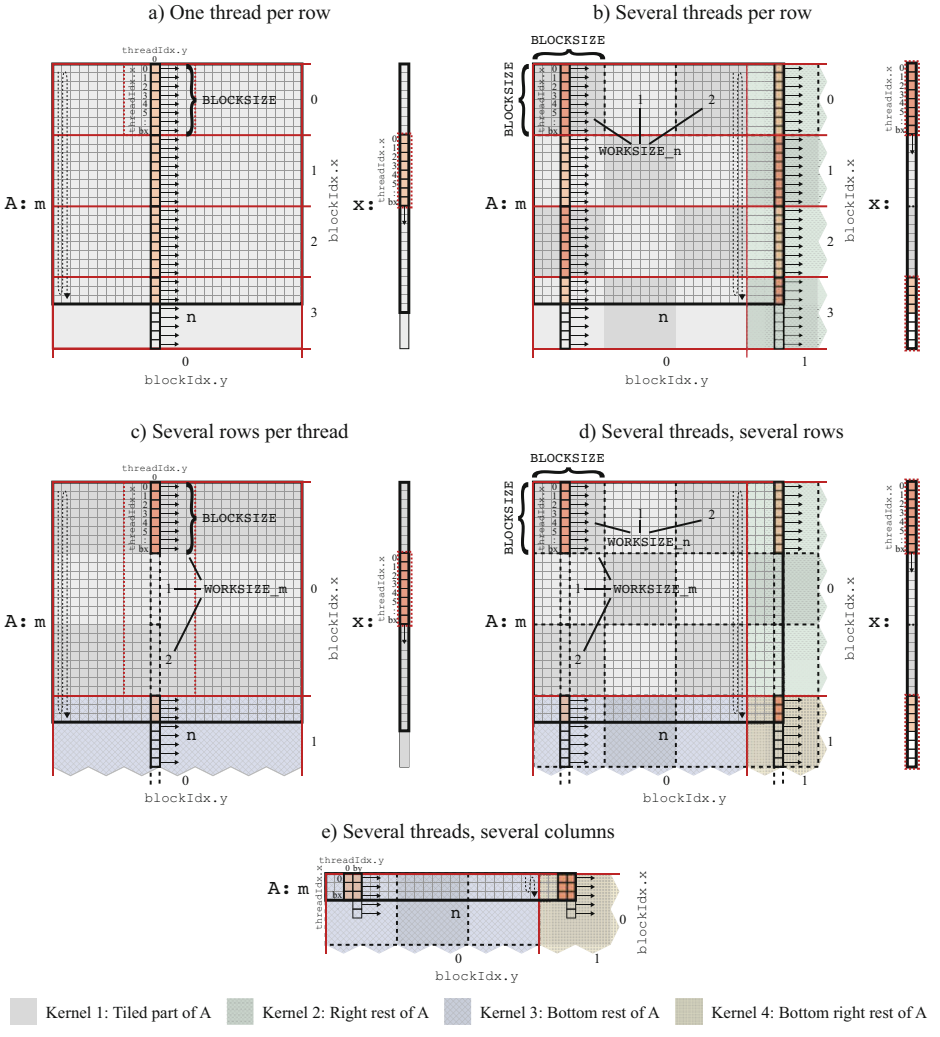

Fig. 2. Schematic illustrations of the matrix-vector multiplications kernels implemented in this work. The transversal of **A** can be conveniently separated into distinct device kernels $1-4$ as indicated by color in the figures. The red lines show the division of the elements of **A** into work-chunks and the CUDA keywords `blockIdx.x` and `blockIdx.y` show how to map these onto a grid of blocks. Vector **x** is read to shared memory for data re-use indicated by the red-dotted boxes. `BLOCKSIZE`, `WORKSIZE_m` and `WORKSIZE_n` are tuning parameters. Memory storage is assumed to be column major.

3.2 Several Threads per Row

The low utilization of hardware for the one-thread-per-row kernel when $\mathbf{A}$ is not tall and skinny is mainly due to the lack of grid-level parallelism in the kernel design. A Fermi GPU can support up to 8 resident blocks per SM, giving up to 112 blocks for full utilization, which is out of reach for shorter $\mathbf{A}$ using a reasonable `BLOCKSIZE` and a 1D grid. The utilization can be improved by allowing several threads per row and thereby introducing a 2D grid for the kernel.

As illustrated in Fig. 2 (b), each thread of each block in this kernel then does part of a row only and adds its partial result to the results of other threads from other blocks in order to produce an element of $\mathbf{y}$. We introduce a new parameter `WORKSIZE_n` to designate how many elements of a row each thread should handle. For simplicity in our implementation this parameter represents multiples of the parameter `BLOCKSIZE`. The values of $\mathbf{x}$ are still read into shared memory in chunks of size `BLOCKSIZE` only once per block and then accessed from here to facilitate reuse of data. We use the CUDA function `atomicAdd()` [1] for the inter-block reduction of partial results in order to avoid race conditions.

The several-threads-per-row kernel is launched with a 2D grid of dimension (`GRIDSIZE_m`, `GRIDSIZE_n`), where

$$\texttt{GRIDSIZE\_m} = (m + \texttt{BLOCKSIZE} - 1)/\texttt{BLOCKSIZE},$$

$$\texttt{GRIDSIZE\_n} = (n + \texttt{BLOCKSIZE} * \texttt{WORKSIZE\_n} - 1)/(\texttt{BLOCKSIZE} * \texttt{WORKSIZE\_n}),$$

and requires an if conditional in the kernel to make sure the bottom blocks do not access memory outside the m rows of $\mathbf{A}$. Since only the right most column of blocks require an if conditional to stay within the columns n of $\mathbf{A}$, it is convenient to design this kernel as two device kernels, 1 and 2, that takes care of the left fully tiled part of $\mathbf{A}$ and the right rest of $\mathbf{A}$, respectively. Device kernels in CUDA work similarly to inline functions in C++. Threads in the fully tiled part of $\mathbf{A}$ add up results for a fixed number of elements `BLOCKSIZE` $*$ `WORKSIZE_n`. Threads in the right rest part of $\mathbf{A}$ possibly do less. In Fig. 2 (b) the case of `WORKSIZE_n` $= 3$ and grid dimension $(4, 2)$ is shown.

As is illustrated in Fig. 1, the several-threads-per-row design performs well for most shapes of $\mathbf{A}$, i.e., those that are close to square or wide. The most significant performance limitation for this kernel is the use of the `atomicAdd()` function, which reads a 32-bit word in global memory, adds a number to it, and writes the result back to the same address. No other thread can access the address until the operation is complete, so until then those other threads working the same row might be stalled. As a rule of thumb, we find that if $\mathbf{A}$ has less than the 21504 rows needed for full occupancy of all SMs on the C2050 card and more than $n > 100$ columns, the gain from an increase in grid-level parallelism and hardware utilization significantly outweighs the loss from having stalled threads.

3.3 Several Rows per Thread

If $\mathbf{A}$ has more than 21504 rows it becomes less beneficial to have more threads per row since all SMs can have the supported 8 active blocks utilized with one-thread-per-row if we use less than 192 threads per block. In fact, for cases where

A is very tall, e.g., having hundreds of thousands of rows, it is a major advantage to let each thread handle several rows. The performance gain from doing this is mainly related to the decrease in shared memory accesses for elements of **x** when each thread handles more rows.

We have implemented a several-rows-per-thread kernel which is illustrated in Fig. 2 (c). In addition to the parameter `BLOCKSIZE` we introduce the parameter `WORKSIZE_n` to designate how many rows each thread should handle. The kernel is launched with a 1D grid of dimension

$$\texttt{GRIDSIZE\_m} = (m + \texttt{BLOCKSIZE} * \texttt{WORKSIZE\_m} - 1)/(\texttt{BLOCKSIZE} * \texttt{WORKSIZE\_m}),$$

and only the bottom block requires an if conditional to stay within the rows m of **A**. The other blocks assigned to the top fully tiled part of **A** always work on the same fixed number of rows. Again, it is convenient to design this kernel as two device kernels, 1 and 3 (see figure), that takes care of the top fully tiled part of **A** and the bottom rest of **A**, respectively.

3.4 Several Threads, Several Columns

Until now all kernels were designed for 1D blocks having each thread assigned to a different row. However, for matrix-vector multiplication with matrices **A** that have less than `BLOCKSIZE` rows this can give rise to a large percentage of idle threads. For matrices with very wide and fat shapes, the performance will significantly decrease when some threads are not working. In order to avoid this it is necessary to use either 2D blocks or index the threads of the 1D block differently, e.g., as illustrated in Fig. 2 (e). The new indexing distributes the threads of a block along the column-wise layout of **A** instead of assigning m of them to distinct rows and leaving the rest idle. As long as there are threads within a block to fill an entire additional column of **A**, these threads will be put to work.

The design of this kernel makes it possible to have several-threads-per-row both within a block and between different blocks and all of them are required to add up their partial results to obtain an element of **y**. This can have a considerable cost on performance, which is also seen from the results in Sect. 4.1, but still makes up the best design for wide and fat shapes of **A**. In our implementation, we again use the CUDA function `atomicAdd()` for the reduction of partial results. Alternatively, one could apply shared memory reduction techniques for the intra-block reduction, e.g., as presented by Harris et. al. [11], but such methods complicates the implementation and does not result in a significant performance boost compared to using `atomicAdd()` on the C2050 card. As indicated in Fig. 1, this kernel is performing well for wide matrices having less than ~ 50 rows.

3.5 Several Threads, Several Rows

In order to have high-performance for all matrix shapes we combine the designs of the four above kernels into a single versatile kernel. The implementation is

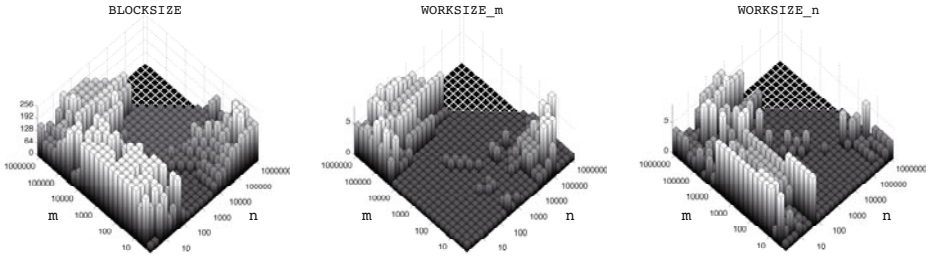

Fig. 3. Result of the auto-tuning process indicating the best values of the tuning parameter `BLOCKSIZE`, `WORKSIZE_m`, and `WORKSIZE_n` at different shapes $m \times n$ of $\mathbf{A}$. See the respective kernel for which the parameters are selected in Fig. 1.

illustrated in Fig. 2 (d) and requires three parameters `BLOCKSIZE`, `WORKSIZE_m` and `WORKSIZE_n`. It uses a 2D grid of dimension (`GRIDSIZE_m`, `GRIDSIZE_n`), where

$$\texttt{GRIDSIZE\_m} = (m + \texttt{BLOCKSIZE} * \texttt{WORKSIZE\_m} - 1)/(\texttt{BLOCKSIZE} * \texttt{WORKSIZE\_m}),$$
$$\texttt{GRIDSIZE\_n} = (n + \texttt{BLOCKSIZE} * \texttt{WORKSIZE\_n} - 1)/(\texttt{BLOCKSIZE} * \texttt{WORKSIZE\_n}),$$

and includes as special cases all the previous three kernels.

4 Results

In this section, we present various performance results for the high-performance GPU matrix-vector multiplication kernels developed in this paper. All kernels are implemented for single-precision arithmetic and auto-tuned for optimal performance. We use the Nvidia Tesla C2050 graphics card having 3 GB device memory on a host with a quad-core Intel(R) Core(TM) i7 CPU operating at 2.80 GHz. The GPU has 448 cuda cores with a peak performance of 1.03 GFlops and a theoretical bandwidth peak of 144 GB/s (ECC is on). Note that the performance timings do not include transfer of data between host and GPU.

4.1 Auto-tuner Results

We run the auto-tuner on a 24×24 logarithmic tuning mesh (see Fig. 1) to find the best matrix-vector multiplication kernel (from 3 implementations) and the best parameters from an exhaustive search of the parameter space

$$\texttt{BLOCKSIZE} \in \{32, 64, 96, 128, 160, 192, 224, 256\},$$
$$\texttt{WORKSIZE\_m, WORKSIZE\_n} \in \{1, 2, 3, 4, 5, 6, 7, 8\},$$

corresponding to $3 \times 8 \times 8^2 = 1536$ kernels for each particular size of $\mathbf{A}$. In order to increase the quality of the kernel selection for this very coarse tuning mesh, the auto-tuner is set up to measure performance on a finer 3×3 logarithmically

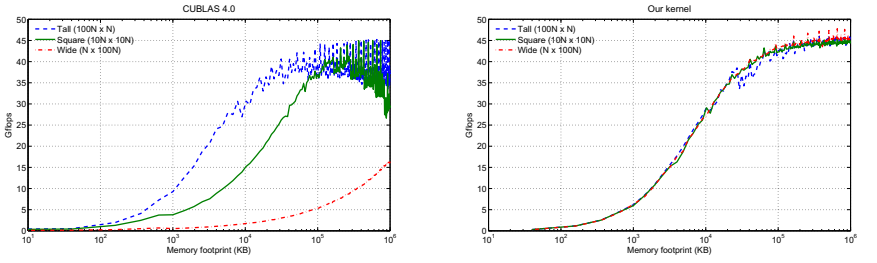

Fig. 4. Performance of matrix-vector multiplication (`SGEMV`) on a Nvidia Tesla C2050 graphics card for matrices having different shapes (tall, square, and wide) as a function of memory footprint. The curves are obtained by calling `cublasSgemv` in the CUBLAS 4.0 library (left) and our auto-tuned kernel (right) and show the average throughtput in Gflops from ten subsequent calls. Note the logarithmic memory footprint axis.

spaced grid of points within each mesh tile and take the average. The execution time of the auto-tuner on our test platform is 1.8 hours.

In the right part of Fig. 1 we show the auto-tuner result for finding the best kernel out of the four kernels described in Sect. 3. The black area represents the sizes of $\mathbf{A}$ that do not fit into memory on the graphics card. We see that the region of best performance for each kernel corresponds reasonably well to their target region, as illustrated in the left part of the figure.

Fig. 3 shows the best values of the tuning parameters `BLOCKSIZE`, `WORKSIZE_m`, and `WORKSIZE_n`, which was determined by the auto-tuner when selecting the best kernel. For all three parameters, we see that the full range of allowed values are used. The best parameters differ distinctively between kernels, however, with no clear pattern otherwise. These results can be seen as a strong advocation for using auto-tuning for matrix-vector multiplication kernels on GPUs.

4.2 Performance Results

In Fig. 4 we plot the performance in Gflops of our matrix-vector multiplication kernel for different shapes of matrices as a function of memory footprint. The shapes are denoted as tall, square, and wide, and given by sizes $100N \times N$, $10N \times 10N$, and $N \times 100N$, respectively, for $N = 10, 20, \ldots$. Regardless of the shape of $\mathbf{A}$, we observe that the curves show generally the same behavior for our kernel, which is a significant improvement over the similar performance plots for the SGEMV function of the CUBLAS 4.0 library [2] shown on the left.

We note that there are several drops in the tall shape performance in the region starting around 3×10^4 MB and ending at 2×10^5 MB, which is linked to the coarse granularity of the tuning mesh. In this region, the tuning parameters change rapidly (see Fig. 3). Since the several-rows-per-thread kernel, which is selected as the best in this region, is quite sensitive to these parameter changes, a more fine-grained mesh is needed for the kernel to be optimally auto-tuned.

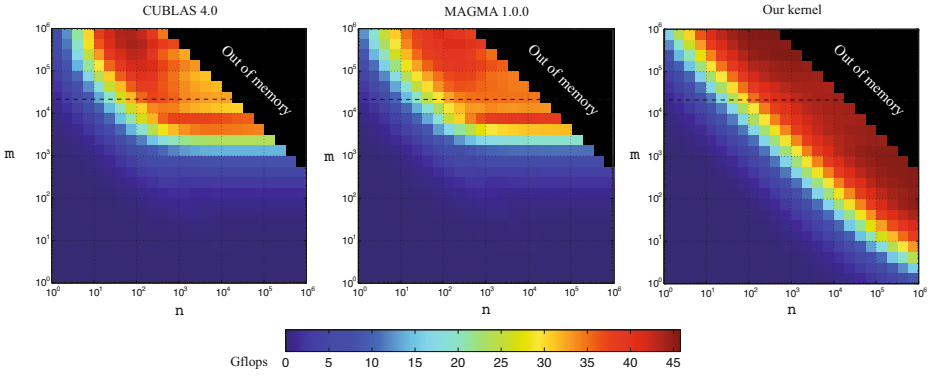

Fig. 5. Performance of matrix-vector multiplication (SGEMV) in color coded form over the 24×24 logarithmic auto-tuning mesh of matrix sizes. Dark blue represents low performance, while dark red represent high performance. The figures compare results from the current versions of the most commonly used numerical libraries for GPUs, the Nvidia CUBLAS 4.0 and the MAGMA 1.0.0, to our auto-tuned kernel.

4.3 Performance Comparison

In Fig. 5, we present the performance of our matrix-vector multiplication kernel in color coded form over the 24×24 logarithmic auto-tuning mesh of matrix sizes. We also show the corresponding performance of the SGEMV routine from the current versions of the most commonly used numerical libraries for GPUs, the CUBLAS 4.0 [2] and the MAGMA 1.0.0 [3]. The performance measurements displayed correspond to averages over 3×3 logarithmically spaced sample points within each mesh tile. We would like to stress that the matrices in these numerical tests are not padded in any way to increase performance.

The figures show that both the CUBLAS 4.0 and MAGMA 1.0.0 matrix-vector multiplication kernels are performing well only above the dashed line (21504 rows), which suggests that they are designed as one-thread-per-row kernels. In particular, the performance for wide matrices, which is problematic for this type of kernel, does not meet the hardware's potential for high-performance. Moreover, the kernels are not auto-tuned, resulting in the several features in the coloring, that indicate lack of performance for certain sizes of matrices.

We see that the figure for our kernel shows good performance for all shapes of matrices, depending primarily on the number of elements in $\mathbf{A}$. The figure appears to be almost skew-symmetric, which is a sign of close to optimal shape-dependence behavior. For very wide and fat matrices, the performance is not as good as for comparable tall and skinny matrices. This is related to the necessary use of the CUDA function `atomicAdd()` for the reduction of partial results to the same output address in the several-threads-several-cols kernel.

5 Conclusion

In this paper, we have developed a high-performance matrix-vector multiplication kernel in the CUDA programming model for the latest generation of Nvidia's high-performance computing GPUs. As a starting point, we designed four different matrix-vector multiplication kernels, each aimed for optimal utilization of the fine-grained parallelism of the GPU hardware, but for different matrix shapes. The four kernels were then combined into a single versatile kernel.

We used auto-tuning of the kernel in order to achieve a high-performance for all problem sizes. The auto-tuning consisted of a heuristic search of a tuning space containing the kernel design and key hardware dependent arguments that sets the number of threads per block, the number of rows per thread, and the number of columns per tread, respectively. The proposed auto-tuning procedure then required a total of 1536 different kernels to be compiled and benchmarked on a 24×24 logarithmic tuning mesh over sizes of the matrix $\mathbf{A}$.

The performance of the matrix-vector multiplication kernel was measured in a series of numerical experiments for different problem sizes. The obtained performance increases as the size of $\mathbf{A}$ increases, until the matrix-vector multiplication kernel can fully utilize the many-core hardware of the GPU. There was very little dependence on the shape of the matrix in the performance of our kernel, which is a significant improvement compared to the current GPU libraries for dense linear algebra, CUBLAS 4.0 and MAGMA 1.0.0, which only reach the GPU hardware's potential for tall matrices.

References

1. NVIDIA Corp.: CUDA C Programming Guide Version 4.0 (2011)
2. NVIDIA Corp.: CUDA CUBLAS Library (2011)
3. Tomov, S., Nath, R., Du, P., Dongarra, J.: MAGMA v0.2 Users' Guide (2009)
4. Sørensen, H.H.B.: Auto-tuning Dense Vector and Matrix-Vector Operations for Fermi GPUs (2011) (submitted)
5. Fujimoto, N.: Faster matrix-vector multiplication on GeForce 8800GTX. In: IEEE International Symposium on Parallel and Distributed Processing (2008)
6. Tomov, S., Nath, R., Dongarra, J.: Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. Parallel Computing 36(12) (2010)
7. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J., Dongarra, J.J., Du Croz, J., Hammarling, S., Greenbaum, A., McKenney, A., Sorensen, D.: LAPACK Users' guide, 3rd edn. SIAM, Philadelphia (1999)
8. Nath, R., Tomov, S., Dongarra, J.: Accelerating GPU kernels for dense linear algebra (2009)
9. Li, Y., Dongarra, J., Tomov, S.: A Note on Auto-tuning GEMM for GPUs (2009)
10. NVIDIA Corp.: Fermi, Whitepaper (2009)
11. Harris, M.: Optimizing Parallel Reduction in CUDA. NVIDIA Dev. Tech. (2008)