The GPU on the Matrix-Matrix Multiply: Performance Study and Contributions

José María CECILIA [a] José Manuel GARCÍA [a] Manuel UJALDÓN [b]
[a] *Computer Engineering and Technology Dept., Univ. of Murcia (Spain)*
[b] *Computer Architecture Department, University of Malaga, Malaga (Spain)*

Abstract.
Modern graphics processing units (GPUs) have been at the leading edge of increasing chip-level parallelism over the last ten years, and the CUDA programming model has recently allowed us to exploit its power across many computational domains. Within them, dense linear algebra algorithms emerge like a natural fit for CUDA and the GPU because they are usually inherently parallel and can naturally be expressed as a blocked computation. In this paper, we extensively analyze the GPU programming and performance of one of the fundamental building blocks in numerical lineal algebra algorithms: The Matrix-Matrix Multiply. Different programming approaches and optimization techniques have already been published in the literature, which we review and analyze to pursue further optimizations and unveil the potential of some hardware resources when programming the GPU under CUDA. Experimental results are shown on a GeForce 8800 GTX and a Tesla C870 GPU with a performance peak of 43 GFLOPS.

Keywords. Graphics Processors, Linear Algebra, High-Performance Computing, CUDA Programming.

1. Introduction

Driven by the demand of the game industry, Graphics Processing Units (GPUs) have completed a steady transition from mainframes to workstations to PC cards, where they emerge nowadays like a solid and compelling alternative to traditional computing, delivering extremely high floating point performance at a very low cost. This fact has attracted many researchers and encouraged the use of GPUs in a broader range of applications, where developers are required to leverage this technology with new programming models which ease the developer's task of writing programs to run efficiently on GPUs.

Nvidia and ATI/AMD, manufacturers of the popular GeForce and Radeon sagas of graphics cards, have released software components which provide simpler access to GPU computing power. CUDA (Compute Unified Device Architecture) [4] is Nvidia's solution as a simple block-based API for programming; AMD's alternative is called Stream Computing [8]. Those companies have also developed hardware products aimed specifically at the scientific General Purpose GPU (GPGPU) computing market: The Tesla products are from NVIDIA, and Firestream is AMD's product line.

Between Stream Computing and CUDA, we chose the latter to program the GPU for being more popular and providing more mechanisms to optimize general-purpose

applications. More recently, the Apple's OpenCL framework [9] emerges as an attempt to unify those two models with a superset of features, but since it is closer to CUDA and inherits most of its mechanisms, we are confident on an eventual portability for the methods described throughout this paper without loss of generality.

The Matrix-Matrix Multiply has traditionally been chosen as benchmark for scoring the highest rates of performance on standard CPU architectures and parallel machines, reporting a number of GFLOPS close to the machine's peak. In GPUs, however, this was not that straightforward, and using shaders and Cg we saw a disappointing period where CPU outperformed most GPU implementations, with only the ATI X800XT producing comparable results to those 12 GFLOPS achieved by a 3 GHz Pentium 4. The advent of CUDA as programming model in 2007 quickly reversed this situation, and significantly faster GPU implementations started to see the light in early 2008.

For example, Ryoo et al. [6] reported 91 GFLOPS, followed by 125 GFLOPS achieved by Nvidia in their CUBLAS library. More recently, Volkov et al. [10] reported 180 GFLOPS in their implementation using a block algorithm similar to those used for vector computers, enabling GPU registers and per-block shared memory to store the data blocks. As the GPU has an unusually large register file, this can be used as the primary scratch space for the computation.

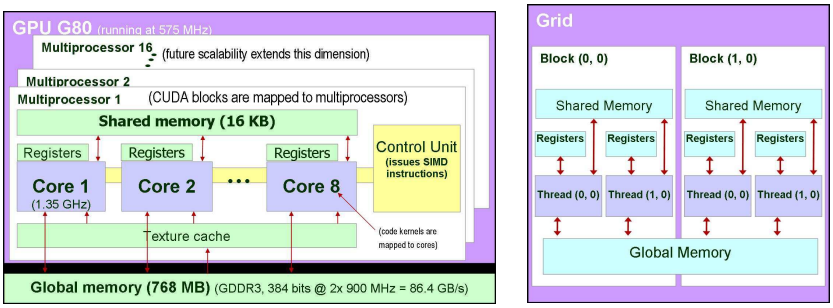
While writing a basic dense Matrix-Matrix Multiply kernel is a fairly simple exercise (see [5] for details), achieving this high level of performance requires much more dedication. This paper tries to illustrate the basic development cycle to achieve this goal while providing the keys for the success.

2. The CUDA programming model and hardware interface

Modern GPUs are powerful computing platforms recently devoted to general-purpose computing using CUDA (Compute Unified Device Architecture) [4]. As a hardware interface, CUDA started by transforming the G80 microarchitecture related to the GeForce 8 series from Nvidia into a parallel SIMD architecture endowed with up to 128 cores where a collection or threads run in parallel. Figure 1.a outlines the block diagram of this architecture. From the CUDA perspective, G80 cores are organized into 16 multiprocessors, each having a set of 32-bit registers, constants and texture caches, and 16 KB of on-chip shared memory as fast as local registers (one cycle latency). At any given cycle, each core executes the same instruction on different data (SIMD), and communication between multiprocessors is performed through global memory.

As a programming interface, CUDA consists of a set of C language library functions, and the CUDA-specific compiler generates the executable for the GPU from a source code where the following elements meet (see Figure 1.b):

1. A program is decomposed into **blocks** that run *logically* in parallel (physically only if there are resources available). Assembled by the developer, a block is a group of threads that is mapped to a single multiprocessor, where they can share 16 KB of memory.
2. All **threads** of concurrent blocks on a single multiprocessor divide the resources available equally amongst themselves. The data is also divided amongst all of the threads in a SIMD fashion with a decomposition explicitly managed by the developer.



(a) (b)

Figure 1. CUDA highlights: (a) Hardware interface for the Nvidia G80 GPU. (b) Programming model.

3. A **warp** is a collection of threads that can actually run concurrently (with no time sharing) on all of the multiprocessors. The developer has the freedom to determine the number of threads to be executed (up to a limit intrinsic to CUDA), but if there are more threads than the warp size, they are time-shared on the actual hardware resources.
4. A **kernel** is the code to be executed by each thread. Conditional execution of different operations can be achieved based on a unique thread ID.

In the CUDA model, all of the threads can access all of the GPU memory, but, as expected, there is a performance boost when threads access data resident in shared memory, which is explicitly managed. In order to make the most efficient usage of the GPU's computational resources, large data structures are stored in global memory and the shared memory should be prioritized for storing strategic, often-used data structures.

3. Benchmarking GPUs to Tune Matrix-Matrix Multiply

3.1. Our metrics

We have used several metrics to evaluate real performance and bandwidth attained on the GPU when running the Matrix-Matrix Multiply.

For performance, we use **FLOPS (Floating-Point Operations Per Second)**, by taking the operations from the PTX code generated by the NVCC compiler with the -ptx flag. This is an internal representation later used by the back-end to produce the actual binary code on a particular platform, so it should be taken as an estimation. Another useful metric we use is **throughput**, calculated as the product between the peak performance and the ratio of FLOPS per instruction, again taken from the PTX code.

For bandwidth with off-chip video memory, we use **GB/sc. (Gigabytes per second)**, calculated by multiplying four magnitudes: the number of accesses to device memory (from the PTX code), the amount of bytes transferred on each access, the actual number of cores the GPU has, and the core clock frequency.

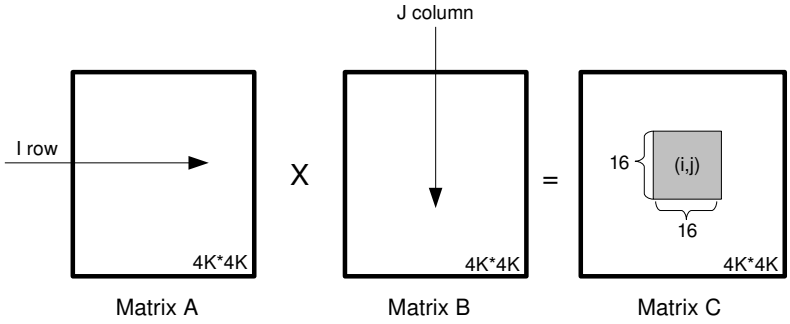


Figure 2. Threads organization for the initial version of our Matrix-Matrix Multiply.

3.2. Departure point

Our first version of the Matrix-Matrix Multiply does not exploit the benefits of the shared memory. One thread is created to produce each output result, loading a row from source matrix A and a column from source matrix B (see Figure 2). Products are performed on pairs of elements from device memory, accumulating them on and on (see Figure 2).

Figure 3 outlines this version, where indexes are calculated based on thread and block coordinates. We use 9 registers per thread, and each block contains 256 threads. Thus, each multiprocessor holds 3 blocks, reaching its maximum of 768 threads and being fully utilized.

```
#define  MATRIX_SIZE  4096
#define  WA  MATRIX_SIZE
#define  WB  MATRIX_SIZE
#define  WC  MATRIX_SIZE
__global__ void matrixMul (float* C, float* A, float* B)
{
  int bx = blockIdx.x;
  int by = blockIdx,y;
  int tx = thredIdx.x;
  int ty = threadIdx.y;
  int indexA = by*BLOCK_SIZE*WA + ty*WA;
  int indexB = bx*BLOCK_SIZE + tx;
  float Csub = 0.0;
  for (int i=0; i<WA; i++)
  {
    Csub += A[indexA] * B[indexB];
    indexA++;
    indexB += WB;
  }
  int indexC = by*BLOCK_SIZE*WC + bx*BLOCK_SIZE;
  C[indexC+WB*ty+tx] = Csub;
}
```

Figure 3. Kernel code for the initial version of our Matrix-Matrix Multiply.

Table 1 shows the performance for this kernel depending on the matrix size and the platform used, where we see how GeForce 8800 GTX slightly outperforms Tesla C870. On a 4Kx4K matrix size, GeForce delivers 64 GFLOP and 36.36 secs., for a total of 1.76 GFLOPS. By examining the PTX code, there are eight instructions in the inner loop where the kernel spends most of its execution time. One of these intructions is a floating-point fused multiply-add (madd). The peak performance for this code is then 43,2 GFLOPS (128 cores * 2 instructions per core* 1.35 GHz * 1/8 FLOP), far from our achievements and also from those 10.58 GFLOPS reported in [6,7].

The PTX code also reveals two loads in the inner loop, so this kernel has 2/8 load operations for a total bandwidth required of 173 GB/sc. (128 SPs * 1/4 instructions * 4

Matrix size	1024x1024	2048x2048	4096x4096
GFLOPS on Tesla C870	1.51	1.60	1.72
GFLOPS on GeForce GTX 280	1.75	1.86	1.76

Table 1. Performance in GFLOPS for the initial version of our Matrix-Matrix Multiply.

Matrix size	1024x1024	2048x2048	4096x4096
Time for accesing A uncoalesced	0.59 msc.	4.42 msc.	35.02 msc.
Time for accessing B coalesced	0.07 msc.	0.54 msc.	4.25 msc.

Table 2. Performance comparison when accessing to matrix A uncoalesced and matrix B coalesced in device memory. We provide overall accessing time for each matrix on the Tesla C870 GPU.

bytes/instruction * 1.35 GHz). This is twice the bandwidth available on GeForce 8800 GTX, so access to device memory becomes the actual bottleneck for this kernel, which also explains why Tesla C870 is slower (its DDR memory is clocked at 2x800 MHz, versus 2x900 MHz on the GeForce).

To reduce the bandwidth requirements, we first use coalesced accesses and then shared memory, also reorganizing computations to exploit tiling.

3.3. Coalescing accesses to device memory

The goal for coalescing is to organize 16 data accesses in a way suitable for being recovered simultaneosly from device memory in a half-warp, so that all threads within it can compute in parallel. The condition for a coalesced access (see [5]) is that each of these 16 threads accesses to the following address: *HalfWarpBaseAddress* + *N*, where N is the thread id. In our case, threads of the same half-warp access to the same elements in the same row of matrix A, so the access to device memory is fully uncoalesced [5]. At the same time, threads of the same half-warp get the data from different columns in matrix B, leaving accesses fully coalesced [5]. We have created a couple of micro kernels to compare the performance of these two access patterns when accessing to device memory. Table 2 shows the execution times, where the access to B is almost ten times faster on large matrices.

These results encourage us to provide coalesced access for matrix A, and to do so, all threads have to access different elements in a way that thread 0 should access the HalfWarpBaseAddress+0 address, thread 1 should access to the HalfWarpBaseAddress + 1 address, and so on. This fulfillment will be combined with the use of shared memory and tiling in what constitutes our next optimization step.

3.4. Tiled version using shared memory

At this point of our work, the bottleneck is the device memory bandwidth. In order to alleviate the pressure on this device memory, we will perform *tiling* [2] to reuse data located in a lower level of the memory hierarchy. This faster level in CUDA is represented by the shared memory.

Three main steps are needed to implement the tiling technique in CUDA:

1. Copy from device to shared memory all the data used by all the threads in a block. These threads cooperate with each other to load the data efficiently, that is, accesses to device memory are coalesced.

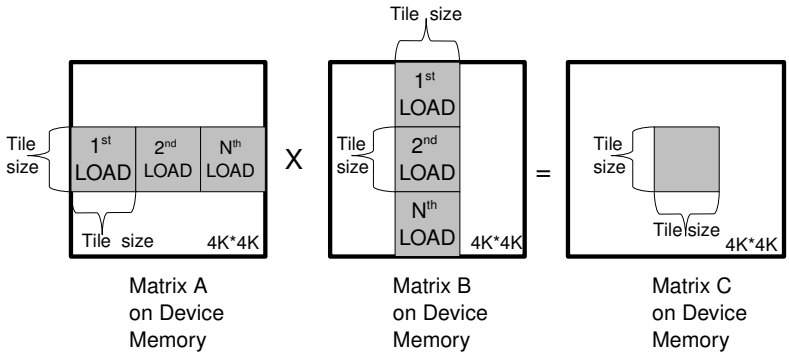


Figure 4. Computing the tiled version of our Matrix-Matrix Multiply.

```
#define MATRIX_SIZE 4096
#define WA MATRIX_SIZE
#define WB MATRIX_SIZE
#define WC MATRIX_SIZE

__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

int indexA = WA*BLOCK_SIZE*by + ty*WA + tx;
int indexB = bx*BLOCK_SIZE + ty*WB + tx;

float Csub = 0.0;
for (int i=0; i<WA/BLOCK_SIZE; i++)
{
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += BLOCK_SIZE;
  indexB += WB*BLOCK_SIZE;
  __syncthreads();
  for (int k=0; k<BLOCK_SIZE; k++)
    Csub += As[ty][k] * Bs[k][tx];
  __syncthreads();
}
int indexC = WB*BLOCK_SIZE*by + BLOCK_SIZE*bx;
C[indexC+WB*ty+tx] = Csub;
```

Figure 5. Tiled version for the code of our Matrix-Matrix Multiply.

2. Perform the actual computation in shared memory, trying to avoid conflicts when accessing memory banks [5].
3. Copy the results back to device memory.

Figure 4 illustrates how tiling is performed on the Matrix-Matrix Multiply. The result matrix C is decomposed in blocks where each thread of a block calculates a single element. Considering the block size as *tile_size* * *tile_size* according to shared memory limitations, this kernel is required to load *tile_size* entire rows from matrix A and *tile_size* entire columns from matrix B, with those rows and columns coordinates matching those of the target block. Figure 5 shows the code for this tiled version.

A major constraint for this tiling technique is imposed by the size of the shared memory, which is 16 KB. in the G80 architecture, and that has to be shared by all the blocks within a multiprocessor. Therefore, depending on the *tile size*, this multiprocessor can execute more or less blocks in parallel. Another upper bounds to account for are described in [5], among which we highlight the 768 threads that may run in parallel on a multiprocessor, and the 8192 registers they are allowed to use.

Table 3 shows the performance we obtain when varying the tile size in our code version. The smaller tile considered, 4x4, has 16 threads per block and uses only 64 bytes per block (16 threads x 4 bytes/thread). Since a maximum of eight blocks may run in

Tile size	No tiling	4x4	8x8	12x12	16x16
GFLOPS for the MxM kernel	1.72	3.76	7.73	11.34	22.10

Table 3. Performance in GFLOPS for different tile sizes of our Matrix-Matrix Multiply using kernels that require 14 registers running on the Tesla C870 GPU.

parallel on a multiprocessor, the shared memory usage is just 512 bytes out of 16 KB. (3.125%) and the amount of extracted parallelism is 128 threads out of 768 (16.7%).

As we increase the tile size, performance improves due to a better use of shared memory together with the cooperation of a higher number of threads in parallel. 16x16 is the largest tile size for the G80 platform, which may reach 256 threads per block as long as the kernel uses ten or les registers. But our kernel requires 14 registers, so only two blocks can be scheduled on a single multiprocessor (2 blocks x 256 threads/block x 14 registers/thread require 7168 registers out of the 8192). Even with this constraint, the 16x16 tiled version improves performance more than any other tile size because it uses more shared memory.

In addition, memory accesses are coalesced in all these tiled versions as computations are rearranged, so device memory bandwidth is fully exploited and data movement between device and shared memory is greatly amortized.

3.5. Increasing arithmetic intensity

An important issue to consider when optimizing codes on graphics processors is *arithmetic intensity*, defined as the percentage of instructions executed on ALUs from a total including branches, memory address calculation, data accesses and so on. One way to greatly improve this parameter through CUDA code transformations is to apply *loop unrolling* [6].

By default, the `nvcc` compiler performs this transformation on small loops like the innermost one in Figure 5. When this is applied, the PTX file shows 63 instructions, and 16 of them are MADD (25.4%). However, without unrolling, the PTX file shows 19 instructions, but only one is MADD (5.2%).

3.6. Optimizing registers usage

Each multiprocessor in our G80 architecture contains 8192 registers which are dynamically partitioned among the threads running on it. When a kernel uses at most ten registers for each of the threads, a maximum of 768 threads can be scheduled per multiprocessor (768 x 10 = 7680 < 8192).

For the 16*16 tiled version of our code with a 4K * 4K matrix size (see Figure 5), we use 14 registers per thread. The PTX code for this kernel also reports that the innermost loop is totally unrolled.

Figure 6 illustrates the development cycle we have followed to minimize the registers usage. With the help of the PTX and `.cubin` files, we can extract valuable information. For example, the `.cubin` file shows the number of registers per thread used by the kernel and the local memory consumed by the kernel, together with some other magnitudes. On the other hand, the PTX code shows an estimation for the registers used by the kernel. The problem with the PTX file is that it uses more registers than those reported by the `cubin` file, so it is difficult to have an idea about the registers usage with only those files. We complement that information with the Decuda application [4],

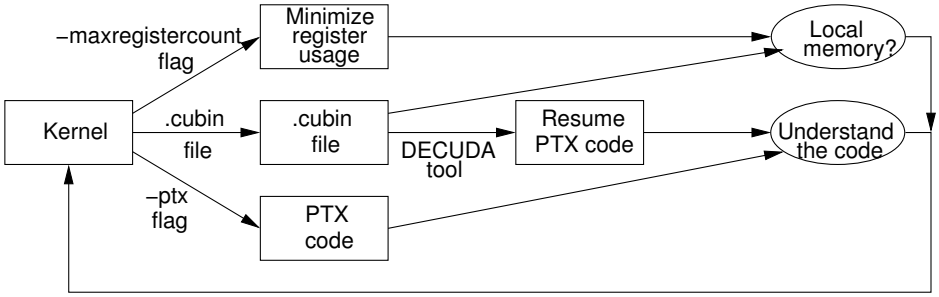


Figure 6. The development cycle we have followed to minimize the registers usage.

which uses the binary code in the `.cubin` file to provide a more comprehensive version for the code generated for the GPU G8x and G9x architectures, including the use of the same number of registers that the `.cubin` file says.

Another useful tool to reduce the number of registers is the compiler flag *-maxregistercount*, which reduces the number of registers used by the kernel at the expense of mapping them onto local memory, which is slower than the registers. However, it is also possible that extra registers can be removed due to compiler optimizations instead of being sent to local memory. So, there is a tradeoff between occupancy and memory speed. Based on our experience, we provide the following heuristics to minimize the number of registers in a CUDA code:

1. Try to use preprocessing instructions instead of instantiate kernel parameters so that the compiler may map those as constants rather than registers.
2. Array indices do not use extra registers, neither block nor thread indices.
3. The tile size has a strong influence on the registers usage. For instance, the 4*4 tiled version uses 9 registers in its unrolling version, whereas 16*16 tiled version uses 14 registers.

3.7. Performance evaluation

Figure 7 shows the results for the experiments we have conducted on different tile sizes and hardware platforms: The Tesla C870 and the Geforce 8800GTX. We also compare them with those published in [6].

Our performance peak is 43 GFLOPS, scored by the 16*16 tiled version unrolled and running on a GeForce 8800GTX (the Tesla C870 performs slightly worse because of a slower memory clock). The registers usage for this version is 10 registers, allowing the maximum of 768 threads to be scheduled per multiprocessor. The PTX code for the 16*16 tiled and unrolled version shows 16 fused multiply-add out of 63 instructions in the main loop. This produces a potential throughput of 87.77 GFLOPS (345.6 Peak * 16/63 MADDs) In terms of device memory bandwidth, 2/63 operations executed during the loop are loads from off-chip memory, which would require a bandwidth of 21.94 GB/s (128 SPs * 2/63 load instructions * 4 bytes/instructions * 1.35 GHz), which is almost 4 times less than GeForce 8800 GTX can deliver. This leads us to conclude that the device memory bandwidth has successfully been removed as a serious bottleneck in the underlying architecture.

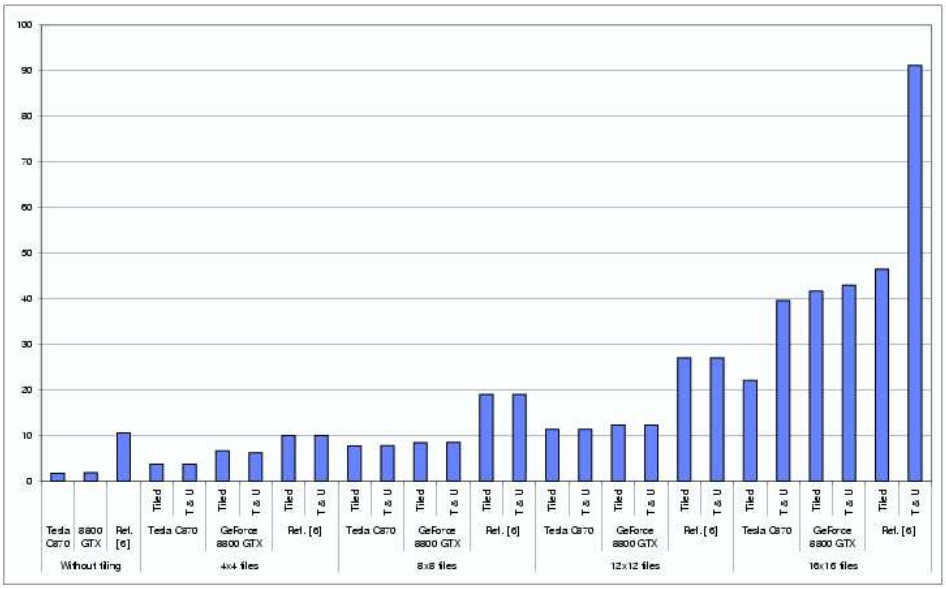


Figure 7. Performance comparison for the Matrix-Matrix Multiply using CUDA in our GPU platforms and with respect to results published in [6] (T & U stands for tiled and unrolling).

4. Summary and conclusions

This work presents a guide for the CUDA implementation of kernels and applications on the GPU using CUDA, taking a typical Matrix-Matrix Multiply as example. This code has been extensively used as benchmark on virtually any existing platform, so we wanted to explore the CUDA capabilities on an emerging and successful architecture like the GPU. We chose two different platforms, the GeForce 8800 GTX and the Tesla C870, to evaluate the influence of three issues: the clock frequency for the cores and the speed and size of the video memory. Our results show that we list them according to their impact in performance, from more to less.

The optimization process was also described as a guideline to tune dense linear algebra codes. Starting from a naive version, we show how to use the shared memory, exploit it efficiently by coalescing accesses and solving conflicts to memory banks, and rearrange the code to increase arithmetic intensity and reduce registers usage. As a result, we reach 43 GFLOPS as performance peak, and device memory bandwidth is removed as the actual bottleneck for the code.

Using CUDA (Compute Unified Device Architecture) to implement scientific applications on the GPU we can fully exploit SIMD programming to populate with work the hundreds of cores the GPU possesses, and these gains will extend in the future thanks to the promising scalability and larger number of cores that GPU architectures will bring to the marketplace at a commodity cost.

References

[1] T. Hartley, U. Catalyurek, A. Ruiz, M. Ujaldon, F. Igual, R. Mayo. *Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores.* Proceedings 22nd ACM International Conference on Supercomputing, 2008 (submitted).

[2] M.D. Lam, E.E. Rothberg, M.E. Wolf. *The cache performance and optimizations of blocked algorithms*. ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, pages 63-74.

[3] W.R. Mark, R.S. Glanville, K. Akeley and M.J. Kilgard. *Cg: A System for Programming Graphics Hardware in a C-like Language.* Proceedings SIGGRAPH 2003, pages 896-907.

[4] Nvidia CUDA. *Home Web Page maintained by Nvidia*. http://developer.nvidia.com/object/cuda.html (accessed, March, 15th, 2009).

[5] Nvidia. CUDA Programming Guide 2.0, 2008.

[6] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, W. Hwu. *Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA*. Proceedings 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM Press, pp. 73-82. February, 2008.

[7] S. Ryoo, C. Rodrigues, I. Christopher, S. Stone, J. Stratton, S. Ueng, S. Baghsorkhi, W. Hwu. *Program optimization carving for GPU computing*. J. Parallel Distributed Computing, Special Issue on General-Purpose Parallel Processing Using GPUs, vol. 68, no. 10, pp. 1389-1401.

[8] Stream Computing. *Home Web Page maintained by AMD*. http://ati.amd.com/technology/streamcomputing/index.html (accessed, January, 31st, 2009).

[9] The Khronos Group. *The OpenCL Core API Specification, Headers and Documentation*. http://www.khronos.org/registry/cl (accessed, March, 15th, 2009).

[10] V. Volkov and J.W. Demmel. *Benchmarking GPUs to tune dense linear algebra*. Proceedings of the 2008 ACM/IEEE conference on Supercomputing. Austin, Texas. November, 2008.