Improving Software Performance in the Compute Unified Device Architecture

Alexandru PIRJAN
Romanian-American University, Bucharest, Romania
alex@pirjan.com

This paper analyzes several aspects regarding the improvement of software performance for applications written in the Compute Unified Device Architecture (CUDA). We address an issue of great importance when programming a CUDA application: the Graphics Processing Unit's (GPU's) memory management through transpose kernels. We also benchmark and evaluate the performance for progressively optimizing a transposing matrix application in CUDA. One particular interest was to research how well the optimization techniques, applied to software application written in CUDA, scale to the latest generation of general-purpose graphic processors units (GPGPU), like the Fermi architecture implemented in the GTX480 and the previous architecture implemented in GTX280. Lately, there has been a lot of interest in the literature for this type of optimization analysis, but none of the works so far (to our best knowledge) tried to validate if the optimizations can apply to a GPU from the latest Fermi architecture and how well does the Fermi architecture scale to these software performance improving techniques.

Keywords: *Compute Unified Device Architecture, Fermi Architecture, Naïve Transpose, Coalesced Transpose, Shared Memory Copy, Loop in Kernel, Loop over Kernel*

1 Introduction

Many software developers have focused their attention lately on General Purpose Computation Graphics Processing Units (GPGPU) as the latest generations of GPU (Graphics Processing Units) architectures are much easier to program than traditional GPUs and offer a significant increase in both memory bandwidth and computational power. A GPGPU has a processing power far beyond than that of a CPU (Central Processing Unit), which is particularly useful in many scientific fields (data extraction, financial data prediction, telecommunication control, neuroscience, medical data analysis). The necessary time for data extraction is considerably reduced as graphics processing units combine hundreds of simplified parallel processing cores, which can be very useful when performing operations on massive data workloads.

Numerous scientific fields like image processing, geometric processing and database can benefit from this high computational power that overcomes the most powerful CPUs. The performance per watt consumed represents another essential aspect when comparing a GPGPU to a classical central processing unit. Taking into account the high performance, low cost and the increasing number of features offered, general-purpose computation graphics processing units prove to be powerful instruments capable of solving an increasingly wide range of applications.

The PeakStream Application Platform [1] from PeakStream executed successfully Monte Carlo simulations for pricing financial instruments. When compared to dual 3.6GHz Xeon processors, the GPU implementation provided a 16X speedup. Scientists from the University of North Carolina at Chapel Hill [2] have developed algorithms for performing fast computation of several common database operations on GPU's (conjunctive selections, aggregations, and semi-linear query). For certain query types, the performance gain was huge.

In this paper, the research is focused on features and generalized optimization methods, on establishing principles and strategies for improving software performance when using the Compute Unified Device Architecture implemented in

the latest generation of graphics processing units (GPU) (like the Fermi architecture). In order to achieve a significant degree of performance and benefit from the Fermi's architecture full potential, massive multithreading must be employed to optimally manage the large number of cores and global memory latency.

2 Related Work

In this section, we briefly review previous work on similar software performance optimizing techniques that are of particular interest for our research.

In [3], authors discuss the GeForce 8800 GTX processor's architecture, features, and generalized optimization strategies. On this platform, the optimization could be achieved by using massive multithreading, taking into account at every step the right balance between each thread's resource usage and the number of simultaneously active threads. The resources to manage include the number of registers and the amount of on-chip memory used per thread, number of threads per multiprocessor and global memory bandwidth. An increase in performance is obtained by reordering accesses to off-chip memory in order to combine requests to the same or contiguous memory locations and it is applied the classical optimizations to reduce the number of executed operations. All these strategies were applied across a variety of applications and domains and achieve between a 10.5X to 457X speedup in kernel. The above-mentioned GPU is capable of impressive performance on a set of disparate non-graphics applications. The paper presents general principles for optimizing applications for this type of architecture, namely having efficient code, utilizing many threads to hide latency and using local memories to alleviate pressure on global memory bandwidth.

In [4] it is described a high-performance parallel radix sort and merge sort routines for many core GPUs, taking advantage of the full programmability offered by CUDA. In order to optimize software performance, authors have carefully designed algorithms that expose substantial fine-grained parallelism and decompose the computation into independent tasks that perform minimal global communication. The optimization techniques made use of the high-speed on-chip shared memory provided by NVIDIA's GPU architecture and efficient data-parallel primitives, particularly parallel scan. They measured the performance on a range of NVIDIA GeForce GPUs: the GTX 280, 9800 GTX, 8800 Ultra, 8800 GT and 8600 GTS. Measurements demonstrate that progressively more parallel devices achieve progressively faster running times.

In [5] we are presented some techiques of optimization for algorithms used in temporal data mining based on the MapReduce programming model. The benchmark has been run on systems using three NVIDIA graphic cards: GeForce 8800, GeForce 9800 GX2 and GeForce GTX 280. In the first technique, the data is stored in texture memory and a strict thread-level parallelism is employed to assign one thread to search for a frequent episode. The second technique uses shared memory to buffer the data prior to searching for a unique episode. Benchmark results highlight the fact that a high-performance implementation on the GPGPU should factor in the problem size, the type of GPU, the type of algorithm and the data-access method when determining the type and level of parallelism. To guide the GPGPU programmer towards optimal performance within such a broad design space, authors provide some general performance characterizations of the data-mining application.

Recently, N. Nakasato [6] has presented benchmark results of optimized dense matrix multiplication kernels for a Cypress GPU (which belongs to AMD's Evergreen family of products). In this paper there are proposed general matrix multiply kernels for single, double and double-double precision. The proposed kernels show 73% and 87% of the theoretical performance of the GPU, respectively. The benchmark leads to some interesting results, including the conclusion that texture cache is very effective on the

Cypress architecture.

3 The Compute Unified Device Architecture (CUDA)

Graphics Processing Units have been used for a long time solely to accelerate graphics rendering on computers. In order to satisfy the increasing need for improved three-dimensional rendering at a high resolution and a large number of frames per second, the GPU has evolved from a one-purpose specialized architecture to multiple purposes complex architectures, able to do much more than just provide video rendering. The acceleration of a broad class of applications became possible once with the introduction of the NVIDIA Compute Unified Device Architecture. The architecture and the main characteristics of the NVIDIA GPUs are summarized in Figure 1.

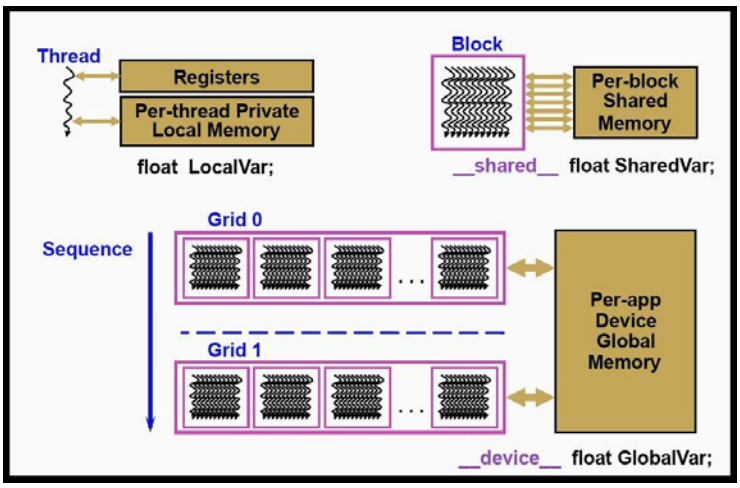

Fig. 1. NVIDIA Compute Unified Device Architecture (CUDA)[7].

CUDA is a software and hardware architecture that enables the NVIDIA graphics processor to execute programs written in C, C++, FORTRAN, OpenCL, Direct Compute and other languages. A CUDA program invokes more parallel program kernels. The kernel processes in parallel each set of parallel threads. The programmer or compiler manages these threads by grouping them into thread blocks (consisting of more threads) and grids of thread blocks (consisting of more thread blocks).

The GPU processor instantiates a kernel program on a grid containing parallel thread blocks. Each thread from the block executes an instance of the kernel and has an unique ID associated to registers, to thread's private memory within the thread block [7].

The Compute Unified Device Architecture hierarchy of threads is mapped to the hierarchy of the graphics processing units' hardware processor; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; the CUDA cores contained in the streaming multiprocessor SM run the threads within blocks. A streaming multiprocessor SM can process up to 32 groups of threads called warps. Regarding memory hierarchy, each multiprocessor contains a set of 32-bit registry with a zone of shared memory, which is easily accessible for each core of the multiprocessor but hidden from other multi-processors. Depending on the generation of a GPU, the number of registry and the size of shared memory vary. Besides shared memory, a multiprocessor contains two read - only memory caches, one for texture and another one for constants.

In order to improve software performance when programming in CUDA, developers have to optimize the number of concomitant active threads and balance each thread's resources: number of registers and threads

per multiprocessor, global memory bandwidth and the amount of on-chip memory assigned per thread. Performance increases have been obtained by reordering accesses to off-chip memory in order to manage requests referring to the same memory locations (or contiguous memory locations). By applying these techniques, many applications improved their execution time up to 457X in kernel codes and 431X at a general level [3].

In the NVIDIA CUDA programming model [7] a system is comprised of a traditional CPU (representing the host) and one or more massively data-parallel coprocessors (representing the devices). The CUDA runtime has library functions for managing both the device memory and transfers from the host to the compute devices.

All concurrent threads are based on the same code even if they may follow different paths of execution because each CUDA device processor supports the Single-Program Multiple Data (SPMD) model [8] and each thread resides in the same global address space. Data parallel functions, called kernels and data structures, corresponding to the compute devices, comply with standard ANSI C extended with keywords. A kernel is usually invoked on thousands of threads and describes the work of a single one. Inside thread blocks, through built-in primitives, threads synchronize their actions and share their data. The CUDA programming model enables a program's components, which are suited for data parallelism, to be separated and executed on a specialized massive data parallelism coprocessor. A detailed overview on the CUDA programming model is depicted in [7].

The G80 on the GeForce 8800 is NVIDIA's first GPU that implements CUDA and has 16 streaming multiprocessors, each of them having 16 KB shared memory and eight streaming processors (SPs) resulting in a total of 128 SPs [7]. Later on, the GT200 architecture implemented in the GeForce GTX200 series succeeded the G80. The amount of streaming multiprocessors is 30 resulting in a total of 240 SPs. The architecture also offered double precision floating-point capability. The newest NVIDIA's architecture is called Fermi and became commercially available on March 26, 2010. This architecture is implemented in the GeForce GTX400 series and it features 16 SMs, each of them having 32 SPs (in the Fermi architecture the streaming processors are called CUDA cores) and 64 KB shared memory which is configurable as larger shared memory or larger L1 cache (48/16 KB or 16/48 KB). The total amount of SPs is 512 and the whole GPU shares a L2 cache of 768 KB. Fermi offers eight times faster double precision performance, IEEE 754-2008 FP precision and error correcting code (ECC) memory, especially required for consistency requirements of scientific computing [7]. A comparison between the three architectures is depicted in Table 1. This architecture offers a high degree of flexibility when it comes about allocating local resources like registers or local memory in threads.

Table 1. Comparison of the three major CUDA GPU architecture specifications (Note that the numbers of streaming processors are maximum values).

Architecture's Codename	**G80**	**GT200**	**Fermi**
Release Year	2006	2008	2010
Fabrication Process	90 nm	65 nm	40 nm
Number of Transistors	681 million	1.4 billion	3.0 billion
Streaming Multiprocessors (SM)	16	30	16
Streaming Processors (per SM)	8	8	32
Streaming Processors (total)	128	240	512
Single Precision FP Capability	128 MAD ops/clock	240 MAD ops/clock	512 FMA ops/clock
Double Precision FP Capability	None	30 FMA ops/clock	256 FMA ops/clock
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB

The programmer divides local resources among threads and every CUDA core can process a variable number of threads. Although this flexibility offers a high degree of control over an application performance, it also has a great impact on optimizing the performance of applications. Another important aspect is related to how the GeForce GTX480 can execute applications and what are the elements that improve or limit its performance. Numerous software applications were ported and evaluated on the CUDA platform as a result of its huge data processing power [9].

According to a study from Stanford University [10], when one chooses to execute code on the CUDA platform, he must follow some major guidelines in order to improve the software performance:

- understand how software maps to architecture,
- use heterogeneous CPU+GPU computing,
- use massive amounts of parallelism,
- understand SIMT (Single Instruction Multiple Thread) instruction execution,
- enable global memory coalescing,
- understand cache behaviour,
- use shared memory,
- optimize memory copies,
- understand PTX (a low-level parallel thread execution virtual machine) instructions.

In order to improve the software performance the following technical issues must be taken into consideration:

- To assure a reduced bandwidth usage and to minimize the redundant execution, a programmer must optimize the use of the on-chip memory. This memory is called shared memory, is software managed and along with a register file it represents the working memory within a group of cores. The shared memory has low latency and is partitioned among all the thread blocks that belong to the same streaming multiprocessor during the runtime. The inter-thread data can be reused because all data in the shared memory is shared among threads from the same thread block. Even if there is a small increase in the registers or shared memory usage per thread, the number of simultaneous executed threads diminishes greatly.
- Using synchronization each thread can communicate only with other threads within the same thread block and there is no communication within threads from other blocks. Therefore, hardware resources do not have to be virtualized and so the hardware becomes highly scalable. The same program written in CUDA can be executed successfully on different generations of GPUs (for example one can use a GTX480 as well as a GTX280) but a single kernel call has a limited parallelism that can be applied.
- Every GPU thread has its own private per thread memory, private registers, program counter and thread execution state. Each thread can execute an independent code path. The GPU processor executes and manages at hardware level hundreds of concurrent threads avoiding scheduling overhead and hiding memory latency. The Fermi architecture offers 512 execution cores; a GTX480 has 480 execution cores available for use. Hundreds of threads are needed for all these cores to be completely occupied. The high latency of global memory is also an important technical issue that must be taken into consideration when a programmer defines the threads in order to improve the software performance in CUDA. While CPU designs use large caches to hide memory latencies, CUDA generates and uses thousands of active threads. In contrast to traditional multicore systems, programmers may have to define threads at a finer granularity in order to assure that there is a sufficient number of threads and see that there is a high compute-to-memory-access ratio in order to avoid saturation of memory channels.

4 Optimizing Performance for Programs Written in the CUDA Programming Model

When algorithms are developed in the CUDA programming model, the basic concern of developers is to divide the work required in fragments that can be processed by a x number of thread blocks, each containing n threads. For optimum performance, it is recommended that the number of thread blocks match the number of processors, although the threads within a block will be executed by more cores within a streaming multiprocessor. The most important factor in achieving performance consists in repartitioning the tasks to be performed between the x thread blocks.

A single thread block can be considered as equivalent to a PRAM model (parallel random-access-machine) which allows processors to behave arbitrarily asynchronous CRCW (concurrent-read, concurrent-write) [11].

Thus, PRAM algorithms are most efficient at block level [4] and they have to be decomposed into separate kernels because of the need for global synchronization of data flows, synchronization that can be achieved only by successive calls of the kernel.

When optimizing an application for the Fermi architecture we have to consider the floating point throughput of an application and the fact that the Fermi architecture now supports by default subnormal numbers at the hardware level and also all four IEEE 754-2008 rounding modes (nearest, zero, positive infinity, and negative infinity) [7].

Subnormal numbers consist of small numbers between zero and the smallest normalized number of a given floating point number system. The GPUs generations prior to Fermi incurred a loss of accuracy by flushing subnormal operands and results to zero. Subnormal numbers are handled at hardware level, allowing values to gradually underflow to zero without a performance penalty unlike the CPUs that perform subnormal calculations in exception-handling software, which consumes thousands of cycles.

In computer graphics, linear algebra and scientific application one often needs to multiply two numbers and add the product to a third number (E.g. $D = A \times B + C$). In prior generations of GPUs the multiply-add (MAD) instruction was used and both operations were performed in a single clock by performing a multiplication with truncation, followed by an addition with round-to-nearest even. Nvidia implemented the new fused multiply-add (FMA) instruction in the Fermi architecture for both 32-bit single-precision and 64-bit double-precision floating-point numbers (The GT200 supports FMA only in double precision). The fused multiply-add instruction brings several improvements when compared to multiply-add by withholding full precision in the intermediate stage (Figure 2). A significant number of algorithms (used in iterative mathematical calculations, rendering fine intersecting geometry) benefit from the increased precision.

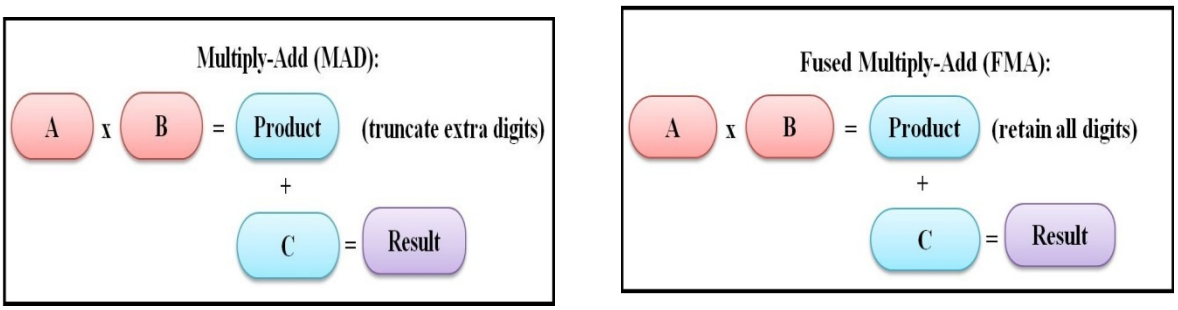

Fig. 2. Differences between the fused multiply-add and the new fused multiply-add

The Fermi architecture offers a Single Precision Floating Point Capability of 512 FMA ops /clock and a Double Precision Floating Point Capability of 256 FMA ops /clock. In order for this performance to be reached, the CUDA streaming processors

must be fully loaded and for this to happen an application must have many threads with few synchronizations without consuming the global memory bandwidth. The kernel speeds up if the number of instructions that do not contribute to data computation is reduced [12].

In order for an application to reach maximum performance, the developers must properly manage global memory latency by creating enough threads to fully load the streaming processors while other threads are pending on global memory accesses. The threads in the CUDA programming model must have a finer granularity than those used for traditional multicore execution. The number of global accesses and long-latency operations in an application determines the necessary number of threads. The available shared memory and registers' size may restrict the number of active threads and generate memory latency.

According to the official documentation "Official CUDA Programming Guide" [7] the limitation of memory can be overcome in two ways. A possible option is the memory paging technique that successively moves portions of memory and then processes them. The developer can use also the CUDA direct access memory option, "zero-copy" but the bandwidth available for this technique is very low and the memory should be declared as "pinned" thus allowing the memory pages to be maintained in real memory all the time. This method is less effective than the paging one as both the GPU and the operating system have limits concerning the pinned memory that is under 4 GB.

The Fermi architecture overcomes all the above-mentioned limitations and significant efforts are made to develop a CUDA programming environment to provide the necessary facilities for the typical programmer. A Fermi GPU can execute and run genuine C++ code as a result of a unified memory hierarchy address space. A programmer can access dynamic arrays in registry memory resulting in enhancements that allow improved execution times for complex algorithms.

Although the GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM, the memory is a significant limitation of the hardware and is still insufficient considering that in practice many databases' sizes are of the order of terabytes or even petabytes.

The global memory bandwidth influences and limits the throughput of the system. In this case, the developers cannot improve performance by increasing the number of threads. The number of simultaneously executed threads is limited by the necessity of reusing data and therefore, it imposes the use of additional registers and shared memory. The usage of these resources is very difficult to balance, often is non-intuitive and some applications will run within the limits of resources which are different from those specified by this architecture.

5 Experimental Results

In this section, we analyze the main aspects of CUDA application performance and GPU memory management through a sequence of progressively optimized kernels as applied to a matrix transpose. In the beginning there are depicted some matrix transpose characteristics, then a few issues regarding the code-behind, performance measurements and a sequence of copy and transpose kernels that progressively address various performance bottlenecks. We address three aspects concerning memory usage: coalescing data transfers to and from global memory, shared memory bank conflicts, partition camping. Shared memory bank conflicts are related to on-chip shared memory (presented in Section 2 of this paper) while coalescing and partition camping relates with data transfers between global device and on-chip memories. The analyzed issues address basic CUDA programming concepts: kernels, threads, blocks of threads, different memory spaces accessible by CUDA threads. A detailed overview for these concepts is presented in [7].

For the transpose optimization we choose a

matrix of floats so that the input and output matrices address different memory locations as recommended by Greg Ruetsch and Paulius Micikevicius in [13]. The Whitepaper recommends using square matrices having a multiple of 32 dimension. If one decides to change these dimensions and choose some arbitrary size matrices, he must make some slight modifications in the source code.

The main analyzed tasks consist in launching and timing of several kernels, data allocation and transfer between the host and the device, validating of the results and freeing host and device memory. For benchmarking purposes, we also run kernels that execute matrix copies, not only matrix transposes. The effective bandwidth is calculated in GB/s for the matrix copy and the transposed one using a NVIDIA GTX480 and a GTX280. The calculated bandwidth is chosen as twice the size of the matrix (corresponding to the operations of reading and writing the matrix) divided by the time of execution as proposed in [13]. In the benchmarking, the following configuration has been used: Intel Core2 Quad Q9550 at 2.8 GHz with 4 GB (2x2GB) of 1333 MHz DDR3 SDRAM. Programming and access to the GPUs used the CUDA toolkit 3.2. RC with NVIDIA driver version 260.93. In addition, all processes related to graphical user interface have been disabled to reduce the external traffic to the GPU.

At the top of the code in the "TranspunereMatrice.cu" file we define the variable "NUMAR_REPETITII" that specifies the number of loops that normalize the effective bandwidth. The looping is executed a "NUMAR_REPETITII" times over the code and the measurement is calculated when looping takes place over kernel and within the kernel as shown below:

```
// masuram lansarea la nivelul kernel
cutilSafeCall( cudaEventRecord(inceput,
0) );
for (int i=0; i < NUMAR_REPETITII; i++)
{
      kernel<<<grid,
threads>>>(device_odata, device_idata,
dimensiune_x, dimensiune_y, 1);
}
cutilSafeCall( cudaEventRecord(sfarsit,
0) );
cutilSafeCall(
cudaEventSynchronize(sfarsit) );
    float TimpKernelExtern;
cutilSafeCall(
cudaEventElapsedTime(&TimpKernelExtern,
inceput, sfarsit) );
cutilSafeCall( cudaMemcpy(host_odata,
device_odata, dimensiune_mem,
cudaMemcpyDeviceToHost) );
    CUTBoolean rezultat =
cutComparef(trans, host_odata,
dimensiune_x*dimensiune_y);
if (rezultat == CUTFalse) {
      shrLog(" %s eroare Kernel \n",
DenumireKernel);
      reusita = CUTFalse;
}

// masuram inauntrul kernel-ului
cutilSafeCall( cudaEventRecord(inceput,
0) );
    kernel<<<grid,
threads>>>(device_odata, device_idata,
dimensiune_x, dimensiune_y,
NUMAR_REPETITII);
    cutilSafeCall(
cudaEventRecord(sfarsit, 0) );
cutilSafeCall(
cudaEventSynchronize(sfarsit) );
    float TimpKernelIntern;
cutilSafeCall(
cudaEventElapsedTime(&TimpKernelIntern,
inceput, sfarsit) );
cutilSafeCall( cudaMemcpy(host_odata,
device_odata, dimensiune_mem,
cudaMemcpyDeviceToHost) );
rezultat = cutComparef(trans,
host_odata, dimensiune_x*dimensiune_y);
if (rezultat == CUTFalse) {
      shrLog(" %s eroare Kernel \n",
DenumireKernel);
      reusita = CUTFalse;
}
```

The timing is achieved through the "for" loop and by passing the variable "NUMAR_REPETITII" to the kernel:

```
for (int i=0; i < NUMAR_REPETITII; i++)
{
  kernel<<<grid,
threads>>>(device_odata, device_idata,
dimensiune_x, dimensiune_y, 1);
    }
……….

 kernel<<<grid, threads>>>(device_odata,
device_idata, dimensiune_x,
dimensiune_y, NUMAR_REPETITII);
```

The two timings presented above differ in the overhead of the kernel launch and must be consistent between different kernels and when calculating the matrix's indices when

the kernel launches.
The looping over kernel acts as a synchronization mechanism because when the kernel is launched multiple times from a loop, all the blocks from inside one kernel launch must have executed completely before another launch can occur. Once every loop, the set of active blocks and memory access patterns reset and thus resources synchronize. When the loop takes place within the kernels it is more likely for the set of active thread blocks to diverge during the timing loop. The two timing code methods represent a useful tool for measuring the overall performance and the data movement times between kernels.
We benchmark the copy and transpose operations on a GTX480 and a GTX280 in order to analyze the optimization methods regarding coalesced global memory accesses and shared memory bank conflicts.
The testing methodology follows the guidelines from NVIDIA [13] and benchmarks the following operations: Simple Copy, Shared Memory Copy, Naïve Transpose, Coalesced Transpose, Bank Conflict Free Transpose, Fine-grained Transpose, Coarse-grained Transpose and Diagonal. The performance of copy and transpose kernels obtained on a 1024x1024 square matrix (composed from 64x64 tiles, each tile having 16x16 size) using a GeForce GTX280 and a GeForce GTX480 are recorded after 10 consecutive runs for each device regarding kernel over and kernel in.
a) Simple copy test. Results are presented in Table 2 and Figure 3.

Table 2. Bandwidth in the simple copy test.

	Loop over kernel (GB/s)		**Loop in kernel (GB/s)**	
Test number	GTX280	GTX480	GTX280	GTX480
1	72.1	110.76	105.87	358.43
2	91.67	114.95	106.08	352.7
3	91.64	118.69	106.03	358.38
4	67.85	118.85	105.91	355.49
5	91.61	106.68	105.86	353.92
6	72.21	118.73	106.07	356.63
7	72.11	118.76	106.32	356.27
8	70.84	107.77	105.95	348.68
9	91.66	118.71	105.86	352.44
10	72.06	118.65	105.89	348.93

Comparing the obtained results in this case, one can observe that in both situations (loop over kernel and loop in kernel) the bandwidth throughput is higher when using the GTX480. Differences are notable in the loop in kernel test, the bandwidth being up to 3 times higher for GTX480 than for GTX280. In the loop over kernel test differences between the two graphic cards are on average 45% higher for the GTX480 than for the GTX280. These differences are justifiable if we take into account the improvements of Fermi architecture.

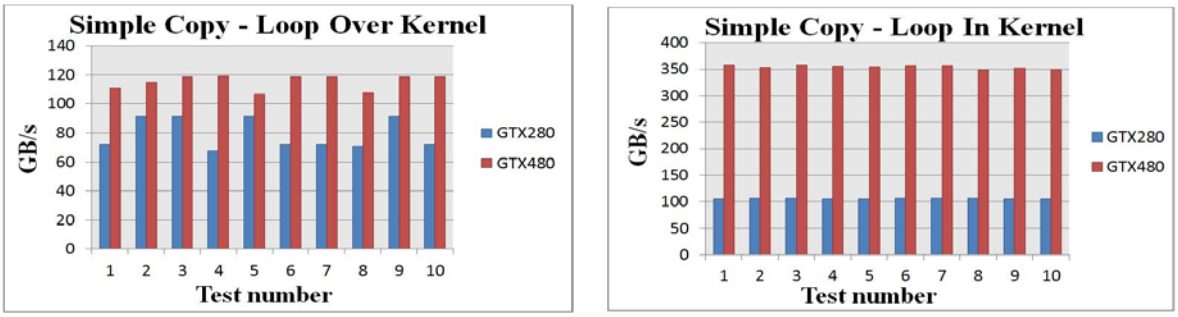

Fig. 3. Simple copy test – graphical results.

b) The naïve transpose test. Results are presented in Table 3 and Fig. 4.

Table 3. Bandwidth in the naïve transpose test.

	Loop over kernel (GB/s)		**Loop in kernel (GB/s)**	
Test number	GTX280	GTX480	GTX280	GTX480
1	2.42	60.6	2.48	92.09
2	2.43	60.72	2.48	92.55
3	2.43	57.9	2.48	92.54
4	2.42	55.38	2.48	92.45
5	2.42	55.41	2.46	92.37
6	2.42	60.65	2.48	92.44
7	2.43	60.6	2.47	92.42
8	2.42	57.06	2.47	92.42
9	2.43	57.02	2.48	92.13
10	2.42	60.55	2.48	92.47

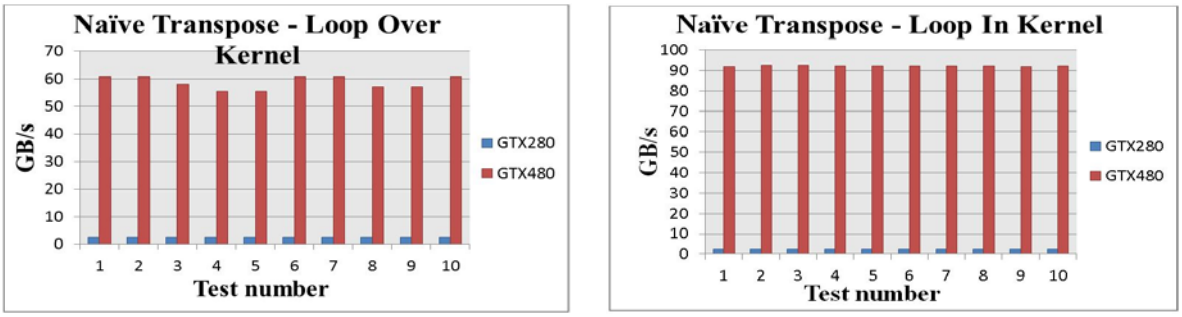

Fig. 4. Naïve transpose test – graphical results.

Comparing the obtained results in the naïve transpose case, one can observe that in both situations (loop over kernel and loop in kernel) the bandwidth throughput is tremendously higher when using the GTX480. The GTX480 has a L1 cache that helps caching temporary register spills of complex programs. The GPUs generations prior to Fermi used registers directly to DRAM and this increased latency. The L1 cache scales the performance tremendously. The GTX480 also features a 768 KB unified L2 cache that provides efficient high speed data sharing across the GPU. Algorithms such as sparse matrix multiplication, physics solvers and raytracing benefit greatly from the cache hierarchy. Differences of performance between the simple copy and the naïve transpose tests can be alleviated through the global memory coalescing optimization technique.

c) Coalesced Transpose. Results are

presented in Table 4 and Figure 5.
GTX480 offers a significant increased performance in the coalesced transpose test, in both situations (loop over kernel and loop in kernel) over the GTX280. In both mentioned above situations the bandwidth throughput is higher in the coalesced transpose test than in the case of the naïve transpose test, but these results are much lower than those obtained in the simple copy test.

Table 4. Bandwidth in the coalesced transpose test.

	Loop over kernel (GB/s)		**Loop in kernel (GB/s)**	
Test number	GTX280	GTX480	GTX280	GTX480
1	17.7	91.96	19.37	167.74
2	18.29	91.7	19.41	167.69
3	18.31	81.53	19.35	167.49
4	17.81	91.76	19.35	167.4
5	17.29	91.68	19.36	167.31
6	17.56	91.86	19.33	167.89
7	18.05	91.85	19.33	167.81
8	17.8	91.86	19.37	167.86
9	17.28	91.9	19.4	167.81
10	18.02	91.69	19.36	167.27

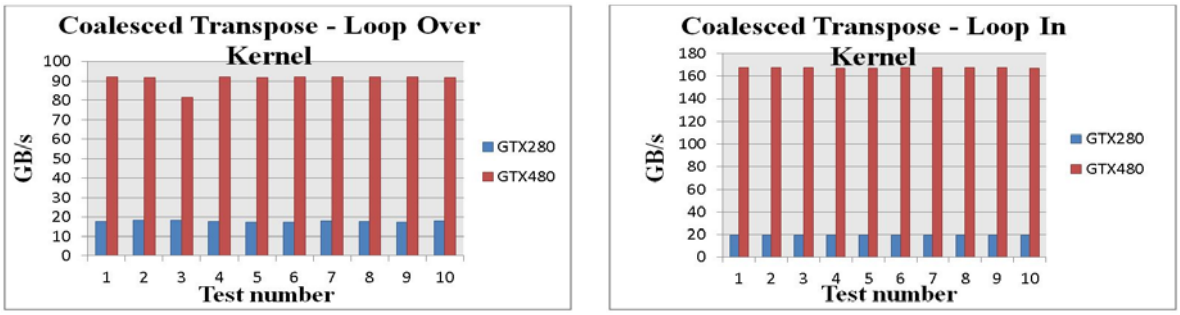

Fig. 5. Coalesced transpose test – graphical results.

A synchronization barrier required in the coalesced transpose explains this performance gap.

d) Shared memory copy. Results are presented in Table 5 and Figure 6.
When comparing results obtained in this test by the GTX280 and the GTX480 one can notice that the second device offers better performance. In this test the copy kernel utilizes shared memory. Threads do not share data during the execution phase and the purpose of this test is to assess the cost of the synchronization barrier (mentioned in the coalesced transpose case). The use of shared memory with a synchronization barrier has little effect on the performance as suggested by the results obtained.

Table 5. Bandwidth in the shared memory copy test.

	Loop over kernel (GB/s)		**Loop in kernel (GB/s)**	
Test number	GTX280	GTX480	GTX280	GTX480
1	38.65	96.71	87.8	192.82
2	39.99	96.85	87.78	193.25
3	39.98	96.67	87.69	193.27
4	36.75	96.69	87.82	193.17
5	34.32	96.77	87.79	193.29
6	36.66	96.82	87.96	192.98
7	38.03	96.69	87.94	192.81
8	37.09	86.8	88.03	193.07
9	36.56	81.36	88	193.65
10	38.31	96.64	88.09	192.76

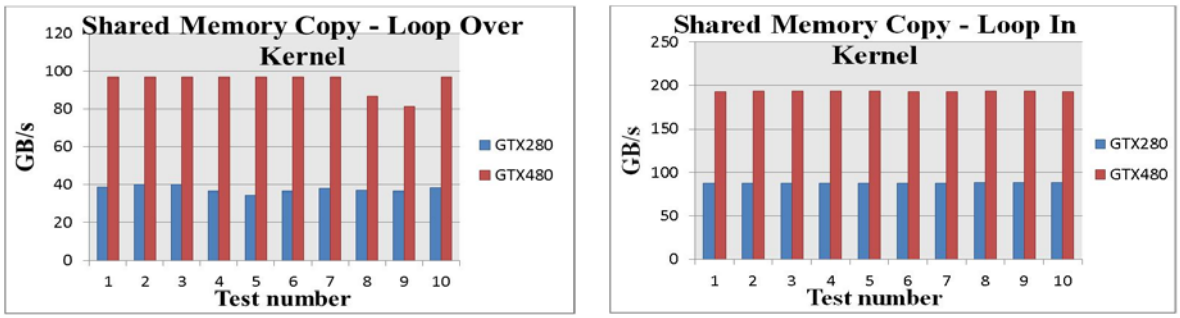

Fig. 6. Shared memory copy test – graphical results.

When comparing the simple copy and shared memory copy for the GTX280 the "Loop in kernel" column indicates closed values for the measured bandwidth.

e) Shared memory bank conflicts. Results are presented in Table 6 and Figure 7.

Table 6. Bandwidth in the shared memory bank conflicts test**.**

	Loop over kernel (GB/s)		**Loop in kernel (GB/s)**	
Test number	GTX280	GTX480	GTX280	GTX480
1	18.38	102.57	19.39	241.21
2	18.6	102.5	19.38	240.16
3	18.61	102.56	19.37	240.13
4	18.33	102.62	19.41	241.29
5	18.07	95.21	19.36	237.26
6	18.32	102.53	19.35	240.16
7	18.37	90.96	19.37	239.69
8	18.38	102.42	19.37	241.54
9	18.36	102.4	19.41	239.51
10	18.36	102.4	19.31	241.61

The results recorded on the Fermi architecture, the GTX480, are many times higher than the results recorded on the previous architecture, the GT200. The significant difference between the performances of these two architectures is explainable if we analyze the way shared memory bank conflicts occur and how the Fermi architecture manages the parallel threads.

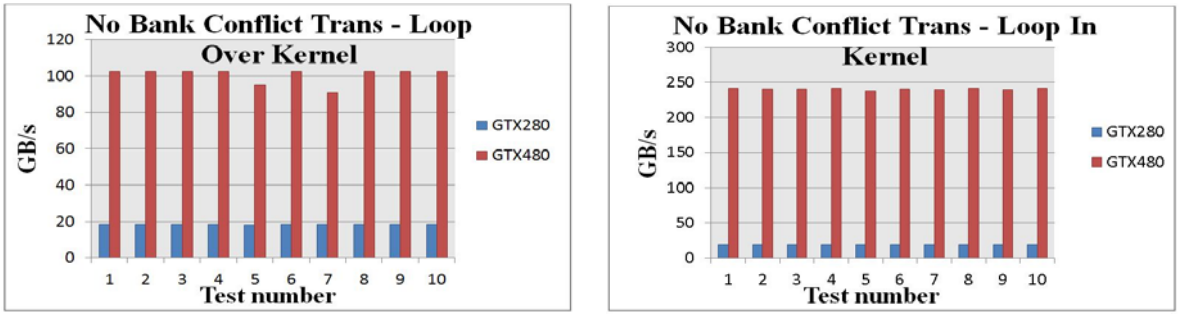

Fig. 7. Shared memory bank conflicts test – graphical results.

CUDA shared memory is divided into more memory banks (equally sized memory modules). Consecutive array accesses through consecutive threads are very fast as each memory bank holds a successive 32-bit value (e.g. a float variable). Multiple data requests from the same bank generate bank conflicts. The requests can originate from the same address or multiple addresses may map to the same bank. The hardware serializes the memory operations when the conflict occurs and this force all the threads to wait until all memory requests are fulfilled. Serialization is avoided if all threads read from the same-shared memory address, because a broadcast mechanism is automatically triggered. The broadcast mechanism is an excellent high-performance method to deliver data simultaneously to many threads.

When improving the performance of a CUDA application a developer must differentiate between local multiprocessor memory types' characteristics. The "registers" are the fastest memory on the multi-processor. Registers are accessible by the thread and exist only during its execution. Shared memory is as fast as a register if no bank conflicts occur or when the same memory address is accessed. Unlike registers, shared memory is accessible by any thread within the block where it has been created and exists as long as the block exists. Global memory exists during the application, is accessible from the device and is approximately 150x slower than register or shared memory. Local memory resides in global memory, is accessible only by the thread and exists only during the lifetime of a thread.

When profiling CUDA applications a programmer can determine if shared memory bank conflict occurs in any of the kernels by using the warp serialize flag. In the Fermi architecture the streaming multiprocessor schedules threads in groups of 32 parallel threads called warps. Two warp schedulers and two instruction dispatch units make it possible for two warps to be created and executed in the same time. The dual warp scheduler takes two warps and dispatches one instruction from each of them to a group consisting of 16 cores, 16 load/store units or 4 Special Function Units. The Fermi's scheduler does not have to check for dependencies inside the instruction stream as warps execute independently from one another. Most of the instructions can be dual issued (two floating instructions, two integer instructions or a mix of integer, floating point etc) [7].

f) Decomposing Transpose. In the next section, we break the transpose into components to determine the cause for the significant difference of performance between the coalesced and shared memory bank conflict free transpose and the shared memory copy. Results are presented in Table 7 and Figure 8 for the fine-grained transpose test and Table 8 and Figure 9 for the coarse-grained transpose test.

Table 7. Bandwidth in the fine-grained transpose test.

	Loop over kernel (GB/s)		**Loop in kernel (GB/s)**	
Test number	GTX280	GTX480	GTX280	GTX480
1	60.32	105.11	92.11	242
2	63.64	92.4	91.13	241.47
3	70.28	104.76	90.56	239.8
4	60.33	104.97	89.61	239.86
5	60.24	105.03	90.64	240.46
6	60.35	105.04	91.44	239.47
7	70.23	104.77	91.6	240.82
8	60.33	104.71	91	238.18
9	60.37	104.7	92.07	240.63
10	60.36	105.01	91.24	239.66

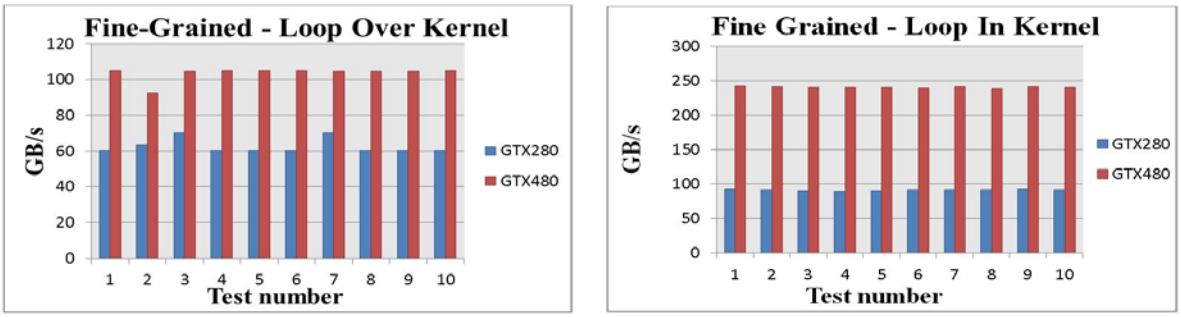

Fig. 8. Fine-grained transpose test – graphical results.

Table 8. Bandwidth in the coarse-grained transpose test.

	Loop over kernel (GB/s)		**Loop in kernel (GB/s)**	
Test number	GTX280	GTX480	GTX280	GTX480
1	18.41	103.97	19.42	239.6
2	19.26	103.99	19.33	239.26
3	18.63	104.12	19.36	237.46
4	18.36	103.97	19.37	241.39
5	18.37	104	19.41	240.98
6	18.41	104.03	19.37	241.39
7	18.41	96.07	19.36	238.98
8	18.42	90.87	19.35	231.84
9	18.4	100.92	19.38	230.45
10	18.41	100.5	19.34	240.19

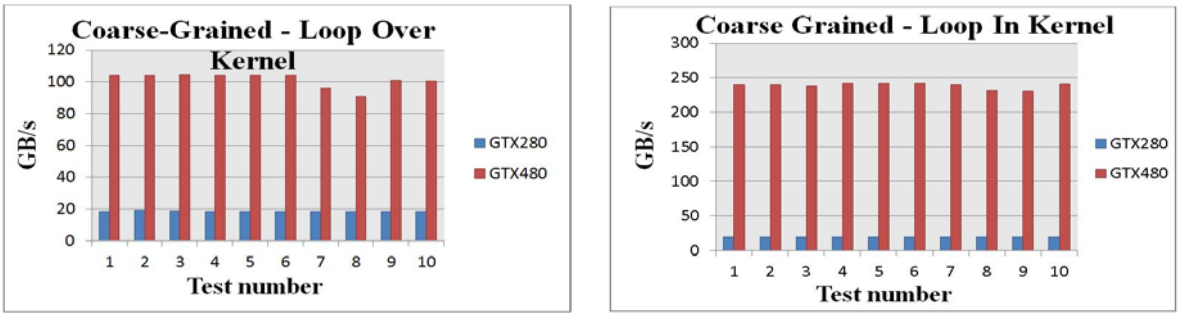

Fig. 9. Coarse-grained transpose test – graphical results.

The coarse-grained transpose kernel does not transpose the data within the tile, it only writes the tile to the transposed location while the fine-grained transpose kernel does. The coarse-grained transpose has almost the performance of the coalesced and bank conflict free transposes, while the fine-grained transpose has a performance similar to the shared memory copy. A performance bottleneck occurs when writing data in global memory to the transposed location. A decrease in performance can occur when accessing global memory through partition camping just like in the case of shared memory where performance degrades through bank conflicts. For a general understanding of the partition camping issue, see [14].

g) Diagonal block reordering. Results are presented in Table 9 and Figure 10.

Table 9. Bandwidth in the diagonal block reordering test.

	Loop over kernel (GB/s)		**Loop in kernel (GB/s)**	
Test number	GTX280	GTX480	GTX280	GTX480
1	26.37	78.7	101.56	256.11
2	28.1	78.7	101.21	256.21
3	26.88	78.6	101.64	255.54
4	26.26	78.66	101.51	255.89
5	26.32	78.73	101.42	256.98
6	25.83	78.78	101.34	255.33
7	26.35	71.69	101.45	256.03
8	26.38	71.76	101.31	254.85
9	26.38	78.72	101.44	256.83
10	26.38	71.51	101.44	254.45

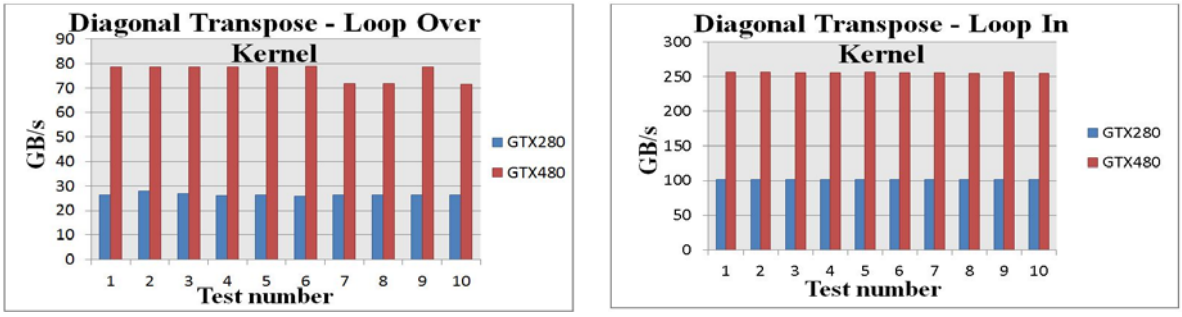

Fig. 10. Diagonal block reordering test – graphical results.

Diagonal reordering can solve the partition camping problem mentioned before. In the diagonal case, when reading from the input matrix and writing to the transposed one, pairs of tiles cycle through the partitions [13]. The read and write operations, when

looping in the kernel, represent only a few percent of the shared memory copy. The performance degrades slightly if looping takes place over the kernel. The diagonal transpose is more efficient than the other transpose types analyzed in this paper when it comes about bandwidth throughput. The performance increase happens despite the performance degradation mentioned above.

Table 10. Average bandwidth recorded in all the tests.

Test number	Test name	Loop over kernel (GB/s)		Loop in kernel (GB/s)	
		GTX280	GTX480	GTX280	GTX480
1	Simple copy	79.375	115.255	105.984	354.187
2	Naïve transpose	2.424	58.589	2.476	92.388
3	Coalesced transpose	17.811	90.779	19.363	167.627
4	Shared memory copy	37.634	94.2	87.89	193.107
5	Shared memory bank conflicts	18.378	100.617	19.372	240.256
6	Fine-grained transpose	62.645	103.65	91.14	240.235
7	Coarse-grained transpose	18.508	101.244	19.369	238.154
8	Diagonal transpose	26.525	76.585	101.432	255.822

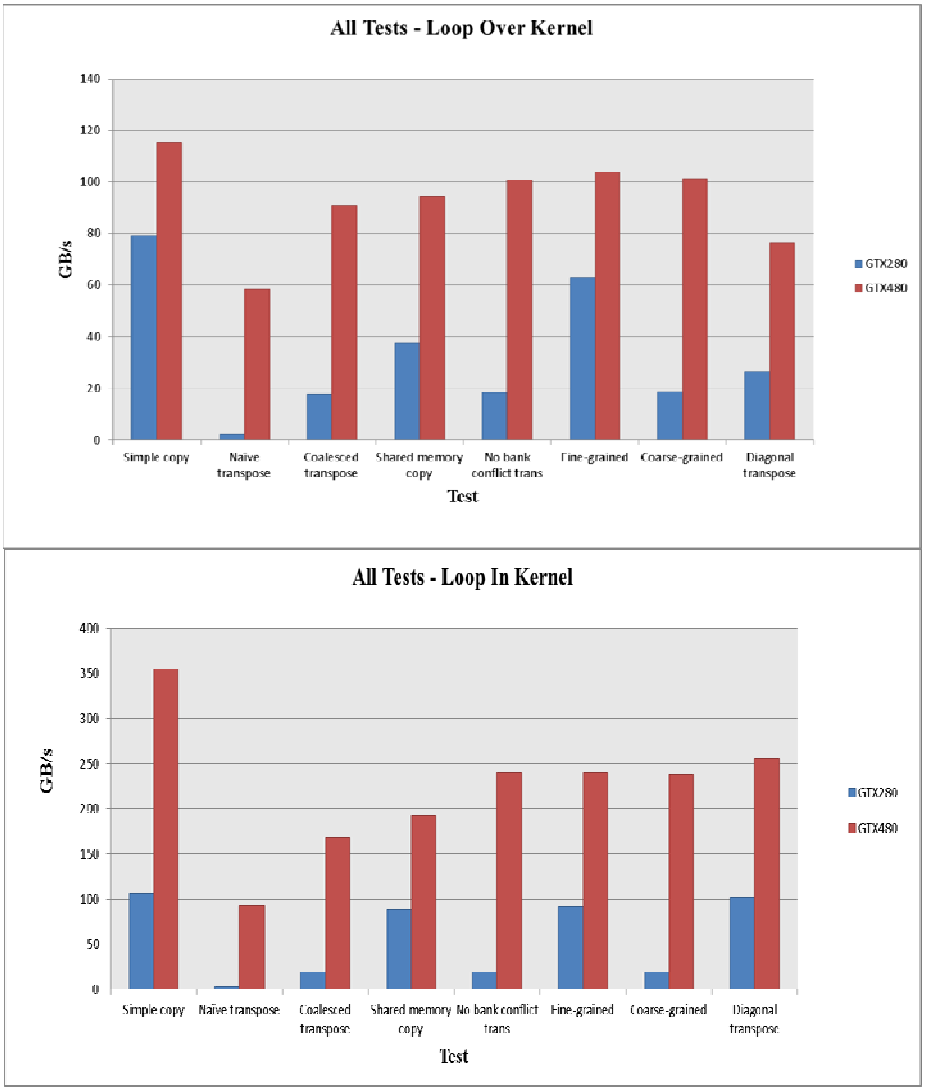

Fig. 11. Average bandwidth recorded in all the tests – graphical results.

The performance improvement is more notable for the GT200 architecture.
Finally, we present below a synthetic analysis, in order to give an overview of the experimental results obtained when testing the performance of NVIDIA's latest two architectures: GT200 and Fermi. Average bandwidth recorded in all the tests are presented in Table 10 and Figure 11, on both architectures, using the Loop over kernel and Loop in kernel methods.

6 Conclusions and Future Work

In this paper, we have analyzed several aspects regarding the improvement of performance for applications written in CUDA. We addressed an issue of paramount importance when programming an application in CUDA: GPU memory management through transpose kernels that are progressively optimized. We have also benchmarked and evaluated the performance for progressively optimizing a transposing matrix application in CUDA.
One particular interest was to research how well the optimization techniques, applied to software application written in CUDA, scale to the latest generation of general-purpose graphic processors units (GPGPU), like the Fermi architecture implemented in the GTX480 and the previous architecture implemented in GTX280.
Lately, there has been a lot of interest in the literature for this type of optimization analysis, but none of the works so far (to our best knowledge) tried to validate if the optimizations can apply to a GPU from the latest Fermi architecture and how well does the Fermi architecture scale to these software performance improving techniques. In this context, we performed the following tests on both architectures: simple copy, naïve transpose, coalesced transpose, shared memory copy, shared memory bank conflicts, fine-grained transpose, coarse-grained transpose and diagonal transpose.
Future work involves a more thorough optimization using a larger selection of CUDA applications and an exhaustive benchmarking process. We intend to analyze how the new CUDA architecture can optimize the data extraction process.

References

[1] M. M. Heck, High performance modelling of derivative prices using the peakstream platform, *PeakStream Financial Services Technical Note*, September 2006, pp. 73-81.
[2] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, D. Manocha, "Fast Computation of Database Operations using Graphics Processors", *Proceedings of ACM SIGMOD*, 2004, pp. 215-226.
[3] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, W. W. Hwu, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73-82.
[4] N. Satish, M. Harris, M. Garland - Designing Efficient Sorting Algorithms for Manycore GPUs, in *Proc. 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009, pp. 1-10.
[5] J. Archuleta, Y. Cao, W. Feng, T. Scogland - *Multi-Dimensional Characterization of Temporal Data Mining on Graphics Processors*, Technical Report TR-09-01, Computer Science, Virginia Tech, 2009.
[6] N. Nakasato, *A Fast GEMM Implementation on a Cypress GPU*, 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems, November 2010.
[7] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide, Version 3.0, *NVIDIA Whitepaper* 2010, pp. 11-161.
[8] M. J. Atallah, *Algorithms and Theory of Computation Handbook***,** CRC Press LLC, 1998.
[9] P. Bakkum, K. Skadron - Accelerating SQL Database Operations on a GPU

with CUDA, in *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, Vol. 425, 2010, pg. 94-103.

[10] J. Nickolls, GPU Parallel Computing Architecture and CUDA Programming Model, *IEEE Hot Chips 19 Symposium*, 2007, pp. 1-12.

[11] C. Martel, R. Subramonian, A. Park - Asynchronous PRAMs are (almost) as good as synchronous PRAMs, in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, , vol.2, 2006, pp. 590-599.

[12] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng and W. W. Hwu. Program optimization study on a 128-core GPU, in *The First Workshop on General Purpose Processing on Graphics Processing Units*, October 2007, pp. 30-39.

[13] G. Ruetsch, Paulius Micikevicius, Optimizing Matrix Transpose in CUDA, *NVIDIA Whitepaper*, 2010, pp. 3-23.

[14] D. B. Kirk, W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach (Paperback)*, Morgan Kaufmann Publishers, 2010.

Alexandru PIRJAN has graduated the Faculty of Computer Science for Business Management in 2005. He holds a MA Degree in Computer Science for Business from 2007. He joined the staff of the Romanian-American University as a teaching assistant in 2005 and a Lecturer Assistant in 2008. He is a PhD candidate since 2009 at the Doctoral School from the Bucharest Academy of Economic Studies. He is currently a member of the Department of Informatics, Statistics and Mathematics from the Romanian-American University. He is the author of more than 15 journal articles and a member in 4 national scientific research projects. His work focuses on database applications, artificial intelligence and quality of software applications.