Understanding the Impact of CUDA Tuning Techniques for Fermi

Yuri Torres, Arturo Gonzalez-Escribano, Diego R. Llanos
Departamento de Informática
Universidad de Valladolid, Spain
{*yuri.torres* | *arturo* | *diego*}@*infor.uva.es*

ABSTRACT

While the correctness of an NVIDIA CUDA program is easy to achieve, exploiting the GPU capabilities to obtain the best performance possible is a task for CUDA experienced programmers. Typical code tuning strategies, like choosing an appropriate size and shape for the threadblocks, programming a good coalescing, or maximize occupancy, are inter-dependent. Moreover, the choices are also dependent on the underlying architecture details, and the global-memory access pattern of the designed solution. For example, the size and shapes of threadblocks are usually chosen to facilitate encoding (e.g. square shapes), while maximizing the multiprocessors' occupancy. However, this simple choice does not usually provide the best performance results. In this paper we discuss important relations between the size and shapes of threadblocks, occupancy, global memory access patterns, and other Fermi architecture features, such as the configuration of the new transparent cache. We present an insight based approach to tuning techniques, providing lines to understand the complex relations, and to easily avoid bad tuning settings.

KEYWORDS: GPU, Fermi, performance, code tuning.

1. INTRODUCTION

Modern graphics processing units, such as new NVIDIA GPUs, can be used as general purpose architectures with a large amount of small processing cores. Nowadays, these devices are an important resource for intensive-data computing.

NVIDIA CUDA architecture and its associated parallel programming model [3, 5] was proposed in order to simplify the encoding of parallel, general-purpose applications, on heterogeneous systems with GPUs devices. Although the first steps with CUDA are easy, due to its reduced primitives set, it remains a difficult task to tune the code to efficiently exploit the underlying GPU architecture.

Fermi is the NVIDIA's latest generation of CUDA architecture [7]. Compared with earlier versions, the new architecture presents significant changes, such as an improved double precision performance; a transparent cache hierarchy; variable-size shared memory; and faster atomic operations.

Although a CUDA kernel is ensured to run correctly on any CUDA device, its global performance can vary greatly depending on how the code is tailored for a particular CUDA architecture. Optimizing a parallel GPU code is not a trivial task. There are several common optimization strategies in order to make an efficient use of hardware resources (see e.g. [3]). The details of these hardware-dependent strategies change with each new architecture release. Therefore, the programmer should take into account the underlying hardware design and threading model in order to exploit these strategies to achieve a good performance.

The CUDA programming model forces the programmer to divide the original problem into blocks of threads, whose size and shape should be tailored to the specific processing units present in the hardware. More specifically, the size and shape of the threadblock, together with the global memory access pattern of the threads, affect significantly to the SM occupancy and the access coalescence.

In this paper we provide new insights into the relationship between occupancy, threadblock size and shape, Fermi cache hierarchy configuration, and thread access pattern to global memory. We show that these factors interact in non-intuitive ways, and that their understanding is key to exploit GPUs for best performance.

The rest of the paper is organized as follows. Section discusses some related work. Section presents the Fermi architecture, highlighting the differences between its predecessors. Section shows the tradeoffs between different sizes and shapes for threadblocks and their relationship with coalescence and occupancy. Section describes the experimental environment, the benchmarks used and their access patterns. Section discusses the experimental results obtained, while Section concludes the paper.

2. RELATED WORK

Tuning strategies to improve performance, such as Coalescing, Prefetching, Unrolling and Occupancy maximization, are introduced in classical CUDA text books, such as [3]. A typical, intuitive idea is that the best option when choosing the threadblock size is to try to maximize the multiprocessors Occupancy to hide latencies when accessing global device memory. However, there is no discussion about how the size and shape of the threadblocks are related to other tuning techniques and how their may affect global performance.

In [8] the authors not only discuss the different tuning strategies, but also show how the hardware resources usage is critical for Occupancy and performance. However, the entire study has been focused on a pre-Fermi architecture.

Focusing on Fermi, a description of how the cache memory helps to take advantage of data locality at run time is presented in [11]. However, the cache makes performance very hard to predict, and dependent on algorithm parameters. The authors study a particular case, adjusting automatically the shared memory and the amount of data assigned to threads to optimize the execution. The paper does not depend on the different cache memory effects that appear when the global memory access pattern is not perfectly coalesced. General trends, and performance effects related with the threadblock size and shape are not discussed.

Regarding hardware metrics, several metrics related to hardware architecture and workload problems are presented in [2]. They help to predict more accurately the performance of CUDA kernel codes. In addition, they present the Ocelot's transaction infrastructure, where several low-level optimizations are automatized.

An optimizing compiler for GPGPU programs is presented in [12]. They propose compiler techniques to generate memory coalesced code. However, they only work with the same one naive matrix multiplication problem, and one memory access pattern. The only variable in the optimized code is the matrix size.

Other authors (see e.g. [4, 10]), have developed compilers that transform easy and high level input specifications to optimized CUDA code. The input code is annotated by the programmer, and translated to an optimized CUDA code using pre-Fermi state-of-the-art tuning techniques. In [9] the author introduces models in order to provide a methodology for predicting execution time of GPU applications, identifying and classifying the major factors that affect both, single and multiple-GPU environments. Nevertheless, none of these works relate their results to the critical choice of size and shape of threadblock. So far, there is a lack of literature about specific optimization and tuning problems for the Fermi architecture. The new two-level cache memory hierarchy, incorporated by this architecture, introduce new behaviors, not yet studied, in the execution of CUDA kernels.

Our work focuses on new Fermi architecture problems. We introduce key points and strategies to choose an appropriate size and shape for the threadblock depending on the global memory access pattern. Furthermore, we expose factors related to the threadblock choice and the configuration of the cache memory hierarchy, which may lead to thrashing and bandwidth bottleneck problems.

Table 1. Summary of CUDA Architecture Parameters

Parameter	pre-Fermi	Fermi
SPs (per-SM)	8	32
Registers (per-SM)	16 KB	32 KB
Max. number of threads (per-SM)	1024	1536
Max. number of threads (per-block)	512	1024
Warp size	32	32
Warp scheduler	Single	Dual
Shared memory banks (per-SM)	16	32
L1 cache (per-SM)	-	0/16/48 KB
L2 cache	-	764 KB
Global memory banks	8	6
Size of global memory transaction	32/64/128 B	32/128 B

3. FERMI ARCHITECTURE

Fermi is NVIDIA's latest generations of CUDA architecture [6,7]. It was launched early on 2010. The main changes introduced by Fermi architecture are: Transparent L1 and L2 cache hierarchy, increased shared memory size, ECC support, faster atomic operations, and improved double precision support. Anyway, to make a good parameter setting and code tuning, the programmer must take into account some features described in the following sections.

Table 1 shows a summary of several architecture parameters that have been modified in the new Fermi architecture. Before Fermi arrival, each SM only had 16 KB of shared memory to be addressed explicitly by the programmer, without any transparent cache memory. Fermi introduces a two-level transparent cache memory hierarchy. Each SM has 64 KB of on-chip memory, divided into shared memory and transparent L1 cache. The programmer may choose between two configurations: 48 KB of shared memory and 16 KB of L1 cache (the default option), or 16 KB of shared memory and 48 KB of L1 cache memory. Besides this, the L1 cache memory can be deactivated.

Previously, the memory transaction segment sizes were variable (32, 64, and 128 bytes). Depending on the amount of memory needed and the memory pattern access (scattering or contiguous data in memory), the segment size was automatically selected to avoid wasted bandwidth. In Fermi architecture, the memory transaction segment sizes are determined as follows: When L1 cache memory is enabled, the hardware always issues segment transactions of 128 bytes, the cache-line size; otherwise, 32 bytes segment transactions are issued. Finally, the Fermi architecture currently defines a non-configurable, 768 KB L2 cache.

A frequent problem in pre-Fermi architectures is the *partition camping* problem [1]. Partition camping is a fairy common problem, mostly because the number of memory controllers in these architectures are always power of two, and programmers tend to allocate arrays whose sizes per dimension are also a power of two. In Fermi architecture, the partition camping problem is alleviated due to the existence of a L2 cache, thus reducing the number of repetitive, conflicting access to DRAM. Moreover, current Fermi cards present a DRAM divided into five or six banks, and the alignment described above is less frequent.

4. CHOOSING THE THREADBLOCK SIZE AND SHAPE

A key decision in CUDA programming is the choice of the size and shape of the threadblock. The cardinalities of the three different dimensions of each block must be defined on each CUDA kernel launch. For a given problem encoding, the different threadblock sizes and shapes can significantly affect the overall code performance.

4.1. Threadblock Size and Occupancy

Fermi architecture supports at most $1\,024$ threads per threadblock, $1\,536$ threads per multiprocessor, and eight blocks simultaneously scheduled per multiprocessor. The most widespread and intuitive threadblock choice criterion aims to maximize Occupancy. For example, 8×8 and 16×16 threadblocks will produce different SM Occupancy. With 8×8 blocks, each block will have 64 threads, and we will need $1\,536/64 = 24$ blocks in a SM to reach its maximum number of threads. However, the limit of eight blocks per SM prevents to achieve maximum occupancy. On the other hand, 16×16 blocks lead to 256 threads per block. Since $1\,536/256 = 6$, maximum Occupancy is obtained with six blocks per SM. Recall that the number of threads per blocks should be multiple of 32, to avoid idle processors when executing a warp.

This implies: (1) The number of threads per block should be an integer divisor of the maximum number of threads per SM ($1\,536$ in Fermi); (2) The number of threads per block should be multiple of 32 to fill up warps (32 threads), but also big enough to generate no more than 8 blocks per SM (no less than $1\,536/8 = 192$ threads per block). Nevertheless, more threads imply more use of SM's resources (registers and shared memory addresses). If resources are exhausted, not all the expected threadblocks may be scheduled to the same SM, thus reducing Occupancy. Remind that Fermi introduces new SM resource sizes, and that shared memory and transparent L1 cache sizes may also be configured by the programmer, adding a degree of freedom.

Finally, it is remarkable that not only the size, but the shape of the threadblock in 2- or 3-dimensional problems, is an important tuning parameter. In general, there are several combinations of dimension sizes with the same number of threads. Maximum Occupancy may be achieved with different shapes, see discussion of Fig. 1 in section . The usual form of coding coalescing is to use the global thread indexes to directly access the corresponding matrix elements. In these cases, shapes with less than 32 columns compromise coalescing. Thus, the last dimension of the threadblock shape should be a multiple of 32; the warp size.

4.2. Coalescing and Cache Hierarchy

One of the most important and common optimization strategies in order to leverage the hardware resources is Coalescing. This technique aims to maximize global memory bandwidth usage, by reducing the number of bus transactions. Coalescing is obtained by forcing threads with adjacent global indexes in a block to request contiguous data from global memory. In this way, a single segment transfer can transport up to 32 requests of integer or float elements when L1 cache is active. Reducing bus transactions is critical to improve performance, since warps are blocked until their memory requests are solved.

When the code uses a perfectly coalescent global memory access pattern, the best threadblock size is intuitively one that maximizes Occupancy. However, the algorithm may not lead to a clear Coalescing pattern by itself, and/or rewriting the CUDA kernel to improve Coalescing may be

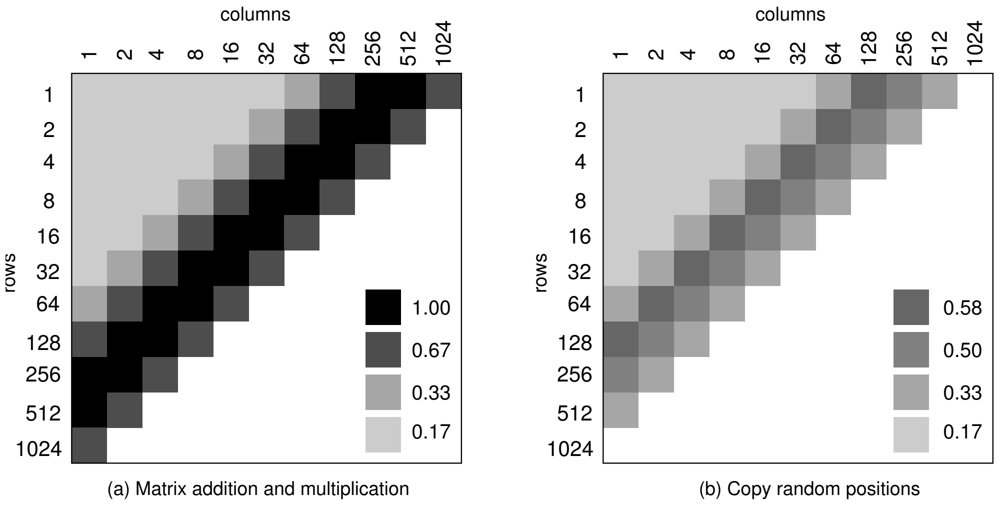

Figure 1. Occupancy of Our Benchmark Programs for Different Threadblock Shapes

too expensive. In these situations both L1 and L2 transparent caches may help to solve the situation. Nevertheless, if the pattern does not exploit some regularity and proximity, cache misses degrade performance.

Continuous cache-failure, evicting useful cache lines to store new ones, is known as cache-thrashing. When global memory access patterns are not coalesced, and the memory accesses do not reuse the same cache lines, the cache-failures number is quickly increased. Therefore both, L1 and L2 cache memories, suffer the cache-thrashing effect. Moreover, in Fermi architecture, cache-thrashing does not only leads to lose the beneficial performance effect related to cache-lines reutilization. When many different cache lines are requested simultaneously, the global memory bandwidth may be an important bottleneck. These combined effects can introduce significant delays in the program execution.

Sometimes, changing the configuration of the transparent L1 cache size, increasing its size to 48 KB, may help to alleviate the problem. Nevertheless, this increase in cache also means a reduction of the manually addresses shared-memory to 16 KB, that may lead to a reduction of Occupancy in some cases. Codes that heavily rely on global-memory accesses, with small use of shared-memory, and good reutilization of cached data, are candidates for this configuration change.

On the other hand, Fermi L1 cache may be disabled by the programmer. In that case, the size of the segment transactions is decreased form 128B to 32B. When only one data element is used per transaction segment (like in many non-coalesced and random global memory access patterns), smaller segments reduce the transfer time, and global-memory bandwidth bottleneck. New programmers are used to transparent caches that, even in thrashing situations, do not hinder the overall performance. In Fermi, they need to be aware that disabling L1 cache is a powerful and simple tuning technique for codes with sparse global-memory accesses. Not disabling it may produce an important slowdown.

5. DESING OF EXPERIMENTS

In this sections we present experiments devised to verify the performance effects discussed in the previous section. Although we focus on 2-dimensional problems and threadblock shapes, results may be extrapolated to 3-dimensional cases. We use as working examples the following problems: (1) Copy of random elements of a matrix to another matrix; (2) matrix addition; (3) two matrix multiplication algorithms.

The first problem simulates run-time or data-dependent memory access patterns (such as those associated to graph algorithms). We have developed a simple code in which each thread computes two random indexes, and copies the element in that position from matrix A to matrix B. There are as many threads as matrix elements. Due to the random choice of indexes, several positions are copied by different threads, while some others are not copied. Several similar codes have been developed and tested for one-dimensional vectors, finding the same results discussed on the next section. the same results discussed bellow.

Table 2. Execution Time (ms.) of the Program with Random Access Pattern

Rows \ Columns	1	2	4	8	16	32	64	128	256	512	1024
1	11058.98	6947.53	4179.44	2279.51	1223.05	653.31	393.02	348.15	354.68	411.52	*
2	7071.43	4130.58	2296.95	1230.77	655.25	392.73	348.24	353.67	411.41	*	-
4	4072.45	2292.78	1244.79	655.47	393.31	348.37	352.98	410.99	*	-	-
8	2239.78	1231.08	657.29	393.39	348.8	353.1	411.26	*	-	-	-
16	1207.69	652.23	394.05	349.08	353.68	411.27	*	-	-	-	-
32	649.39	391.78	349.82	354.46	411.86	*	-	-	-	-	-
64	390.88	348.32	354.86	412.22	*	-	-	-	-	-	-
128	346.73	352.73	412.63	*	-	-	-	-	-	-	-
256	352.01	410.91	*	-	-	-	-	-	-	-	-
512	409.31	*	-	-	-	-	-	-	-	-	-
1024	*	-	-	-	-	-	-	-	-	-	-

We have selected as second benchmark a matrix addition ($C = A + B$) algorithm. It presents a very simple global memory access pattern, with no reutilization of the same matrix element on different threads. Each thread is associated with a particular matrix position. This implies three global memory accesses per thread (two reads and one write).

Our third benchmark algorithm is matrix multiplication ($C = A \times B$). We have tested two implementations. The first one is very simple and straightforward for a non-experienced programmer. Each thread is associated with a single C position, and computes the dot product of a row of matrix A, and a column of matrix B. The global memory access pattern is complex, and inefficient. There is also reutilization of data between threads in the same block. Thus, it is interesting for our study of code tuning properties. We also consider a second more sophisticated implementation using an iterative block product, using tuning techniques such as coalesced copy of matrix blocks to local shared memory before computing.

The programs have been tested for different combinations of square- and rectangular-shaped threadblocks. The experiments have been conducted with integer and float matrices. In this work we present results for the integer matrices experiments. As the storage size of both types is the same, the effects on the memory hierarchy, are similar. Float matrices experiments simply present slightly higher execution times due to the extra computation cost associated to the floating point operations.

We use matrices with 6144 rows and columns. This size is small enough to allocate three matrices in the global memory of the GPU device. The dimensions of the matrices are multiples of the threadblock shapes considered in this study, and the number of global memory banks. Thus, matrix accesses on any threadblock are always aligned with the matrix storage, generating the same access pattern.

The experiments have been run on an Nvidia GeForce GTX 480 device. The CUDA toolkit version used is 3.0. The host machine is an Intel(R) Core(TM) i7 CPU 960 3.20GHz, 64 bits compatible, with a global memory of 6 GB DDR3. It runs an UBUNTU desktop 10.4 (64 bits) operative system.

We present performance measures, considering only the total execution time of the kernel functions in the GPU, presented in milliseconds. We skip initialization, and CPU–GPU communication times. The results show the mean of three executions. Deviations are fairly low, as discussed bellow for each application.

6. EXPERIMENTAL RESULTS

In this section we discuss the experimental results. We explain the performance effects, how they relate to Fermi hardware details, and the delicate choice of the size and shape of the threadblocks. We also introduce simple ideas to avoid the counter-intuitive situations that may appear when using the new Fermi architecture, and to find appropriate threadblock settings for good performance.

6.1. Occupancy

As discussed in section different multi-dimensional threadblock shapes have the same number of threads per block, and lead to the same occupancy of the SMs. Figure 1 shows the occupancy reported by the CUDA profiler for our benchmark programs. The pattern on the diagonals clearly indicates the shapes with the same number of threads. The matrix addition and multiplication programs do not use enough resources (registers, shared memory, etc.) to produce a reduction of Occupancy due to resource exhaustion. Indeed, the left plot in Fig. 1 shows the maximum Occupancy that may be achieved by any CUDA kernel, for the threadblock shapes considered in our study. The plot on the right, shows the occupancy of the program with the random-

Table 3. Matrix Addition: Execution Time, and Cache Lines Requested per SM

Columns / Rows	1	2	4	8	16	32	64	128	256	512	1 024
1	261.1	145.9	87.1	58.7	43.8	36.4	33.2	32	32.03	32.1	33.07
2	155.07	92.5	61.01	45.6	37.1	33.3	32.1	31.8	31.8	33.05	-
4	103.28	66.12	48.09	38.79	33.63	31.98	31.93	31.91	33.08	-	-
8	77.42	53.49	41.97	35.27	32.3	31.85	31.94	33.36	-	-	-
16	65.97	47.76	37.93	33.75	32.21	31.93	33.62	-	-	-	-
32	58.22	44.13	36.94	33.3	32.48	33.43	-	-	-	-	-
64	56.28	43.27	37.17	34.12	34.34	-	-	-	-	-	-
128	66.9	48.35	39.83	38.03	-	-	-	-	-	-	-
256	73.19	53.05	43.91	-	-	-	-	-	-	-	-
512	81.05	56.94	-	-	-	-	-	-	-	-	-
1 024	98.23	-	-	-	-	-	-	-	-	-	-

a) Execution Time (ms.)

Columns / Rows	1	2	4	8	16	32	64	128	256	512	1 024
1	124	124	124	124	124	124	48	96	144	144	96
2	48	48	48	48	48	48	96	144	144	96	-
4	96	96	96	96	96	96	144	144	96	-	-
8	192	192	192	192	192	144	144	96	-	-	-
16	288	288	288	288	288	144	96	-	-	-	-
32	288	288	288	288	288	96	-	-	-	-	-
64	192	192	192	192	192	-	-	-	-	-	-
128	384	384	384	384	-	-	-	-	-	-	-
256	768	768	768	-	-	-	-	-	-	-	-
512	1 536	1 536	-	-	-	-	-	-	-	-	-
1 024	3 072	-	-	-	-	-	-	-	-	-	-

b) Theoretical Number of Cache Lines Requested by a Block

access pattern. The occupancy reduction is produced by the amount of registers used by the algorithm implemented to generate the random indexes. Blocks with 1024 threads lead to occupancy 0, because the SM registers are exhausted even for a single block.

6.2. Random Pattern for Memory Accesses

Table 2 presents the execution times of the program with random memory accesses. The first observation is the bad performance for the threadblock shapes with light gray numbers. These blocks are too small, with not enough threads to fill up even a single warp. Thus, inner SM parallelism is not exploited. The performance is greatly reduced when approaching the upper-left corner of the table. These small block shapes should not be considered as a choice by the programmer.

The second observation is that the execution times for shapes in the same diagonals is practically the same. The deviation of the results on several executions is in the order of one millisecond. In programs where the memory access pattern is random, and with no possibility of exploiting coalescing or data reutilization, the performance is directly related to the Occupancy. We can conclude that, in absence of a defined memory access pattern, the best strategy is to maximize occupancy, focusing on the number of threads, and not in the shape of 2- or 3-dimensional blocks.

The random global accesses of this program imply continuous cache misses, one for each memory transaction. Deactivating the transparent L1 cache reduces the size of the transaction segment from 128 bytes to 32 bytes, alleviating the global memory bandwidth bottleneck. The deactivation of the L1 cache simply improves the performance results shown, without losing the occupancy-performance relation discussed.

6.3. Matrix Addition

The program exploits Coalescing, with a memory access pattern where threads in the same warp access contiguous global memory positions. Each matrix element is requested and used only once, and no reutilization may be exploited.

Table 3(a) shows the execution times obtained for this program. We can observe performance changes in the same diagonal (shapes with the same occupancy). Execution times clearly increase on the left of the table, when the threadblock shape have less than 32 columns. Remind that cache-

Table 4. Matrix Multiplication: Execution Time, and Cache Lines Requested per SM

Rows \ Columns	1	2	4	8	16	32	64	128	256	512	1024
1	867 208	432 212	215 609	107 867	53 982	27 015	13 766	7 780	6 237	6 502	7 935
2	433 990	216 378	107 948	53 970	27 016	13 592	7 154	5 856	5 874	7 212	-
4	217 653	108 544	54 162	27 076	13 609	7 269	6 155	6 337	7 352	-	-
8	109 559	54 653	27 277	13 669	7 328	6 163	6 431	7 990	-	-	-
16	74 759	38 522	18 448	12 316	7 313	6 348	8 638	-	-	-	-
32	112 494	56 336	28 248	14 267	6 550	8 355	-	-	-	-	-
64	126 553	63 547	30 550	14 730	8 123	-	-	-	-	-	-
128	166 618	73 826	31 344	11 856	-	-	-	-	-	-	-
256	184 812	80 330	28 562	-	-	-	-	-	-	-	-
512	190 406	69 624	-	-	-	-	-	-	-	-	-
1024	194 297	-	-	-	-	-	-	-	-	-	-

a) Execution Time (ms.)

Rows \ Columns	1	2	4	8	16	32	64	128	256	512	1024
1	-	-	-	-	-	379	429	512	520	503	488
2	-	-	-	-	329	202	252	264	252	244	-
4	-	-	-	365	195	212	341	298	123	-	-
8	-	-	366	179	349	355	365	64	-	-	-
16	-	3 538	186	154	936	351	37	-	-	-	-
32	4 294	4 294	3 635	1 831	797	30	-	-	-	-	-
64	4 294	4 294	3 829	1 811	45	-	-	-	-	-	-
128	4 294	4 294	3 843	195	-	-	-	-	-	-	-
256	4 294	4 294	3 615	-	-	-	-	-	-	-	-
512	4 294	4 294	-	-	-	-	-	-	-	-	-
1024	4 294	-	-	-	-	-	-	-	-	-	-

b) Number of Global Load Misses in L1 Cache (in Millions)

lines on Fermi have 128 bytes, enough size for 32 integer elements. Blocks with less than 32 columns are requesting a full cache-line for each row of threads, but they are not using all the data in the cache-line. It is straightforward to relate the performance results obtained to the theoretical number of cache lines requested by a block. See Tab. 3(b).

6.4. Matrix Multiplication

Table 4(a) shows the performance obtained for the naive matrix multiplication program. The deviation of the measurements is in the order of milliseconds. We have already discussed the bad performance on the upper-left corner of the table, where small blocks with non-full warps lead to poor inner parallelism in the SMs. And also the performance degrading effect associated with block shapes with less than 32 columns. However, we observe some relevant differences on the execution times found in the same diagonal, for blocks with more than 32 columns.

In this benchmark, a straight-forward implementation lead to access A by rows, while B is accessed by columns. Each thread computes a dot product of a row of A, and a column of B, storing the corresponding C element at the end. A elements are reused by all threads with the same row index, and B elements by all threads with the same column index. Coalescing is exploited on A and C accesses, although the impact of the small number of C accesses is very small. The performance of this program is ruled by the non-coalesced accesses to B elements. The L1 cache helps. Elements of B are reused by other threads in the same column of the block, during different iterations of the dot product. The key of the performance of this program is to avoid cache-thrashing, in order to keep B elements in cache until all threads in the same block-column have used them.

With the delicate deal of cache misses produced by A, B, and C accesses, it is difficult to model the cache behavior for the full computation. Table 4(b) shows the number of global load misses (in millions), as reported by the CUDA visual profiler. The relation of the load misses and the performance is clear on the right part of the table. We can see how the best performance results are obtained for threadblocks of 2×128 and 2×256 threads, where we find the minimum number of cache misses, and maximum occupancy.

It is remarkable the big amount of cache misses, and the bad performance obtained for shapes with only one row. In these blocks, the B elements are not reused, because there is only one thread on each block with the same column index. We also notice effects derived from partition camping on the global memory banks. Each global memory bank has lines of 256 bytes. Thus, the requests of two consecutive cache-lines of 128 bytes (32 consecutive integer elements), are camping for the same global-memory bank. L2 cache helps to minimize the impact of this effect, for cache-lines requested many times due to L1 cache misses. Some other stochastic effects derived from scheduling, and L1/L2 cache reutilization across different blocks, slightly modify the final execution times.

Experiments with other non-naive matrix multiplication implementations have been also conducted, finding always the same relations on performance, occupancy, and cache-misses. A proper choice of the threadblock shape may achieve a good exploitation of the L1 cache, producing the same performance effects as explicitly encoding coalesced copies of global memory blocks to local shared memory, before computing block products.

6.5. Discussion

The conclusion is that it is easy to identify the threadblock sizes and shapes which produce maximum occupancy. For shapes that produce maximum occupancy, performance is related to the number of global memory transactions, represented in Fermi by cache misses. Due to the 128B cache-line size, the maximum performance is obtained for shapes with at least 32 columns when accessing integer or float elements. Deriving a complete cost model is really difficult, as cache hierarchy behavior is complicate to predict. However, even for this narrowed search space, it is not so difficult to relate performance with conceptually simple measures of cache requests and elements reutilization.

It is interesting to notice that many matrix computations are implemented with square blocks for code simplicity. However, square blocks lead to threadblock shapes that are not good candidates for best performance. Moreover, focusing only on square-shaped blocks, the programmer cannot see the important relation of performance and cache-misses on other possible shapes with the same number of threads.

The different threadblock sizes and shapes have an important interaction with the cache, and the global memory access patterns. Thus, this is the first decision to take when facing a kernel tuning. Other tuning techniques should be applied after selecting an appropriate threadblock, considering the insight provided about the interactions with the GPU architecture model, and in particular, the memory hierarchy.

7. CONCLUSION

Writing efficient CUDA codes is demanding, because the programmer needs to know the underlying architecture specifications. Fermi architecture incorporates a two-level cache hierarchy, introducing new parameters on the optimization procedure. Although caches behavior is difficult to predict, they are helpful for the programmer who understands their constraints and advantages.

CUDA programmers should first use well-known tuning techniques to reduce the number of resources used by a threadblock, in order to obtain the maximum occupancy possible for any block size. After that, the threadblock shape is a key choice, as occupancy and shape present a predefined relationship.

Since cache-misses clearly determine the overall performance, the global memory access pattern is essential. Coalescing, a somewhat mystic CUDA tuning technique, is easily understood in Fermi when thinking in terms of cache-lines accesses. CUDA programs use the thread and block indexes to create global memory access patterns. Thus, it is straightforward to relate the threadblock shape to both, the cache-misses and the occupancy.

We have also found that a good shape choice may derive on an efficient use of the L1 cache, even for the non-coalesced access patterns, avoiding the need to program explicit copies of the global memory to the shared memory. Our results also indicate that codes using square-blocks, such as many typical matrix computations, are not exploiting efficiently the multicore devices.

Understanding all these relationships and interactions leads to a more systematic approach to code tuning, and greatly simplifies the optimization procedure.

ACKNOWLEDGMENTS

This research is partly supported by the Ministerio de Industria, Spain (CENIT MARTA, CENIT OASIS, CENIT OCEANLIDER), Ministerio de Ciencia y Tecnología (CAPAP-H3 network, TIN2010-12011-E), and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative.

REFERENCES

[1] Paulius Micikevicius Greg Ruetsch. Nvidia optimizing matrix transpose in CUDA, June 2010. Last visit: Dec 2, 2010.

[2] Andrew Kerr, Gregory Diamos, and Sudakhar Yalamanchili. Modeling GPU-CPU workloads and systems. In *Proc. GPGPU'10*, Pittsburg, PA, USA, Apr. 2010.

[3] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, February 2010.

[4] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Manikandan Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proc. GPGPU'10*, pages 51–61, Pittsburgh, PA, USA, March 2010.

[5] NVIDIA. *NVIDIA CUDA ProgrammingGuide 3.0 Fermi*, 2010.

[6] NVIDIA. Tuning CUDA Applications for Fermi, July 2010.

[7] NVIDIA. Fermi Architecture Home Page, Last visit: August 2, 2010. `http://www.nvidia.com/object/fermi_architecture.html`.

[8] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. PPoPP '08*, pages 73–82, Salt Lake City, UT, USA, 2008.

[9] Dana Schaa. Modeling execution and predicting performance in multi-GPU environments. In *Electrical and Computer Engineering Master's Theses*, Boston, Mass, 2009. Department of Electrical and Computer Engineering, Northeastern University.

[10] Michael Wolfe. Implementing the PGI accelerator model. In *Proc. GPGPU'10*, pages 43–50, Pittsburg, PA, USA, 2010.

[11] Changyou Zhang Xiang Cui, Yifeng Chen and Hong Mei. Auto-tuning dense matrix multiplication for GPGPU with cache. In *Proc. ICPADS'2010*, pages 237–242, Shanghai, China, December 2010.

[12] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. An optimizing compiler for GPGPU programs with input-data sharing. In *Proc. PPoPP '10*, pages 343–344, Bangalore, India, 2010.