Using Fermi architecture knowledge to speed up CUDA and OpenCL programs

Yuri Torres
Dpto. Informática,
Univ. Valladolid, Spain
Email: yuri.torres@infor.uva.es

Arturo Gonzalez-Escribano
Dpto. Informática,
Univ. Valladolid, Spain
Email: arturo@infor.uva.es

Diego R. Llanos
Dpto. Informática,
Univ. Valladolid, Spain
Email: diego@infor.uva.es

Abstract—**The NVIDIA graphics processing units (GPUs) are playing an important role as general purpose programming devices. The implementation of parallel codes to exploit the GPU hardware architecture is a task for experienced programmers. The threadblock size and shape choice is one of the most important user decisions when a parallel problem is coded. The threadblock configuration has a significant impact on the global performance of the program. While in CUDA parallel programming model it is always necessary to specify the threadblock size and shape, the OpenCL standard also offers an automatic mechanism to take this delicate decision.**

In this paper we present a study of these criteria for Fermi architecture, introducing a general approach for threadblock choice, and showing that there is considerable room for improvement in OpenCL automatic strategy.

Index Terms: GPGPU, automatic code tuning, Fermi, CUDA, OpenCL

I. Introduction

Many-core Graphics Processing Units (GPUs) have become an important computing platform in many scientific fields, such as intensive-data or biomedical computing. Their main advantages are their low operational cost, their easy and friendly programming environments, and a high performance peak. GPGPU (General Purpose GPU) programming has been simplified by the introduction of high level data parallel languages, such as CUDA [1], or OpenCL [2]. However, maximizing the performance of any parallel problem implementation requires in depth knowledge about GPU underlying architecture, as well as an expertise on the use of code tuning techniques. Therefore, fully exploiting the GPU capabilities is a tedious task only suited for experienced programmers.

In CUDA, it is always necessary to define a grid of threadBlocks that are scheduled to the different *SMs (Stream multiprocessors)*. The choice of the threadblock size and shape is a very important decision to develop a highly-tuned implementation of a parallel problem on GPUs. Currently, many programmers choose the threadblock size and shape by trial and error. Although there are well-known basic techniques to eliminate bad candidates, to explore the remaining alternatives is time-consuming. Moreover, little modifications done to the code may force to restart the search, and there are no solid strategies to guide the programmer.

CUDA is not the only alternative for the NVIDIA GPU programming. OpenCL standard offers a common programming API for GPUs of different vendors, hiding many hardware resource details by the use of abstractions. Therefore, it is simpler to use, but more difficult to tune for high-performance. Regarding the threadblock selection, OpenCL offers a mechanism to automatically select the threadblock configuration.

In this paper we present a practical study of the Fermi architecture, focused on how the threadblock parameters and the use of hardware resources affect performance. Our experimental results show that it is possible to use hardware knowledge to better squeeze the GPU potential, with no need of time-consuming tests. Many obvious and typical threadblock configuration strategies, such as choosing square shapes, or blocks with the maximum number of threads, are not always appropriate. Moreover, we also show that OpenCL automatic strategy to select the threadblock is too simplistic, implicating performance degradations up to 65% in our experiments. We show how it can be easily improved.

II. NVIDIA Fermi architecture

Fermi is NVIDIA's latest generation of CUDA architecture [3], released in early 2010. The main characteristics introduced by this new architecture include double precision performance, error correction code support, transparent L1/L2 cache hierarchy, configurable L1 and shared memory, faster Context Switching, and faster atomic operations. This section discusses the details of the Fermi architecture that are relevant for our study of the relation between performance and program characteristics.

(A) Transparent L1/L2 cache memory. Fermi introduces an L1/L2 transparent cache memory hierarchy (see Fig. 1). The programmer can choose between two configurations: 48 KB of shared memory and 16 KB of L1 cache (default option), or 16 KB of shared memory and 48 KB of L1 cache. Besides this, the L1 cache memory can be deactivated by the user at compilation time.

(B) Threadblocks, Warps and SMs. The number of registers per multiprocessor in Fermi is 32KB, and the number of SP (Streaming Processor) per SM (Streaming Multiprocessor) is 32. The maximum number of threads per threadblock is doubled, from 512 to 1 024, while the maximum number of threads per SM is 1 536. Note that these changes force the programmer to re-calculate block parameters used in implementations for previous architectures, in order to maximize the use of SM resources. When the SMs require more workload, the threadblocks are dispatched to the SMs in row-major order [4].

In programs where the computational workload of all threads is similar (most data parallel programs), the blocks are scheduled to the SMs at regular intervals. This implies a

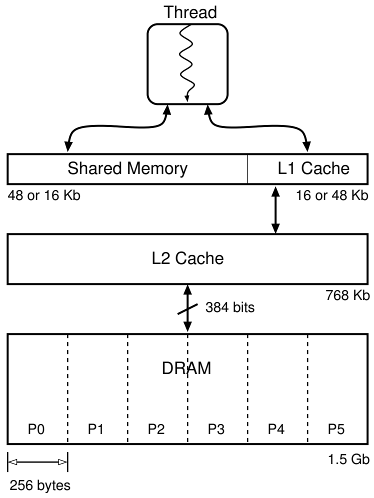

Fig. 1. FERMI memory hierarchy (NVidia GTX-480).

high reutilization of data on the L2 cache, that works as a fast transparent shared memory for all SMs. As the computation advances at the same pace on each SM, the situation appears again and again. Moreover, the application can be optimized through a proper selection of threadblock size and shape.

Regarding the dual warp scheduler present in Fermi, each scheduler has its own instruction dispatch unit (IDU). The SM executes simultaneously two halves of different warps. It is important to notice that this may influence the span of data that is requested to the cache hierarchy at the same time. Although the SMs have only 16 load/store units, each half warp issues its memory requests on a different flank, allowing them to issue up to 32 memory requests on the same cycle, 16 from each half-warp.

(C) Shared memory conflicts and global memory access. Fermi has 32 shared memory banks. Currently, the global memory is also distributed on 5 or 6 banks in Fermi. The memory addresses are scattered across the banks. A frequent problem in pre-Fermi architectures is the *partition camping* problem [4]. This problem arises when concurrent threads request at the same time memory locations belonging to different transaction segment in the same data bank. In Fermi, the problem is alleviated by the L2 cache.

III. Selecting the threadblock size and shape

In this section we present a discussion on how the Fermi architecture details affect the programmer decisions about threadblock size and shape. The implication of other configurable parameters, such as deactivating the L1 cache or modifying its size, are also discussed.

A. Threadblock size

1) Maximize occupancy: One SM can execute two half-warps at the same time. The warps of several thread blocks can be queued on the same SM. When the threads of a warp issue a global memory request, these threads are blocked until the data arrives from memory. During this high latency time, other warps in the queue can be scheduled and executed. Thus, it is important to have enough warps queued in the SM to hide the global memory latencies by overlapping them with computation, or with other memory accesses.

The first consideration to maximize Occupancy is to select a proper block size. In Fermi, the number of threads per block should be an integer divisor of the maximum number of threads per SM, and higher than or equal than 192, to allow to fill up the maximum number of threads per SM with no more than 8 blocks ($1536/8 = 192$). These values are 192, 256, 384, 512, and 768.

Finally, in computations that use a grid with a small number of total threads, it may be beneficial to use very small blocks to distribute the computational load across the available SMs in the GPU. For example, to execute 512 threads, using only one block for all of them forces to execute all the load in one SM.

2) Coalescing and high ratio of global memory accesses: Memory Coalescing is a technique used in kernel coding to force consecutive threads in the same warp to concurrently request consecutive logical addresses from global memory. This allows to minimize the number of transaction segments requested to the global memory. Typically, it is done by properly associating data elements to thread indexes when traversing dense data-structures, such as multidimensional arrays. Classical examples include many dense matrix and linear algebra operations. Coalescing is particularly important on codes with a high ratio of global memory accesses. For this study, we consider high a ratio of one or more memory accesses per arithmetic operation, omitting the typical computation of global thread indexes at the start of the kernel.

A common technique to ease the programming of coalescing is to ensure that the dense data-structures are aligned with the global memory banks and transfer segments. Thus, arrays with the last dimension multiple of 32 simplify the programming of Coalescing. Moreover, to avoid partition camping problems, it is better to use data structures that are aligned to the number of global memory controllers or banks on the target architecture.

For kernels with a high-ratio of coalesced global memory accesses, and considering the block sizes that maximizes Occupancy, we may expect that the best performance would be obtained maximizing the number of blocks in any SM, which means minimizing the threadblock size (192 to keep maximum Occupancy). The rationale behind this hypothesis is that programs with continuous accesses to global memory need at all times 48 warps in an SM to properly hide latencies. This effect is mainly noticeable on kernels with a short number of total operations per thread, because the blocks finish, and need to be replaced, faster.

On the other hand, when the code in the kernel is not requesting data fast enough to need 48 warps to hide latencies, we do not need maximum occupancy to obtain the best performance. Examples are codes with high computational workload between global memory requests, that are not issued at the same time. In this situation, the computation of one warp may hide the latencies of the memory requests of other warps, and the best performance results could also be obtained with block sizes that does not maximize Occupancy.

3) Non-coalesced accesses: Codes with non-coalesced global memory accesses request many different memory trans-

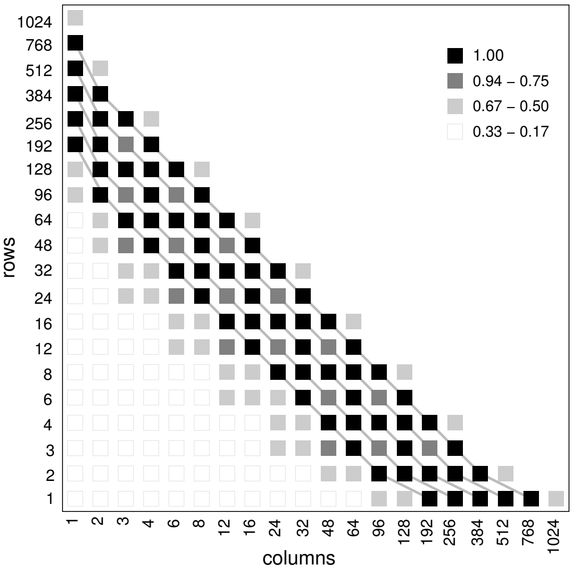

Fig. 2. Maximum Occupancy for different threadblock shapes.

fer segments from the same warp at the same time. The memory requests are serialized and it is much more difficult to have a code with enough computational workload, and computation vs. memory accesses overlapping, to hide such latencies. Moreover, in these cases, reducing the number of requests has another beneficial impact on the partition camping problems that arise in the global memory banks. Thus, reducing the blocks up to 32 threads, may keep all possible parallelism (eight blocks with their maximum of 32 threads per block), minimum global memory bandwidth requested, and minimum number of bank conflicts.

The global memory bandwidth bottleneck, and the partition camping problems, may become so expensive that it may even compensate to reduce the number of active SPs per warp using blocks with less than 32 threads. However, reducing it too much may lead to waste parallelism capabilities and to lose performance. Without using more information about the architecture details it is difficult to predict the optimum block size.

In general, codes with scatter accesses benefit from deactivating the L1, since the transaction segment size is reduced, thus alleviating bottlenecks.

B. *Shape in several dimensions*

Figure 2 shows the Occupancy obtained for different combinations of threadblock shape dimensions, when the code does not exhaust SM resources (registers and shared memory). The block shapes with the same block size, that also maximize Occupancy, are linked by a gray line. Besides the effect of Occupancy, the chosen shape has also a significant impact on coalescing, partition camping, and memory bottlenecks.

In programs with good Coalescing, the best performance results will be obtained with shapes with no less than 32 columns. One warp should request 32 contiguous elements to reduce the total memory bandwith. With perfect Coalescing, the first half-warp request 16 elements, obtaining a full cache line with 32 elements: The 16 elements requested by the first half-warp, and 16 more elements which are needed by the second half-warp. Thus, the second half-warp finds all needed elements in the cache, skipping the global memory latency.

In Fermi, we may find 320 or 380 consecutive elements spread across the full span of the five or six memory banks. Thus, perfectly coalesced codes using a threadblock shape with that amount of columns, minimize the global memory bank conflicts. Reducing the number of columns to increment the number of rows immediately derives in an increment of these conflicts, and in performance degradation.

C. *Tuning techniques and threadblock size and shape*

Some of the common code tuning strategies [1] heavily interact with, or are dependent on, the chosen block size and shape. Although it may seem intuitive to first choose the block shape, and then apply the other tuning techniques, some of them may need specific block sizes or shapes to exploit their potential. For example, the CUBLAS optimized matrix operations [5] uses specific block shapes to improve performance while keeping correctness. It is an open question when, and where, it is possible to isolate the threadblock configuration from the application of other tuning techniques.

D. *Threadblock size and shape in OpenCL*

OpenCL [2] (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs, and other processors. It provides developers with a portable and efficient software to manipulate these heterogeneous processing platforms.

OpenCL provides a mechanism to select manually the threadblock shape. The behaviour of programs when using the CUDA driver on OpenCL are similar to direct CUDA programming. The effects discussed on previous sections are dependent on the hardware architecture, and thus, they will affect performance in the same way.

OpenCL includes a convenient mechanism to automatically select the threadblock shape, letting the programmer to skip this decision. However, our study on the use of this OpenCL automatic mechanism shows that it is focused on using the maximum number of threads per block (1024 on Fermi). Based on the previous discussion, the reader may expect that other smaller block sizes will lead to better performance. In Sec. VII we present specific experimental results.

IV. Design of experiments

In this section we introduce the design of experiments to verify the previously presented hypotheses and deductions derived from the architecture observation. We will run different benchmarks on a Fermi architecture platform. We have selected real applications and kernels, as well as synthetic codes, to isolate and test different application features. Kernels used are intentionally simple, to minimize the interactions among different hardware effects, such as coalesced vs. scattered accesses, different ratios of workload vs. number of memory accesses, cache reutilization, etc.

The algorithms and coding ideas of some of the benchmarks are obtained from examples included in the CUDA and OpenCL SDK, or well-known linear algebra libraries, such as

CUBLAS [5] and MAGMA [6]. The original codes cannot be directly used in our study because their optimizations and tuning strategies are dependent on specific threadblock sizes and shapes. For example, the threadblock sizes for the basic matrix multiplication on CUBLAS and MAGMA libraries is fixed to 512 and 256 respectively. We have adapted and simplified the codes to make them completely independent of the threadblock shape. We avoid the use of sophisticated tuning techniques to isolate the different effects of the block shape on each benchmark.

Although we focus on 1- and 2-dimensional problems, results can be extrapolated to 3-dimensional cases. The programs have been tested for different combinations of square- and rectangular-shaped threadblocks. We use shapes with a number of rows or columns which are powers of 2, or powers of 2 multiplied by 3. This include all the combinations that maximize Occupancy.

The experiments have been conducted with 1- and 2-dimensional arrays, using integer and float elements. In this work we present results for the integer arrays experiments. As the storage size of both types is the same, the effects on the memory hierarchy are similar. Float arrays experiments simply present slightly higher execution times due to the extra computation cost associated to the floating point operations. The 1-dimensional benchmark repeats 16 times a reduction of an input vector with 1023 k-elements. Therefore, we can use blocks with eight or more threads, generating grids with no more than the maximum number of blocks allowed in CUDA for any dimension (65535). For the 2-dimensional benchmarks we use input matrices of 6144 rows and columns. The size chosen has several advantages. First, this size is small enough to allocate up to three matrices in the global memory of the GPU device. Second, this size also ensures that not all blocks can be executed at the same time, and therefore most of them are queued. In this way, this size mimics the behavior of bigger matrices, as long as data is aligned in the same way. Moreover, the dimensions of the matrices are multiples of: (1) all the block-shape dimensions tested; and (2) the number of global memory banks in our test platform (described below). 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 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 10.10 (64 bits) operative system. The programs have been developed using CUDA and OpenCL. The CUDA driver used was the version included in the 4.0 toolkit. All benchmarks were also executed with both OpenCL 1.0 and 1.1, with the same performance for both versions. We measure performance considering only the total execution times of the kernel functions in the GPU. We skip initialization, and CPU–GPU communication times. Our results are the mean of the measures of three or more executions.

V. Benchmarks

In this section we describe the criteria to select and/or design the different benchmarks.

A. Coalesced accesses

1) Vector reduction: We use a reimplementation of one of the CUDA SDK examples modified to simplify the change of threadblock shape. The kernel is launched in several synchronized stages. On each stage each thread reduces two contiguous data elements, storing the result in a properly compacted ancillary data structure, used as input for the next stage. Thus, each thread issues two contiguous read requests to global memory, in two consecutive load operations. After the single arithmetic operation, each full warp writes the results in a single global memory transaction segment of 128 KB. The number of blocks is divided by two on each stage. The main code execute the kernel 16 times to generate enough load to obtain stable results.

2) Adaptive-block vector reduction: In Sec. III-A1 we remarked that executing a small amount of threads in the whole GPU can benefit more from using many small blocks than a single bigger one, in order to spread the workload and exploit more parallelism on the SMs. Taking into account these observations, we have introduced an improvement on the vector reduction code. The first stages are computed with a fixed block size. When we need less than 15 blocks (the number of SMs in our GPU testing device) to process the data, we divided the threadblock size to increase the number of blocks and the potential parallelism. This improvement is done on stages with low workload. We expect a slight performance improvement, noticeable for small input data sets.

3) Matrix addition: We also test a matrix addition ($C = A + B$) algorithm. Each thread is associated with a particular matrix position. This implies three global memory accesses per thread (two reads and one write). It presents a full coalesced memory access pattern, with no reutilization of the same matrix elements by other threads.

4) Overloaded kernel: We have generated a synthetic program based on the matrix addition code. It simply adds an arbitrary number of dummy arithmetic operations (10 000) to the original single addition after loading the two elements. This code keeps the matrix addition access pattern, but introduce an overload between the load and the store global memory accesses.

5) Overlapped memory accesses kernel: We propose another modification to the matrix addition code to force different warps in the same block to issue load global memory operations at different times. There is a maximum of 48 warps in a SM. We use the warp number to select the exact amount of dummy arithmetic operations carried out before the loads (warpId × 1000), and between the loads and stores ((48 - warpId) × 1000). We have tested that 1000 dummy arithmetic operations take more time than the global memory latency. Thus, we completely overlap the communication latencies of the load operations with computation across the different warps.

6) Matrix multiplication: We have coded two versions of matrix multiplication ($C = A \times B$) which are independent of the block size and shape. A *naïve matrix multiplication* and an *iterative block-products matrix multiplication*. The first one is very simple and straightforward for a non-experienced programmer. There are reutilization of data between threads in the same block at different stages of the dot product. This benchmark is interesting due to the relationship of the reutilization of caches with the coalesced memory access pattern. We also consider a second, more sophisticated implementation using iterative block products.

B. Non-coalesced accesses

1) Regularly-scattered accesses: This synthetic benchmark is designed to create a simple scattered access pattern in which each thread requests a different memory transfer segment. Each thread accesses to a single matrix element. The position is computed multiplying the column index of the thread by the maximum size of a transfer segment (32 elements). The obtained value is modified with a simple arithmetic operation and copied into the same element to reuse the same transfer segment. Only one single arithmetic operation is issued on each thread.

2) Random accesses: This benchmark is a modified version of the previous one. Each thread copies one value from a random position of one matrix, in the same random position of another matrix. The workload associated with computing the random indexes is higher than in the previous benchmark, and comprises around 20 arithmetic operations. Two memory transfer segments are requested per thread, one per each matrix access. The random indexes force most threads to request elements on different transfer segments, with little or no reuse of transaction segments. This benchmark simulates random scattered accesses that typically appears in graph traversing algorithms, or codes for other sparse data structures.

VI. Experimental results

In this section we present the results obtained by our experiments for both, CUDA and OpenCL implementations, and discuss their relation with the Fermi architecture details commented in previous sections. We first describe the results obtained with CUDA, and then, we discuss the differences with OpenCL.

A. Coalesced global memory accesses

1) Small kernels with no data reutilization: Table I(A) shows the execution times of the matrix addition benchmark for different shapes. The results for the block sizes that maximizes Occupancy are presented in boldface. The table skips the first columns where the warps are with at least three quarters of their threads idle, and the execution times grow quickly.

The table confirms the expected results, as previously discussed in section III: (1) The best performance is obtained with block sizes that maximize Occupancy; (2) considering the maximum Occupancy block sizes, diagonals of smaller blocks present better performance, with the optimum in blocks of 192 threads; (3) blocks with 48 columns, or less than 32 columns, perform really bad due to the lose of coalescing; and (4) blocks with more columns and less rows, imply less conflicts when accessing the global memory banks.

Table II(A) shows the execution times of the vector reduction benchmark. For the vector sizes chosen, we cannot choose blocks sizes below 32, due to the maximum number of blocks per dimension in CUDA. Having similar properties than the matrix addition, the effects observed are similar, and the best performance is found in 192 threads per block. Our results indicate that the adaptive-block vector reduction improves the performance only for small input data sets. For example, 3% to 4% for 1023 k-elements. This techniques have more impact of computations with a higher workload per thread.

As expected, for coalesced codes without data reutilization, deactivating the L1 cache does not affect performance. There are four more transaction segment requests, but the segments are four times smaller. The final global memory bandwith does not significantly change.

2) Higher loaded kernels: Table I(B) shows the execution times of the overloaded kernel. As expected for high loaded coalesced codes, with no data reutilization across threads, and low number of memory accesses comparing with arithmetic operations, the results indicate that any shape that maximizes occupancy produces a similar performance. Blocks with very few columns that prevents coalescing have a very small effect on total performance because the extra cost of global memory accesses are hidden by the big computation costs. The effect of the faster replacement of ending warps when smaller blocks finish, is negligible comparing with the overall computation. In this code, all warps begin executing the load accesses at almost the same time. Thus, latencies are not really well hidden across the warps of a block.

3) Hiding global memory latencies: Table I(C) shows the execution times of the overlapped memory accesses kernel. Recall that this benchmark ensures that the latencies of global memory loads on any warp are completely overlapped with the computation of other warps in the block. We observe that the best performance is obtained for blocks with less than 192 threads. As expected, in this type of code maximum Occupancy is not needed to hide latencies, because they are hidden by the computation overlap.

4) Intensive data reutilization: Table II(B) shows the execution times of the naïve matrix multiplication code. The tables for this benchmark skip all the columns where the warps have idle threads and the execution time explodes. Due to the high reutilization of data, bigger block sizes have more opportunities to reuse the cache lines. Thus, the best performance results are found for the biggest block size that maximizes Occupancy (768). For a given block size, we also observe a trend that lead to obtain better performance results when using a shape with more columns and less rows. As commented in Sect. III-B, blocks with more columns, up to 384 (due to the 6 global memory banks on our device) reduce the number of bank conflicts. It also impacts on the reutilization and trashing of the L1 cache due to the algorithm properties. Table III shows the number of L1 cache misses as reported by the visual profiler included in the CUDA toolkit. For maximum Occupancy we observe a clear correlation of L1 and L2 cache misses and performance. Blocks with one row have no opportunity to exploit any reutilization of data on the second matrix. Thus, they produce many more cache misses and its performance is degraded. The best performance is found for the shape 2×384. Increasing the L1 cache size to 48 KB, reduces cache misses and produces an improvement

(A) Matrix Addition: Execution Times

	Columns														
Rows	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024
128	9,01														
96	6,35														
64	5,49	5,48	5,78												
48	5,24	5,60	4,00												
32	4,86	4,80	3,36	3,88	4,23										
24	4,70	4,68	3,28	4,17	3,07										
16	5,05	4,43	3,23	3,45	3,04	3,38	4,14								
12	5,42	4,58	3,28	3,55	3,02	3,43	3,05								
8	6,05	4,97	3,42	3,41	2,96	3,05	2,95	3,17	4,45						
6	7,18	5,52	3,87	3,53	2,95	3,06	2,94	3,46	3,19						
4	9,70	6,73	5,06	3,92	3,11	3,10	2,90	3,00	3,05	3,19	4,44				
3	11,94	8,48	6,16	4,64	3,53	3,12	2,89	2,96	2,95	3,35	3,18				
2	16,43	11,43	8,55	6,12	4,54	3,68	3,08	2,93	2,93	2,95	2,96	2,99	3,95		
1	29,97	20,24	15,16	10,51	7,74	5,73	4,40	3,49	3,08	2,89	2,89	2,91	2,94	3,01	3,94

(B) Overloaded Kernel: Execution Times

	Columns														
Rows	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024
128	1353														
96	1351														
64	1349	1350	1351												
48	1349	1351	1350												
32	1349	1349	1349	1349	1350										
24	1349	1349	1349	1349	1348										
16	1349	1349	1349	1349	1349	1349	350								
12	1349	1499	1349	1349	1348	1349	1348								
8	1352	1349	1349	1349	1349	1349	1349	1348	1350						
6	1802	1799	1349	1499	1349	1349	1348	1349	1348						
4	2322	1803	1352	1349	1349	1349	1349	1349	1349	1348	1350				
3	3096	2403	1802	1799	1349	1499	1349	1349	1349	1349	1348				
2	4648	3099	2325	1803	1352	1349	1349	1349	1349	1349	1349	1348	1350		
1	9286	6195	4645	3099	2325	1803	1352	1349	1349	1349	1349	1348	1349	1348	1350

(C) Overlapped Memory Access Kernel: Execution Times

	Columns														
Rows	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024
128	901														
96	873														
64	851	872	895												
48	841	851	873												
32	831	840	850	873	896										
24	827	835	841	849	873										
16	822	828	831	842	852	873	896								
12	819	915	828	834	842	847	873								
8	828	819	824	828	832	842	851	873	896						
6	1103	1092	819	914	828	834	842	848	873						
4	1438	1104	828	819	823	828	832	842	851	873	896				
3	1916	1469	1103	1092	818	914	828	834	842	848	873				
2	2871	1915	1436	1103	828	819	823	828	831	842	852	873	896		
1	5741	3828	2871	1915	1437	1103	828	818	823	828	832	842	852	873	896

(D) Regularly-scattered: Execution Times

	Columns														
Rows	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024
128	12,82														
96	12,68														
64	12,58	12,60	12,63												
48	12,44	12,47	12,51												
32	12,29	12,34	12,33	12,39	12,33										
24	12,13	12,18	12,16	12,17	12,19										
16	11,89	11,99	11,98	12,01	12,00	12,05	11,93								
12	11,63	11,78	11,83	11,87	11,86	11,81	11,88								
8	11,25	11,41	11,53	11,71	11,74	11,73	11,71	11,75	11,65						
6	10,89	11,15	11,28	11,50	11,63	11,65	11,63	11,54	11,67						
4	11,53	11,52	11,52	11,16	11,31	11,52	11,55	11,53	11,52	11,52	11,28				
3	10,28	10,42	10,56	10,95	11,08	11,36	11,45	11,48	11,46	11,31	11,44				
2	10,23	9,99	10,06	10,53	10,67	11,01	11,14	11,39	11,41	11,38	11,35	11,35	10,92		
1	12,80	11,42	10,02	9,79	9,91	10,43	10,54	10,88	11,09	11,34	11,35	11,31	11,24	11,24	11,15

(E) Regularly-scattered without L1 cache: Execution Times

	Columns														
Rows	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024
128	10,49														
96	10,38														
64	10,27	10,04	10,22												
48	10,09	9,78	10,07												
32	9,90	9,73	9,86	9,89	9,68										
24	9,61	9,48	9,58	9,59	9,52										
16	9,13	9,20	9,33	9,38	9,27	9,36	9,06								
12	8,57	8,88	9,06	9,12	9,05	8,96	9,02								
8	7,79	8,02	8,42	8,89	8,85	8,90	8,80	8,85	8,63						
6	7,10	7,50	7,79	8,41	8,65	8,70	8,63	8,44	8,69						
4	8,29	8,30	8,31	7,60	7,85	8,29	8,37	8,34	8,34	8,30	7,83				
3	6,97	6,58	6,63	7,21	7,25	8,02	8,14	8,14	8,16	7,84	8,05				
2	7,94	6,79	6,44	6,61	6,59	7,17	7,22	7,84	7,86	7,82	7,80	7,86	7,30		
1	10,91	8,78	7,64	6,64	6,29	6,42	6,42	6,73	7,13	7,74	7,69	7,67	7,48	7,62	7,34

TABLE I

EXECUTION TIMES FOR BENCHMARKS CONSIDERED (PART 1). TIME IN MILLISECONDS.

of performance between 0.3% to 7.5% for block sizes with maximum Occupancy.

The iterative block-products version performs worse than the naïve implementation for the same shapes. The naïve version achieves better reutilization of data, as all warps work on the same parts of the first matrix during the dot product evolution. Naive multiplication algorithm is more suitable for multi- or many-cores architectures with cache hierarchies.

B. Non-coalesced accesses

1) Regularly-scattered accesses: Table I(D) shows the execution times of regularly-scattered accesses benchmark. As discussed in section III-A3, this type of code does not need to maximize Occupancy, due to the big amount of simultaneous memory requests which latencies cannot be hidden. Moreover, reducing the number of threads per block also alleviates the global memory bandwith bottleneck. We observe that

(A) Vector reduction

	Columns										
Rows	32	48	64	96	128	192	256	384	512	768	1 024
1	1,1087	0,9048	0,7893	0,6852	0,6582	0,6268	0,6358	0,6312	0,6330	0,6635	0,7885

(B) Naive Matrix Multiplication

	Columns										
Rows	32	48	64	96	128	192	256	384	512	768	1 024
32	6441										
24	5842										
16	5218	6094	6579								
12	5121	6478	5979								
8	4982	5862	5265	5479	6470						
6	4860	5775	5293	5940	5457						
4	6177	4746	4855	4898	4915	4743	6066				
3	7960	5918	4653	4928	4649	5421	4520				
2	11890	8121	6103	4415	4339	4325	4450	4288	6172		
1	23730	16086	12073	8399	6967	5855	5866	5909	6120	5951	7223

(C) Random access

	Columns										
Rows	32	48	64	96	128	192	256	384	512	768	1 024
32	*										
24	347,27										
16	388,41	345,63	*								
12	334,81	367,85	347,79								
8	330,47	332,90	388,28	347,56	*						
6	324,86	325,46	334,77	369,78	347,23						
4	325,50	347,18	330,47	334,99	388,41	347,18	*				
3	328,07	357,41	324,82	327,35	334,75	369,59	347,30				
2	370,40	326,11	325,48	324,88	330,43	334,72	388,42	347,17	*		
1	634,43	490,32	370,41	328,04	325,47	324,82	330,43	334,80	388,49	347,45	*

TABLE II
EXECUTION TIMES FOR BENCHMARKS CONSIDERED (PART 2). TIME IN MILLISECONDS.

	Columns										
Rows	32	48	64	96	128	192	256	384	512	768	1 024
32	30										
24	344										
16	352	568	38								
12	362	257	342								
8	357	625	366	331	64						
6	357	580	372	196	306						
4	201	610	334	354	300	256	124				
3	177	379	291	313	264	166	190				
2	200	343	253	277	255	250	249	246	245		
1	366	603	437	490	513	529	520	509	504	496	488

TABLE III
NAIVE MATRIX MULTIPLICATION. L1 CACHE MISSES.

the best performance is obtained for blocks with only 24 threads (shapes with 1×24, or 2×12 threads). As reasoned in section III-A3, having even some idle SPs per warp is compensated by the reduction of the memory bandwith bottleneck, and the bank conflicts.

Table I(E) shows the execution of this benchmark with L1 cache deactivated. As expected, the smaller memory transaction segments have a big impact on performance, obtaining improvements in the range of 20% to 40% for block sizes of 16 or more threads. Reducing the transfer segments alleviate the global memory bandwith problem up to the point that it is not needed to have idle SPs, and the best block size moves from 24 to 32 threads per block (warps with all threads active).

2) Random accesses: Table II(C) shows the execution times of the random accesses code. This program uses many registers for the computation of the random indexes. Thus, the Occupancy is reduced for all shapes. The cells with a star indicate block sizes that cannot be execute due to an exhaustion of resources that leads to Occupancy zero.

This program has a medium ratio of global memory accesses per arithmetic operations (2 global memory accesses vs. 20 arithmetic operations). This load is not enough to eliminate the effect of improving performance when reducing the block size, up to 192 for maximum Occupancy. As expected, the improvement is less noticeable than in kernels with lower workload. For this type of codes, the block size is the key decision. The shape is not relevant due to the random access pattern used on each thread, that distributes global memory accesses across global memory banks.

The medium workload on the threads helps to hide the global memory access latencies when the L1 is active. Although this code has a non-coalescent pattern, the deactivation of L1 cache only improves the performance slightly (less than 1% for any shape).

VII. OpenCL RESULTS

All the results obtained with OpenCL 1.0 and 1.1 consistently show the same effects discussed for the CUDA results. OpenCL introduces a performance penalty on the Fermi architecture tested, due to the OpenCL abstraction layer.

We have tested the mechanism provided by OpenCL to select automatically the block size and shape. For all our benchmarks, the results indicate that this mechanism systematically chooses threadblock sizes of 1024 threads, maximizing the number of threads per block. However, due to the Fermi architecture particular features, this block size does not maximize Occupancy, Thus, the performance obtained with this mechanism is always far from the optimum, with performance degradations between 28% and 65% in our benchmarks.

This technique can easily be improved using a simple and conservative strategy, selecting blocks of 768 threads for big kernels, and 192 for small ones. More sophisticated techniques can be devised using code analysis to detect other code features, such as coalescing vs. scattered accesses, ratio of global memory accesses, total workload, etc.

VIII. RELATED WORK

The most appropriate and common code tuning strategy is to choose the threadblock that maximize the SM Occupancy in order to reduce the memory latencies when accessing global device memory [1]. The authors focus on block shapes that simplify the programming task, such as square shapes, or dimensions that are power of two. They do not explore the relationships between the threadblock size and shape and the underlying performance impact on the hardware resource utilization. The work by Wynters [7] shows a naïve matrix multiplication implementation where several threadblock sizes are tested on pre-Fermi architecture. However, the threadblock shapes are not considered. The authors in works like [8]–[11] use advanced compilation techniques to transform high-level constructors into optimized CUDA code. However, these frameworks use automatic optimizations that only take into account the pre-Fermi state-of-art tuning techniques. A simple performance model is introduced in [12]. Nevertheless, none of these works relate their results to the critical choice of size and shape of threadblock. In [13] the authors show several global memory access strategies in an attempt to maximize the Coalescing factor, as well as other common tuning strategies. This work is developed on pre-Fermi architecture and does not consider global programming parameters, such as L1 cache configurations and threadblock size and size choice. In [14] the authors show how to use the GeForce 8800 NVIDIA hardware resource in order to improve the SM Occupancy. A brief discussion about the different tuning strategies is also included. These studies are focused on pre-Fermi architectures. In [15], [16] the authors show several interesting metrics related to hardware architecture. The authors use these metrics in an attempt to predict the performance of CUDA kernel codes once the block shape is manually chosen. In addition, the Ocelot's transaction infrastructure is presented where several optimization are automatized on PTX (Parallel Thread Execution) low-level code. Focusing on Fermi, in [17] the authors show how the cache memory hierarchy helps to take advantage of data locality significantly improving the global performance. However, taking into account the cache hierarchy leads to a very complicated performance prediction model. They do not consider the effects of the block size and shape.

IX. Conclusions

One of the most important decisions when programming a GPU is to choose global programming parameters, such as threadblock size and shape, in order to achieve the highest performance. However, these parameters are usually chosen by a trial-and-error process.

The choice of global parameters are closely related to the particular parallel problem implementation. We show in this paper that a combined analysis of the knowledge of a specific GPU card architecture, and code features such as the type of global memory access pattern (coalesced vs. scatter), the total workload per thread, the ratio of global memory read-write operations, can significantly help to choose important programming parameters, such as threadblock shape and L1 cache memory configuration.

While OpenCL introduces a mechanism to automatically select the appropriate threadblock size and shape, our results show that its current strategy is far from optimum, with performance degradation of up to 65% in our benchmarks. Using architecture knowledge to choose an appropriate block shape for each particular application leads to important performance benefits.

Acknowledgements

This research is partly supported by the Ministerio de Industria, Spain (CENIT OCEANLIDER), MICINN (Spain) and the European Union FEDER (CAPAP-H3 network TIN2010-12011-E, TIN2011-25639), and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative.

References

[1] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, Feb. 2010.

[2] J. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66 –73, may 2010.

[3] NVIDIA, "Whitepaper: NVIDIA's next generation CUDA compute architecture: Fermi," 2010, http://www.nvidia.com/object/fermi_architecture.html,Last visit: Nov, 2010.

[4] P. M. Greg Ruetsch, "Nvidia optimizing matrix transpose in cuda," http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/CUDA/website/C/src/transposeNew/doc/MatrixTranspose.pdf, Jun. 2010, Last visit: Dec 2, 2010.

[5] NVIDIA, *CUDA CUBLAS Library*, 2010.

[6] R. Nath, S. Tomov, and J. Dongarra, "An improved magma gemm for fermi graphics processing units," *Int. J. High Perform. Comput. Appl.*, vol. 24, pp. 511–515, November 2010. [Online]. Available: http://dx.doi.org/10.1177/1094342010385729

[7] E. Wynters, "Parallel processing on nvidia graphics processing units using cuda," *J. Comput. Small Coll.*, vol. 26, pp. 58–66, January 2011.

[8] A. Leung, N. Vasilache, B. Meister, M. M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction," in *Proc. GPGPU'10*, Pittsburgh, PA, USA, Mar. 2010, pp. 51–61.

[9] M. Wolfe, "Implementing the PGI accelerator model," in *Proc. GPGPU'10*, Pittsburg, PA, USA, 2010, pp. 43–50.

[10] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "An optimizing compiler for GPGPU programs with input-data sharing," in *Proc. PPoPP '10*, Bangalore, India, 2010, pp. 343–344.

[11] M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs," in *Compiler Construction*, ser. Lecture Notes in Computer Science, R. Gupta, Ed. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2010, vol. 6011, ch. 14, pp. 244–263.

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

[13] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," in *"ASPLOS' 11.* New York, NY, USA: ACM, 2011, pp. 369–380.

[14] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. PPoPP '08*, Salt Lake City, UT, USA, 2008, pp. 73–82.

[15] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling GPU-CPU workloads and systems," in *Proc. GPGPU'10*, Pittsburg, PA, USA, Apr. 2010.

[16] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A framework for dynamically instrumenting gpu compute applications within gpu ocelot," in *"GPGPU-4".* New York, NY, USA: ACM, 2011, pp. 9:1–9:9.

[17] C. Z. Xiang Cui, Yifeng Chen and H. Mei, "Auto-tuning dense matrix multiplication for GPGPU with cache," in *Proc. ICPADS'2010*, Shanghai, China, Dec. 2010, pp. 237–242.