J Supercomput (2013) 65:1150–1163
DOI 10.1007/s11227-013-0921-z

uBench: exposing the impact of CUDA block geometry in terms of performance

Yuri Torres · Arturo Gonzalez-Escribano · Diego R. Llanos

Published online: 3 April 2013

Abstract The choice of thread-block size and shape is one of the most important user decisions when a parallel problem is written for any CUDA architecture. The reason is that thread-block geometry has a significant impact on the global performance of the program. Unfortunately, the programmer has not enough information about the subtle interactions between this choice of parameters and the underlying hardware.

This paper presents uBench, a complete suite of micro-benchmarks, in order to explore the impact on performance of (1) the thread-block geometry choice criteria, and (2) the GPU hardware resources and configurations. Each micro-benchmark has been designed to be as simple as possible to focus on a single effect derived from the hardware and thread-block parameter choice.

As an example of the capabilities of this benchmark suite, this paper shows an experimental evaluation and comparison of Fermi and Kepler architectures. Our study reveals that, in spite of the new hardware details introduced by Kepler, the principles underlying the block geometry selection criteria are similar for both architectures.

Keywords GPU · Benchmarking · CUDA · Fermi · Kepler · Performance measurement

1 Introduction

Currently, Graphics Processing Units (GPUs) are among the most powerful High Performance Computing (HPC) devices. The amount of single cores and the low power

Y. Torres · A. Gonzalez-Escribano (✉) · D.R. Llanos
Universidad de Valladolid, Valladolid, Spain
e-mail: arturo@infor.uva.es

Y. Torres
e-mail: yuri@infor.uva.es

D.R. Llanos
e-mail: diego@infor.uva.es

consumption are two characteristics of the hardware accelerators that make them a valuable resource for high-performance computing. CUDA is a high-level parallel language for NVIDIA GPUs that aims to reduce the programmer's burden in writing parallel applications. In GPUs, the thread-block size and shape has a significant impact on both performance and GPU hardware resources usage. In CUDA, it is always needed to specify the thread-block and Grid size-shape. So far, many programmers do not use any thread-block selection criteria, thus spending a significant amount of time to find a good configuration by trial-and-error. There exist tools to automatically select the block size and shape, but they cannot be applied to any kind of problem, and/or they do not lead to good choices in some situations (see, e.g., [1]).

In this paper, we present uBench, a complete suite of micro-benchmarks to explore the impact on performance of (1) the thread-block size and shape choice criteria, and (2) the GPU hardware resources and configurations. This benchmark suite covers the hardware details of Fermi [2] and Kepler [3] architectures. Each micro-benchmark has been designed as simple as possible to focus on a single effect derived from the hardware and thread-block parameter choice. This paper correlates the block geometry with several performance issues, including bottlenecks in access to GPU global-memory or L1/L2 caches, global-memory bank conflicts, or thrashing on L1 cache memory. As a case study, we have used uBench to compare both Fermi and Kepler architectures, analyzing their similarities and differences related to block geometry and L1 cache memory configuration. Our experimental results show that, in spite of the new hardware details introduced by Kepler, the principles underlying the block geometry selection criteria are the same for both architectures. The uBench suite is available upon request.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 briefly reviews the architecture of CUDA supported GPUs. Section 4 presents the uBench micro-benchmark suite. Section 5 shows an experimental evaluation on different architectures and discusses the results. Finally, Sect. 6 concludes our paper.

2 Related work

In this section, we discuss some related work on the choice of an adequate block geometry for NVIDIA CUDA supported GPUs, and prior attempts in GPU benchmarking.

Regarding the thread-block geometry choice, the most common policy is to choose a thread-block that maximizes the SM (Streaming Multiprocessor) Occupancy. Thus, maximizing opportunities to hide the latencies when accessing global memory of the device [4, 5]. Many authors focus on block shapes that simplify the programming task, such as square shapes.

Focusing on Fermi, in [6], it is shown 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. We have presented in [1, 7] a practical study of the Fermi architecture, focused on how the thread-block parameters and the use of hardware resources affect performance. These works do not consider each single hardware effect separately or how new hardware resources may affect the thread-block selection criteria.

The use of benchmarks to evaluate hardware configurations has a long tradition. In this work, we focus on benchmarking of GPU devices.

The authors in [8] introduce a suite of micro-benchmarks to measure GPU performance, and how it changes when a specific optimization strategy is used. The authors measure execution times, deriving latencies for the same thread-block configuration. The study focuses on pre-Fermi architectures.

In [9], the authors introduce a performance model for pre-Fermi GPUs. This model is based on the results of a set of micro-benchmarks in order to measure the time of each kind of instruction, and the time of global/shared memory accesses. The authors always use the same thread-block shapes (square), showing the associated memory data transfer bandwidth.

In summary, these works do not systematically explore a wide range of the thread-block configuration space, and do not relate the thread-block configuration with the underlying hardware effects. Moreover, several of these tests have been conducted using previous NVIDIA CUDA architectures.

3 Brief review of NVIDIA GPUs architectures

Pre-Fermi is the first NVIDIA CUDA supported architecture, launched in early 2007. Fermi is the second generation of CUDA architectures [2], launched on early 2010. The latest generation of CUDA architecture is Kepler [3, 10], released on early 2012. Table 1 summarizes their main characteristics.

Each new architecture generation has increased the number of SPs (Streaming Processors), and the maximum number of threads, per SM (Streaming Multiprocessor). This leads to different relations between the thread-block configuration and the occupancy on the SMs. Thus, the policies to select a good thread-block configuration are potentially different.

The main change introduced by Fermi is a transparent L1/L2 cache hierarchy that has been maintained in Kepler. However, the sizes and configurations possibilities are different. The programmer can select to enable/disable the L1 cache. When the

Table 1 Summary of CUDA architecture parameters (pre-Fermi, Fermi, and Kepler)

Parameter	Pre-Fermi	Fermi	Kepler
SPs (per-SM)	8	32	192
Max. number of blocks (per-SM)	8	8	16
Max. number of threads (per-SM)	1024	1536	2048
Max. number of threads (per-block)	512	1024	1024
L2 cache	–	768 KB	≥512 KB
L1 cache (per-SM)	–	0/16/48 KB	0/16/32/48 KB
Size of global memory transaction	32/64/128 B	32/128 B	32/128 B
Global memory banks	8–9	5–6	4

L1 cache is active, the size of the cache and local SM memory has two possible configurations in Fermi, and three in Kepler.

The size of the memory transaction segment can be adjusted. In Pre-Fermi it is automatically chosen by the compiler, while in Fermi and Kepler is associated with the L1 configuration chosen. Nevertheless, this maximum size is always 128 bytes. This size is relevant for the alignment of data in memory.

The global memory is organized is several banks. The number of banks has been decreased on Fermi and Kepler. Concurrent data-accesses directed to the same bank produce conflicts that can affect the performance. Thus, the number of memory banks becomes important for decisions related to code optimizations, data alignment, and thread-block shape.

4 The uBench suite

We present in this sections the uBench micro-benchmark suite. These benchmarks are designed to explore different performance effects for a significant set of thread-block sizes and shapes.

4.1 uBench design principles

We first discuss the design principles of the uBench micro-benchmark suite. To better control the cache use, and isolate effects derived from patterns used to access the global memory, all benchmarks use as input/output parameter a single array structure. Some benchmarks logically access to it as a two- dimensional matrix, while others access to it as a vector.

4.1.1 Data sizes and storage order

Matrices are stored in row-major order. All benchmarks work with integer elements. Kernels working with single-precision float elements have a similar behavior as integer computations. Float elements use the same memory space as integers, and each SP also has one unit for single precision arithmetic operations. On the other hand, double precision numbers require double memory space, but there are not as many double precision arithmetic units as SPs. Instead, two SPs are coordinated to issue one single double precision operation. Thus, considering one double precision number as two floats, most performance effects can be extrapolated.

The number of threads launched by the uBench kernels is equal to the number of matrix elements. We have designed micro-benchmarks that fulfill the following guidelines: (a) each thread access only one global memory location, a different one for each thread, and (b) each thread access several global memory locations with a given pattern, exploring effects that appear in scenarios where there are more input data elements than the number of kernel threads.

Coalescing is one of the most important issues that affect the code performance. Different micro-benchmarks explore different classes of coalescing patterns. Nevertheless, negative performance effects can also appear due to conflicts in global memory banks (this effect is known as Partition Camping [11, 12]). Due to the introduction of L2 cache memory in Fermi and Kepler architectures, the global-memory bank conflicts are significantly reduced on applications with high transaction-segment reutilization. However, with low data reutilization, the L2 cache has a limited beneficial effect, and the global-memory bank conflicts can appear. We design microbenchmarks with different data reutilization degrees to test this effect.

4.1.2 Data sizes and alignment

Threads accessing out of bounds of the data structures should be avoided in kernels. In codes that correlate thread indexes with data-structure indexes, CUDA programmers tackle this problem in two possible ways. Either the kernels include divergent branches to skip processing for out-of-bounds threads, or data padding is added to the data structures to align them with the chosen thread-block shape. In both cases, the performance impact is very small. More irregular applications may need more sophisticated codes to deal with the alignment of threads and data structures. Their behavior can be extrapolated from results of micro-kernels that test access patterns not correlated to thread indexes in array structures.

Alignment of data structures to thread-block sizes. To keep the micro-kernel and launcher codes simple, we select matrix sizes with dimensional cardinalities, which are multiples of any of the thread-block geometries to be considered in the study. In this way, we can avoid data padding, or trivial divergent branches, without losing generality. Due to the maximum number of threads per thread-block supported by Fermi and Kepler architectures (1024 threads) the thread-block shapes should fulfill the following criteria: (1) (#rows and #columns) $\in [1, 1024]$; (2) (#rows $\times$ #columns) ≤ 1024. To reduce the search space, we only use thread-block geometries where (3) #rows and #columns are multiple of two and/or three. This ensures that we include in the tests all possible combinations that can derive in maximum Occupancy in any current CUDA architecture. Recall that the maximum number of concurrent threads per SM in Fermi is 1536, which is a multiple of three.

Memory bank alignment. In order to selectively introduce memory access patterns that reproduce or skip the effects of the global-memory bank conflicts described above, the cardinalities should also include multiples of (a) the number of memory banks, and (b) the width of the memory-bank, for any CUDA architecture studied.

Choosing the order of magnitude of array sizes. Coalescing patterns can effectively hide global memory latencies when there are enough warps scheduled in the SM during the computation. If the total amount of threads is not enough to fill all the SMs in the GPU device, there are not enough active warps to hide global memory latencies. In our study, the total amount of threads in the whole computation (TT) is related to the total size of the data set. The maximum number of active threads in the whole device (MT) is the product of the maximum of active threads per SM by the number of SMs (recall that in Fermi MT $= 1536 \times [14, 16]$, while in current Kepler release MT $= 2048 \times 8$). We choose matrix sizes with the following criteria: (1) TT less than MT; (2) TT slightly higher than MT (latency hiding may start to happen); (3) TT much higher than MT (latency hiding can be fully exploited). To keep execution times bounded, for this last category we select two different matrix sizes: The first one, for the kernels with high computational load, is 1024 times bigger than the size in category (2); the second one, for those kernels with low computational load, is

9×1024 times bigger than the size in category (2) (to generate an array that is near to fill up the global memory of the devices).

The matrix sizes chosen to achieve all the previous criteria in the different CUDA architectures considered are 96×96, 192×192, and 6144×6144 or 18432×18432.

4.1.3 Computational load and independence of L1 cache configuration

The micro-kernels are designed to execute at most one basic arithmetic operation with each data element accessed. Some of them include an extra loop with dummy computations on constant values, or on data read, to generate a configurable load by changing the number of iterations. Thus, overload can be added to ensure that the computation times between memory accesses are higher than the global-memory latencies. This allows to test the impact of hiding global-memory latencies by coalescing, and/or by overlapping computation with communication.

All uBenchs are designed to correctly work regardless of the L1 cache configuration. This allows to test the performance behaviors using the different cache configurations (enabled with 16 K, enabled with 48 K, or disabled).

4.1.4 Access patterns

To isolate the effects produced by single global-memory access patterns, the first subset of uBench micro-kernels are designed with only one pattern. These kernels do not read data: They use constant values or computed data to write in global memory. To study the interaction of read-write patterns, a second set of micro-kernels combine more sophisticated read patterns with a simple writing pattern.

These read/write patterns do not intend to represent the whole space of access patterns that can appear in an application. Instead, we focus on pattern classes that can produce different performance trends due to hardware effects. The selected patterns cover: The main categories of coalesced patterns; patterns that spread concurrent accesses across the global memory banks vs. patterns that stress the bank-conflicts; and patterns that interleave accesses across thread-blocks vs. patterns that make threads in different blocks traverse data structures simultaneously.

Regarding writing patterns, we have chosen first two basic types of coalesced patterns. Both represent patterns where each thread access only one data element. Then we also add three more types to study non-coalesced patterns, and other special situations.

– *Pattern I:* Each thread accesses one matrix element using the thread global coordinates (y index indicates the row, and x index indicates the column):

$$\begin{aligned} \boldsymbol{row} &\leftarrow \mathit{blockIdx.y} \times \mathit{blockDim.y} + \mathit{threadIdx.y} \\ \boldsymbol{column} &\leftarrow \mathit{blockIdx.x} \times \mathit{blockDim.x} + \mathit{threadIdx.x} \end{aligned}$$

– *Pattern II:* Each thread is assigned to a single uni-dimensional coordinate used to access the array as a vector. For a given grid, independently of the thread-blocks shape and size, each thread always accesses to the same data position.

$$\begin{aligned} \boldsymbol{index} \leftarrow\ & (\mathit{blockIdx.y} \times \mathit{gridDim.x} + \mathit{blockIdx.x}) \times \mathit{blockSize} \\ & + (\mathit{threadIdx.y} \times \mathit{blockDim.x} + \mathit{threadIdx.x}) \end{aligned}$$

- *Pattern III:* The threads use their local block indexes to compute a flattened index: $threadIdx.y \times blockDim.x + threadIdx.x$. Then each thread writes in the matrix element corresponding to element of the main diagonal with the thread index on both dimensions. This completely non-coalesced pattern ensures that all threads in the block accesses to a different transaction segment. But all blocks access to the same small set of transaction segments. Thus, there are not cache trashing effects, and there is high reutilization across blocks.
- *Pattern IV:* Only one thread per block (the one with $threadIdx.x = threadIdx.y = 0$) writes in a matrix position selected as in Pattern I. The remaining threads do not perform any access. This pattern recreates the effect of sparse patterns but, due to minimum number of accesses per block, there are no cache thrashing effects involved.
- *Pattern V:* All threads access to the first position of the array ($vector[0] = value$). This pattern has been designed to produce a high degree of memory bottleneck.

Finally, reading patterns include different types of coalescing and memory alignment techniques, including examples in which threads read many data elements. They produce different degrees of data reutilization.

- *Pattern A:* Each thread reads the full column of the matrix with its global x index. It is a perfectly coalesced pattern with reutilization of the data by other threads of the same column in different iterations. Thus, reutilization inside the block is dependent on the exact shape. In different iterations, the same memory banks are accessed.
- *Pattern B:* Each thread reads the full row of the matrix with its global y index. Data that are in consecutive positions in global memory are read in different loop iterations. This pattern can be considered coalesced in the sense that on each read operation, threads in the same warp are accessing to data in the same transaction segment. There is reutilization of the transaction segments on the cache across iterations, and also by threads in the same row. Reutilization inside the block is dependent on the exact shape. The accesses of all the threads from the same block are concentrated in the same memory bank on each iteration, cyclically changing the bank as the loop advances.
- *Pattern C:* The threads compute a starting position in the array using the block global indexes and the block size. All the threads in the same block obtain the same position. Threads from different blocks obtain positions, which differ in a multiple of the block size. The threads execute the same loop to read all the array positions corresponding to the block. All threads in the same block reuse the same data independently of the shape.
- *Pattern D(s):* All threads traverse once the whole array structure as a vector. However, threads from different blocks start at a different position, traversing the vector cyclically. The position is computed using the global block identification: $blockIdx.y \times gridDim.x + blockIdx.x$, multiplied by a stride parameter (s). When $s = 1$, there is a high overlapping of blocks accessing to the same transaction segments, producing bank conflicts, but there is also a very high reutilization of L1 and L2 caches. When $s = 32$, we ensure that each block is accessing to different transaction segments, and the accesses are balanced across memory banks. Bank conflicts are reduced but reutilization of caches, specially L2, also decreases.

4.2 uBench suite description

This section includes an enumeration and short description of the benchmarks included in the uBench suite.

uBench-0 This kernel is designed to test the performance impact of the device schedulers when facing different thread-block shapes, without memory access interferences. It does not do computation, and neither do the threads access any data.

uBench-1 No read. Each thread copies the same constant value in its position using Pattern I.

uBench-2 No read. Each thread copies the same constant value in its position using Pattern II.

uBench-3 No read. The same as uBench-1 but with an overload loop with one thousand iterations.

uBench-4 No read. The same as uBench-2 but with an overload loop with one thousand iterations.

uBench-5 No read. Each thread copies the same constant value in a position using Pattern III.

uBench-6 No read. Only the thread with $threadId.x = 0$ and $threadId.y = 0$ of each block stores in its global matrix position the same constant value.

uBench-7 No read. Each thread copies a calculated value to the first vector position. The values are calculated using a loop of one thousand iterations.

uBench-8 Each thread copies in its position the sum of the matrix values in the column with its global x index.

uBench-9 Each thread copies in its position the sum of the matrix values in the row with its global y index.

uBench-10 The same as uBench-8 but with an overload loop with one thousand iterations.

uBench-11 The same as uBench-9 but with an overload loop with one thousand iterations.

uBench-12 Each thread sums the values of a matrix block selected with Pattern C, and stores the result in a position selected using Pattern II.

uBench-13 Each thread stores in its position the sum of all the elements of the whole data structure. Each thread starts to cyclically traverse the array at a different position with stride 1.

uBench-14 The same as uBench-13 but using stride 32 to compute the starting position.

4.3 uBench classification criteria

The uBench benchmarks have been classified according to the following criteria, summarized in Table 2:

1. Types of global-memory access patterns.
2. Ratio of arithmetic instruction per thread compared to the number of global memory access (high, low, or none).
3. Ratio of L1 cache memory lines evictions compared to the size of this memory (low, medium, or high).

Table 2 uBench classification, according to the criteria proposed

uBench	(1) Access patterns	(2) Load ratio	(3) L1 eviction ratio	(4) Data reutilization in/across blocks
uBench-0	-.-	None	None	None
uBench-1	-.I	Low	Low	Low
uBench-2	-.II	Low	Low	Low
uBench-3	-.I	High	Low	Low
uBench-4	-.II	High	Low	Low
uBench-5	-.III	Low	Low	Low/High
uBench-6	-.IV	Low	Low	Low/High
uBench-7	-.V	High	Low	High/High
uBench-8	A.I	Low	High	Shape/Medium
uBench-9	B.I	Low	Medium	Shape/Medium
uBench-10	A.I	High	High	Shape/Medium
uBench-11	B.I	High	Medium	Shape/Medium
uBench-12	C.II	Low	Medium	High/Low
uBench-13	D(1).I	Low	Medium	High/High
uBench-14	D(32).I	Low	Medium	High/Medium

4. Ratio of global memory data reutilization across threads in the same block, and across blocks, compared to the number of global memory accesses per thread (low, medium, or high). We also consider a special class (*Shape*) for those benchmarks in which in-block reutilization is dependent on the exact shape of the block.

5 Evaluation and experimental results

Experiments have been conducted for all the benchmarks described using both Fermi and Kepler architectures. We have explored, in terms of execution time, all the combinations of sizes for each thread-block dimension that complies with the shape restrictions proposed (1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, 256, 384, 512, 768, 1024), L1 configurations (enabled, disabled, increased), and input data sizes (96, 192, and 6144 or 18432).

To show an example of how the results obtained with uBench can be extrapolated to real life applications, we have tested two CUDA implementations of a real-life application: Cannon's algorithm for matrix-matrix multiplication [13]. The first implementation is a direct translation of the algorithm, and the second one modifies the order in which matrix blocks are accessed to force each thread-block to start reading from a different global memory bank. The codes are included in the uBench release.

The experiments have been run on a GeForce GTX 480 (Fermi) and a GForce GTX 680 (Kepler) NVIDIA GPU devices. The uBenchs have been developed using CUDA 4.2 toolkit and the 295.41 64-bit driver.

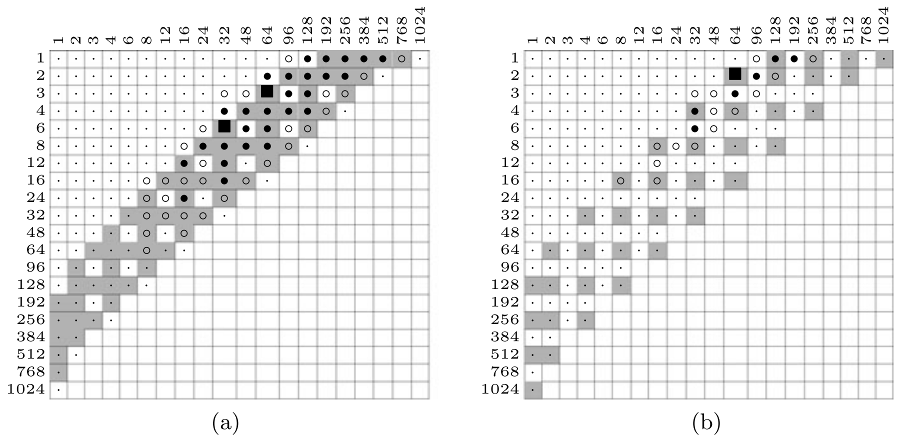

Fig. 1 Example of diagrams for results tables for uBench-1, with default L1 configuration, matrix size 6144 × 6144 (TT ≫ MT), in (**a**) Fermi, and (**b**) Kepler architectures. *Grey* shaded cells indicate block geometries that lead to maximum occupancy. Symbols used: ■ for best performance results; • for execution times up to 5 % more than the optimum; ○ for execution times up to 25 % more than the optimum

All the numerical results are compiled into 270 tables for the uBench kernels, and 36 tables for the Cannon's algorithm implementations. The tables are presented in a technical report publicly available [14]. The tables are accompanied by graphical diagrams. We include two example diagrams in Fig. 1 to help the reader to understand how we used them in the following discussion of results. During the discussion, we consider good choices for thread-block geometry those that lead to execution times with less than 5 % difference with the best execution time obtained with any geometry.

5.1 Thread-block geometry and the scheduling system

The scheduling system of both Fermi and Kepler architectures works faster with blocks of medium size. The results of uBench-0, with no memory accesses and no computation on any thread, show this effect. As the total number of threads and blocks increases the good thread-block choices concentrate more and more in the geometries with a medium size (384 in Fermi, 256 in Kepler), independently of the shape. Compared with Fermi, Kepler presents a performance degradation of up to 20 % for the good thread-block geometries. Kepler's block/warp scheduling system is slower than Fermi's. There are many more units than in Fermi, and they are more complex. This effect is compensated by the faster memory system. The impact of this effect can also be partially seen in kernels with scarce data accesses and low computation, like uBench-6.

5.2 Coalesced patterns for simple writing operations

The thread-block geometry choice has a great impact in the performance of typical coalesced patterns. Results for uBench-1 to uBench-4 show the key trends. uBench-1 explores a classical coalesced pattern, with low ratio of arithmetic operations per thread, and reutilization of the same transaction line only due to coalescence. Not only the size, but the shape of the thread-block, highly influence the performance (see Fig. 1). Good results are obtained for the smallest block sizes that lead to near maximum occupancy (more than 90 %), and shapes with a number of columns equal to or greater than the size of the SM's scheduling unit: Half a warp in Fermi (16 threads); a whole warp in Kepler (32 threads). Smaller blocks derive in faster replacement of the blocks that finish by new blocks, in the SM queues. While in Fermi similar good results are obtained for sizes from 192 to 512, the trend to obtain better results for smaller blocks is more clear in Kepler, where good candidates are those with 128 and 192 threads. In Fermi architecture, even blocks with 128 threads (leading to occupancy of 67 %) also derive in good performance. These results confirm and generalize previous observations [1].

uBench-2 results also confirm that the performance obtained for good candidates in uBench-1 can be obtained for any shape of the same size if the proper pattern can be devised for the specific application.

For a very small number of total threads there are not enough warps to hide latencies due to coalescing. Thus, the restrictions on the thread-block shape, and the policy of trying to achieve maximum occupancy of the SMs become less relevant. This is confirmed by the results of all these benchmarks with small input sets ($TT < MT$).

When there is a high ratio of arithmetic operations per data access, the global-memory latencies are overlapped with computation, obtaining good performance for all sizes and shapes that lead to near maximum occupancy. uBench-3 and 4 are versions of 1 and 2 with extra computational load. Their results for enough number of threads $TT \gg MT$, show this effect. The same effect is also noticeable in the results of uBench-7, that also includes an overloaded loop that allows to hide memory latencies.

For these benchmarks, when $TT \gg MT$, performance for the best thread-block geometry choices is not affected by changing the L1 cache configuration.

5.3 Non-coalesced and scarce patterns

Non-coalesced patterns cannot benefit from the previous global-memory latency hiding effects. We do not find shape restrictions, and for certain applications the best performance results are found with small thread-block sizes that lead to medium occupancy factors. This effect is more noticeable in Fermi due to the faster scheduling system. For example, uBench-5 results for Fermi, show that the best choices are blocks with 64 threads, while in Kepler the best choices are blocks with 128 to 192 threads. uBench-6 uses a pattern where only one thread per block writes. But each block writes in a different position. The amount of writing operations is very small and there is no memory bottleneck. Due to the small amount or writing operations the good block sizes are bigger than for uBench-5: 256 to 768 threads in Fermi, 256 to 512 in Kepler.

Changing the configuration or the L1 cache does not significantly change the performance results for the best thread-block geometry choices, even in the presence of a non-coalesced writing pattern.

5.4 Reading multiple data elements with coalesced patterns

When each thread traverses whole parts of a data structure, the amount of data reutilization across the threads of the same block depends on the form of the shape.

For example, uBench-8 and uBench-9 present different types of coalesced patterns, while traversing a complete row or column of a matrix. uBench-8 has the classical coalesced pattern with consecutive threads in a warp reading consecutive data elements from the same transaction segment. Good results are obtained for the smallest block sizes that lead to near maximum occupancy, and shapes with a number of columns equal to or greater than the size of the SM's scheduling unit. The same conclusions as presented for uBench-1, which have a similar pattern for writing.

uBench-9 presents a complete different type of coalescence, where consecutive threads in the warp access to the same data element in the same loop iteration, and to consecutive data elements across loop steps. There is only one transaction segment required per block row simultaneously. Results indicate that in this case, the columns limitation due to the scheduling unit also appears, but any block size that achieves a near to maximum occupancy produces good performance. uBench-10 and 11 are versions of uBench-8 and 9 with a loop that introduces extra computational load between consecutive accesses. As expected, the exact shape becomes irrelevant and any block with a size that produces near maximum occupancy obtains good performance results.

A different scenario appears when threads in the same block highly reutilize the same data, and there is no reutilization across blocks. In this case, the good choices for block size are related to the transaction segments size, like in classical tiling techniques. For example, the results of uBench-12 show that the good block sizes are between 24 and 32 for both architectures.

The impact of L1 cache configurations is much more noticeable in Kepler than in Fermi. Kepler supports twice the same blocks and threads in an SM than Fermi, while the L1 cache size is the same.

There are several techniques to alleviate memory bottlenecks that lead to performance improvements. In general, these techniques improve performance without changing significantly the conclusions about the thread-block geometry choice. For example, uBench-13 overlaps the accesses of consecutive blocks of the grid. Results show similar behaviour as the reading coalescing pattern in uBench-8, with L2 cache alleviating the bank conflicts to obtain better performance. uBench-14 tries to alleviate memory bottlenecks distributing the accesses of consecutive blocks across banks, like it is explained in [11]. For uBench-14, the good thread-block geometries are those with 384 threads, independently of the shape.

5.5 Extrapolation to real life applications

Observation of the main loop in the threads of the CUDA implementation of Cannon's algorithm easily reveals the access patterns for the two input matrices (Pattern A, and Pattern B, respectively), and for the output matrix (Pattern I). The results show that the choice of the thread-block geometry is influenced mainly by Pattern A and Pattern I (smallest block sizes that lead to near maximum occupancy, and shapes with columns equal to or greater than the size of the SM's scheduling unit). But the influence of Pattern B also relaxes the first condition, leading to similar performance in blocks with slightly bigger or smaller sizes than the previous good candidates.

For the modified version, we apply the technique discussed in [11] to alleviate bank conflicts spreading accesses from consecutive blocks to different banks. As expected, the results show performance improvements for the same good thread-block geometries. For the best thread-block choices, we observe a reduction in execution times of 23 % in Fermi, and 18 % in Kepler. It is more noticeable in Fermi due to its higher number of banks.

6 Conclusions

This paper introduces uBench, a suite of micro-benchmarks designed for exploring the impact on performance derived from the combination of (1) the thread-block geometry choice criteria, and (2) the GPU hardware resources and configurations. The suite has been tested with the NVIDIA Fermi and Kepler architectures. Our results show that uBench can be used to get a deeper insight of the performance impact of the thread-block geometry choice for different architectures and applications. This understanding improves the ability of a programmer to develop better policies for thread-block selection, and also to apply code tuning techniques. Finally, uBench can be used as a test bed for autotuning techniques that automatically select the thread-block geometry.

Acknowledgements This research is partly supported by the Ministerio de Industria, Spain (CENIT OCEANLIDER), MINECO (Spain) and the European Union FEDER (MOGECOPP project TIN2011-25639, CAPAP-H network TIN2010-12011-E and TIN2011-15734-E), Junta de Castilla y León (VA172A12-2), and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission—Capacities Area—Research Infrastructures Initiative.

References

1. Torres Y, Gonzalez-Escribano A, Llanos DR (2012) Using Fermi architecture knowledge to speed up CUDA and OpenCL programs. In: Proc. ISPA'2012, Leganes, Madrid, Spain, 2012
2. NVIDIA (2010) NVIDIA CUDA programming guide 3.0 Fermi
3. NVIDIA (2012) NVIDIA CUDA programming guide 4.2: Kepler
4. Kirk DB, Hwu WW (2010) Programming massively parallel processors: a hands-on approach, February 2010. Morgan Kaufmann, San Mateo
5. Ryoo S, Rodrigues CI, Baghsorkhi SS, Stone SS, Kirk DB, Hwu WW (2008) Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proc. PPoPP'08, Salt Lake City, UT, USA, pp 73–82
6. Xiang Cui CZ, Chen Y, Mei H (2010) Auto-tuning dense matrix multiplication for GPGPU with cache. In: Proc. ICPADS'2010, Shanghai, China, December 2010, pp 237–242
7. Torres Y, Gonzalez-Escribano A, Llanos DR (2011) Understanding the impact of CUDA tuning techniques for Fermi. In: Intl. conf. on high performance computing and simulation, HPCS 2011, pp 631–639
8. Wong H, Papadopoulou M-M, Sadooghi-Alvandi M, Moshovos A (2010) Demystifying GPU microarchitecture through microbenchmarking. In: Proc. ISPASS'2010, March 2010, pp 235–246
9. Zhang Y, Owens J (2011) A quantitative performance analysis model for gpu architectures. In: Proc. HPCA'2011, February 2011, pp 382–393

10. NVIDIA (2012) NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Last visit: June 2012. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf
11. Greg Ruetsch PM (2010) NVIDIA optimizing matrix transpose in CUDA, June 2010. Last visit: December 2, 2010. http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/CUDA/website/C/src/transposeNew/doc/MatrixTranspose.pdf
12. Aji AM, Daga M, Feng W-c (2011) Bounding the effect of partition camping in GPU kernels. In: Proc. 8th ACM int. conf. on computing frontiers, ser. CF'11. ACM, New York, pp 27:1–27:10 (online). Available: http://doi.acm.org/10.1145/2016604.2016637
13. Cannon LE (1969) A cellular computer to implement the Kalman filter algorithm. Ph.D. dissertation, Montana State University, 1969 (online). Available: http://portal.acm.org/citation.cfm?coll=GUIDE/&dl=GUIDE/&id=905686
14. Torres Y, Gonzalez-Escribano A, Llanos DR (2012) uBench: performance impact of CUDA block geometry. Dept. Informatica, Universidad de Valladolid, Tech. Rep. IT-DI-2012-0001, December 2012. http://www.infor.uva.es/investigacion/publicaciones.html