CUDA tuning and configuration parameters on Fermi architecture

Yuri Torres*,
Arturo González Escribano*,[1],
Diego R.Llanos Ferraris†,

* *Dpto. Informática, Univ. Valladolid, Spain*

ABSTRACT

Writing a NVIDIA CUDA parallel program is easy. However, efficiently exploiting the underlying GPU hardware resources and capabilities is a task for CUDA experienced programmers. The programmer has to know the CUDA tuning techniques and choose the optimal global configuration parameters, such as the sizes of the L1-cache and the threadblock to reach a good performance. In this paper we present an insight of CUDA global configuration parameters, common code tunning techniques on Fermi architecture, and their impact on performance.

KEYWORDS: CUDA, GPU, Fermi, tuning, configuration parameters

1 Introduction

Nowadays, the different GPU devices provide easy programing interfaces initially designed for graphic-applications environment. CUDA architecture and its associated parallel programming model [KmWH10] is developed by NVIDIA, and was proposed in order to simplify the encoding of parallel general-purpose applications.

At the beginning of 2010 NVIDIA launched the current architecture named Fermi [NVI10]. This architectures presents significant changes compared with the earlier versions, such as transparent L1/L2 cache hierarchy, configurable shared memory and faster atomic operations.

On the other hand, there are several common tuning optimization strategies [KmWH10] oriented to exploit efficiently the underlying hardware resources. They include Coalescing, loop-unrolling, data-prefetching and Occupancy maximization among others. However, the different effects of these strategies on performance change with each new architecture release.

[1]E-mail: {yuri.torres|arturo|diego}@infor.uva.es

In CUDA programing model it is necessary to divide the threads space into blocks, named threadblocks with a specific size. This configuration parameter, and the global memory access pattern of the threads, affect significantly the SM Occupancy and the accesses Coalescing. These are important factors to exploit the GPUs capabilities.

In this paper we introduce a new insight into the relationships among global parameters, Occupancy, and global-memory access patterns.

2 Related work

The common code tuning strategies such as Coalescing, loop-unrolling and Occupancy maximization are used to exploit, as far as possible, the underlying hardware resources and capabilities, and they are well explained in [KmWH10]. Kerr et.al. [KDY10] discuss the different tunning strategies and also show several metrics related to hardware resource usage. Schaa, in [Sch09], introduces a model to predict the execution time in multi-GPU systems. However, all these works do not consider the Fermi architecture, and do not take into account the global configuration parameters as the main influential factors for performance.

3 Fermi architecture

Fermi was introduced at early 2010, as the current NVIDIA's CUDA architecture [NVI10]. The main changes in this new architecture which are relevant for our study are:

- **Threadblocks and warps**: The number of single processors(SP) are multiplied by 4 reaching 32 SPs per SM. The current number of threads per threadblock has increased from 512 to 1024. Each SM supports at most 1536 concurrent threads. Finally, two different half-warps are executed at the same time in the same SM, requesting two memory transaction in the same cycle if the accesses are perfectly coalesced.

- **Transparent L1/L2 cache memories**: This new architecture introduces two cache memories. Each SM has 64KB on-chip memory, divided into shared memory and L1 cache. However, the programmer may choose between two different possibilities: 48KB of shared memory and 16KB of L1 cache (default option), or 16KB of shared memory and 48KB of L1 cache memory. Besides, the L1 cache memory can be deactivated by the programmer with the appropriate compiling flag.

- **Bank conflicts and device global memory access**: Both pre-Fermi and Fermi architecture organize the device global memory in different banks, each one with an independent memory controller. Thus, simultaneous memory accesses to the same bank are serialized. When many threads access to the same memory bank, this bottleneck produces a problem named Partition Camping [GR10]. This problem may be alleviated in Fermi exploiting adequately the cache hierarchy.

4 Experiments and results

In this section we show an experimental study to deduce appropriated choices for a good L1-cache and threadblock sizes, in order to improve performance by a better GPU resources

usage. We discuss results for some or our benchmarks [TGEL11]:

- **Matrix addition**: This is a bi-dimensional problem where the threads present a simple and coalesced memory access pattern. Each matrix element is accessed only once, and the elements are not reused. Figure 1 (a) shows the execution times obtained using different threadblock shapes. The best results are closely related with threadblocks that maximizes the SMs Occupancy (256 and 512 threads). Threadblocks with less than 32 threads do not fill a single warp. Therefore, some SPs are idle. Threadblocks with 32 or more threads, but a shape with less than 32 columns, have full warps without a coalesced access pattern. Both cases lead to very bad performance. The rest of Threadblock shapes produce a reasonable performance. However, the best results are obtained for block with maximum Occupancy, marked in the table with bold-face.

- **Naive matrix multiplication**: This is also a bi-dimensional problem ($C = A \times B$) with a different global memory access pattern for each matrix. Each thread is associated with a single C position and computes the dot product of a row of A matrix and a column of B matrix. There is reutilization of data between the threads of the same threadblock. Thus, the cache memories play a significant role. Figure 1 (b) shows the execution times. As expected, the best results are obtained with threadblocks that maximize the Occupancy (256 and 512 threads), with 32 columns or more. However, there are performance differences for the same threadblock size. They are directly related to the usage of cache and global memory banks. More horizontal shapes lead to less Partition Camping effects and less cache misses; except for blocks with one row due to the algorithm. Increasing the L1 cache memory size improves performance [TGEL11].

5 Conclusion and Future work

Writing an efficient CUDA code, choosing a good global configuration, is a complicate task. The programmer has to know the underlying hardware architecture to maximize Occupancy or determine the minimum columns in the threadblock shape that allows to properly exploit Coalescing.

Moreover, the requested cache lines and Partition Camping effects play an important role in the threadblock shape decision. Also, it is important to detect the application features that indicate to the programmer that increasing the L1 cache size is appropriated. The modeling of Fermi cache memories is a very important issue to understand the global memory accesses effects.

We are currently developing more sophisticated benchmarks to better understand the performance effects related to the underlying architecture details. The same experiments are also being implemented in OpenCL parallel model.

References

[GR10] Paulius Micikevicius 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`, June 2010. Last visit: Dec 2, 2010.

Rows \ Columns	1	2	4	8	16	32	64	128	256	512	1 024
1	230.66	115.64	58.13	30.1	15.23	7.8	7.83	4.49	**2.98**	**3.02**	4.03
2	123.95	62.49	31.67	16.51	8.63	4.64	3.17	**3.01**	**3.04**	4.04	-
4	70.00	35.49	18.27	9.69	5.04	3.19	**2.98**	**3.13**	4.53	-	-
8	44.68	22.62	12.23	6.14	3.52	**3.05**	**3.04**	4.44	-	-	-
16	32.27	17.30	9.57	5.14	**3.13**	**3.12**	4.22	-	-	-	-
32	27.81	15.59	8.64	**4.94**	**3.45**	4.50	-	-	-	-	-
64	31.02	16.44	**9.06**	**5.57**	5.84	-	-	-	-	-	-
128	40.32	**20.05**	**11.64**	9.13	-	-	-	-	-	-	-
256	**45.08**	**24.45**	15.61	-	-	-	-	-	-	-	-
512	**52.23**	28.98	-	-	-	-	-	-	-	-	-
1 024	72.61	-	-	-	-	-	-	-	-	-	-

a) Matrix addition execution time

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	-	-	-	-	-	-	-	-	-	-

b) Naive matrix multiplication execution time

Table 1: Matrix sum and Naive matrix multiplication execution times both in milliseconds

[KDY10] Andrew Kerr, Gregory Diamos, and Sudakhar Yalamanchili. Modeling gpu-cpu workloads and systems. In *Third Workshop on General-Purpose Computation on Graphics Processing Units*, Pittsburg, Pennsylvania, USA, 4 2010.

[KmWH10] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, February 2010.

[NVI10] NVIDIA. *Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2010.

[Sch09] Dana Schaa. Modeling execution and predicting performance in multi-gpu environments. In *Electrical and Computer Engineering Master Theses*, Boston, Mass, 2009. Department of Electrical and Computer Engineering, Northeastern University.

[TGEL11] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Simplyfing cuda tuning techniques for fermi. In *To appear in Proceedings of the Workshop on Exploitation of Hardware Accelerators (WEHA 2011)*. IEEE, 2011.