Benchmarks Based on Anti-Parallel Patterns for the Evaluation of GPUs

International Conference on Parallel Computing (ParCo)
2011, Ghent, Belgium

Jan G. CORNELIS, and Jan LEMEIRE
Vrije Universiteit Brussel (VUB), Dept. of Electronics and Informatics (ETRO), Pleinlaan 2, B-1050 Brussels, Belgium.
Interdisciplinary Institute for Broadband Technology (IBBT), Dept. of Future Media and Imaging (FMI), Gaston Crommenlaan 8 (box 102), B-9050 Ghent, Belgium.
email: {jgcornel, jan.lemeire}@vub.ac.be

Abstract. We put forward "anti-parallel patterns" to guide the parallel performance analysis process. Anti-parallel patterns or APPs are common parts of parallel programs that cause these programs to have less than ideal performance, where the ideal speedup equals the number of processors. We present benchmarks to model the behavior of APPs on parallel platforms. Each benchmark contains only one APP and is configurable to mimic all its instances. We show how benchmarks can be used to qualitatively and quantitatively understand and compare parallel hardware. Experiments with NVIDIA and AMD GPUs reveal their differences.

Keywords. performance analysis, performance modeling, patterns, gpu

Introduction

This paper presents the use of benchmark programs based on "anti-parallel patterns" to evaluate the performance behavior of GPUs. It is part of ongoing research around the use of anti-parallel patterns to guide the performance analysis process.

Anti-parallel patterns or APPs are common parts of parallel programs that cause these programs to have less than ideal performance, where the ideal speedup equals the number of processors. For example, on SIMD platforms branching is an APP, given that all processing elements have to execute the same instruction at the same time. On MIMD platforms branching does not incur performance degradation.

GPUs provide a cost-effective manner to accelerate many applications and have become increasingly easier to program thanks to general-purpose languages like CUDA and OpenCL. Nevertheless, creating a program that fully exploits the performance potential of a GPU remains a challenging task. Furthermore, modifications of the code that might improve the performance on one GPU might have no impact or deteriorate the performance on another GPU.

The next section gives the global goals of anti-parallel patterns.

1. Orthogonality and Goals of APPs

1.1. Orthogonality

The main hypothesis of our research is that the performance of a program running on a parallel platform is a combination of the effects of the APPs it contains. Depending on the orthogonality of the patterns this combination will be either a simple composition or a more complicated function.

If our assumption (partly) holds, it makes sense to model the performance loss caused by an APP on a given parallel platform. We will do this by combining (i) the platform specifications and (ii) the results of running benchmark programs.

We devise benchmark programs as programs that contain only one APP and that are configurable to mimic any instance of this APP. The first use of such a benchmark program is to model the behavior of an APP on a given parallel platform. The second use is to compare different parallel platforms by observing how they behave in the presence of APPs. This is where the focus of this paper lies.

1.2. Goals

We hope to achieve the following goals with anti-parallel patterns:

1. Compare different parallel platforms: running the benchmark programs on different parallel platforms allows us to compare the effect of an APP between them.
2. Model the performance behavior of one APP on a given parallel platform.
3. Obtain a qualitative and quantitative understanding of the performance loss of a program running on a parallel platform by decomposing it into its APPs.
4. Improve the performance of parallel programs by applying remedies or solutions that overcome the parallel overhead caused by the presence of an APP.
5. Provide an umbrella concept to collect the results of related research. We intend to create a catalog of APPs that serves as a central point of information for parallel programmers.

We introduced anti-parallel patterns in [3]. It contained a broad study of the concept and showed its potential. The next section compares two GPUs using benchmark programs. It illustrates the first goal of APPs.

2. GPU Comparison

We used APPs to compare the behavior of two GPUs. We identified four APPs and compared their behavior on each platform using our benchmark programs. For the NVIDIA GTX 280 GPU the benchmarks were written in CUDA; for the AMD Radeon 5850 GPU they were written in OpenCL.

2.1. APPs

We identified and wrote benchmarks for the following anti-parallel patterns:

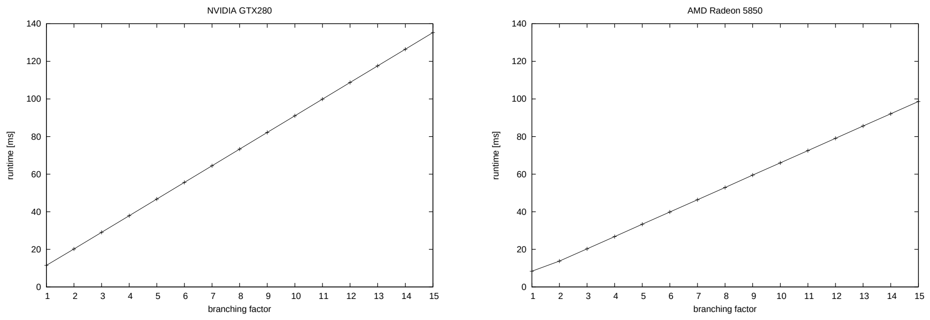


Figure 1. Branching Kernel Runtime (array of 8388608 elements).

APP1, branching refers to code that contains conditional statements, such that, when run in parallel, it causes different processing elements to follow different execution paths.

APP2, lack of parallel data access refers to the lack of concurrent access by different processing elements to shared memory.

APP3, synchronization points are points in the program at which threads should join up to ensure that part of the work has been done.

APP4, partitioning and mapping refers to the way data is mapped to the processing elements that execute the computation.

2.2. Benchmarks and Results

For the sake of brevity we will exclusively use CUDA terminology in what follows. More specifically we will speak about threads, warps and thread blocks. These terms corresponds to work items, wavefronts and work groups in OpenCL terminology.

2.2.1. APP1, Branching

We want to measure the impact of branch divergence by varying the number of branches followed by a warp. Therefore we created a benchmark kernel that contains one big switch statement that branches on the data element that is processed. Each branch executes the same operation. The branching is parameterized. By "abusing" the input array as a map of thread indices and branch numbers, we can mimic any possible branching configuration.

The results are shown in figure 1 for both the NVIDIA and AMD GPU. Our observations correspond with our expectations: the runtime increases in a linear fashion with the average number of branches followed per warp.

Note that this benchmark may be used to determine the number of threads in a warp as follows. Populate the input such that every x consecutive elements have the same value and determine the smallest x for which the runtime is minimal. We found the warp size to be 32 for NVIDA and 64 for AMD.

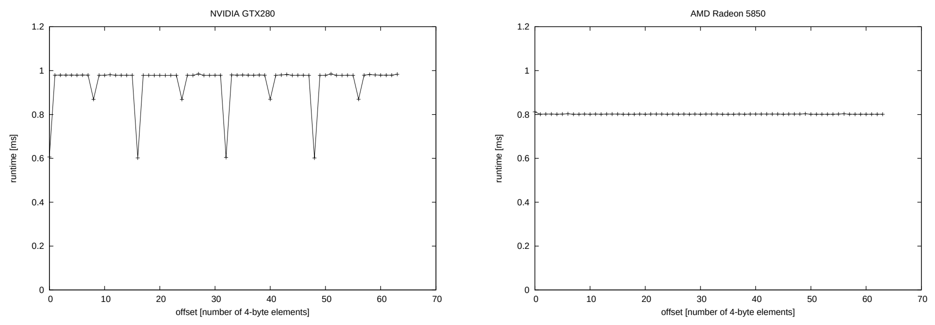


Figure 2. Offset Copy Kernel Runtime (array of 8388608 elements).

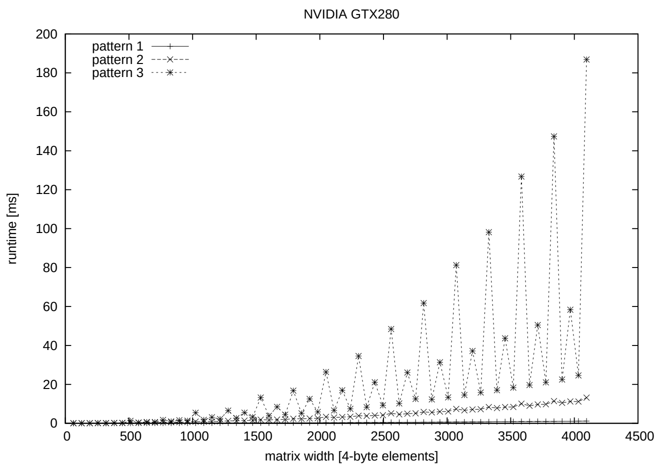
2.2.2. APP2, Lack of Parallel Data Access

We created two benchmark kernels. Kernel 1 copies an input array to an output array. Kernel 2 copies an input matrix to an output matrix. Both kernels can be configured by the memory access pattern for reading and writing the data. Because it is not possible to measure or discuss all possible access patterns, we will look at a small number of them that provide interesting results.

We use kernel 1 to test the effect of “offsetting”. We let a thread copy the element with an index equal to the thread index plus some offset. For NVIDIA offsetting has a negative effect given that different memory transactions are necessary for accessing different memory segments. Indeed the runtime for the NVIDIA GPU is minimal for multiples of 16 4-byte elements or 64 bytes. The AMD GPU exhibits a constant runtime for all offsets (figure 2).

We use a square matrix that we regard as constituted of 16 x 16 matrix tiles as input to kernel 2. We run this kernel in a thread grid that consists of 16 x 16 thread blocks. We will have a look at 3 interesting access patterns. Pattern 1 maps each thread block to the corresponding matrix tile and each thread to the corresponding element. This constitutes the best case. Pattern 2 maps each thread block to the corresponding matrix tile but the thread rows of a thread block are mapped to the element columns of a matrix tile. This introduces strided access with the stride equal to the matrix width. Pattern 3 maps thread block rows to matrix tile columns in addition to mapping thread rows of a thread block to element columns of a matrix column.

When we look at the runtime of our kernels as a function of the matrix width we can make a number of interesting observations (figure 3). First, we expect the runtime to be proportional with the square of the matrix width. This is the case for pattern 1 on the NVIDIA GPU, but for the same pattern on the AMD GPU a peaked behavior is observed. This peak is maximal for a matrix width of 4096 4-byte elements and corresponds to a halving of the data throughput. Second, the slowdown due to access pattern 2 converges to a fixed value (more or less 10) for the NVIDIA GPU while it oscillates between 1 and 5 for the AMD GPU. Finally, access pattern 3 introduces a peaked behavior on both GPUs. The NVIDIA GPU performs worst under these conditions. Here the actual performance degradation depends highly upon the actual matrix dimensions.



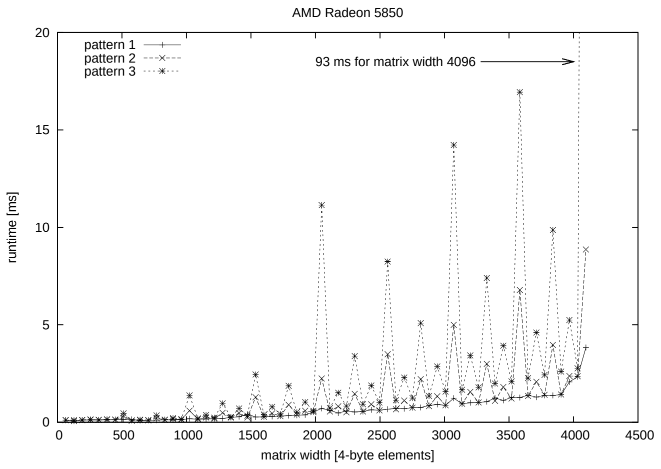


Figure 3. Matrix Copy Kernel Runtime.

These results make explicit the different memory architecture of NVIDIA and AMD GPUs. Furthermore, they can serve as an aid to the developer to translate the memory performance guidelines of both AMD and NVIDIA to concrete facts and figures.

2.2.3. APP3, Synchronization Points

GPUs offer two possibilities to synchronize computations: threads of the same thread block may be synchronized using a barrier; all threads are implicitly synchronized across kernel calls. We will refer to the former as internal synchronization and to the latter as external synchronization. We test internal synchronization using a kernel that executes a fixed number of operations that are interspersed with synchronization calls. The number of operations between two synchronization calls and thus the number of synchronization calls is configurable. We test external synchronization using a kernel that executes a configurable number of operations. We run this kernel the number of times necessary

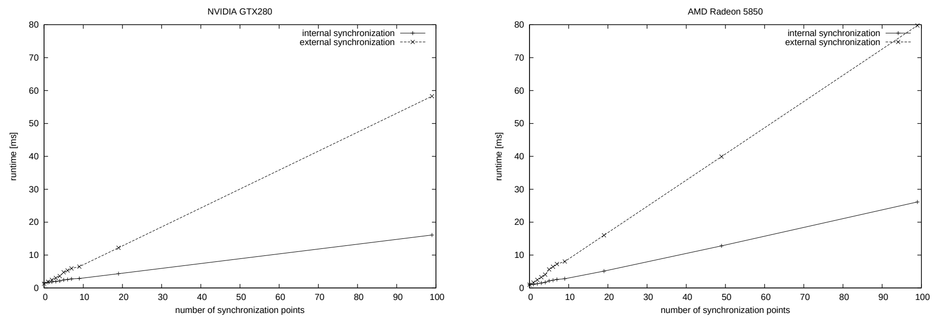


Figure 4. Synchronization Kernels Runtime (array of 8388608 elements).

to execute a fixed number of operations in total. In this case the number of operations executed during one kernel call determine the number of synchronization points.

The results for both GPUs are similar: the runtime is proportional to the number of synchronization points during execution. The runtime of the NVIDIA GPU seems to decrease slightly less than the AMD GPU (figure 4).

2.2.4. APP4, Partitioning and Mapping

The size of the thread blocks and the mapping of threads to data elements can greatly influence the performance of a kernel on both NVIDIA and AMD GPUs. However, it would be very difficult to create a benchmark that measures the effect of both variables. This is because the effect of this APP is highly dependent on the kernel and the data it processes. This combination will determine the amount of latency that is created - mainly due to data access - and the potential of latency hiding that is produced. To get an idea of this phenomenon we have adapted a one-dimensional copy kernel such that each thread copies more than one element. With this adapted kernel we were able to achieve the best throughput at thread blocks consisting of no more than 64 threads. In this case we have increased the data parallelism within one thread. In another case we might also increase the instruction level parallelism within one thread. In general the exact configuration determines both the resource usage and the potential for latency hiding.

The above discussion suggests that this anti-parallel pattern is not orthogonal to the other patterns. We will discuss the first orthogonality results next.

2.2.5. Orthogonality Results

APP1 is clearly orthogonal for both GPUs. The overhead is independent from the occurrence of other patterns. APP3 and APP4 determine to which degree the latency of memory access can be hidden. Furthermore, APP4 immediately determines the memory access patterns (APP2) and thus the amount of latency that should be hidden. As such we ask ourselves the question whether APP4 should be considered as an APP or as a parameter of the other APPs. It is however clear that latency hiding should be modeled explicitly, as has recently been done by [10].

3. Discussion of Related Work

Traditional approaches and tools for performance analysis, for example [4], [6] and [7], help programmers by finding the main bottlenecks caused by a running program that result in a less than ideal performance. Mapping these bottlenecks to parts of the program that should be optimized is done by a tool or left to the programmer. We analyze performance in the opposite direction by mapping parts of the program's source code corresponding to APPs to sources of overhead. We hypothesize that the relevant program characteristics can be determined from the APPs it contains.

Our contribution to current work is the addition of an extra dimension, namely the viewpoint of program patterns. These patterns play an important role in GPU programs because the execution of the different threads is greatly intertwined. On the other hand, the anti-parallel patterns of message-passing programs are rather trivial: the interaction of the programs is explicitly encoded by send and receive calls. These calls generate communication and idle overhead, where the latter might be due to message delays, load imbalances etc. The patterns of message-passing programs having an impact on the performance are therefore limited in complexity and well-understood. The same conclusion holds for multi-threaded programs when each thread runs on individual cores.

We point out that APPs are not in itself a novel idea. A lot of work (for example [8] and [9]) has already been done identifying disadvantageous patterns and proposing solutions or remedies to alleviate their effect on the performance. However the results of these efforts are scattered. As mentioned before, APPs can be used to collect this work and offer programmers a central point of information.

Rodinia [2] provides a benchmark consisting of applications and kernels which choice is inspired by Berkeley's dwarf taxonomy. Our benchmark kernels focus on individual patterns, while their kernels will typically contain combinations of several patterns. GPUBench [1] provides a number of benchmarks to measure performance characteristics of a GPU such as the floating-point memory bandwidth, data upload and readback, instruction rate and numerical precision. In brief, they test peak performance while we test conditions in which the peak performance cannot be attained. Similarly to GPUBench, [10] use microbenchmarks to measure performance in specific situations. Finally, [5] presents a similar idea. They include both patterns that work against available parallelism and patterns of inefficient computation.

4. Conclusion

We presented the concept of anti-parallel patterns and discussed its potential in the domain of parallel performance analysis. We discussed benchmark programs to model the behavior of an APP on a given platform and to compare different platforms qualitatively and quantitatively. In particular we compared an NVIDIA and AMD GPU and revealed both differences (memory access) and similarities (branching and synchronization). Finally we showed that our approach is complementary with the ones followed in traditional performance analysis, but the benchmarks should be extended to model latency hiding.

Acknowledgements

This work was supported by the GUDI project, “A combined GPGPU/FPGA Desktop System for accelerating Image Applications”.

References

[1] Ian Buck, Kayvon Fatahalian, and Pat Hanrahan. Gpubench: Evaluating gpu performance for numerical and scientific applications. In *Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, 2004.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Procs of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[3] Jan Cornelis. Anti-parallel patterns in fine-grain data-parallel programs. Master’s thesis, Vrije Universiteit Brussel, Belgium, 2010.

[4] Mark E. Crovella and Thomas J. LeBlanc. Parallel performance prediction using lost cycles analysis. In *Proceedings of Supercomputing ’94*, pages 600–609, 1994.

[5] Dominic Eschweiler, Daniel Becker, and Felix Wolf. Patterns of inefficient performance behavior in gpu applications. In *Proc. of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Ayia Napa, Cyprus*, pages 262–266. IEEE Computer Society, February 2011.

[6] A. Espinosa, F. Parcerisa, Tomàs Margalef, and Emilio Luque. Relating the execution behaviour with the structure of the application. In *Proceedings of the 6th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 91–98, London, UK, 1999. Springer-Verlag.

[7] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool, November 1995.

[8] Shucai Xiao and Wu-chun Feng. Inter-Block GPU Communication via Fast Barrier Synchronization. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, Georgia, USA, April 2010.

[9] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining gpu applications on the fly: thread divergence elimination through runtime thread-data remapping. In *ICS ’10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 115–126, New York, NY, USA, 2010. ACM.

[10] Yao Zhang and John D. Owens. A quantitative performance analysis model for GPU architectures. In *Proc. of the 17th IEEE Int. Symposium on High-Performance Computer Architecture (HPCA 17)*, pages 382–393, February 2011.