An overview of techniques for predicting the performance of GPU accelerated applications

Rafael Keller Tesser, Philippe O. A. Navaux
Instituto de Informática, Universidade Federal do Rio Grande do Sul - Brazil
{rktesser,navaux}@inf.ufrgs.br

Abstract

The ability to predict the performance of applications in large-scale parallel systems is essential. One of the main incentives for this is the high cost of executing non-production tasks on these systems. An entity may also want to predict the performance in a system that does not yet exist. One popular alternative for increasing a systems performance is the use of accelerators. This created the need for new methods for performance prediction, that take into account the use of these components. In this paper, we will present some recent works in the area of performance prediction in systems with accelerators. More specifically, we will focus on systems that use Graphics Processing Units (GPUs) as accelerators.

1. Introduction

The fastest computer in the world today, according to the Top500 list [1], is the Sequoia supercomputer, from the Lawrence Livermore National Laboratory (LLNL), in the USA. This system scored 16.32 petaflops in the LINPACK benchmark. One petaflop means 10^{15} floating point operations per second (FLOPS). The scientific and engineering community, however, still needs more performance [5]. The amount of data they generate is rapidly reaching the exabytes scale. Additionally, the computational power expected for processing this data is in the exaflops (10^{18} FLOPS) range. The high-performance computing community expects exascale performance to be reached in 2018 [3].

One challenge for attaining such performance lies in the processor technology. According to [5]: "...scaling in the number of transistors is expected to continue through the next decade". However, "clock rates are no longer keeping pace, and may in fact be reduced ... to reduce power consumption". As a result, exascale systems "will likely be composed of hundreds of millions of arithmetic logic units (ALUs)". The greatest challenge, however, is the energy consumption. The Defense Advanced Research Projects Agency (DARPA) of the United States of America has set power limit of 20MW for high-performance computers [11]. Sequoia already consumes more than 7.8 MW.

One way to increase performance without increasing the clock rate of the CPU is to use accelerators. These components can run parts of the application with a higher performance than the CPU. Examples of accelerators are Field-programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs), and the CELL processor. Systems that employ such components are called *heterogeneous* or *hybrid* systems.

Currently, the most used kind of accelerator are GPUs. Those systems have several advantages, like high degree of thread-level parallelism, higher energy efficiency and higher memory bandwidth than the CPU. Additionally, the GPUs are already going toward meeting the two challenges we pointed. First, they have hundreds of processing units, running at relatively low clock rates. Second, they are more energy-efficient than the CPUs that are usually employed in high-performance systems.

Due to the high cost of ownership, running applications in large-scale systems is very expensive. This leads to a big push to improve performance of these applications. The problem is that performance measurements also use machine time. Therefore, it costs money that the owner of the machine is not willing to waste with non-production tasks. One way to overcome this problem is to use prediction techniques to estimate the performance of the applications without actually running them in the target system.

Taking into account the context we presented above, we see an opportunity for research of performance prediction involving hybrid systems. More specifically, we intend to model the performance of accelerated applications. We intend to use this model to develop a model-based simulator. We also want to integrate this performance predictor with a full system simulator.

Our work is still in the planning stage. Currently, we are studying the state of the art on performance prediction, with focus on hybrid systems. In this paper we will present some

recent works on the performance prediction of GPU accelerated applications.

In the next section we deal with hybrid systems, with a special focus on GPU accelerators. In section 3 we talk about performance prediction. In subsection 3.1 we present recently published work on performance prediction of GPU applications. Finally, in section 4, we wrap up the paper and present our conclusions.

2. Hybrid systems

The use of co-processors, like GPUs and FPGAs, to speed up high-performance computers has been a trend for some years now. These components are called *accelerators*. Most of these components have different architectures than the main CPU in the system. So we can say that machines that employ them have *heterogeneous architectures*. As each node systems has a combine general-purpose CPUs with accelerators, we can say that they are *hybrid systems*.

Currently, the most popular kind of accelerator being used in HPC systems are the GPUs [8]. In the subsection below we will present some details about the use of GPU as accelerators for high-performance applications.

2.1. GPUs as accelerators for HPC

The design of GPUs was shaped by the video game industry, which expects a massive number of calculations to be made for each video frame. The solution was to optimize the execution throughput of a massive number of threads. The resulting hardware takes advantage of the large number of threads to maintain the processor occupied when some of them are waiting for memory operations to complete [10]

GPUs can be seen as *many core* processors. This kind of architecture focuses on execution throughput. A NVIDIA Geforce GTX 280 GPU has 240 cores. Each of these cores is a heavily-multithreaded processor, that shares its control and instruction cache with seven other cores [10].

One advantage of GPUs over CPUs is their energy efficiency. DARPA has specified 20 MW as the reasonable power consumption of an exascale system [11]. This would require an efficiency of 20 picojoules (pJ) per floating point operation. According to [8]: "A modern CPU (Intel's Westmere) consumes about 1.7 nanojoules (nJ) per floating-point operation... A modern Fermi GPU consumes about 20 pJ per FLOP".

Another advantage of GPUs over CPUs is the memory bandwidth. For example, the NVIDIA GT 200 chip supports about 150 GB/s. Microprocessor system memory bandwidth will probably not grow beyond 50GB/s until 2013 [10].

Until 2006, GPUs were very hard to use for HPC, because programmers needed to use graphic application programming APIs. In 2007 NVIDIA released CUDA [12], which allowed the programmers to program GPU applications using the C/C++ programming language. OpenCL [9] is a cross-platform parallel computing API based on the C language. Different from CUDA, which is vendor-specific, OpenCL supports multiplatform and multivendor portability. This advantage, however, comes at the cost of a more complex programming model.

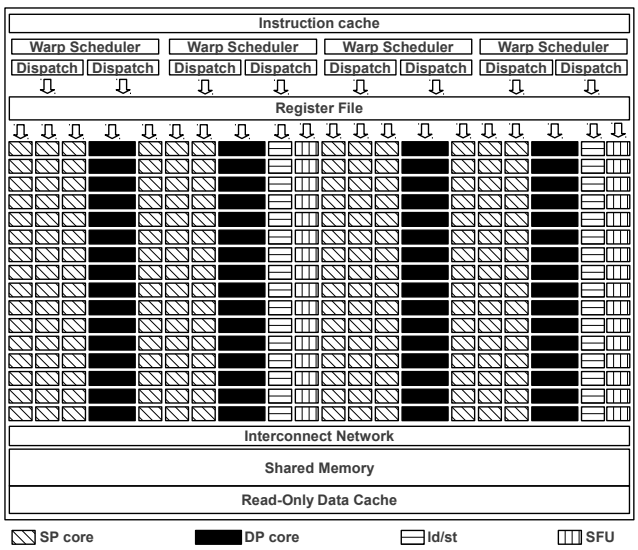

Figure 1. Simplified illustration of a stream multiprocessor

In the next subsections we will touch some aspects of the GPU's architecture and programming model. For the sake of brevity we will focus on the upcoming NVIDIA Kepler GK110 architecture[13] and on the CUDA framework.

2.1.1. GPU Architecture A typical GPU has a large number of cores, capable of executing a massive number of threads in parallel. Another common characteristic is the presence of a memory hierarchy that focuses on improving the throughput of data transfers. The upcoming NVIDIA Kepler GK110 GPU has 15 *streaming multiprocessors* (SM), which share a global memory. Figure 1 shows a simplified illustration of one of these SMs.

Each SM is composed of several processing units. In the Kepler GK110, each SM has 192 single precision (SP) cores, and 64 double precision (DP) units, 32 *special function units* (SFU) and 32 *load/store* (ld/st) units. All the cores in a SM can access a *shared memory* and a *read-only* data cache. Additionally, each SM has a register file to store thread specific variables.

2.1.2. GPU Programming model A typical GPU program contains code to be executed on the host CPU and on the GPU. Functions that will be executed on the

GPU are called *kernels* in CUDA. These kernels generate a large number of parallel threads which are collectively called a *grid*. The grid is further broken into one or more thread *blocks*. All blocks in a grid have the same number of threads. During execution, each block is assigned to a streaming microprocessor. Once assigned to the SM, the block is further broken into *warps*. Warp is the unit of thread scheduling in SMs. When a warp executes a long-latency operation, like a memory access, another warp can be selected for execution. The work done by this new warp will hide the latency in the previously scheduled thread's operation.

In each cycle, the hardware will execute the same instruction for all threads in a warp. This is called *single-instruction, multiple-thread* (SIMT) execution. To improve parallelism, it is important to avoid divergence in the code. It is also important to make good use of the memory hierarchy. One aspect that is crucial to the performance are the accesses to shared memory. The hardware is projected to improve the access to contiguous addresses in this memory, by issuing only one memory request and fetching the data in parallel. So, it is interesting that the threads in a warp access contiguous regions in the shared memory. This is called a *coalesced memory access*.

3. Performance prediction

For expensive large-scale systems, execution cost is large, and often dominant component of total cost [6]. So, it is important to focus in optimizing performance. The problem is that these same costs make it too expensive to use these systems for performance tuning experiments.

To overcome this problem, the developer needs to use performance estimation techniques. The technique that will be used depends on the required accuracy and on the available resources. The choices are [6]: analytical modeling, cycle accurate simulation, and model based simulation.

Analytical modeling uses purely analytical expressions to model application performance. With this technique, one must be careful to balance the number of parameters and the required accuracy. More parameters mean more accuracy. On the other side, too many parameters can make the model too complex to use. Additionally, "the high-level insight that can be gained from the model is generally higher with less parameters" [6].

Cycle-accurate simulation is very accurate, but it is often thousands of times slower than the actual execution, and consumes significant memory resources [6]. Additionally, the low level of the provided information can make it difficult to get some insight on a higher level.

Model-based Simulation is a more abstract (less detailed) form of simulation. Its running time can be less than the actual execution time of the application, while still capturing accurately important details of the application's execution.

3.1. Performance prediction for GPU applications

As part of our research we are studying the state-of-the art on the performance prediction of GPU accelerated applications. Below, we present some recent works in this area.

Hong and Kim [7] proposed a model to estimate the execution time of massively parallel programs in a GPU. Their goal was "to provide insights into the performance bottlenecks of parallel applications on GPU architectures". The basis for their analytical model is that estimating the cost of memory operations is the key for estimating the performance of GPGPU applications. They estimate the number of parallel memory requests that can be executed concurrently by the application. They also calculate how much computation can be done by other warps while one warp is waiting for memory values. Using these two variables, the authors estimate the actual cost of the memory requests, therefore estimating the execution time of the application.

Schaa and Kaeli [14] proposed a methodology to predict the execution time on GPUs while varying their number and the size of the input data. Their objective is to help researchers to choose the most appropriate GPU distribution for their application, without the need to buy new hardware or modify the code for multiple GPUs. The authors take into account two kinds of multiple-GPU configurations: *shared-system GPUs* and *distributed GPUs*. They present a methodology to model the CPU and GPU execution, the PCI express transfer cost, disk paging costs, and network communication cost. Using these models, the authors claim to be able to predict execution times for variable data sets and numbers of GPUs.

Baghsorkhi et al [2] presented a technique to generate a model of the execution of GPU applications. Their technique is based on an abstract representation of the program, called *work flow graph* (WFG). This is an extension of the *program dependence graph* (PDG). They use a technique called symbolic evaluation in certain parts of the code. This way they determine loop bounds, data access patterns, control flow patterns, etc. This information is useful for estimating the effects of control flow divergence, memory bank conflicts and memory coalescing. The WFG is built from the PDG, and is augmented with the information obtained through symbolic evaluation. Additionally, the arcs are weighted according to hardware characteristics. According to the authors, their technique can be used to automatically extract the performance model from a kernel code.

Zhang and Owens [15] developed a micro-benchmark based performance model for NVIDIA GPUs. Their goal is to predict the benefits of potential program optimizations

and architectural improvements. Their technique is based on simple throughput models for the instruction pipeline, shared memory and global memory costs. These models are derived from the results of micro-benchmarks that are executed on the GPU. Their model is based on native GPU code. They use a disassembler on the original binaries, instrument the code, and then reassemble the instructions back to binary code. Additionally, they use the Barra GPU simulator [4], to gather dynamic execution information on how many times each instruction is executed.

Barra [4] is a simulator of Graphic Processor Units (GPUs) for general purpose processing. It receives as input CUDA executables. Barra emulates the GPU and generates detailed statistics about the execution of applications. The simulator emulates the Tesla ISA, therefore it is capable of running native GPU code.

4. Conclusion

The high performance community aims to achieve exascale performance. The main obstacle in their path is energy consumption. The excessive power usage has already lead to difficulties in increasing processor clock rates. This resulted in a major focus on multicore systems. Another consequience was that the community began to explore the use of co-processors, called accelerators, to increase performance.

The need for increasing performance raises the need to tune it to the target system. The problem is that the cost-of-ownership of a large scale system makes it too expensive to run performance experiments in the production environment. The solution is to use prediction techniques to estimate the performance of the applications. With the advent of performance accelerators, we need new prediction schemes that take these components into account.

GPU is the most popular kind of accelerator. We presented some works that use analytical modeling to estimate the performance of GPU applications. We also presented a simulator, that emulates the execution of the application on a GPU, in order to obtain performance statistics. While these are interesting works, there isn't yet a widely accepted prediction technique for GPU applications performance evaluation.

We see an opportunity to explore other approaches to performance prediction of GPU applications. More specifically, we believe we can use model based simulation to estimate GPU applications performance. Additionally, it would be interesting to study ways to use profiling information in order to predict the performance of these applications in different hardware configurations. Another promising research subject would be the integration of the simulation of the GPU applications with full-system simulation.

References

[1] TOP500 Supercomputing Sites, 2012. Available at: <http://www.top500.org/>. Accessed in July 2012.

[2] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An adaptive performance modeling tool for gpu architectures. *SIGPLAN Notices*, 45(5):105–114, Jan. 2010.

[3] D. L. Brown and P. M. Ed. Scientific grand challenges: Crosscutting technologies for computing at the exascale. Technical report, U.S. Department of Energy, 2010. Avalilable at: <http://science.energy.gov/ /media/ascr/pdf/program-documents/docs/crosscutting_grand_challenges.pdf>. Accessed in May 2012.

[4] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A parallel functional simulator for gpgpu. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 351 –360, aug. 2010.

[5] J. Dongarra, P. H. Beckman, et al. The international exascale software project roadmap. *International Journal of High Performance Computer Applications*, 25(1):3–60, 2011.

[6] T. Hoefler, W. Gropp, W. Kramer, and M. Snir. Performance modeling for systematic performance tuning. In *State of the Practice Reports*, SC '11, pages 6:1–6:12, New York, NY, USA, 2011. ACM.

[7] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[8] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31:7–17, 2011.

[9] Khronos Group. The OpenCL Specification (version 1.2), November 2011. Available at <http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf>. Accessed in May 2012.

[10] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[11] P. Kogge. The tops in flops. *Spectrum, IEEE*, 48(2):48 –54, february 2011.

[12] NVIDIA Corporation. Nvidia cuda c programming guide, 2012.

[13] NVIDIA Corporation. NVIDIA's next generation CUDA computer architecture: Kepler GK110, 2012. Available at: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.

[14] D. Schaa and D. Kaeli. Exploring the multiple-gpu design space. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –12, may 2009.

[15] Y. Zhang and J. Owens. A quantitative performance analysis model for gpu architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382 –393, feb. 2011.