Abstract

We are witnessing an increasing adoption of GPUs for performing general purpose computation, which is usually known as GPGPU. The main challenge in developing such applications is that they often do not fit in the model required by the graphics processing devices, limiting the scope of applications that may benefit from the computing power provided by GPUs. Even when the application fits GPU model, obtaining optimal resource usage is a complex task [1].

In this work we propose a profiling tool for GPGPU applications. This tool use a profiling strategy based on performance predicates and is able to quantify the major sources of performance degradation while providing hints on how to improve the applications. We used our tool in CUDA programs and were able to understand their performance.

1 Introduction

Despite the evolution of the various computer architectures and the programming environment and tools, designing and implementing scalable and efficient parallel applications, in particular irregular ones, is still a challenge [1, 2] The wide availability of multi and many-core architectures open a whole new scenario that must be better exploited and understood.

In particular, we have witnessed a continuous development and improvement in high-performance graphic cards (i.e., GPUs) that enabled their use beyond image rendering and they became general purpose co-processors. However, the effective, concurrent use of all available computational power for the creation of efficient parallel algorithms is a challenge, which we address in this work.

Currently, for sake of general purpose processing, GPUs are almost always used as co-processors, that is, the main processor handles portions of the computation that needs to be performed by the GPU, in a fork-join fashion. The application development begins by implementing GPU versions of the functions of interest, called kernels, and changing the original application code to insert calls for those GPU functions. The CPU side of the application also performs all the necessary data transferring. Given this initial version of the application, which are normally inefficient, programmers have to pursue the optimization steps to create a code that fully exploits the GPU processing power.

When porting general purpose applications to GPUs, programmers may observe performance degradation during all phases of GPU usage [3, 4]. First of all, since data may be transferred, we should carefully evaluate whether it is worth using the GPU, considering the amount of computation to be performed. We then need to understand to what extent we are being able to use its processing capacity, which depends on the SIMD parallelism opportunities exploited. Later it is important to observe the application data access pattern, as it can be used to improve the reading operations, for instance, with coalesced data access. The configuration used to execute a code within the GPU is also important, because it can deeply affect the achieved performance [5, 3, 1].

In this massive parallel programming scenario, where programmers are responsible for developing applications that can fully exploit this computa-
tional power, it is also important to create tools that assist during the application development, thus, in this paper we propose a profiling tool for GPU applications.

Our tool uses a profiling strategy based on predicates that define a set orthogonal and complete performance categories which help the programmer to understand the potential sources of performance degradation. These categories are quantified experimentally through automatically-inserted instrumentation. Besides describing the profiling strategy and its implementation, we illustrate out tool may be used for understanding the performance issues of two programs, which were not possible to perceive with other tools.

2 Architecture Overview

CUDA stands for Compute Unified Device Architecture and is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device. It provides a C extension programming language, alleviating the burden on the programmer [5].

In the CUDA programming model, the GPU is viewed as a compute device suitable for data parallel applications. It acts as a co-processor, accelerating computations of host machine.

The GPU has a high number of processors and its own memory and may run a very high number of parallel threads. These threads are grouped in blocks which, in turn, are grouped into a grid. Such structured set of threads is launched on a kernel of code, processing the data stored in the device memory. Threads of the same block may share data through fast unsynchronized on-chip memory. Required access synchronization must be explicitly added by the programmer. Different blocks of threads are independent, only synchronizing at the termination of the kernel.

Figure 1 depicts the architecture of an Nvidia GPU. For example, the GeForce 260 GTX has 24 streaming multiprocessors (SM), each with 8 streaming processors (SP), or cores [3]. Thread instructions are executed by the cores in a SIMD (single instruction, multiple data) model. The scheduling unit used within the GPU is the warp, that’s group of 32 threads, executed by the multiprocessor every four cycles. Each multiprocessor can execute up to 32 warps (1024 threads) in SMT, alternating warp execution while a SMT CPU switch threads.

3 Profiling Strategy

In this section we describe our strategy for profiling GPU applications. We start by describing the measurements that are performed at the thread level and then how they are combined to generate a performance profile.

There are several GPU profiling challenges, as lack of interrupts and SMT. Due to lack of interrupts we chose to measure times instead of periodic sampling. SMT is a great challenge because the multiprocessor transparently switch threads and we don’t have a way to account how many cycles it spent with each thread. For now, if the programmer run multiple warps our tool will report cycles taken by a thread to do some action, but we can’t report the cycles taken by the multiprocessor to perform that action.

3.1 Thread Profiling

The thread profiling is the basis of profiling strategy, since the other levels just combine the measurements taken at the thread level.

We implemented a tool that reads CUDA PTX assembly [6] code and generates instrumented code. During the code instrumentation our tool adds the instrumentation code to the PTX. For example, to measure explicit barriers it adds code to read time and store it in a register before and after the barrier; calculate elapsed time and add it to a third register whose value will accumulate the time spent in barriers.

1PTX is a virtual instruction set that abstracts some architecture details like available registers, so registers are declared like C variables.
As CUDA does not allow to access the GPU %clock from the CPU, times are measured using the gettimeofday() system call when measurement is done from CPU, while we used the GPU multiprocessor %clock cycle counter register when the measurement is done in the GPU.

The measurements at the thread level are taken at a kernel basis, that is, we perform the measurements for each kernel in each thread, combining them to obtaining the thread-level measurements. Our starting points for measurement are the application execution time, measured by the CPU and each kernel execution time, measured inside the GPU. We then divide the execution time of a kernel being executed by a thread $t$ into the following eight orthogonal categories:

Idle ($I$): $t$ is not active. It is calculated by subtracting the kernel execution time and communicating time from application execution time.

Communicating ($C$): A data transfer is being performed between the host CPU and GPU, and no processing is taking place. We use the gettimeofday() system call to measure the time spent by the CPU in cudaMemcpy() function.

Computing ($P$): $t$ is processing. It is calculated by subtracting all other categories from the kernel execution time.

Reading ($R$): $t$ is the time spent by reading data instructions.

Writing ($W$): $t$ is the time spent in writing instructions by the kernel.

Explicit Synchronization ($B$): $t$ is blocked because of an explicit synchronization, that is, executing the function __syncthreads(). These barriers appear in PTX code as bar.sync instruction.

Implicit Synchronization ($S$): Threads from the same warp may take divergent execution paths, and the compiler inserts barriers so that the threads synchronize after such event. $S$ is the time blocked at such barriers. We distinguish between implicit and explicit synchronization because the former is a consequence of code characteristics. The implicit barriers do not appear explicitly in PTX code, but they are always inserted after labels, so we measure the elapsed time to pass through labels. If it is above the single instruction count, we assign such time in excess to $S$.

Wasted Computation ($W$): In order to avoid that threads from a warp perform branches to execute small code segments, the GPU architecture allows the conditional execution of instructions, which may be a more efficient approach for such scenarios. These instructions are executed only if a predicate (a boolean variable) is true, allowing the removal of branches and the threads in a warp to execute together. If the predicate is false, the instruction execution time is accounted as $W$.

3.2 Warp Profiling

Once we take the measurements at the thread level, we combine them at the warp level. We distinguish two groups of categories here. The first group comprises those thread categories that are basically averaged, providing the quantitative measure at the warp level: Communicating ($C$), Computing ($P$), Reading ($R$), Writing ($W$), Explicit Synchronization ($B$), Implicit Synchronization ($S$), and Wasted Computation ($W$). The Idle ($I$) time is split among Inactive and GPU Idle.

Inactive is characterized by the existence of some threads computing and some idle, meaning that there are not enough threads to keep all GPU multiprocessors busy. For instance, for an efficient use of a multiprocessor, it should execute a minimum of 512 threads (50% occupation). Considering that current GPUs contain up to 30 multiprocessors, an application should be split into 15360 threads, which is a parallelism level that may not be possible to many applications.

GPU Idle is characterized by the simultaneous inactivity of all threads. Such inactivity is associated with portions of code that are not executed in the GPU. The intuition here is that we measure the amount of GPU computational power that is not being used.

3.3 Application Profiling

The overall application profile is determined by combining the warp profiles in two dimensions: kernel and application. The kernel profile just averages the warp level profile for all warps, providing a GPU-wide perspective. The profiles of all kernels, as well as the time the GPU was idle are combined as the application profile, which is the starting point for understanding the performance of GPU applications.

4 Experimental evaluation

The experiments were run in a machine with dual AMD Opteron 2350 2 GHz processors, 16 GB DDR2 667 MHz ECC RAM in dual channel, 500 GB SATA disk and Gigabit Ethernet, and
NVIDIA GeForce 260 GTX, with 24 SMs, 896 MB GDDR3 memory (111.9 GB/s bandwidth) connected in a PCI-Express x16 Gen 1 slot. This bus connection provides up to 4GB/s bandwidth.

4.1 Kernels

In this section, we present the programs used to evaluate our framework: (i) Matrix multiplication, and (ii) Quicksort. The Matrix multiplication was chose as it is the canonical example for GPU programming development and its optimization steps are well known. On the other hand, Quicksort [7] was evaluated because it is an irregular and highly optimized application, which performance is more difficult to understand. In the following subsection we discuss each of these programs.

4.1.1 Matrix Multiplication (MM)

Matrix multiplication is widely using in many scenarios and, as discussed before, it is a canonical example of GPU programming optimizations. It is an operation of multiplying a given matrix to a scalar or another matrix, without loss of generality, in our example we multiply two square matrices (A and B), and store the result to a third matrix C.

4.1.2 GPU-Quicksort (QS)

GPU-Quicksort [7] is an efficient sort algorithm suitable for highly parallel multi-core graphics processors. It is designed to take advantage of the high bandwidth of GPUs by minimizing the amount of bookkeeping and inter-thread synchronization needed. It achieves this by using a two-phase design to keep the inter-thread synchronization low and coalescing read operations and constraining threads so that memory accesses are kept to a minimum. It can also take advantage of the atomic synchronization primitives found on newer hardware to, when available, further improve its performance.

In each partition iteration a new pivot value is picked and as a result two new subsequences are created that can be sorted independently. After a while, there will be enough subsequences available that each thread block can be assigned to one. For this reason, the algorithm has two, albeit rather similar, phases: the first when there are less partitions than thread blocks, so the threads need to work together on the same sequences and the second, where each thread block works on one partition.

This algorithm doesn’t use in place sorting because it would require expensive thread synchronization, that would quickly increase as the application in GPU can use thousands of threads and the main reason to use in place sorting is good cache locality, but GPUs doesn’t have caches. The algorithm writes sorted data in an auxiliary buffer and in the end of each iteration, the buffers change their roles.

In partitioning, to allocate space for writing data, each thread counts the number of elements greater and smaller than the pivot and then they compute a prefix-sum to calculate where each thread will write its data. This is an expensive operation and is used even when each thread block work on its own partition.

4.2 Results

4.2.1 Matrix multiplication

In order to evaluate the development process guided by our profiler, we created two versions of MM kernel, while we optimized the second version and shown the time spent in each profiling category for each of them. During this process, each kernel was instrumented and normally executed in the GPU.

The first version of the matrix multiplication, whose code is shown in Figure 2(a), is a multiple blocks kernel. This version creates a thread to calculate each single element of the result matrix, but these threads are grouped into multiple blocks that are responsible for different regions of the result matrix, called tiles or sub-matrices. Thus, at the beginning of the kernel each thread have to identify the data index it should process according to its block and thread ids.

![MM: multiple blocks/per warp](image)
The experiments for this kernel version of MM were performed using matrices 4096 elements (64 * 64), which are divide in 16 blocks of 16 * 16. The configuration with 16 blocks have been chosen to avoid concurrency between blocks into a same multiprocessor, thus each multiprocessor is responsible for just one block. In Figure 3 we present the time spent by each warp in categories. In this configuration, the kernel is able to execute 128 warps, that are divide among 16 SMs in groups of 8 warps in each. The time spent by categories shows that memory loads are the most expensive category.

When memory access is identified as the application bottleneck, the programmer should verify how efficiently the application is exploiting the GPU memory architecture. A simple analysis of the multiple blocks version of MM is sufficient to perceive that it reads each element of matrices A and B from the global memory, what may not be an efficient approach. Thus, reading it to the shared memory and reusing the data could reduce the amortized memory access cost.

Therefore, we optimized the memory access of MM by dividing the source matrices (A and B) in tiles that are read to shared memory, and reused between the threads. As shown in Figure 2(b), threads of the same block read data element to shared memory, synchronize to guarantee that the whole tile have been read, and latter compute the multiplication using data in shared memory. It do not only provides faster access as data is stored in a faster memory, but also optimize the global memory loads as data accesses are contiguously.

Finally, the results for MMMs last version are shown in Figure 4. The average execution time of this kernel warps have been reduced by a factor of 2. It also shows categories that have not appeared or were negligible: barriers, implicit barriers, and stores. The barriers are related to the synchronization instructions introduced behind warp threads, the implicit barriers are more significant as kernel is faster, and stores’ cost have increased because we have a higher level of concurrency over the I/O mechanism. It is important to understand that the optimization presented could be easily detected as the kernel is simple, but more complex kernels require better profilers as may be difficult to understand their performance.
4.2.2 Quicksort

The implementation of QS algorithm, discussed in Section 4.1.2, to GPU been divided in two phases, which can further be implemented using multiple kernels. The first algorithm’s phase was implemented using three kernels: (i) prefix-sum that counts how many number are smaller than pivot, (ii) data copy kernel implements the step of copying data to the one of the partitions a new pivot creates, and (iii) pivot writer which only write the pivot to its position after copying data. The second phase of the QS is implemented by lqsort kernel, that does the same of last three kernels dividing the partitions in blocks which are executed in multiple SMs. For performance reasons, when the partitions’ size are equal or smaller than 512 items it uses bitonic sort to each of them.

In this section we evaluate our three time relevant QS kernels: copy data, prefix-sum, and lqsort. Although our GPU has 24 SMs, experiments were executed with 16 blocks because the second phase of QS requires power of 2 number of blocks, and we would like to evaluate all kernels under the same hardware resources, for instance, 16 SMs are used as one thread block can not be executed in multiple SMs. Thus input data we used in the experimentation has 16 million unsigned integers.

In Figures 5(a) and 5(b), we show the results for prefix-sum and data copy kernels. As presented, the last warps have a higher idle time, which is due to application’s intrinsic characteristics as it uses less threads in the last calls of these kernels during final iterations of the first phase.

In Figures 5(c) and Figure 6, respectively, we can see the second phase kernel and the entire quicksort execution. It becomes evident that the second phase dominates the execution time, being responsible for 85% of it. Is also important to note that both results show a high load unbalancing between thread warps 0-7 and 8-15, as the second set of warps time is dominated by idle category.

The reasons for the detected load unbalancing was directly related to the pivot choice strategy. When we analyzed the application’s first pivot choice we saw that first two partitions have 72% and 28% of the numbers, respectively, which were responsible for the GPU sub-utilization. In order to verify our hypothesis, we manually modified the first pivot value to get a balanced division in first two partitions, actually with 53% and 47% of the values. The performance of this new execution, presented in Figure 7, have reduced the execution time in 13%, while we used the same hardware resources. These gains were derived from a better utilization of the GPU, as load unbalancing (idle time) has been reduced.

In GPU, a common approach to optimize application’s execution time is to execute the maximum number of threads in parallel, as it may better exploit the available resources because, for instance, warps running in a same SM may be in I/O operations while others are doing computations. Due to that, we also evaluated the QS application as we increased the number of warps executing in each SM. Figure 8 shows the average time spent by all threads warps per category as the number of warps are increased.

These results first show that increasing number of threads running in parallel may be efficient, as the execution time have be reduced by a factor of 3. It is also important to see that as we increased the number of thread warps (parallelism), some categories that were negligible as barriers and stores become important to the whole execution time. It also shows the most expensive tasks, as computa-
5 Related work

There are several works about profiling sequential and parallel CPU programs and how programmers can optimize code, based on profiling informations.

The GNU profiler (gprof) is a tool for measuring the performance of a CPU program. It records the number of calls to each function and the amount of time spent for them, on a per-function basis [8]. Functions which consume a large fraction of the run-time can be identified easily from the output of gprof. Efforts to speed up a program should concentrate first on those functions which dominate the total run-time.

Another tool is Paradyn [9], a parallel application profiler. This tool is able to insert instrumentation dynamically, restricting the intrusion only to parts of the code that are being analyzed. The tool was first implemented to profile only distributed applications, but was modified by Xu et al. [10] to support multithreaded applications.

Beside the CPU profilers, there are recent works about the power of multithreaded GPUs and how specific applications can benefit from it, although few of them address general techniques or tools for developing and optimizing applications into such a complex environment.

Boyer et al. present an automated system [11] for analyzing two classes of common problems: race conditions and shared memory bank conflicts. They employed an approach of automatic instrumentation of CUDA codes, which are latter executed under emulation mode (running on CPU). Although this profiler is interesting as it can identify the analyzed classes, the work fails to measure the impact of the memory bank conflicts to the application because it is not running on GPU. In this work we present an approach to profiling CUDA applications, where we instrument the intermediary assembly code of the applications and are able to measure the impact of the most important tasks performed by the applications during execution on the GPU.

The NVIDIA CUDA Visual Profiler [12] (CVP) is provided by NVIDIA for helping the application development process. It creates the application kernels profile using different counters as: coalesced and uncoalesced accesses to the global memory; local loads and stores; and divergent branches. This profile has at least three important limitations we address in this work: (1) the counters it uses are based on events within a warp, so it has an indication of the actual performance of the application but not the real impact of each category of performance it measures, while we show the number of cycles spent in each of them; (2) the analysis is based on only one multiprocessor, so it is not useful for irregular applications where different warps running on several multiprocessors do not have the same profile; and, (3) our system is able to create a profile for each
threads/warp/multiprocessor, and the CVP only shows the profile for the entire kernel on a specific multiprocessor.

The work [3] studies the GeForce 8800 GTX processor’s organization, features, and the optimization strategies adopted by the developers to achieve high performance.

6 Conclusions and Future Work

The paper addressed the performance profiling of GPGPU applications. Our approach defines a complete and orthogonal set of categories and how they may be measured in current GPUs. We illustrate the effectiveness of our approach through two programs, where we were able to detect, measure, and understand their sources of performance degradation.

Future work includes analyzing other programs, as well as improve the instrumentation, for instance, to distinguish between coalesced and non-coalesced memory accesses. Finally, we also intend to investigate whether the integration of dynamic and static profiles are a better option for sake of performance understanding.

References


