CGO: U: Unleashing The Power of Graphics Processing Units (GPUs): Bottleneck Analysis

Yi Lu, Louisa.Lu@nyu.edu, New York University
{ Advisor: Mohamed Zahran, mzahran@cs.nyu.edu }

Abstract

Many computations can be performed much faster on GPUs than on a traditional processor. This is why GPUs are present now in almost all computers (from some portable devices up to supercomputers); and the majority of Top 500 supercomputers in the world are built around GPUs. GPUs are now used for a diverse set of applications and domains. However, making the best use of the GPU's hardware is still far from a trivial task. In this work, we are proposing a technique to allow GPU programmers to identify the main bottlenecks in the code. Bottlenecks here are caused by non-optimal interaction between the software and hardware that leads to slower programs. The GPU programmer cannot modify the hardware, but can modify the software to make it a better match for the underlying hardware. This is our ultimate goal from this study.

Problem & Motivation

Graphic Processing Unit (GPU) is first originated to handle images and videos. However, with the development of more sophisticated software applications, and more flexible and versatile GPU hardware, researchers and professionals start to explore GPU for executing more general applications, trying to leverage GPU's massive data processing capability, hence the name, General Purpose Graphic Processing Units (GPGPU). Figure 1 is one member of NVIDIA GeForce GPU architecture family. All instructions can be executed in parallel in the green section called cooperative thread array (CTA). All GPUs from NVIDIA follow the same design with differences only in the number of CTAs, memory size, or execution units. Despite all that, using GPGPU in general purpose applications is far from an easy task and users often find out that the performance of applications running on GPGPUs are not as fast as they've expected. This research work proposes a way to make GPGPUs more efficient and a bit friendlier to use.

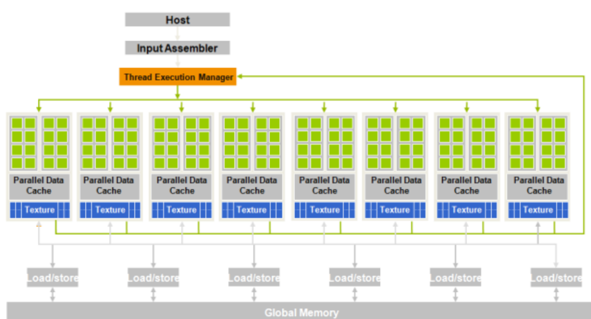

Figure 1 shows NVIDIA Geforce GPU Architecture

In order to see how much far GPUs are from their peak performance, we calculated the theoretical peak performance of a benchmark suite and compared it to the actual performance (the simulator and the benchmark will be shown later in this study). We calculated the theoretical peak in an approximated way by analyzing the code of the each benchmark program, count the number of operations, and see if the GPU under study works in its full computational power (ignoring memory latency and other factors) what performance we get. We found that the theoretical performance is several orders of magnitude more than the actual performance. This means the software and hardware are not playing well together. This is a clear indication to us that there is a lot of room for improvement.

We analyze in details the problem of performance bottlenecks when executing an application on GPUs. Basically, we try to answer questions like: Why is my application not as fast as I expected? Is it due to software issue, or hardware issue? What can I do about it? We then provide a technique that enables programmers to decide how to arrange the code in a way to make the best use of the underlying hardware. This paper starts out by giving a brief introduction on possible bottleneck performance issues a GPU might face, followed by a detailed description regarding the three phases this research undergoes: kernel application characteristic, theoretical performance and optimization framework.

Background & Related Work

The first factor that can be a potential bottleneck is GPU memory system. Actually memory system is a potential bottleneck for all computer systems, not only GPUs, due to the big gap in speed between the memory system and the execution units. Some memory issues that can significantly affect application performances include memory bandwidth

saturation (i.e. not enough bandwidth to feed the execution units or CTAs with data), memory latency (memory does not respond fast enough), PCI bus delivery (i.e. the connection between the GPU memory and the main system memory is not fast enough).

Among these factors, memory bandwidth is much more important than memory latency as the latter can be compensated by GPU's multiple data parallelism programming. That is, doing more computations while other computations are waiting for the memory to respond1. Programs that have this characteristic is called to have data-parallelism. For memory-intensive program (i.e. programs that need to access memory much more than doing computations), research has found that cutting half of bandwidth channels can reduce 50% of the performance [1].

With the development of data parallelism on GPU, data transfer from/to the GPU to/from host (i.e. the main CPU) is becoming more and more important. Although in pre-GPU era, the speed of data transfer between two devices has been fairly uniform, GPU exhibits a non-uniform data transfer between itself and a host. Current design connects the GPU to system's memory using PCI express (PCI-e) bus and unfortunately, PCI express is extremely narrow compared to system's other data path [2]. As a consequence, it is one of the bottlenecks for many applications and it is very important to optimize the data movement to improve performance [2].

Also the more nodes (i.e. GPUs or CPUs) connected to the PCI-e bus, the more complex and inefficient bus scheduling is. This is due to the fact the PCI express has to coordinate more nodes with the memory system and possibly fetching more data from global memory As a result, a system that contains multiple GPUs might end up performing slower than a system having fewer GPUs due to this complication [2].

Besides memory inefficiency, the other big factor that can significantly impact the GPU performance is branch divergence due to irregular control flow [3]. A program executed on GPUs may contain many if-else program structures. GPUs execute the same program in parallel but each instance works with a different data. Each instance is called a thread. If some threads execute the "if" part, and some other threads execute the "else" part then performance drops significantly. This problem is called branch divergence.

So summarize, the main bottlenecks of performance are: GPU memory bandwidth, latency and lack of enough bandwidth for connecting GPU memory to CPU memory, and branch divergence. Of course there are some other factors that affect performance, such as under-utilization of execution units (CTAs). But we cited the major ones.

Methodology

Kernel Application Characteristic

A Kernel is a part of a program assigned by the programmer to be executed on a GPU of a CPU. Our main idea of bottleneck analysis is the following: any piece of hardware has a peak performance, and any piece of software if given a piece of hardware powerful enough also has a peak performance. When we run that piece of software on that piece of hardware we get a performance that is lower than both peaks, because the hardware and software do not play well together due to bottlenecks. We want to identify those bottlenecks. In order to do so, the first step is to know the personality of each kernel (i.e. its characteristics).

First, we profiled each kernel in our benchmark suite and gathered a lot of measurements that we use to characterize each kernel.

As our benchmark suite throughout the research, we simulate ispass2009-benchmarks [4] -using the simulator GPGPU-Sim [5] with GTX480 configuration [6]. The simulated GPU as a memory of about 1.5GB, 15 CTAs (called Streaming Multiprocessor in Nvidia lingo, with maximum bandwidth between GPU and memory of 177.4GB/s, and peak computation performance of 1345 GFLOPS (Giga Floating Point Operations per second).

Table 1: Four types of kernels

Branch Divergence / Percentage of Global Memory	< 200	> 200
> 0.1%	Rabbit	Devil
< 0.1%	Turtle	Tiger

After having a detailed profile on each kernel, we classify them into four intuitive types similar to the study exhibited by [7]. The aforementioned study was done for multicore applications. We used similar logic but for GPU applications. Each kernel is categorized into one of four categorizes based on branch divergence and total memory access, as shown in Table 2.

The first key measurement to determine a kernel application type is potential branch divergence given by GPGPU-SIM (We say *potential* because it is a data dependent measurement. So it is based on the

data we used during profiling. However, many simulations have shown that it is a good prediction for other data too). The second characteristic to determine a kernel type is the ratio of Global memory instruction and total instruction. This measurement gives a good overview on kernel's memory-related operations. The numbers and percentages mentioned above are experimental based on our profiling experiments. Although there are many other measurements that we take to profile a kernel, these two measurements can give a nice prediction on kernel type because they address the most important issues that determine a kernel's performance on GPU and also encompass all the bottlenecks we mentioned above. The four types that we developed to categorize kernels are:

- *Rabbit*: This type of kernel exhibits high percentage of global memory operations but lower branch divergence. Although they are very active, it is relative easy to optimize them.
- *Turtle*: This type of kernel shows both a low percentage of global memory operations and a low branch divergence rate. We name this type Turtle.
- *Devil*: This type shows high percentage of global memory operations like rabbit, but also have high branch divergence, making them the most inefficient kernels, hence the name "Devil".
- *Tiger*: This type of kernel shows a low percentage of global memory operations, but a high percentage of branch divergence. Since Tiger spends the majority of its time sleeping and it moves dangerously when it is awake, we name this type Tiger.

After determining the kernel's type, we get a sense on which area to focus on for optimization and the rest of the work will determine how to and to what extend to optimize different type.

Theoretical Performance Peak

We calculate each application theoretical performance peak based on its code structure and the level of data parallelism. We do that by formulating a mathematical model relating the amount of parallelism in the application (there are several types of parallelism but GPU exploits mainly data parallelism so we concentrate on that type) to the size of input data. Then we get the peak theoretical performance of the GPU that will execute this kernel (from the GPU specifications) and predict the expected performance from executing this kernel on GPU. It is to be noted that the above two peaks are upper bound approximations and used mainly to guide optimizations. A kernel type tells us which area to optimize, and the theoretical peak gives us a sense of how far are we from optimal performance.

Optimization Framework

Besides helping us categorizing each kernel, the profiling step also gives us the *real* performance of each kernel as well as ton of statistics regarding the kernel behavior. The type of each kernel helps us determining which statistics to look at and how to use it to optimize the kernel. So, for example, if a kernel is a turtle, it means it does not suffer either from branch divergence or from memory bottlenecks. This means we have to concentrate on the minor factors such as the hardware utilization. So the programmer has to concentrate on increasing the amount of parallelism in the code either by restructuring the code or by using a different algorithm for the problem at hand. As we can see, using our classification reduces the amount of effort required by the programmer to localize the bottlenecks and lets the programmer concentrate only on the issue that leads to a higher effect. In addition, hardware researchers can also benefit from this framework by understanding how the interaction of hardware with software affects the performance of applications that run on GPGPU.

The next section shows how we profiled and use our profiling information.

Results

Parallelism:

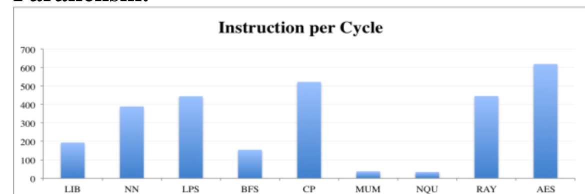

Figure 2 shows instruction per cycle for each kernel

Instruction per cycle gives an indication on how well the GPU is doing executing an application. The higher the number, the more efficient an application is leveraging GPU's parallelism. Although a kernel runs faster with higher parallelism, it is not the only factors that determine efficiency of kernels.

Memory Management

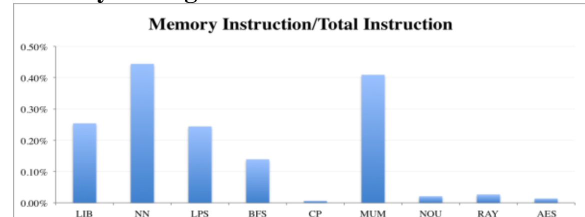

Figure 3 shows the ratio of total memory instructions (read & write) and total instructions

The ratio of memory instruction and total number of instruction is an indicator of how much activity is going on in memory system. MUM, the least efficient application can contribute some of its performance to high proportion of memory-related instructions. Although it takes significantly more time to read and write from
global memory, its effect on kernel efficiency is not direct as in the case of NN, it is still able to reach some efficiency. CP has the least proportion of memory instructions over total instructions, which contribute to its high performance.

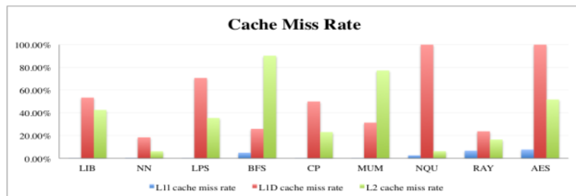

Figure 4 shows cache miss rate at all levels for all kernel applications

L1I is cache level 1 for instruction and L1D is cache level 1 for Data. L1D is a more important measurement for memory management because L1I usually has a low miss rate even in traditional multi-cores. The higher L1D and L2 miss rate, the less efficient memory system is because the kernel spends more time fetching data from global memory. Figure 4 shows that NQU has the highest L1D miss rate even though its memory instruction only counts little fraction of total instruction. Similarly, AES also has a very high miss rate for L1D although unlike NQU, it is still able to execute large amount of IPC. This shows that a kernel's parallelism isn't related to memory efficiency.

Branch Divergence

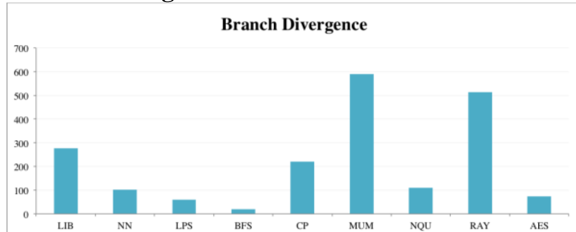

Figure 5 shows the number of cycles when all available warps are issued to the pipeline and are not ready to execute the next instruction.

Figure 5 shows each kernel's branch divergence through wrap occupancy distribution. The higher the value, the more sever the kernel facing the problem of branch divergence. MUM has the most instance of branch divergence, which might be the cause of its inefficiency. Interestingly notice that BFS contains the least amount of branch divergence yet it is not the most efficient kernel applications due to its low parallelism.

Kernel Application Type

Table 2: Kernel type result

Type	Rabbit	Turtle	Devil	Tiger
Kernel	NN, LPS,BFS	NUQ, AES	LIB, MUM	CP, RAY

For Tiger category, where kernel applications have low proportion of memory instructions over total instruction and high proportion branch divergence, the performance can be further optimized to reduce number of branch divergence via job swapping. Although CP is categorized as high branch divergence, its divergence is just near the cutoff point for branch divergence. RAY has a high efficiency regardless of its high divergence might be contributed by high parallelism. For type Devil, both LIB and MUM has the lowest efficiency level and have the most room to improve efficiency. A rabbit type of kernel should focus on improving memory management such as memory coalesce to reduce the time of global memory instructions. A turtle type, since both divergence and memory operation are low, increase parallelism can improve application performance.

Conclusions

The major bottlenecks for GPU applications are branch divergence, memory operations and parallelism. As shown in the results, all three aspects are equally important and they are somehow dependent on each other. For example, an application that has more branch divergence also tends to have a low parallelism. However, an application's parallelism isn't related to memory operations. As a consequence, when a programmer optimizes kernel in the aspect of branch divergence, more often, he also improves the kernel's parallelism.

For each different kernel types, a programmer should focus on optimizing one aspect over another and should arrange different types of kernels to be executed in parallel.

- *Rabbit*: Optimize memory-related issue, such as cache management. Since it has a low branch divergence, it can be executed along an optimized Tiger type.
- *Turtle*: It is the most efficient kernel types. Arrange Devil kernel with it.
- *Devil*: Optimize both memory-related issue and parallelism.
- *Tiger*: Optimize branch divergence through schemes such as job swap.

References

[1] Wang, Guibin. "Memory Access Characterization of Scientific Applications on GPU and Its Implication on Low Power Optimization." *2011 International Conference on Computational and Information Sciences (2011)*. Print.

[2] Meredith, Jeremy, Philip Roth, Kyle Spafford, and Jeffrey Vetter. "Performance Implications of Nonuniform Device Topologies in Scalable Heterogeneous Architectures." *IEEE Micro 31.5* (2011): 66-75. Print.

[3] Wu, H., G. Diamos, J. Wang, S. Li, and S. Yalamanchili. "Characterization and Transformation of Unstructured Control Flow in Bulk Synchronous GPU Applications." *International Journal of High Performance Computing Applications* 26.2 (2012): 170-85. Print.

[4] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. “Analyzing cuda workloads using a detailed GPU simulator ” In IEEE ISPASS, April 2009.

[5] GPGPU-SIM simulation software website: http://www.gpgpu-sim.org

[6] GeForce 480 specifications: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480

[7] Xie, Yuejian, and Gabriel H. Loh. "Dynamic Classification of Program Memory Behaviors in CMPs." *In the 2nd Workshop on CMP Memory Systems and Interconnects (CMP-MSI)* (2008)