Alexander Powell

CSCi 524

12.16.2016

Final Research Report

Evaluating Cache Performance on GPUs

Introduction

In order to achieve any sort of high performance computing, caches performance has to be evaluated in order to take full advantage of temporal and spatial locality. The importance of an efficient cache managing system on today's computers cannot be overstated. The current philosophy is that we should construct our programs and applications to maximize the number of times something is fetched from a cache and minimize the number of times one must fetch from data memory. This is the case because the memory latency is so much slower than cache latency on most modern machines. There are two general directions that multithreaded programming is headed. One on hand we are moving towards machines with many cores, where a chip can hold dozens of cores. This design relies on efficient cache utilization to mask the memory access latency, and are known as many-core machines. However, it's one thing to evaluate cache performance while running a single program. In reality, especially in a GPU environment, many programs will be running simultaneously and have the potential to cause program interference. This is because of the shared access to the cache and main memory, which can affect the cache hit and miss rate of the individual programs. Additionally, parameters like memory bandwidth can play a role in the overall performance of simultaneously running applications.

The alternative to this practice is to build machines that can process numerous threads at the same time. In the past, many-thread machines were more suited towards GPUs for processing graphics and other multimedia applications and generally do not use caches at all, but instead rely on thread level parallelism. There are pros and cons to both approach and some perform better in different environments than others. The interesting thing to note

between the two strategies is that while both extremes might yield a high performance, a valley forms between the two extremes. Figure 1. below displays this behavior.

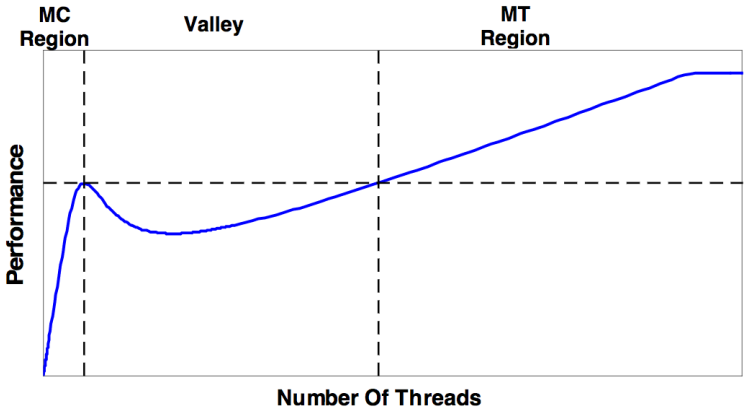

Figure1. Many-core vs many-thread machines

You can see above that on the left side of the graph the performance improves at a rapid rate for each additional thread that is added. This is denoted as the many-core (MC) region. However, at a certain point this performance slows down, and actually begins to drop by some amount before slowly increasing again. Eventually the performance meets and exceeds what it reached initially and keeps increasing for some time until it eventually plateaus. The many-thread (MT) region begins at the point where the performance reaches its last highest point.

The significant thing to take away from this graph is that between the many-core and many-thread regions, there's actually a valley where program performance drops. This is because the cache becomes too small for covering the growing stream of access requests and memory latency is no longer adequately masked by the cache. Guz et.al. propose a model to explain this "valley" effect based on a number of different parameters, like number of processing elements, cache size, cycles per instruction, memory latency, cache latency, frequency and cycle time. By using their model and slightly modified parameters, we can represent a similar, albeit less smooth, figure displaying the valley behavior previously mentioned.

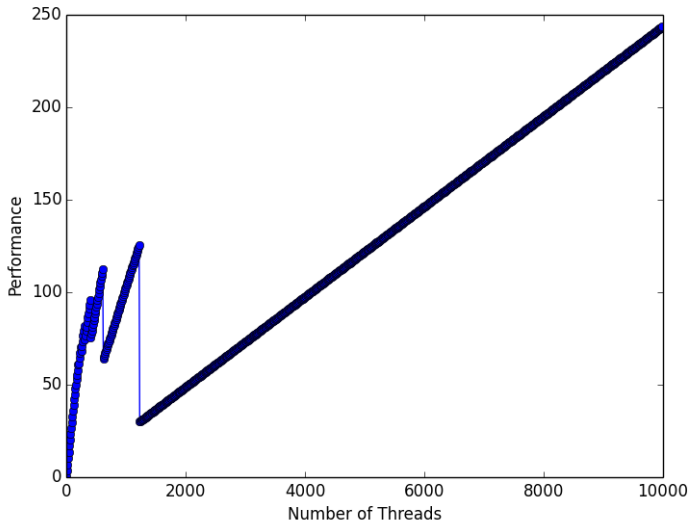

Figure 2. Shows the relationship between the number of threads and the performance in GOPS. While not as smooth as Figure 1., we can clearly see a relatively steady increase in performance until around 1000 threads, and then a sharp drop. The trajectory stays constantly increasing afterwards.

Motivation

For the sake of simplicity, this model assumes that threads can't share any data. Since this is only a model of cache behavior, the parameters can clearly be altered to represent different scenarios. For example, the more computation instructions per memory instructions are given, the steeper the performance climbs. This is due to the fact that as more instructions become available, fewer threads are needed to fill the stall time. This results in the "plateau" being reached sooner and as computation instructions per memory instructions approaches infinity, the valley disappears entirely. Another factor that affects the shape of the plot is memory latency. It's no surprise that as the latency for memory accesses grows larger and larger the effect of the valley will be felt more and more. This behavior is significant because the memory latency is only expected to grow in the coming years. Additionally, as cache size

increases the plot gets shifted to the right. That is, a larger cache will inflate the many-core area.

However, it's not always easy to predict how a given application will perform on a given architecture. In order to update our model, we need to account for both the architecture as well as the application characteristics. Many of the parameters previously mentioned still play a factor in this new model, but now we must also take into account things like off-chip bandwidth and operand size.

Figure 3. below shows how the performance of a program eventually evens out and plateaus. The point that this occurs is known as the bandwidth saturation point.

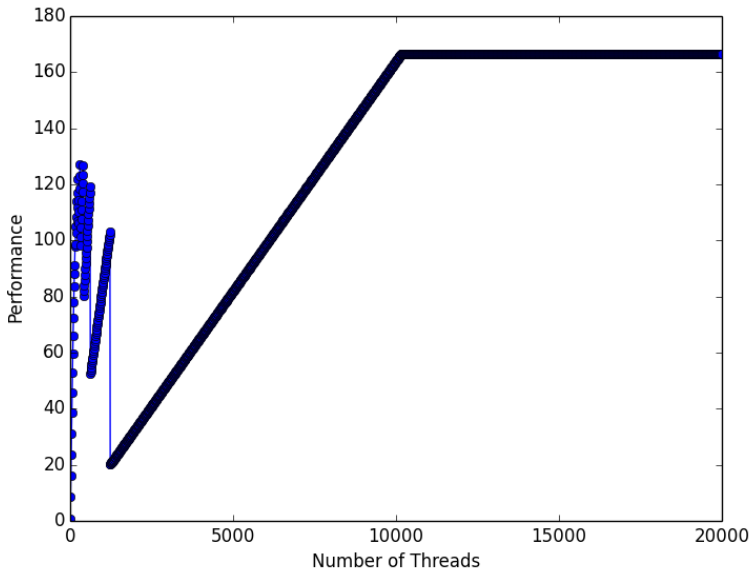

Figure 3. We can see that performance increases with compute intensity, until the bandwidth saturation point is hit.

It's clear that workloads with higher data locality better exploit the cache and therefore the "plateauing" effect is delayed. The shape of the graph is dependent on how fast the hit rate degrades as a function of the number of threads. In general, it can be said that a high compute/memory ration decreases the need for caching and multi-threaded scaling.

In conclusion, there are two main philosophies of multithreaded programming. One side promotes a many-core architecture while the other promotes many-thread. While both of

these can offer optimized performance, it's important to understand the relationship with the cache that can lead to a performance "valley" somewhere between the two extremes. That is, the number of threads is not necessarily directly correlated with the performance of a system.

Details/Results

So far we have looked at how different factors influence cache performance which ultimately influences the performance of a single program. However, now that we are entering further into the age of supercomputers and virtual machines and cloud based computing, it's becoming more and more important to evaluate the cache performance when running multiple programs concurrently. This naturally relates to GPU processes but multi-application execution in GPUs is not nearly as well explored as multi-application execution on a CPU. GPUs are a natural addition to computer systems due to their ability to accelerate applications with many arithmetic operations in a data level of parallelization. Notably, a specific type of GPU, known as a general-purpose graphics processing unit (GPGPU) is becoming increasingly common to use to run a compute kernel. These are often used in certain applications that require a large number of vector operations, as they can yield several orders of magnitude higher performance than a conventional CPU. GPGPUs are generally suited to high-throughput type computations that exhibit data-level parallelism to exploit the wide vector width SIMD architecture of the GPU. Because modern GPUs have so many cores compared to CPUs, they are able to perform ALU type instructions very quickly; however, potential speedup becomes more complicated when multiple programs are running at the same time. Since GPUs rely on a shared cache, if two programs are running at the same time, one might keep making changes to the cache that results in a miss for the other process. If this cycle continues often, it could result in poor speedup times. This is known as multi-application interference, and the exact relation between cache parameters and performance is not known. When multiple applications are scheduled at the same time on GPU hardware, they can interfere at various levels of the memory hierarchy, such as interconnect, cache, and memory. However, it's not sufficient to compare two programs running concurrently solely based on their cache miss rates. The bandwidth usage of applications must also be taken into account to fully understand how one program might interfere with another. In cases where both programs have a high bandwidth demand, it is

likely that significant drops in performance will be observed. To test this, we used to benchmarks running at the same time on the same GPU: Blackscholes (BLK) and breadth first search (BFS2). When each application was running individually, the overall cache miss rate was very low for both (less than 0.01). However, when running simultaneously, the performance of both applications degrade significantly because they do not receive the same share of bandwidth from when they were running stand-alone. Furthermore, DRAM bandwidth is also important when predicting the performance of simultaneous processing on GPUs. These are only a couple of the parameters that can influence the overall program performance. Other factors include the number of processing elements, cache size, number of threads, cycles per instruction, processor frequency, cache latency, memory latency, etc. The main problem that simultaneously executing applications face relates to data locality. If possible, similarly structured programs should be scheduled to run together so that the overall cache miss rate will be minimized. However, in reality this is not always feasible. More practical solutions to this problem involve invoking different schedules to run the tasks in the order that produces the best speedup.

Conclusion

In conclusion, optimized cache performance is essential to generating the most efficient code. It's clear that the added latencies of loading values from main memory make caches so important to a programs overall performance. However, there are many related factors that can affect cache performance and they require close attention to deliver the best performance. For one, we have seen how there's a certain valley affect, where an increased number of threads don't necessarily improve runtime on a multi-core machine. However, if you most past this "valley", eventually the performance continues to rise once more before it eventually plateaus. This is the basis of the many-core vs many-thread debate. Additionally, we have seen that running multiple programs side-by-side on a GPU can interfere with the cache performance of the other. This is often due to conflicting cache writes that decrease the overall hit rate. The lesson that should be taken away from this is that many other parameters, such as memory bandwidth and number of threads, need to be considered and not just cache performance alone.

References

1. Guz, Bolotin, Keidar, Kolodny, Mendelson, Weiser, *"Many-Coren vs. Many-Thread Machines: Stay Away From the Valley"*, IEEE Computer Society, Intel Corporation, May 2009.
2. Guz, Itzhak, Keidar, Kolodny, Mendelson, Weiser, *"Threads vs. Caches: Modeling the Behavior of Parallel Workloads"*, IEEE Computer Society, 2010.
3. Jog, Bolotin, Guz, Parker, Keckler, Kandemir, Das, *"Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications"*, ACM Society, March 2014.
4. Jog, Kayiran, Kesten, Pattnaik, Bolotin, Chatterjee, Keckler, Kandemir, Das, *"Anatomy of GPU Memory System for Multi-Application Execution"*, ACM Society, October 2015.
5. Beckmann, Sanchez, *"Cache Calculus: Modeling Caches Through Differential Equations"*, MIT-CSAIL, December 2015.
6. Nugteren, Van den Braak, Corporaal, Bal, *"A Detailed GPU Cache Model Based on Reuse Distance Theory"*, IEEE Computer Society, February 2014.
7. Rogers, O'Connor Aamodt, *"Cache-Conscious Wavefront Scheduling"*, MICRO-45, December 2012.