University of Minho
Department of Informatics

Master's Degree in Informatics Engineering
Computer Graphics - Integrated Project

Performance API for CUDA

A critical analysis

Ana Catarina Macedo
a54773@alunos.uminho.pt

Miguel Palhas
pg19808@alunos.uminho.pt

Pedro Costa
pg19830@alunos.uminho.pt

Braga, July 2012

Abstract

1 Introduction

Intro

Explain here that all numbers refer to the Fermi architecture, unless specified otherwise

2 Performance API

Performance API [ICL, 2012] (commonly refered to as PAPI) is a library developed by the Innovative Computing Laboratory from the University of Tennessee that allows programs to access counters and measurement instructions that operate on the hardware level.

Most modern processors provide a set of built-in counters that can be programatically used to keep track of information that may be helpful to profile the program. PAPI allows the programmer to use these counters, using the library's functions, and measure a set of events within a block of code.

The events that can be measured by the library are varied. The most common counters measure events like clock cycles, number of total instructions (from which some can measured separately, like additions, multiplications, divisions, integer or floating-point operations), number of hits and misses of each cache level, branch instructions, or stall cycles.

The actual set of counters that can be measured is architecture-specific. Because of this, PAPI provides tools that show what counters can be measured on a specific machine, and provides a more intuitive alias for each one, instead of the hexadecimal code provided by the manufacturer.

PAPI also attempts to deal with some of the limitations of hardware counters, like it's limited amount. For instance, an Intel Xeon cenas coisas has only X real hardware counters.

Substituir cenas coisas por um processador a sério, e X pelo nº de contadores

.

For this, PAPI provides multiplexing, where a set of counters issued by the programmer share a time slice of the hardware counter, and the final result is estimated based on the total results, and the percentage of time that each counter consumed. Of course, this doesn't allow counter results to be nearly as accurate as they would be if measured on separate runs, but given the measured code block is large enough to minimize multiplexing errors, should provide a good enough alternative for programs where the execution time difficults multiple executions to be made.

3 CUDA

Compute Unified Device Architecture, or CUDA™ is a parallel computing platform developed by NVidia®[1342]that enabled massive performance increases for highly data-parallel applications, by providing a programming model more suited to graphics processing units.

Contrary to a CPU, which focus on executing a small amount of threads quickly, by emplying methods like branch-prediction and superscalarity, a GPU follows a different filosophy, executing much more threads slowly. Even though each individual thread runs more slowly, that latency is hidden by the fact that many more threads are executing concurrenlty, and the hardware level scheduler can switch the context with only a few clock cycles.

Of course, since the programming model is completely different, a CPU thread isn't comparable in any way to a CUDA thread. For example, while a simple algorithm might consist in looping through a collection of elements, applying an operation to each one, in CUDA a more suitable model would be to launch one thread for each element, and let the hardware scheduler manage them.

3.1 Programming Model

A CUDA device (GPU) tipically consists on a group of Streaming Multiprocessors (SM's), each one containing up to 64 CUDA Cores [1] This Cores are the the elementary

[1]This refers to the CUDA architecture up until Fermi. The new Kepler architecture features 192 CUDA Cores in each new extended

processing unit of a GPU. Each one is capable of one simple sequential operation at a time (arithmetic operations for example). More complex operations, like trigonometric functions, and square roots are executed on four Special Function Units, which process them at a rate of one instruction per clock-cycle.

Each SM can only issue one instruction each clock cycle, which means that all CUDA Cores of a given SM must execute the same instruction (altough the indexing of each CUDA Thread allows them to operate on different data). Thus, threads are grouped into warps, 32 Threads each, which are executed at the same time on an SM. Two schedulers per SM allow near peak performance by selecting two instructions from two different warps to be issued every clock cycle, increasing concurrency.

Note that this greatly limits concurrency when introducing branches. If threads within the same warp diverge in a branch, the entire warp will be required to wait as much as it would have if all threads executed both branch statements. This can be optimized by the programmer, if the branch pattern can be recognized to group non divergent threads in the same warps

Besides warps, threads are also grouped into blocks, which in turn are grouped int a grid. A block is the main organizational unit of threads. A single block can contain up to 1024 threads, and is assigned to a SM at the begginning of a kernel. The SM will later organize those threads into warps.

A grid is nothing more than a collection of blocks, organized in a bidimensional space, where each block will be assigned to a specific SM to execute the kernel function.

3.1.1 Synchronization

Having thousands of threads executing concurrently raises the issues of synchronization already familar from other parallel models. Sometimes a barrier is required to allow other threads to compute results that the current thread depends on.

In CUDA, synchronization is implicitl within a single warp, since all threads of a warp will execute the same instruction. On a higher level, the CUDA primitive `__syncthreads()` is available to create a barrier for all threads within a block, allowing the entire block to reach the same point.

As for grid level synchronization, it is not possible within the GPU, since each SM will run their blocks independently, but it can be achieved by using different kernel calls, implicitly creating a synchronization point between kernels.

3.2 Profiling CUDA

Explain (no need for much detail) what can be profiled in a GPU,, how it works, what info can be extracted from the results, and how it differs from CPU profiling. Maybe add subsubsections?

Somewhere in here, talk about built-in profilers, like CUPTI

Streaming Multiprocessor (SMX)

Since CUDA 4.0, profiling is done mostly via the CUDA Profiling Tools Interface (CUPTI). These library provides four main API's: Activity, Callback, Event and Metric.

3.2.1 Activity API

The Activity API allows asynchronous recording of all CUDA activity, both on CPU and GPU. Each Activity type as an associated ata structures that is used to provide details on each individual activity.

3.2.2 Callback API

The CUPTI Callback API allows the programmer to register callbacks in it's own code, which will be invoked upon calling a CUDA runtime or driver function.

Not only is this useful when profiling, it might also be a good tool to automate a task that the programmer wants to execute each time a speficic event occurs (for example, create a custom log of all memory transactions between the CPU and the GPU).

3.2.3 Event API

This API provides accesses to counters on a CUDA-enabled device. Much like PAPI, although with some differences due to architectural aspects, this counters provide access to hardware level counters to measure specific events on the GPU.

The workflow of this API is straighforward, by simply defining a group of events to measured prior to the kernel invocation, and retrieving the results at the end.

Some care needs to be taken with this, as event registration is asynchronous, meaning it is the programmer's responsibility the relevant kernel code only starts to run after the registration, and not thread-safe, meaning that, if other threads in the application are also using the GPU, chances are they are concurrently executing kernels, which will interfere with the measurements.

The set of events that can be measured is dependant on the compute capability of the device used. Prior to compute capability 2.0, the event set was quite limited, allowing only measurements of cache hits, misses, number of instructions (optionally of a given type), divergent branches, and counters related to memory accesses.

There are also 8 generic purpose triggers that can be used by the programmer to measure specific programmer-defined events in the code.

As for Compute Capability 2.0 and greater, most of the events from prior versions are still compatible, but a lot more are added. These events are split over for domains:

`domain_a` For events related with cache misses/hits, conflicts. Also relates to texture cache;

`domain_b` Various read/write requests for each memory hierarchy level (cache, Texture, DRAM);

`domain_c` For global load/store operation counters. Multiple counters are provided for the different amount of bytes requested.

`domain_d` Instruction related counters, like number of branches, divergences, active warps/cycles, number of instructions and memory requests. 8 general purpose counters are also included in this domain

In addition to the event set, other differences distinguish CUDA devices with compute capability below 2.0. Perhaps the most important of them is the fact that on those devises, all events are counted for only one SM. Starting on compute capability 2.0, events from `domain_d` are counted for multiple (but no all) SM's. Therefore, to get the most consisted results, it is best to use a grid that has a number of blocks multiple of the total number of SM's in the device used.

3.2.4 Metric API

CUPTI's Metric API calculates metrics for each kernel, based on one or more event values. These metrics might be useful to get a higher level overview of the GPU usage and efficiency before having to look at the more raw information provided by the Event API.

Like the Event API, it evolved in version 2.0. Initial versions only provided a small set of metrics:

Branch efficiency Shows the overall ratio of non-divergent branches to total branches, effectively indicating wether or not branching operations are greatly hindering performance.

Load/Store efficiency Can be useful to assert if memory accesses are sufficiently coherent to take advantage of memory coalescense.

Memory Throughput For metrics indicating effective throughput of memory load/store operations

Compute Capability 2.0, in addition to the previous metrics, provides additional ones:

SM Efficiency Ratio of time at least one warp was active on a multiprocessor to total time

Occupancy Ratio of average active warps per active cycle to the maximum amount of warps supported

IPC Instructions per Cycle

Replay overhead The performance loss due to memory replays

Cache Hit Rate Hit rate of L1 cache for both global/local loads and stores

4 PAPI CUDA

Talk about PAPI's API for CUDA, via CUPTI

Se quiseres fala aqui daquilo que achas que está bem/mal com a forma de utilizar o PAPI

PAPI CUDA is an attempt to create a CUDA component to integrate with the PAPI API, allowing for the same simple workflow of the library to add and remove counters, retrieve values. This component is implemented based on CUPTI, and works as an abstraction layer to the Driver calls required when using raw CUPTI code.

In order to use this component, PAPI must be compiled with the appropriate options, linking with the CUDA and CUPTI directories

5 Testing

Explain the small tests used, to understand the usage of both CUPTI and PAPI-CUDA

5.1 Test Case

Talk here about the testing done on the pathtracer

Explain what the test case is, the tests performed, and the methodology (no need for appendices i think, unless this becomes too large)

6 Results

Results from profiling the test case. Not too extensive, since the interest here is in the analysis of the profiler, and not the actual results

7 Analysis

The real juice. Talk shit about stuff. A LOT

Can also serve as a conclusion

References

[ICL, 2012] ICL (2012). PAPI: Performance API. `http://icl.cs.utk.edu/papi/`.

[Nvidia, 2009] Nvidia (2009). Fermi Architecture.

[Nvidia, 2011] Nvidia (2011). CUDA Tools SDK: CUPTI User Guide.

[Patterson, 2009] Patterson, D. (2009). The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges. NVIDIA Whitepaper .