Review on Heterogeneous CPU-GPU Processors Optimisation Opportunities compared to Discrete GPU Computing Pipeline

January 19, 2016

Abstract

Emerging heterogeneous CPU-GPU processors brings unified memory spaces and cache coherence, thus eliminating memory copy overheads. Meanwhile, traditional GPU computing applications relying on discrete GPU architectures may benefit from different application-level optimizations to run efficiently on heterogeneous CPU-GPU architecture.

Understanding these optimizations is the key to leverage the power of these processors in new programming models. We can quantify inefficiencies and find optimization targets using modified GPU benchmarks simulation which account for memory copies removal.

Considerable inefficiencies in core and cache usage are outlined in traditional synchronous software pipelines, which leads to optimization opportunities for heterogeneous processors: more efficient cores load balancing, improved producer-consumer data sharing and caching.

I. Introduction

By the end of 2010, a lot of desktop computers and even embedded devices had multicore processors, traditionaly built to perform serial tasks. Multicore processing brought some challenges, with interesting applications in GPU development, built to do parallel operations on large data sets. Initially used for their ability to improve graphics performance by offloading specific tasks from the CPU to the GPU, the latter are becoming more and more attractive for general purpose computing, coinciding with customers appetite for rich visual experiences and advanced multitasking support.

The drive to improve performances with the constraints on power and scalability have led semiconductors and systems designers to look for the GPU vector processing capabilities and integrate them with a CPU scalar approach, still best suited for some necessary tasks, in a new unified heterogeneous processor architecture. Ground-breaking systems allow CPUs and GPUs to share a common virtual address space, but they don't suppress the cost of data movement between the separate chips memories at a physical level. Heterogeneous CPU-GPU processors on the other hand allow unified memory space and coherent communication using cache coherence between cores.

Major APIs are evolving to integrate optimizations and better support for these processors. Indeed, heterogeneous processors computations benefit from major optimizations linked to the architectural improvements. The purpose of the research article is to quantify the performance improvements in GPU applications running on heterogeneous processors and point at some application design optimization targets.

II. Hypothesis

A discrete GPU architecture requires a data transfer through a PCI-E link between the CPU and GPU memories, which is often a bottleneck due to its speed compared to the chips memories and it delays the consumer task execution. In fact, this design influences programmers to create programs in a serial way: one unit needs to wait for a computation to terminate followed by the output data transfer. Thanks to a unified physical memory, heterogeneous CPU-GPUs share memory without the cost of memory copies, which could lead to immediate run-time performance gains of the order of the eliminated time spent copying.

More optimizations can be achieved in discrete GPU architectures with a data independance analysis and explicit algorithm improvements to allow parallelism of producer and consumer tasks. With the innate data synchronisation in coherent memory on heterogeneous CPU-GPU architectures, these optimizations could streamline this process by leveraging in-memory synchronisation mechanisms.

Finally, chuncking the work leads on both architectures to an increased core usage. Nevertheless, the same amount of data has to be copied through the PCI-E link to the discrete GPU memory during the runtime of the program, and even small copies can lead to cache invalidation and thus performance losses. However, bringing closer the producer and consumer tasks, and migrate work to underutilized cores could improve heterogeneous processors cache usage.

III. Method

To explore different kind of optimization targets, a broad range of tests with different application-level structures are derived from open-source GPU computing benchmarks.

1. Design

These tests emphasize on different producer/consumer relationships. For instance, some of them work on graph-like data structures, prone to irregular memory access. Later on, work queues use to track available tasks, entail an irregular control flow. Last but not least, other algorithm relying on image or signal processing use more regular memory accesses and control flows.

Input sets were chosen to ensure that the benchmarks are made of a minimum number of instructions for both the CPU and GPU, with a sufficient execution time and enough memory footprint to ensure significant results.

Practically, heterogeneous systems are approximated with a "limited-copy" version of the algorithms by mirroring the CPU and GPU data. This version is measured up to the original "copy" version. Copies are removed using a combination of manual code modifications and automatic libraries.

2. Simulation and independant variables

These algorithms are simulated in a controlled environment by using a full-system emulation of discrete GPU and heterogeneous CPU-GPU architectures. Specific configuration (independant variables) are defined for the core, memory and components characteristics like the PCI-E copy engine to ensure similar component capabilities in both architecture. Consequently, result differences come theoretically only from application-level architectural design decisions.

A span of the execution flow called the "region of interest", during which measurements are taken, is used to characterize a common core among the variety of benchmark algorithms, right after the data initialization/loading and just before the memory copy to the GPU or the GPU kernels launching.

Nonetheless, interferring events can happen during this timespan and cause significant performance degradations. Indeed, the page faults and the memory address translation when accessing a virtual memory address are handled on the GPU side for discrete architectures. Unlike heterogeneous processors and this simulation, in which the memory address translation table has to be synchronized between the CPU and the GPU so they can share a common virtual address space. Thus, the GPU raises an interrupt on the CPU which performs memory page mapping on behalf of the GPU and returns the page.

3. Measurements and analysis

Measurements in the region of interest on both versions of the algorithms are taken. First of all, the memory footprint. A memory zone can be touched for different reasons: copied and shared with the GPU, exclusive to the GPU (temporary or intermediate data), shared between the GPU and the CPU, CPU exclusive, owned by the CPU and shared by copy, and finally only copied. It indicates the data purpose, as an example, copied-only memory could be avoided by on-demand page synchronisation, and is mostly eliminated in the limited-copy version.

Afterwards, percentage of memory accesses indicate which component (either CPU, GPU or copy engine) consumes more memory and could suggest issues with the cache, or the type of algorithm at stake. For instance, CPU exceptions from the GPU page handling cause increasing memory accesses from the CPU, same as issues with memory allocation alignment (due to the GPU computing library not handling memory pages anymore) which cause increase in memory accesses from the GPU.

In the end, the run-time percentage shows the components relative activity. For instance, the GPU or CPU exclusives activities, the overlapping CPU-GPU activity, the copy engine, etc. Comparisons with the base benchmarks could point out underutilized resources or increased usage.

IV. Results

The copy and limited copy benchmark versions, whithout further optimization apart from removing memory copies, are the baseline for further improvement estimates using analytical modelling concerning the pipeline organisation and the cache performance.

1. Memory copies

Most GPU computing applications are built for discrete GPU systems and thus, use explicit memory copies to share data between the CPU and the GPU.

Suppression of memory copies leads to huge improvements in the memory footprint. First and foremost, most of the benchmark's data is copied at some point. A justification is that on average, 70% of the data is used on the GPU, which explain the necessary sharing and such overhead. Moreover, some computations don't even touch their whole data set after a copy, with for instance algorithms that only traverse their linked data structure such as graphs. Memory can be saved by using smart pagination and asynchronous mechanisms like on-demand page synchronisation.

This analysis is confirmed by the memory accesses statistics: copy accesses account for more than 10% of the total memory accesses. Little cache efficiency is gained by simply removing memory copies, which explains that the CPU and GPU memory access proportions are similar for both versions.

Finally, removing memory copies results in a run-time improvements of about 10%, by the order of magnitude of the PCI-E engine copy time when exclusively handling the data. Improvement in the CPU execution time, by preventing some further developed cache invalidation by memory copies, are experienced. However, some benchmarks are affected by agressive serialized GPU interupts on the CPU, reducing the GPU performance.

2. Pipeline

Results indicate that most of the GPU computing application structures use serialized CPU and GPU operations, as few portion of the time plot overlapped CPU and GPU execution. Some performance improvements can be gained by overlapping the execution even more.

Common techniques include kernel fission and asynchronous streams to improve discrete GPU performance. Heterogeneous processors could use easier in memory synchronisation mechanisms to realize the same transformation. In practice, a 3 % gain can originate from masking kernel launch latency and control code. Most of the differences between copy and limited-copy are eliminated this way. Regular data structure benchmarks benefits from these optimizations as kernel fission and asynchronous copy are trivial in this case to implement. Moreover, CPU controlled GPU kernel launching could signal events during the kernel execution and overlap control code.

Besides, the overall trend demonstrate poor balance between CPU and GPU work. The natural answer is to shift some work to underutilized cores, but in this case, it is necessary to address the drawbacks of migrating data, which remains difficult for discrete GPU. Assuming the work is shifted automatically, moved computation cannot exceed FLOP rate and memory bandwidth. There are plausible improvements, mostly for heterogeneous CPU-GPU processors which don't require memory copy, that could reach 30% run-time improvements.

3. Cache

Memory copies elimination cuts cache invalidation and heterogeneous processors even have additional advantages with a shared cache. It's use is not optimized for bulk-synchronized pipelines and some memory accesses cannot benefit from this cache anyway. For instance, long-range data accesses, which occur repeatedly accross all the computation with too long timespans.

Inefficient data reuse from one stage to the next can provoke repeated memory access called “spills”. Write then read accesses suggest a producer-consumer relationship, while read then read accesses suggest that the same input data is used for consecutive stages. Cache spills can occur if the inter-stage data produced or consummed exceed the cache capacity. Eliminating these cache spills result in significant 2% CPU performance increase by overlapping CPU and GPU execution in heterogeneous processors.

Finally, cache contentions occur when a computation stage accesses a memory footprint so large that data is evicted from the cache before it can even be used. Usally these are read-read contention, with a special case of writeback-read contention. It doesn't only deeply degrade performances, but also lead to excessive use of memory bandwidth and more inefficiencies. In most benchmarks, contention accounts for 60% of memory accesses, which means the data set for a given computing stage greatly exceed available cache memory.

V. Implications

One key item is to move the consumer and producer tasks in closer temporal proximity. Methods are being develop to help these transformations, but they are often complex. Moreover, they don't prevent from encountering resources run out such as bandwidth limitation which could lead to negative effects like cache spills. Asynchronous kernel launching is under work in GPU programming APIs, aiming for heterogeneous processors, but seems to introduce an important overhead.

A research path could focus on increasing the programmer's control over the cache and help him identify key data for concurrency, to reduce manipulated sets in a computing pipeline and avoid cache contention for example.

It seems plausible that new applications targeting heterogeneous processors will focus on lightweight dependency handling using in-memory synchronisation primitives, and focus on better shared cache usage.

VI. Conclusions

The comparison of GPU computing application optimization opportunities for discrete GPU and heterogeneous CPU-GPU processors leads to results that indicates same potential gains from parallelism and core utilization. Heterogeneous processors will likely provide easier programming models thanks to the shared memory and thus in-memory synchronisation mechanisms.

A key difference is the cache efficiency. Traditional bulk-synchronous GPU programming models lead to inefficient cache usage, and even resource limitation issues. Heterogeneous processors will likely provides better performances compared to discrete GPUs by leveraging easier work migration and load-balancing, flexible shared memory data handling and smart caching strategies, which in practice could lead to up to 70% performance improvements over the stock programming models.