Titel der Seminararbeit

Seminararbeit
André Merboldt

Lehrstuhl für Hochleistungsrechnen, IT Center,
RWTH Aachen, Seffenter Weg 23,
52074 Aachen, Germany
Betreuer: Dipl.-Inf. Vorname Nachname

This paper discusses the basics of the CUDA Runtime and Development process. It will furthermore introduce the reader into the CUDA hardware architecture and memory management. We will then explore some memory transfer and access optimization techniques. Additionally, synchronization is briefly covered. This paper is done as a pro seminar at the High Performance Computing at the RWTH Aachen in 2015.

Keywords: *CUDA, optimization, synchronization, memory, unified memory*

1 Introduction

CUDA is an API by NVIDIA to utilize GPUs for accelerated computation. This Framework is currently in its 7th version and we will discuss it at this point in time. GPUs have a different approach to computation, as they have far more cores and are therefore able to process code more concurrently. To utlize this potential we have to pass data efficiently to the GPU and process it in a parallelized manner. We will focus in this paper on the memory transfer and access optimization. First, we have to understand the hardware architecture to recognize the possible limitations and bottlenecks. After that, we will discuss some memory transfer optimization and then two memory access patterns.

2 Hardware Architecture

2.1 Overview

In GPGPU (general purpose graphical processing unit): some definitions:
Host: typically the CPU which executes most of the serialized code and can access the GPU
Device: the GPU itself, there can be multiple GPUs
textbfKernel: procedure which is executed on the device and called from the Host.
Streaming Processor (SP) each GPU contains multiple SP which are (mostly) independent processors which can execute code. **Thread**: Single Instance of method invocation
Block: Threads are grouped into Blocks
Warps: a collection of 32 threads which are guaranteed to run on one streaming processor.
Memory: We differentiate between Host and Device memory, which are both divided into finer units which are discussed later.

2.2 Memory

2.2.1 Host

There are multiple types of Host memory. But we highlight only the RAM and CPU Cache.

2.2.2 Device

each GPU consists of global memory (usually 1236712389 Bytes) which is accessable to all threads and blocks and shared memory (usually 64(???) kB) which is only accessable to threads within the same block. Additionally there are L2 Cache and Register Memory. Global memory slow, shared memory fast, reg + l2 ultra fast

2.2.3 Transfer

As GPUs are usually(? exclusively?) connected to hosts via PCI the bandwidth is limited by the protocol and hardware. PCI-Express (? explaint

Express?) has several iterations with different theoretical bandwidth limitations. (? make table) source: pcisig.com ***show GPU Global memory transfer rates*** Device Memory » PCIe Transfer

As we can easily see, the GPU memory is multiple times faster than the PCIe protocol theoretical maximum bandwidth.
We will exploit this knowledge to achieve (?) high computational throughput (?).

2.3 Kernel

A *Kernel* is a procedure (of code?) which runs on a GPU and is called from the Host. One invocation of a kernel is organized in a thread, which are grouped into warps, which is a logical unit of (always ?) 32 threads. Threads can further be put into one block. The programmer has the choice to call a kernel multiple times in a block (? bad explanation...) and into multiple parallel blocks.

3 Memory Management

3.1 Memory Allocation

We need to differentiate the allocation of host and device memory. While we can allocate host memory using malloc, but not exclusively, device memory is generally allocated using cudaMalloc. *cudaMalloc()* can allocate a linear memory in the device memory, given the device has enough free memory. Important to note is that *cudaMalloc()* is always blocking, as an alternative can cudaMallocAsync be used which uses streams (...?).

As we will see in the fifth chapter 4.3, it is important to access memory correctly and to simplify this, we can use *cudaMallocPitch* and *cudaMalloc3D()* to allocate 2D and 3D memory respectively.

Allocation and Deallocation are expensive operations and as a consequence we should reuse memory whereever it is possible. (maybe show timings?)

3.2 Streams and Synchronization

3.3 Unified Virtual Addressing

3.4 Unified Memory

4 Memory Transfer Optimization

4.1 Pinned Memory

One of the most important aspects of optimization is the memory transfer from device and host ("und umgekehrt"/ and in reverse?). As we have seen in "Hardware Spec" (figure pci), PCIe 2.0 has a theoretical bandwidth of 8 GB/s. Unfortunely, this theoretical bandwidth is limited by several factors, most impactful the processor on the Host (? citation needed). To bypass this limitation it is possible to *pin host memory* to the device.
This means that we have direct memory access (DMA) to this memory (RAM). The important bit of this technique is, that this pinned memory is page-locked, so it can't be swapped (or even accessed??) by the host system. As this method locks the CPU out of the equation (through DMA), we can achieve a bandwidth of 6 GB/s (in comparision to... GB/s with page-able memory) The drawback of this method is the impact on the host system, as it can lower the systems performance (because it blocks /lowers host memory).

4.2 Zero-Copy

Zero-Copy is a feature introduced in CUDA 4.0 (?) and is particulary useful if data is accessed once or if the GPU is integrated. *cudaHostAlloc* allocates pinned host memory, which is mapped into device address space. Using this features enables programmers to access host memory without copying it to the device memory. This is in several scenarios useful, especially when the GPU is integrated, in which case the GPU has no device anyways and uses host memory. Another use for Zero-Copy is accessing complex structures only once, because it often tedious to create deep copies (for examples of linked lists or trees). We use this technique to enable data transfer concurrency without the use of streams. (...!)

4.3 Memory Access Optimization

4.4 Global Memory Access

While the access latency and bandwidth to global memory on a device is much better in comparision to the transfer (PCIe 2.0), it can still be limited (? word choice!) by the programmer. To understand the limiting factors, we have to understand the memory access method (?) used by CUDA.
CUDA automatically combines memory access transactions (?) from threads in the same warp into one transaction, provided they access adjacent memory locations. While this is mostly beneficial to the programmer, it can happen to stall or limit the memory access heavily.
As we can see in Fig9.8 (CUDA programms: A dev's guide to parallel programming), ...

Generation	Transfer-Rate	per Lane	16-lane
PCIe 1.0	2.5 GT/s	2 GBit/s	32 Gbit/s
PCIe 2.0	5.0 GT/s	4 GBit/s	64 GBit/s
PCIe 3.0	8.0 GT/s	7.87 GBit/s	126 Gbit/s
PCIe 4.0	16.0 GT/s	15.754 GBit/s	252 GBit/s

Table 1: Comparison of different iterations of the PCI Express protocol

4.5 Memory Access Patterns Optimizations

While we can access global memory more efficiently, often it is more important that threads in the same block can communicate fast (? block??). Fast Communication and synchronization is possible with shared memory, which is quite limited in its size (max 64 kbytes?), it is much faster than the global memory.
Similarly to global memory it can happen, that unfortune memory access patterns stall the computation. Shared memory is separated into multiple, so called *banks*, of 32 Bits or 4 Bytes.
Bank conflicts happen, when multiple threads in the same warp access the same bank, resulting into serialized memory access. Therefore (?) it is desirable to use memory access patterns where different threads access different banks to avoid this.

5 Conclusions

The discussed methods and techniques have shown how to transfer and access different kinds of memory efficiently. Because we can compute and access data way faster on the device than transfer it to it, it is often better/more efficient to do some calculations/computations on the device even if they don't get processed faster, but to avoid memory transfer. This results in more GPU code and therefore higher complexity. (?)