Introduction to Massively Parallel Computing (with CUDA)

September 30, 2016

Outline

What is CUDA?

Heterogeneous Architecture

CPU/GPU Design Considerations

Single Instruction Multiple Data

Reprise

Hello World

What is CUDA?

CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing - an approach known as GPGPU [computing].[1]

[1] Wikipedia

What is CUDA?

CUDA is a parallel computing platform and programming model that makes using a GPU for general purpose computing simple and elegant. The developer still programs in the familiar C, C++, Fortran, or an ever expanding list of supported languages, and incorporates extensions of these languages in the form of a few basic keywords.[2]

[2] blogs.nvidia.com

Heterogeneous Architecture

- A GPU is not a standalone platform but rather a co-processor to the CPU
- GPUs must operate in conjunction with CPU-based host through a PCI-Express bus
- In GPU computing terms, the CPU is called the *host* and the GPU is called the *device*
- Heterogeneous applications include both
  - Host code (C, C++) which runs on the CPU
  - Device code (C-lang *Kernels*) which runs on the GPU
- The CPU code is responsible for managing the environment, code, and data for the device before loading compute-intensive tasks on the device

Heterogeneous Architecture

- Since the GPU is physically separate from the CPU, it can most generically be referred to as a *hardware accelerator*
- The general idea is that GPUs are used only to accelerate portions of application code which exhibit rich amounts of data parallelism (i.e. linear algebra, image proc, FFTs, etc)

CPU/GPU Design Considerations

- GPU computing is not meant to replace CPU computing. Each approach has advantages for certain kinds of applications.
- Multicore microprocessors, like Intel Core i7, are optimized for for sequential instruction execution performance. These multicore CPU designs make use of sophisticated control logic and very large shared caches to allow instructions from a single thread of execution to execute in parallel or even out of sequence while maintaining the appearance of sequential execution
- On the other hand, the massively parallel streaming processor design of GPUs utilize many threads of execution over huge number of smaller cores. For example, GeForce TitanX has 3072 cores (24 SM $\times$ 128 cores/SM).

CPU/GPU Design Considerations

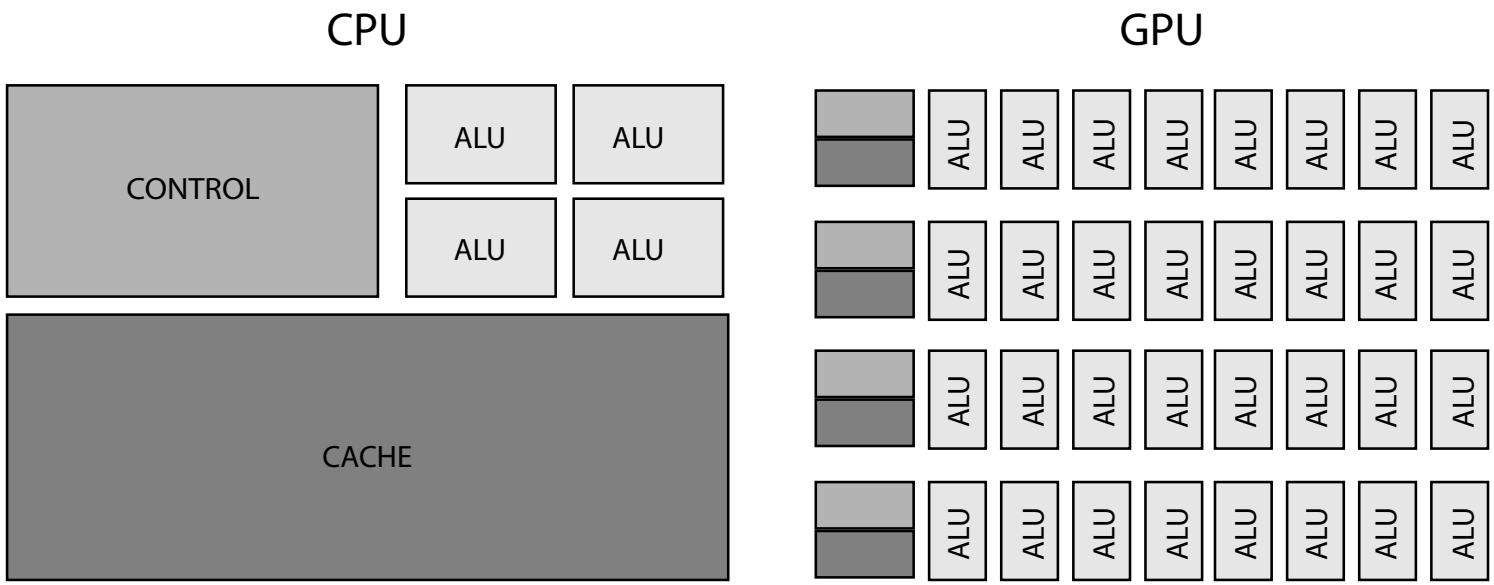

Figure: CPUs and GPUs have fundamentally different design philosophies

CPU/GPU Design Considerations

- The design philosophy of the GPU was historically shaped by the fast-growing video game industry that exerted tremendous economic pressure for the ability to perform massive number of floating-point calculations per video frame.
- This demand motivated GPU architects to look for ways to maximize the chip area and power budget dedicated to floating-point arithmetic.
- The prevailing solution is to optimize for the execution throughput of massive numbers of threads by allowing piplined memory channels and arithmetic ops to have long latency.
- This design minimizes chip area and power of both memory access hardware and ALUs allowing for more of each on chip thus increasing total execution throughput.

CPU/GPU Design Considerations

- This GPU chip design style is commonly referred to as *throughput-oriented* design since it strives to maximize the total execution throughput of a large number of threads.
- The CPU chip design, on the other hand, strives to minimize the execution latency of a single thread and is often referred to as *latency-oriented* design.
- Think of these design contrasts as the difference between local versus global optimization strategies.

CPU/GPU Design Considerations

- OK, I get it . . . but why is throughput-oriented effective on GPUs?
- For a typical CPU scenario, each thread of execution is a different application (e.g. Microsoft Office, Netflix, Angry Birds). That is, multiple concurrent threads executing independent applications.
- However, on a GPU all threads are working in parallel to execute a single kernel/task (e.g. multiply these two matrices, white balance this image, etc).
- Therefore, the throughput-oriented kernel execution says something like "*I don't really care about individual instruction execution times when I multiply my two matrices together, just make the whole thing as fast as possible*."

Massively Parallel Algorithms in SIMD

- Right, so, GPU hardware is designed for maximizing thread execution throughput
- And, this throughput-oriented design is effective since all the GPU threads are working a single kernel in parallel
- But, how do all those GPU threads know what to do?
- Each GPU thread execute the same kernel. Once each individual thread has finished sequentially executing the kernel code, the kernel task as a whole is complete.
- That is, each thread executes the same kernel but on it's own chunk or piece of the data. This style of programing is called *single instruction multiple data* (SIMD) pronounced "sim-dee"

Massively Parallel Algorithms in SIMD

- In SIMD each thread executes the same instructions (i.e. kernel code) but only on it's own small piece of the data.
- Begs the question: how do threads know what data to use?
- Each thread must have index information to provide context.
- Threads in SIMD use there own index info as a key to determine which data to access.
- This means that in massively parallel SIMD algorithms, thread organization plays a very important role!
- Kernel thread organization should closely mirror the data dimensions so that thread index is aligned with data index.

Massively Parallel Algorithms in SIMD

- The organization of threads executing a kernel will be organized in a way that mirrors how the data is organized.
- For example, kernel threads operating on a data vector of size $1 \times N$ would be organized as a "block" of threads with dimensions 1 by N. Here each thread would have index one through N which would loosely map to the item(s) in the array to work on.
- Likewise, kernel threads operating on a data matrix of size $N \times M$ would be organized as a "block" of threads with dimensions N by M where each thread would have a row and a column index/context information associated with it.

What is CUDA?

- **C**ompute **U**nified **D**evice **A**rchitecture (CUDA)
- The NVIDIA device driver allows CPU hardware to interoperate with general purpose GPU hardware accelerators.
- CUDA is a C language library (including a small set C-language extensions) written on top of the device driver.
- To leverage the throughput-oriented design of GPU devices, tasks are programed as CUDA kernels (i.e. device code) written in SIMD programing paradigm.
- CUDA provides convenient way to specify how threads should be organized for execution of a particular kernel (i.e. thread organization to mirror data dimensions)
- Today, CUDA does a lot more and has many fancy features for all different types of applications and problem domains.

CUDA First Contact


```
1  #include<stdio.h>
2
3  // GPU Kernel definition
4  __global__ void sayHello(void){
5      printf("Hello World from the GPU!\n");
6  }
7
8  int main(void){
9
10     //launch kernel
11     sayHello<<<1,5>>>();
12
13     //wait for the kernel to finish
14     cudaDeviceSynchronize();
15
16     //that's all
17     return 0;
18 }
```

Listing 1: Hello world in CUDA