Multi-core Programming
Assignment 4

Ali Gholami

Department of Computer Engineering & Information Technology
Amirkabir University of Technology

https://aligholamee.github.io
aligholami7596@gmail.com

Abstract

Keywords.

1 Introduction

Each system has different parts. Some parts of a system could be one or more CPU, GPU, RAM, HDD and SSD. These parts come together differently as generations develop in hardware architecture. This integration confines the performance of a system. The following questions will point out some of the considerations we take into account while designing these kind of systems. Please refer to the book *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs* for more information on these questions. Explain and justify your answers.

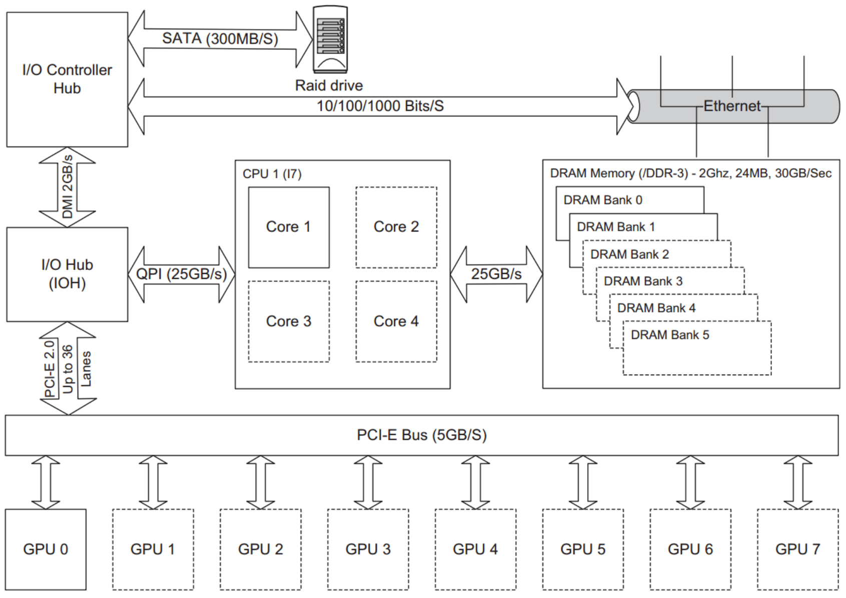

Figure 1.1: Nehalem/X58 System

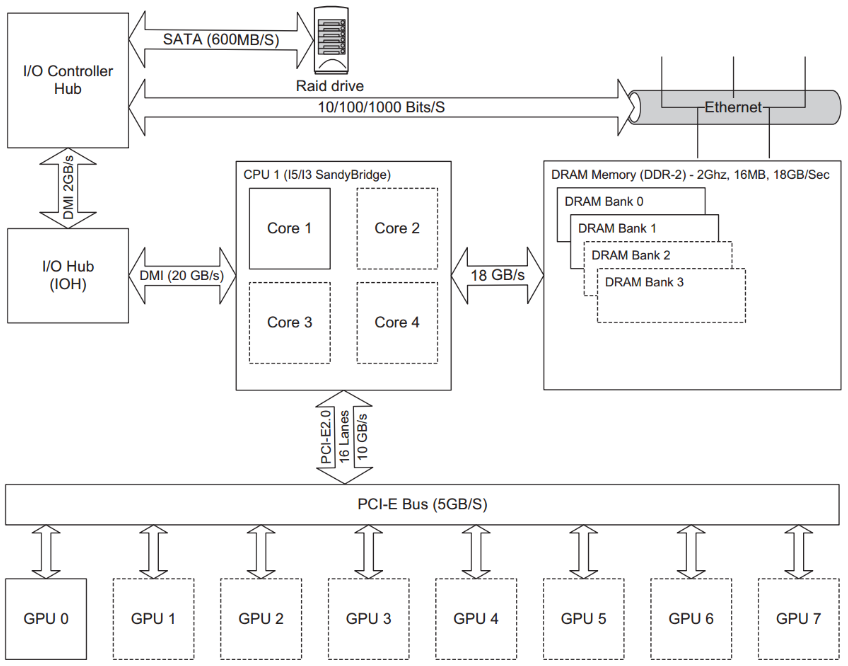

Figure 1.2: Sandybridge Design

- **What is the difference between peripheral connection in these architectures?**

  One of the most noticeable improvements of Sandybridge was the support for the SATA-3 standard, which supports 600 MB/s transfer rates. This, combined with SSDs, allows for considerable input/output (I/O) performance with loading and saving data. While in the Nehalem/X58 architecture peripheral transfer rate is only 300 MB/s. One significant advantage of the AMD chipsets over the Intel ones is the support for up to six SATA (Serial ATA) 6 GB/s ports. SATA3 can very quickly overload the bandwidth of Southbridge when using multiple SSDs (solid state drives). A PCI-E bus solution may be a better one, but it obviously requires additional costs.

- **Does peripheral connection speed matters? Explain.**

  If you consider that the slowest component in any system usually limits the overall throughput, this is something that needs some consideration. For example, SATA3 can very quickly overload the bandwidth of Southbridge when using multiple SSDs (solid state drives). As another fact, If using MPI (Message Passing Interface), which is commonly used in clusters, the latency for this arrangement can be considerable if the Ethernet connections are attached to the Southbridge instead of the PCI-E bus. Consequently, dedicated high-speed interconnects like InfiniBand or 10 Gigabit Ethernet cards are often used on the PCI-E bus.

- **What kind of port is used to connect to the GPU? What are the attributes of this port?**

  PCI-E is used. PCI-E (Peripheral Communications Interconnect Express) is an interesting bus as, unlike its predecessor, PCI (Peripheral Component Interconnect), it's based on guaranteed bandwidth. In the old PCI system each component could use the full bandwidth of the bus, but only one device at a time. Thus, the more cards you added, the less available bandwidth each card would receive. PCI-E solved this problem by the introduction of PCI-E lanes. These are high-speed serial links that can be combined together to form X1, X2, X4, X8, or X16 links. Most GPUs now use at least the PCI-E 2.0, X16 specification, as shown in Figure 3.1. With this setup, we have a 5 GB/s full-duplex bus, meaning we get the same upload and download speed, at the same time. Thus, we can transfer 5 GB/ s to the card, while at the same time receiving 5 GB/s from the card. However, this does not mean we can transfer 10 GB/s to the card if we're not receiving any data (i.e., the bandwidth is not cumulative).

- **According to the given architectures in figure 1.1 and figure 1.2, how many GPUs can be connected in each system? What about their bandwidth? Explain.** In both cases, 8 GPUs can be connected to the system via PCI-E bus. however, the big downside of socket 1155 Sandybridge design: It supports only 16 PCI-E lanes, limiting the PCI-E bandwidth to 16 GB/s theoretical, 10 GB/s actual bandwidth. Thus each GPU is able to use $\frac{10}{8} = 1.25$ GB/s of bandwidth available. On the other hand, the *Nehalem/X58* provides $\frac{16}{8} = 2$ GB/s of bandwidth.

2 GPU Architecture

Main components of a GPU are given below. Describe each of these components and the relationship between them.

- **Memory (Global, Constant, Shared)**: CUDA C makes available a region of memory that we call **shared** memory. This region of memory brings along with it another extension to the C language akin to __device__ and __global__. As a programmer, we can modify our variable declarations with the CUDA C keyword __shared__ to make this variable resident in shared memory. The CUDA C compiler treats variables in shared memory differently than typical variables. It creates a copy of the variable for each block that you launch on the GPU. Every thread in that block shares the memory, but threads cannot see or modify the copy of this variable that is seen within other blocks. This provides an excellent means by which threads within a block can communicate and collaborate on computations. Furthermore, shared memory buffers reside physically on the GPU as opposed to residing in off-chip DRAM. Because of this, the latency to access shared memory tends to be far lower than typical buffers, making shared memory effective as a per-block. However, if your program is using too much shared memory to store data, or your threads simply need to share too much data at once, then it is possible that the shared memory is not big enough to accommodate all the data that needs to be shared among the threads. In such a situation, threads always have the option of writing to and reading from **global** memory. Global memory is much slower than accessing shared memory; however, global memory is much larger. For most video cards sold today, there is at least 128MB of memory the GPU can access. The CUDA language makes available another kind of memory known as constant memory. As the name may indicate, we use constant memory for data that will not change over the course of a kernel execution. NVIDIA hardware provides 64KB of **constant** memory that it treats differently than it treats standard global memory. In some situations, using constant memory rather than global memory will reduce the required memory bandwidth.

- **Streaming Multiprocessors & Streaming Processors**: A GPU is an array of SMs. Each of these SMs have N cores. There are number of key components making up each SM, however, not all are shown here for reasons of simplicity. The most significant part is that there are multiple SPs in each SM. Note that Each SM has a separate bus into the shared memory, constant memory, and global memory spaces.

- **What is GDDR memory? Explain the differences with a DDR memory? Why not using a DDR in graphics card?** Global memory is supplied via GDDR (Graphic Double Data Rate) on the graphics card. This is a high-performance version of DDR (Double Data Rate) memory. Memory bus width can be up to 512 bits wide, giving a bandwidth of 5 to 10 times more than found on CPUs, up to 190 GB/s with the Fermi hardware. Thus, the main difference between DDR and GDDR is that the *Bandwidth* on GDDR is much higher than DDR memories. That's mainly because of the high number of cores accessing the memory simultaneously.

- **Explain the inner components of figure 2.1.** The very first component is the Global Mem-

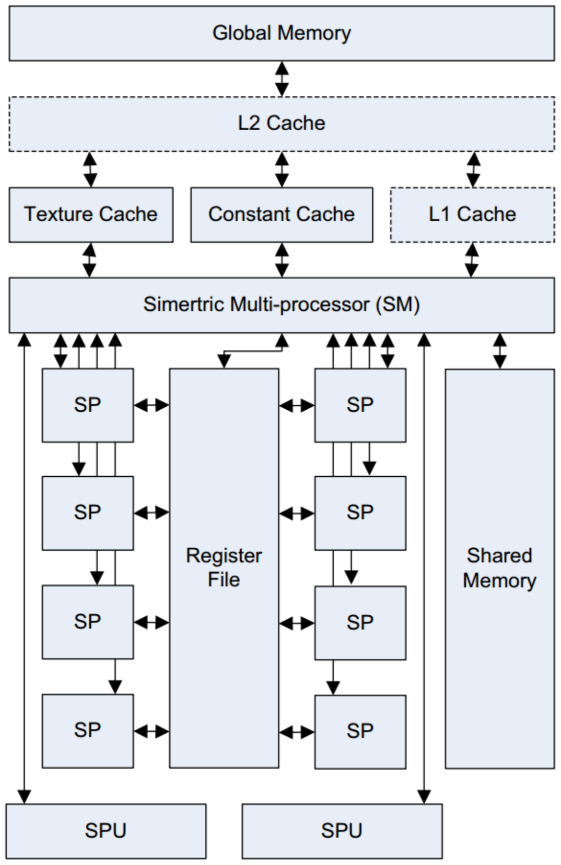

Figure 2.1: Inside an SM

ory which is called GDDR in graphics processing unit. There are also 2 levels of cache in a GPU. The first layer cache is divided into three types called *Texture Cache, Constant Cache and L1 Cache*. Each GPU Consists of multiple SMs that they contain some Streaming Processors (A.K.A Cores). Each of these SPs have their own registers in a huge register file. Each SM also has two or more special-purpose units (SPUs), which perform special hardware instructions, such as the high-speed 24-bit sin/cosine/exponent operations. Double-precision units are also present on GT200 and Fermi hardware.

- **What are the differences between *Texture, Global and Constant* memories?** Texture memory is a special view onto the global memory, which is useful for data where there is interpolation, for example, with 2D or 3D lookup tables. It has a special feature of hardware-based interpolation. Constant memory is used for read-only data and is cached on all hardware revisions. Like texture memory, constant memory is simply a view into the main global memory. Global memory is supplied via GDDR (Graphic Double Data Rate) on the graphics card. This is a high-performance version of DDR (Double Data Rate) memory. Memory bus width can be up to 512 bits wide, giving a bandwidth of 5 to 10 times more than found on CPUs, up to 190 GB/s with the Fermi hardware.

3 Compute Capability

The *compute capability* of a device is represented by a version number, also sometimes called its ”SM version”. This version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU. The compute capability comprises a major revision number X and a minor revision number Y and is denoted by X.Y. Devices with the same major revision number are of the same core architecture. The major revision number is 5 for devices based on the Maxwell architecture, 3 for devices based on the Kepler architecture, 2 for devices based on the Fermi architecture, and 1 for devices based on the Tesla architecture. The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features.

4 Occupancy in CUDA

Represents that how much of the threads in an streaming multiprocessor are occupied during a process. It can be computed using the following formula:

$$OCC = \frac{Threads\ in\ SM}{Max\ Threads\ in\ SM} \tag{4.1}$$

Occupancy subjects to various constraints:

- Completed blocks assigned to each SM.
- If combined register usage exceeds SM limits then we'll have to limit the number of blocks in an SM.

- If combined shared memory usage exceeds SM limits then we'll have to limit the number of blocks in an SM.

5 Vector Addition Indexing

Suppose we want to add two vectors together. Find the proper index for the output vector in case we want each thread to compute a single component of the vector.

$$i = blockIdx.x * blockDim.x + threadIdx.x;$$

6 Vector Addition Grid

Suppose two vectors with 8000 elements. We want each thread to compute one of these elements and the block size of the threads is 1024. Programmer adjusts the *Kernel Launch* so that the minimum blocks of threads are used. Find the number of threads in a grid in this case.

There will be obviously 8192 threads in the grid while 192 of them are doing nothing.