INTRO

GPUs and GPGPU

- Made in 1990's and early 2000's to deal with demand for highly realistic video games and video animations.
- Are designed to improve performance of programs that render many detailed images
- People soon started to look into using the power of GPU's to solve general purpose computing problems
    - Things such as searching and sorting, rather than graphics.
    - This became known as GPGPU
        - General purpose graphics processing unit
- Several API's were created for the purpose of GPGPU's
    - The most used are CUDA and OpenCL
        - CUDA was developed for NVIDIA GPU's only
            - We will be focusing on CUDA since its much easier to implement
        - OpenCL was developed for all GPU's
            - To ensure this portability, an OpenCL program must include a good deal of code providing information about which systems it can be run on and information about how it should be run

GPU architectures

- We often think of CPU's as being SISD (Flynns taxonomy):
    - the processor fetches an instruction from memory and executes the instruction on a small number of data items
- GPU's are composed of SIMD (Single instruction Multiple data stream) processors
    - SIMD processor as being composed of a single control unit and multiple datapaths.
    - The control unit fetches an instruction from memory and broadcasts it to the datapaths.
        - Each datapath either executes the instruction on its data or is idle.
    - Example of SIMD

For example, suppose there are n datapaths that share an n-element array `x`. Also suppose that the ith datapath will work with `x[i]`. Now suppose we want to add 1 to the nonnegative elements of `x` and subtract 2 from the negative elements of `x`. We might implement this with the following code:

```
/* Datapath i executes the following code */
if (x[i] >= 0)
   x[i] += 1;
else
   x[i] -= 2;
```

In a typical SIMD system, each datapath carries out the test `x[i] >= 0`. Then the datapaths for which the test is true execute `x[i] += 1`, while those for which `x[i] < 0` are *idle*. Then the roles of the datapaths are reversed: those for which `x[i] >= 0` are idle while the other datapaths execute `x[i] -= 2`. See Table 6.1.

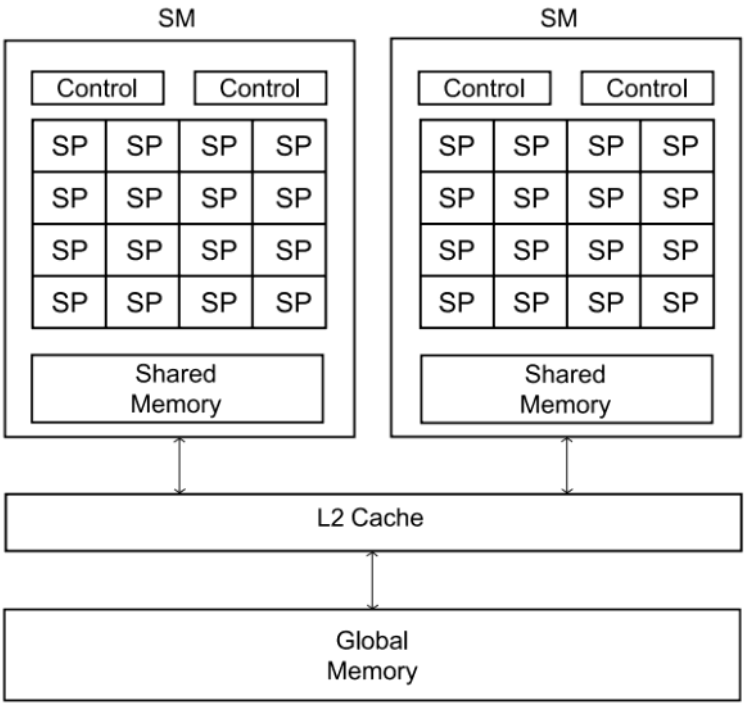
Time	Datapaths with `x[i] >= 0`	Datapaths with `x[i] < 0`
1	Test `x[i] >= 0`	Test `x[i] >= 0`
2	`x[i] += 1`	Idle
3	Idle	`x[i] -= 2`

- A GPU can be thought of as being composed of many SIMD processors (each processor lives in an SM)
    - Nvidia GPUs are composed of Streaming Multiprocessors (SMs)
        - One SM can have several control units and many more datapaths
            - an SM can be thought of as consisting of one or more SIMD processors
        - The SMs, however, operate asynchronously: there is no penalty if one branch of an if—else executes on one SM, and the other executes on another SM.
    - So in our preceding example, if all the threads with x[i] >= 0 were executing on one SM, and all the threads with x[i]

Time	Datapaths with `x[i] >= 0` (on SM A)	Datapaths with `x[i] < 0` (on SM B)
1	Test `x[i] >= 0`	Test `x[i] >= 0`
2	`x[i] += 1`	`x[i] -= 2`

        - Only two stages required here
- In Nvidia language, we will refer to cores as SP (streaming processor)
    - Note, one SM can have 120 SP's
- Nvidia uses SIMT (single input multiple thread) instead of SIMD.
    - the term is used because threads on an SM that are executing the same instruction may not execute simultaneously: to hide memory access latency, some threads may block while memory is accessed and other threads, that have already accessed the data, may proceed with execution.
- Each SM has a relatively small block of memory that is shared among its SPs.
    - This memory can be accessed quickly by its SP's
- All of the SM's on a single chip also have access to a much larger memory block that is shared among all the SP's.
    - Accessing this larger block of memory is slow
- The CPU and GPU are usually physically seperated alongside its memories.
    - The CPU and its memory is often called the host
    - The GPU and its memory is often called the device
    - Because of this separation an explicit function would be needed to transfer data from the GPU to CPU or vice versa
        - This isnt often the case anymore
- What a GPU might look like

-



Heterogeneous computing

- Up this point we have assumed that our programs are running on processors which have similar architectures.
- This is not the case now, program that runs on a GPU is an example of heterogeneous computing.
    - The reason is that the program makes use of both the host (CPU) and device (GPU), both have different architectures
- We will still write a single program but we will write functions for the CPU and functions we write for the GPU
    - In a way we are writing two programs
- CPU and GPU