# Parallel Processing

## Unit 1



Dr. Ahmed Sallam

# Understanding Parallel Environment

# QUIZ

What are 3 traditional ways HW Designers make computers run faster?

- ☑ Faster Clocks
- ☐ Longer Clock Period
- ☑ More Work per Clock Cycle
- ☐ Larger Hard Disk
- ☑ More Processors
- ☐ Reduce amount of memory

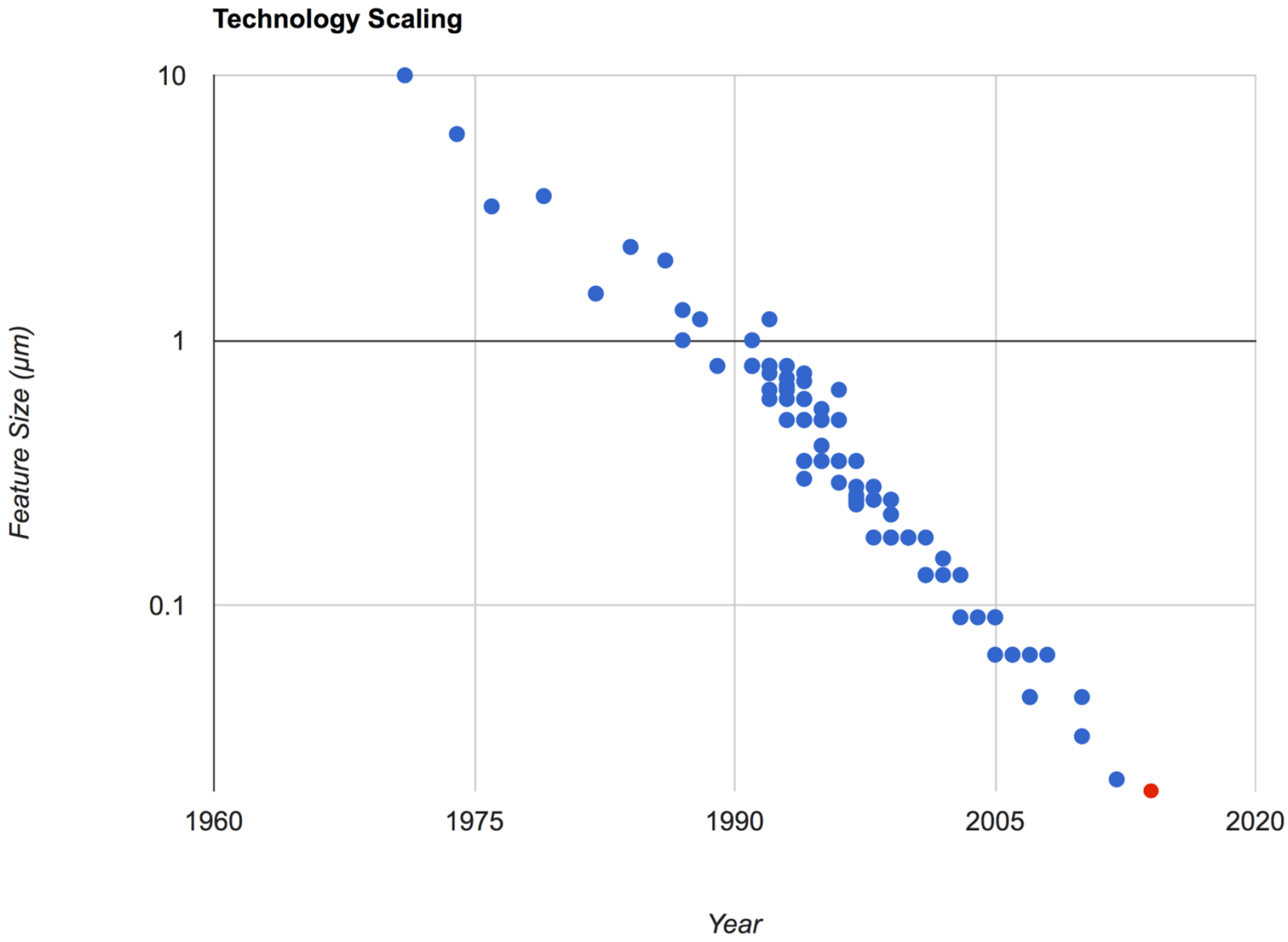
# Seymour Cray (Super Computer Designer)

- If you are plowing a field, which would you rather use?
  - Two strong oxen.
  - 1024 chickens

# Parallel Computing

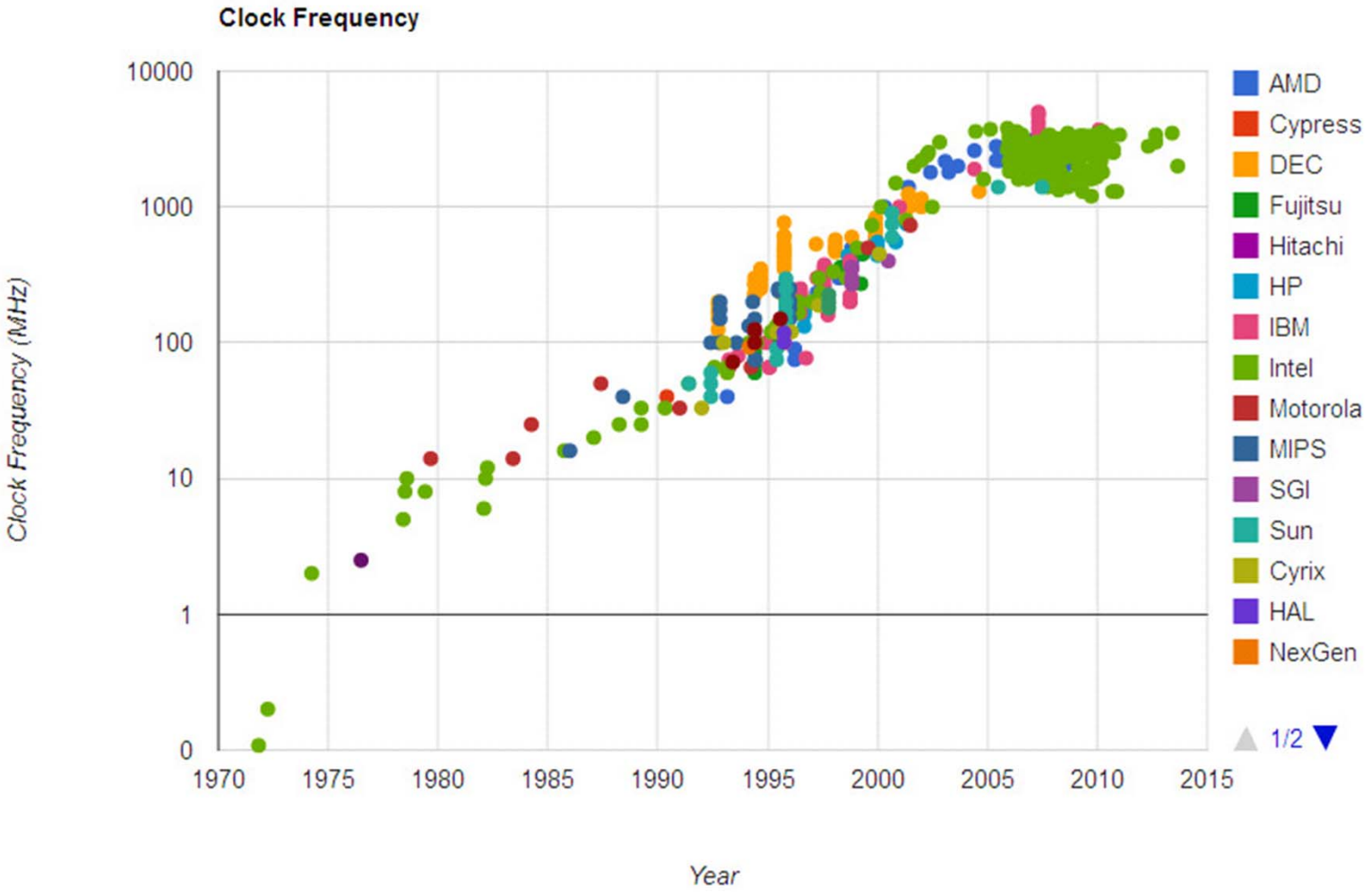
- It was intended to be used by super computing.
- Now all computers/mobiles are using parallel computing.
- Modern GPUs
  - Hundred of processors
  - Thousand of ALUs (3,000)
  - Ten or thousands of concurrent threads.
- This requires a different way of programming than a single scalar processor
- General purpose programmability over GPU (GPGPU.)

# Transistors continue on Moore's




Source: Stanford CPUDB

# Clock Speed (no more speed)



# QUIZ

- Are processing today getting faster Because

- ❑ We are clocking their transistors faster
- ☑ We have more transistors available for computation.

- Why don't we keep increasing clock speed of a single processor instead of multiprocessors with a less clock speed?
  - No, we can't because of power (heat)

# WHAT KIND OF PROCESSORS WILL WE BUILD?

- Assume major design constraint is **Power**

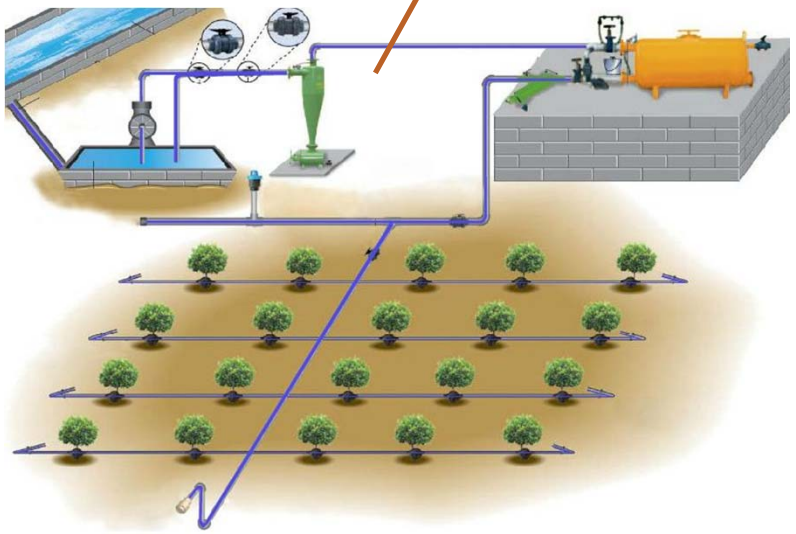
- Why are traditional CPU-like processors are not the most energy efficient processors?
  - It has complex control hardware
  - This increase flexibility and performance
  - And increase power consumption and design complexity as well
- How to increase power efficiency (GPU-like)?
  - Build simple control structure.
  - Take those transistors and devote them to support more computation on the data path
  - The challenge becomes how to program?

# More to understand

# More to understand (Cont.)

More speed with complex structure

Less speed with simple structure



Power

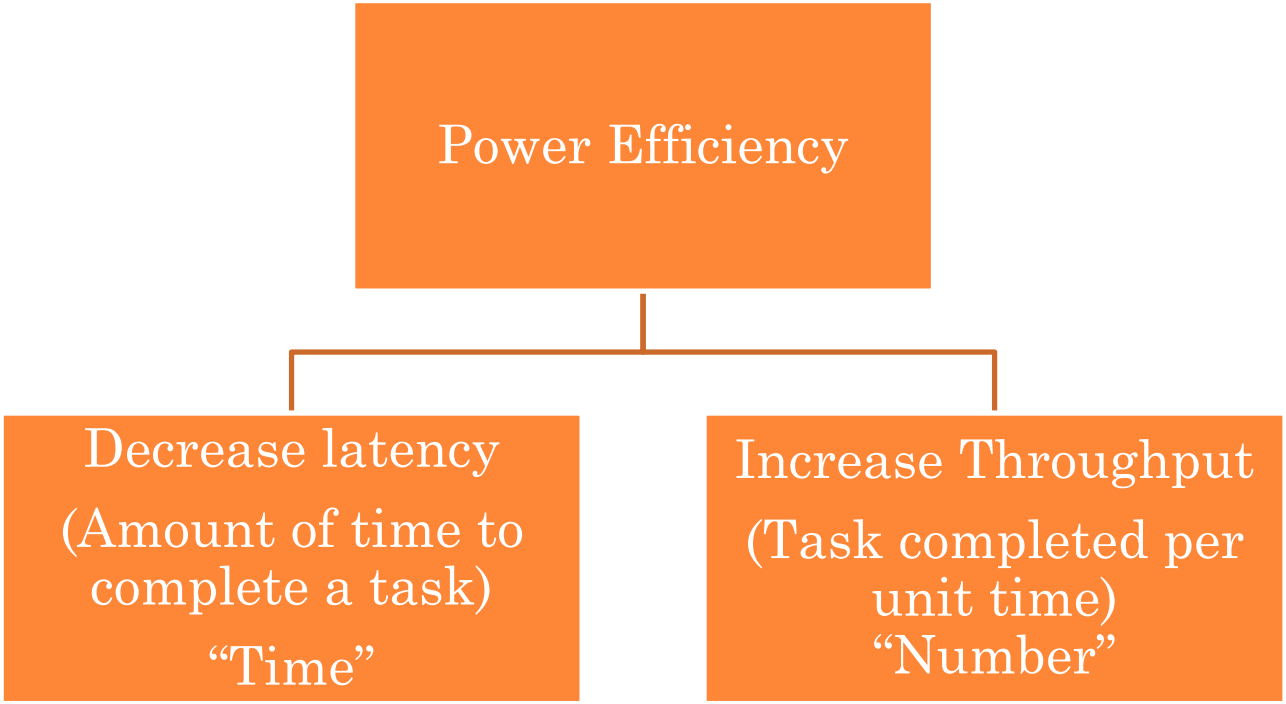
Less Power

# QUIZ

- Which techniques are computer designer using today to build more power-efficient chips?

- Fewer, more complex processors
- ✓ More, Simpler processors
- Maximizing the speed of the processor clock
- Increasing the complexity of the control hardware

# Another factor for power Efficiency




- The two goals are not aligned
  - CPU-like: design to decrease latency
  - GPU-like: design to increase throughput
- The choice depends on the **application** (Image processing prefer to increase the throughput)

# SUPER QUIZ

- Why do I say GPU-like and not saying Multi-core CPU? Is there a deference ?!
  - They both build for parallel programming. However, Multi-core CPUs can be used for sequential and parallel programming as well (provides branches and interrupts ). On the other hand GPU build for parallel programming from scratch.

# GPU DESIGN BELIEVES

- Lots of simple compute units
- Explicitly parallel programming model
  - We know there are many processors and we didn't depend on the complier for example to parallel the task for us.
- Optimized for throughput not latency

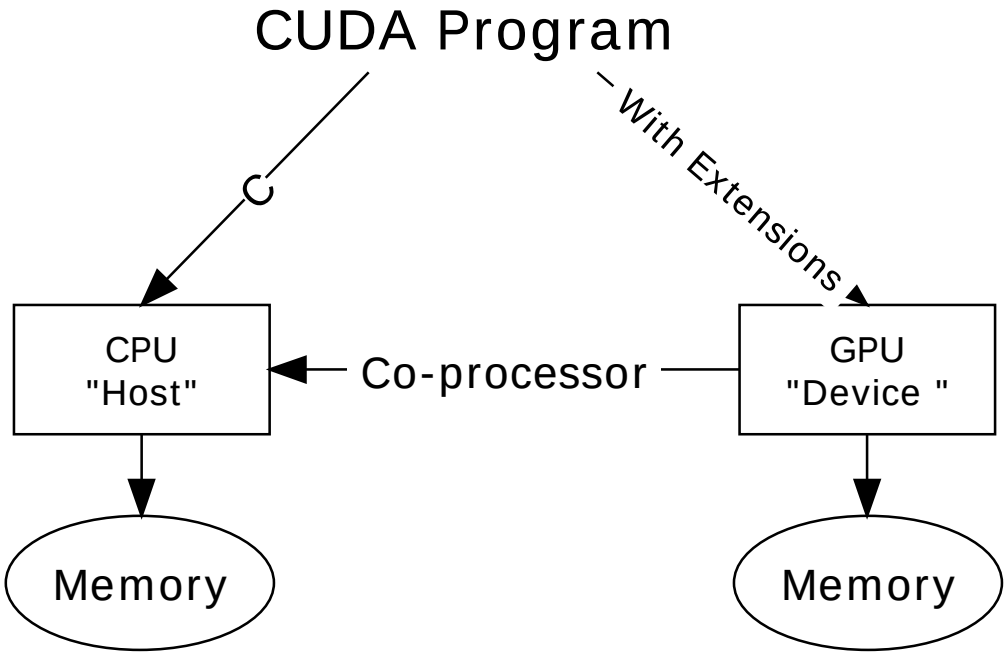
# Intro to Parallel Programming

# Importance of Parallel Programming

- Intel 8 core Ivy bridge
- 8-wide AVX vector operations/core
- 2 threads core (hyper threading)

- This means the processor has 128 way of parallelism
- Parallel programming is more complex however Running sequential C program means using less than 1% of this processor power

# CUDA PLATFORM




- CUDA compiler generate two separated program one for CPU (Host) and another for GPU (Device).
- CPU in charge and control the GPU
  - Moves data between memories (cudaMemcpy)
  - Allocates memory on GPU (cudaMalloc)
  - Invokes programs (kernels) on the GPU: ”Host lunches kernels on the Device”

# QUIZ

The GPU can do the following:

- Initiate data send from GPU to CPU
- ✓ Respond to CPU request to send data from GPU to CPU
- Initiate data request from CPU to GPU
- ✓ Respond to CPU request to receive data from CPU to GPU
- ✓ Compute a kernel lunched by CPU
- Compute a kernel lunched by GPU ←

# Typical GPU Program

- CPU allocate storage on GPU
- CPU copy input data from CPU to GPU
- CPU lunches the kernels on the GPU to process the data
- CPU copies results back to the CPU from the GPU

- If you need to move data many times between CPU and GPU, CUDA is not good for your program because it takes many steps to do so as showing above

# Main Issue

- Defining the GPU computation
  - Write a  Kernel like serial program
  - When lunching the kernel  tell the GPU how many threads to lunch

# QUIZ

What is the GPU good at?

- Lunching  a small number of threads efficiently
- ✓ Lunching a large number of threads efficiently
- Running one thread very Quickly
- Respond to CPU request to receive data from CPU to GPU
- Running one thread that does lots of work  in parallel
- ✓ Running a large number of threads in parallel

# GPU Power

- Example:
  - In : [1, 2, 3, …., 64]
  - Out: [$0^2$, $1^2$, $2^2$, …., $64^2$]

- Sequential solution:

```
for(int i=0;i<64;i++)
    Out[i]=in[i]*in[i];
```

  - here we have 1 thread do 64 multiplications each takes 2 ns.

# GPU Power (cont.)

- Example:
  - In : [1, 2, 3, …., 64]
  - Out: [$0^2$, $1^2$, $2^2$, …., $64^2$]

| CPU | GPU |
| --- | --- |
| • Allocate memory<br>• Copy data to/from GPU<br>• launch kernel | out= in * in |

- Parallel solution: j
  - CPU code: square kernel <<<64>>>(out, in)
  - here we have 64 thread each do 1 multiplication which take 10 ns.

# Example

start

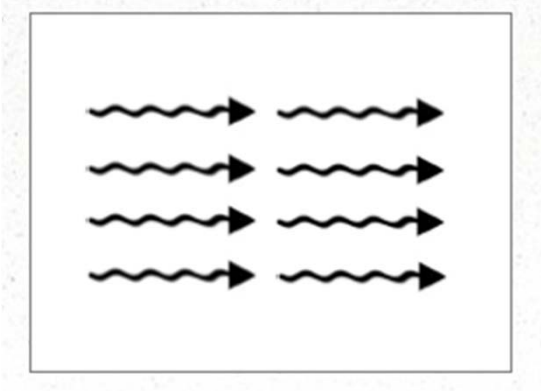
# Threads and Blocks

# THREADS

A single execution units that run kernels on the GPU. Similar to CPU threads but there's usually many more of them. They are sometimes drawn as arrows

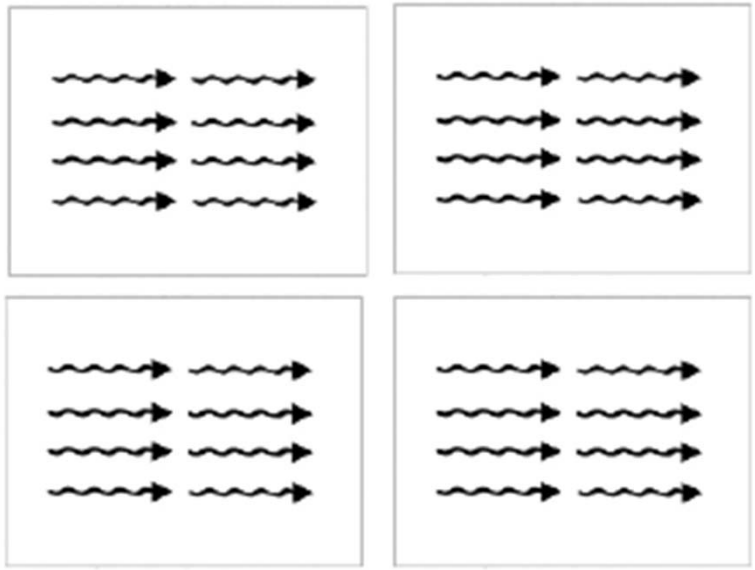
# BLOCKS

- Thread blocks are a virtual collection of threads.
- All the threads in any single thread block can communicate



# GRID

- A kernel is launched as a collection of thread blocks called the grid.



# Maximums

- You can launch up to 1024 threads per block (or 512 if your card is compute capability 1.3 or less).
- You can launch $2^{32}$-1 blocks in a single launch(or $2^{16}$-1 if your card is compute capability 2.0 or less).
- So my relatively inexpensive GeForce GT 440 can launch a rather ridiculous 67,108,864 threads.

# WHY BLOCKS AND THREADS?

- You may be wondering why not just say "launch 67 million threads" instead of organizing them into blocks.
- Suppose you wrote a program for  a GPU can which can run 2000 threads concurrently. Then you want to execute the same code on a higher GPU with 6000 threads. Are you going to change the whole code fore each GPU?
- Each GPU has a limit on the number of threads per block but (almost) no limit on the number of blocks. Each GPU can run some number of blocks concurrently, executing some number of threads simultaneously.
- By adding the extra level of abstraction, higher performance GPU's can simply run more blocks concurrently and chew through the workload quicker with absolutely no change to the code.
- nVidia has done this to allow automatic performance gains when your code is run on different higher performance GPU's.

# DIM3

# DIM3 DATA TYPE

- Dim3 is a 3d structure or vector type with three integers, x, y and z. You can initialize as many of the three coordinates as you like:
  - dim3 threads(256); // Initialize with x as 256, y and z // will  both be 1
  - dim3 blocks(100, 100); // Initialize x and y, z will be 1
- dim3 anotherOne(10, 54, 32); // Initialises all three values, x
  - // will be 10, y gets 54 and z
  - // will be the 32.

# THREAD ACCESS PARAMETERS

- Each of the running threads is individual, they know the following:

- threadIdx ← Thread index within the block
- blockIdx ← Block index within the grid
- blockDim ← Number of threads in the block
- gridDim ← Number of blocks in the grid

- Each of these are dim3 structures and can be read in the kernel to assign particular workloads to any thread.

# THREAD ACCESS PATTERN

- Its common to have threads calculate a unique id within the kernel to process some specific data. If we launch a kernel with:

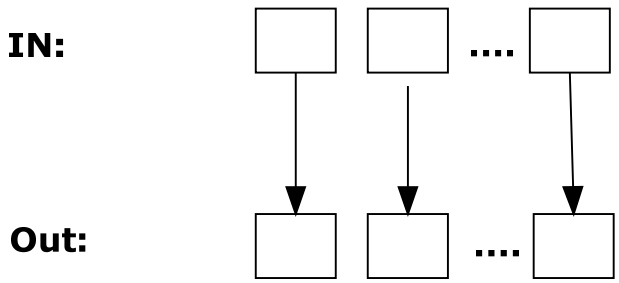
- SomeKernel<<<100, 25>>>(...);
- Inside the kernel, each thread can calculate a unique id with:
  - int id = blockIdx.x * blockDim.x + threadIdx.x;
- So the 5th thread of the 4th block would calculate:
  - int id = 4 * 25 + 5 = 105
- The 14th thread of the 76th block would calculate:
  - int id = 76 * 25 + 14 = 1914

# Mapping

# Map

- Set of elements to process [64 floats]
- Function to run on each element [square]

Map(element, function)



# QUIZ

Which programs can be solved using Map

- Sort an input array
- ✓ Add one to each element of an input array
- Sum up all elements of an input array
- Compute the average of and input array