GPU Computing with CUDA lecture 2

A component of Scalable Computing module Semester 2, 2013

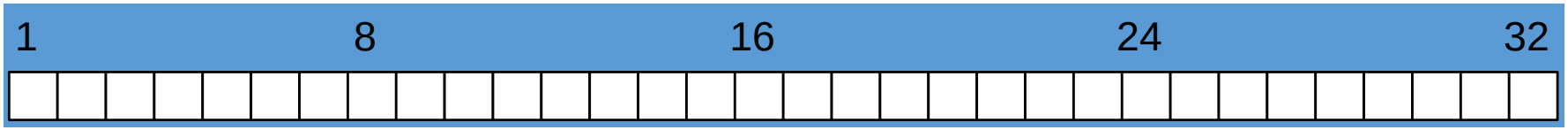
John O'Rourke
X00070416@ittd.ie

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

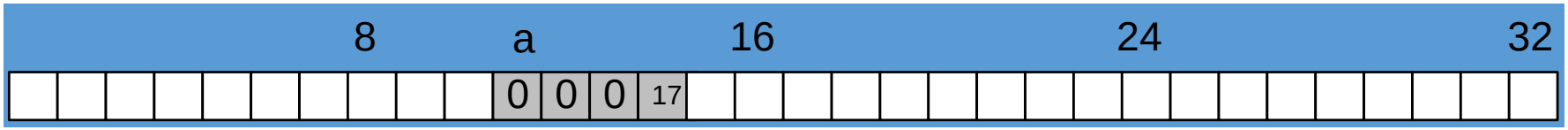
Memory and C data types

High level languages (e.g. Java, C#) hide the job of memory allocation and management from the programmer. Lower level languages (e.g. C/C++) allow us to modify memory directly!

Logically, a computer's memory (DRAM) can be thought of as a continuous set of boxes, each representing one byte and each byte with it's own address.



When we declare a variable (e.g. `int a = 17`), it is placed into memory at a particular address and will occupy a certain number of contiguous bytes depending on the data type. For instance, most compilers treat an integer as 4 bytes (32 bits) :



In this case, the integer `a` is placed at address 11. It occupies 4 bytes. `a` gives us the value of a (=17) and `&a` gives us the address of a (=11).

C Pointers

Sometimes in C we will wish to declare a variable which holds the address of a type (e.g. the address of some integer) rather than a value. These variables are called pointers.

```
int *pa;            // declare a variable "pa" which points to an integer

int a = 17;         // declare an integer a
pa = &a;            // assign the address of a to our pointer
```

One of the most common reasons for using pointers in C is for passing data structures (especially arrays) to functions without having to make a copy of the array in the function. We pass the address of the data to the function and it can manipulate the data at that address.

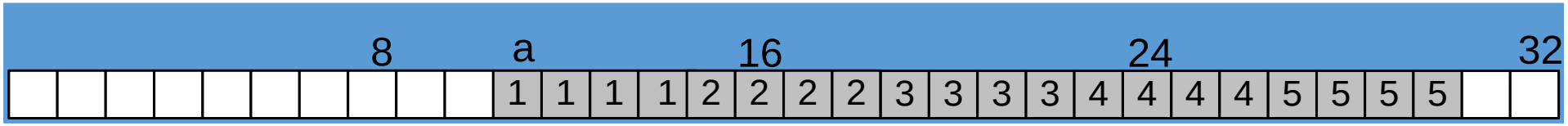
```
myFunction(int *a);   //Declare a function which takes a pointer to a integer
int ages={50,23,… }   //Declare an integer array
```

An array name on it's own (without an index) is really a pointer to the first element of the array.

```
myFunction(ages);     //Call the function, passing in a pointer to the first element
```

C Pointers, Arrays and Arithmetic

Arrays in C are declared (e.g. `int a[5]`). This creates an array of 5 integers, occupying 20 bytes of memory. The variable `a` (as with all array variables in C) is in fact a pointer to the first element of the array. The array index notation (e.g. `a[3]`) returns the value found at address a + 3 integers (= a + 12 bytes) which is the 4th element (remember array indexes are zero based; `a[0]` is the first element).



So why do we bother giving a pointer a type if it always just a memory address?
e.g. `int *a, float *b`

The reason is we can perform arithmetic on the pointers!
If we add 1 to an integer pointer, the compiler will increment the pointer value (i.e. the address) by 4 bytes. That way we can traverse any array by only knowing the address of its first element and the element type.

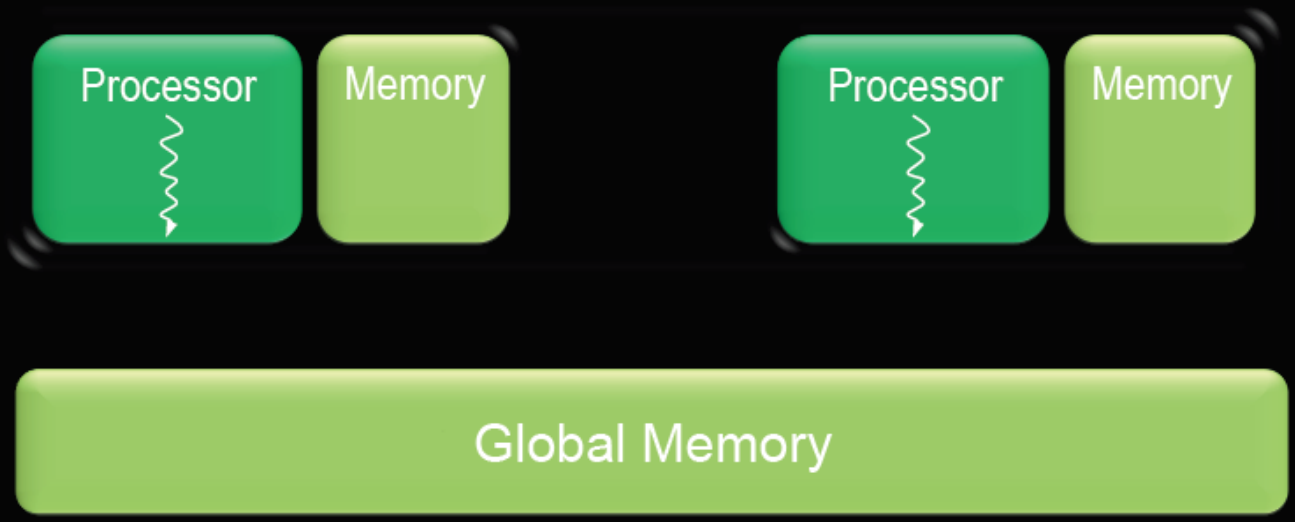
```
int *a;
```

`a` = The address of the first element…

`a + 4` = The address of the 5th element (should be equal to the address in a + 16)

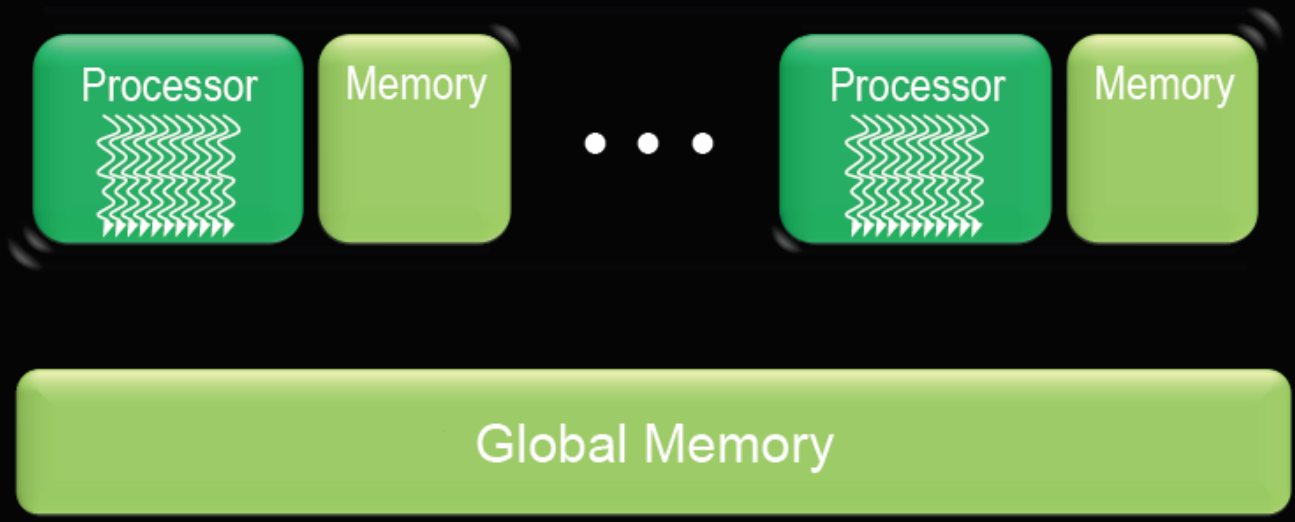
Multicore v Manycore …

Generic Multicore Chip



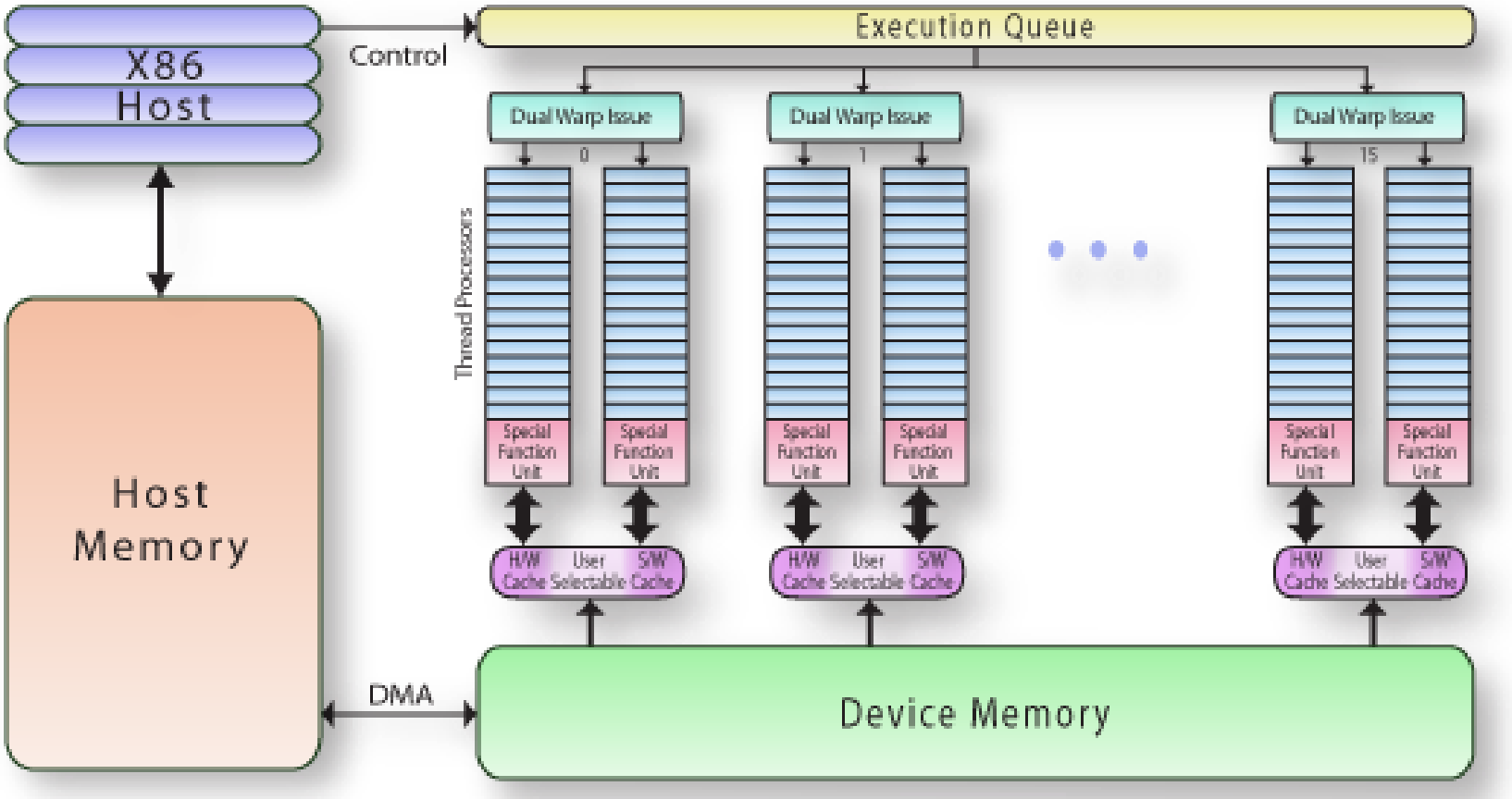
- Handful of processors each supporting ~1 hardware thread
- On-chip memory near processors (cache, RAM, or both)
- Shared global memory space (external DRAM)

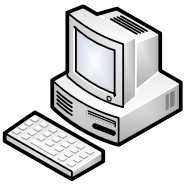
Generic Manycore Chip



- Many processors each supporting many hardware threads
- On-chip memory near processors (cache, RAM, or both)
- Shared global memory space (external DRAM)

Recap 1…

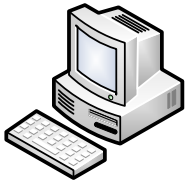

X86
Host
Control
Execution Queue
Dual Warp Issue
Dual Warp Issue
Dual Warp Issue
0
1
15
Thread Processors
Special Function Unit
H/W Cache
User Selectable
S/W Cache
Host Memory
DMA
Device Memory



I wish to send you some floating point data to process, so please reserve `n` bytes of your memory! I will need to know the start address of the memory you allocate; you can assign that to my local pointer variable `gpu_intp`.

```
int *gpu_intp;
cudaMalloc((void**)&gpu_intp, n);
```

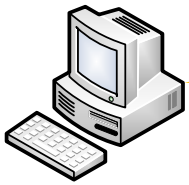
OK, that memory is now reserved and I have put the starting address of it into the pointer you sent me (`gpu_intp`).



Now please copy the data from my memory to yours! You can find the data starting at address `a` in my memory, you can store it at address `gpu_intp` in your memory and it is `n` bytes in size.

```
cudaMemcpy(gpu_intp, a, n, cudaMemcpyHostToDevice)
```

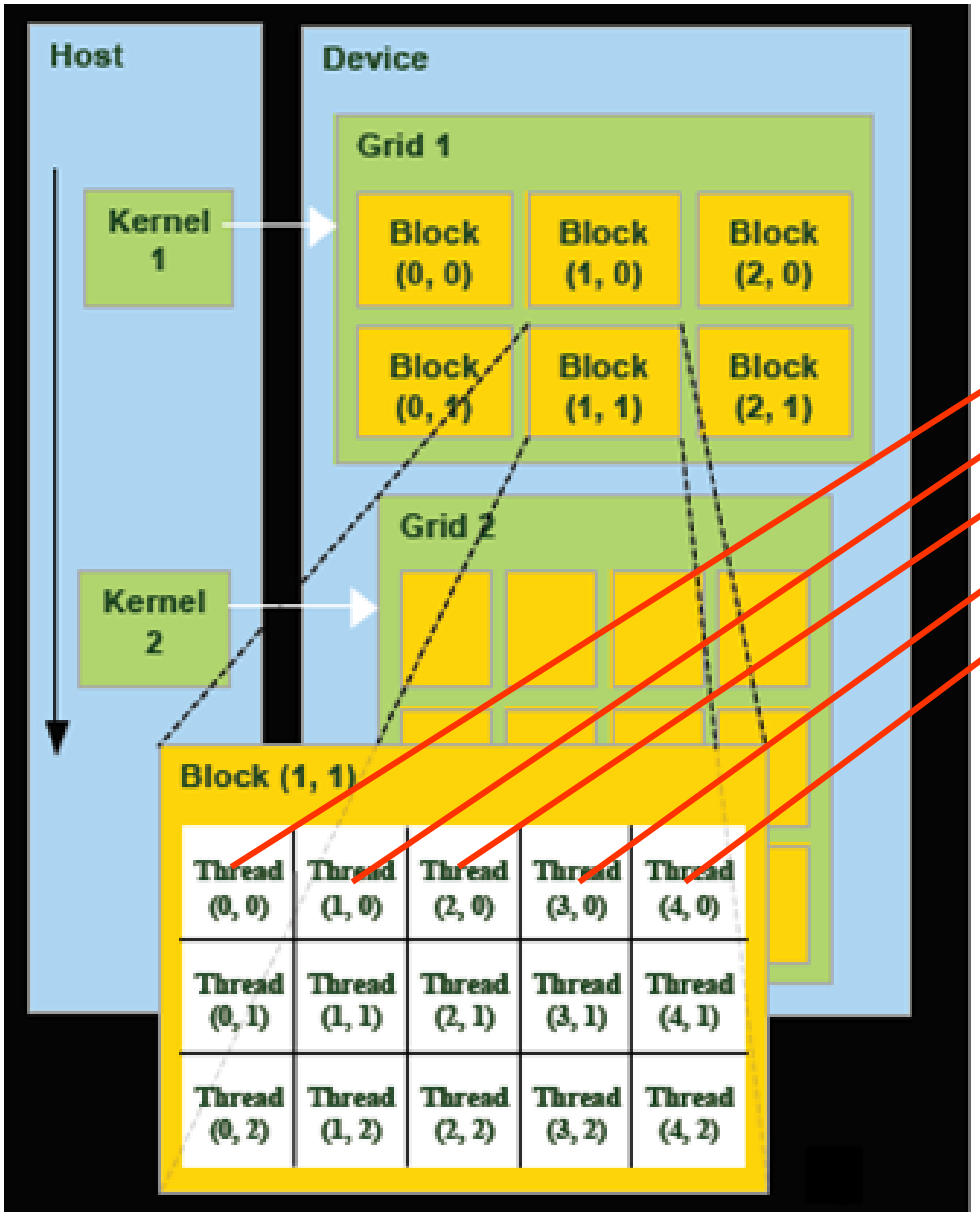
Done! I now have that data stored, starting at address `gpu_intp`.

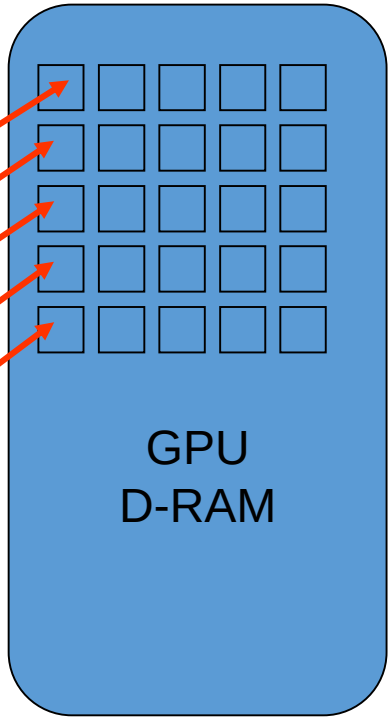


OK, now process the data! Launch the kernel called `MyKernel`. The kernel threads will need to know where all the data on the GPU is , so I'm passing the starting address of it into the kernel and also the amount of data (in bytes) you should expect to find there.

```
MyKernel<<<dimG,dimB>>>(gpu_intp, n);
```

Recap 2…





Declare kernel:

___global___ MyKernel(int *a);

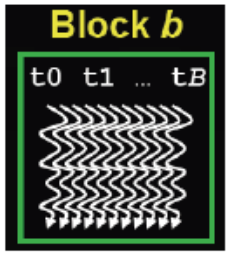
Call kernel:

MyKernel<<<DimG, DimB>>>(a)

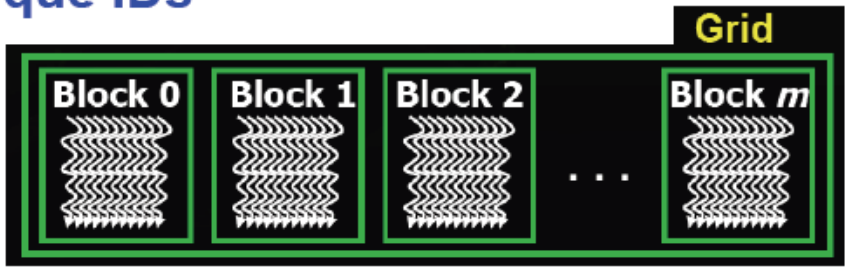
Hierarchy of Concurrent Threads

- **Parallel kernels composed of many threads**
  - —all threads execute same sequential program
  - —use parallel threads rather than sequential loops

- **Threads are grouped into thread blocks**
  - —threads in block can sync and share memory



- **Blocks are grouped into grids**
  - —threads and blocks have unique IDs
    - threadIdx: 1D, 2D, or 3D
    - blockIdx: 1D or 2D
  - —simplifies addressing when processing multidimensional data



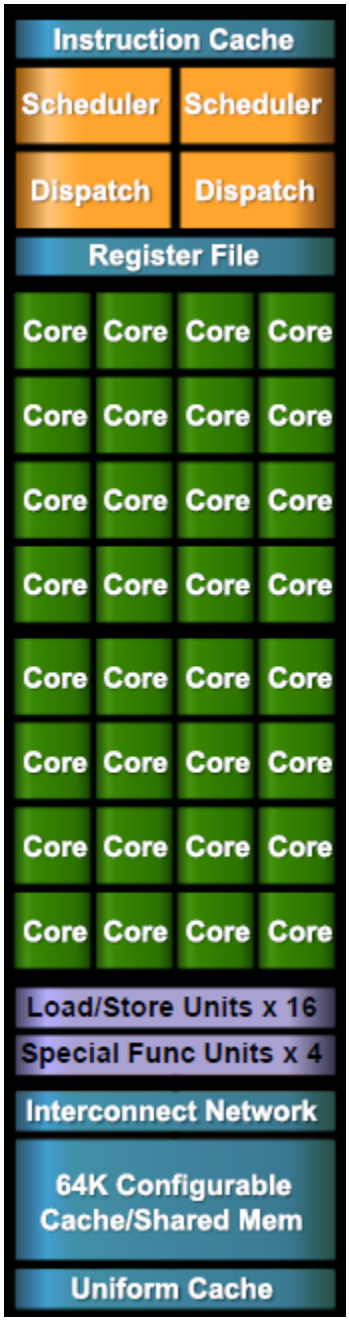
Slide credit: Patrick LeGresley, NVidia

The logical blocks of kernel instances (threads) must be mapped to the hardware!

177.4 GB/s

GPU
D-RAM
(1.5 GB)

(Device Memory)



Warp scheduling

Each Streaming Multiprocessor (SM) contains a number of individual cores (8, 32, 48 or 192 depending on the CUDA architecture).

The GPU "work distributer" component is responsible for allocating waiting blocks to a SM. The SM contains two scheduler components which allocate the block threads to the individual cores in groups of 32 called "warps". Blocks should therefore be sized in multiples of 32 (why? So there are no redundant cores!).

Each of the 32 threads in a warp shares one instruction counter; therefore, each warp thread is executing the same instruction at the same time. (Or is it?? What would happen when there is an "if" statement in the kernel?)

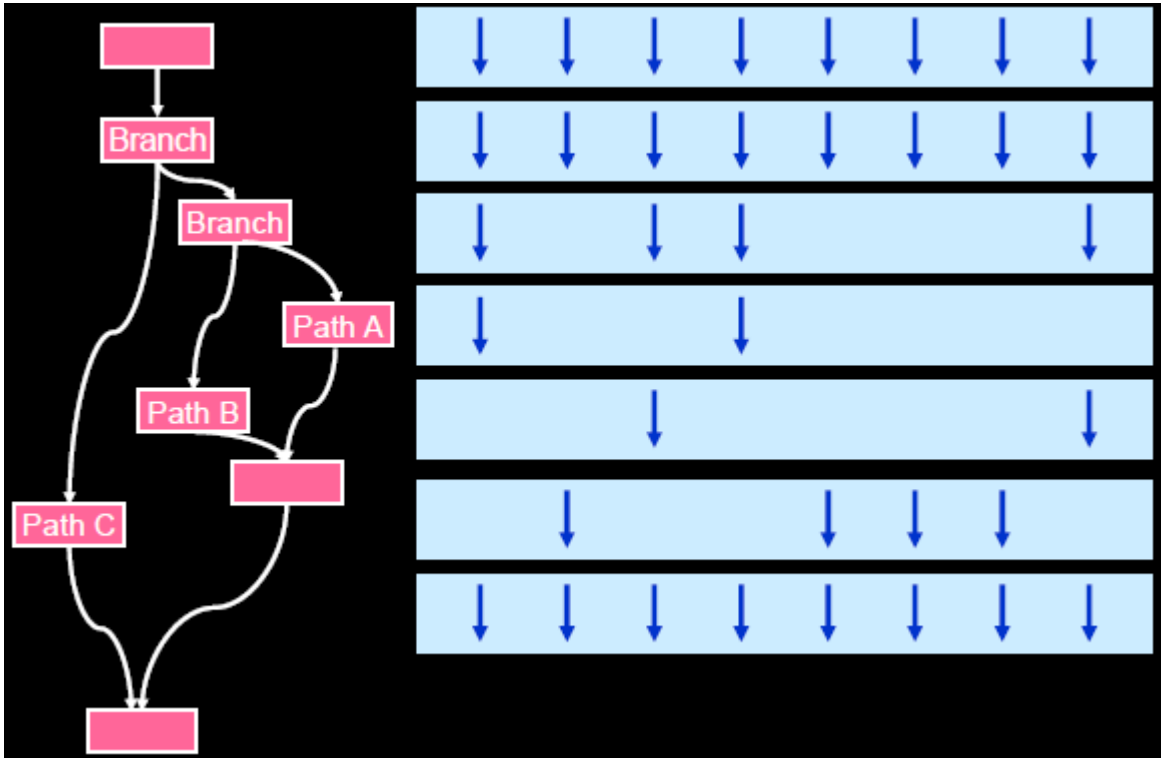
After a certain number of GPU clock cycles, the warp scheduler retires the current warp (e.g. warp A) and selects another warp (e.g. warp F) who's threads are all ready to execute their next statement. The SM will later select warp A again to execute it's next instruction. In this way, warps that are waiting for the results of their instruction (e.g. fetch data from memory) are temporarily retired whilst other warps are executing instructions. This is a CUDA architectural feature known as "Latency Hiding".

This cycle continues until all warps are concluded whereupon control passes back to the CPU.

This process is known as "Single Instruction Multiple Thread" (SIMT) ; the instruction counter executes a single instruction across a warp of 32 threads.

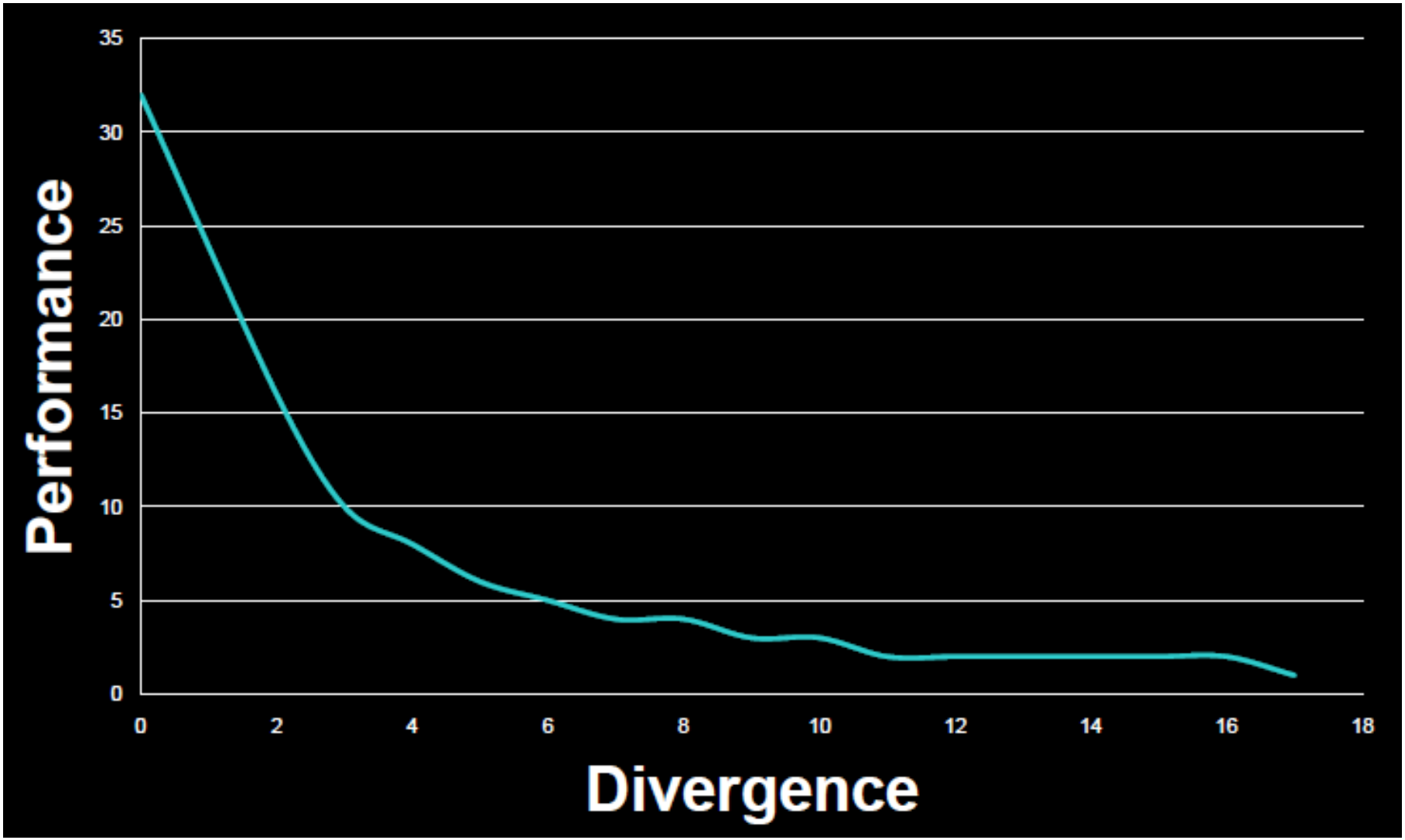
Thread Divergence

Each thread is free to execute any instruction. However, we need to be mindful of how divergent instructions behave inside a kernel…



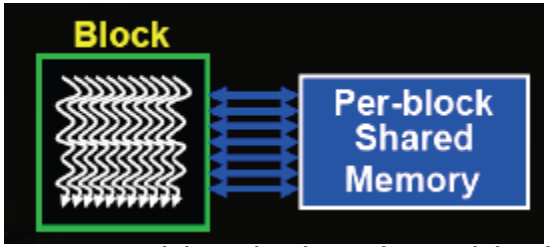

```
If (functionA(threadIdx.x))
{
  if(functionB(threadIdx.x))
   do_A();
  else
   do_B();
}
Else
  do_C();
```

Divergent threads force the rest of the threads in a warp to wait until they have completed their branches. Thread divergence happens within block/warp level only; simultaneous blocks running on different SMs can diverge without penalty.

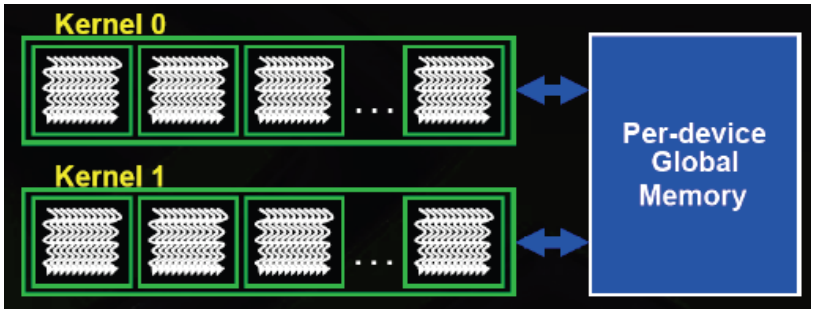

Performance
35
30
25
20
15
10
5
0
0
2
4
6
8
10
12
14
16
18
Divergence

3 key types of GPU memory : local , shared and global.

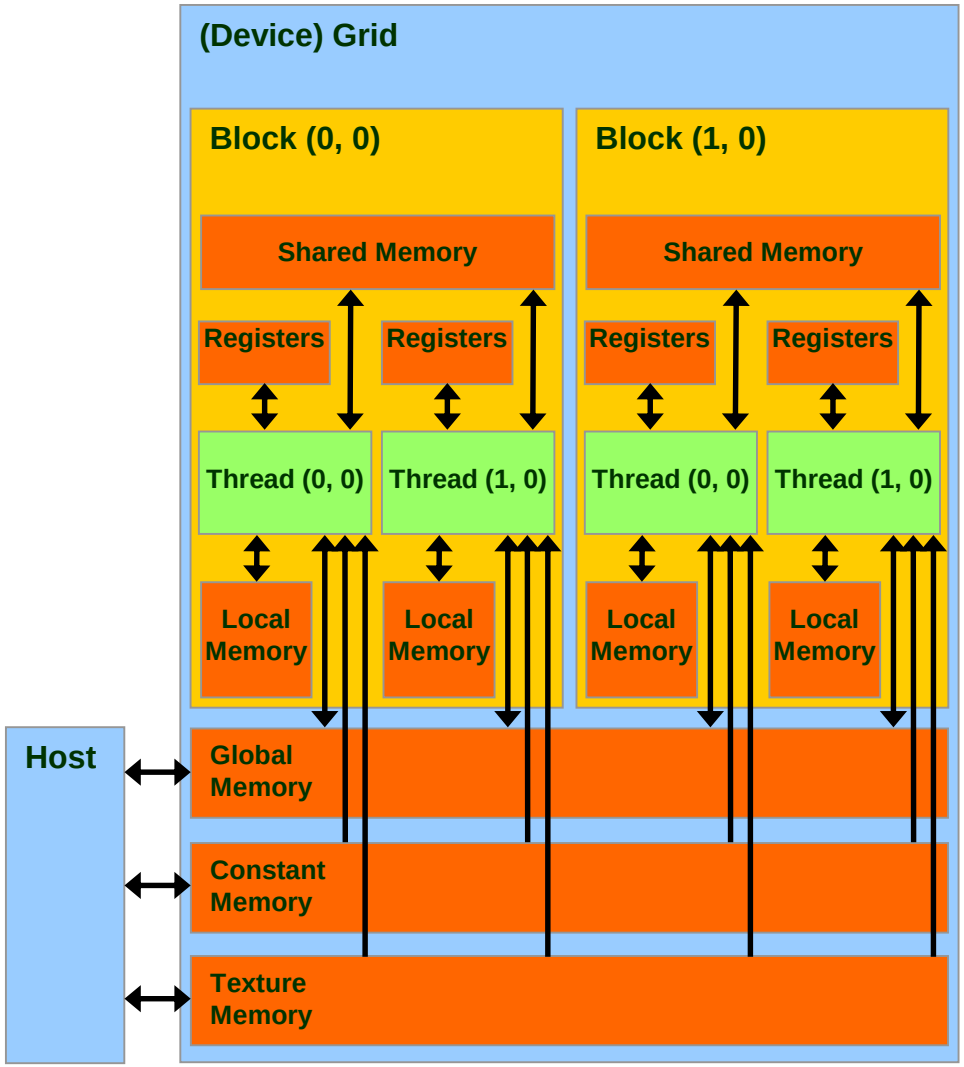
e.g. local variable declared in kernel, thread scope only. (very fast)



e.g. variables declared in a block with __shared__. Intra block scope only (fast).



e.g. a memory pointer to the general RAM, passed into kernel. Grid scope. (slowest)



CUDA Evolution

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

CUDA Evolution and Resource Constraints

The performance of CUDA GPUs differs according to the generation of architecture (Tesla (earliest), Fermi, Kepler (latest))

“Compute Capability” is NVIDIA’s term for their architecture versions. Generally, Tesla have “compute capability” 1, Fermi have 2.0 or 2.1 and Kepler have 3 and 3.5.

The GT480 (used in college) is a Fermi device, with compute capability 2.0.

“CUDA C Programming Guide” has resource tables for all architectures and compute capabilities.

Some important features and constraints for the GT480 are :

Core clock-rate :	700 Mhz
Shader (SM) clock-rate:	1401 Mhz
Threads per warp :	32 (this is the same for all architectures)
Max threads per block :	1024
Max blocks per grid :	65536

Lab 2

Vector addition!

Perform a vector addition on 2 large vectors on the CPU and GPU, and compare the results. A program scaffolding will be provided and you will be asked to fill in the blanks.

Check out the NVIDIA Visual Profiler tool when the GPU vector addition is working. This tool gives us information about our kernel, including resource usage, time taken etc.