Thread and Memory Model

Tran Giang Son, tran-giang.son@usth.edu.vn

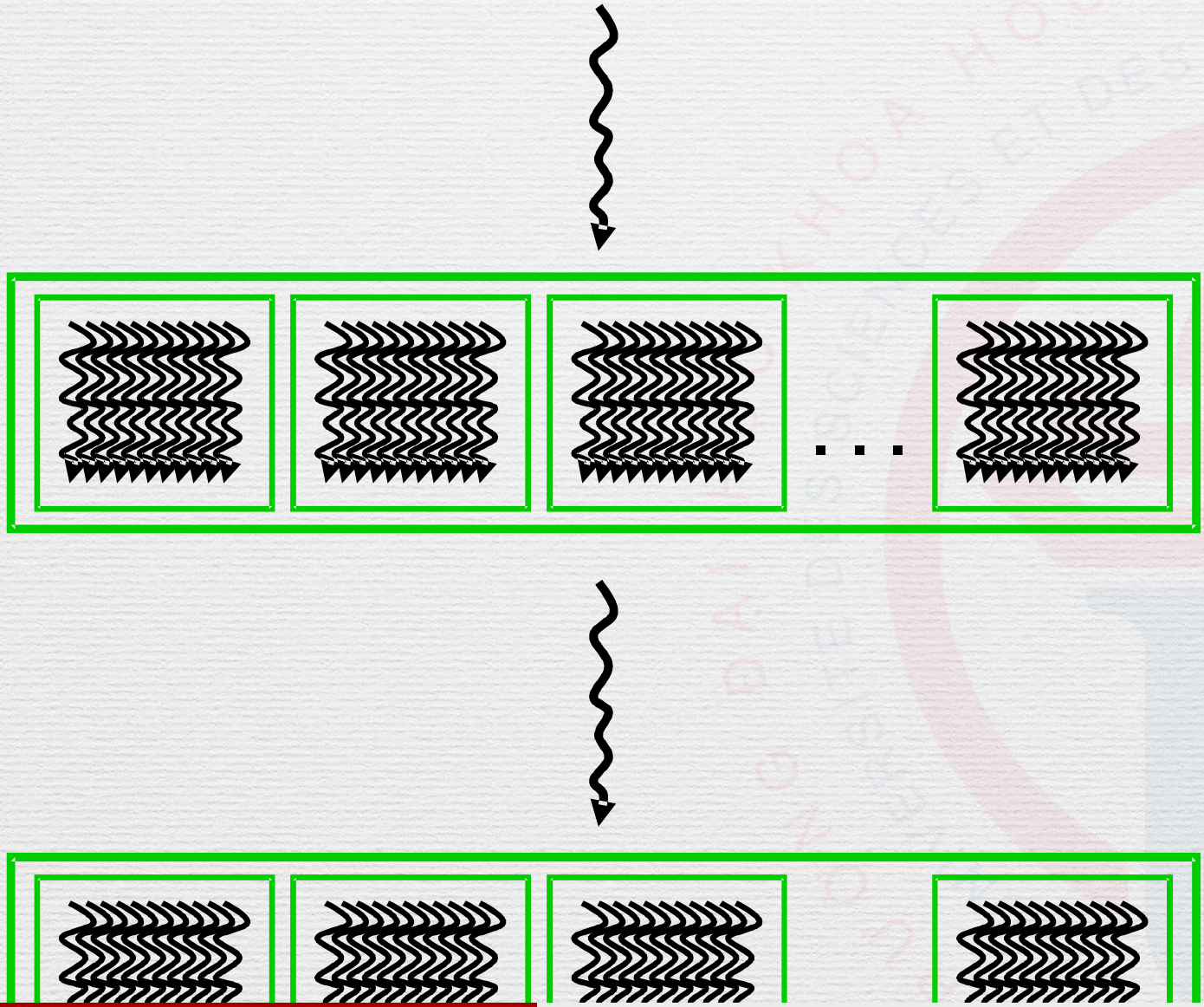
ICT Department, USTH

Thread Model

Thread

- What? a single sequential of execution
- SIMT on GPU
  - Same instruction
  - Same time
  - Different data
  - Natural fo graphics and scientific computing
- A way to simplify core

Thread



Thread: Software View

- Thread: a single flow of kernel execution
- Block: a bunch of thread (1D, 2D, 3D)
  - `blockDim.x`, `blockDim.y`, `blockDim.z`
- Grid: a bunch of block (1D, 2D, 3D)
  - `gridDim.x`, `gridDim.y`, `gridDim.z`

Thread: Restrictions

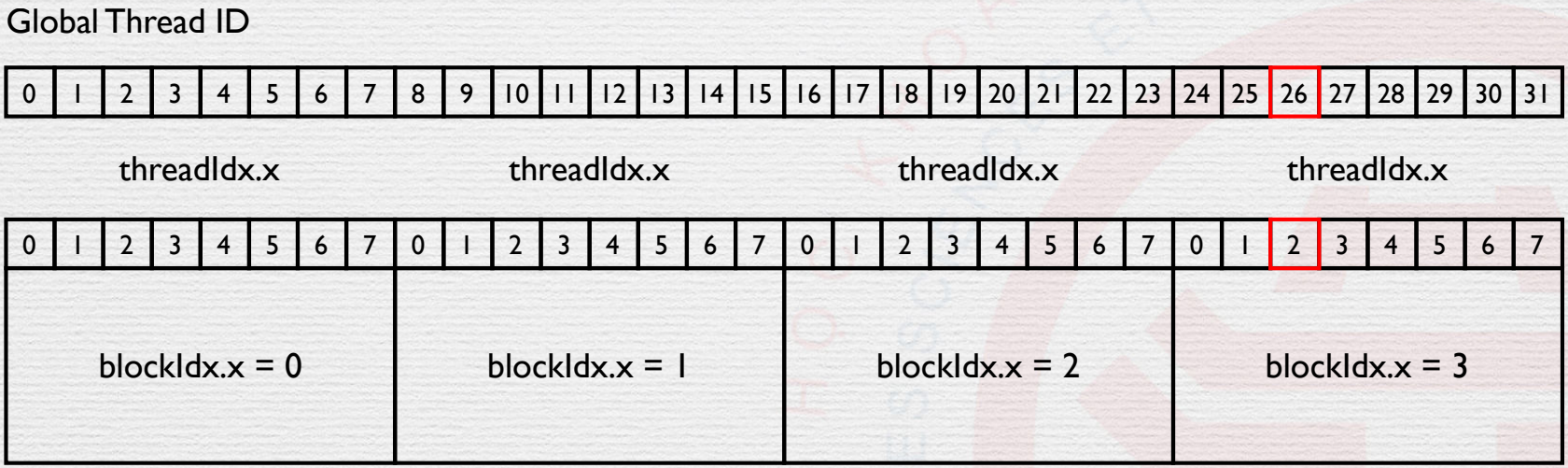
- Dimensions is fixed after kernel launch
- All blocks in a grid have the same dimension
- Block size and grid size are upper bounded

Thread: Restrictions

- Dimensions is fixed after kernel launch
- All blocks in a grid have the same dimension
- Block size and grid size are upper bounded

```
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

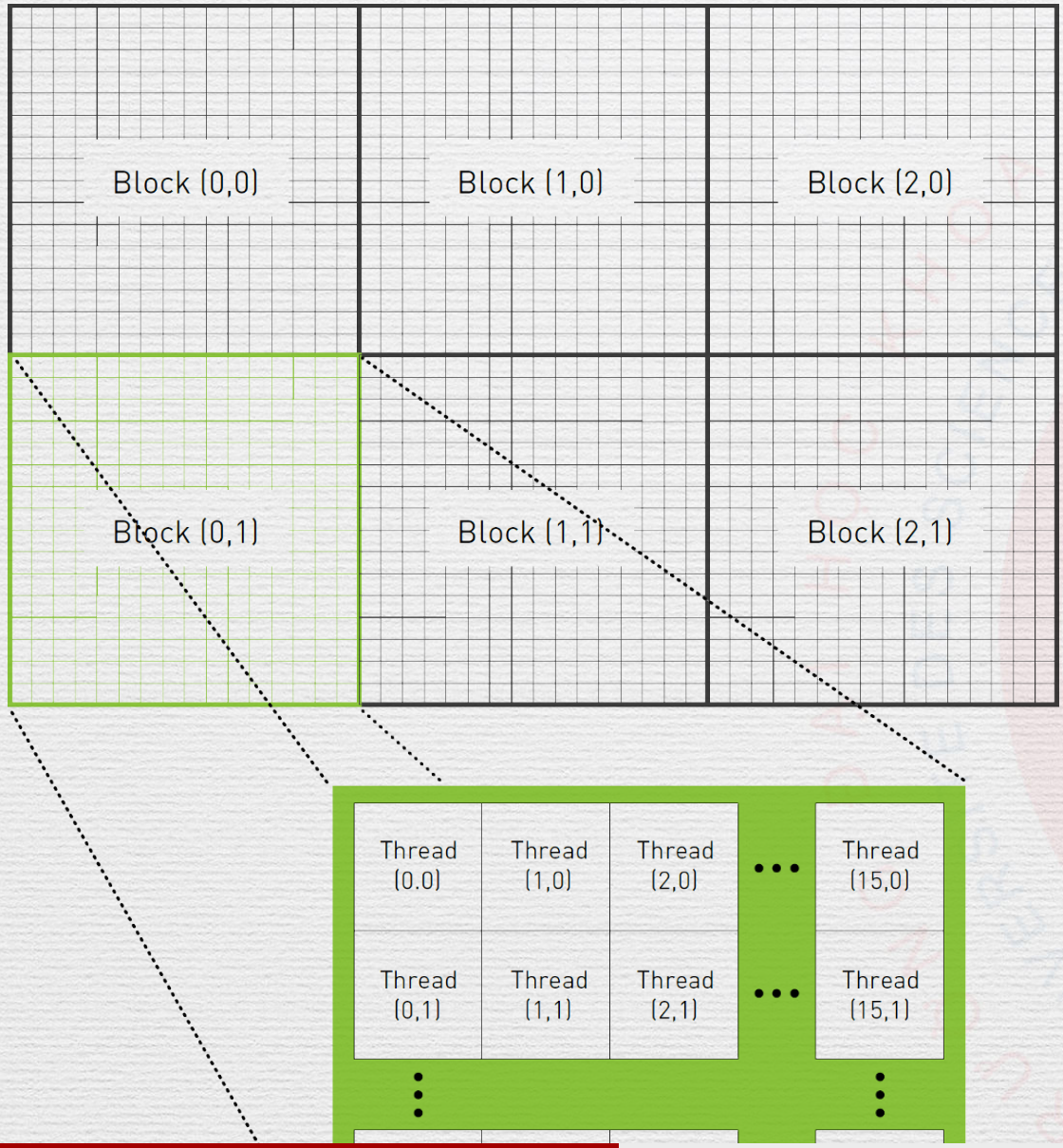
Thread: Software View




```
blockDim.x = 8

int globalThreadId = threadIdx.x + blockIdx.x * blockDim.x
```

Thread: Software View



Thread: Software View

- Where are we?
  - 1D: `x = threadIdx.x + blockIdx.x * blockDim.x`
  - 2D: `y = threadIdx.y + blockIdx.y * blockDim.y`
  - 3D: `z = threadIdx.z + blockIdx.z * blockDim.z`

Thread: Software View

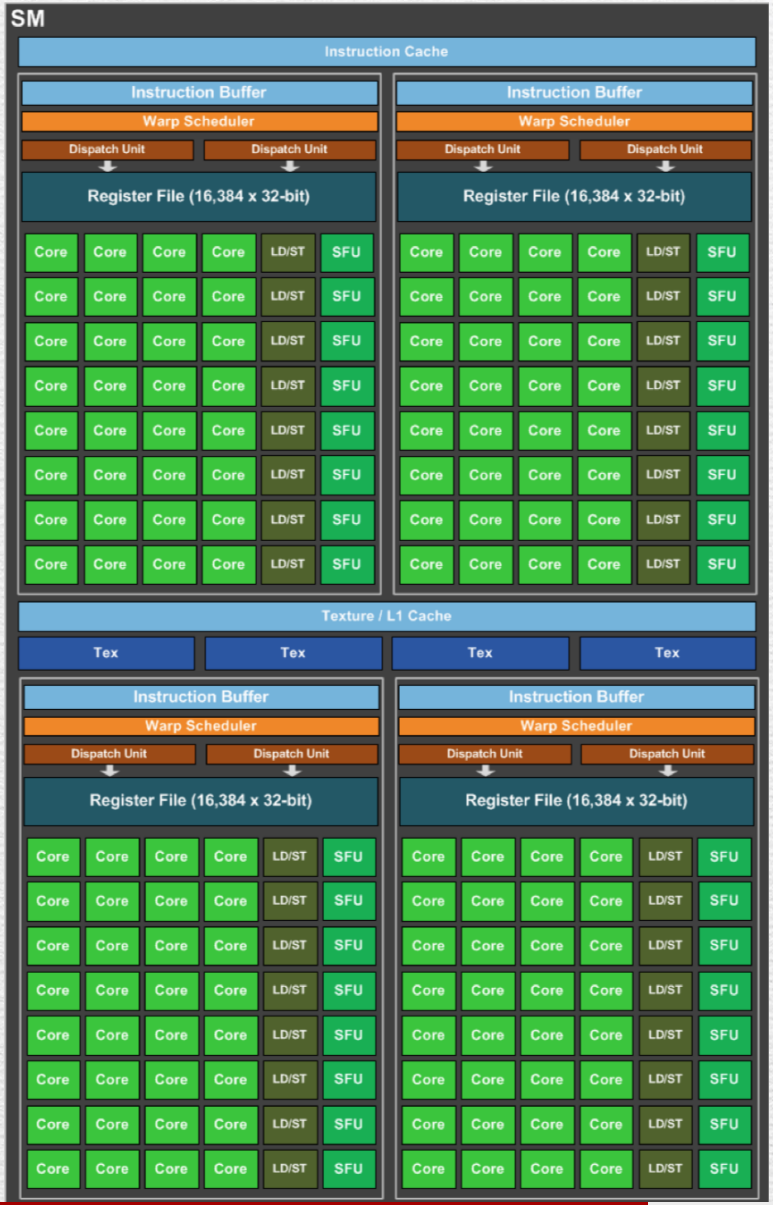
- Where are we?
  - 1D: `x = threadIdx.x + blockIdx.x * blockDim.x`
  - 2D: `y = threadIdx.y + blockIdx.y * blockDim.y`
  - 3D: `z = threadIdx.z + blockIdx.z * blockDim.z`
- How about `gridDim`?
  - Number of blocks in each dimension in the grid
  - Use case: 1D grid for a 2D image
    - Length of a row: `w = blockDim.x * gridDim.x`
    - Next row: `x += w`

Thread: Hardware View

- Streaming Processor (CUDA cores)
- Streaming Multiprocessor : A bunch of Streaming Processors plus some extra Special Function Units (sine/cosine/...)
- Graphics Processing Cluster : A bunch of Streaming Processors
- Many simple cores $\Rightarrow$ better performance

Thread: Hardware View

Thread: Hardware View



Thread: Assignment

- Each SM has "multiple of 32" cores

Thread: Assignment

- Each SM has “multiple of 32” cores
- Threads in SM execute in group of 32 threads
  - A group of 32 thread inside a SM is called « Warp »
  - Warp is unit of thread scheduling in SMs

Thread: Assignment

- Each SM has “multiple of 32” cores
- Threads in SM execute in group of 32 threads
  - A group of 32 thread inside a SM is called « Warp »
  - Warp is unit of thread scheduling in SMs
- Blocks are assigned to SMs into multiple of warps
  - Number of blocks per SM is constrained

Thread: Assignment

- Each SM has “multiple of 32” cores
- Threads in SM execute in group of 32 threads
  - A group of 32 thread inside a SM is called « Warp »
  - Warp is unit of thread scheduling in SMs
- Blocks are assigned to SMs into multiple of warps
  - Number of blocks per SM is constrained
- No specific mapping between thread and core

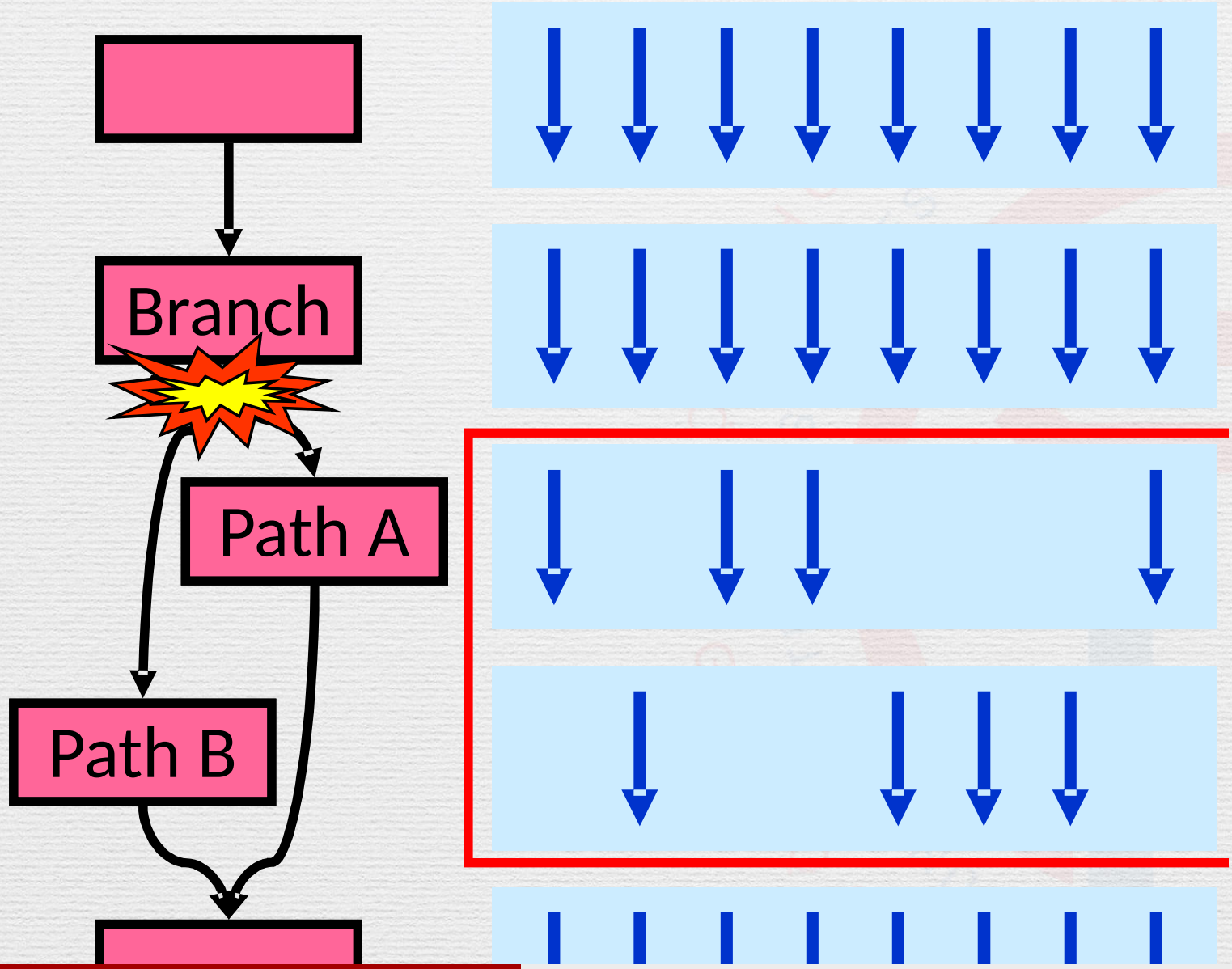
Thread: Assignment

- Each warp is executed in SIMD
  - All threads must execute same instruction at any time
- Fact
  - Not all warps are scheduled at anytime
  - Wait for data
  - Branch divergence

Thread: Assignment

- CUDA virtualizes the physical hardware
  - Thread : virtualized scalar processor
    - registers
    - PC
    - state
  - Block is a virtualized multiprocessor
    - threads
    - shared memory

Thread: Branch divergence



Thread: Branch divergence

- When?
  - Condition
- Divergence

```
if (threadIdx.x > 2) { }
```

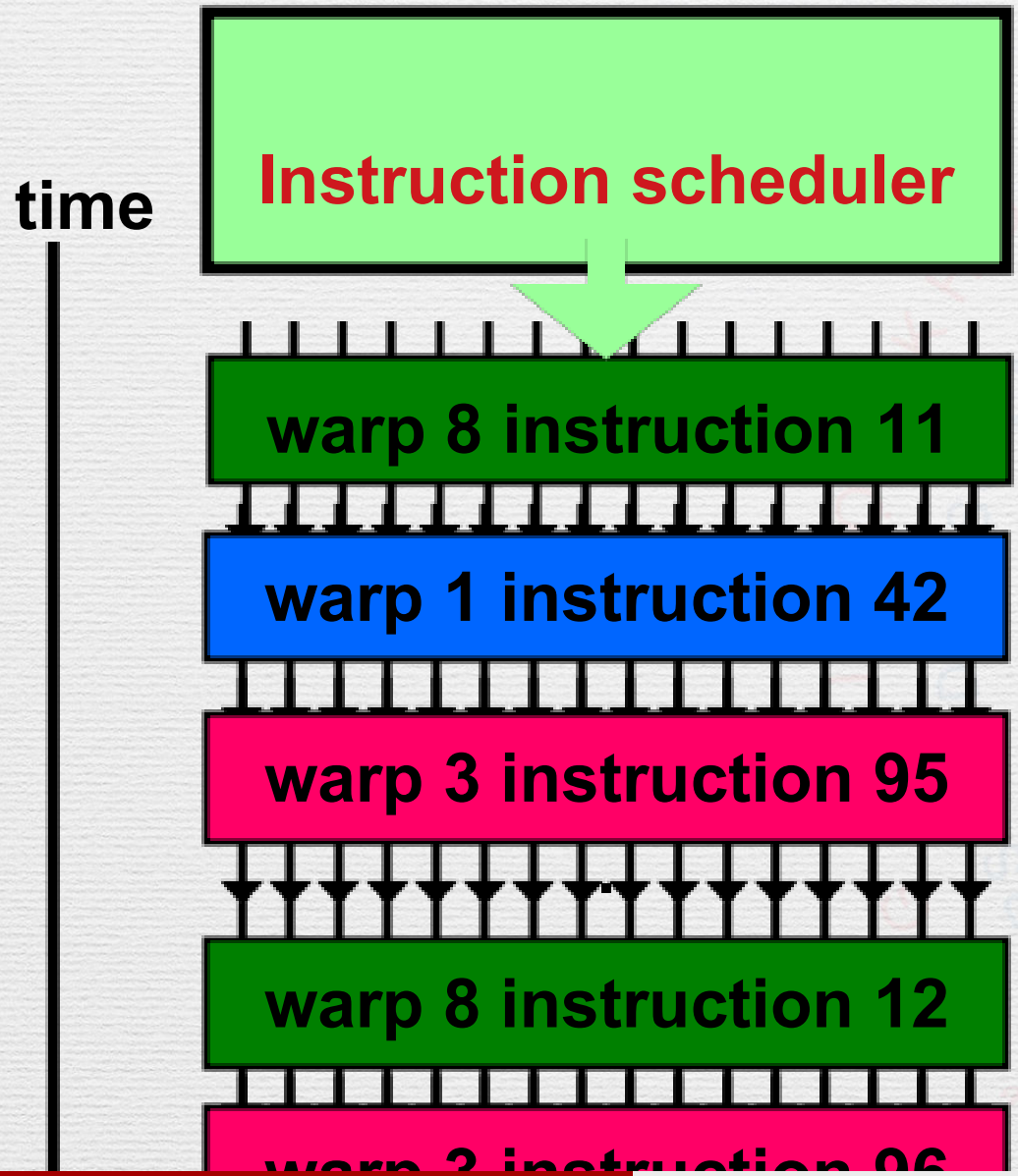
- No divergence

```
if (threadIdx.x / WARP_SIZE > 2) { }
```

Thread: Latency Tolerance

- When a warp does something with high latency
  - Pause it
  - Schedule next warp
- No context switch
  - Large register file
  - No need to "switch" register content to memory
  - Zero overhead

Thread: Latency Tolerance



Thread: Latency Tolerance

- Latency tolerance relies on many warps
- Branch divergence does not affect GPU high throughput like CPU
- CPU focuses on low latency
  - Branch is important
  - Branch prediction is even more important

Block size in CUDA

- Previously, in launching kernel

```
kernelName<<<numBlock, blockSize>>>(args...)
```

- Example

```
int pixelCount = imageWidth * imageHeight;
int blockSize = 64;
int numBlock = pixelCount / blockSize;
grayscale<<<numBlock, blockSize>>>(devInput, devOutput);
```

- This is 1D kernel launch
  - `numBlock` is essentially `gridDim.x`

Block size in CUDA

- For 2D kernel launches
  - Grid size is `dim3`
  - Block size is `dim3`
  - Use `dim3()` constructor
- Launch a kernel with of 8×8 blocks, each block has 32×32 threads

```
dim3 gridSize = dim3(8, 8);
dim3 blockSize = dim3(32, 32);
grayscale<<<gridSize, blockSize>>>(devInput, devOutput);
```

Labwork & Exercises 4: Threads

- Copy labwork 3 code to labwork 4
- Improve labwork 4 code to use 2D blocks
- Use existing profiling class `Timer` to measure speedup
- Write a report (in LaTeX)
  - Name it « Report.4.threads.tex »
  - Explain how you improve the labwork
  - Try experimenting with different 2D block size values
  - Plot a graph of block size vs speedup
  - Compare speedup with previous 1D grid
  - Answer the questions in the upcoming slides, explain why
- Push the report and your code to your forked repository

Thread: Exercises 1

Consider a GPU having the following specs (maximum numbers):

- 512 threads/block
- 1024 threads/SM
- 8 blocks/SM
- 32 threads/warp

What is the best configuration for thread blocks to implement grayscaling?

- 8×8
- 16×16
- 32×32

Thread: Exercises 2

Consider a device SM that can take max

- 1,536 threads
- 4 blocks

Which of the following block configs would result in the most number of threads in the SM?

- 128 threads/blk
- 256 threads/blk
- 512 threads/blk
- 1,024 threads/blk

Thread: Exercises 3

Consider a vector addition problem

- Vector length is 2,000
- Each thread produces one output
- Block size 512 threads.

How many threads will be in the grid?

Memory

Memory

- Example of a kernel doing vector addition

```
__global__ add(float *out, float *in1, float *in2) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    out[tid] = in1[tid] + in2[tid];
}
```

Memory

- Example of a kernel doing vector addition

```
__global__ add(float *out, float *in1, float *in2) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    out[tid] = in1[tid] + in2[tid];
}
```

- GTX 1080: 352 GB/s global memory bandwidth

Memory

- Example of a kernel doing vector addition

```
__global__ add(float *out, float *in1, float *in2) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    out[tid] = in1[tid] + in2[tid];
}
```

- GTX 1080: 352 GB/s global memory bandwidth
- Single precision float : 4 bytes

Memory

- Example of a kernel doing vector addition

```
__global__ add(float *out, float *in1, float *in2) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    out[tid] = in1[tid] + in2[tid];
}
```

- GTX 1080: 352 GB/s global memory bandwidth
- Single precision float : 4 bytes
- Max 88 giga single precision float loaded from/to global memory per sec

Memory

- Example of a kernel doing vector addition

```
__global__ add(float *out, float *in1, float *in2) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    out[tid] = in1[tid] + in2[tid];
}
```

- GTX 1080: 352 GB/s global memory bandwidth
- Single precision float : 4 bytes
- Max 88 giga single precision float loaded from/to global memory per sec
- If no cache: 2 in, 1 out per FLOP $\Rightarrow$ max 29.3 GFLOPS

Memory

Something's wrong.

GeForce 10 (10xx) series

Model	Launch	Processing power (GFLOPS) Single precision (Boost)
GeForce GTX 1080	May 27, 2016	8228 (8873)
GeForce GTX 1080 Ti	March 10, 2017	10609 (11340)
NVIDIA TITAN X	August 2, 2016	10157 (10974)

8 more rows

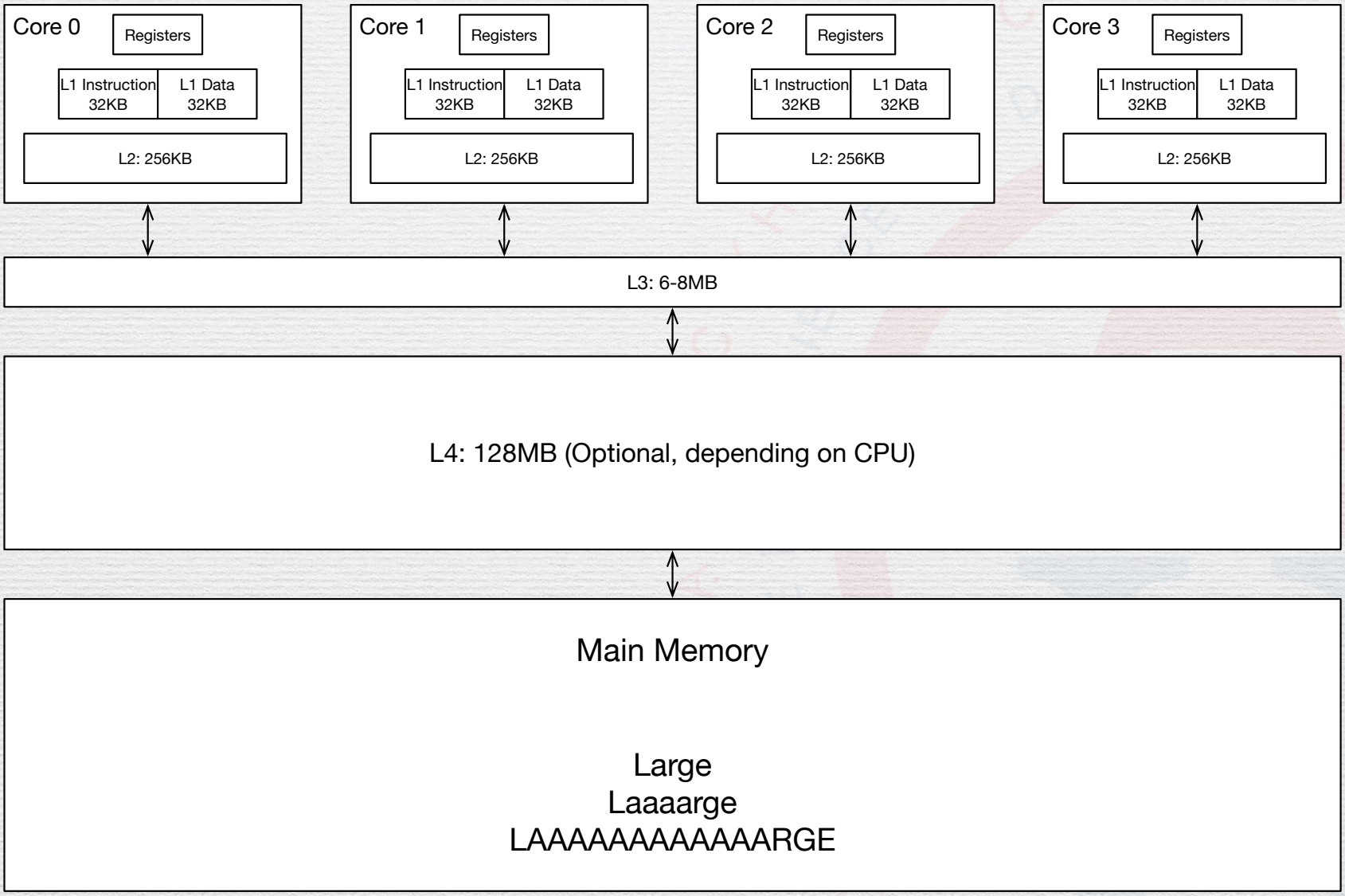
GeForce 10 series - Wikipedia
https://en.wikipedia.org/wiki/GeForce_10_series

About this result Feedback

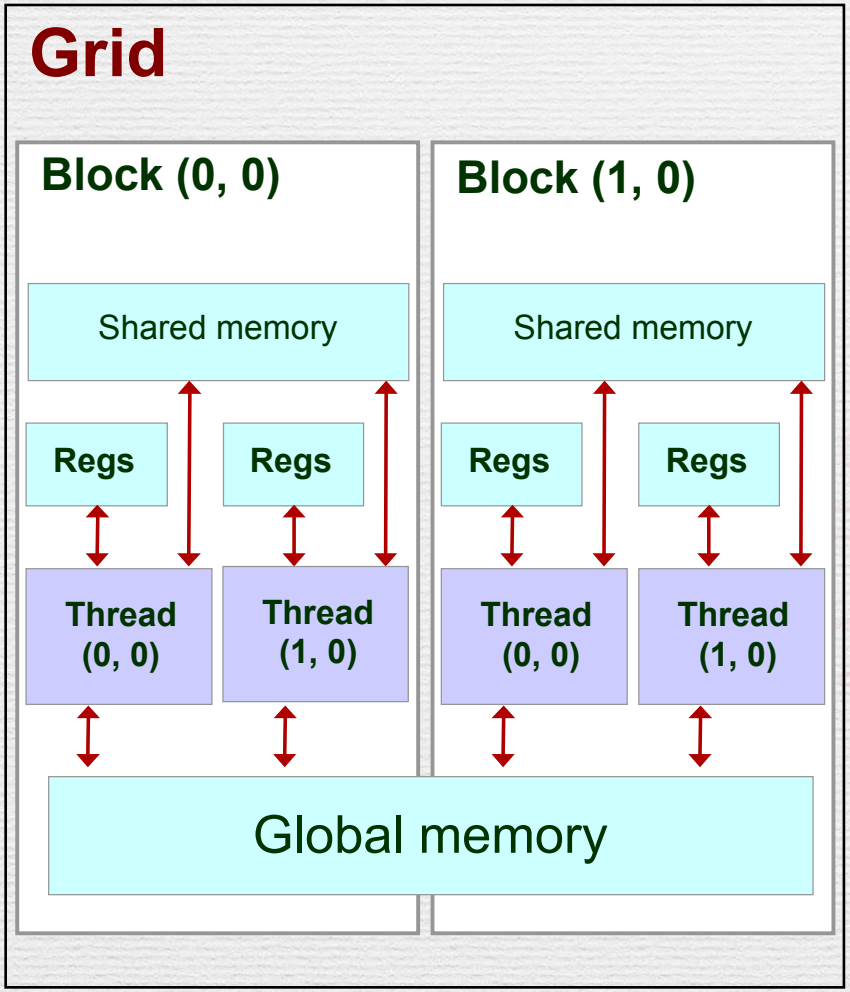
Memory

- Key challenge
  - Fast computation but slow memory?
  - Lots of memory
  - Fast + Lots == Expensive
- Hierarchical design

Memory Hierarchical Design: Host



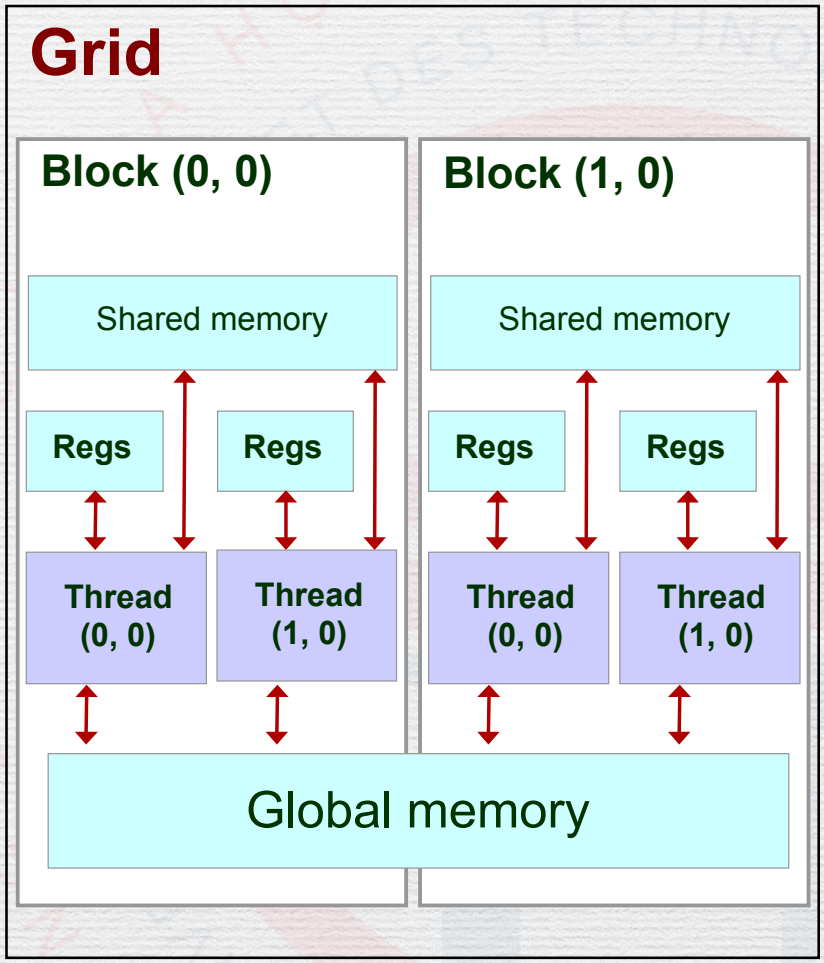
Memory Hierarchical Design: Device



Memory Hierarchical Design: Device

Registers

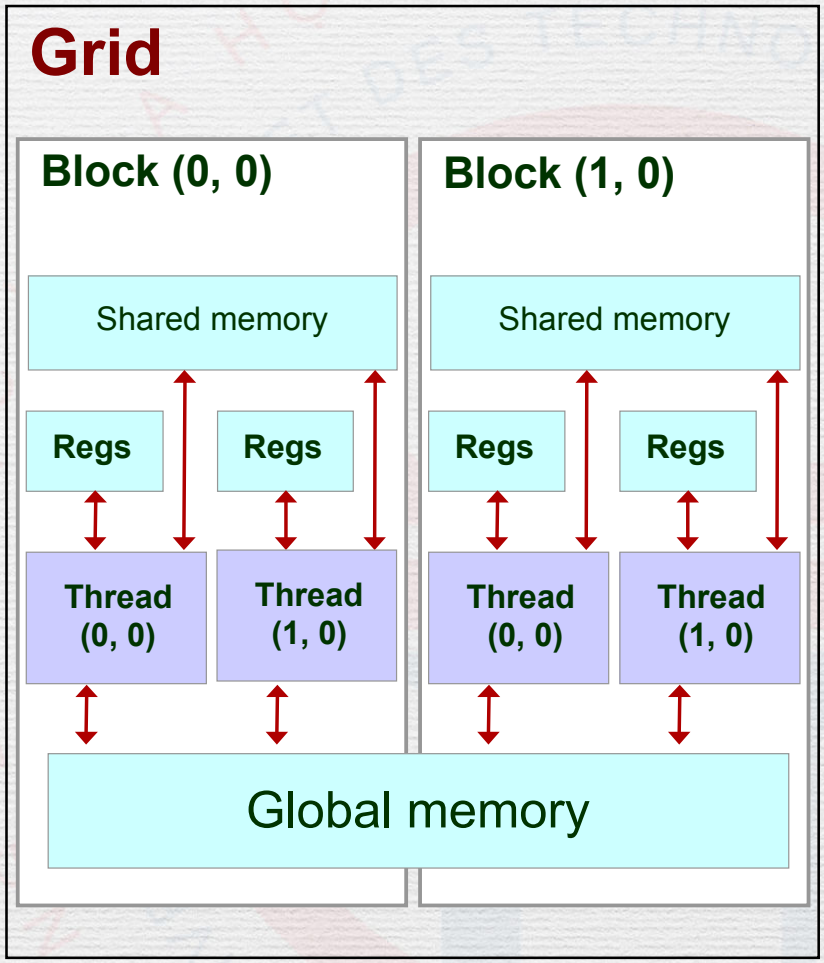
- Fastest
- On-chip only
- No off-chip bandwidth
- Only accessible by a thread
- Lifetime of a thread



Memory Hierarchical Design: Device

Shared Memory

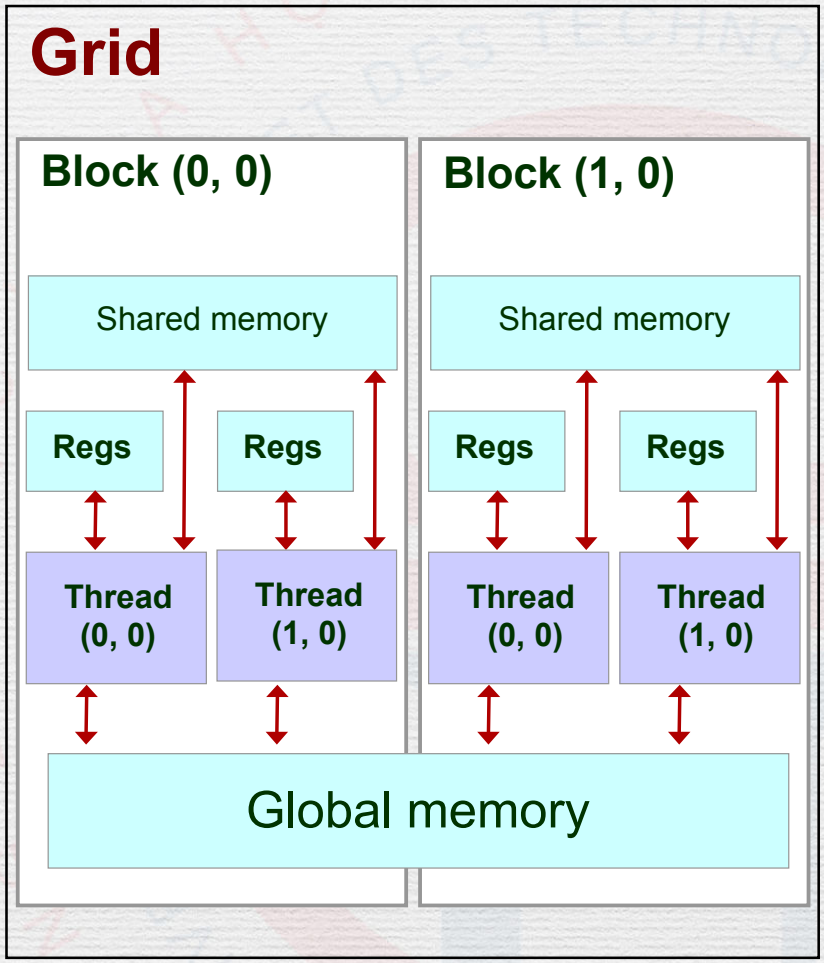
- Extremely fast
- Highly parallel
- Restricted to a block



Memory Hierarchical Design: Device

Global Memory

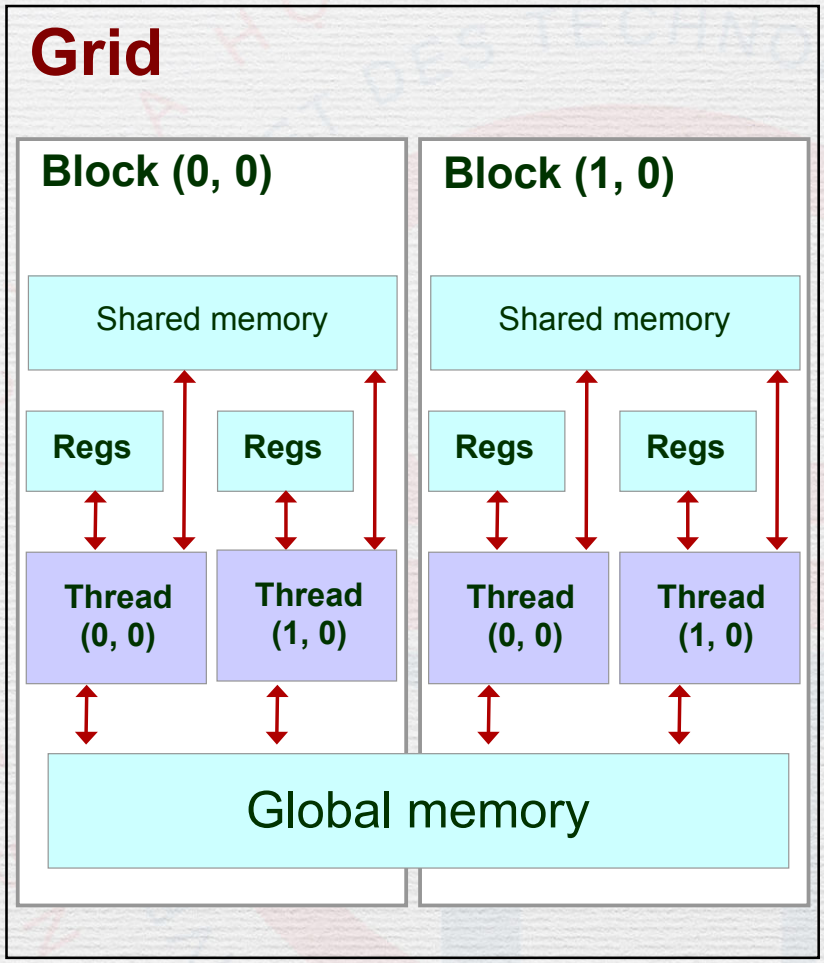
- Typically implemented in DRAM
- High access latency: 400-800 cycles
- Finite access bandwidth
- Potential of traffic congestion
- Throughput up to 900GB/s (Volta V100 on HBM2)



Memory Hierarchical Design: Device

Constant Memory

- Small : 64KB/block
- Read only from device
- Writable from host
- Short latency and high bandwidth
  - If warps accesses the same location



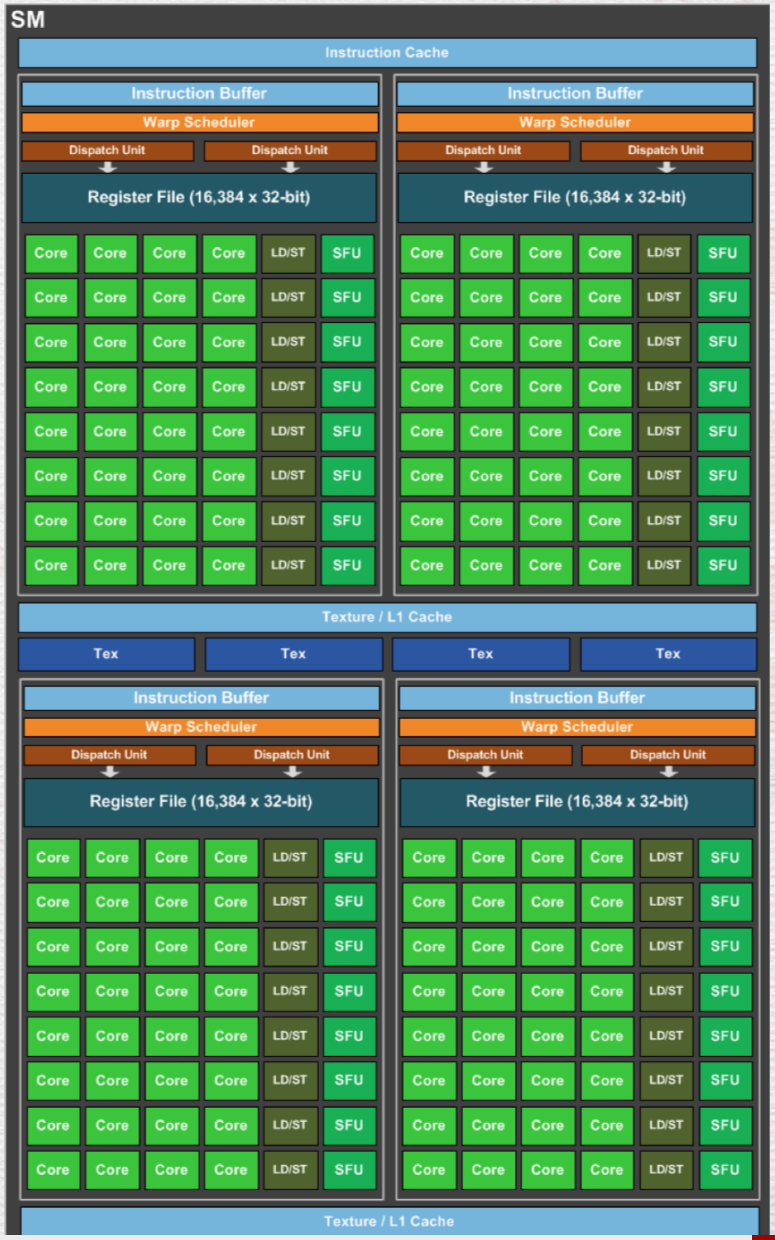
Memory Hierarchical Design: Device

Variable	Memory	Scope	Lifetime	Latency
`int localVar;`	register	thread	thread	1x
`int localArray[10];`	local	thread	thread	100x
`__shared__ int sharedVar;`	shared	blocks	thread	1x
`__device__ int globalVar;`	global	grid	app	100x
`__constant__ int constVar;`	constant	grid	app	1x

Note: “local” memory is in fact a part of the global memory.

Memory of GTX 1080

- GDDR5X
- 256-bit wide bus
- 352GB/s (ref: PCIEx3: 985MB/sec/lane)
- Unified 2MB L2 cache
- 1 GPC consists of 5 SMs, each SM
  - 4x 64KB registers
  - 96KB shared memory
  - 48KB L1 cache
- Memory compression engine



Maximizing Computation

Previously...

```
__global__ add(float *out, float *in1, float *in2) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    out[tid] = in1[tid] + in2[tid];
}
```

29.3 GFLOPS

Maximizing Computation: Memory Architecture

- Execution speed is based on data locality
  - Temporal locality: just-accessed is likely to be accessed again
  - Spatial locality: nearby data is likely to be used soon (image, video, sound)
- Order of performance
  - Registers
  - Shared memory / Constant memory (temporal locality)
  - Texture memory (spatial locality)
  - Global memory

Maximizing Computation: Memory Architecture

- **YOU** dictate:
  - visibility
  - access speed
- How?
  - Access to registers need fewer instructions than global memory
  - Aggregate register files bandwidth ~ two orders of magnitude that of the global memory
  - Shared memory is part of the address space
    - Requires load/store

Maximizing Computation: Memory Architecture

- Global memory access is performance bottleneck
  - Less global memory access, better perf
  - Tiling partition the data into small chunks, fittable into shared memory
  - Can speed up with coalesced read/write

Maximizing Computation: Memory Coalesce

- Memory access are in transactions
  - A block of 32, 64, 128, 256 bytes
- Coalesced read/writes:
  - Parallel read/writes from threads in a block
  - Sequential memory locations...
  - ...with appropriate alignment
- Minimize global memory bandwidth requirement

Maximizing Computation: Memory Alignment

- Addresses being powers-of-two bytes (4 to 16) are aligned
- Aligned addresses can be accessed with a single memory instruction
- All other accesses are split in multiple instructions.

$\Rightarrow$ Better performance with aligned addresses

Maximizing Computation: Coalesce and Alignment

- Structure of array vs Array of structure

```
struct {
    uint8_t r, g, b;
} AoS[N];

struct {
    uint8_t r[N];
    uint8_t g[N];
    uint8_t b[N];
} SoA;
```

Maximizing Computation: Coalesce and Alignment

- Array of Structs
  - More readable: objects are kept together
  - Better cache locality: members are accessed together
    - Better coalesce
    - e.g. RGB are used together in case of grayscaling
- Struct of Arrays
  - Potentially more efficient in several cases
    - e.g. processing one channel only
  - Less paddings: only between array, not between struct

Maximizing Computation

- Shared memory is fast, **IF**
  - All threads in warp access the same location
  - Or linear access
- Shared memory's random access is slow
  - Bank conflict

Maximizing Computation

Thread local computation

- Where are we?

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

- Load data from global memory (coalesced)

```
char r = inputImage[tid].x;
char g = inputImage[tid].y;
char b = inputImage[tid].z;
```

- Do computation with registers

```
char gray = (r + g + b) / 3;
```

- Write back to global memory (coalesced)

```
inputImage[tid].x = gray;
```

Maximizing Computation

Block local computation

- Where are we? ...
- Load data to **shared** memory

```
__shared__ char tile[BLOCK_SIZE];
int tid = ...;
tile[threadIdx.x] = input[tid].x;
```

- Synchronize between threads in a block

```
__syncthreads()
```

- Calculate on shared memory
- Write back to global memory (coalesced)

Labwork 5: Gaussian Blur Convolution

- Copy your grayscaling kernel in labwork 4 to labwork 5
- Change it to 7x7 Gaussian blur convolution
  - Without shared memory
  - With shared memory
- Use existing profiling class `Timer` to measure speedup
- Write a report (in LaTeX)
  - Name it « Report.5.gaussian.blur.tex »
  - Explain how you implement the Gaussian Blur filter
  - Try experimenting with different 2D block size values
  - Plot a graph of block size vs speedup (with/without shared memory)
- Push the report and your code to your forked repository

Extra: Gaussian Blur Convolution

- Convolution
- Mostly to blur the input image
- The 2D kernel follows a normal distribution

$$G(x,y) = \frac{1}{2\pi\sigma^2} \exp\left[-\frac{(x-\mu_x)^2 + (y-\mu_y)^2}{2\sigma^2}\right]$$

- σ : standard deviation of the distribution
- μ_x : Mean of the kernel in horizontal axis
- μ_y : Mean of the kernel in vertical axis

Extra: Gaussian Blur Convolution

- Example 7 x 7 (1003 total)

0	0	1	2	1	0	0
0	3	13	22	13	3	0
1	13	59	97	59	13	1
2	22	97	159	97	22	2
1	13	59	97	59	13	1
0	3	13	22	13	3	0
0	0	1	2	1	0	0