Parallel Programming Language CUDA (C extension)*

Jiwon Seo

*Slides are based on Caltech CS179 course

GPU Programming – First Try

CUDA Programs

1. CUDA Programming Interfaces (Library)
2. Extended Syntax

GPU Programming – First Try

STEP 1. Add function needs to run on GPU:

```
// CUDA Kernel function to add the elements of two arrays
on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

GPU Programming – First Try

STEP 2. Memory allocation on GPU:

```
// Allocate Unified Memory -- accessible from CPU or GPU
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

...

// Free memory
cudaFree(x);
cudaFree(y);
```

GPU Programming – First Try

STEP 3. Launch add() kernel on GPU

```
add<<<1, 1>>>(N, x, y);
```

GPU Programming – First Try

STEP 3a. Launch add() kernel on GPU

```
add<<<1, 1>>>(N, x, y);
```

GPU Programming – First Try

STEP 3b. Wait until add() kernel finishes

```
add<<<1, 1>>>(N, x, y);

cudaDeviceSynchronize();
```

GPU Programming – First Try

Compile with NVCC and run!

```
$ nvcc add.cu -o add_cuda
$ ./add_cuda
Max error: 0.00000
```

GPU Programming – First Try

Compile with NVCC and run!

```
$ nvcc add.cu -o add_cuda
$ ./add_cuda
Max error: 0.00000
```

Profile with nvprof

```
$ nvprof ./add_cuda
```

```
$ nvprof ./add_cuda
==3355== NVPROF is profiling process 3355, command: ./add_cuda
Max error: 0
==3355== Profiling application: ./add_cuda
==3355== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  463.25ms         1  463.25ms  463.25ms  463.25ms  add(int, float*, float*)
...
```

From 1 Thread to 256 Threads

Launching add() kernel with 256 threads

```
add<<<1, 256>>>(N, x, y);
```

From 1 Thread to 256 Threads

Launching add() kernel with 256 threads

```
add<<<1, 256>>>(N, x, y);
```

In add() kernel, 256 threads do the same computation

```
// CUDA Kernel function to add the elements of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
```

From 1 Thread to 256 Threads

Launching add() kernel with 256 threads

```
add<<<1, 256>>>(N, x, y);
```

Let 256 threads do different computation

```
__global__
void add(int n, float *x, float *y) {
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}
```

From 1 Thread to 256 Threads

Launching add() kernel with 256 threads

```
add<<<1, 256>>>(N, x, y);
```

nvprof for profiling

```
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  2.7107ms         1  2.7107ms  2.7107ms  2.7107ms  add(int, float*, float*)
```

463ms → 2.7ms : 200X speedup!

From 1 Block to multiple Blocks

First parameter for kernel launch

```
add<<<1, 256>>>(N, x, y);
```

Number of thread blocks

From 1 Block to multiple Blocks

First parameter for kernel launch

```
add<<<1, 256>>>(N, x, y);
```

➔ **Multiple thread blocks**

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

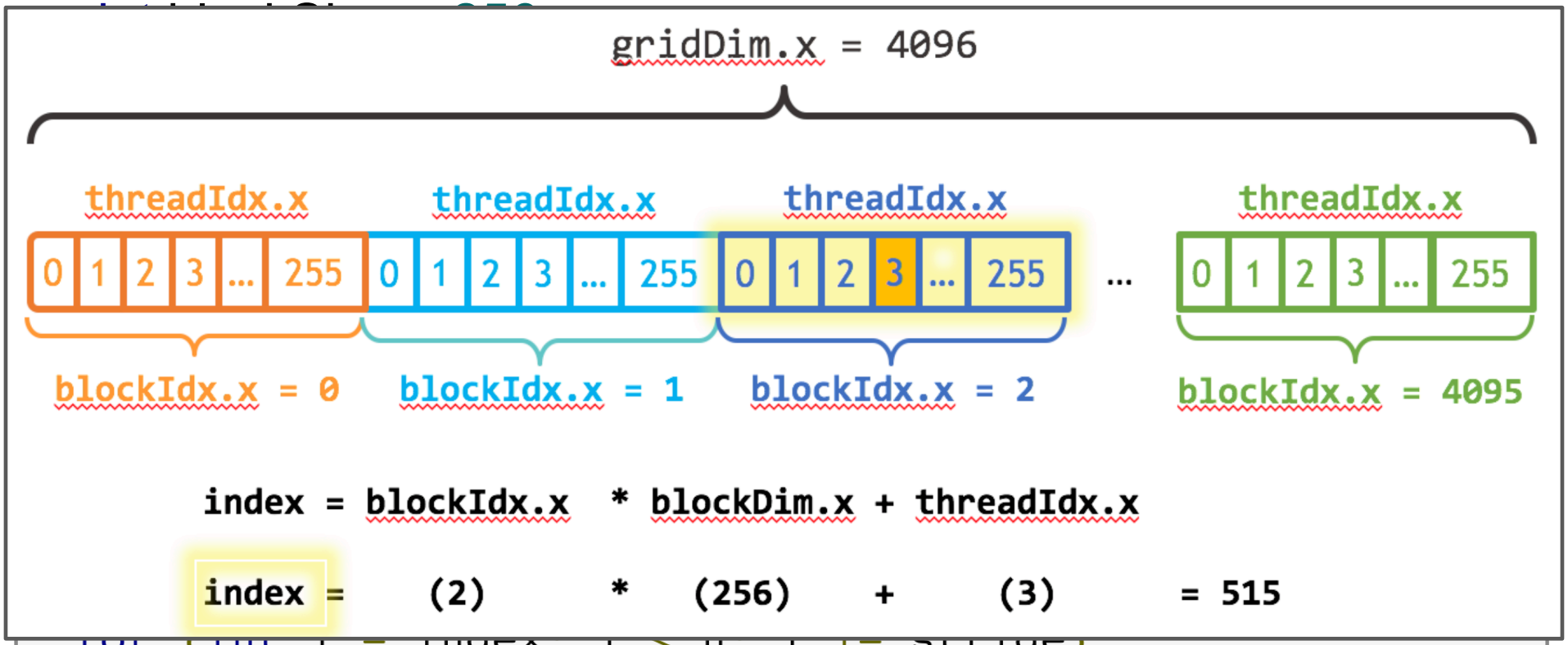
From 1 Block to multiple Blocks

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

Need to update kernel function

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

From 1 Block to multiple Blocks




```
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

From 1 Block to multiple Blocks

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

nvprof for profiling

```
Time(%)      Time     Calls       Avg       Min       Max
100.00%  94.015us         1  94.015us  94.015us  94.015us
```

463ms → 2.7ms → 94us : 4000X speedup!

Real-world GPU programming

- Setup inputs on the host (CPU-accessible memory)
- Allocate memory for outputs on the host
- Allocate memory for inputs on the GPU
- Allocate memory for outputs on the GPU
- Copy inputs from host to GPU
- Start GPU kernel (function that executed on GPU)
- Copy output from GPU to host

NOTE: Copying can be synchronous or asynchronous

Setting up Inputs and Outputs on Host (CPU)

```
float *h_a, *h_b, *h_c;
h_a = new float [N];
h_b = new float [N];
h_c = new float [N];
```

Allocate memory (Input & Output) on GPU

```
float *d_a, *d_b, *d_c;
cudaMalloc((void**)&d_a, N*sizeof(float));
cudaMalloc((void**)&d_b, N*sizeof(float));
cudaMalloc((void**)&d_c, N*sizeof(float));
```

Copy Input from Host to GPU

```
cudaMemcpy (d_a, h_a, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy (d_b, h_b, N*sizeof(float), cudaMemcpyHostToDevice);
```

Start GPU Kernel Function

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, d_a, d_b);
cudaDeviceSynchronize();
```

Start GPU Kernel Function

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, d_a, d_b);
cudaDeviceSynchronize();
```

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

Copy Output from GPU to Host

```
cudaMemcpy(h_c, d_c, N*sizeof(float), cudaMemcpyDeviceToHost);
```

Copy Output from GPU to Host

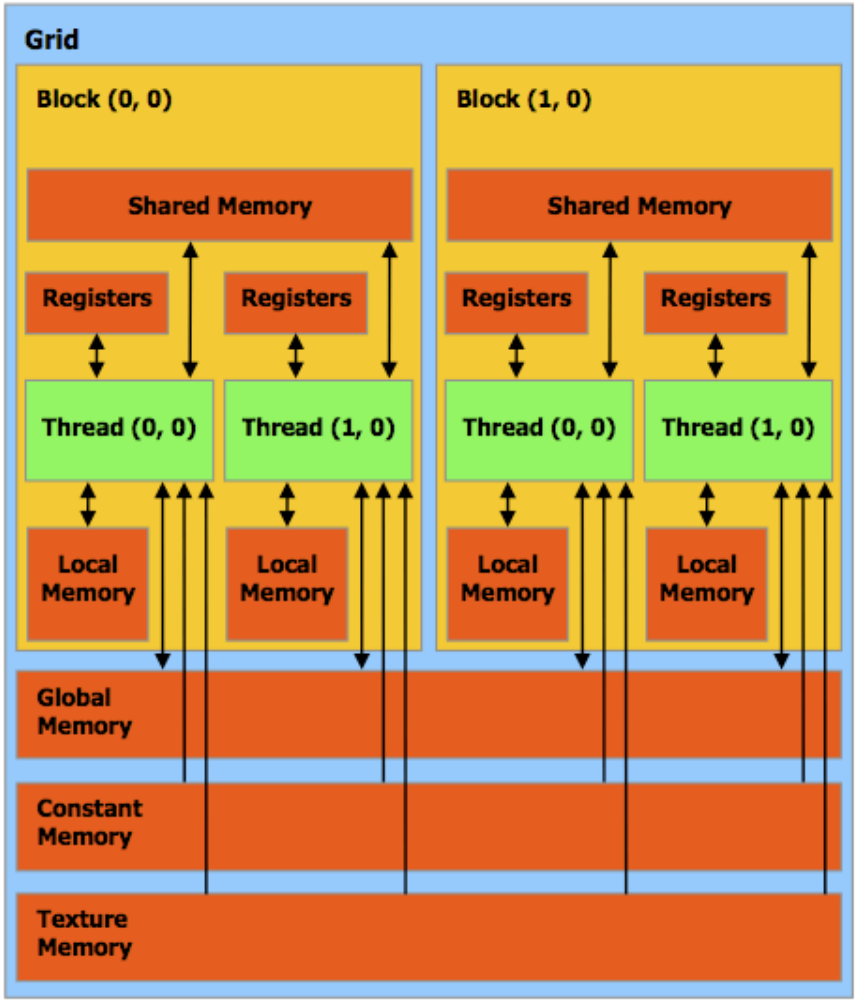
```
cudaMemcpy(h_c, d_c, N*sizeof(float), cudaMemcpyDeviceToHost);

free(h_a);
free(h_b);
free(h_c);
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

C Extensions for GPU Computing

- Straightforward extension to C++
  - Separate CUDA code into .cu and .cuh files and compile with nvcc to create object files (.o files)
- File structures
  - Extension code in .cu/.cuh vs host code in .cpp/.hpp
  - .cu/.cuh is compiled by nvcc to produce a .o file
  - .cpp/.hpp is compiled by g++ and the .o file from the CUDA code is simply linked in using a "#include xxx.cuh" call
    - No different from how you link in .o files from normal C++ code

Thread Block Organization Keywords

- Keywords you MUST know :
  - **Thread** - Distributed by the CUDA runtime (threadIdx)
  - **Block** - A user defined group of 1 to ~512 threads (blockIdx)
  - **Grid** - A group of one or more blocks. A grid is created for each CUDA kernel function called



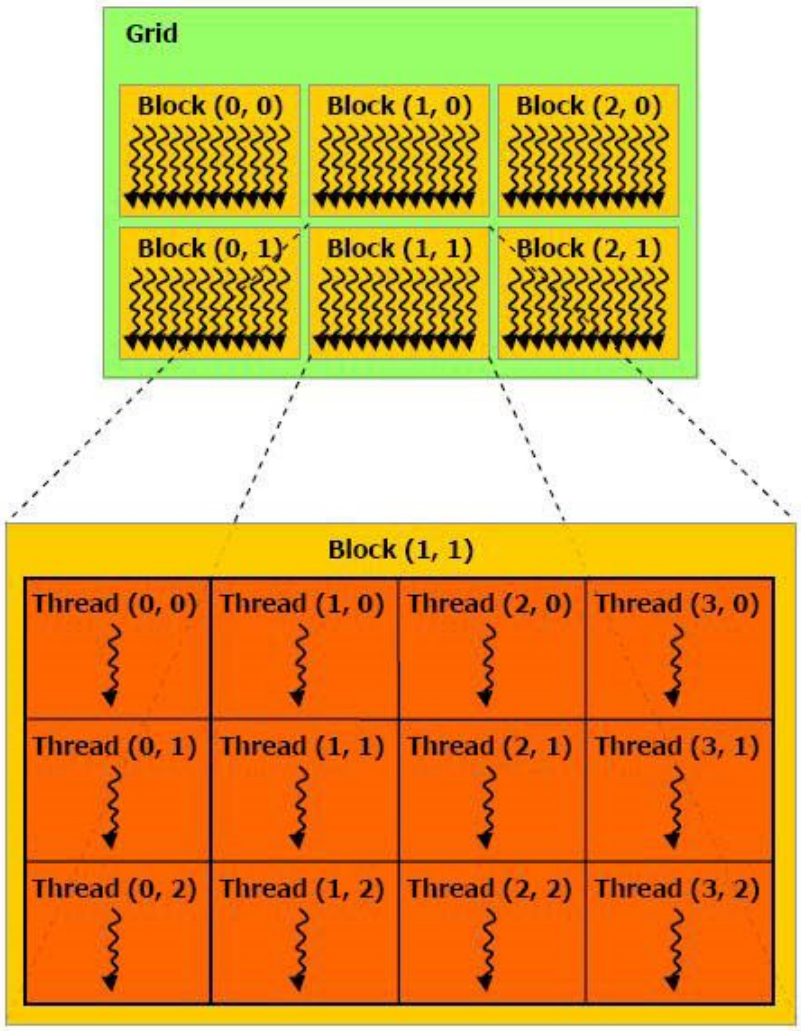
Block and Grid Dimensions

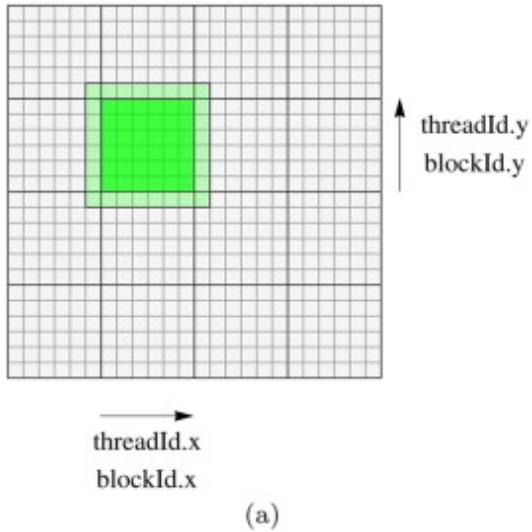
- You can use a struct (defined in vector_types.h) called dim3 to define your Grid and Block dimensions.

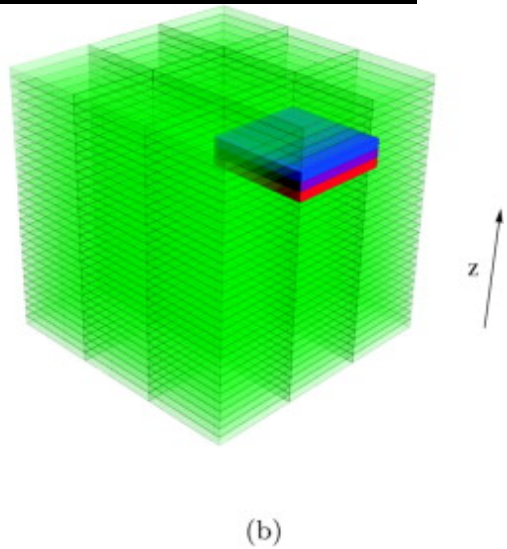
```
struct __device_builtin__ dim3
{
    unsigned int x, y, z;
#if defined(__cplusplus)
    __host__ __device__ dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1) : x(vx), y(vy), z(vz) {}
    __host__ __device__ dim3(uint3 v) : x(v.x), y(v.y), z(v.z) {}
    __host__ __device__ operator uint3(void) { uint3 t; t.x = x; t.y = y; t.z = z; return t; }
#endif /* __cplusplus */
};
```

- dim3 grid(256); // defines a grid of 256 x 1 x 1 blocks
- dim3 block(512, 512); // defines a block of 512 x 512 x 1 threads
- foo<<<grid, block>>>(...);

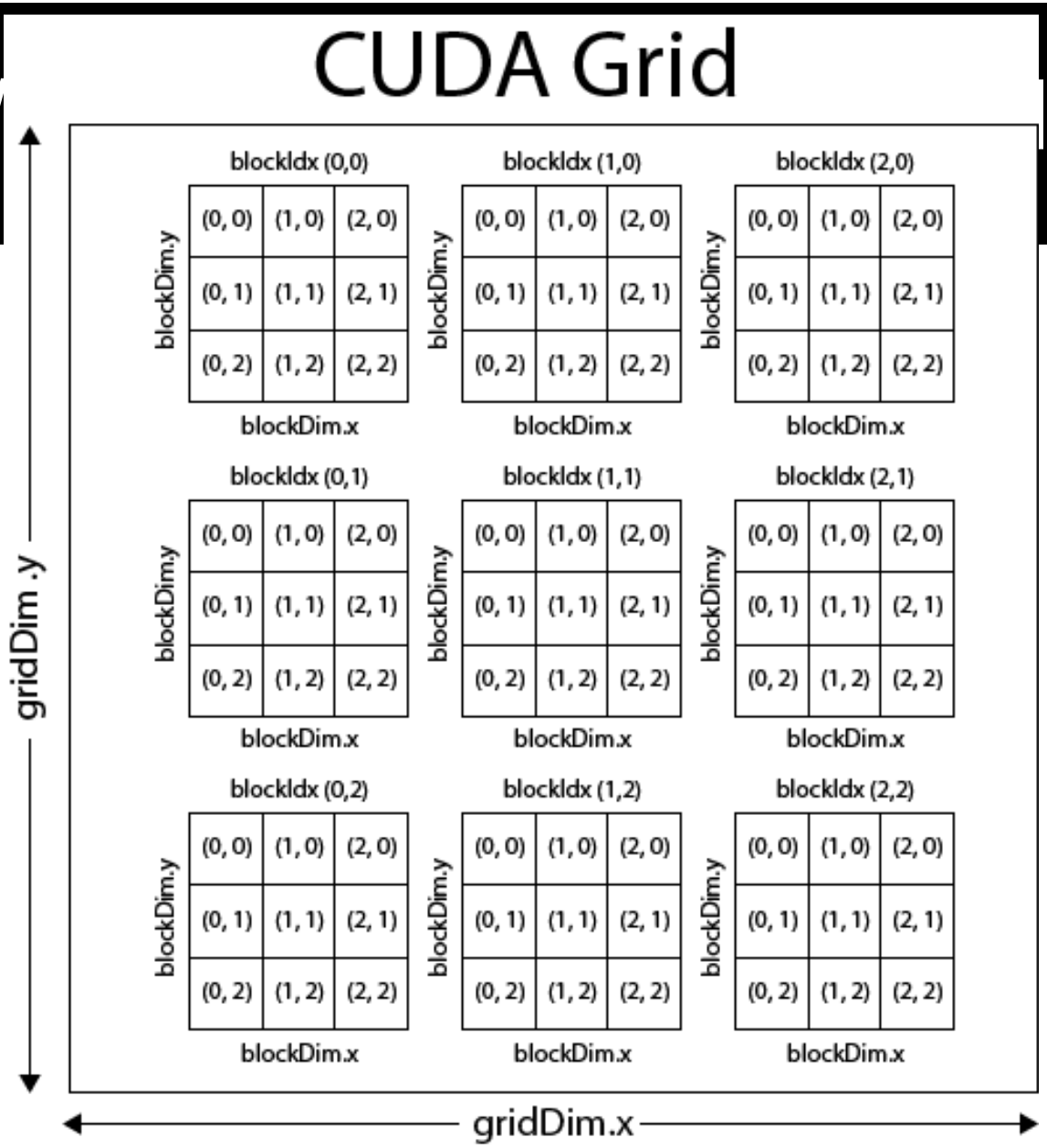
Grid/Block/Thread Visualized







Grid/


CUDA Grid
gridDim .y
gridDim.x
blockIdx (0,0)
blockIdx (1,0)
blockIdx (2,0)
blockIdx (0,1)
blockIdx (1,1)
blockIdx (2,1)
blockIdx (0,2)
blockIdx (1,2)
blockIdx (2,2)
blockDim.y
blockDim.x
(0, 0) (1, 0) (2, 0)
(0, 1) (1, 1) (2, 1)
(0, 2) (1, 2) (2, 2)

More Keywords

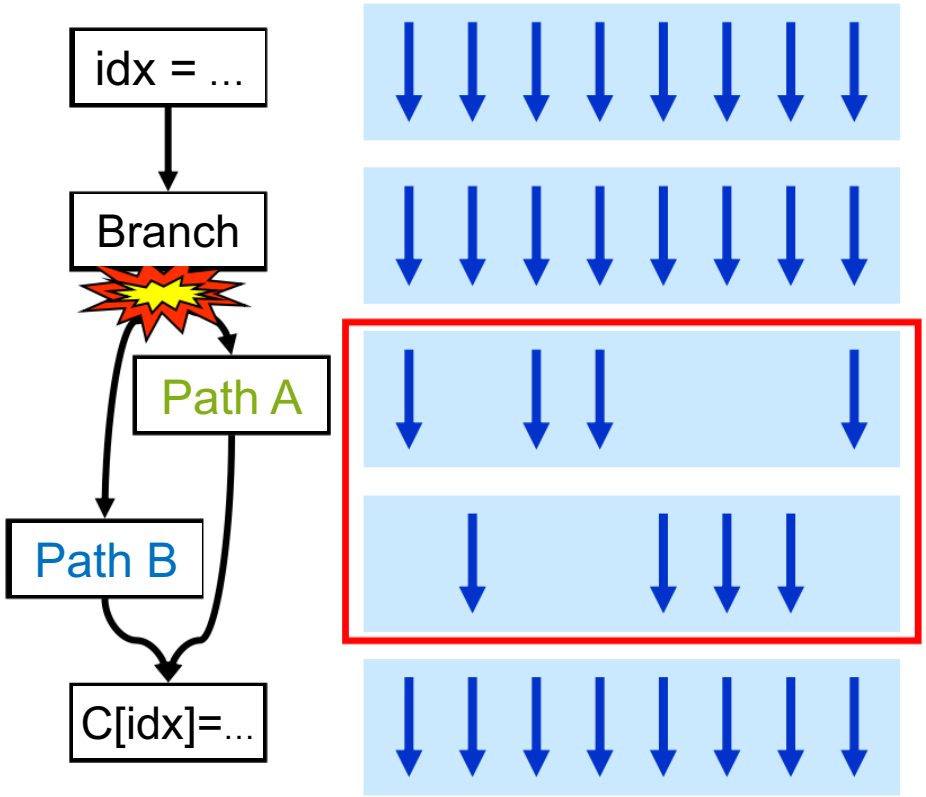
- Streaming Multiprocessor (SM) – contains ~128 CUDA cores (which execute a thread) and their associated cache.
- Warp - scheduling unit of up to 32 threads (all in the same block)
- Warp Divergence – when threads within a warp need to execute different instructions in the kernel.
  - Makes threads to execute sequentially, bad for parallel performance
  - Kepler (2012) architecture 2 branches/warp,

    Volta (2017) * branches/warp.

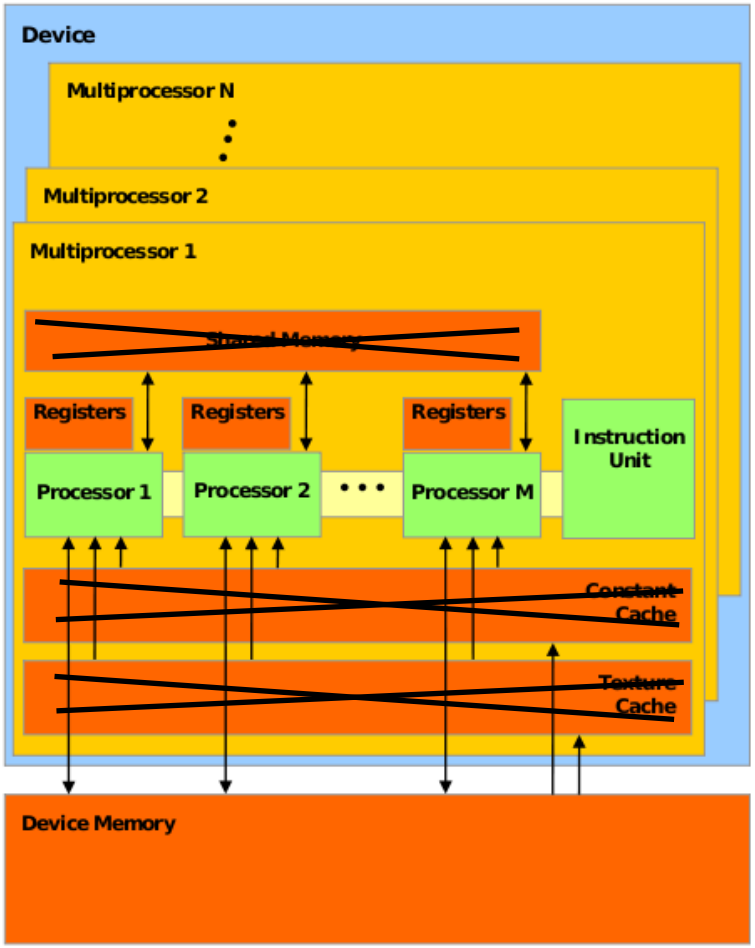
    For this class assume 2 branches/warp

Warp Divergence

```
idx = … // idx computation
if (A[idx] > 42) { // Branch
    // Path A
    B[idx] = A[idx];
} else {
    // Path B
    B[idx] = A[idx]*2;
}
C[idx] = B[idx]+1;
```



Inside a GPU



The black Xs are just crossing out things you don't have to think about just yet. You'll learn about them later

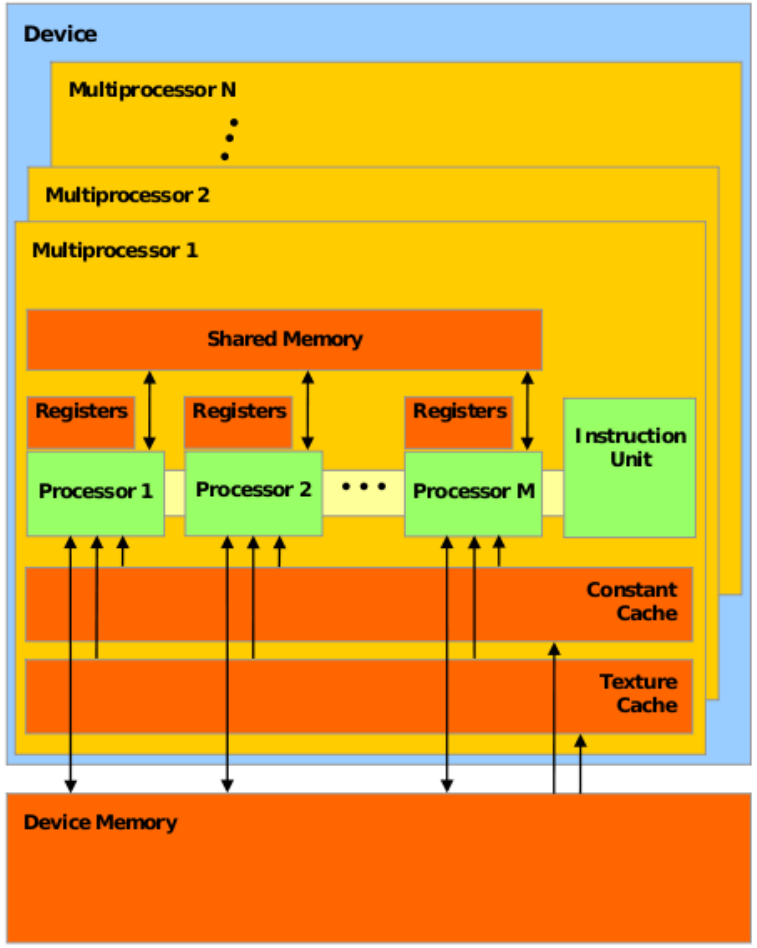
Inside a GPU

Think of **Device Memory** (we will also refer to it as **Global Memory**) as a RAM for your GPU

- Faster than getting memory from the actual RAM but still can be faster
- Will come back to this later

GPUs have many **Streaming Multiprocessors (SMs)**

- Each SM has multiple processors but only one instruction unit
- Groups of processors must run the exact same set of instructions at any given time with in a single SM



Inside a GPU

When a kernel (the thing you define in .cu files) is called, the task is divided up into threads

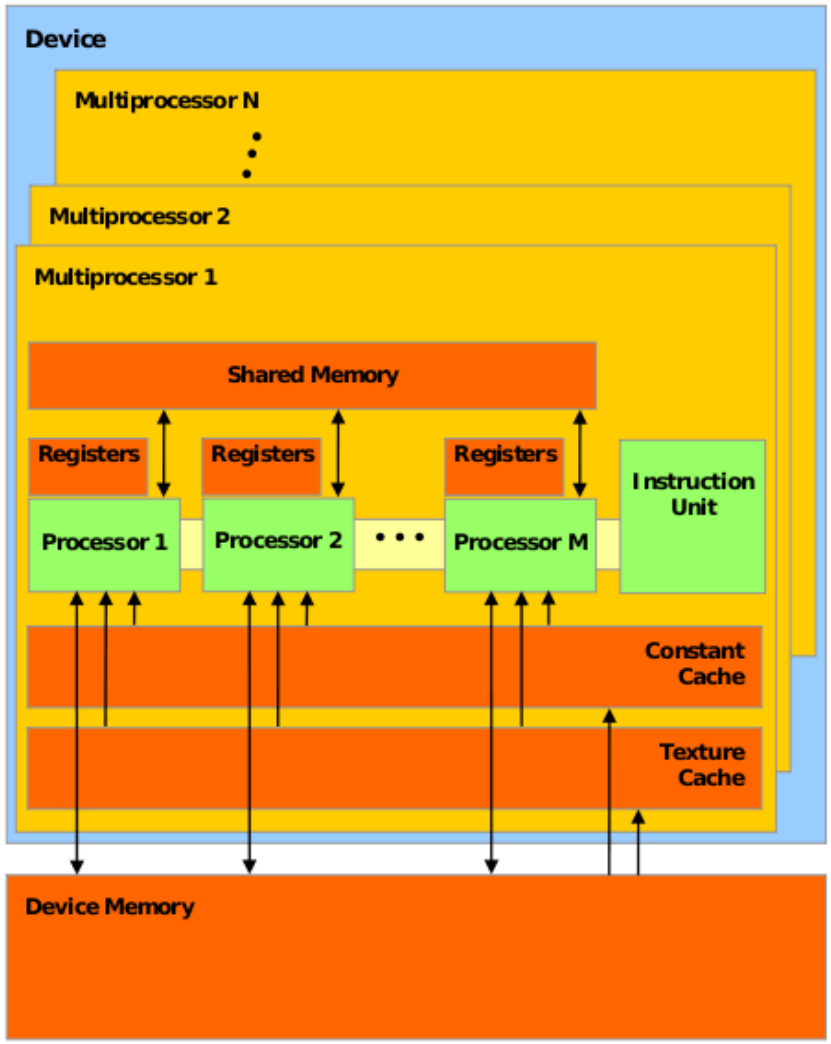
- Each thread handles a small portion of the given task

The threads are divided into a **Grid** of **Blocks**

- Both Grids and Blocks are 3 dimensional
- e.g.

```
dim3 dimBlock(8, 8, 8);
dim3 dimGrid(100, 100, 1);
Kernel<<<dimGrid, dimBlock>>>(...);
```

- However, we'll often only work with 1 dimensional grids and blocks
- e.g. Kernel<<<block_count, block_size>>>(...);

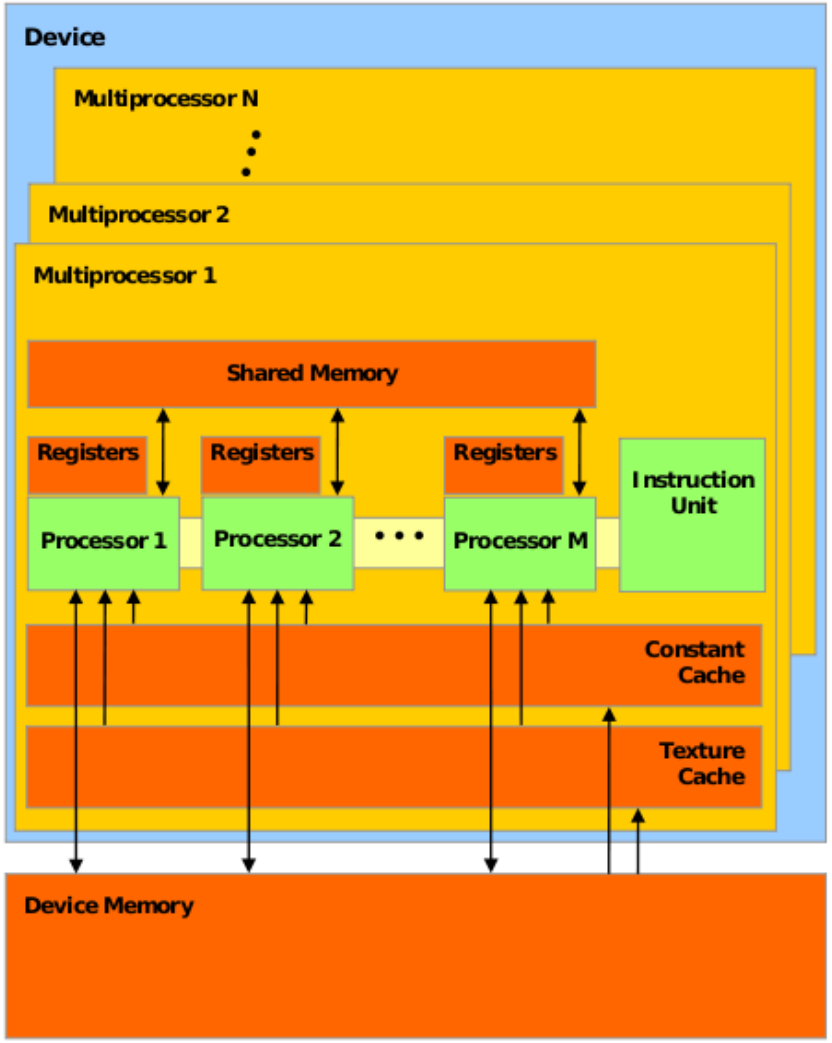


Inside a GPU

Maximum number of threads per block count is usually 512 or 1024 depending on the machine

Maximum number of blocks per grid is usually 65535

- If you go over either of these numbers your GPU will just give up or output garbage data
- Much of GPU programming is dealing with this kind of hardware limitations! Get used to it
- This limitation also means that your Kernel must compensate for the fact that you may not have enough threads to individually allocate to your data points
  - Will show how to do this later

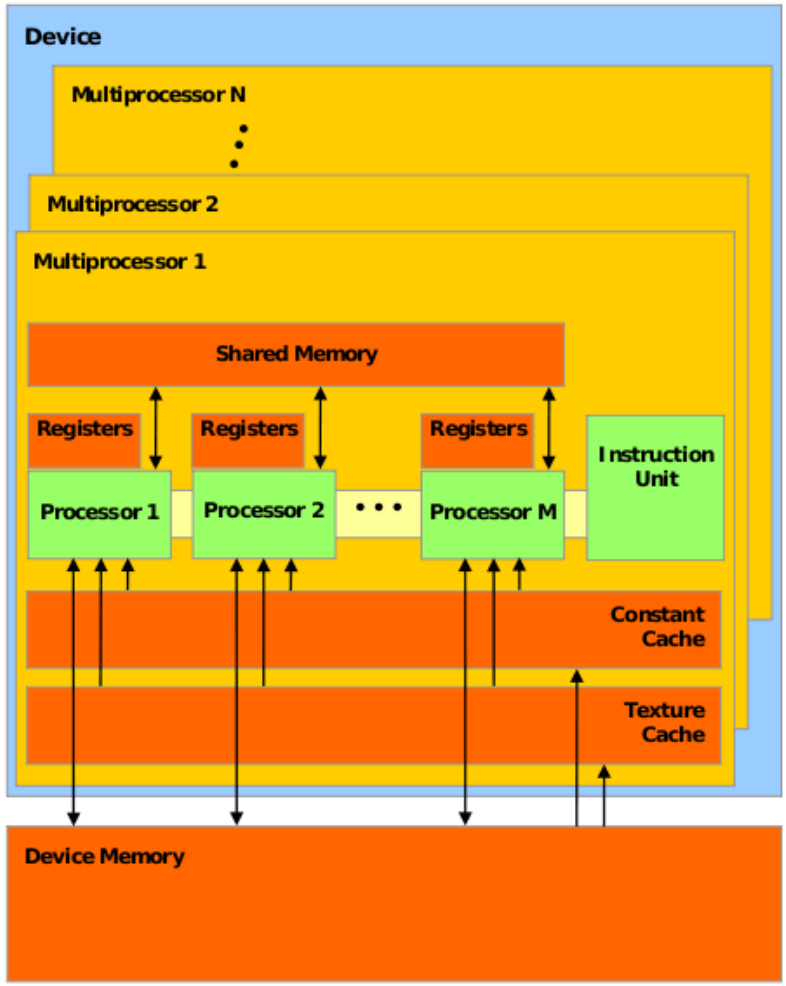


Inside a GPU

Each block is assigned to an SM

Inside the SM, the block is divided into **Warps** of threads

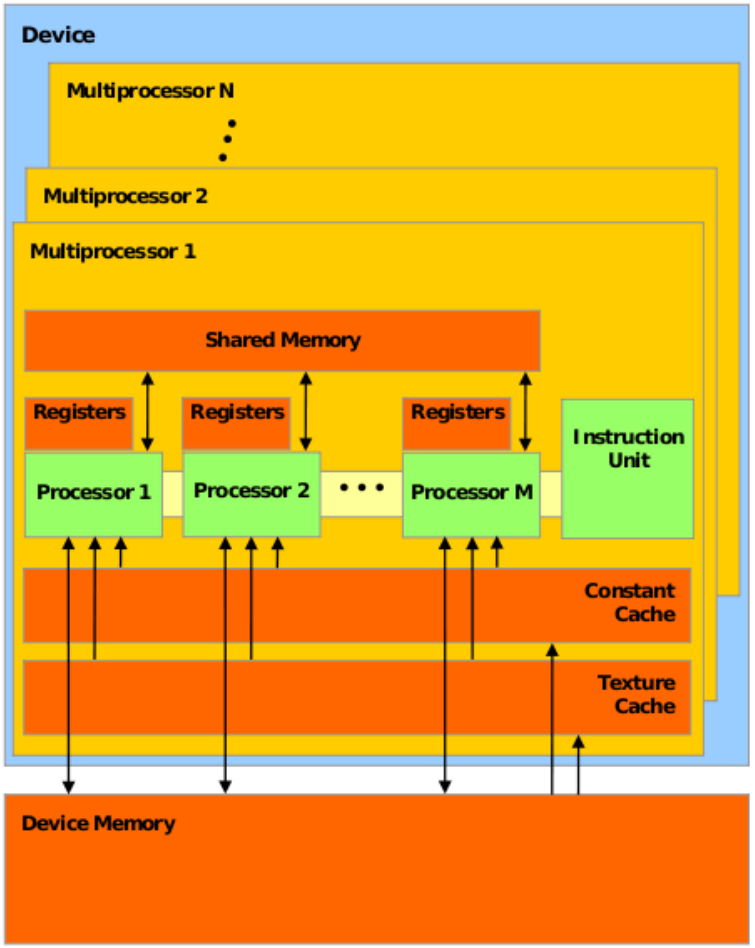
- Warps consist of 32 threads
- All 32 threads MUST run the exact same set of instructions at the same time
  - Due to the fact that there is only one instruction unit
- Warps are run concurrently in an SM
- If your Kernel tries to have threads do different things in a single warp (using if statements for example), the two tasks will be run sequentially
  - Called **Warp Divergence** (NOT GOOD)



Inside a GPU (more hardware info)

In Fermi Architecture (i.e. GPUs with Compute Capability 2.x), each SM has 32 cores, later architectures have more.

- e.g. GTX 400, 500 series
- 32 cores is not what makes each warp have 32 threads. Previous architecture also had 32 threads per warp but had less than 32 cores per SM
- Some early Pascal (2016) GPUs (GP100) had 64 cores per SM, but later chips in that generation (GP104) had 128 core model.



Streaming Multiprocessor

- Shown here is a Pascal GP104 GPU Streaming Multiprocessor that can be found in a GTX1080 graphics card.
- The exact amount of Cache and Shared Memory differ between GPU models, and even more so between different architectures.
  - Whitepapers with exact information can be gotten from Nvidia (use Google)
    - https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf
    - http://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf
  - “nvidia kepler whitepaper”