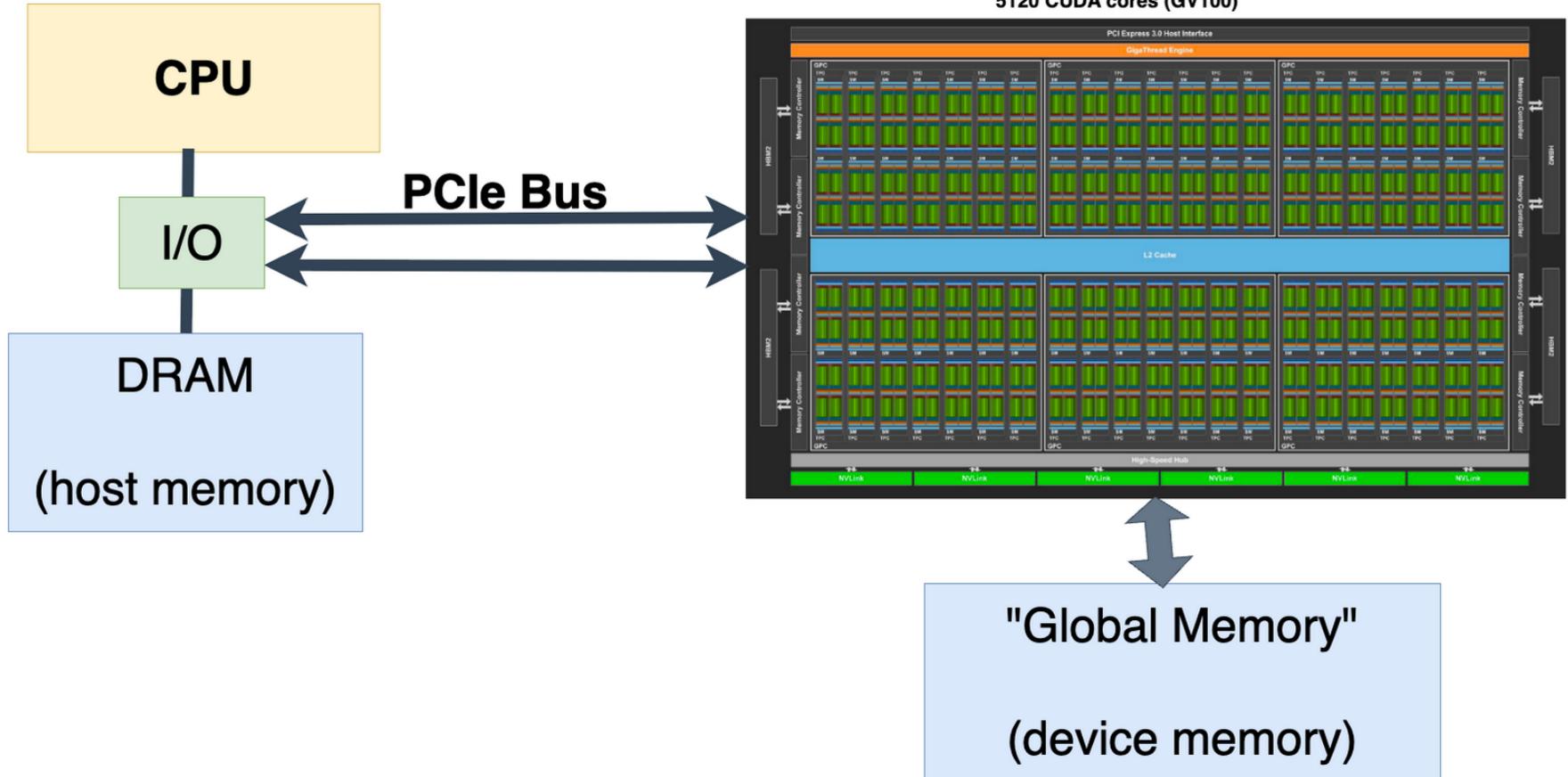


Lecture 3: Introduction to CUDA 1

CS599: Programming Massively Parallel Multiprocessors and Heterogenous Systems
(Understanding and programming the devices powering AI)

Jonathan Appavoo

Recall: CPU/GPU Organization



CUDA

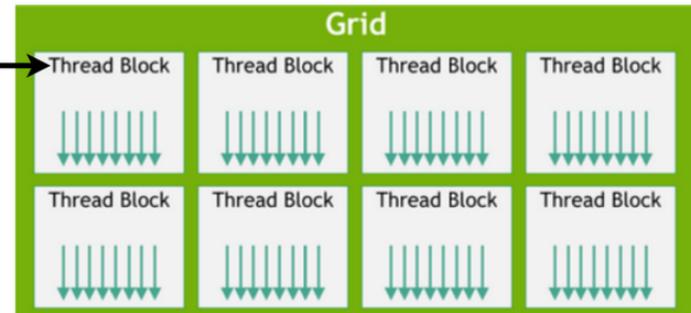
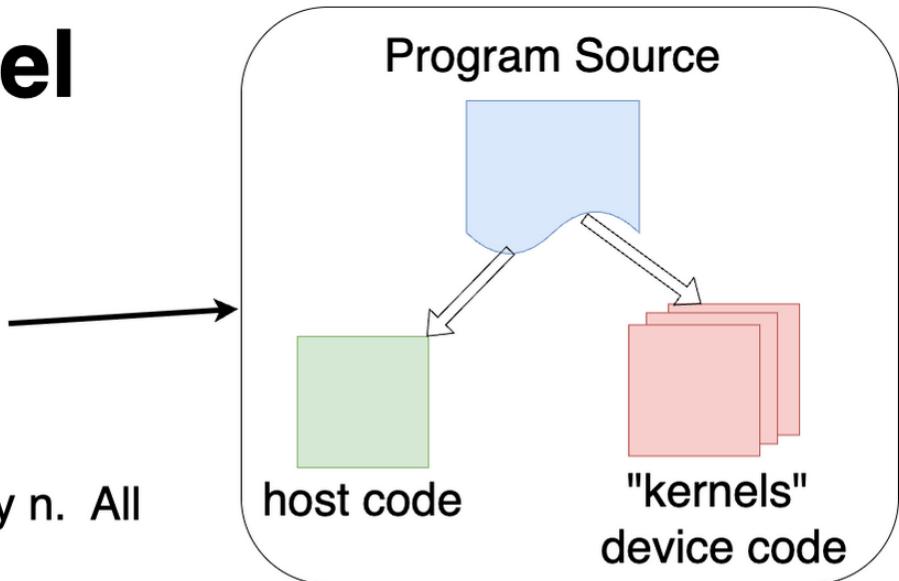
CUDA (Compute Unified Device Architecture) is NVIDIA's programming development environment.

- based on C/C++ with some extensions
- FORTRAN support provided through NVIDIA's PGI toolkit
- Python supported through CUDA Python and libraries like Numba, PyCUDA
- Third-party bindings to other languages. E.g. Java, .NET, RUST, ...
- lots of examples, including an NVIDIA-provided collection of samples.
- NVIDIA provided documentation
- large user community on NVIDIA forum, and stack-overflow

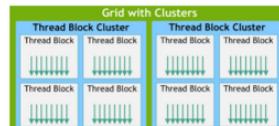
Competitors: AMD's ROCm/HIP, Intel's oneAPI, open standards OpenCL, SYCL, and Vulkan Compute.
There are also AI-specific alternatives (eg, OpenAI Triton, etc)

CUDA Software: High-level Overview

- CUDA program has two parts
 1. code that executes on the CPU (host)
 2. "**kernel**" that executes on GPU (device)
- A kernel is executed by n threads, where you specify n. All threads run the same code.
- The n threads are grouped into "**thread blocks**" of max 1,024 threads, specified by the programmer. The collection of blocks is called a "**grid**"
- Each thread block executes within an SM
- Threads of a thread block are automatically/transparently divided into **warp**s (*groups of 32 threads*) and execute in "lockstep"



NVIDIA Recently (in Compute Cap 9.0) added another level of organization "Thread Block Cluster" we are going to ignore this



CUDA Software: High-level Overview (cont.)

- CPU and kernel code written in C
- CUDA API to
 - control kernel execution
 - manage GPU memory
 - transfer data between memories
 - synchronization
 - error handling
- Developer responsibility: partition problem into many subproblems, each solved by a thread.
 - You will want many more threads than available cores
 - each thread executes the same kernel code.
 - Threads must be partitioned into thread blocks

Hello World I

hello.cu `#include <stdio.h>`

```
__global__ void helloFromGPU(void)
{
    printf("Hello from GPU\n");
}

int main(void)
{
    printf("Hello from CPU\n");

    // other code later
}
```

specifies function will be called from CPU and executed on GPU

Kernel:

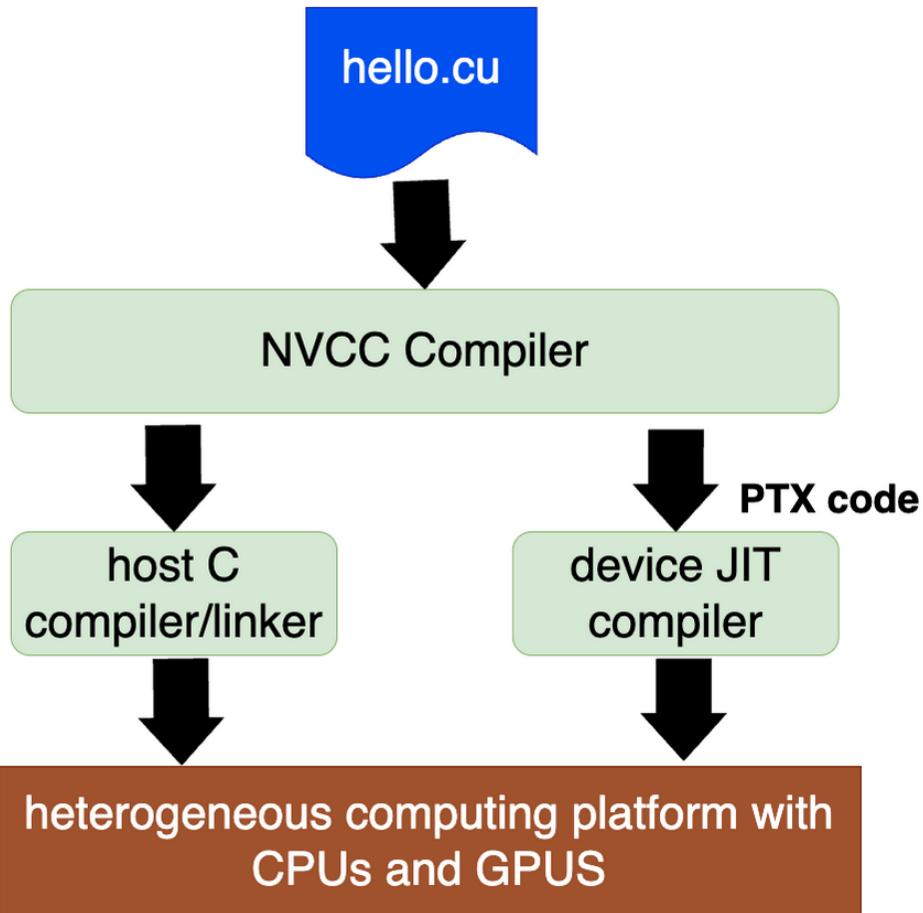
A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new `<<<...>>>` execution configuration syntax (see Execution Configuration). Each thread that executes the kernel is given a unique `thread ID` that is accessible within the kernel through built-in variables.

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels>

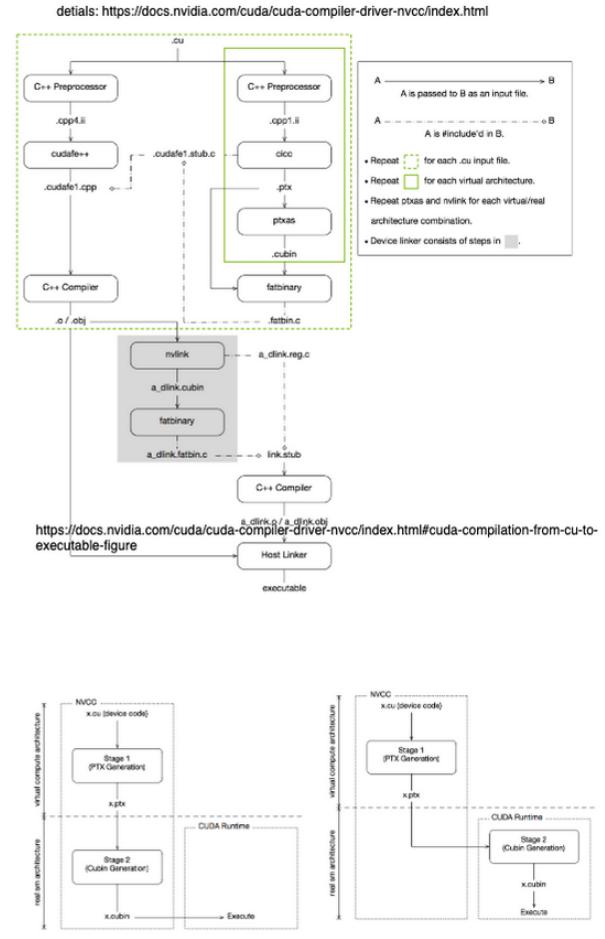
compile: \$ nvcc -o hello hello1.cu
run: \$./hello
Hello from CPU

output: "Hello from CPU"

CUDA Compilation



Programming Guide has nice overview of compilation flow and runtime
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compilation-with-nvcc>



<https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#virtual-architectures-virtual-architectures>

<https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#just-in-time-compilation-just-in-time-compilation>

Hello World II

```
__global__ void helloFromGPU(void)
{
    printf("Hello from GPU\n");
}

int main(void)
{
    printf("Hello from CPU\n");
    helloFromGPU <<< 1, 10 >>>();
}
```

Call from CPU thread to
GPU code

Call is asynchronous

Specifies # & structure of threads.

In this case 1 block of 10 threads

Hello World III

```
__global__ void helloFromGPU(void)
{
    printf("Hello from GPU\n");
}

int main(void)
{
    printf("Hello from CPU\n");
    helloFromGPU <<< 1, 10 >>>();
    cudaDeviceSynchronize();
}
```

Waits for all GPU threads to complete.

(Needed because call to GPU kernels are asynchronous)

Hello World

```
#include <stdio.h>
```

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global>

```
__global__ void helloFromGPU(void)
```

```
{
```

```
    printf("Hello from GPU\n");
```

```
}
```

```
int main(void)
```

```
{
```

```
    printf("Hello from CPU\n");
```

```
    helloFromGPU <<< 1, 10 >>>();
```

```
    cudaDeviceSynchronize();
```

```
    cudaDeviceReset();
```

```
}
```

Explicitly destroys and cleans up all resources. Not necessary at the end but so that you know.

Hello World

```
compile: $ nvcc -gencode arch=compute_70,code=sm_70 hello.cu -o hello  
run:   $ brun ./hello
```

Hello from CPU

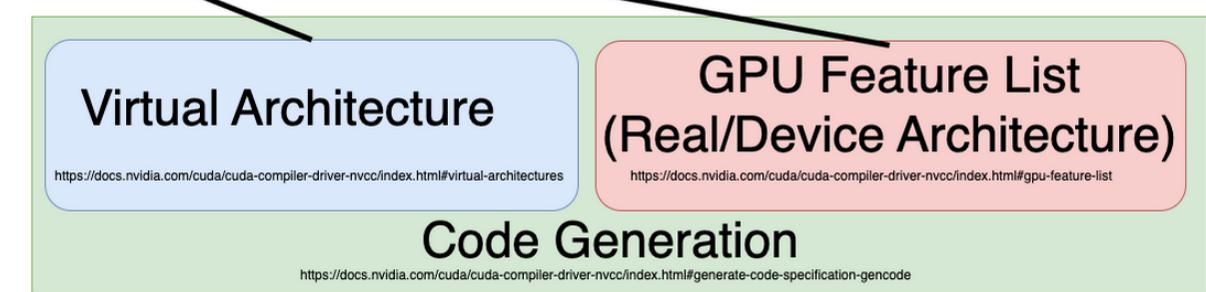
Hello from GPU

RUNDIR: jobs/job-v100-80058

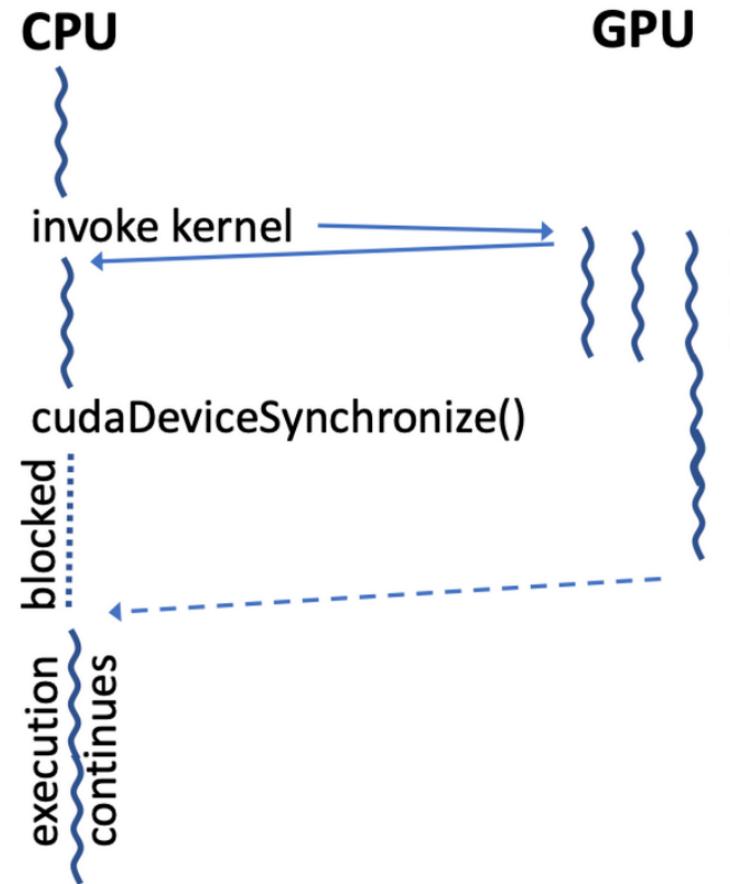
\$

To have the compiler generate code for our specific GPU we must add compilation flags.

For GV100: '-gencode arch=compute_70,code=sm_70'



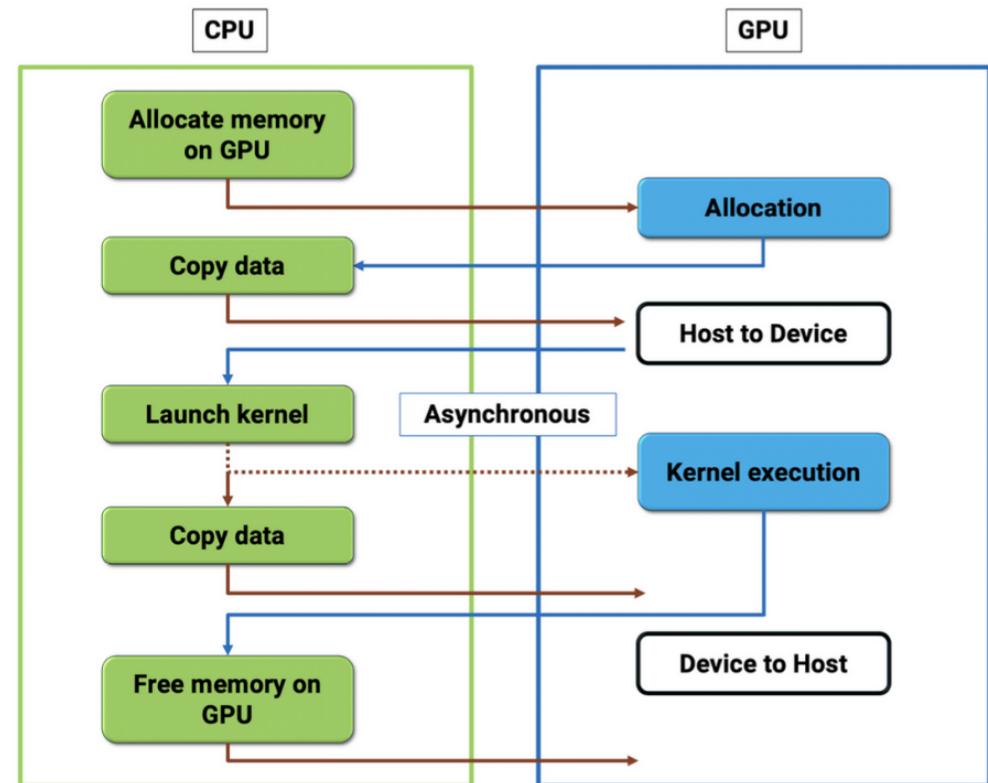
Thread Behaviour



More General CUDA Program Structure

Six Steps:

1. allocate GPU memory
2. copy data from CPU memory to GPU memory
3. launch (invoke) kernel
4. synchronize
5. copy data back from GPU to CPU
6. cleanup



Managing Memory

Standard C (host)	CUDA C (device)
malloc()	cudaMalloc()
memcpy()	cudaMemcpy()
memset()	cudaMemset()
free()	cudaFree()

E.g.:

```
cudaError_t cudaMalloc( void **devPtr, size_t size );  
cudaError_t cudaFree ( void * devPtr );
```

GPU address space ptr

in bytes

Memory Copies

```
cudaError_t cudaMemcpy( void *dest, void *src,  
                      size_t count, enum cudaMemcpyKind kind );
```

where kind is one of:

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice
- cudaMemcpyHostToHost

Error Checking

All CUDA functions (except kernel launches) return error code of enumerated type `cudaError_t`

- e.g., `cudaSuccess`
- convert error code to text with
`char* cudaGetString(cudaError_t error)`

```
cudaError_t err = cudaSomeFCT( ... );
if( err != cudaSuccess ) {
    printf( "%s in %s at line %d\n", cudaGetString( err ),
        __FILE__, __LINE__ );
    exit( EXIT_FAILURE );
}
```

Pointer Naming Convention

Because we are dealing with two address spaces, with pointers one can easily get confused what address space they are pointing to.

→ common to use convention

- prepend names of pointers within GPU memory with “d_”
e.g.,

`d_devPtr`

- prepend names of pointers within CPU memory with “h_”.

Parallel Programming

- CUDA exposes the hardware to the programmer
- Primary task of programmer
 - manually partition work into threads organized into blocks to exploit the hardware

Where to find parallelism I

One way to look at it

- think of computation as series of loops:

```
for (i=0; i<big_number; i++)
    a[i] = some function
```

```
for (i=0; i<big_number; i++)
    a[i] = some other function
```

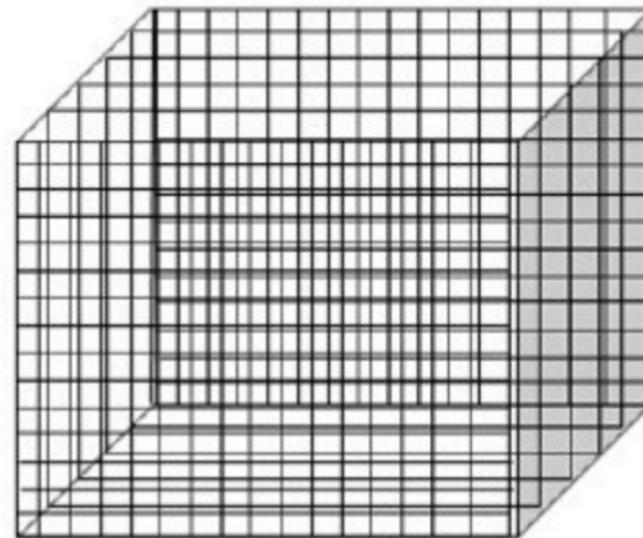
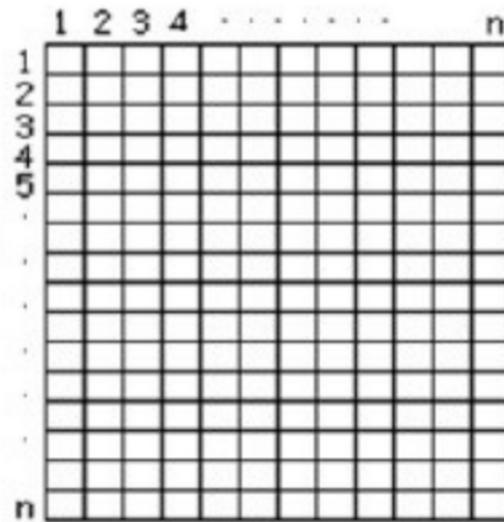
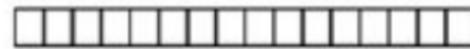
```
for (i=0; i<big_number; i++)
    for (j=0; j<other_big_number; j++)
        a[i,j] = yet some other function
```

Each
assignment a
thread?

Where to find parallelism II

Data is in the form of an array, matrix, or a cube

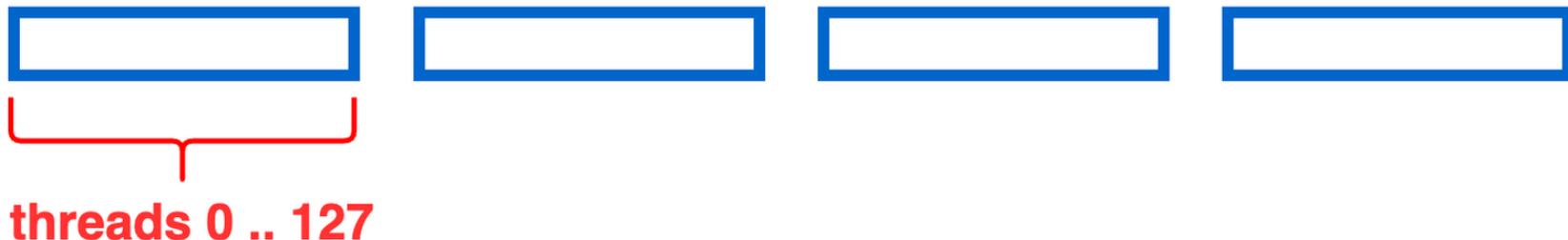
- think of computation as series of loops:



Example: Array processing

Consider: `for (i=0; i<512; i++)
 a[i] = some_func(...);`

- Idea: assign a thread to each array element
 - partition threads into 4 blocks



Invoke kernel as:

`kernel_routine<<<4, 128>>>(...args...);`

More generally: Execution Configuration

definition: `__global__ void func(...args...) { ... }`

`func<<< Dg, Db, Ns, S >>> (...args...)`

- **Dg** : grid size -- size & structure of blocks : most generally a 3 dimensional:
 $\#blocks = Dg.x * Dg.y * Dg.z$
- **Db** : block size -- size & structure of threads in a block : most generally a 3 dimensional: $\#threads \text{ in block} = Db.x * Db.y * Db.z$
- **Ns** : optional number of bytes in shared memory dynamically allocated per block in addition to static amount (default is 0)
- **S** : associated cuda stream (default is 0) -- advanced ability to launch multiple kernels
- args : simple types: no references, function pointers, complex types, ...

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=kernel%2520function%2520parameters#function-parameters>

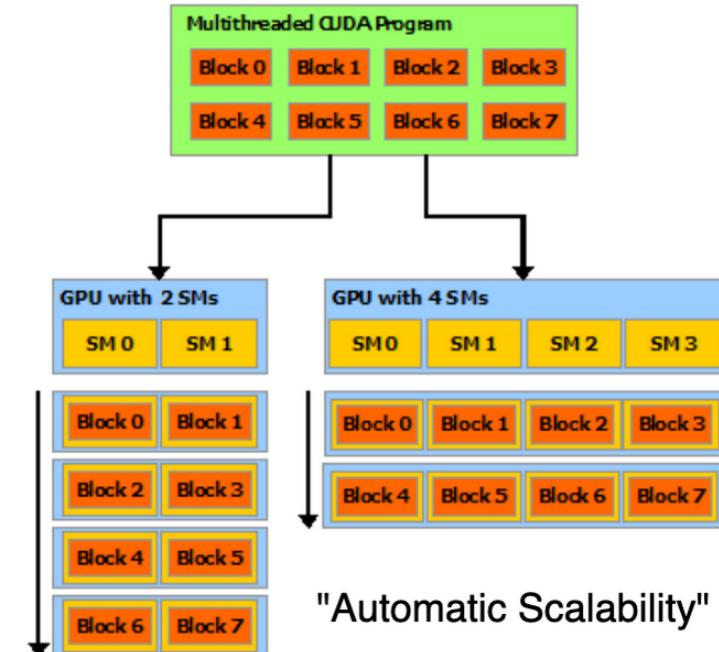
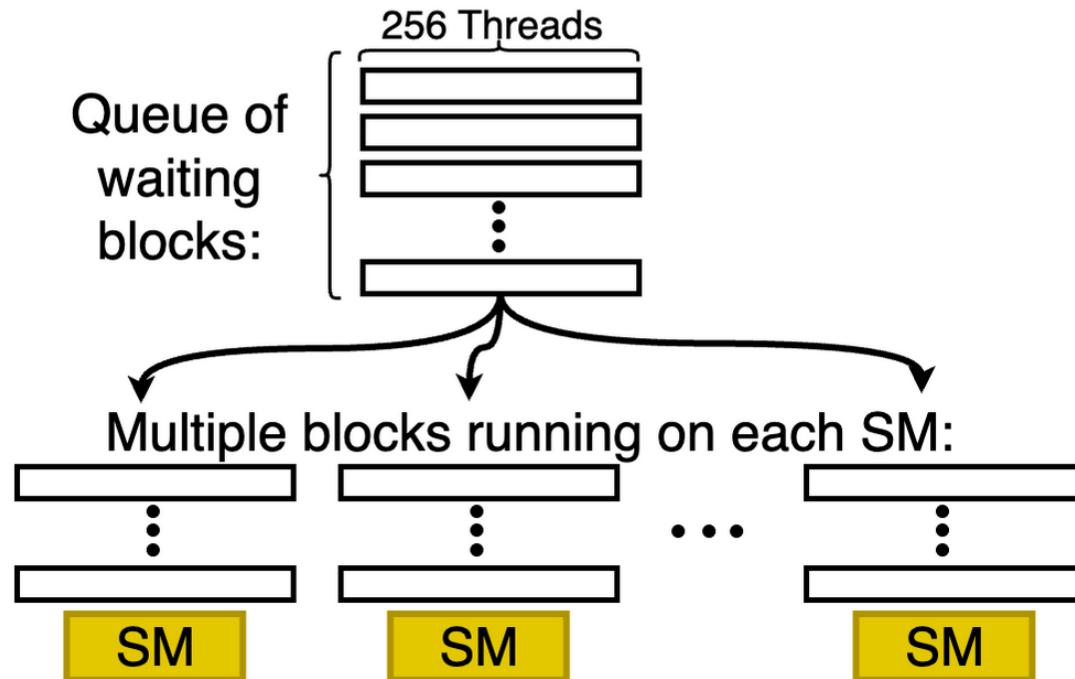
Invoke kernel as [simple case of Dg=(4,1,1) & Db=(128,1,1)]

`kernel_routine<<<4, 128>>>(...args...);`

Thread and Block Dispatching I

Suppose we have specified 1000 blocks of 256 threads (=256000 threads).
How do blocks get dispatched?

- Assuming a V100 with 80 SM's each capable of "processing" 2048 threads (8 blocks) at once implies that 640 blocks can processed at once -- we have specified 360 more than this.



Thread and Block Dispatching II

SM decomposes Thread Blocks into warps of 32 threads -- **deterministically** in groups of consecutive thread id: Eg each 256 block defines 8 warps

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=hello#sm1t-architecture>

- At every instruction issue time, an SM warp scheduler selects a warp that has threads "ready" to execute its next instruction and issues the instruction to those threads
 - selects from those threads not waiting for
 - data coming from device memory (memory latency)
 - from prior instructions (pipeline delay -- register dependencies)
 - memory fence or synchronization (more warps of the same block don't help --- need more blocks or better yet try and eliminate)

Programmer doesn't have to worry about this level of detail -- just make sure there are lots of threads / warps (read "**Multiprocessor Level**" for details of above CPG)

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=hello#hardware-multithreading>

Note: you cannot make any assumptions about the ordering of execution of threads beyond those in the same warp

What a thread knows about / has access to

- Passed-in arguments, including pointers to data in global memory
- global constants and variables in device memory
- shared memory and private registers/local variables
- some special Built-in variables, key ones are:
 - **gridDim** size (& dimensions) of grid of blocks
 - **blockDim** size (& dimensions) of each block of threads
 - who am I
 - **blockIdx** index (or 2D/3D indices) of block in grid
 - **threadIdx** index (or 2D/3D indices) of thread in block
 - **warpSize** always 32 so far, but could change

Only threads within a block can share data in shared memory and synchronize!

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=hello#thread-hierarchy>

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=gridDim#built-in-variables>

How a thread determines what to computer

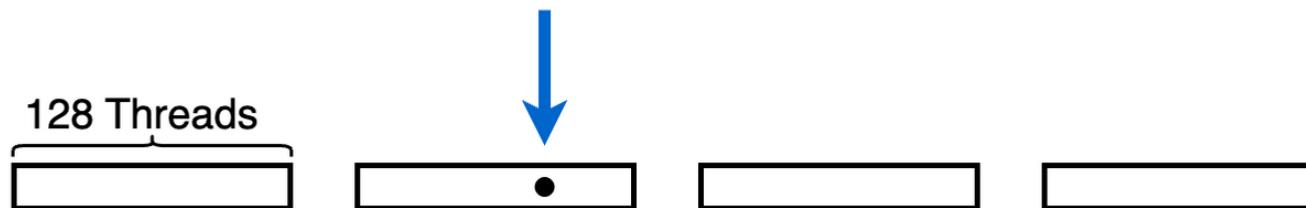
Each thread needs to determine what to compute.

Standard way to do this : map id to data -- there is locality in ids --> map this too!

ID mapping example: 1D grid with 4 blocks each with 128 threads:

- **gridDim = 4**
- **blockDim = 128**
- **blockIdx.x = 0 .. 3.** [dim3 (x,y,z)]
- **threadIdx = 0 .. 127.** [dim3 (x,y,z)]

blockIdx.x = 1, threadIdx.x = 100



Kernel code

```
__global__ void kernel_routine(float *x)
{
    int tid = threadIdx.x +
              blockDim.x * blockIdx.x;

    x[tid] = some_function( args );
}
```

Note here:

- each thread sets one element of the x array (assumes some args)
- each thread has a unique value for tid (element in set 0 .. 512)

Host code

```
int main(int argc, char **argv) {
    cudaError_t rc;
    float *h_x, *d_x; // h=host, d=device
    int nblocks=4, nthreads=128;
    int nsize = nblocks * nthreads;

    h_x = (float *) malloc(nsize * sizeof(float));
    rc = cudaMalloc((void **)&d_x, nsize*sizeof(float));
    assert(rc == cudaSuccess);

    filldata(h_x, nsize);[]
    rc = cudaMemcpy( d_x, h_x, nsize*sizeof(float), cudaMemcpyHostToDevice );
    assert(rc == cudaSuccess);

    kernel_routine<<<nblocks,nthreads>>>( d_x );

    rc = cudaMemcpy( h_x, d_x, nsize*sizeof(float), cudaMemcpyDeviceToHost );
    assert(rc == cudaSuccess);

    for (int n=0; n<nsize; n++) printf("%d,%f\n",n, h_x[n]);

    cudaFree(d_x); free(h_x);
}
```

Host code

```
int main(int argc, char **argv) {
    cudaError_t rc;
    float *h_x, *d_x; // h=host, d=device
    int nblocks=4, nthreads=128;
    int nsize = nblocks * nthreads;

    h_x = (float *) malloc(nsize * sizeof(float));
    rc = cudaMalloc((void **)&d_x, nsize*sizeof(float));
    assert(rc == cudaSuccess);

    filldata(h_x, nsize);  
    rc = cudaMemcpy( d_x, h_x, nsize*sizeof(float))
    assert(rc == cudaSuccess);

    kernel_routine<<<nblocks,nthreads>>>( d_x );
    rc = cudaMemcpy( h_x, d_x, nsize*sizeof(float), cudaMemcpyDeviceToHost );
    assert(rc == cudaSuccess);

    for (int n=0; n<nsize; n++) printf("%d,%f\n",n, h_x[n]);
}

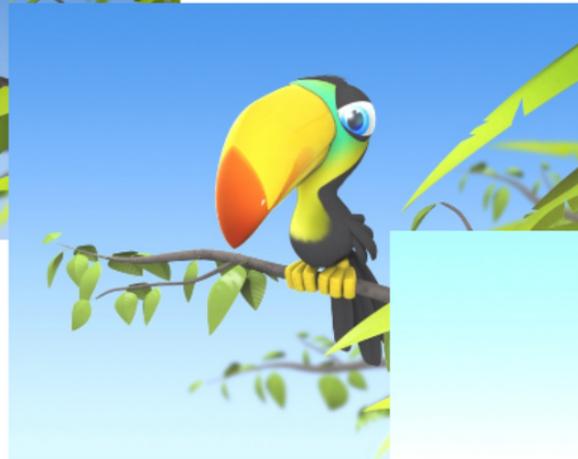
cudaFree(d_x); free(h_x);
}
```

Sync not needed
due to implicit sync between cuda calls (on same stream)

cudaDeviceReset();

2D block & thread structure

Sample, trivially parallelizable app with 2D structure:

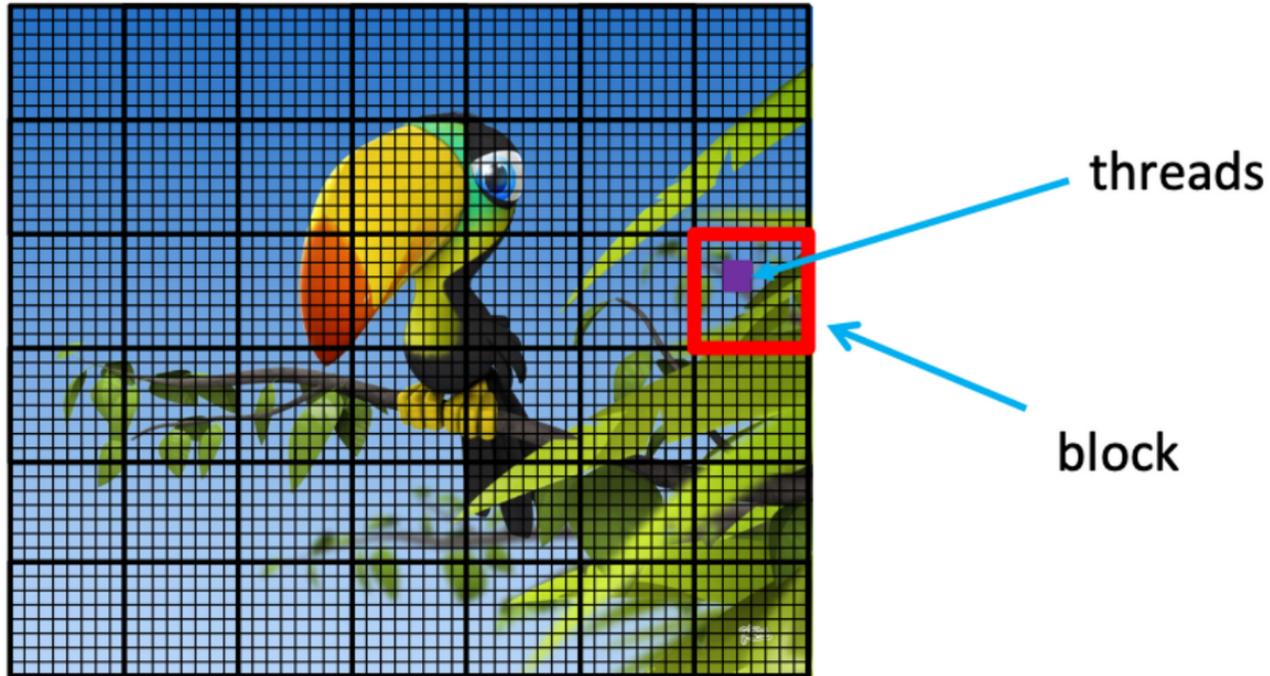


2D block & thread structure

Each thread processing one pixel:

for all elements do in parallel

```
a[i] = a[i] * fade;
```



Other mappings are possible and often desirable.
More on this when we talk about how to optimize for performance

Code Skeleton

CPU

- Initialize image from file
- Allocated IN and OUT buffers on GPU
- Copy image to IN buffer on GPU
- Launch GPU kernel
 - Reads IN
 - Produces OUT
- Copy OUT back to CPU
- Write image to a file

GPU

- Launch a thread per pixel

Kernel Pseudo-code

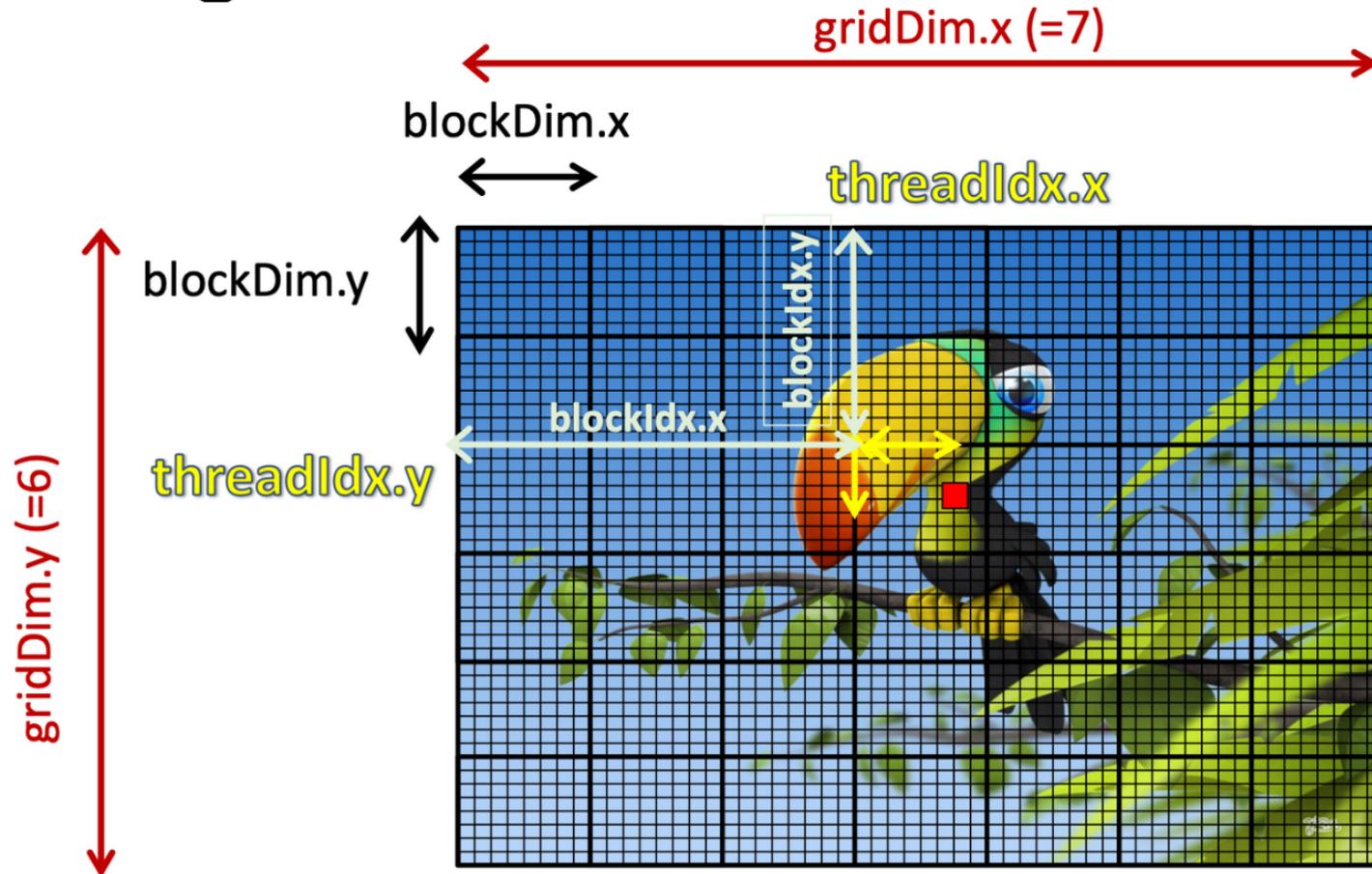
where each thread processes one pixel:

```
__global__ void fade( unsigned char *d_in, unsigned char *d_out,
                      float f, int xmax, int ymax) {
    unsigned int idx, v;

    // code to determine x and y for thread goes here

    idx = y * xmax + x;
    v = d_in[idx] * f;
    if (v>255) v = 255;
    d_out[idx] = v;
}
```

Determining which thread does which pixel



$$x = blockIdx.x * blockDim.x + threadIdx.x$$
$$y = blockIdx.y * blockDim.y + threadIdx.y$$

Kernel Pseudo-code

where each thread processes one pixel:

```
__global__ void fade( unsigned char *d_in, unsigned char *d_out,
                      float f, int xmax, int ymax) {
    unsigned int idx, v;
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;

    if ((x >= xmax) || ( y >= ymax)) return;

    idx = y * xmax + x;
    v = d_in[idx] * f;
    if (v>255) v = 255;
    d_out[idx] = v;
}
```

Kernel Code

Kernel code looks fairly normal once you get used to it:

- code written from the point of view of a single thread
 - all local variables are private to that thread
- must be of type void
- need to think about where each variable lives (more on this in future lectures)
 - any operation involving data in the device memory forces its transfer to/from registers in the GPU (albeit some support for caching now)
 - often better to copy the value into a local register (or shared memory)

There are C restrictions eg. limited stdlib, limited recursion and function pointer support, no variable length arrays, smalls stack

fade kernel launch (host code):

```
dim3 nblocks( 7, 6 );
dim3 nthreads( 16, 16 );
fade<<<nblocks,nthreads>>>(d_in, d_out, f, xmax, ymax);
```

where dim3 is a special CUDA data with 3 components .x, .y, .z, each initialized to 1 by default.

Warps

How do 2D / 3D threads get divided into warps?

- 1D thread ID defined by

2D: $\text{threadIdx.x} + \text{threadIdx.y} * \text{blockDim.x}$

3D: $\text{threadIdx.x} + \text{threadIdx.y} * \text{blockDim.x} + \text{threadIdx.z} * \text{blockDim.x} * \text{blockDim.y}$

and this is then broken up into warps of size 32 within a block (technically 32 is system dependent)

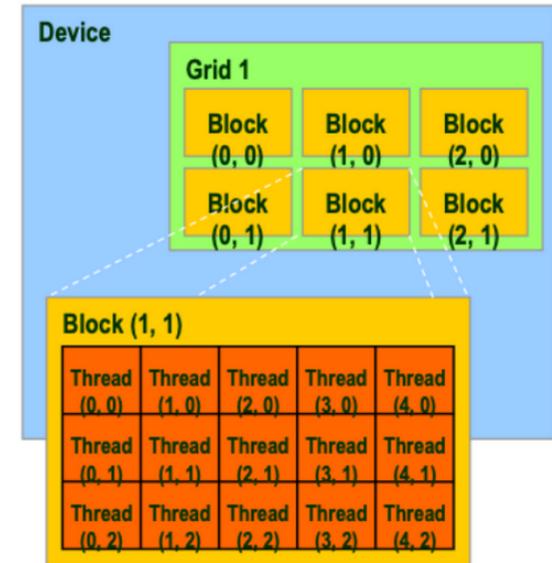
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy>

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#sm1t-architecture>
The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.

Grids of Thread Blocks: Dimension Limits

- Grids of Thread Blocks: **1D, 2D, or 3D**
 - Max x-dimension of a grid of thread blocks: **$2^{31}-1$**
 - Max y- or z-dimension of a grid of thread blocks: **65535**
- Thread Blocks: **1D, 2D, or 3D**
 - Max number of threads per thread block: **1024**
 - Max x- or y-dimension of a thread block: **1024**
 - Max z-dimension of a thread block: **64**

More generally limits are device "compute capability" specific: see "Technical Specification per Compute Capability" in docs and use API to query for your device.



READINGS

[CPG] CUDA Programming Guide (online)

- READ: Chapter 1 - 5
- READ: Formatted Output (`printf`) -- note Limitations <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=hello#formatted-output>
- Chapter 10: Language Extensions -- READ: 10.1 and 10.2 -- rest can be used as reference -- know that this info exists
- Be familiar with how devices map to compute capability -- Chapter 9: CUDA-enabled GPUs (see "Useful Facts" slide)
- We will cover Chapter 8: Performance Guidelines in our Performance Optimization

Useful facts

Feature Support per Compute Capability

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=hello#features-and-technical-specifications-feature-support-per-compute-capability>

*** Technical Specification per Compute Capability ***

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=hello#features-and-technical-specifications-technical-specifications-per-compute-capability>

Multiprocessor Level

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=hello#multiprocessor-level>

Device Memory Accesses

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=hello#device-memory-accesses>