Code of the Day

- `std::ceil(num)` computes the least integer value not less than `num`
  - `std::ceil(4.1) = 5`
  - `std::ceil(2.5) = 3`
  - `std::ceil(7.0) = 7`

- Assume both `x` and `y` are integers,

the ceil of x/y equal to `(x + y – 1) / y`

  - std::ceil(7 / 3) = 3 is equal to (7 + 3 - 1) / 3 = 9 / 3 = 3

  - Why? By adding `y - 1` to `x`, any remainder from dividing `x` by `y` is effectively “rounded up” because the numerator is close to the next multiple of `y`.

ECE 455

GPU Algorithm and System Design

[Fall 2025]

Kernel Execution Configuration

10/6/2025

Before we get started…

- Quick overview, things discussed last time
  - Hardware for GPU computing and its comparison with CPU architecture
    - What is a GPU and why do we need a GPU
  - Looking at the basics of GPU computing using CUDA

- Purpose of today's lecture: Cover the basics of GPU computing
  - Kernel execution configuration
    - Go through several examples, including a basic GPU-parallel matrix multiplication
  - Device-level scheduling for kernel

- Miscellaneous
  - Assignment #4 – due on **10/10 at 23:50 PM**
  - Final project proposal – due on **10/17 at 23:50 PM**

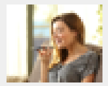
Languages Supported in CUDA

- CUDA is a language extension from C and C++
  - CUDA is a very good friend of C and C++
  - Current nvcc can support up to C++17 and partially C++20
  - Compiler support for C++ language is avail here: https://en.cppreference.com/w/cpp/compiler_support

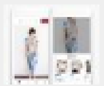
- CUDA also supports other popular languages in scientific computing:
  - CUDA Fortran is available from the NVIDIA HPC SDK (https://developer.nvidia.com/hpc-sdk)
  - PyCUDA maps the entire CUDA API into Python (https://documen.tician.de/pycuda/)

The CUDA-enabled Ecosystem for GPU Computing

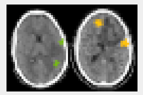
USE-CASES

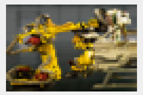

Speech

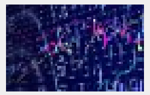
Translate


Recommender

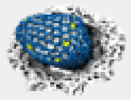
CONSUMER INTERNET

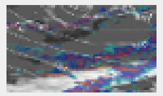

Healthcare

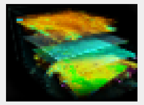

Manufacturing


Finance

INDUSTRIAL APPLICATIONS


Molecular Simulations


Weather Forecasting


Seismic Mapping

SUPERCOMPUTING

APPS & FRAMEWORKS

TensorFlow, mxnet, ONNX, RAPIDS, PYTORCH, Chainer

Amber, NAMD, +600 Applications

CATIA, Ps, AUTODESK 3DS MAX, Windows 10

CUDA-X LIBRARIES

MACHINE LEARNING: cuDF, cuML, cuGRAPH

DL / HPC: cuDNN, CUTLASS, TENSORRT, CUDA Math Libraries

LANGUAGES: python, OpenACC, C++, LLVM Compiler For CUDA

CUDA

CUDA TOOLKIT: CUDA COMPILER; DEVELOPER TOOLS (DEBUGGERS, PROFILERS); CUDA C++ CORE

CUDA DRIVER: MEMORY MANAGEMENT; WINDOWS & GRAPHICS; COMMS LIBRARIES

OS PLATFORMS

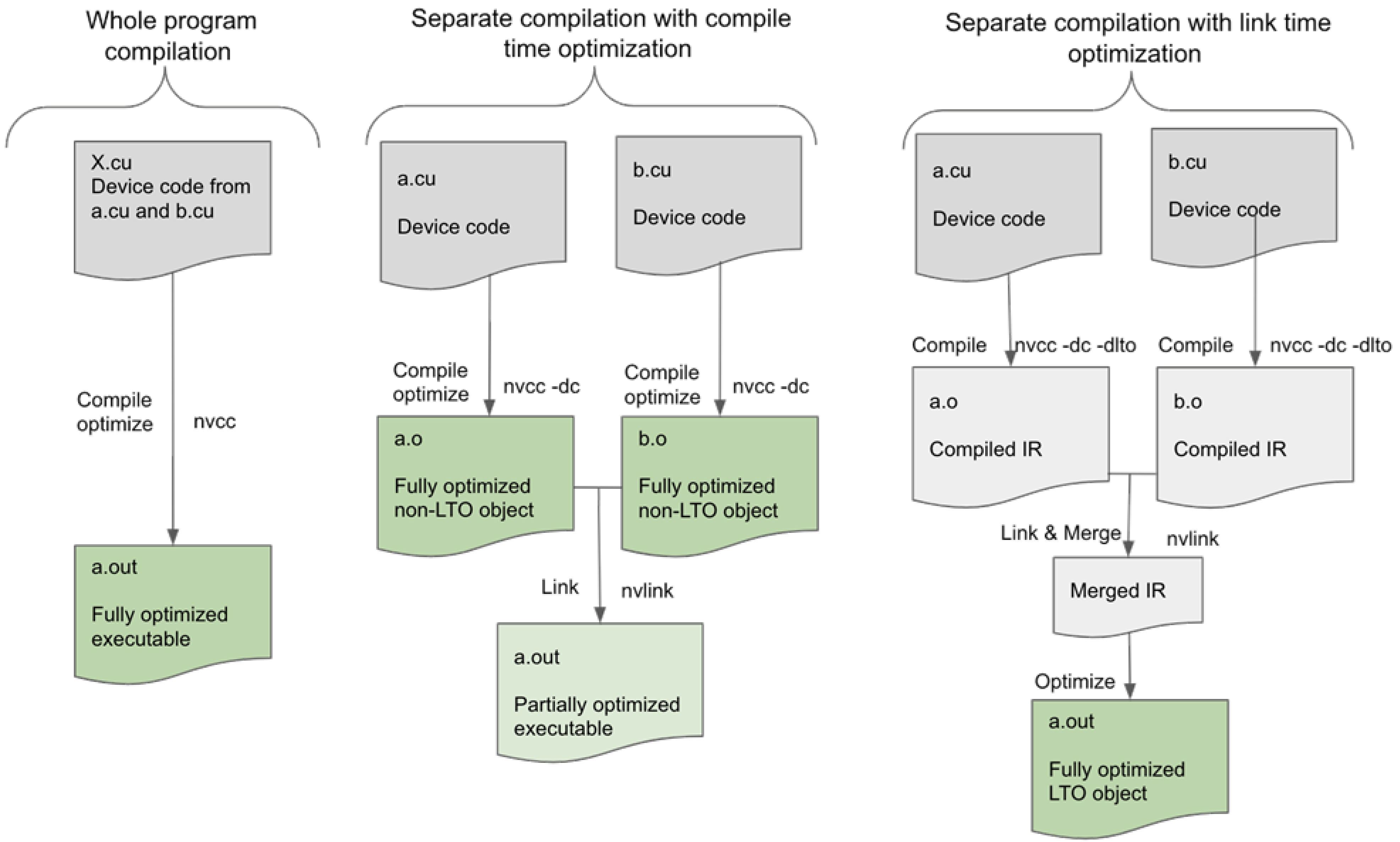
CentOS, SUSE, Windows Server

GPU Kernel Execution Configuration

CUDA Programming Concept Consists of “Host” and “Device”

- The HOST
  - This is your CPU executing the “master” thread – basically everything runs on CPU
- The DEVICE
  - This is the GPU card running your GPU kernels, connected to the HOST through a PCIe (or NVLink)
- The HOST (the master CPU thread) instructs the DEVICE to execute a KERNEL
  - When launching the KERNEL, HOST must inform DEVICE how many threads should each execute KERNEL
  - HOST must also ensure the required data has been allocated and copied to DEVICE memory

Three Possible Compilation Flows for CUDA Programs (.cu)


Whole program compilation
X.cu Device code from a.cu and b.cu
Compile optimize
nvcc
a.out Fully optimized executable
Separate compilation with compile time optimization
a.cu Device code
b.cu Device code
Compile optimize
nvcc -dc
Compile optimize
nvcc -dc
a.o Fully optimized non-LTO object
b.o Fully optimized non-LTO object
Link
nvlink
a.out Partially optimized executable
Separate compilation with link time optimization
a.cu Device code
b.cu Device code
Compile
nvcc -dc -dlto
Compile
nvcc -dc -dlto
a.o Compiled IR
b.o Compiled IR
Link & Merge
nvlink
Merged IR
Optimize
a.out Fully optimized LTO object

[New Topic] The concept of Execution Configuration

- A kernel function must be called with <<< execution configuration parameters >>>:
  - A three-dimensional grid to configure blocks – *how many blocks?*
  - A three-dimensional block to configure threads – *how many threads per block*
  - The shared memory size used by each block
  - Execution stream (job queue) to which this kernel is inserted
    - The CUDA stream is an in-order, first-come-first-server queue

```
__global__ void kernelFoo(...); // declaration

dim3   DimGrid(100, 50);        // 2D grid structure, w/ total of 5000 thread blocks
dim3   DimBlock(4, 8, 8);       // 3D block structure, with 256 threads per block

kernelFoo<<<DimGrid, DimBlock, shm, stream>>>(... arg list); // 5000x256 threads
```

Example

- The number of threads running a kernel is co-decided by “grid” and “block”

- The host call below asks the GPU to execute the kernel “**`foo`**” using 25,600 threads
  - One dimensional grid of 100 blocks
  - One dimensional block of 256 threads

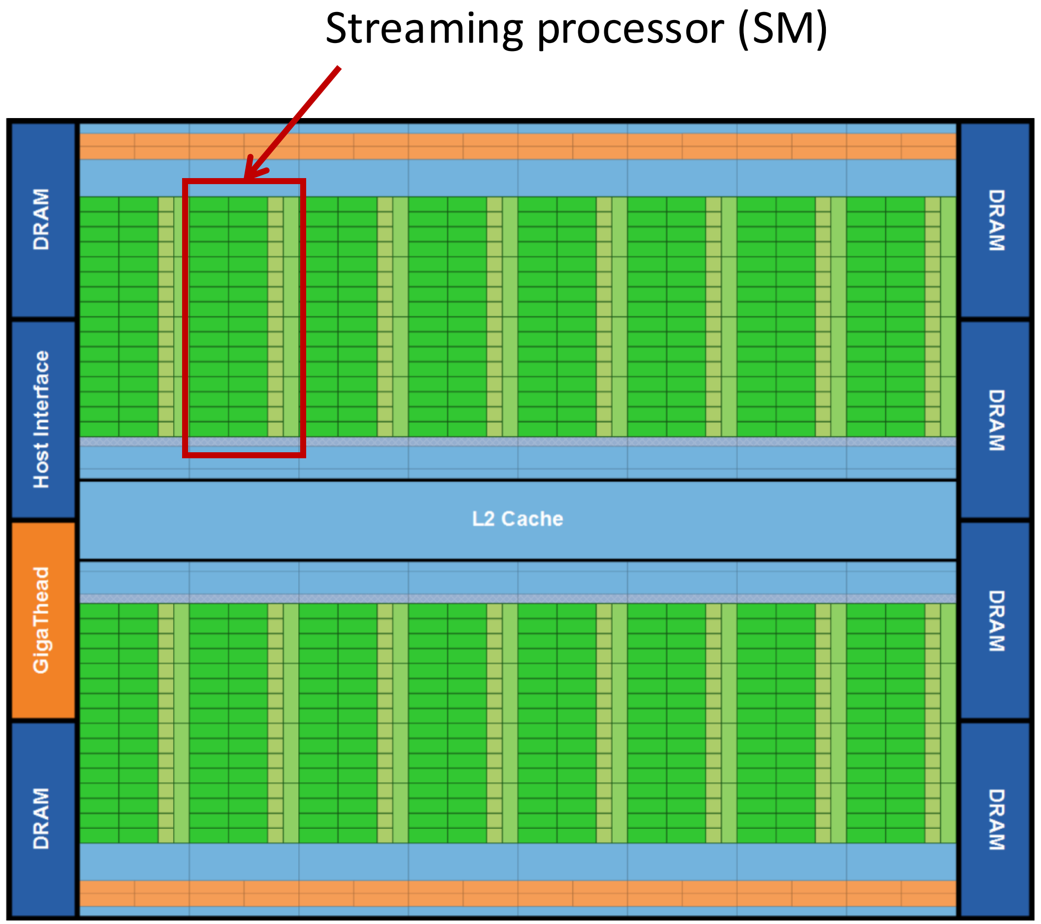
```
foo<<<100, 256>>>(p_matrixD, p_vectorD);
```

- The above execution configuration instructs the GPU to run 100 blocks each of 256 threads

A "Block of Threads" Eventually Gets Assigned to an SM

- The concept of block is important since the "block of threads" represents the entity that gets executed by a streaming multiprocessor (SM)
    - SM is a collection of CUDA cores to run your kernel

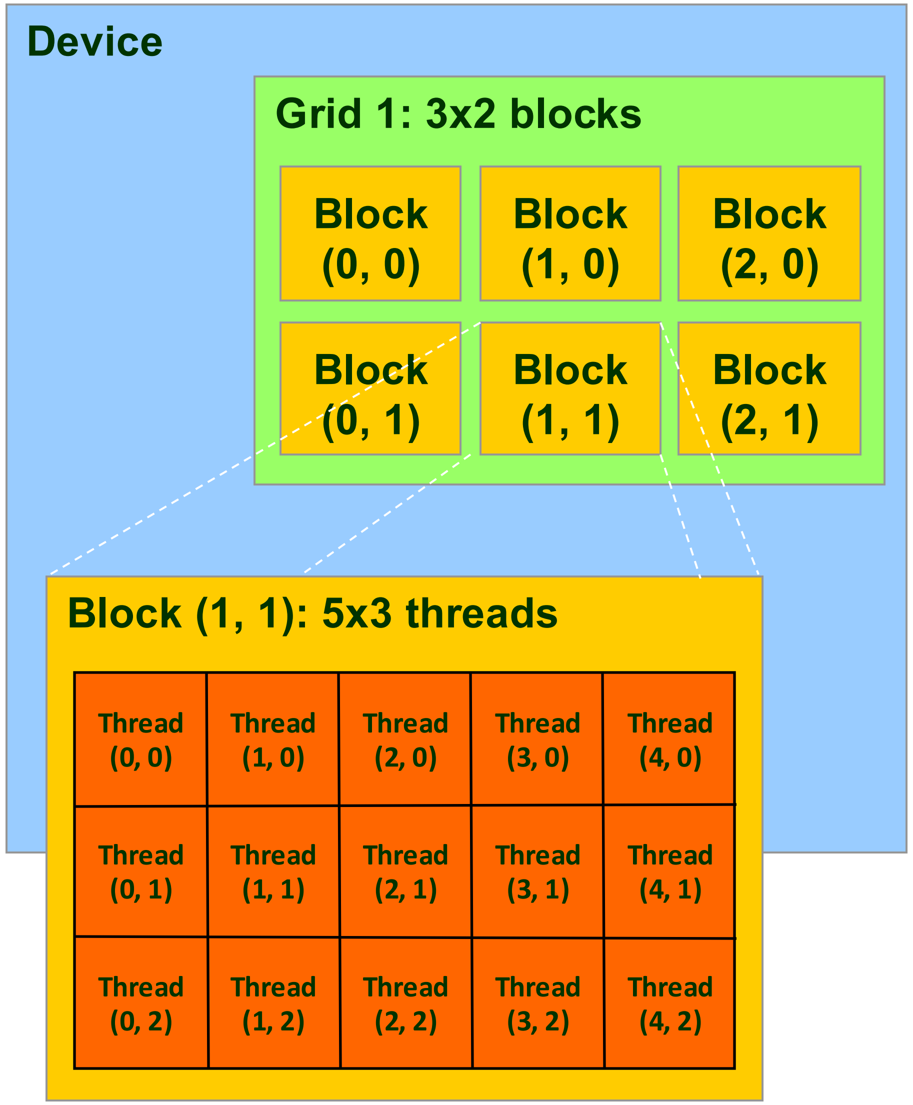
- Most often, an SM executes more than one block at the same time
    - There'll be between 2 and 10 blocks that are "in flight" on one SM
    - Each SM has a maximum limit on the number of blocks that it can run



Execution Configuration Constraints

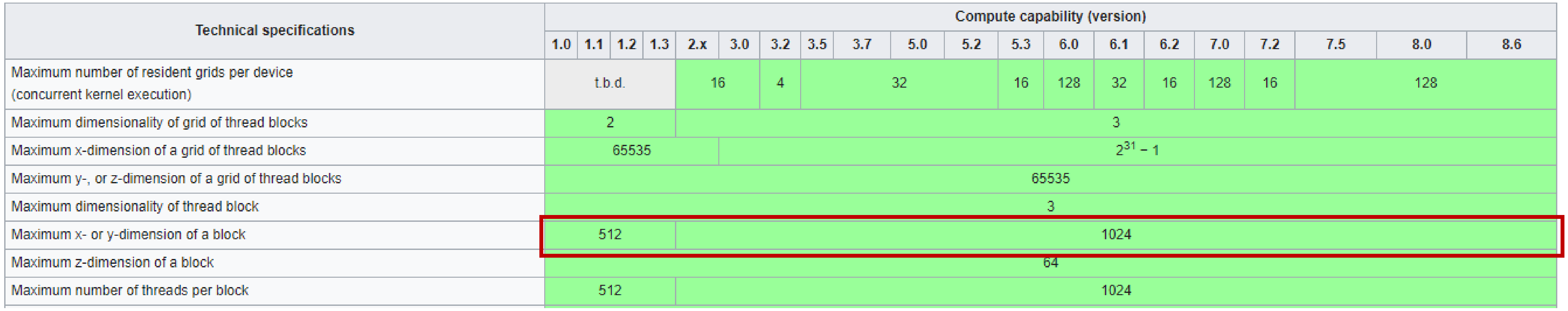
- Within a block:
  - The threads can only be organized as a 3D structure (x, y, z)
  - Maximum x- or y-dimension of a block is 1024
  - Maximum z-dimension of a block is 64
  - **NOTE**: The maximum total number of threads in a block is 1024
    - Was 512 in older GPU

- Within a grid:
  - The blocks can only be organized as a 3D structure (x, y, z)
  - Max of 2^{31}-1 or 65,535 by 65,535 grid of blocks



Constraints are Different at Different Compute Capabilities

- Execution configuration constraints are different from generation to generation
  - Actually, they're evolving towards being more powerful and parallel in the newer generation



Technical specifications	Compute capability (version)																			
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4	32				16	128	32	16	128	16	128		
Maximum dimensionality of grid of thread blocks	2				3															
Maximum x-dimension of a grid of thread blocks	65535					$2^{31} - 1$														
Maximum y-, or z-dimension of a grid of thread blocks	65535																			
Maximum dimensionality of thread block	3																			
Maximum x- or y-dimension of a block	512				1024															
Maximum z-dimension of a block	64																			
Maximum number of threads per block	512				1024															

[Wikipedia]→

In Theory, You can Use Millions of GPU Threads …

- Max number of threads a kernel can be invoked with

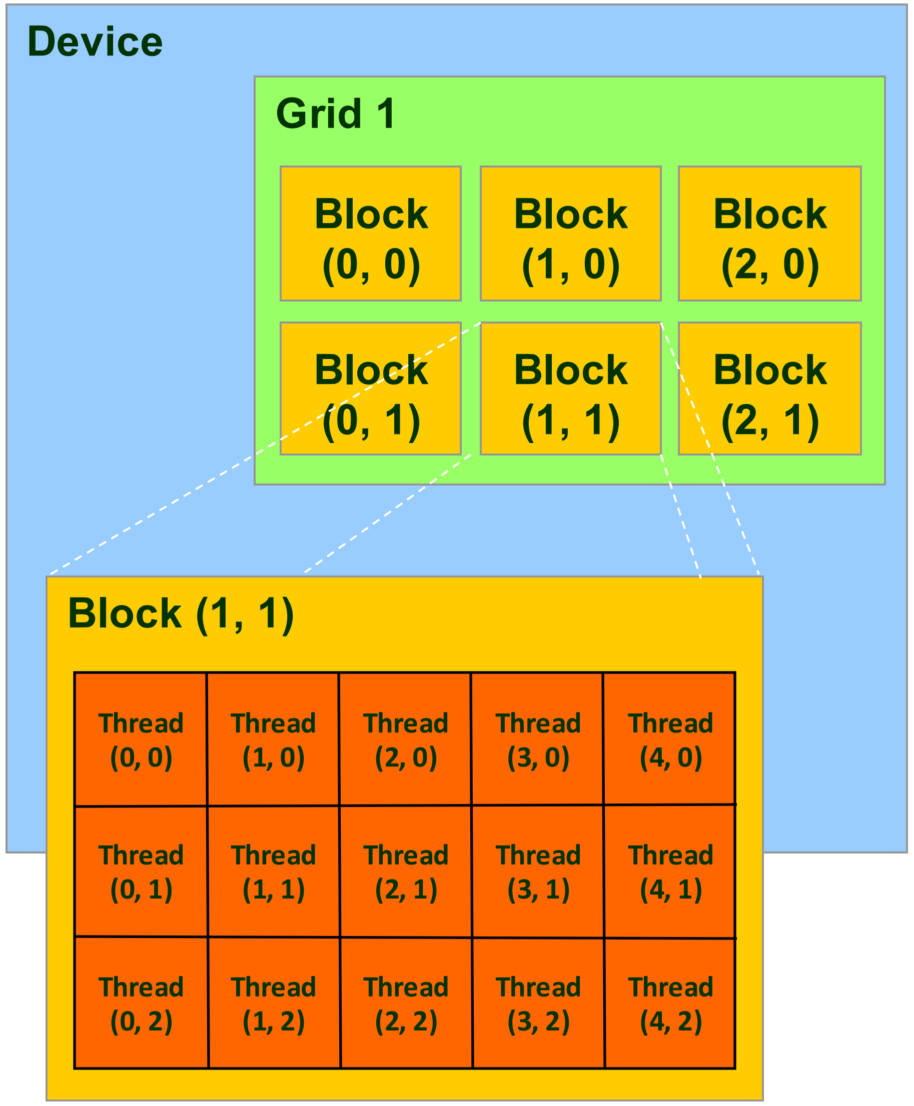
Max number of threads in x-direction: 1024

$$\underbrace{2^{31}}_{\text{Max number of blocks in x-direction}} \times \underbrace{2^{16}}_{\text{Max number of blocks in y-direction}} \times \underbrace{2^{16}}_{\text{Max number of blocks in z-direction}} \times 2^{10} = 2^{\wedge}73$$

- About 9.444732965739290e+21 threads can execute a kernel
  - What if you want more threads to execute that kernel? You need to partition your algorithm into multiple launches of the same or different kernel

Applications can Query Block and Thread Index (Idx)

- Threads and blocks have indices in their dimensions
  - **Used by each thread to decide what data to work on**
  - Block Index: a triplet of `uint`
  - Thread Index: a triplet of `uint`
- Why using this 3D layout?
  - Originated from the gaming/graphics applications
    - Rendering applications are mostly 3D
  - Simplifies memory addressing when processing multidimensional data
    - Handling matrices
    - Solving PDEs on 3D subdomains
    - …



A couple (not all) of the built-in CUDA variables

[Critical in supporting the SIMD parallel computing paradigm]

- Each thread can find out the grid and block dimensions and its block index and thread index
  - This info used to figure out what work the thread needs to do (e.g., array index assigned to this thread)

- Each thread has access to the following read-only built-in variables
  - `threadIdx` (`uint3`) – contains the thread index within a block
    - `uint3` is a built-in type that represents a 3-dimensional vector of unsigned integers (`var.x`, `var.y`, `var.z`)
  - `blockDim` (`dim3`) – contains the dimension of the block
  - `blockIdx` (`uint3`) – contains the block index within the grid
  - `gridDim` (`dim3`) – contains the dimension of the grid
  - [`warpSize` (`uint`) – provides warp size, we'll talk about this later…]

Example – simpleKernel<<<1,4>>>(devArray)

```
#include<cuda.h>
#include<iostream>

__global__ void simpleKernel(int* data)
{
    //this adds a value to a variable stored in global memory
    data[threadIdx.x] += 2*(blockIdx.x + threadIdx.x);
}

int main()
{
    const int numElems = 4;
    int hostArray[numElems], *devArray;

    //allocate memory on the device (GPU); zero out all entries in this device array
    cudaMalloc((void**)&devArray, sizeof(int) * numElems);
    cudaMemset(devArray, 0, numElems * sizeof(int));

    //invoke GPU kernel, with one block that has four threads
    simpleKernel<<<1,numElems>>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(hostArray, devArray, sizeof(int) * numElems, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: " << std::endl;
    for (int i = 0; i < numElems; i++)
        std::cout << hostArray[i] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);
    return 0;
}
```

The NVIDIA compiler, available on Euler

```
$ nvcc firstExample.cu -o firstExample
$ ./firstExample
Values stored in hostArray:
0
2
4
6
```

This is a bash shell, running under Linux (ubuntu distro), via WSL2, on a Windows Laptop

NOTE: Do not run like this on Euler, you can get your account suspended. Use Slurm

Quiz: Following Up on the Previous Example
[and segue into the "Execution Configuration"]

```
#include<cuda.h>
#include<iostream>

__global__ void simpleKernel(int* data)
{
    //this adds a value to a variable stored in global memory
    data[threadIdx.x] += 2*(blockIdx.x + threadIdx.x);
}

int main()
{
    const int numElems = 4;
    int hostArray[numElems], *devArray;

    //allocate memory on the device (GPU); zero out all entries in this device array
    cudaMalloc((void**)&devArray, sizeof(int) * numElems);
    cudaMemset(devArray, 0, numElems * sizeof(int));

    //invoke GPU kernel, with one block that has four threads
    simpleKernel<<<1,numElems>>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(hostArray, devArray, sizeof(int) * numElems, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: " << std::endl;
    for (int i = 0; i < numElems; i++)
        std::cout << hostArray[i] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);
    return 0;
}
```

- What happens if we invoke the kernel like this:

`simpleKernel<<<1,12>>>(devArray)`

- What happens if we invoke the kernel like this:

`simpleKernel<<<2,4>>>(devArray)`

Matrix Multiplication Example

Simple Example: Matrix Multiplication

- We will go through a straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA
    - By straightforward, we mean the following:
        - Use only global memory (off-chip DRAM on the GPU card)
        - Assume matrix will be of small dimension; job can be done using one block of threads
        - Concentrate on
            - Thread ID usage
            - Memory data transfer API between host and device
- NOTE: Related to your next lab assignment

Preamble: on the Matrix Data Structure

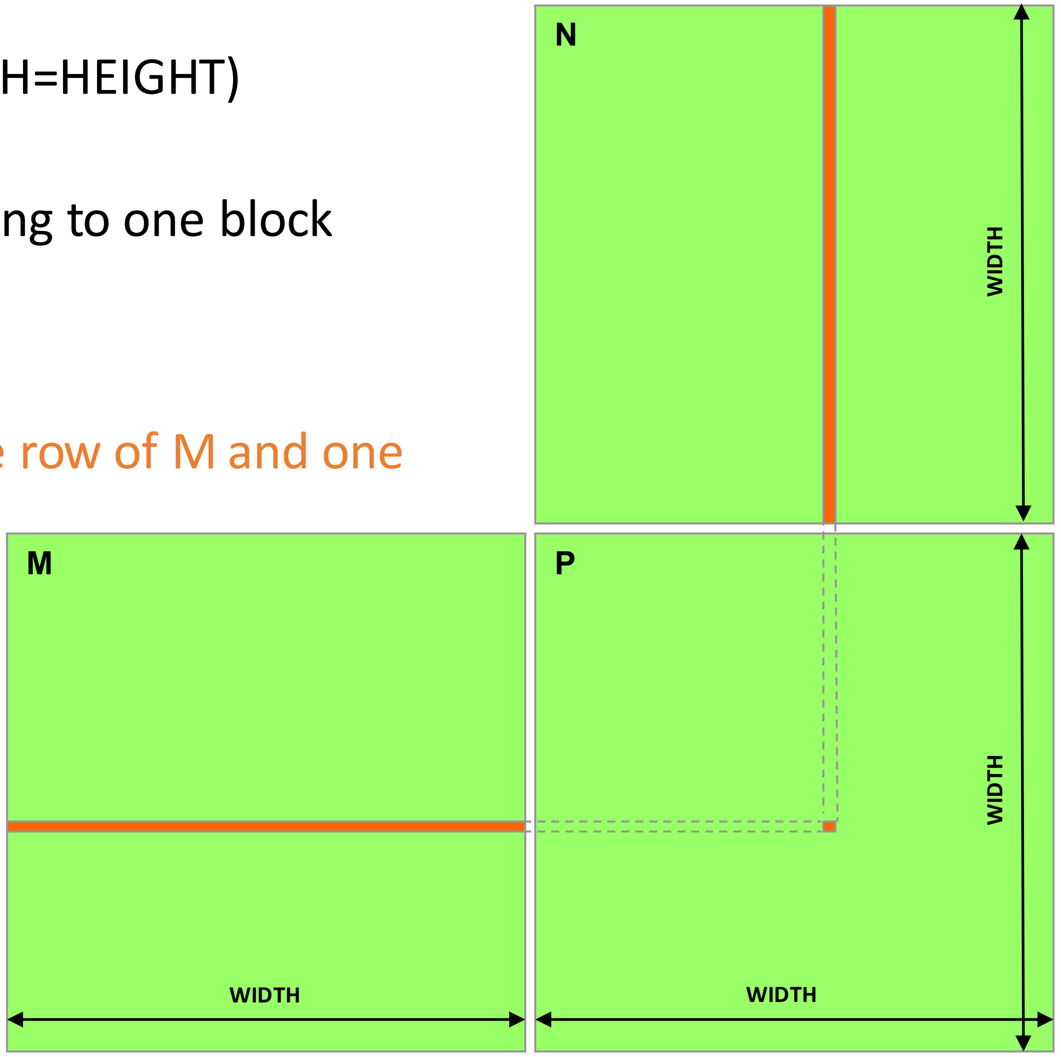
- Define a struct to represent the 2D matrix
  - `width` and `height` represent the dimension of the matrix
  - The matrix is stored in row-major order in a one-dimensional array pointed to by `elements`

```
// IMPORTANT - Matrices are stored in row-major order:
// M(r, c) = M.elements[r * M.width + c]

typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```

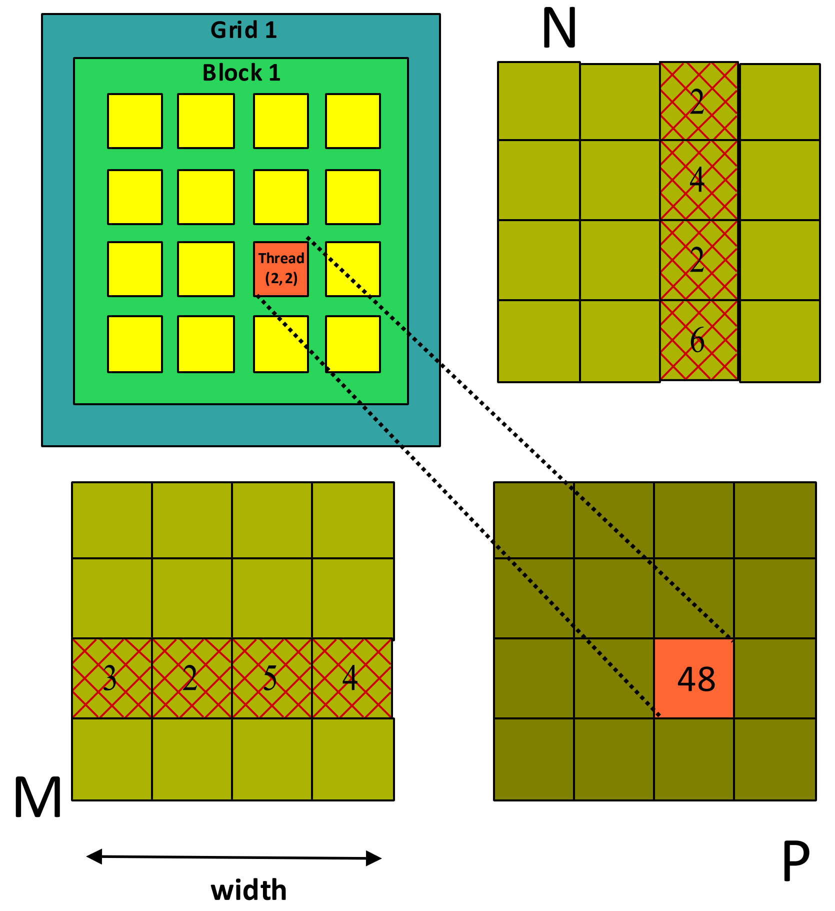
Square Matrix Multiplication Example

- Compute P = M * N
  - The matrices P, M, N are of square size (WIDTH=HEIGHT)
  - Assume WIDTH was defined to be 32
  - Each matrix has 32x32 = 1024 elements – fitting to one block
- Kernel design decisions:
  - One thread handles one element of P
  - Each thread will access all the entries in one row of M and one column of N
    - 2*WIDTH read accesses to global memory
    - One write access to global memory



Multiply M by N using One Block

- One block of threads computes matrix P
  - Each thread computes one element of P
- Each thread does a dot product
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of elements in the loaded row and column
  - In the right example, we have 4+4 global memory read and 1 global memory write
- Size of matrix limited by the number of threads allowed in a thread block



Matrix Multiplication: sequential approach, coded in C++

This solution runs fully on the CPU; doesn't have anything to do with GPU computing

```
// Matrix multiplication on the (CPU) host in double precision;

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i) {
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];  // loop along a row of M
                double b = N.elements[k * N.width + j];  // loop along a column of N
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
    }
}
```

Note: On CPU, you need 3 loops; On GPU, we don't need the 2 outer loops by parallelizing them via GPU threads

Step 1: Matrix Multiplication, Host-side. Main Program Code

This solution leverages GPU computing

```
int main(void) {
    // Allocate and initialize the matrices.
    // The last argument in AllocateMatrix: should an initialization with
    // random numbers be done? Yes: 1.  No: 0 (everything is set to zero)
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);

    return 0;
}
```

Step 2: Matrix Multiplication [host-side code]

```
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);

    // Setup the execution configuration
    dim3 dimGrid(1, 1, 1);
    dim3 dimBlock(WIDTH, WIDTH);

    // Launch the kernel on the device
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```

Step 3: Matrix Multiplication - Device-side Kernel Function

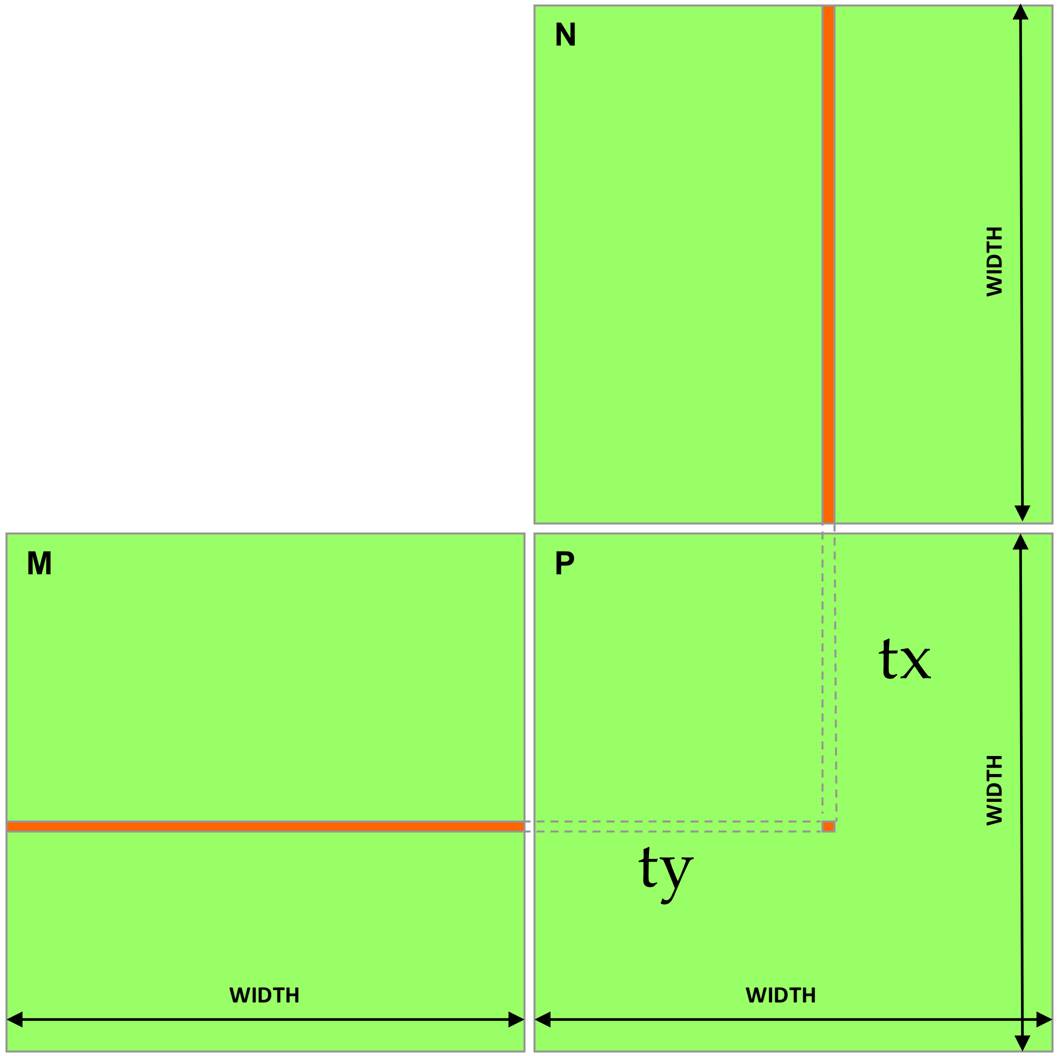
```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P) {
    // 2D Thread Index; computing P[ty][tx]…
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue will end up storing the value of P[ty][tx].
    // That is, P.elements[ty * P. width + tx] = Pvalue
    float Pvalue = 0.;

    for (int k = 0; k < M.width; ++k)  {
         float Melement = M.elements[ty * M.width + k];
         float Nelement = N.elements[k * N. width + tx];
         Pvalue += Melement * Nelement;
    }

    // Write matrix to device memory; each thread one element
    P.elements[ty * P. width + tx] = Pvalue;
}
```

Note: `Pvalue` is a local value, stored in a register (very fast). Global memory accessed only once, at the very end



Step 4: Other Helper Functions

```
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M) {
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost) {
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size, cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice) {
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size, cudaMemcpyDeviceToHost);
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}
void FreeMatrix(Matrix M) {
    delete[] M.elements;
}
```

Takeaway from the Matrix Multiplication Example

- GPU computing: identify the loop of independent iterations and parallelize it via GPU threads
    - This replaces the purpose of the "`for`" loop
    - Number of threads & blocks is established at run-time depending on the data size

- Thus, in many cases, the rule is Number of threads = Number of data items (or tasks)
    - You'll have to come up with a rule to match a thread to a data item that this thread needs to process

- Understanding what thread does what job is very important but also a very common source of bugs in GPU computing
    - Out of boundary – segmentation fault
    - Multiple threads mapped to the same element – data race
    - …

Takeaway for Array Indexing

- In GPU computing, you typically need to use many blocks (each of which contains the same number of threads, say M) to get a job done
    - Why many blocks? Since there is an upper limit on number of threads in a CUDA block – specifically, $M \leq 1024$

- Fundamental question that a thread asks in GPU computing:

What work, or which task, do I have to do?

- In our previous example, we have the following 1D mapping:

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

[short topic, two slides]: Timing Your Application

- Timing support through `cudaEvent_t` – part of the CUDA API
  - You use it as soon as you include `<cuda.h>`
  - Provides cross-platform compatibility
  - Deals with the asynchronous nature of the device calls by relying on events and forced synchronization

- Reports time in milliseconds, accurate within 0.5 microseconds

- From NVIDIA CUDA Library Documentation:

  Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). If either event has not been recorded yet, this function returns `cudaErrorInvalidValue`*. If either event has been recorded with a non-zero stream, the result is undefined.*

Timing Example: timing a GPU call

```
#include<iostream>
#include<cuda.h>

int main() {
    cudaEvent_t startEvent, stopEvent;
    cudaEventCreate(&startEvent);
    cudaEventCreate(&stopEvent);

    cudaEventRecord(startEvent, 0);

    yourKernelCallComesHere<<<NumBlk,NumThrds>>>(args);

    cudaEventRecord(stopEvent, 0);
    cudaEventSynchronize(stopEvent);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);
    std::cout << "Time to get device properties: " << elapsedTime << " ms\n";

    cudaEventDestroy(startEvent);
    cudaEventDestroy(stopEvent);
    return 0;
}
```

Execution Scheduling Issues

Execution Scheduling

- Starting point observation:
  - You launch a grid with many blocks, each block containing many threads

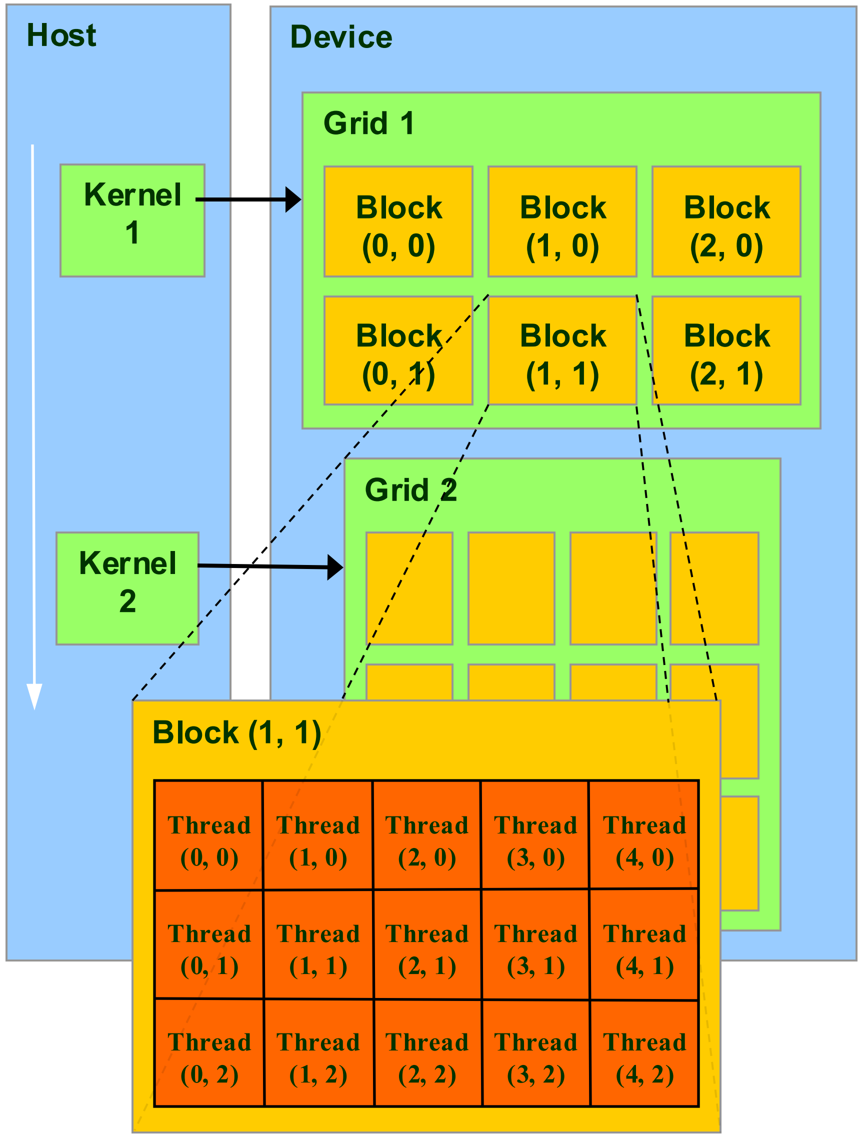
- Main questions we will answer:
  - Is there an order in which these blocks are picked up and executed?
  - How are the threads in a block executed?
  - Who orchestrates the execution of the threads?

Kernel Scheduling at a High-level Perspective

- There are two schedulers at work in GPU computing
  - A device-level scheduler: assigns one [more] block to an SM that signals room to run a block
    - This scheduler is also referred to as “GigaThread engine” by Nvidia
  - An SM-level scheduler: schedules the execution of the threads in a block onto the SM functional units
    - This is the more interesting scheduler that actually assigns threads to run your kernel code
    - Example functional units include FP16, FP32, INT32 processors, etc.

Device-Level Scheduler

- This is the first level of scheduling:
  - For running a large number of blocks given that we have a relatively small number of SMs
  - There are perhaps tens of thousands of blocks that will eventually get executed in 108 SMs (on an A100)
- Thread Blocks are distributed to the SMs
  - Potentially more than one block scheduled per SM
- As a Thread Block completes kernel execution on an SM, resources on that SM are freed
  - Device level scheduler will then dispatch next thread block in line
- There are limits for resident blocks and threads on an SM:
  - 32 blocks on the Pascal SM, Volta SM, and Ampere SM
  - No more than 2048 threads can be hosted on an SM

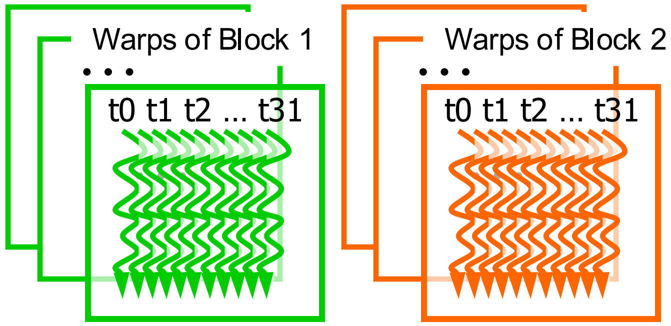


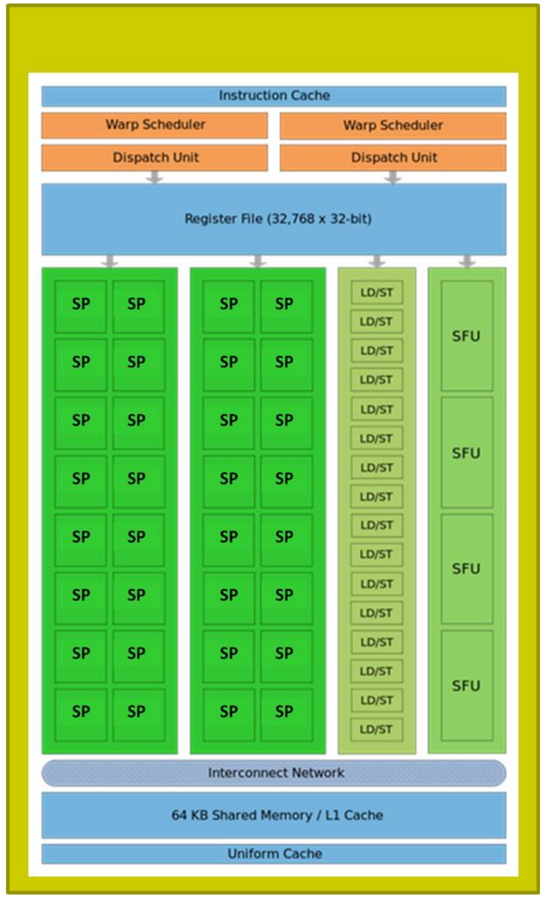
Device-Level Scheduler (cont’d)

- Once a block is picked up for execution by one SM, the block does not leave that SM before each thread of that block finishes executing the kernel
  - If the block has many threads idle due to boundary condition, you waste a lot of thread resources
- Once a block is finished & retired, only then can another block land for execution on that SM
  - The Device-Level Scheduler dispatches a block of threads to an SM that indicates that it has excess capacity
- Obviously, if your GPU has many SMs, many blocks will be running in parallel
  - The more expensive the GPU, the more SMs it has (just like a CPU having more cores)

SM-Level Schedulers

- Each block of threads is divided into 32-thread **warps**
  - “32”: selected by NVIDIA, programmer has no say
  - Warp: A group of 32 threads of consecutive **ID**s
- Warps are the basic scheduling unit on the SM
  - Everything eventually boils down to warp-level execution
- There is a limit on the number of resident warps per SM
  - 64: on Kepler, Maxwell, Pascal, Volta, Ampere (i.e., 2048 resident threads)
  - 48: on Fermi (i.e., 1536 resident threads)
  - 32: on Tesla (i.e., 1024 resident threads)
- Although in theory you can create many many threads …
  - There are always hardware limits on the number of threads per block, number of blocks per SM, number of warps per SM, etc.





[short/important topic]: Thread Index vs. Thread ID

[critical in (i) understanding how SIMD is supported in CUDA, and (ii) understanding the concept of "warp"]

- Each block organizes its threads in a 3D structure defined by its three dimensions: D_x, D_y, and D_z that you specify.
- A block cannot have more than 1024 threads $\Rightarrow$ $D_x \times D_y \times D_z \leq 1024$.
- Each thread in a block can be identified by a unique index (x, y, z), and

$$0 \leq x < D_x \qquad 0 \leq y < D_y \qquad 0 \leq z < D_z$$

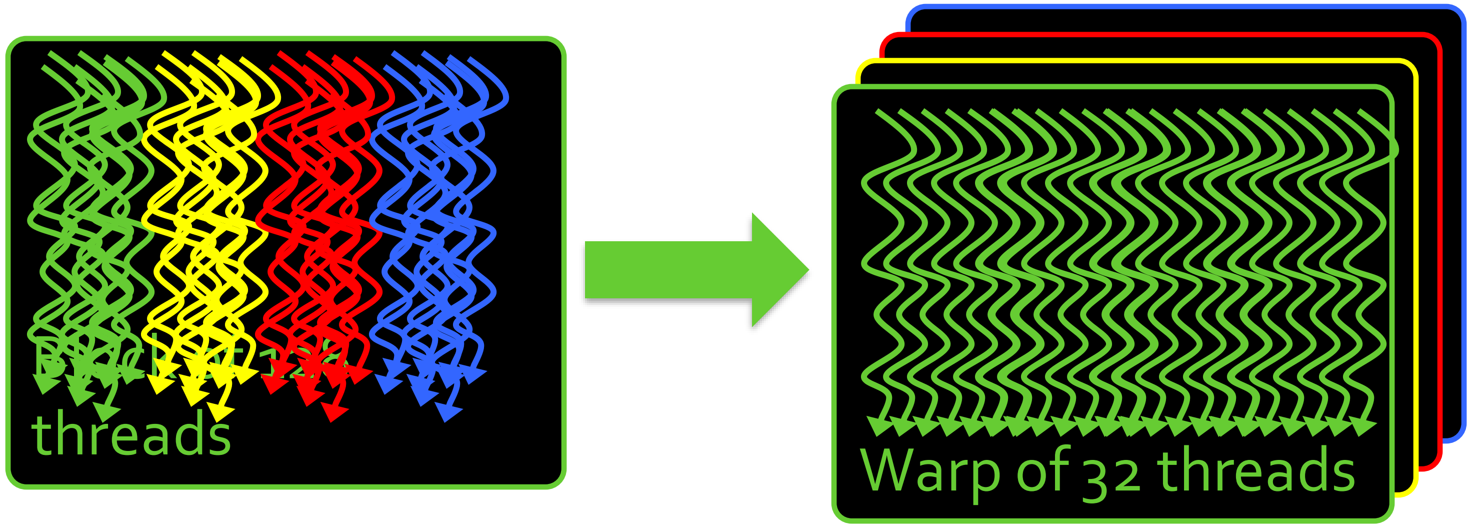
- A triplet (x, y, z), called the thread index, is a high-level representation of a thread in the economy of a block. Under the hood, the same thread has a simplified and unique id, which is computed as $\boxed{t_{id} = x + y * D_x + z * D_x * D_y}$. You can regard this as a "projection" to a 1D representation. The concept of thread id is important in understanding how threads are grouped together in warps (more on "warps" later).
- In general, operating for vectors typically results in you choosing $D_y = D_z = 1$. Handling matrices typically goes well with $D_z = 1$. For handling PDEs in 3D you might want to have all three block dimensions nonzero.

Organizing Threads into Warps

- Thread IDs within a warp are consecutive and increasing
  - Related to the 1D projection from thread index to thread ID
  - In CUDA multidimensional blocks, the x thread index runs fastest, then y, then z
  - Threads with IDs (0...31) combine into warp 0, threads (32…63) into warp 1, etc.
- Partitioning of threads into warps is always the same
  - Warp size has always been 32 and is unlikely to change soon
- While you can rely on ordering among threads, DO NOT rely on any ordering among warps (there is no such thing)
  - Warp scheduling is not under user control in CUDA
  - Which warp runs first or later is completely decided by the SM-level scheduler

Blocks of Threads are Organized and Executed as Warps

- Each thread block split into one or more warps
- If thread block size is not multiple of warp size, unused threads go wasted
  - Example: block w/ 50 threads – takes 2 warps of 64 threads, of which the last 14 threads go wasted

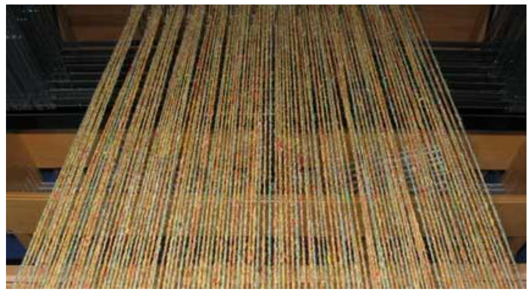


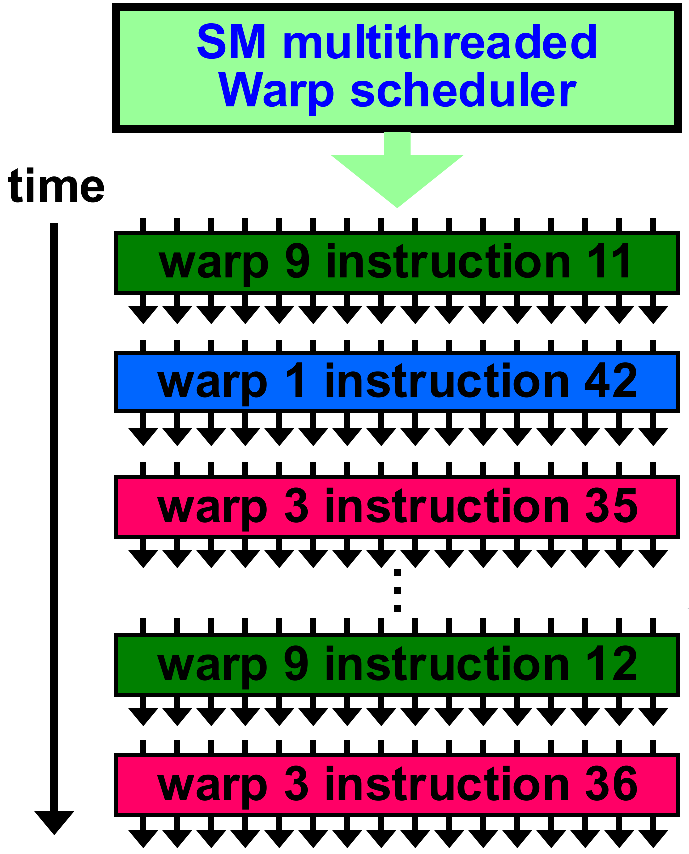
- The hardware schedules each warp independently
- Different warps within a thread block execute independently

Quiz

- If it turns out that 3 blocks are processed by an SM and each Block has 256 threads, how many warps are managed by the SM?
  - Each block of threads is divided into 256/32 = 8 warps
  - There are 8 * 3 = 24 warps
  - At any point in time, there are 24 warps that can be selected for execution

SM-Level Warp Scheduling is Very Fast





- SM hardware implements almost zero-overhead warp scheduling
  - A warp whose next instruction has its operands ready for use is eligible for execution
  - Eligible warps are selected for execution on a prioritized scheduling policy managed by hardware
  - All threads in a warp execute the same instruction, in lockstep fashion – all threads complete that instruction together and then move on

Context switch overhead among warps is very low (unlike CPU)

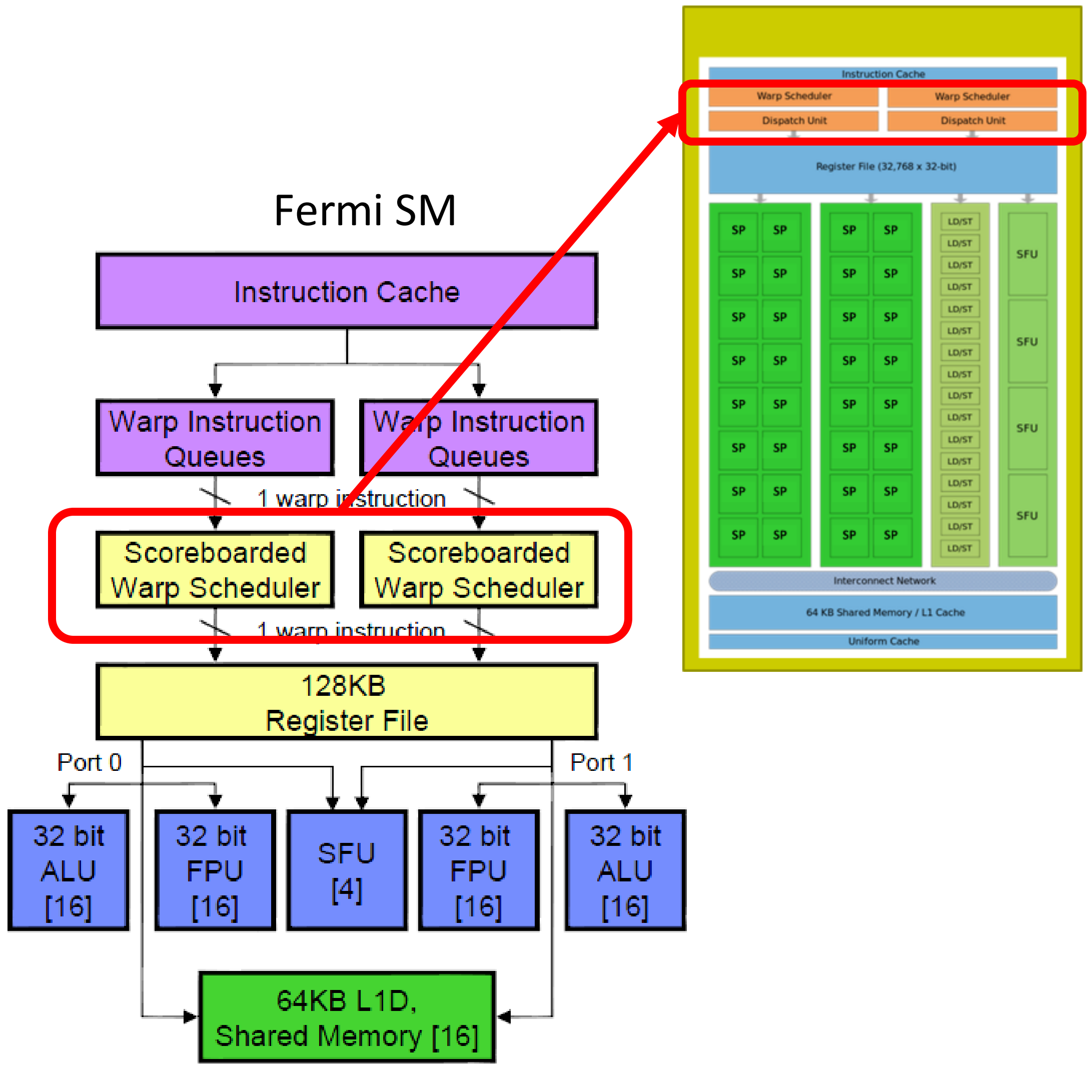
Fermi Specifics

- There are two warp schedulers per SM that can each issue one warp of threads

- One warp issued at each clock cycle by each scheduler (very fast!)

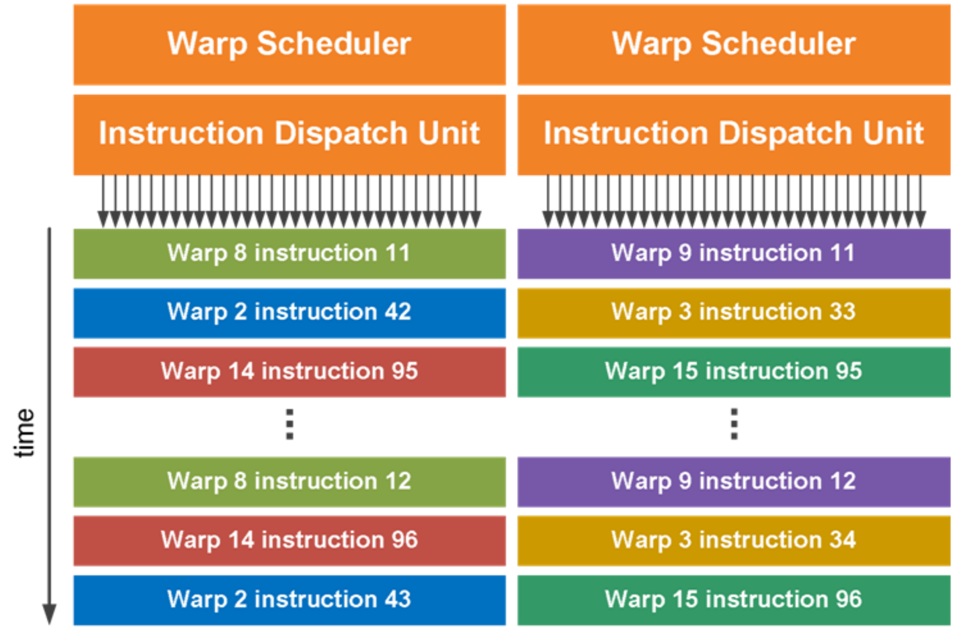
- No more than two warps can be dispatched for execution on the SM's functional units during any given cycle

- Scoreboarding used to figure out which warp is ready



GPU Architecture is Pipelined

- One instruction can be retired with each clock cycle
  - Multiple instructions can be pipelined too
- However, ILP support is relatively limited
  - To the best of my knowledge:
    - There is no out-of-order execution smarts
    - There is no prefetching
    - There is no speculative execution
- The odd warp index are handled by a scheduler, the even ones are handled by the other scheduler
  - No exchanging warps at run time between schedulers


Warp Scheduler
Warp Scheduler
Instruction Dispatch Unit
Instruction Dispatch Unit
time
Warp 8 instruction 11
Warp 9 instruction 11
Warp 2 instruction 42
Warp 3 instruction 33
Warp 14 instruction 95
Warp 15 instruction 95
Warp 8 instruction 12
Warp 9 instruction 12
Warp 14 instruction 96
Warp 3 instruction 34
Warp 2 instruction 43
Warp 15 instruction 96