Day 5:
High Performance Computing
CMT106

David W. Walker

Professor of High Performance Computing

Cardiff University

http://www.cardiff.ac.uk/people/view/118172-walker-david

Day 5

- 9:30 – 10:30am: Lecture on programming GPUs with CUDA; vector addition and other example codes.
- 10:30 – 10:50am: Break.
- 10:50am – 12:00pm: Lecture on programming GPUs with CUDA; Laplace equation solver.
- 12:00 – 1:30pm: Lunch break.
- 1:30 – 3:00pm: Lab session, try out the CUDA example code yourself (includes 15min break) on the Linux lab machines.
- 3:00 – 3:15pm: Review of the lab session and overview of the third piece of coursework .
- 3:15pm – 4:30pm: Lecture on tiled matrix multiplication in CUDA; optimizing CUDA codes.

Topics Covered on Days 1-4

- *Day 1:* Introduction to parallelism; motivation; types of parallelism; Top500 list; classification of machines; SPMD programs; memory models; shared and distributed memory; OpenMP; example of summing numbers.
- *Day 2*: Interconnection networks; network metrics; classification of parallel algorithms; speedup and efficiency.
- *Day 3*: Scalable algorithms; Amdahl's law; sending and receiving messages; programming with MPI; collective communication; integration example.
- *Day 4:* Regular computations and simple examples – the wave equation and Laplace's equation.

Topics Covered on Days 5-7

- *Day 5: Programming GPUs with CUDA; CUDA device memory architecture; simple programming examples.*
- *Day 6*: Dynamic communication and the molecular dynamics example; irregular computations; the WaTor simulation .
- *Day 7*: Load balancing strategies; message passing libraries; block-cyclic data distribution.

CUDA

- "Compute Unified Device Architecture" introduced by NVidia in 2007.
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute – graphics-free API
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

Useful CUDA Resources

1. Textbook: “Programming Massively Parallel Processors,” David B. Kirk and Wen-mei W. Hwu, third edition, pub. Morgan Kaufmann, 2016. ISBN 978-0-12-811986-0. https://www.elsevier.com/books/programming-massively-parallel-processors/kirk/978-0-12-811986-0
2. *NVidia CUDA Programming Guide*, available at http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html - this is for v9.0.

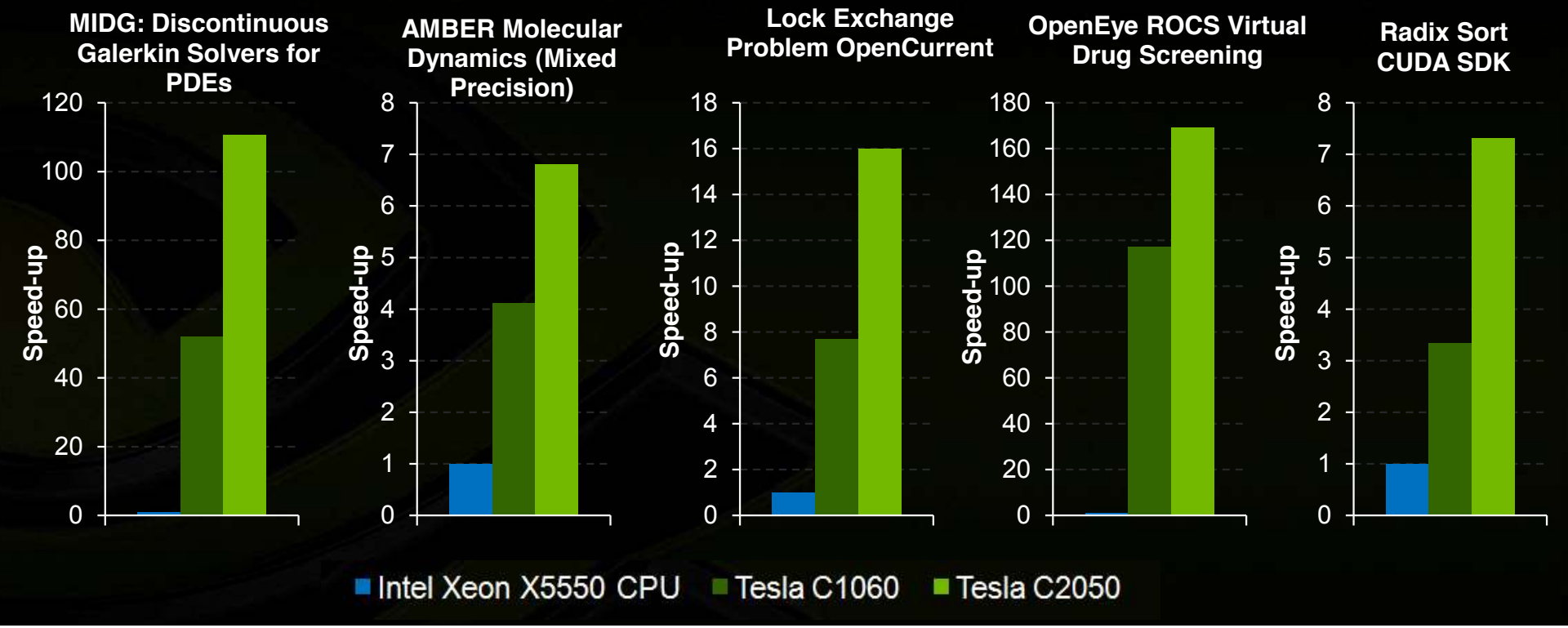
Can You Run CUDA?

- If you have a laptop with an Nvidia GPU, then you may be able to run CUDA on it.
- Find out what GPU your Laptop has and then check if it supports CUDA at:

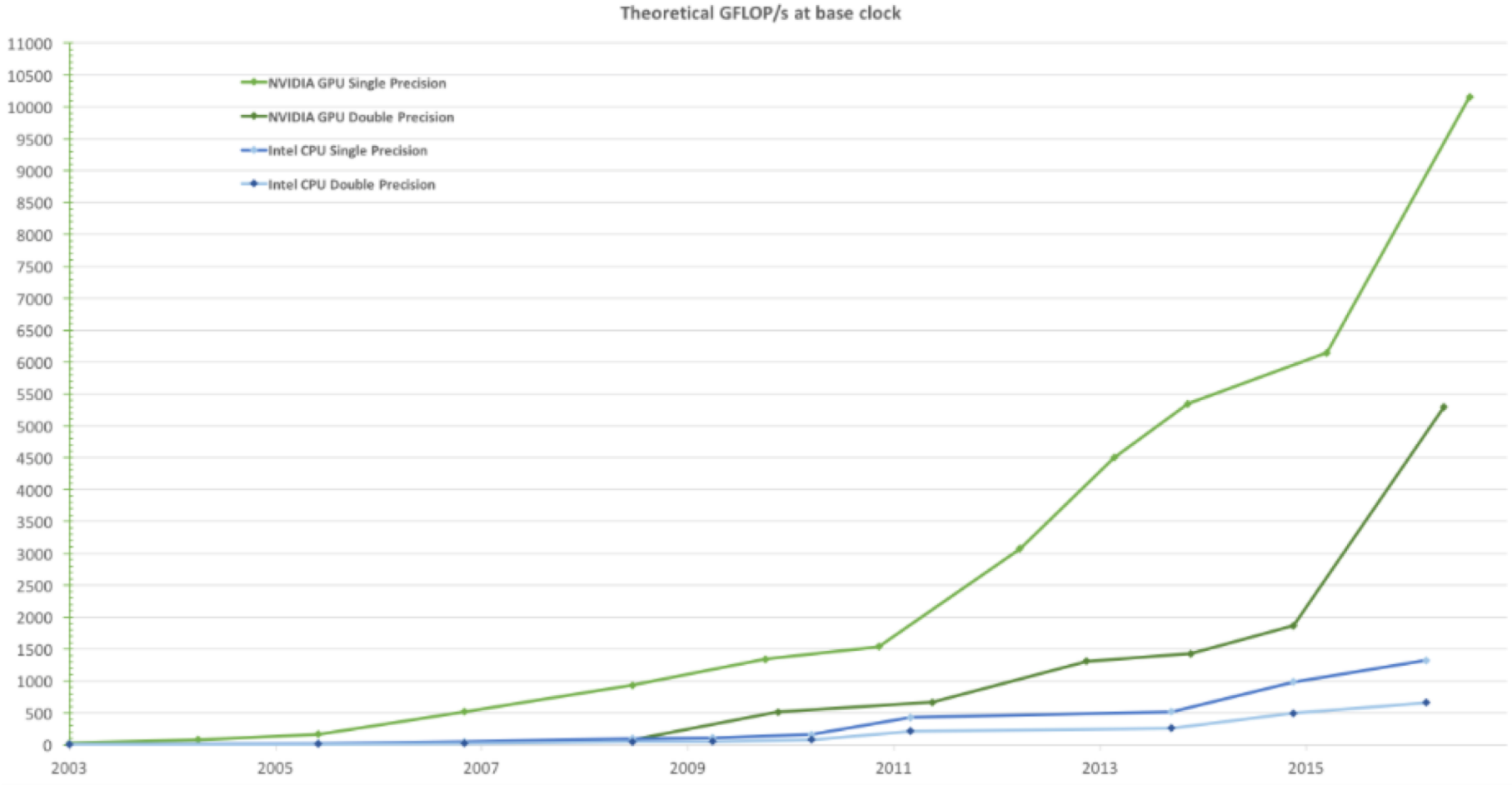
  https://developer.nvidia.com/cuda-gpus
- If you can run CUDA then you can download it from:

  https://developer.nvidia.com/cuda-toolkit

Performance Summary

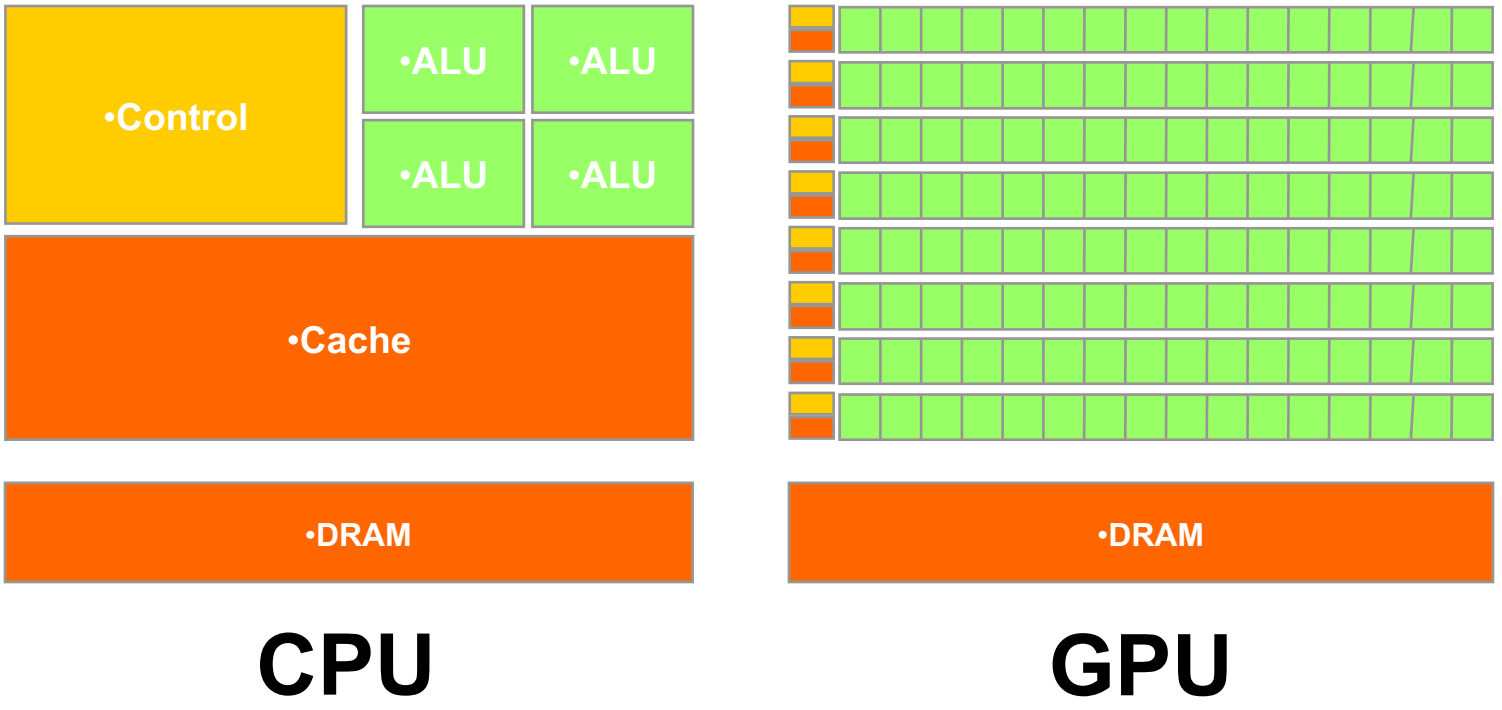


Why Massively Parallel Processing

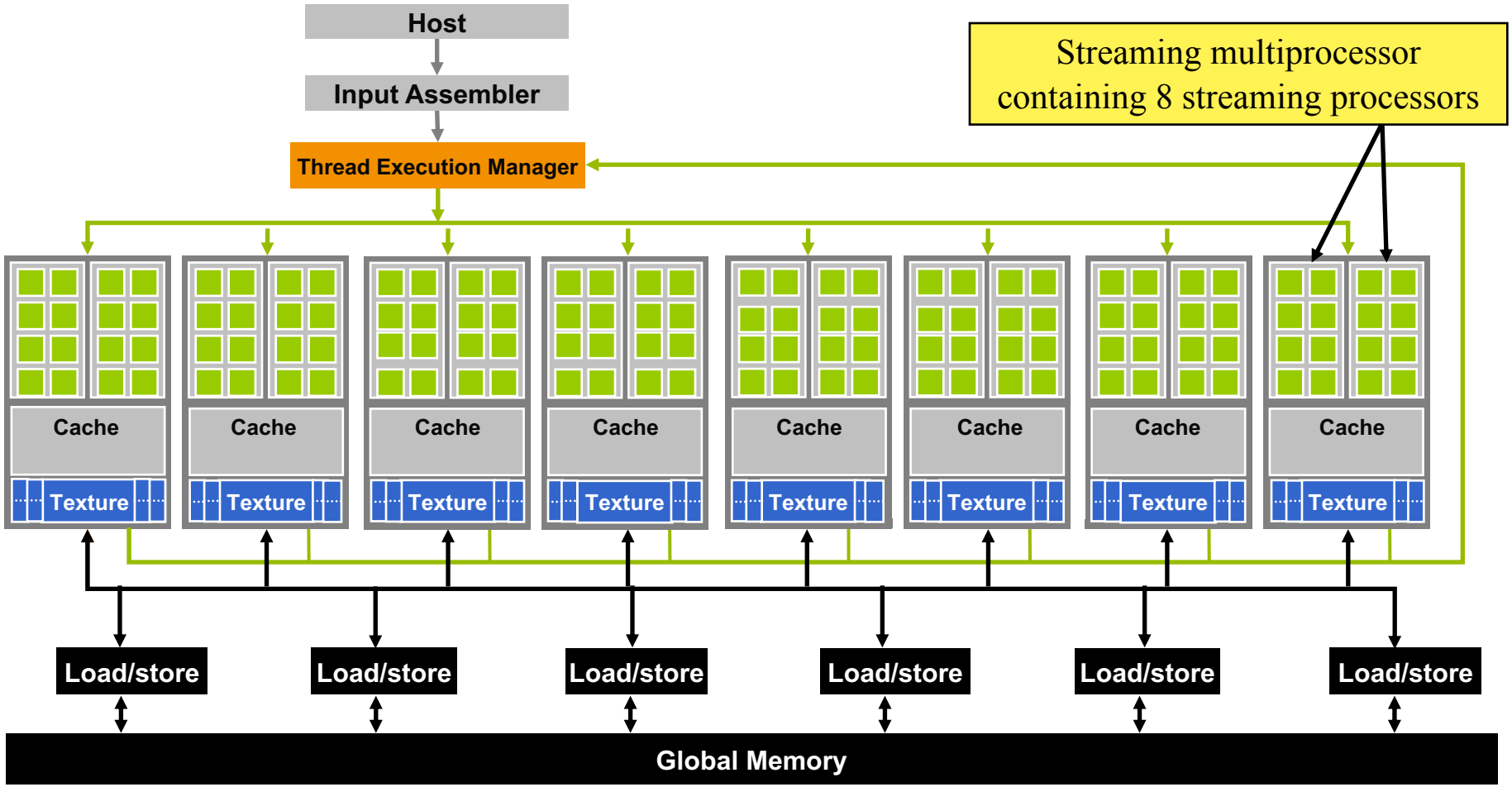


- A quiet revolution and potential build-up
  - Calculation: 1 TFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 100 GB/s vs. 8.4 GB/s
- GPU in every PC and workstation – massive volume and potential impact

CPUs and GPUs have fundamentally different design philosophies



Architecture of a CUDA-capable GPU

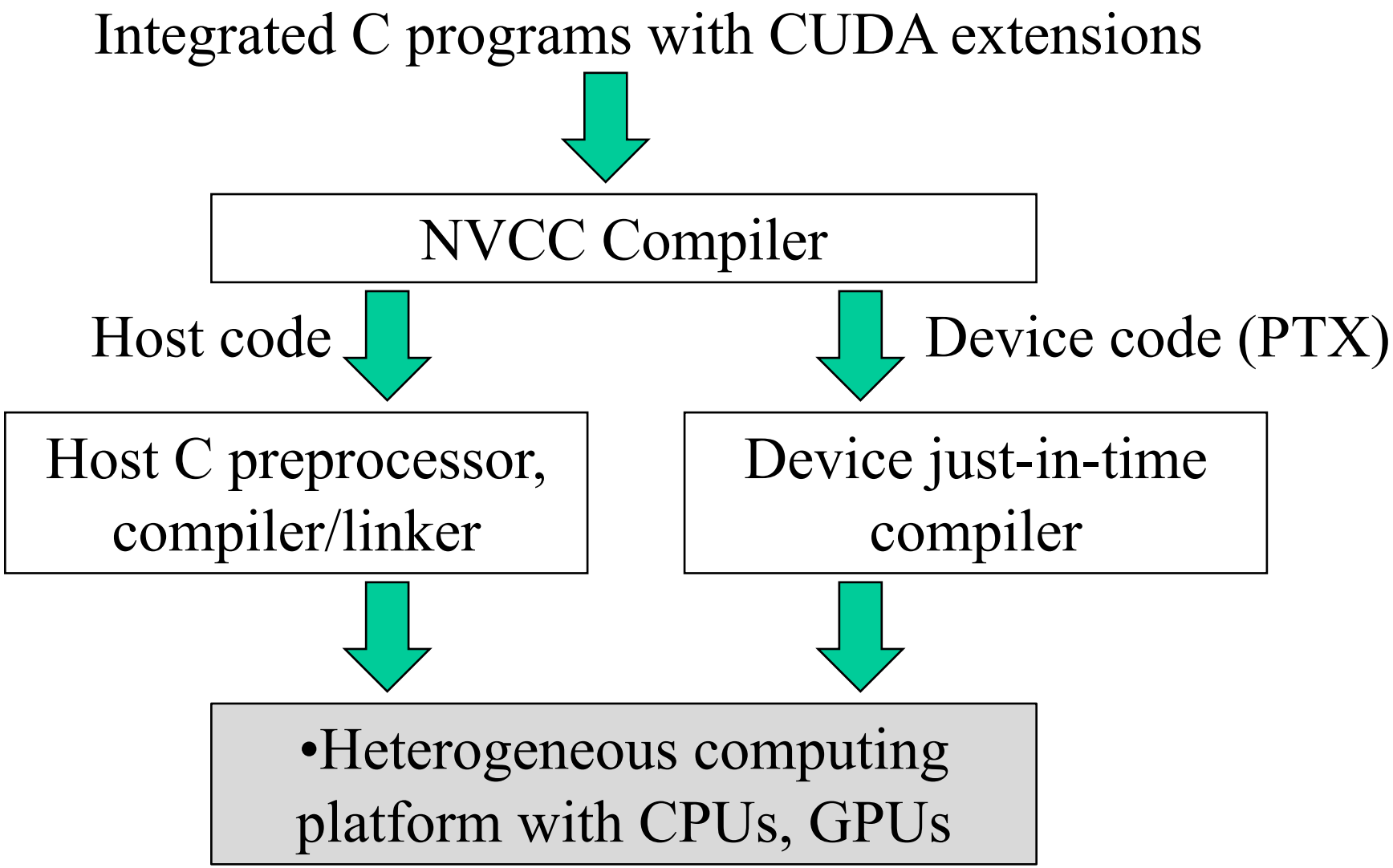


Structure of a CUDA Program

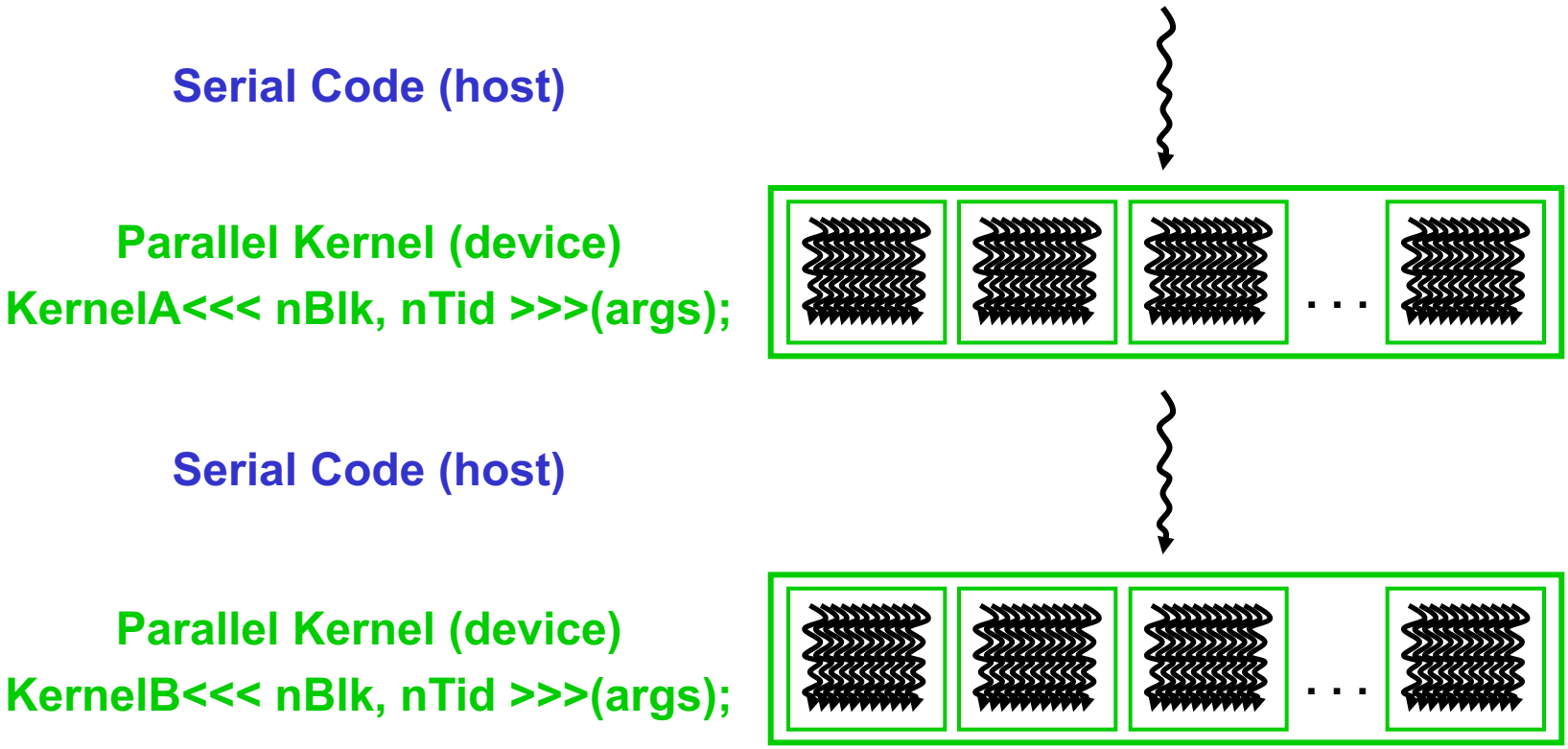
- CUDA programs involve coordination between a *host* (CPU) and one or more *devices* (GPUs).
- A CUDA source file may consist of both host and device code.
- Can add device functions and data declarations to any traditional C code.

Structure of a CUDA Program

- CUDA source files have .cu suffix.
- Device code is marked with CUDA keywords for labelling data-parallel functions called *kernels*.
- Source file is compiled with *nvcc* compiler which gives standard host executable and device code in PTX format.
- PTX is a low-level parallel thread execution virtual machine and instruction set architecture.


Integrated C programs with CUDA extensions
NVCC Compiler
Host code
Device code (PTX)
Host C preprocessor, compiler/linker
Device just-in-time compiler
•Heterogeneous computing platform with CPUs, GPUs

Execution of a CUDA Program



- Host launches KernelA and then KernelB on the GPU.
- All the threads generated by a kernel launch are collectively called a *grid*.

Thread Generation

- Launching a kernel typically launches a large number of threads to exploit data parallelism.
- Can assume that these threads take very few clock cycles to generate and schedule due to efficient hardware support.
- This contrasts with traditional CPU threads that typically take thousands of clock cycles to generate and schedule.

Lab Equipment

- We shall use the Linux lab machines in C/2.08.
- Most of these have one NVidia GeForce RTX 2700 GPU with 2 GB of memory.
- Using CUDA 10.0
- Compile with nvcc:

  nvcc –o code code.cu

Device Properties

- CUDA library provides routines for finding out the number of GPUs in a system, and their properties.
- cudaGetDeviceCount(int *dev_count);
- cudaGetDeviceProperties(cudaDeviceProp *dev_prop, int n)
- cudaGetDeviceProperties populates a C struct with properties of the GPU.
- cudaSetDevice(n) selects GPU n for use.

query.cu

```
#include <cuda.h>
#include <stdio.h
int main (int argc, char
    int ndev, maxtp
    cudaGetDeviceCount(&ndev);
    printf("Number of GPUs = %4d\n",ndev);
    for(int i=0;i<ndev;i++){
        cudaDeviceProp deviceProps;
        cudaGetDeviceProperties(&deviceProps, i);
        maxtpb = deviceProps.maxThreadsPerBlock;
        printf("GPU device %4d:\n\tName: %s:\n", i,deviceProps.name);
        printf("\tCompute capabilities: SM %d.%d\n",
            deviceProps.major, deviceProps.minor);
        printf("\tMaximum number of threads per block: %4d\n”,maxtpb);
        printf("\tMaximum number of threads per SM: %4d\n",
            deviceProps.maxThreadsPerMultiProcessor);
        printf("\tNumber of streaming multiprocessors: %4d\n",
            deviceProps.multiProcessorCount);
        printf("\tClock rate: %d KHz\n",deviceProps.clockRate);
        printf("\tGlobal memory: %lu bytes\n",
            deviceProps.totalGlobalMem);
    }
    cudaSetDevice(0);
}
```

Output on labx03

```
Number of GPUs =    1
GPU device    0:
        Name: GeForce GTX 2700:
        Compute capabilities: SM 7.5
        Maximum number of threads per block:   1024
        Maximum number of threads per SM:   1024
        Number of streaming multiprocessors:    36
        Clock rate: 1620000 KHz
        Global memory: 8335327232 bytes
```

- Can also get GPU information using nvidia-smi at command line

CUDA Example: Vector Add

- Suppose we want to add two vectors of length n.
- Do the actual addition in function vecAdd().
- Initialize vectors to be added in main program, and check resut

Host Sequential Code

```
#include <stdio.h>
#define VLEN 1000
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for(i=0;i<n;i++) h_C[i]=h_A[i]+h_B[i];
}

int main()
{
    int n = VLEN,i,ok;
    float h_A[VLEN], h_B[VLEN], h_C[VLEN];
    for(i=0;i<n;i++){h_A[i]=(float)i;
                     h_B[i]=(float)(i+1);}
    vecAdd(h_A,h_B,h_C,n);
    ok = 1;
    for(i=0;i<n;i++){ ok &= (h_C[i]==(float)(2*i+1));}
    if(ok) printf("Everything worked!\n");
    else printf("Something went wrong!\n");
}
```

Outline of CUDA Code

- Performing the vector addition on the GPU requires coordination between the host and the GPU.

1. The host allocates memory for the 3 vectors on the GPU.
2. The input vectors are copied from the host to the GPU.
3. The host launches the kernel code, and the GPU performs the vector addition.
4. The output vector is copied back from the GPU to the host.
5. The host frees the memory for the 3 vectors on the GPU.

CUDA Device Memory Allocation

- cudaMalloc() ← Used in Step 1
  - Allocates object in the device Global Memory
  - Requires two parameters:
    - **Address of a pointe**r to the allocated object
    - **Size of** the allocated object
- cudaFree() ← Used in Step 5
  - Frees object from device Global Memory
  - Requires one parameter:
    - **Pointer** to freed object
- These are CUDA library calls made on the host to control memory allocation on the GPU.

Host Code: Allocating Device Memory

```
#include <cuda.h>
#include <stdio.h>
#define VLEN 1000

// Later we place the kernel code here

void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    float *d_A, *d_B, *d_C;
    int size=sizeof(float)*n;
    cudaMalloc((void **)&d_A,size);
    // Do steps 2, 3, and 4.
    cudaFree(d_A);
}

int main()
{
    // Same as for the sequential host code
}
```

Used to store addresses of vectors in GPU global memory

And similar for d_B, d_C

Comment on Pointers

- cudaMalloc((void **)&d_A, int size)
- d_A is a location in host memory, and can store a pointer to a float.
- After the call to cudaMalloc(), d_A contains the address in device global memory where the vector is stored.
- First argument to cudaMalloc() must be "&d_A" because a value will be placed there on return.

CUDA Host-Device Data Transfer

- cudaMemcpy() ← Used in Steps 2 and 4
  - Asynchronous memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device

Steps 2 and 4: Transfer between Host and Device

```
// This code slots in after the cudaMalloc() calls

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Launch kernel code to create threads and perform
// vector addition on GPU

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

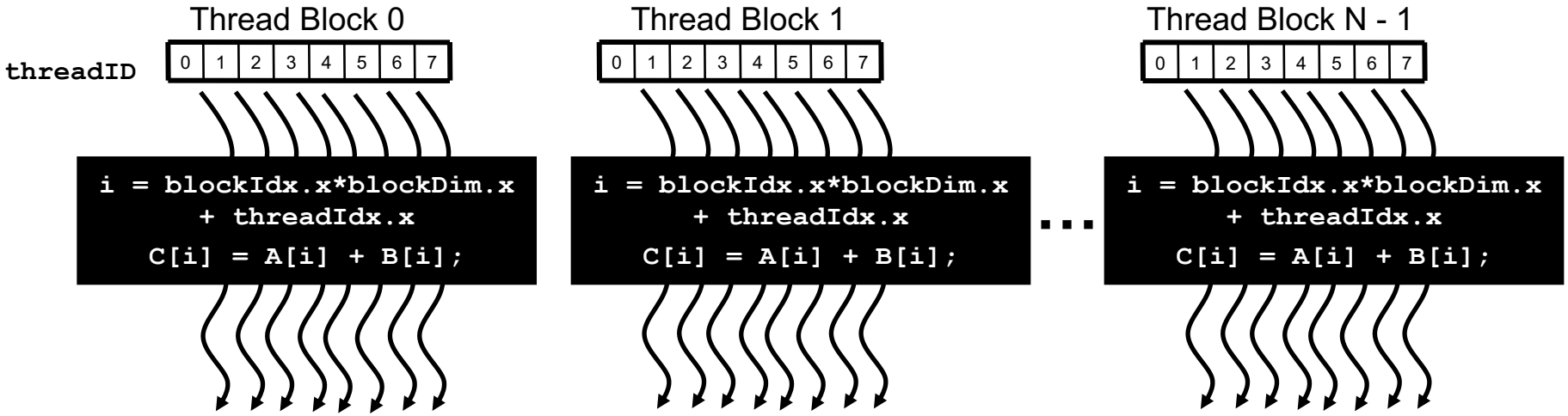
Kernel Functions and Threads

- When a host launches a kernel, the CUDA runtime generates a *grid* of threads.
- A grid is organized as a two-level hierarchy.
- Each grid is organized into an array of *thread blocks*.
- Each block can have up to some maximum number of threads (usually 512 or 1024).

Thread Blocks

Divide monolithic thread array into multiple blocks

- Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization** (more later)
- Threads in different blocks cannot cooperate
- All threads in a grid execute the same kernel code.



Here we show 8 threads per block – typically this would be larger

Thread Blocks

- Number of threads in each block is specified by the host code when the kernel is launched. In the kernel code this is available in the *blockDim* variable.
- Dimensions of thread blocks should be multiples of 32 for hardware efficiency.

Thread ID

- Each thread in a block has a unique *threadIdx* value: 0, 1, 2, …, etc.
- The variable *blockIdx* is used to store the block that a thread is in: 0, 1, 2,…, etc
- Each thread will evaluate C[i] = A[i] + B[i] for one value of i, so each thread needs to figure out its value of i:

```
i = blockIdx.x*blockDim.x + threadIdx.x
```

Thread ID Example

- Suppose there are 8 threads per block, i.e., blockDim.x = 8
- Suppose vectors are of length 1000
- We need ceil(1000/8.0) = 125 thread blocks

```
i = blockIdx.x*blockDim.x + threadIdx.x
```

- For block 0, i runs from 0 to 7.
- For block 1, i runs from 8 to 15.
- For block 124, i runs from 992 to 999.

Thread ID Example 2

- Suppose now vectors are of length 950
- We need ceil(950/8.0) = 119 thread blocks

```
i = blockIdx.x*blockDim.x + threadIdx.x
```

- For block 0, i runs from 0 to 7.
- For block 1, i runs from 8 to 15.
- For block 118, i runs from 944 to 951.
- Last two threads in block 118 are inactive because they have i values > 949.

Kernel Code

CUDA keyword indicating that the function is a kernel and that it can be called from the host to generate a grid of threads on a device. Note double underscores.

```
__global__
void vecAddKernel(float *A, float *B, float *C, int n)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i<n) C[i] = A[i] + B[i];
}
```

This makes sure that the value of i for each thread corresponds to a valid entry in the vectors A, B, and C.

Each thread has a private copy of i.

Note that there is no loop in the kernel code – this has been replaced by the grid of threads. Each thread in the grid corresponds to one iteration of the original loop.

CUDA Keywords for Functions

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	Device	Device
`__global__ void KernelFunc()`	Device	Host
`__host__ float HostFunc()`	Host	Host

- By default all functions in a CUDA program are host functions.
- Can use both __host__ and __device__ in a function declaration. Compiler will produce one version of code for host and another for the device.

Launching the Kernel

- When the host launches the kernel it must specify the configuration parameters:
  - the number of thread blocks in the grid.
  - the number of threads in a block.
- These parameters are placed before the traditional C function arguments between <<< and >>>.

Step 3: Launching the Kernel

```
// This code slots in after the cudaMalloc() calls

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

- We have specified 256 threads per block.
- So we must have **`ceil(n/256.0)`** thread blocks.
- If the length of each vector is 1000, there will be 4 thread blocks of 256 threads each = 1024 threads.
- The last 24 threads in block 3 will be inactive.

The Complete vecAdd() Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    float *d_A, *d_B, *d_C;
    int size=sizeof(float)*n;
    cudaMalloc((void **)&d_A,size);
    cudaMalloc((void **)&d_B,size);
    cudaMalloc((void **)&d_C,size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

CUDA Summary

- Can use __global__, __device__, and __host__ in a function declaration to instruct compiler to generate a kernel function, a device function, or a host function.
- In kernel launch place execution configuration parameters between <<< and >>>. Parameters are the number of blocks and the number of threads per block.
- Pre-defined variables, threadIdx, blockDim and blockIdx, allow threads to identify the area of data each thread should work on.

Compiling with NVCC

- Code for host and GPU is all in one file.
- Code must have .cu suffix, e.g., mycode.cu
- Compile using nvcc:

```
nvcc -o mycode mycode.cu
```

- Can check GPU properties with:

```
deviceQuery
```

CUDA Devices and Threads

- A compute device
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device kernels which run on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Data-Parallel Execution in CUDA

- In CUDA fine-grained, data-parallel threads are used to exploit parallelism.
- When a CUDA kernel is launched a grid of threads is created that all execute the kernel function.
- The kernel function specifies the C statements that each thread executes at runtime.

Thread Index

- Each thread uses a unique coordinate, or thread index, to identify which portion of the data its works on.
- In the vecAdd() example the threads in a block, and the blocks in a grid, were organized one-dimensionally:

```
i = blockIdx.x*blockDim.x + threadIdx.x
```

- In general, threads and blocks can be organized multi-dimensionally to handle multi-dimensional arrays.

Multi-dimensional Blocks

- A grid consists of one or more blocks, and a block consists of one or more threads.
- All the threads in a block share the same block index, **`blockIdx`**.
- In general, a grid is a 3D array of blocks, and each block is a 3D array of threads.
- The programmer can choose to use fewer dimensions by setting unused dimensions to 1.

Kernel Configuration Parameters

- When launching a kernel the grid organization can be specified between <<< and >>>, as follows:

```
someKernel<<<dimGrid,dimBlock>>>(…)
```

- The first parameter specifies the grid dimensions.
- The second parameters specifies the block dimensions.
- In general, **`dimGrid`** and **`dimBlock`** are C structures of type **`dim3`**, with unsigned integer fields x, y and z.

Setting dimGrid and dimBlock

- In the vecAdd example we could have launched the kernel as follows:

```
dim3 dimGrid(128,1,1);
dim3 dimBlock(32,1,1);
vecAddKernel<<<dimGrid,dimBlock>>>(…);
```

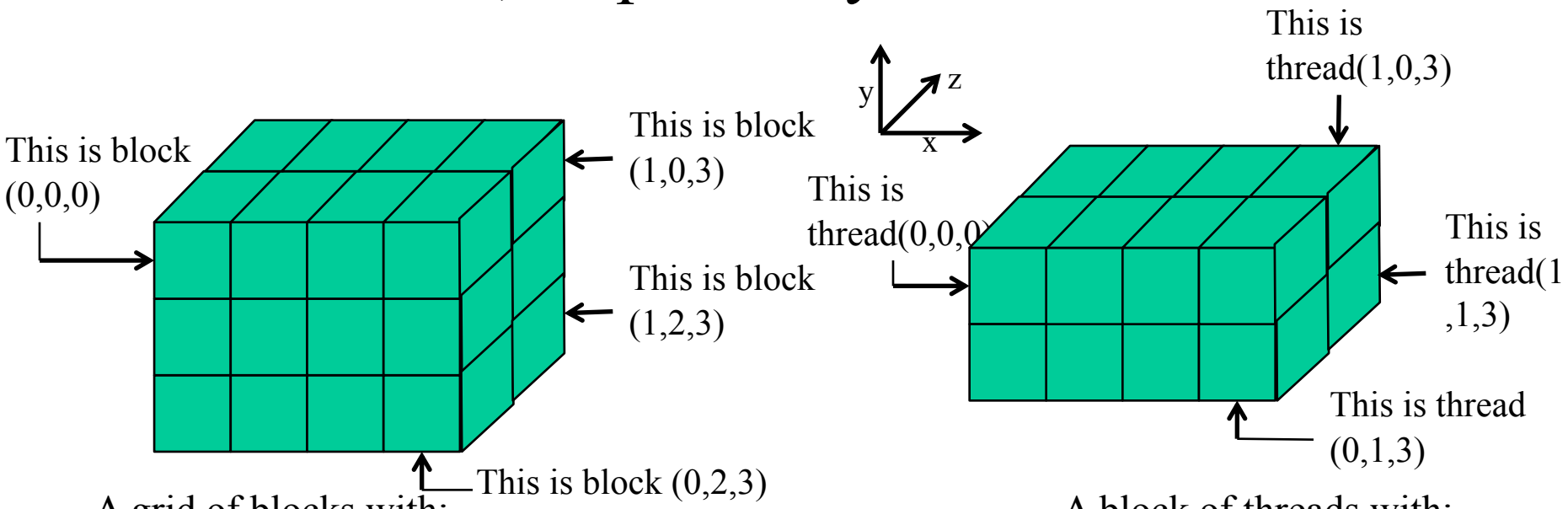
Note that we set the unused dimensions to 1

- This creates a 1D grid of 128 1D blocks, each containing 32 threads.
- For 1D case we can use integers instead of **`dim3`** structs:

```
vecAddKernel<<<128,32>>>(…);
```

gridDim and blockDim

- Within the kernel function gridDim and blockDim give the 3D layout of the blocks and threads, respectively.



A grid of blocks with:

gridDim.x = 4

gridDim.y = 3

gridDim.z = 2

Note that in labelling blocks and threads the highest dimension (z) comes first, then y, and then x. This is the reverse of the ordering used in the configuration parameters.

A block of threads with:

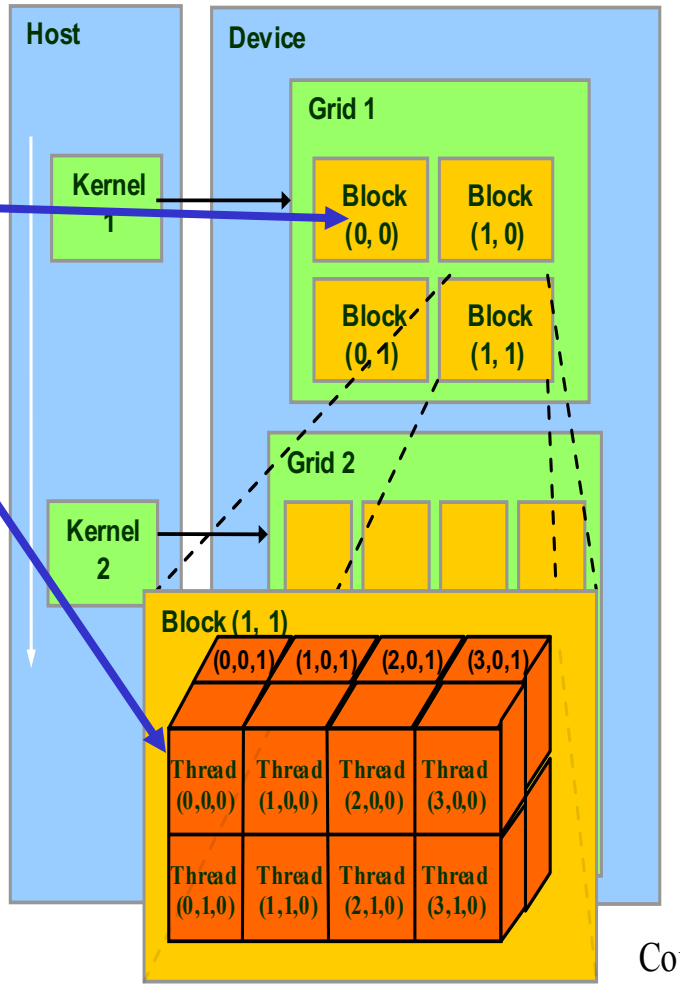
blockDim.x = 4

blockDim.y = 2

blockDim.z = 2

Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D, 2D, or 3D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

Grids of Blocks

- In CUDA C, the allowed values of **`gridDim.x`**, **`gridDim.y`** and **`gridDim.z`** range from 1 to 65536.
- The **`blockIdx.x`** value ranges from 0 to **`gridDim.x-1`**.
- The **`blockIdx.y`** value ranges from 0 to **`gridDim.y-1`**.
- The **`blockIdx.z`** value ranges from 0 to **`gridDim.z-1`**.

Blocks of Threads

- The maximum number of threads in a block is either 512 or 1024, depending on the GPU.
- The **`threadIdx.x`** value ranges from 0 to **`blockDim.x-1`**.
- The **`threadIdx.y`** value ranges from 0 to **`blockDim.y-1`**.
- The **`threadIdx.z`** value ranges from 0 to **`blockDim.z-1`**.

Examples of Thread Blocks

- Suppose we are using a GPU for which the maximum number of threads per block is 512. Consider the following **`blockDim`** (x,y,z) values:

  (512, 1, 1): Valid

  (2, 16, 8): Valid

  (8, 8, 8): Valid

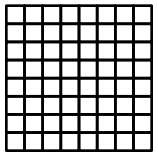
  (16, 8, 8): Not valid as 16*8*8>512.

Mapping Threads to Multi-Dimensional Data

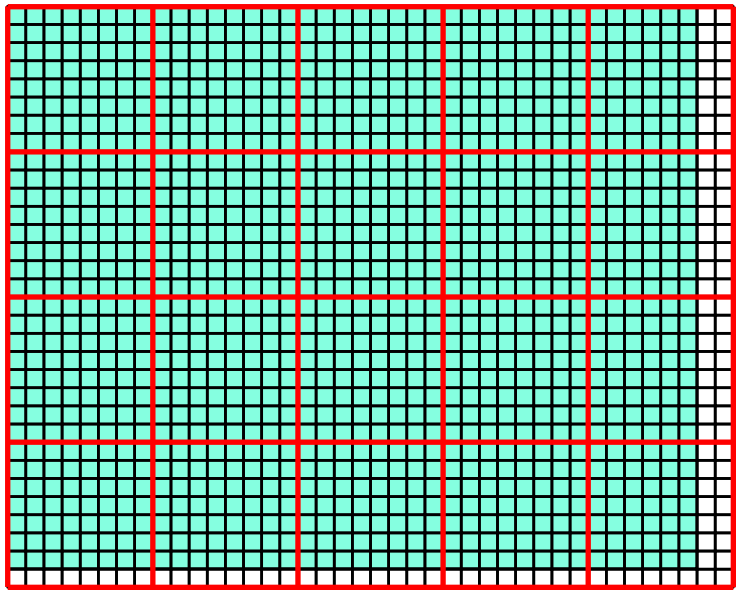
- How threads are organized depends on the nature of the data.
- For example, for a 2D array of pixels it is natural to use a 2D grid consisting of 2D thread blocks.

Mapping Threads Example

- Suppose we have an image of 31x38 pixels (31 rows and 38 columns of pixels), and we choose 8x8 blocks.



An 8x8 thread block



A 5x4 grid of blocks

- We need ceil(38/8.0)=5 blocks in the x-direction, and ceil(31/8.0)=4 blocks in the y-direction.
- The total number of threads is 8x5x8x4 = 1280.
- But the threads in the unshaded region are inactive because they don't correspond to pixels in the image.
- 102 threads are unused.

Kernel Parameters

- Suppose in general that image is mxn pixels.
- Suppose that we have allocated device memory for the input and output images at pointer variables d_Pin and d_Pout, respectively, and have copied the pixel data to device memory.
- Then the host can launch the 2D kernel as follows:

```
dim3 dimGrid(ceil(n/8.0), ceil(m/8.0), 1);
dim3 dimBlock(8,8,1)
imageKernel<<<dimGrid, dimBlock>>>(d_Pin, d_Pout, n, m);
```

Larger Example

- Suppose we have a 750x1000 pixel image and use 8x8 thread blocks.
- There will be ceil(1000/8.0)=125 blocks in the x direction, and ceil(750/8.0)=94 blocks in the y-direction.
- In the kernel function:

  gridDim.x = 125
  gridDim.y = 94
  blockDim.x = 8
  blockDim.y = 8

Multi-Dimensional Arrays in CUDA

- Ideally we would like to access row *j* and column *i* of the array as d_Pin[j][i].
- But since d_Pin is dynamically allocated as a single block of memory, we must access d_Pin as a 1D array.
- If the number of columns is *nc* then we access row *j* and column *i* of the array as d_Pin[j*nc+i].
- This is termed a *row-major layout*.

A Simple Running Example: Matrix Multiplication

- A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs.
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device
  - Assume square matrix for simplicity

Matrix Multiplication

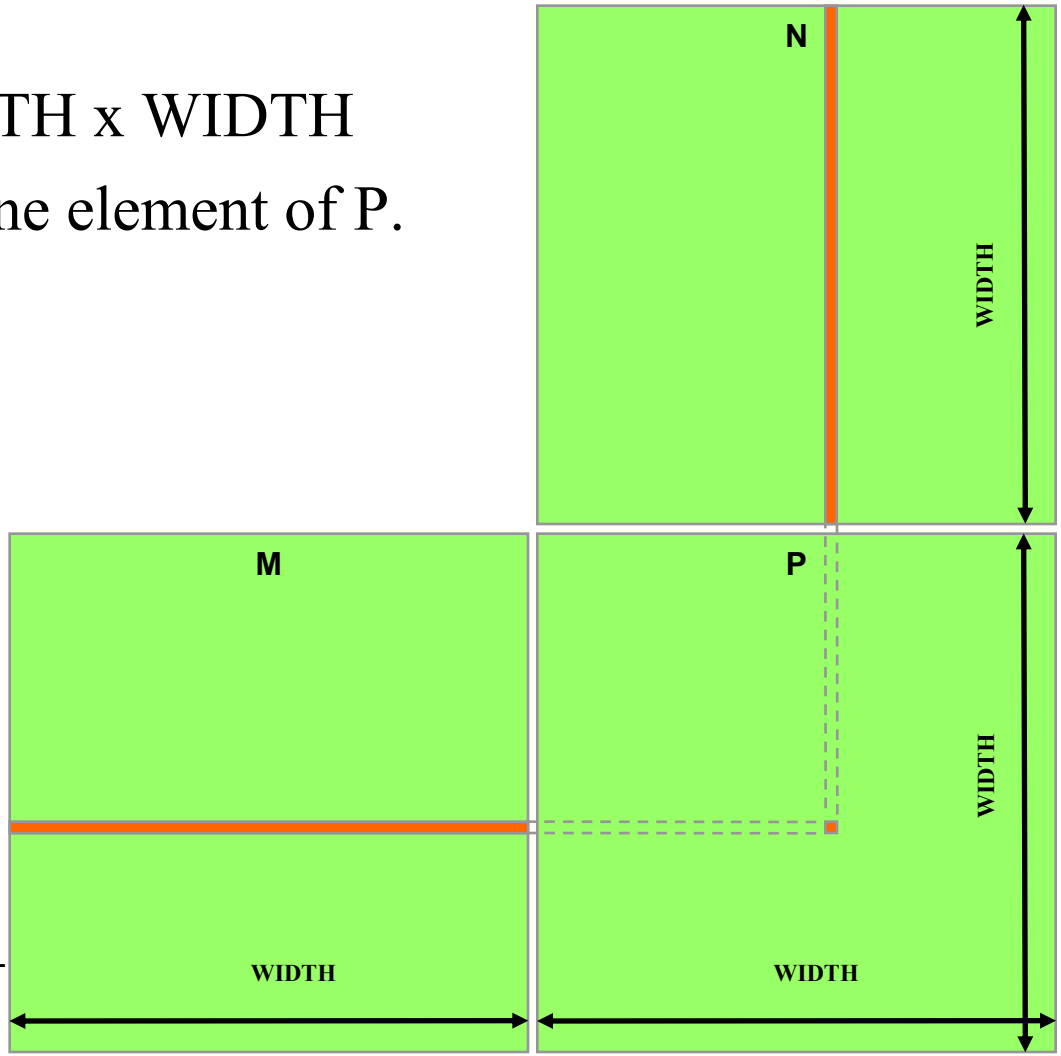
- Matrix multiplication P = M*N for nxn matrices can be expressed is terms of elements as:

$$P_{ji} = \sum_{k=0}^{n-1} M_{jk} N_{ki}$$

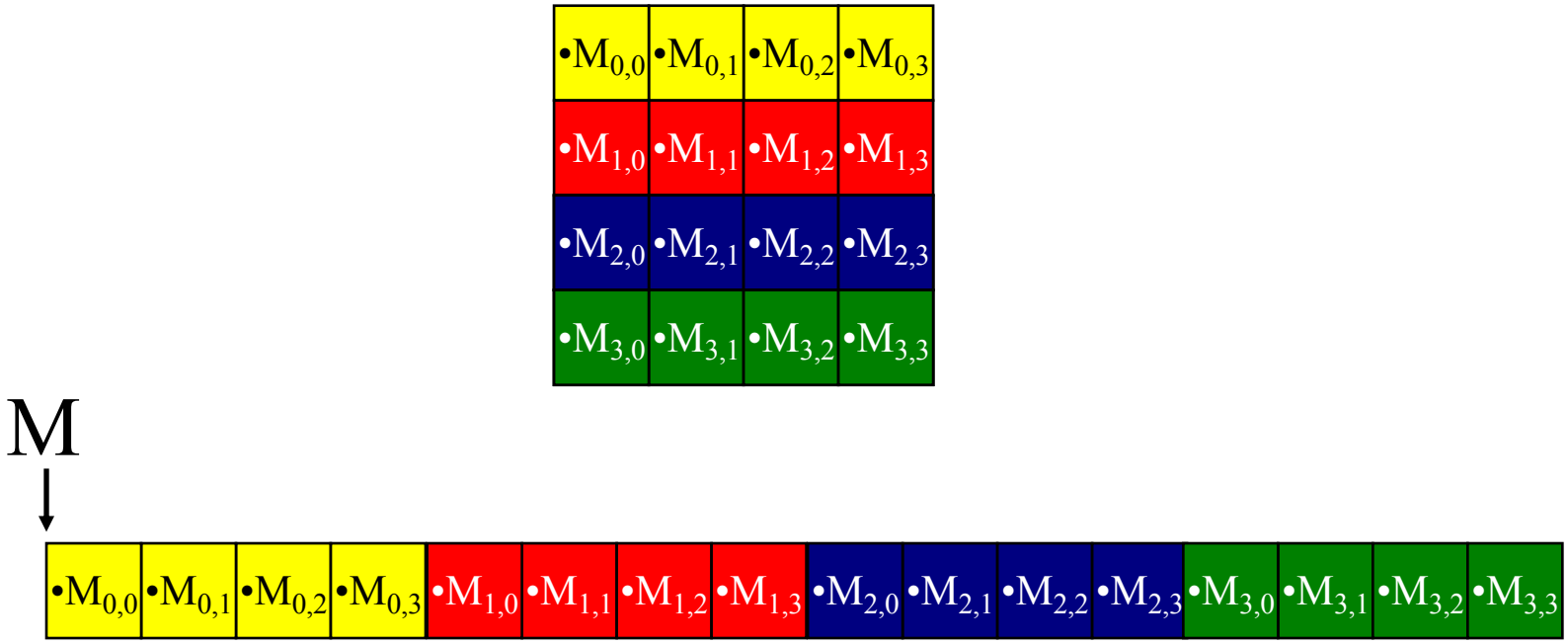
- This means that element P_{ji} is obtained by :
  - taking row j of M, and column i of N
  - multiplying corresponding elements together
  - adding these multiples together

Programming Model: Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- One thread calculates one element of P.

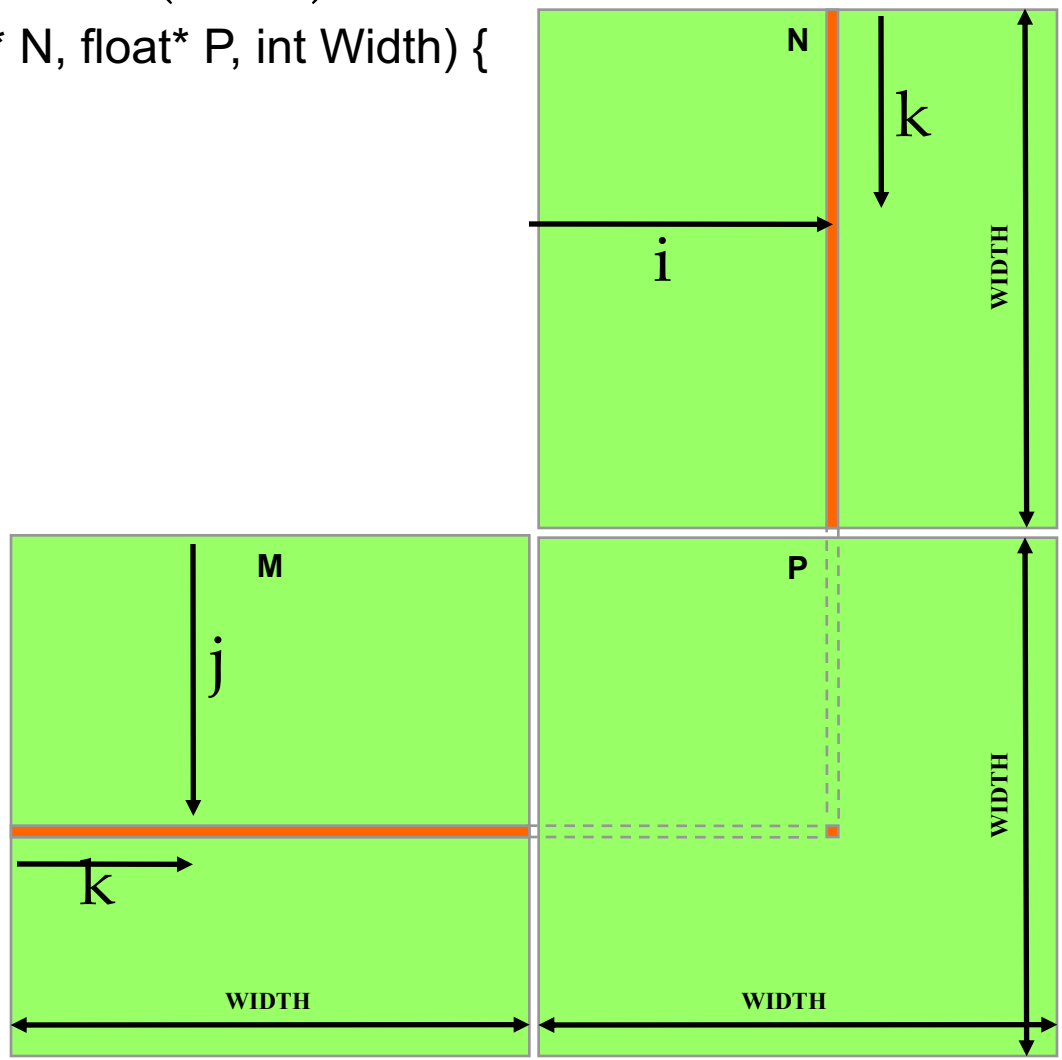


Memory Layout of a Matrix in C



Matrix Multiplication: A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host
void MatrixMulOnHost(float* M, float* N, float* P, int Width) {
  for (int j = 0; j < Width; ++j)
    for (int i = 0; i < Width; ++i) {
      float sum = 0;
      for (int k = 0; k < Width; ++k) {
        float a = M[j * Width + k];
        float b = N[k * Width + i];
        sum += a * b;
      }
      P[j * Width + i] = sum;
    }
}
```



Step 1: Allocate Device Memory and Copy Over Input Matrices

```
void matMul(float *h_M, float *h_N, float *h_P, int Width)
{
    int size = Width * Width * sizeof(float);
    float *d_M, *d_N, *d_P;
// Step 1: Allocate and Load M, N to device memory
    cudaMalloc((void **)&d_M, size);
    cudaMemcpy(d_M, h_M, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **)&d_N, size);
    cudaMemcpy(d_N, h_N, size, cudaMemcpyHostToDevice);
// Step 2: Allocate P on the device
    cudaMalloc((void **)&d_P, size);
// Step 3: Launch kernel here …
// Step 4: Copy back result, and free memory on device
    cudaMemcpy(h_P, d_P, size, cudaMemcpyDeviceToHost);
    cudaFree(d_M); cudaFree(d_N); cudaFree(d_P);
}
```

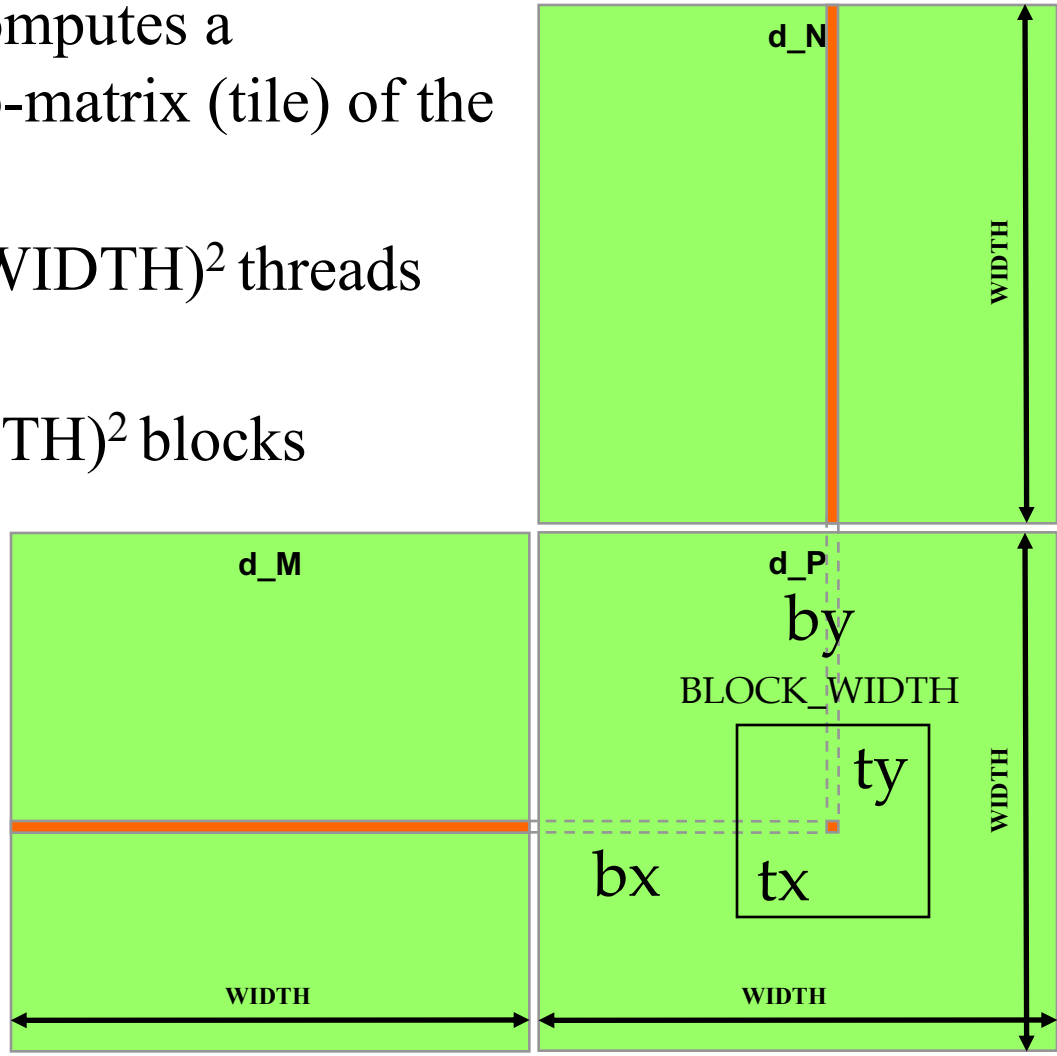
Kernel Function

```
__global__
void matMulKernel(float *d_M, float *d_N, float *d_P, int Width)
{
    int Row = blockIdx.y*blockDim.y + ThreadIdx.y;
    int Col = blockIdx.x*blockDim.x + ThreadIdx.x;
    int k;
    if ((Row<Width)&&(Col<Width)){
        float Pvalue = 0.0;
        for(k=0;k<Width;k++)
            Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
        d_P[Row*Width+Col] = Pvalue;
    }
}
```

Setting the Kernel Configuration Parameters

- Each 2D thread block computes a $(BLOCK\_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - Each has $(BLOCK\_WIDTH)^2$ threads
- Generate a 2D grid of $(WIDTH/BLOCK\_WIDTH)^2$ blocks

You still need to put a loop around the kernel call for cases where WIDTH/BLOCK_WIDTH is greater than max grid size (64K)!



Step 3: Kernel Launch

```
// Setup the execution configuration
   int numBlocks = ceil(Width/(float)BLOCK_WIDTH);
   dim3 dimGrid(numBlocks,numBlocks);
   dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);
// Launch the device computation threads!
   matMulKernel<<<dimGrid, dimBlock>>>(d_M, d_N, d_P, Width);
```

- For BLOCK_WIDTH=16 and Width=500:

  numBlocks = 32

  Total number of threads = 32*32*16*16 = 262144.

  Number of unused threads = 12144.

Synchronizing Threads

- CUDA allows threads in the same block to coordinate their activities by calling a barrier synchronization function:

```
__syncthreads( )
```

- When a kernel function calls **`__syncthreads()`** all the threads in a block wait at the calling location until every thread in the block reaches the location.

Synchronizing Threads

- CUDA does not provide any way of synchronizing threads in different blocks.
- This allows the CUDA runtime to execute blocks in any order relative to each other since none of them will need to wait for each other.
- Thus, on devices with more resources, more blocks can be run simultaneously.
- The same code can run on different CUDA devices with transparent scalability.

Assigning Resources to Blocks

- Threads are assigned to resources on a block-by-block basis.
- In current hardware the execution resources are organized into streaming multiprocessors (SMs).
- Multiple blocks can be assigned to each SM, and can run simultaneously up to some limit.
- The CUDA runtime maintains a list of blocks that need to be run and assigns them to SMs as previous blocks are completed.

Warps

- Once a block is assigned to a SM it is further divided into *warps*.
- A warp is a set of threads with consecutive threadIdx values.
- Usually there are 32 threads in a warp, but the number is device dependent and can be obtained from **`dev_prop.warpSize`**.
- The warp is the unit of thread scheduling in SMs.
- Warps are not part of the CUDA API, but knowing about them can help optimize performance.

Warps and Thread Divergence

- All threads in a warp follow the SIMD execution model, so at any instant an instruction is fetched and executed by all the threads in the warp.
- If the threads in a warp execute different branches of an *if* statement performance will be affected – this is called *thread divergence*.

Warps and Latency Tolerance

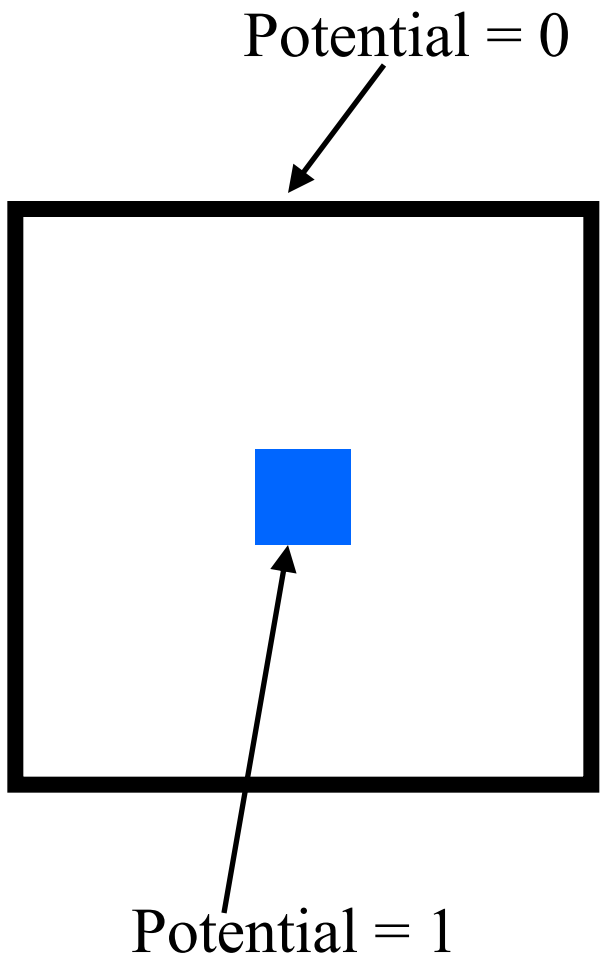
- When an instruction executed by the threads in a warp needs to wait for the result of a previously initiated long-latency operation (such as a global memory access), the warp is not selected for execution.
- Another resident warp not waiting for results is selected for execution.
- This mechanism of filling the latency time of operations with work from other threads is called *latency tolerance* or *latency hiding*.

Choosing the Right Block Size

- Suppose we have a CUDA device that allows up to 8 blocks and 1024 threads per SM, and up to 512 threads per block.
- For matrix multiplication should we use 8x8, 16x16, or 32x32 blocks?
- If we use 8x8 blocks each block has 64 threads. Since there is a limitation of 8 blocks per SM we would have 64x8=512 threads in each SM. SM resources will be underutilized, and there won't be enough warps to hide the latency of long operations.
- If we use 16x16 blocks we have 256 threads per block and so can fit 4 blocks on an SM. Good choice.
- 32x32 blocks would have 1024 threads per block which exceeds the maximum of 512 threads per block.

Laplace Equation Problem

- The next problem we shall look at may be used to determine the electric field around a conducting object held at a fixed electrical potential inside a box also at a fixed electrical potential.
- As with the vibrating string problem, this problem can also be expressed mathematically as a partial differential equation, known as the Laplace equation.
- We shall design a parallel MPI program to solve the partial differential equation.



Sequential Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NSTEPS 500
#define NPTSX 200
#define NPTSY 200
int main (int argc, char *argv[])
{
   float *h_phi;
   float *h_oldphi;
   int *h_mask;
   int nsize1=sizeof(float)*NPTSX*NPTSY;
   int nsize2=sizeof(int)*NPTSX*NPTSY;
   h_phi = (float *)malloc(nsize1);
   h_oldphi = (float *)malloc(nsize1);
   h_mask = (int *)malloc(nsize2);
   setup_grid (h_oldphi, NPTSX, NPTSY, h_mask);
   performUpdates(h_phi,h_oldphi,h_mask,NPTSX,NPTSY,NSTEPS);
   output_array (h_phi, NPTSX, NPTSY);
}
```

Sequential Code: Updates

```
void performUpdates(float *h_phi, float * h_oldphi, int
*h_mask, int nptsx, int nptsy, int nsteps)
{
    int x, i, j, k;
    for(k=0;k<nsteps;++k){
        for(j=0;j<nptsy;j++)
            for(i=0;i<nptsx;i++){
                x = j*nptsx+i;
                h_oldphi[x] = h_phi[x];
            }
         for(j=0;j<nptsy;j++)
            for(i=0;i<nptsx;i++){
                x = j*nptsx+i;
                if(h_mask[x]) h_phi[x] = 0.25f*(
                 h_oldphi[x+1]+h_oldphi[x-1]+h_oldphi[x+nptsx]+
                 h_oldphi[x-nptsx]);
            }
    }
}
```

Copy current solution to old solution

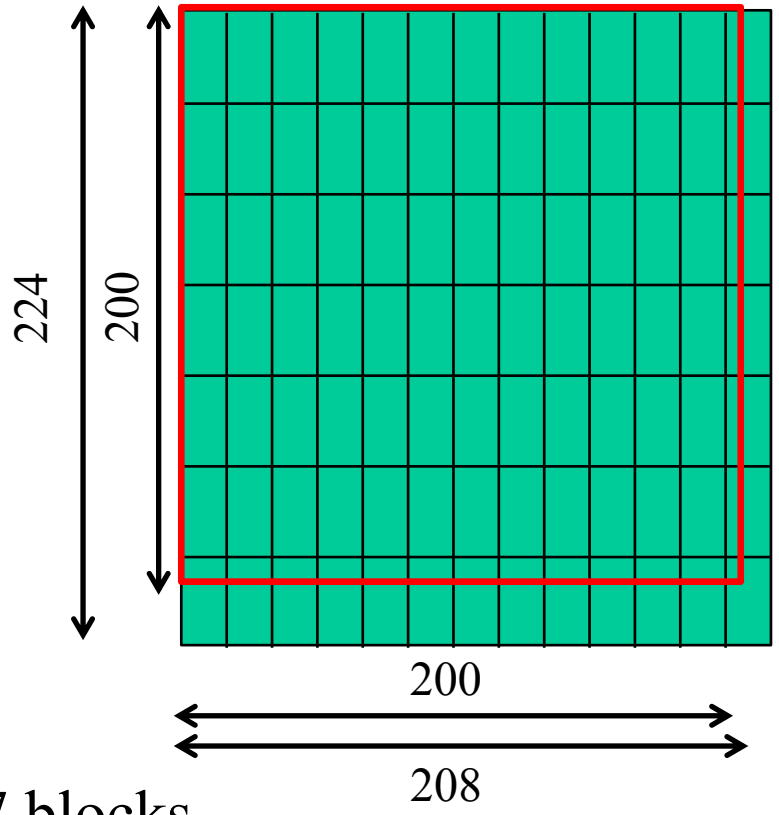
Update solution

Device Memory Allocation and Copying

```
void performUpdates(float *h_phi, float *h_oldphi,
       int *h_mask, int nptsx, int nptsy, int nsteps)
{
    float *d_phi, *d_oldphi;
    int *d_mask;
    int k;
    int sizef = sizeof(float)*nptsx*nptsy;
    int sizei = sizeof(int)*nptsx*nptsy;
    cudaMalloc((void **)&d_phi,sizef);
    cudaMalloc((void **)&d_oldphi,sizef);
    cudaMalloc((void **)&d_mask,sizei);
    cudaMemcpy(d_phi,h_phi,sizef,cudaMemcpyHostToDevice);
    cudaMemcpy(d_mask,h_mask,sizei,cudaMemcpyHostToDevice);
   // Set up thread blocks and launch kernel here
    cudaMemcpy(h_phi,d_phi,sizef,cudaMemcpyDeviceToHost);
    cudaFree(d_phi); cudaFree(d_oldphi); cudaFree(d_mask);
}
```

Setting Up the Thread Blocks

```
#define TX 16
#define TY 32
……
int by = ceil(nptsy/(float)TY);
int bx = ceil(nptsx/(float)TX);
dim3 dimBlock(TX,TY,1);
dim3 dimGrid(bx,by,1);
```



In our case (nptsx = nptsy = 200):

- Each block has 16x32 threads
- by = 7, bx = 13, so there are 13x7 blocks
- Total threads = 16*13 x 32*7 = 208x224 = 46592
- 6592 threads are unused.

Launching the Kernels

```
void performUpdates(float *h_phi, float * h_oldphi,
          int *h_mask, int nptsx, int nptsy, int nsteps)
{
    // Allocate arrays in device memory
    // Copy oldphi and mask arrays to device
    for(k=0;k<nsteps;++k){
        doCopyKernel<<<dimGrid,dimBlock>>>
                    (d_phi,d_oldphi,d_mask,nptsx,nptsy);
        performUpdatesKernel<<<dimGrid,dimBlock>>>
                    (d_phi,d_oldphi,d_mask,nptsx,nptsy);
     }
   // Copy back solution from device
   // Free arrays in device memory
}
```

Copy current solution to old solution

Update solution

doCopyKernel

```
__global__
void doCopyKernel(float *d_phi, float *d_oldphi,
                  int *d_mask, int nptsx, int nptsy)
{
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
    int x = Row*nptsx+Col;

    if(Col<nptsx && Row<nptsy)
         d_oldphi[x] = d_phi[x];
}
```

performUpdatesKernel

```
__global__
void performUpdatesKernel(float *d_phi, float
*d_oldphi, int *d_mask, int nptsx, int nptsy)
{
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
    int x = Row*nptsx+Col;
    int xm = x-nptsx;
    int xp = x+nptsx;

    if(Col<nptsx && Row<nptsy)
        if (d_mask[x]) d_phi[x] = 0.25f*(
            d_oldphi[x+1]+d_oldphi[x-1]+
            d_oldphi[xp]+d_oldphi[xm]);
}
```

Avoid writing off of end of array

Each thread in use updates location x

Why Not Just Use One Kernel?

```
__global__
void performUpdatesKernel(float *d_phi, float *d_oldphi,
int *d_mask, int nptsx, int nptsy)
{
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
    int x = Row*nptsx+Col;
    int xm = x-nptsx;
    int xp = x+nptsx;

    if(Col<nptsx && Row<nptsy)
        d_oldphi[x] = d_phi[x];
        if (d_mask[x]){
            d_phi[x] = 0.25f*(d_oldphi[x+1]+d_oldphi[x-1]+
            d_oldphi[xp]+d_oldphi[xm]);

        }
}
```

Some threads could update d_phi[x] before other threads have copied neighboring values to d_oldphi.

Global Memory Accesses

- Having many threads available for execution can hide long access times to global memory.
- However, traffic congestion in the global memory access paths can still prevent all but a few threads from making progress.
- This can make some of the streaming multiprocessors idle.
- Aim to use other types of memory to avoid access to global memory.

Compute-To-Global-Memory-Access Ratio

- The CGMA ratio is the number of floating point operations performed for each global memory access.
- In the matrix multiply kernel we do:

```
for(k=0;k<Width;k++)
    Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
```

- This fetches 2 values from global memory and performs 2 floating point operations.
- So CGMA ratio is 1:1 or 1.0.

CGMA Ratio for Laplace Solver

- In the Laplace solver update kernel we do

```
if (d_mask[x]) d_phi[x] = 0.25f*(d_oldphi[x+1]
        +d_oldphi[x-1]+d_oldphi[xp]+d_oldphi[xm]);
```

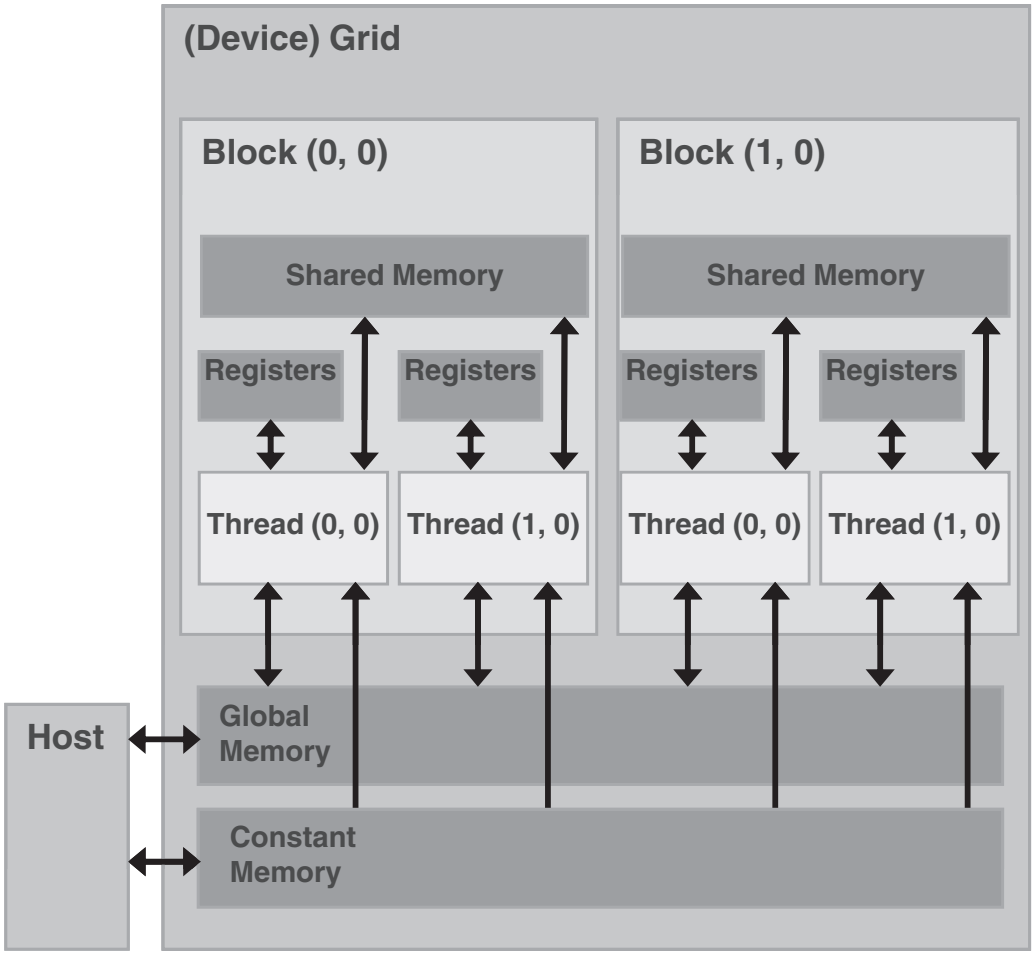
- This fetches 5 values from global memory and writes 1 value to global memory, while performing 4 floating point operations.
- So CGMA ratio is 4:6 or 0.6667.
- We expect better performance when many operations are done for each memory access.
- So we want the CGMA ratio to be high.

Memory Access Limitations

- Suppose the global memory bandwidth is 200 GB/s.
- If a floating point value is 4 bytes we can load no more than $(200/4)x10^9=50x10^9$ floats per second.
- Thus, maximum performance is $50x10^9$xCGMA.
- For matrix multiply this is 50 Gflop/s.
- This is much less than typical peak performance of GPU of about 1000 Gflop/s.

CUDA Device Memory Model

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - Transfer data to/from per-grid global and constant memories



CUDA Memory Types

- Want to use different types of memory to increases CGMA ratio.
- *Constant memory* supports short-latency, high-bandwidth, read-only access.
- *Registers* and *shared memory* are on-chip memories.
- Shared memory is allocated to thread blocks.

Registers

- Registers are allocated to individual threads, and are used in kernel functions to hold frequently-accessed variables that are private to each thread.
- Performing an operation on register variables does not require a memory load operation.
- Number of registers available to each thread is small.
- Accessing variables in registers also uses much less energy than accessing global memory.

Shared Memory

- Shared memory is allocated to thread blocks, and provides an efficient way for threads to cooperate.
- Shared memory is on-chip so can be accessed with much lower latency and higher bandwidth than global memory.
- When a variable in shared memory is accessed a load memory operation is performed, so shared memory has longer latency and lower bandwidth than registers.
- Accesses to registers and shared memory do not contribute to the CGMA ratio.

Device Memory Declarations

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Global	Thread	Kernel
__device__ __shared__ int SharedVar;	Shared	Block	Kernel
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application

- If a variable's scope is "Thread" a private version of that variable is created for every thread.
- If a variable's lifetime is "Kernel" it must be declared in the kernel function body. If the kernel is invoked several times, the value of the variable is not maintained across these invocations.

Shared Variables

- Declaration of a shared variable is in a kernel or device function.
- All threads in a block see the same version of a shared variable.
- A private version of the shared variable is created for, and used by, each thread block.
- When a kernel terminates the contents of its shared variables cease to exist.
- Access to shared variables is fast and highly parallel.

Constant Variables

- Declaration of a constant variable must be outside any function body.
- Often used for variables that provide input values for kernel functions.
- Access can be extremely fast and parallel.
- Current total size for constant variables is 65536 bytes.
- A constant variable can be seen by all threads throughout an application.

Kernel Function For Matrix Multiply

```
__global__
void matMulKernel(float *d_M, float *d_N, float *d_P, int Width)
{
    int Row = blockIdx.y*blockDim.y + ThreadIdx.y;
    int Col = blockIdx.x*blockDim.x + ThreadIdx.x;
    int k;
    if ((Row<Width)&&(Col<Width)){
        float Pvalue = 0.0;
        for(k=0;k<Width;k++)
            Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
        d_P[Row*Width+Col] = Pvalue;
    }
}
```

d_P, d_M, and d_N are pointers to locations in global memory.

Row, Width, k, and Pvalue are all automatic variables and so each thread has a private version. They are stored in registers so access is fast.

Tiled Algorithms

- Global memory: large but slow.
- Shared memory: small but fast.
- Partition data into subsets, or *tiles*, so that each fits into shared memory.
- It is important that the kernel computation can be done independently on each tile.

Tiled Matrix Multiply: P = MN

- Use grid of 2x2 blocks, with each block consisting of 2x2 threads.

block (0,0)

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

$$P_{0,0} = M_{0,0}*N_{0,0}+M_{0,1}*N_{1,0}+M_{0,2}*N_{2,0}+M_{0,3}*N_{3,0}$$
$$P_{0,1} = M_{0,0}*N_{0,1}+M_{0,1}*N_{1,1}+M_{0,2}*N_{2,1}+M_{0,3}*N_{3,1}$$
$$P_{1,0} = M_{1,0}*N_{0,0}+M_{1,1}*N_{1,0}+M_{1,2}*N_{2,0}+M_{1,3}*N_{3,0}$$
$$P_{1,1} = M_{1,0}*N_{0,1}+M_{1,1}*N_{1,1}+M_{1,2}*N_{2,1}+M_{1,3}*N_{3,1}$$

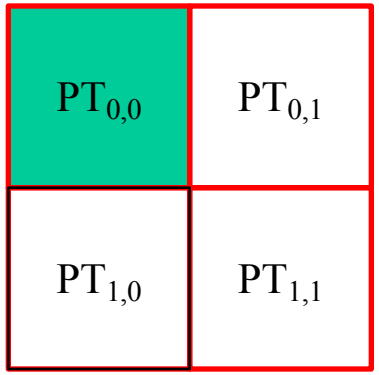
Access order →				
$thread_{0,0}$	$M_{0,0}*N_{0,0}$	$M_{0,1}*N_{1,0}$	$M_{0,2}*N_{2,0}$	$M_{0,3}*N_{3,0}$
$thread_{0,1}$	$M_{0,0}*N_{0,1}$	$M_{0,1}*N_{1,1}$	$M_{0,2}*N_{2,1}$	$M_{0,3}*N_{3,1}$
$thread_{1,0}$	$M_{1,0}*N_{0,0}$	$M_{1,1}*N_{1,0}$	$M_{1,2}*N_{2,0}$	$M_{1,3}*N_{3,0}$
$thread_{1,1}$	$M_{1,0}*N_{0,1}$	$M_{1,1}*N_{1,1}$	$M_{1,2}*N_{2,1}$	$M_{1,3}*N_{3,1}$

- Every M and N element is accessed twice in executing block (0,0).
- For NxN blocks we can reduce global accesses by a factor of N.

Tiled Matrix Multiply Algorithm

- Assume grid of 2x2 blocks, with each block consisting of 2x2 threads.
- Threads collaboratively load M and N elements into shared memory before using them.
- Must remember that size of shared memory is small.
- Divide M and N matrices into smaller tiles. For example, make tiles the same size as blocks.

Tile (0,0)



Phase 1 Phase 2

$PT_{0,0} = MT_{0,0}*NT_{0,0} + MT_{0,1}*NT_{1,0}$

$PT_{0,1} = MT_{0,0}*NT_{0,1} + MT_{0,1}*NT_{1,1}$

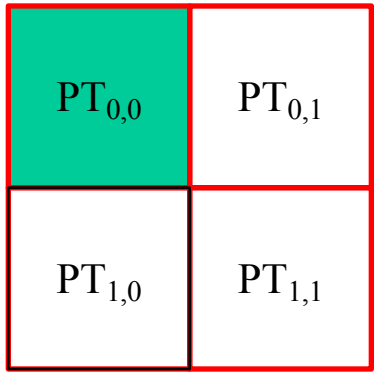
$PT_{1,0} = MT_{1,0}*NT_{0,0} + MT_{1,1}*NT_{1,0}$

$PT_{1,1} = MT_{1,0}*NT_{0,1} + MT_{1,1}*NT_{1,1}$

Tiled Matrix Multiply Algorithm

- Do computation in phases.
- Load tiles for Phase 1 into shared memory.
- Multiply tiles for Phase 1 together.
- Load tiles for Phase 2 into shared memory.
- Multiply tiles for Phase 2 together, and add to result from Phase 1.

Tile (0,0)



Phase 1 Phase 2

$PT_{0,0} = MT_{0,0}*NT_{0,0} + MT_{0,1}*NT_{1,0}$

$PT_{0,1} = MT_{0,0}*NT_{0,1} + MT_{0,1}*NT_{1,1}$

$PT_{1,0} = MT_{1,0}*NT_{0,0} + MT_{1,1}*NT_{1,0}$

$PT_{1,1} = MT_{1,0}*NT_{0,1} + MT_{1,1}*NT_{1,1}$

Execution Phases for Block (0,0)

	Phase 1		Phase 2	
$thread_{0,0}$	$M_{0,0} \rightarrow Mds_{0,0}$ $N_{0,0} \rightarrow Nds_{0,0}$	Pvalue+= $Mds_{0,0}*Nds_{0,0}+$ $Mds_{0,1}*Nds_{1,0}$	$M_{0,2} \rightarrow Mds_{0,0}$ $N_{2,0} \rightarrow Nds_{0,0}$	Pvalue+= $Mds_{0,0}*Nds_{0,0}+$ $Mds_{0,1}*Nds_{1,0}$
$thread_{0,1}$	$M_{0,1} \rightarrow Mds_{0,1}$ $N_{0,1} \rightarrow Nds_{0,1}$	Pvalue+= $Mds_{0,0}*Nds_{0,1}+$ $Mds_{0,1}*Nds_{1,1}$	$M_{0,3} \rightarrow Mds_{0,1}$ $N_{2,1} \rightarrow Nds_{0,1}$	Pvalue+= $Mds_{0,0}*Nds_{0,1}+$ $Mds_{0,1}*Nds_{1,1}$
$thread_{1,0}$	$M_{1,0} \rightarrow Mds_{1,0}$ $N_{1,0} \rightarrow Nds_{1,0}$	Pvalue+= $Mds_{1,0}*Nds_{0,0}+$ $Mds_{1,1}*Nds_{1,0}$	$M_{1,2} \rightarrow Mds_{1,0}$ $N_{3,0} \rightarrow Nds_{1,0}$	Pvalue+= $Mds_{1,0}*Nds_{0,0}+$ $Mds_{1,1}*Nds_{1,0}$
$thread_{1,1}$	$M_{1,1} \rightarrow Mds_{1,1}$ $N_{1,1} \rightarrow Nds_{1,1}$	Pvalue+= $Mds_{1,0}*Nds_{0,1}+$ $Mds_{1,1}*Nds_{1,1}$	$M_{1,3} \rightarrow Mds_{1,1}$ $N_{3,1} \rightarrow Nds_{1,1}$	Pvalue+= $Mds_{1,0}*Nds_{0,1}+$ $Mds_{1,1}*Nds_{1,1}$

Time 

- Each thread has a private copy of Pvalue.
- In phase 2 we overwrite the shared memory with new tile data.
- Each global memory value loaded into shared memory is used multiple times.

Tiled Kernel Function

```
__global__
void matMulTiledKernel(float *d_M, float *d_N, float *d_P, int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int Row = by*TILE_WIDTH + ty;
    int Col = bx*TILE_WIDTH + tx;
    int k, m;
    float Pvalue = 0.0;
    for(m=0;m<Width/TILE_WIDTH;++m){
        Mds[ty][tx] = d_M[Row*Width+m*TILE_WIDTH+tx];
        Nds[ty][tx] = d_N[(m*TILE_WIDTH+ty)*Width+Col];
        __syncthreads();
        for(k=0;k<TILE_WIDTH;k++) Pvalue += Mds[ty][k]*Nds[k][tx];
        __syncthreads();
     }
     d_P[Row*Width+Col] = Pvalue;
}
```

Assume Width is exactly divisible by TILE_WIDTH

bx, by, tx, ty, Row, Col, Pvalue, k, and m are all placed in registers

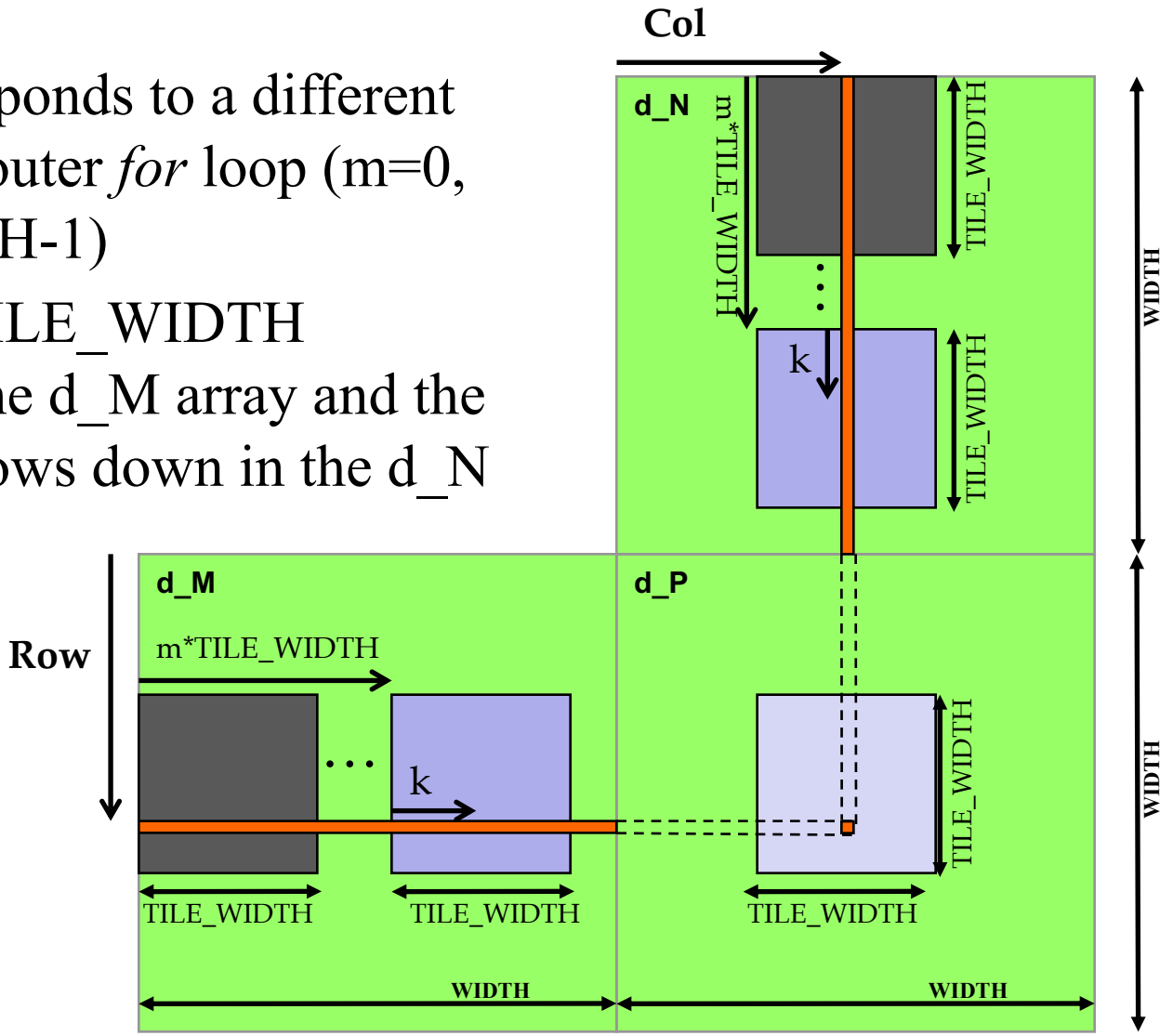
There are Width/TILE_WIDTH phases

Copy current M and N tiles to shared memory

Need to synchronize threads in same block to make sure all threads have (1) copied their value to shared memory, and (2) finished contributing to Pvalue

Calculation of Matrix Indices

- Each phase corresponds to a different value of m in the outer *for* loop (m=0, 1,…,TILE_WIDTH-1)
- Tile m starts m*TILE_WIDTH columns over in the d_M array and the same number of rows down in the d_N array.
- As k increases in the inner *for* loop we pick out elements in row ty of d_M and column tx of d_N.

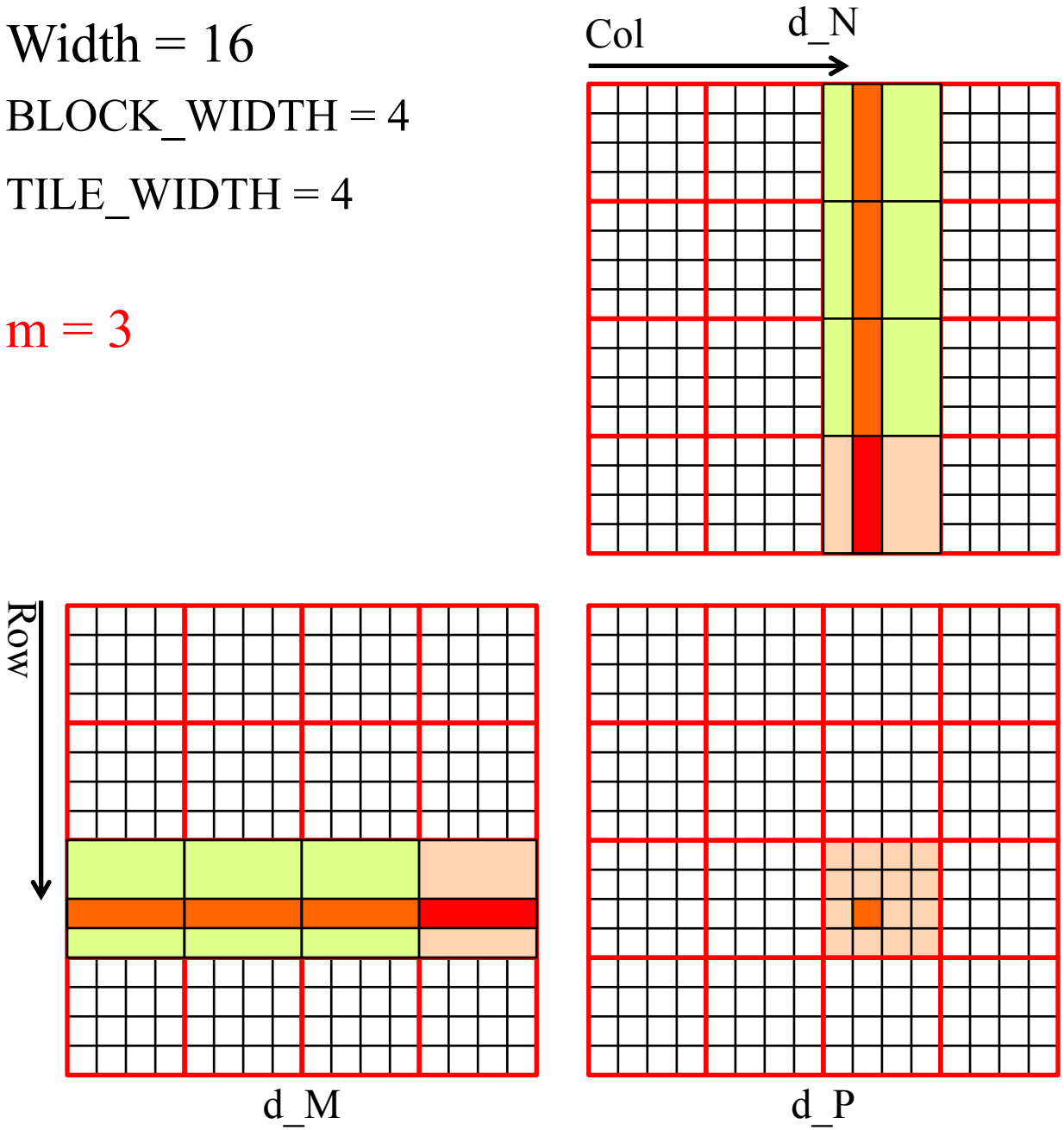


Width = 16

BLOCK_WIDTH = 4

TILE_WIDTH = 4

m = 3



Performance Gained from Tiling

- For matrix multiplication the number of global memory accesses is reduced by a factor of TILE_WIDTH.
- For TILE_WIDTH = 16 the CGMA ratio is 16.0.
- Suppose the global memory bandwidth is 200 GB/s.
- If a floating point value is 4 bytes we can load no more than $(200/4)x10^9=50x10^9$ floats per second.
- Thus, maximum performance is $50x10^9$xCGMA.
- For tiled matrix multiply this is 800 Gflop/s, compared with 50 Gflop/s for the untiled version.

How Memory Can Limit Performance

- Using registers and shared memory can reduced the number of accesses to global memory, but you must be careful not to exceed their capacity.
- In general the more resources (registers, shared memory) each thread requires, the fewer threads can reside simultaneously in a streaming multiprocessor (SM).
- Having fewer threads can reduce the number of warps that are available for scheduling, which can reduce the SM's ability to find useful work in the presence of long-latency operations.

Register Usage Example

- Suppose a SM has 16384 registers and can accommodate 1536 threads.
- With 1536 threads each thread can use only 10 registers.
- If each thread uses 11 registers the number of threads able to be executed concurrently is reduced. This is done on a block basis, so if each block has 512 threads the number of threads that can simultaneously reside on an SM will be 1536-512=1024.
- Can get number of registers on each SM by calling **`cudaGetDeviceProperties()`** and examining the **`regsPerBlock`** field of the device properties variable.

Shared Memory Usage Example

- Suppose a SM has 16384 bytes of shared memory, and can accommodate up to 8 blocks.
- This gives 2Kb of shared memory for each block if 8 blocks reside in a SM.
- However, if each block uses 5Kb of shared memory then only 3 blocks can be assigned to each SM.
- Can get shared memory on each SM by calling **`cudaGetDeviceProperties()`** and examining the **`sharedMemPerBlock`** field of the device properties variable.

Shared Memory In Tiled Matrix Multiply

- For 16x16 blocks and 4-byte floats, Mds and Nds each require 16*16*4 = 1Kb of shared memory, which gives a total of 2Kb.
- This gives 2Kb of shared memory for each block if 8 blocks reside in a SM.
- So if the shared memory on an SM is 16Kb then 8 blocks can reside in a SM simultaneously.
- However, if the maximum number of threads on an SM is 1536 then only 1536/(16*16)=6 blocks are allowed on each SM.
- Thus 4Kb of the shared memory will not be used.

CUDA Occupancy Calculator

- You can download this from:

  https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html

- There are a number of other performance analysis tools:

  https://developer.nvidia.com/performance-analysis-tools

- These can help optimize CUDA code.