

# COMS4040A & COMS7045A: High Performance Computing & Scientific Data Management Introduction to CUDA C (Part I)

Hairong Wang

School of Computer Science,  
University of the Witwatersrand, Johannesburg

Semester one 2025

# Contents

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View
- 5 CUDA
  - C for CUDA
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compilation of CUDA C Program
- 6 References

# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View
- 5 CUDA
  - C for CUDA
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compilation of CUDA C Program
- 6 References

# Objectives

- Learn how to program massively parallel processors and achieve
  - High performance
  - Functionality
  - Scalability
- Acquire knowledge on
  - Principles of parallel programming
  - Processor architecture features and constraints
  - Programming API, tools and techniques

# Overview

- Hardware view
- Software view
- CUDA programming

# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View
- 5 CUDA
  - C for CUDA
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compilation of CUDA C Program
- 6 References

# Why Massively Parallel Processing?

- Computers no longer get faster, just wider
- You will always have more data than cores – build the computation around the data

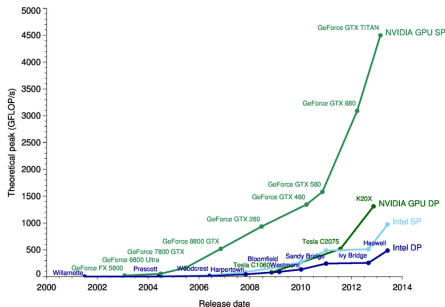


Figure: Comparison of peak performance of CPUs and GPUs

# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View**
- 4 software View
- 5 CUDA
  - C for CUDA
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compilation of CUDA C Program
- 6 References



At the top-level, a PCIe graphics card with a many-core GPU and high-speed graphics device memory sits inside a standard PC/server with one or two multicore CPUs.

- CPU
  - Generic multicore chip
    - Handful of processors each supporting 1 hardware thread
    - On-chip memory near processors (cache, RAM, or both)
    - Shared global memory space (external DRAM)

- GPU
  - Generic manycore chip
    - Many processors each supporting many hardware threads
    - On-chip memory near processors (cache, RAM, or both)
    - Shared global memory space (external DRAM)

# CPUs vs GPUs

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not
- CPU: Latency oriented design. Minimize latency experienced by 1 thread
  - Big on-chip caches (Avoid long latency of memory)
  - Sophisticated control logic (branch prediction, data prefetch etc.)
  - Powerful ALU (reduced operation latency)

- GPU: Throughput oriented design. Maximize throughput of all threads
  - Small caches
  - Energy efficient ALUs, long latency but heavily pipelined.
  - The number of threads in flight limited by resources — lots of resources (registers, bandwidth, etc.)
  - Multithreading can hide latency — skip the big caches. Requires massive number of threads to tolerate latencies
  - Share control logic across many threads (simplified logic)

# Hardware View Cont.

Example – Nvidia products - for gaming, computing, visualization, and more.

Nvidia GPU card architecture has evolved rapidly over the past few decades (2000 -). See the website<sup>[1]</sup> for more detailed information.

- The newest architecture: Nvidia Blackwell architecture <sup>[2]</sup>
- A recent one: Nvidia Ada Lovelace architecture (<https://www.nvidia.com/en-us/technologies/ada-architecture/>)

**CUDA is mainly supported by Nvidia graphics cards.**

---

<sup>[1]</sup><https://www.nvidia.com/en-us/technologies/>

<sup>[2]</sup><https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/>

NVIDIA Ada Lovelace architecture (white paper: <https://www.nvidia.com/en-us/geforce/ada-lovelace-architecture/>)

Lots of active threads is the key to high performance:

- No **context switching**; each thread has its own registers, which limits the number of active threads
- Threads on each SM execute in groups of 32 called **warps** – execution alternates between **active** warps, with warps becoming temporarily **inactive** when waiting for data

- For each thread, one operation completes before the next starts avoids the complexity of pipeline overlaps which can limit the performance of modern processors
- Memory access from device memory has a delay of 200-400 cycles; with 40 active warps this is equivalent to 5-10 operations, so hopefully there is enough computation to hide the latency



# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View**
- 5 CUDA
  - C for CUDA
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compilation of CUDA C Program
- 6 References

# Software View — Heterogeneous Parallel Computing

At the top level, we have a master process which runs on the CPU and performs the following steps:

- 1 Initializes card
- 2 Allocates memory in host and on device
- 3 Copies data from host to device memory
- 4 Launches multiple instances of execution **kernel** on device
- 5 Copies data from device memory to host
- 6 Repeats 3-5 as needed
- 7 De-allocates all memory and terminates

At a lower level, within the GPU:

- Each instance of the execution kernel executes on a SM
- If the number of instances exceeds the number of SMs, then more than one will run at a time on each SM if there are enough registers and shared memory, and the others will wait in a queue and execute later
- All threads within one instance can access local shared memory but can't see what the other instances are doing (even if they are on the same SM)
- There are no guarantees on the order in which the instances execute

# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View
- 5 CUDA**
  - C for CUDA
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compilation of CUDA C Program
- 6 References

CUDA (Compute Unified Device Architecture) is NVIDIA's program development environment:

- Based on C/C++ with some extensions
- Scalable parallel programming model:
  - It is a good fit to GPU architecture, as well as multicore CPUs.
  - Scales to 100s of cores and 1000s of parallel threads

- Minimal extensions to familiar C/C++ environment
- Heterogeneous serial-parallel computing
  - CPUs for sequential part
  - GPUs for parallel parts
- FORTRAN support provided by compiler from PGI (now owned by NVIDIA)
- Lots of example code and good documentation – fairly short learning curve for those with experience of OpenMP and MPI programming
- large user community on NVIDIA forums

# CUDA Components

Installing CUDA on a system, there are 3 components:

**Driver:** ● low-level software that controls the graphics card

**Toolkit** ● `nvcc` CUDA compiler  
● Profiling and debugging tools  
● Libraries

**SDK** ● Lots of demonstration examples  
● Some error-checking utilities  
● Not officially supported by NVIDIA

- Nvidia CUDA installation guide for Linux  
(<https://docs.nvidia.com/cuda/cuda-installation-guide-linux/#meta-packages>)

- Nvidia CUDA installation guide for Windows  
(<https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html>)

# CUDA Programming

CUDA program has two pieces:

- Host code on the CPU which interfaces to the GPU
- Kernel code which runs on the GPU (or device)

At the host level, **runtime** APIs (Application Programming Interfaces) are used for running the kernel code.



At the host code level, there are library routines for:

- Memory allocation on device, i.e., graphics card
- Data transfer between **host memory** and **device memory**
- Synchronization between CPU and GPU
- Error checking
- Timing

# CUDA Programming cont.

- A special syntax for launching multiple instances of the kernel process on the GPU. In its simplest form it looks like:

```
kernel_routine<<<nBlocks, nThreads>>>(args);
```

- Defines the dimension of the grid and blocks that will be used to execute the function on the device
- `nBlocks` specifies the dimension and size of the grid
- `nThreads` specifies the dimension and size of each block
- `args` is a limited number of arguments, usually pointers to arrays in graphics memory, and some constants which get copied by value

# CUDA Programming cont.

At the lower level, when one instance of the kernel is started on a SM it is executed by a number of threads, each of which knows about:

- Some variables passed as arguments
- Pointers to arrays in device memory (also arguments)
- Global constants in device memory
- Shared memory and private registers/local variables
- Some special or built-in variables:
  - `gridDim` size (dimensions) of grid of blocks
  - `blockDim` size (dimensions) of each block
  - `blockIdx` index (or 2D/3D indices) of block
  - `threadIdx` index (or 2D/3D indices) of thread

# CUDA Programming cont.

1D grid with 4 blocks, each with 64 threads:

- `gridDim = 4`
- `blockDim = 64`
- `blockIdx` ranges from 0 to 3
- `threadIdx` ranges from 0 to 63

# Kernel execution

- Each thread is executed by a core;
- Each block is executed by a SM and does not migrate;
- Several concurrent blocks can reside on one SM depending on the resources required by the blocks and the available SM resources;
- Threads may need to cooperate
- Share results or cooperate to produce a single result
- Synchronize with each other.

# Thread blocks

- Thread blocks allow scalability
- They can execute in any order

The kernel code looks fairly normal once you get used to:

- Code is written from the point of view of a single thread
  - Different to OpenMP multithreading
  - all local variables are private to that thread
- Need to think about where each variable lives
- Any operation involving data in the device memory forces its transfer to/from registers in the GPU
- Often better to copy the value into a local register variable

# Example

```
1 void saxpy_serial(int n, float a, float *x, float *y){
2     for(int i=0; i<n; i++)
3         y[i]=a*x[i]+y[i];
4 }
5 .....
6 //call the serial function
7 saxpy_serial(n,2.0,x,y);
```

```
1 __global__ void saxpy_parallel(int n, float a, float *x, float *y){
2     int i=threadIdx.x + blockIdx.x*blockDim.x;
3     if(i<n)
4         y[i]=a*x[i]+y[i];
5 }
6 .....
7 //call the kernel function with 256 threads per block
8 int nblocks=(n+255)/256;
9 saxpy_parallel<<<nblocks, 256>>>(n,2.0,x,y);
```



# Host Code

```
1 int main(int argc, char **argv)
2 {
3     float *h_x, *d_x;
4     int nblocks=2, nthreads=8, nsize=2*8;
5     h_x = (float *)malloc(nsize*sizeof(float));
6     /*memory allocation on the device*/
7     cudaMalloc((void **)&d_x,nsize*sizeof(float));
8     /*launch the kernel*/
9     my_kernel<<<nblocks,nthreads>>>(d_x);
10    /*transfer the data from device to host*/
11    cudaMemcpy(h_x,d_x,nsize*sizeof(float), cudaMemcpyDeviceToHost);
12    for (int n=0; n<nsize; n++)
13        printf(" n, x = %d %f \n",n,h_x[n]);
14    /*free memory on device*/
15    cudaFree(d_x);
16    free(h_x);
17 }
```

```
1 __global__ void my_kernel(float *x)
2 {
3     int tid = threadIdx.x + blockDim.x*blockIdx.x;
4     x[tid] = (float) threadIdx.x;
5 }
```

- `__global__` identifier says it is a kernel function
- Each thread sets one element of array `x`
- Within each block of threads, `threadIdx.x` ranges from 0 to `blockDim.x-1`, so each thread has a unique value for `tid`.

Suppose we have 1000 blocks, and each one has 128 threads — how does it get executed?

- On Maxwell hardware, would probably get 8-12 blocks running at the same time on each SM, and each block has 4 warps — 32-48 warps running on each SM
- Each clock tick, SM warp scheduler decides which warps to execute next, choosing from those not waiting for
  - data coming from device memory (memory latency)
  - completion of earlier instructions (pipeline delay)

# CUDA Programming

In this simple case, we had a 1D grid of blocks, and a 1D set of threads within each block. If we want to use a 2D set of threads, then

- `blockDim.x, blockDim.y` give the dimensions
- `threadIdx.x, threadIdx.y` give the thread indices
- To launch the kernel we would use something like

```
int nblocks = 2;  
dim3 nthreads(16,4);  
my_new_kernel<<<nblocks,nthreads>>>(d_x);
```

where `dim3` is a special CUDA datatype with 3 components  
`.x, .y, .z` each default to 1 if not specified.

# CUDA Programming Model Contd.

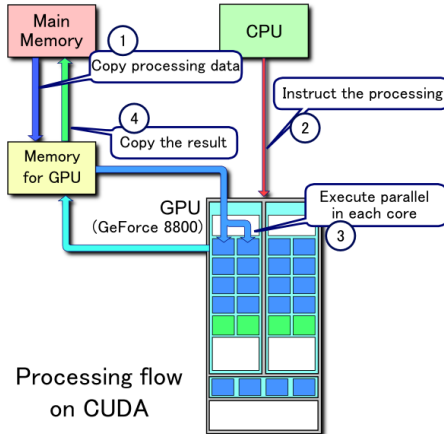


Figure: CUDA program execution flow

# Key Architectural Ideas of GPU

- SIMT (Single Instruction Multiple Thread) execution:
  - Threads run in groups of 32 called **warps**.
  - Threads in a warp share instruction unit (IU).
  - Hardware automatically handles divergence.
- Hardware multithreading
  - Hardware resource allocation & thread scheduling
  - Hardware relies on threads to hide latency
- Threads have all resources needed to run
  - Any warp not waiting for something can run
  - Context switching is (basically) free

# Key Parallel Abstractions in CUDA

- Hierarchy of concurrent threads
  - Parallel kernels composed of many threads, all threads execute the same sequential program
  - Threads are grouped into thread blocks, threads in the same block can cooperate
  - Threads/blocks have unique IDs
- Lightweight synchronization primitives
- Shared memory model for cooperating threads

# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View
- 5 CUDA**
  - **C for CUDA**
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compilation of CUDA C Program
- 6 References



- Philosophy: provide minimal set of extensions necessary to expose power
- Function qualifiers:
  - `__global__ void my_kernel() {}`
  - `__device__ float my_device_func() {}`

	Executed on the	Only callable from the
<code>__device__</code>	device	device
<code>__global__</code>	device	host
<code>__host__</code>	host	host

- Variable qualifiers:

- `__constant__ float my_constant_array[32];`
- `__shared__ float my_shared_array[32];`

- Built-in Vector types:

- `dim3`: Integer vector type used to specify dimensions. It is a structure and the 1st, 2nd, 3rd components are accessible through the fields `x`, `y`, `z`. Any component left unspecified is initialized to 1.

- Execution configuration: **Example**

```
dim3 grid_dim(128, 1, 1);  
dim3 block_dim(32, 1, 1);  
// Launch kernel  
my_kernel <<< grid_dim, block_dim >>> (...);  
  
dim3 grid_dim(100, 50); // 5000 thread blocks  
dim3 block_dim(4, 8, 8); // 256 threads per block  
// Launch kernel  
my_kernel <<< grid_dim, block_dim >>> (...);
```

- Built-in variables and functions valid in device code:

- `gridDim;` // dim3, grid dimension
- `blockDim;` // dim3, block dimension

The size of a block is limited.

- `blockIdx;` // dim3, block index
- `threadIdx;` // dim3, thread index

- `void __syncthreads();` // Thread synchronization

**Note:** The main functionality of `blockIdx` and `threadIdx` variables is to provide threads with a means to distinguish among themselves when executing the same kernel. One common usage for `threadIdx` and `blockIdx` is to determine the area of data that a thread is to work on.

CUDA compute capability: The general specifications and features of a compute device depend on its compute capability. See the technical specifications per Compute Capability at

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#compute-capabilities>

# Example: Vector addition

```
1 // compute vector sum c = a + b
2 // each thread performs one pair-wise addition
3 // device code
4 __global__ void vector_add(float* A, float* B, float* C)
5 {
6     int i = threadIdx.x + blockDim.x * blockIdx.x;
7     C[i] = A[i] + B[i];
8 }
9
10 // host code
11 int main()
12 {
13     // initialization code
14     ...
15     // Run N/256 blocks of 256 threads each
16     vector_add<<< N/256, 256>>>(d_A, d_B, d_C);
17 }
```

# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View
- 5 CUDA**
  - C for CUDA
  - **Kernels**
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compilation of CUDA C Program
- 6 References

CUDA C extends C by allowing the programmer to define C functions, called **kernels** that

- when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.
- The kernels typically generate a large number of threads to exploit *data parallelism*.
- All threads execute the same code, may take different paths.
- Each thread has an ID, and used to select input/output data, and control decisions.



- Declaration: `__global__` qualifier.
- Execution configuration syntax: `<<<Dg, Db, ...>>>` to specify the number of CUDA threads.
  - `Dg`: specifies the dimension and size of the grid, such that `Dg.x * Dg.y * Dg.z` equals the number of blocks being launched.
  - `Db`: specifies the dimension and size of each block, such that `Db.x * Db.y * Db.z` equals the number of threads being launched.

# Kernels Contd.

- A CUDA kernel is executed by a grid (array) of threads.
  - All threads in a grid run the same kernel code (SPMD)
  - Each thread has an index that it uses to compute memory addresses and make control decisions.

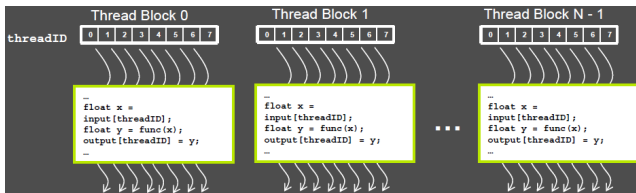


Figure: Grid of thread blocks

# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View
- 5 CUDA**
  - C for CUDA
  - Kernels
  - **Thread Hierarchy**
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compilation of CUDA C Program
- 6 References

# Thread Hierarchy

- `threadIdx`: 3-component vector. Threads can be identified using a 1-, 2-, or 3-dimensional **thread index** forming 1-, 2-, or 3-dimensional thread block.
  - 1-D:  $(x)$ —thread ID is  $x$ .
  - 2-D:  $(x, y)$ —thread ID is  $x + yD_x$ .
  - 3-D:  $(x, y, z)$ —thread ID is  $x + yD_x + zD_xD_y$ .
- There is a limit to the number of threads per block (current 1024).

# Thread Hierarchy Contd.

- Thread blocks are organized into 1-, 2-, or 3-dimensional **grid**.
- The number of blocks is usually dictated by the size of data.
- `blockIdx`: Blocks can be identified using a 1-, 2-, or 3-dimensional **block index** accessible within the kernel through `blockIdx` variable.
- The dimension of the thread block is accessible within the kernel through `blockDim` variable.
- Thread blocks are required to execute independently in any order.
- `__syncthreads()` : Acts as a barrier.

# Thread Hierarchy Contd.

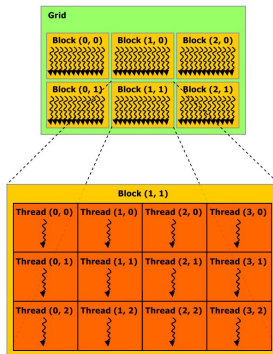


Figure: Organization of threads within a block

# Matrix Addition Example 1

```
1 // Kernel definition
2 __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
3 {
4     int i = threadIdx.x;
5     int j = threadIdx.y;
6     C[i][j] = A[i][j] + B[i][j];
7 }
8 int main()
9 {
10     ...
11     // Kernel invocation with one block of N * N * 1 threads
12     int numBlocks = 1;
13     dim3 threadsPerBlock(N, N);
14     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
15     ...
16 }
```

# Matrix Addition Example 2

```
1 // Kernel definition
2 __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
3 {
4     int i = blockIdx.x * blockDim.x + threadIdx.x;
5     int j = blockIdx.y * blockDim.y + threadIdx.y;
6     if (i < N && j < N)
7         C[i][j] = A[i][j] + B[i][j];
8 }
9 int main()
10 {
11     ...
12     // Kernel invocation
13     dim3 threadsPerBlock(16, 16);
14     dim3 numBlocks((N+threadsPerBlock.x-1) / threadsPerBlock.x, (N+
        threadsPerBlock.y -1) / threadsPerBlock.y);
15     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
16     ...
17 }
```



# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View
- 5 CUDA**
  - C for CUDA
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy**
  - Heterogeneous Programming
  - Compilation of CUDA C Program
- 6 References

# Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution (see Figure 5).

- Local memory
- Shared memory
- Global memory
- Two additional read-only memory: *constant* and *texture* memory space.

Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

# Memory Hierarchy Contd.

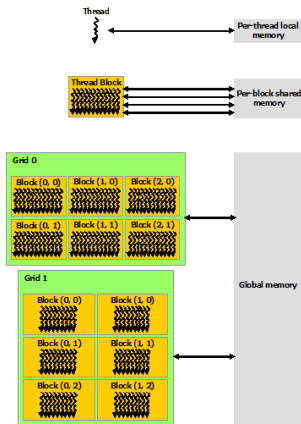


Figure: Memory hierarchy

# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View
- 5 CUDA**
  - C for CUDA
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming**
  - Compilation of CUDA C Program
- 6 References

# Heterogeneous Programming

the CUDA programming model assumes

- The CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program.
- Both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively.
- CUDA execution model:
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code

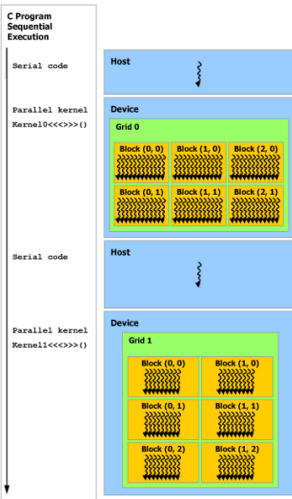


Figure: Heterogeneous programming

# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View
- 5 CUDA**
  - C for CUDA
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - **Compilation of CUDA C Program**
- 6 References

- CUDA instruction set architecture — PTX (an assembly form)
- nvcc's basic workflow
  - separating device code from host code
  - compiling the device code into PTX code and/or binary form (cubin object)
  - modifying the host code by replacing the <<< . . . >>> syntax by the necessary CUDA C runtime function calls to load and launch each compiled kernel from the PTX code and/or cubin object.



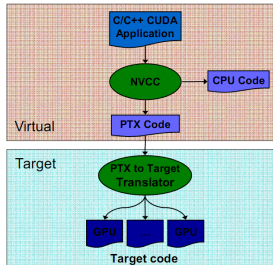


Figure: NVCC workflow

# Device Memory

In CUDA, the host and devices have separate memory spaces.

Device code can

- R/W per-thread registers (on-chip)
- R/W per-thread local memory
- R/W per-block shared memory (on-chip)
- 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.

# Device Memory Contd.

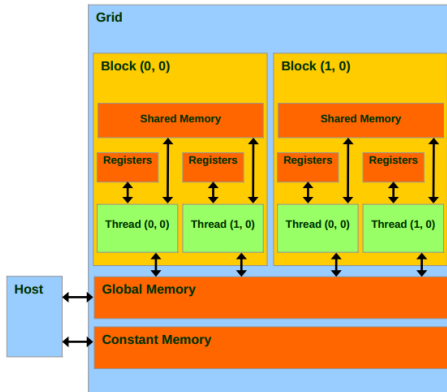


Figure: CUDA device memory model

# Device Memory Contd.

In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer pertinent data from the host memory to the allocated device memory. Similarly, after device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed.

- Linear memory is

- allocated using `cudaMalloc()`

Two parameters:

- Address of a pointer (type `void**`) to the allocated object;
- Size of allocated object in terms of bytes.

- freed using `cudaFree()`

One parameter: Pointer to freed object.

```
float *Md;  
int size = Width * Width * sizeof(float);  
cudaMalloc((void**)&Md, size);  
...  
cudaFree(Md);
```

# Device Memory Contd.

- data transfer between host memory and device memory are done using `cudaMemcpy()`.
  - Pointer to destination
  - Pointer to source
  - Number of bytes transferred
  - Type of transfer: Host to device, device to host.

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);  
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

# Device Memory Contd.

- Shared memory is
  - allocated using the `__shared__`.
  - much faster than global memory

CUDA variable type qualifiers			
Variable declaration	Memory	Scope	Lifetime
Automatic scalar variables	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
<code>__shared__</code>	Shared	Block	Kernel
<code>__device__</code>	Global	Grid	Application
<code>__constant__</code>	Constant	Grid	Application

# Timing Using CUDA Events

```
1  cudaEvent_t start, stop;
2  cudaEventCreate(&start);
3  cudaEventCreate(&stop);
4  cudaEventRecord(start, 0);

6  // do some work on GPU

8  cudaEventRecord(stop, 0);
9  cudaEventSynchronize( stop );
10 float elapsedTime;
11 cudaEventElapsedTime(&elapsedTime, start, stop);
12 printf( "Time elapsed: %3.6f ms\n", elapsedTime );
13 cudaEventDestroy( start );
14 cudaEventDestroy( stop );
```

# Matrix Multiplication Example

Matrix multiplication,  $C = A * B$

- Each thread calculates one element of C
- Each row of A is loaded B.width times from global memory
- Each column of B is loaded A.height times from global memory



# Matrix Multiplication Example Contd.

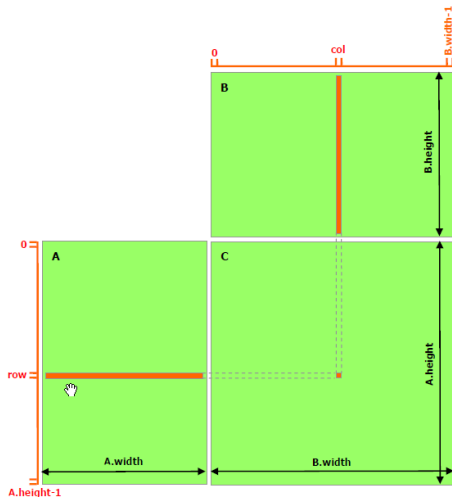


Figure: Matrix multiplication example

# Outline

- 1 Objectives
- 2 Why Massively Parallel Processing?
- 3 Hardware View
- 4 software View
- 5 CUDA
  - C for CUDA
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compilation of CUDA C Program
- 6 References

# Further Reading

- Chapters 2 – 3, Programming Massively Parallel Processors: A Hands-on Approach, third edition, by David B. Kirk and Wen-mei W. Hwu. Morgan Kaufmann Publishers Inc.
- Chapters 2 – 3, CUDA by Example, an Introduction to general-Purpose GPU Programming, by Jason Sanders and Edward Kandrot. Addison-Wesley, 2011.
- Chapter 2, Appendix B.1 – B.4, B.22 (Execution configuration) C Language Extensions, CUDA C Programming Guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/> – visited Mar 23, 2025.