

SC3260 / SC5260

GPU and GPGPU Programming

Lecture by: Ana Gainaru

Table of contents

General Purpose GPUs

- ▶ Programming model
- ▶ GPU architecture
- ▶ Performance difference between CPU and GPU
- ▶ Submitting jobs on ACCRE
- ▶ Post-running analysis (Homework 2)

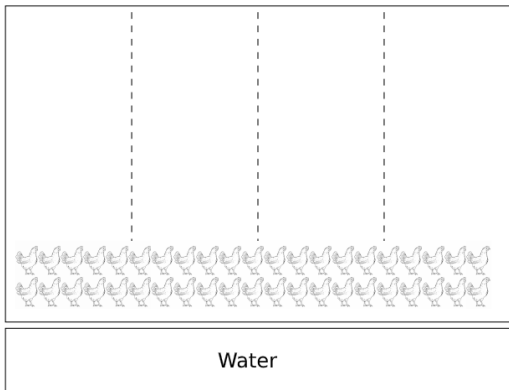
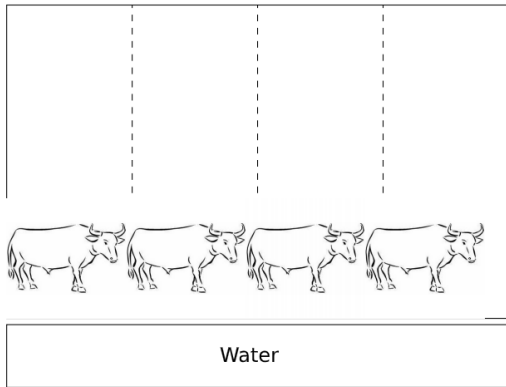


VANDERBILT
UNIVERSITY

General Purpose GPUs

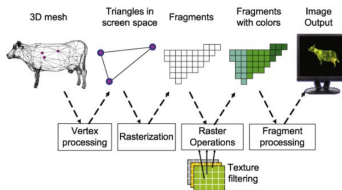
"If you were plowing a field, which would you rather use: two strong oxes or 1024 chickens?"

Seymour Cray



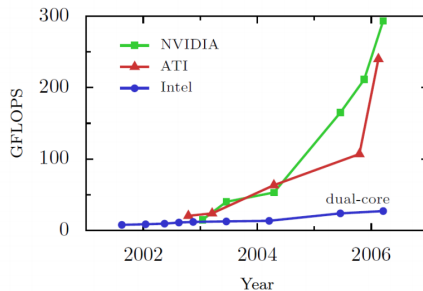
VANDERBILT
UNIVERSITY

- ▶ The CPU has always been slow for Graphics Processing
 - ▶ Visualization
 - ▶ Games
- ▶ Graphics processing is inherently parallel and there is a lot of parallelism $O(\text{pixels})$
- ▶ GPUs were built to do graphics processing only
- ▶ Initially, hardwired logic replicated to provide parallelism
 - ▶ Little to no programmability



Source: Kaufman et al. (2009)

GPU performance



Source: Owens et al., A Survey of General-Purpose Computation on Graphics Hardware



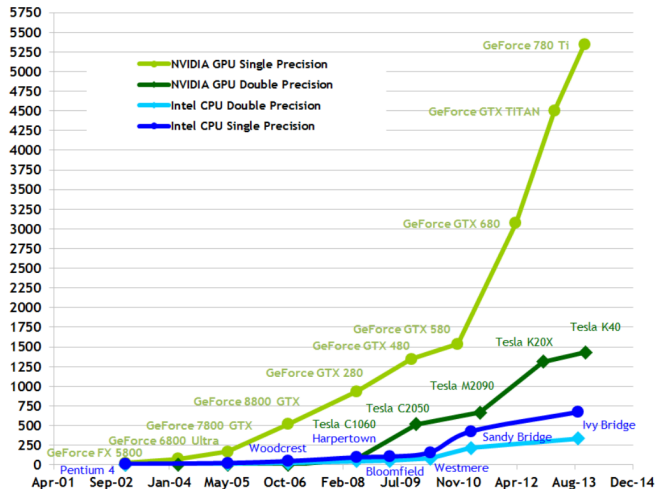
- ▶ Like CPUs, GPUs benefited from Moore's Law
- ▶ Evolved from fixed-function hardwired logic to flexible, programmable ALUs
- ▶ Around 2004, GPUs were programmable "enough" to do some non-graphics computations
 - ▶ Severely limited by graphics programming model (shader programming)
- ▶ In 2006, GPUs became "fully" programmable
 - ▶ NVIDIA releases "CUDA" language to write non-graphics programs that will run on GPUs



VANDERBILT
UNIVERSITY

GPU Performance

Theoretical GFLOP/s



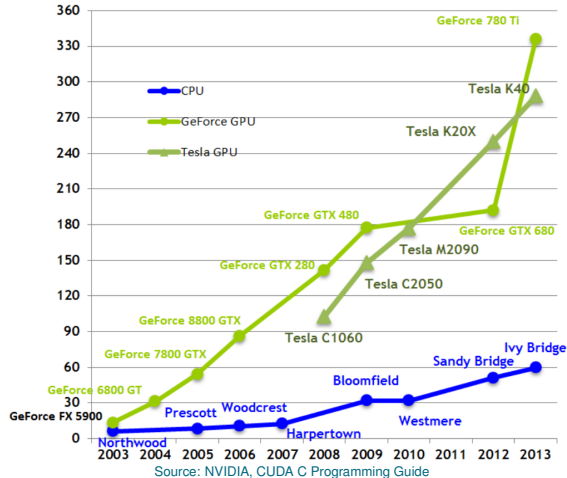
Source: NVIDIA, CUDA C Programming Guide



VANDERBILT
UNIVERSITY

GPU Performance

Theoretical GB/s



VANDERBILT
UNIVERSITY

- ▶ GPUs are widely deployed as accelerators
- ▶ GPUs are so successful that other CPU alternatives are dead
 - ▶ Sony/IBM Cell BE
 - ▶ Clearspeed RSX
- ▶ Kepler K40 GPUs from NVIDIA have performance of 4TFlops (peak)
 - ▶ CM-5, #1 system in 1993 was around 60 Gflops (Linpack)
 - ▶ ASCI White (#1 2001) was 4.9 Tflops (Linpack)



VANDERBILT
UNIVERSITY

Accelerator-based Systems

- ▶ CPUs have always depended on co-processors
 - ▶ I/O co-processors to handle slow I/O
 - ▶ Math co-processors to speed up computation
- ▶ **These have mostly been transparent**
 - ▶ Drop in the co-processor and everything sped up
- ▶ The GPU is not a transparent accelerator for general purpose computations
 - ▶ Only graphics code is sped up transparently
- ▶ **Code must be rewritten to target GPUs**



VANDERBILT
UNIVERSITY

General Purpose Graphic Processing Units

The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)

- ▶ However, the programming is done using threads, **NOT SIMD instructions**
- ▶ To understand this, we will look at some examples
- ▶ But, before that, let's distinguish between
 - ▶ **Programming Model (Software)**
 - vs.
 - ▶ **Execution Model (Hardware)**



Programming Model vs. Hardware Execution Model

- ▶ Programming Model refers to **how the programmer expresses the code**
 - ▶ E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), ...



VANDERBILT
UNIVERSITY

Programming Model vs. Hardware Execution Model

- ▶ Programming Model refers to **how the programmer expresses the code**
 - ▶ E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), ...
- ▶ Execution Model refers to **how the hardware executes the code underneath**
 - ▶ E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, ...



VANDERBILT
UNIVERSITY

Programming Model vs. Hardware Execution Model

- ▶ Programming Model refers to **how the programmer expresses the code**
 - ▶ E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), ...
- ▶ Execution Model refers to **how the hardware executes the code underneath**
 - ▶ E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, ...
- ▶ **Execution Model can be very different from the Programming Model**
 - ▶ E.g., von Neumann model implemented by an OoO processor
 - ▶ E.g., SPMD model implemented by a SIMD processor (a GPU)



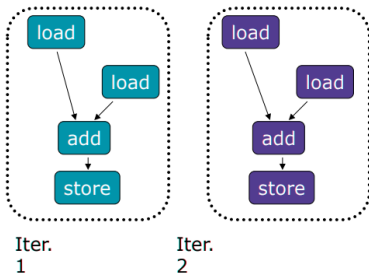
VANDERBILT
UNIVERSITY

```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```

How Can You Exploit Parallelism Here?

Three programming options to exploit instruction-level parallelism present in this sequential code

- 1 Sequential (SISD)
- 2 Data-Parallel (SIMD)
- 3 Multithreaded (MIMD/SPMD)



Source: Lecture 9: GPUs and GPGPU Programming (ETH Zürich, Fall 2017)
<https://safari.ethz.ch/architecture/fall2017>



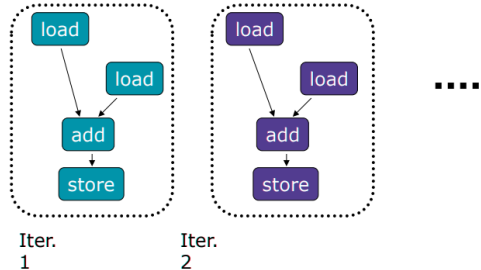
VANDERBILT
UNIVERSITY

Data Parallel (SIMD)

```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```

Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Best executed by a SIMD processor
(vector, array)



Source: Lecture 9: GPUs and GPGPU Programming (ETH Zürich, Fall 2017)
<https://safari.ethz.ch/architecture/fall2017>



VANDERBILT
UNIVERSITY

Multi-threaded Programming Model

```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```

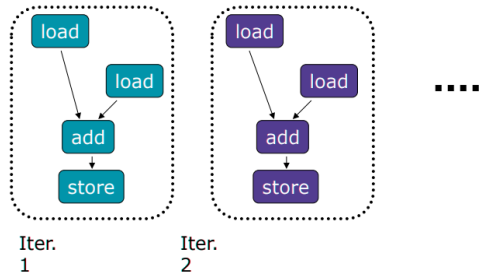
Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

This model is also called:

SPMD: Single Program Multiple Data

Can be executed on a SIMT machine

Single Instruction Multiple Thread



Source: Lecture 9: GPUs and GPGPU Programming (ETH Zürich, Fall 2017)
<https://safari.ethz.ch/architecture/fall2017>



VANDERBILT
UNIVERSITY

A GPU is a SIMD (SIMT) Machine

Except it is not programmed using SIMD instructions

- ▶ It is programmed using threads (SPMD programming model)
 - ▶ Each thread executes the same code but operates a different piece of data
 - ▶ Each thread has its own context (i.e., can be treated/restarted/ executed independently)
- ▶ A set of threads executing the same instruction are dynamically grouped into a warp (wavefront) by the hardware
 - ▶ **A warp is essentially a SIMD operation formed by hardware!**

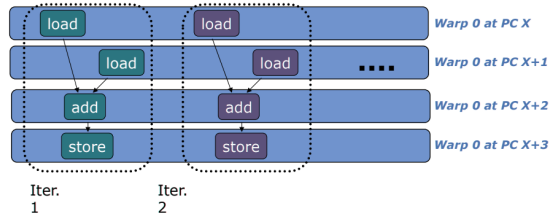


VANDERBILT
UNIVERSITY

SPMD on SIMT Machine

```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```

Warp: A set of threads that execute the same instruction (i.e., at the same PC)



Source: Lecture 9: GPUs and GPGPU Programming (ETH Zürich, Fall 2017) <https://safari.ethz.ch/architecture/fall2017>

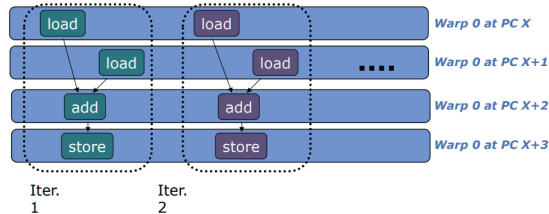


VANDERBILT
UNIVERSITY

SPMD on SIMT Machine

```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```

Warp: A set of threads that execute the same instruction (i.e., at the same PC)



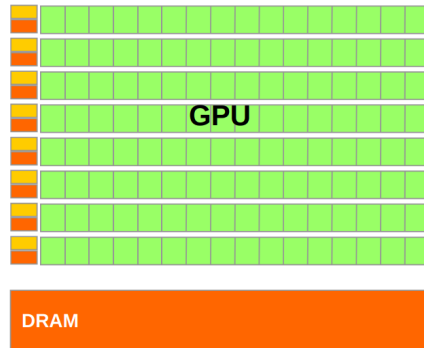
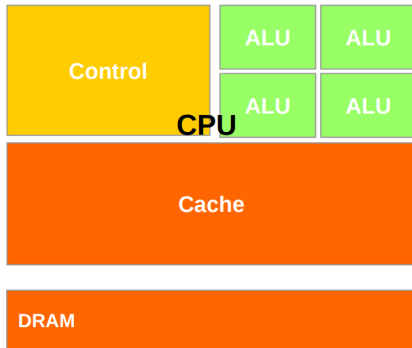
Source: Lecture 9: GPUs and GPGPU Programming (ETH Zürich, Fall 2017) <https://safari.ethz.ch/architecture/fall2017>

Graphics Processing Units SIMD not Exposed to Programmer (SIMT)



VANDERBILT
UNIVERSITY

GPU vs CPU



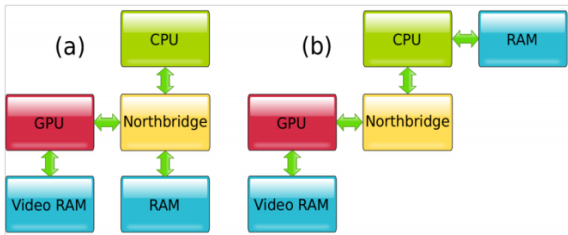
VANDERBILT
UNIVERSITY

Two Types of GPUs

Type 1: Discrete GPUs

Primary benefits: More computational power, more memory B/W

Downside: Data transfer needs to be done explicit

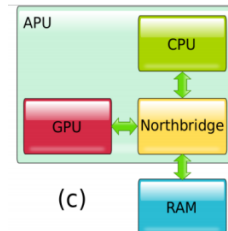
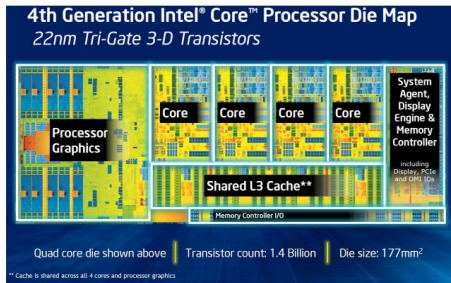


Two Types of GPUs

Type 2: Integrated GPUs

Primary distinction: Share system memory

Primary benefits: Less power (energy)



Using a GPU

- ① You must retarget code for the GPU
 - ▶ rewrite, recompile, translate, etc



VANDERBILT
UNIVERSITY

- ❶ You must retarget code for the GPU
 - ▶ rewrite, recompile, translate, etc

Is Any Application Suitable for GPU?

- ▶ Hard NO
- ▶ You will get the best performance from GPU if your application is
 - ▶ Computation intensive
 - ▶ Many independent computations
 - ▶ Many similar computations
 - ▶ Not a lot of branches

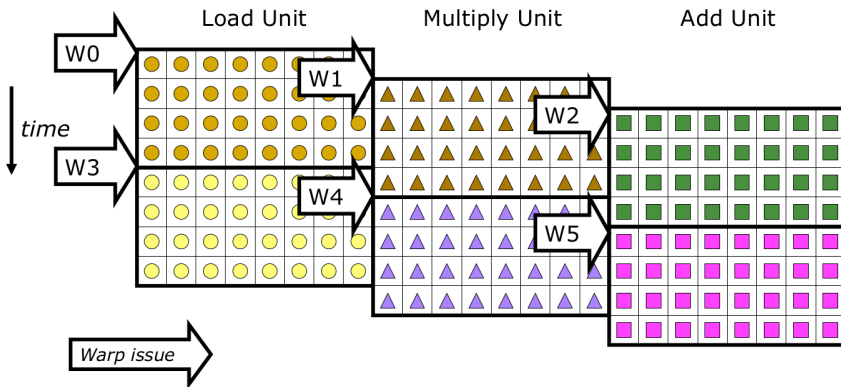


VANDERBILT
UNIVERSITY

Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

- ▶ Example machine has 32 threads per warp and 8 lanes
- ▶ Completes 24 operations/cycle while issuing 1 warp/cycle

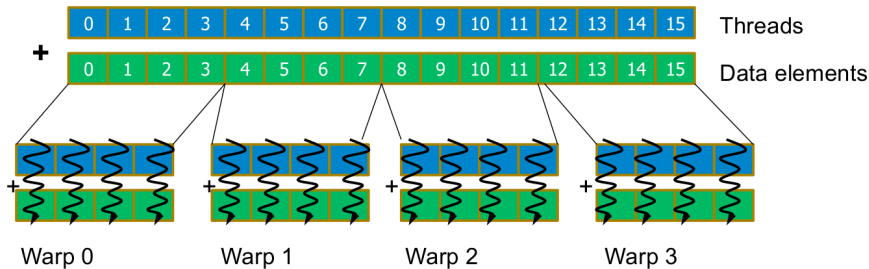


VANDERBILT
UNIVERSITY

SIMT Memory Access

Same instruction in different threads uses thread id to index and access different data elements

Let's assume $N=16$, 4 threads per warp \rightarrow 4 warps



Source: Hyesoon Kim



Warp-based SIMD vs. Traditional SIMD

Traditional SIMD contains a single thread

- ▶ Sequential instruction execution; lock-step operations in a SIMD instruction
- ▶ SW needs to know vector length

Warp-based SIMD consists of multiple threads executing in a SIMD manner (i.e., same instruction executed by all threads)

- ▶ Does not have to be lock step
- ▶ Each thread can be treated individually (i.e., placed in a different warp)
- ▶ SW does not need to know vector length
- ▶ Enables multithreading and flexible dynamic grouping of threads

Basically SPMD programming model implemented on SIMD hardware



VANDERBILT
UNIVERSITY

Single procedure/program, multiple data

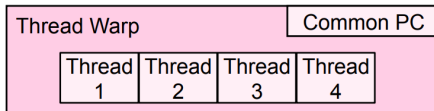
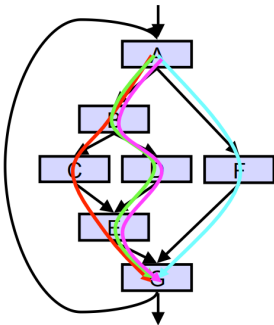
- ▶ Each processing element executes the same procedure, except on different data elements
 - ▶ Procedures can synchronize at certain points in program, e.g. barriers

- ▶ Essentially, multiple instruction streams execute the same program
 - ▶ Each program/procedure
 - ▶ works on different data
 - ▶ can execute a different control-flow path, at run-time
 - ▶ Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors) **Cilk**
 - ▶ Modern GPUs programmed in a similar way on a SIMD hardware **Cuda**



Threads Can Take Different Paths in Warp-based SIMD

- ▶ Each thread can have conditional control flow instructions
- ▶ Threads can execute different control flow paths



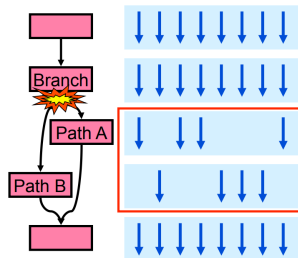
Source: Tor Aamodt



VANDERBILT
UNIVERSITY

Control Flow Problem in GPUs/SIMT

- ▶ A GPU uses a SIMD pipeline to save area on control logic.
 - ▶ Groups threads into warps
- ▶ Branch divergence occurs when threads inside warps branch to different execution paths.



Source: Tor Aamodt

Threads are independent

If we have enough threads

- ▶ We can **find individual threads that are at the same PC**
- ▶ And, **group them together into a single warp dynamically**
- ▶ This reduces "divergence" → **improves SIMD utilization**
 - ▶ SIMD utilization: fraction of SIMD lanes executing a useful operation (i.e., executing an active thread)

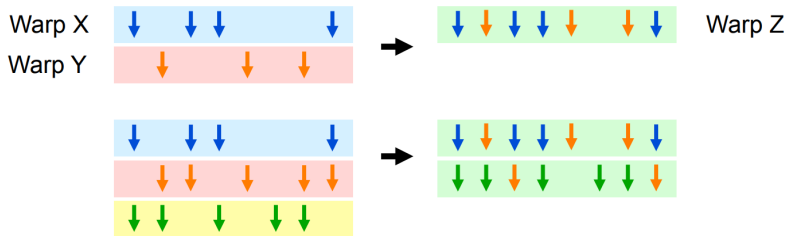


VANDERBILT
UNIVERSITY

Dynamic Warp Formation/Merging

Idea: Dynamically merge threads executing the same instruction (after branch divergence)

- ▶ Form new warps from warps that are waiting
- ▶ Enough threads branching to each path enables the creation of full new warps



- ④ You must retarget code for the GPU
 - ▶ rewrite, recompile, translate, etc
 - ▶ **Overhead of moving the context to GPU**
- ▶ Some applications are better run on CPU while others on GPU
- ▶ Main limitations
 - ▶ The parallelizable portion of the code
 - ▶ The communication overhead between CPU and GPU
 - ▶ Memory bandwidth saturation



Amdahl's Law applies for GPUs also

Execution Time After Improvement = Execution Time Unaffected + (Execution Time Affected / Amount of Improvement)

Example

Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?



VANDERBILT
UNIVERSITY

Amdahl's Law applies for GPUs also

Execution Time After Improvement = Execution Time Unaffected + (Execution Time Affected / Amount of Improvement)

Example

Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?

How about making it 5 times faster?



VANDERBILT
UNIVERSITY

Amdahl's Law applies for GPUs also

Execution Time After Improvement = Execution Time Unaffected + (Execution Time Affected / Amount of Improvement)

Example

Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?

How about making it 5 times faster?

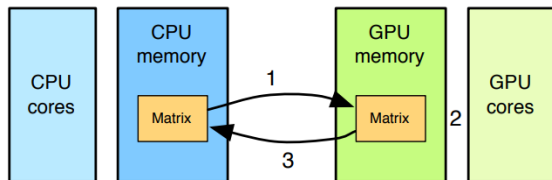
Improvement in your application speed depends on the portion that is parallelized and on the overhead



VANDERBILT
UNIVERSITY

Using a GPU

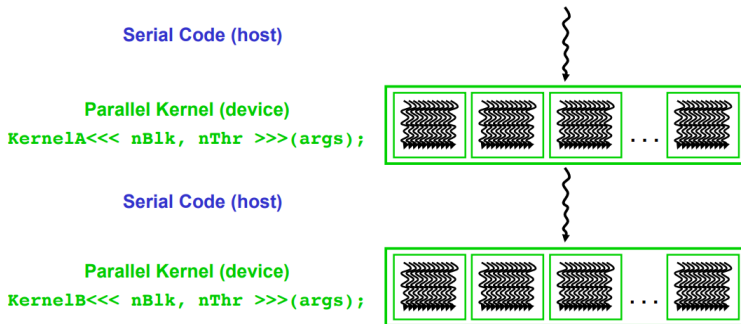
- 1 You must retarget code for the GPU
- 2 The working set must fit in GPU RAM
- 3 You must copy data to/from GPU RAM



Traditional Program Structure

CPU threads and GPU kernels

- ▶ **Sequential or modestly parallel** sections on CPU
- ▶ **Massively parallel** sections on GPU



Traditional Program Structure

CPU threads and GPU kernels

- ▶ **Sequential or modestly parallel** sections on CPU
 - ▶ **Massively parallel** sections on GPU
-
- ▶ The host (typically CPU) allocates memory, copies data, and launches kernels
 - ▶ The device (typically GPU) executes kernels
 - ▶ Grid (NDRange)
 - ▶ Block (work-group): Within a block, shared memory and synchronization
 - ▶ Thread (work-item)

Bulk synchronous programming

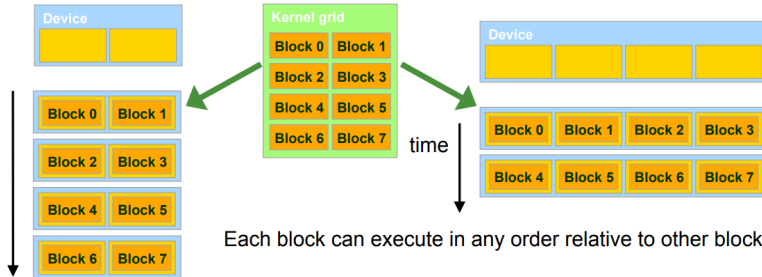
Global (coarse-grain) synchronization between kernels



VANDERBILT
UNIVERSITY

Transparent Scalability

Hardware is free to schedule thread blocks



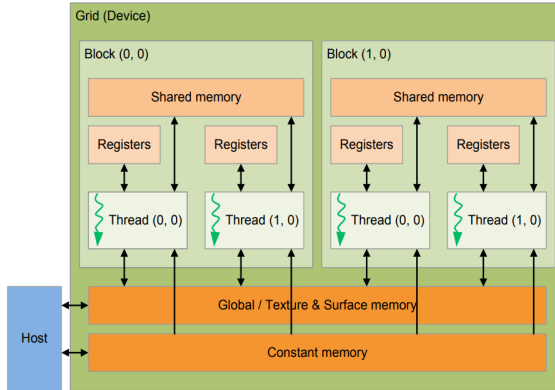
Source: Hwu & Kirk



VANDERBILT
UNIVERSITY

CUDA/OpenCL Programming Model

Memory hierarchy



Source: <https://safari.ethz.ch/architecture/fall2017>



VANDERBILT
UNIVERSITY

Traditional Program Structure

► Function prototypes

```
float serialFunction (...);  
__global__ void kernel (...);
```

► main()

- ❶ Allocate memory space on the device - `cudaMalloc(&d_in, bytes);`
- ❷ Transfer data from host to device - `cudaMemcpy(d_in, h_in, ...);`
- ❸ Execution configuration setup: `#blocks` and `#threads`
- ❹ Kernel call - `kernel«execution configuration»(args...);`
- ❺ Transfer results from device to host - `cudaMemcpy(h_out, d_out, ...);`

► Kernel - `__global__ void kernel(type args, ...)`

- Automatic variables transparently assigned to registers
- Shared memory - `__shared__`
- Intra-block synchronization - `__syncthreads();`



VANDERBILT
UNIVERSITY

CUDA Programming Language

► Memory allocation

```
cudaMalloc((void**)&d_in, #bytes);
```

► Memory copy

```
cudaMemcpy(d_in, h_in, #bytes, cudaMemcpyHostToDevice);
```

► Kernel launch

```
kernel<<< #blocks, #threads >>>(args);
```

► Memory deallocation

```
cudaFree(d_in);
```

► Explicit synchronization

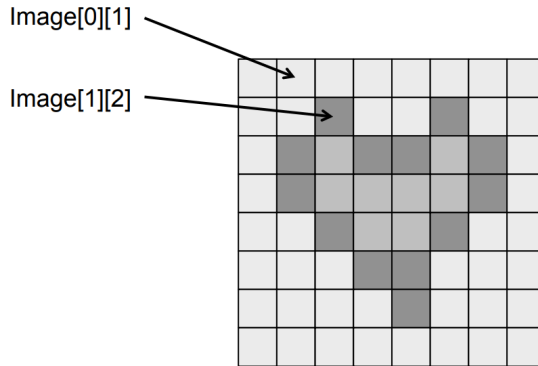
```
cudaDeviceSynchronize();
```



VANDERBILT
UNIVERSITY

Indexing and Memory Access

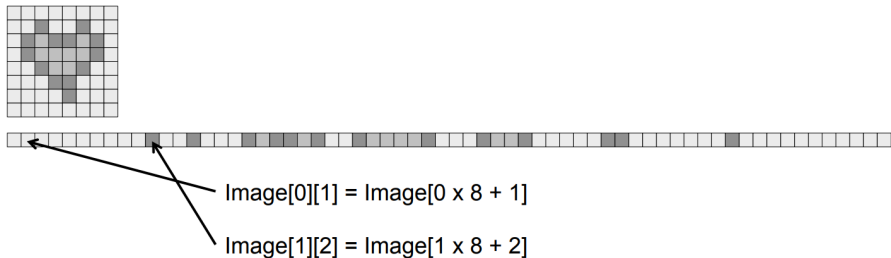
- ▶ Image layout in memory
 - ▶ height x width
 - ▶ $\text{Image}[j][i]$, where $0 \leq j < \text{height}$, and $0 \leq i < \text{width}$



VANDERBILT
UNIVERSITY

Indexing and Memory Access

- ▶ Image layout in memory
 - ▶ Row-major layout
 - ▶ $\text{Image}[j][i] = \text{Image}[j \times \text{width} + i]$

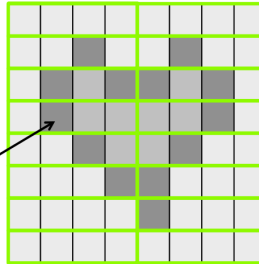


Indexing and Memory Access

- ▶ One GPU thread per pixel
- ▶ Grid of Blocks of Threads

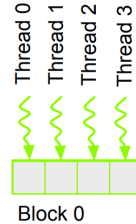
□ blockIdx.x, threadIdx.x
□ gridDim.x, blockDim.x
blockIdx.x
threadIdx.x

Block 0



$$6 * 4 + 1 = 25$$

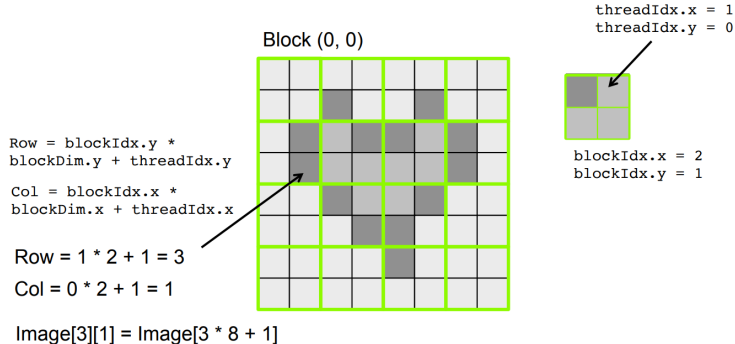
$\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$



VANDERBILT
UNIVERSITY

Indexing and Memory Access

- ▶ 2D blocks
 - ▶ gridDim.x, gridDim.y



Source: <https://safari.ethz.ch/architecture/fall2017>



VANDERBILT
UNIVERSITY

Sample GPU SIMT Code (Simplified)

CPU code

```
for (ii = 0; ii < 100000; ++ii) {  
    C[ii] = A[ii] + B[ii];  
}
```



CUDA code

```
// there are 100000 threads  
__global__ void KernelFunction(...) {  
    int tid = blockDim.x * blockIdx.x + threadIdx.x;  
    int varA = aa[tid];  
    int varB = bb[tid];  
    C[tid] = varA + varB;  
}
```



VANDERBILT
UNIVERSITY

Source: Hyesoon Kim

Sample GPU Code (Less Simplified)

CPU Program

```
void add_matrix(float *a, float *b, float *c, int N){
    int index;
    for (int i=0; i<N; ++i)
        for (int j=0; j<N; ++j){
            index = i + j * N;
            c[index] = a[index] + b[index];
        }
}

int main(){
    add_matrix(a, b, c, N);
}
```

GPU Program

```
__global__ add_matrix(float *a, float *b, float *c, int N){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j * N;
    if (i < N && j < N){
        c[index] = a[index] + b[index];
    }
}

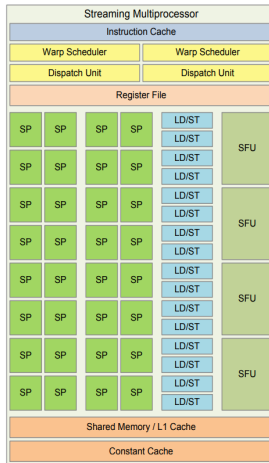
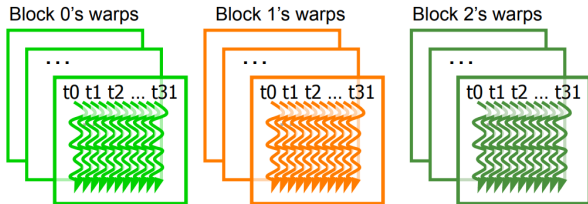
int main(){
    dim3 dimBlock(blocksize, blocksize);
    dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
    add_matrix<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```



VANDERBILT
UNIVERSITY

Putting it all together

- ▶ Blocks are divided into warps
 - ▶ SIMD unit (32 threads)
- ▶ Streaming Multiprocessors (SM)
 - ▶ Streaming Processors (SP)



VANDERBILT
UNIVERSITY

Source: <https://safari.ethz.ch/architecture/fall2017>

Brief Review of GPU Architecture

- ▶ **Streaming Multiprocessors (SM)**
 - ▶ Compute Units (CU)
- ▶ **Streaming Processors (SP)** or CUDA cores
 - ▶ Vector lanes
- ▶ **Number of SMs x SPs**
 - ▶ Tesla (2007): 30 x 8
 - ▶ Fermi (2010): 16 x 32
 - ▶ Kepler (2012): 15 x 192
 - ▶ Maxwell (2014): 24 x 128
 - ▶ Pascal (2016): 56 x 64
 - ▶ Volta (2017): 80 x 64



VANDERBILT
UNIVERSITY

Focus on performance

- ▶ Performance considerations
 - ▶ Memory access
 - ▶ Latency hiding: #blocks #threads
 - ▶ Memory coalescing
 - ▶ Data reuse: shared memory
 - ▶ SIMD utilization
 - ▶ Atomic operations

Practical examples of basic algorithms running on ACCRE



VANDERBILT
UNIVERSITY