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
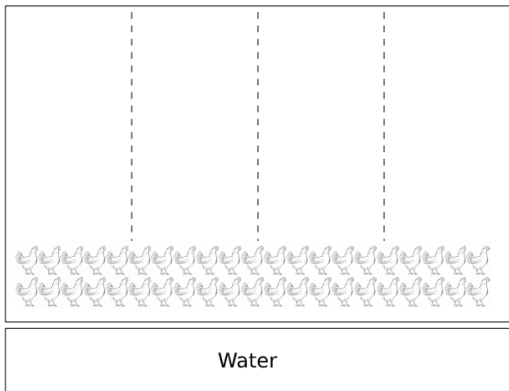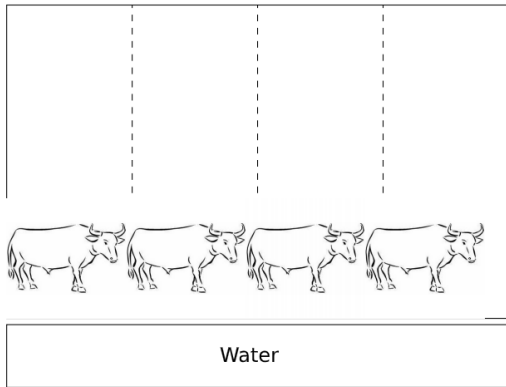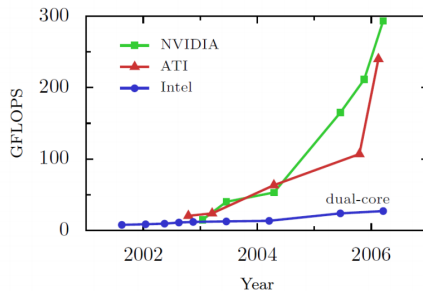- ▶ Post-running analysis (Homework 2)

General Purpose GPUs

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

Water

Water

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

- The CPU has always been slow for Graphics Processing
    - Visualization
    - Games
- Graphics processing is inherently parallel and there is a lot of parallelism **O(pixels)**
- GPUs were built to do graphics processing only
- Initially, hardwired logic replicated to provide parallelism
    - Little to no programmability
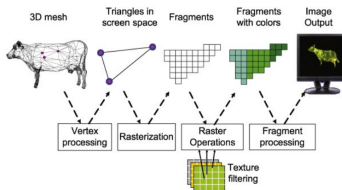
Source: Kaufman et al. (2009)

GPGPU History
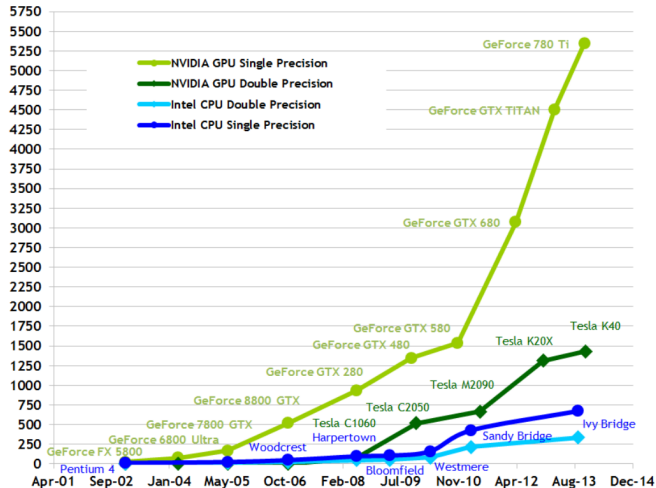
- Like CPUs, GPUs benefited from Moore's Law
- Evolved from fixed-function hardwired logic to flexible, programmable ALUs

- 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
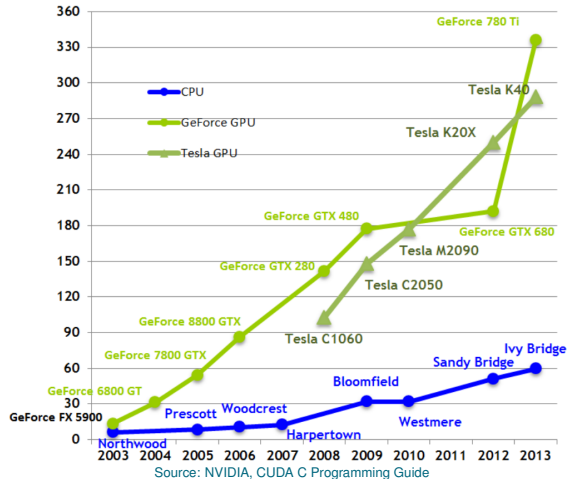
GPU Performance

Theoretical GFLOP/s

Source: NVIDIA, CUDA C Programming Guide

Theoretical GB/s

Source: NVIDIA, CUDA C Programming Guide

- 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), ...

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, ...

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**
  - q E.g., von Neumann model implemented by an OoO processor
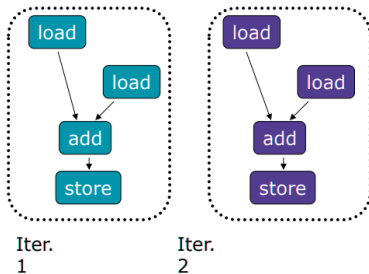 - E.g., SPMD model implemented by a SIMD processor (a GPU)

```
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)

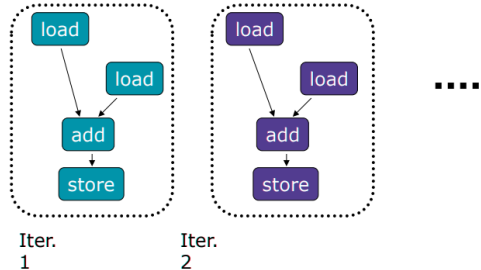Source: Lecture 9: GPUs and GPGPU Programming (ETH Zürich, Fall 2017)
https://safari.ethz.ch/architecture/fall2017

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)**

Iter. 1   Iter. 2

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

Multi-threaded Programming Model

```
for (i=0; i < N; i++)
 C[i] = A[i] + B[i];
```
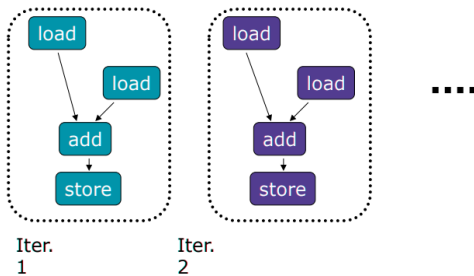
Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

This model is also called:
SPMD: Single Program Multiple Data

Can be executed on a SIMT machine
Single Instruction Multiple Thread

Iter. 1   Iter. 2   ....

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

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
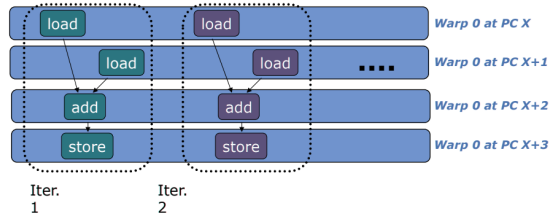 - A warp is essentially a SIMD operation formed by hardware!

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)

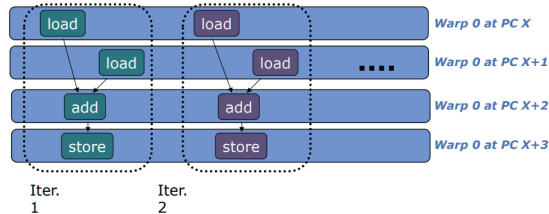Source: Lecture 9: GPUs and GPGPU Programming (ETH Zürich, Fall 2017) https://safari.ethz.ch/architecture/fall2017

```
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
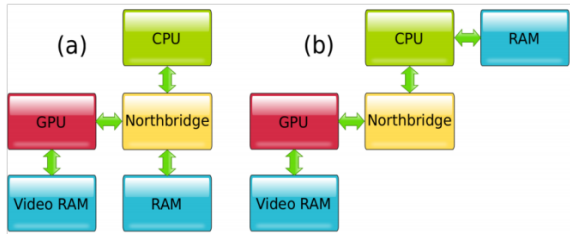
Graphics Processing Units SIMD not Exposed to Programmer (SIMT)
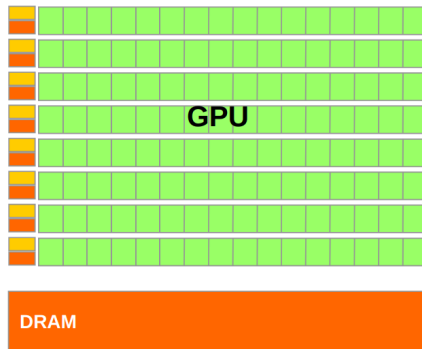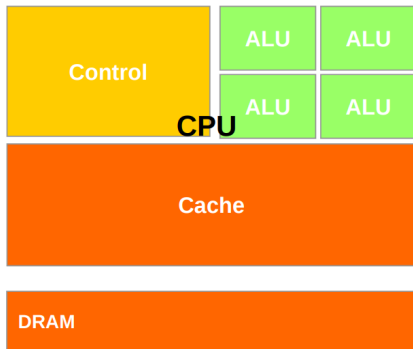
Type 1: Discrete GPUs

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

Downside: Data transfer needs to be done explicit

Type 2: Integrated GPUs
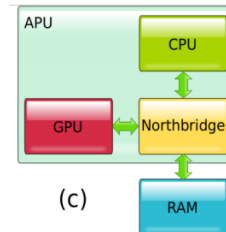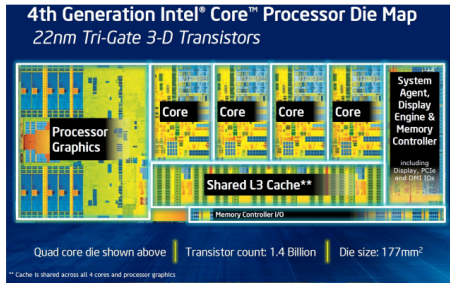
Primary distinction: Share system memory

Primary benefits: Less power (energy)

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

Using a GPU

1. 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
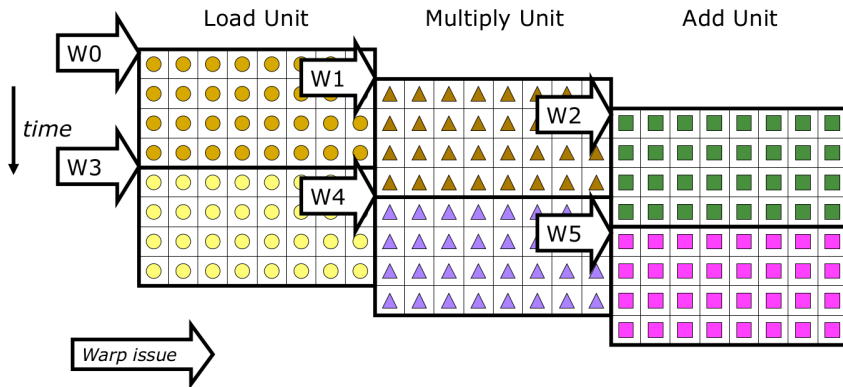 - Many similar computations
  - Not a lot of branches

Warp Instruction Level Parallelism

Can overlap execution of multiple instructions
- Example machine has 32 threads per warp and 8 lanes
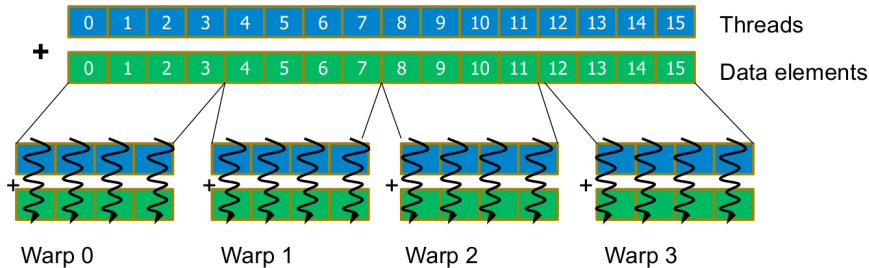- Completes 24 operations/cycle while issuing 1 warp/cycle

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

Let's assume N=16, 4 threads per warp → 4 warps

Source: Hyesoon Kim

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;
}
```

Source: Hyesoon Kim

Sample GPU Code (Less Simplified)

CPU Program

```c
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[inde];
        }
}

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

GPU Program

```c
__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[inde];
    }
}

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

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**

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**

VANDERBILT
UNIVERSITY

Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have conditional control flow instructions
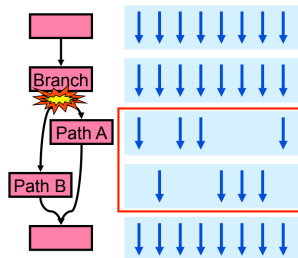- Threads can execute different control flow paths
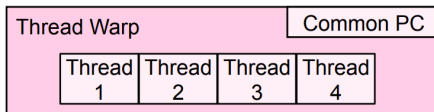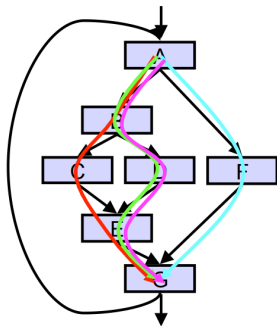
Source: Tor Aamodt

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

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" $\rightarrow$ **improves SIMD utilization**
  - SIMD utilization: fraction of SIMD lanes executing a useful operation (i.e., executing an active thread)

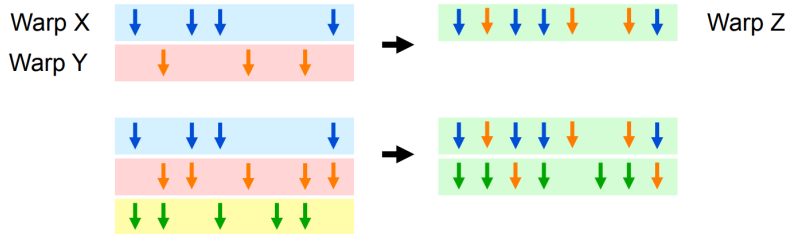# Dynamic Warp Formation/Merging

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

Using a GPU

1. 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?

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?

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

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
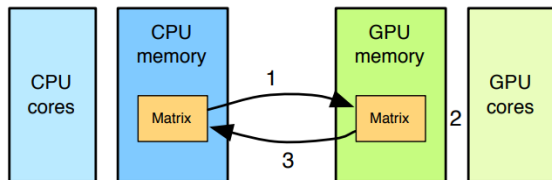
Traditional Program Structure

CPU threads and GPU kernels

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

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nThr >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nThr >>>(args);
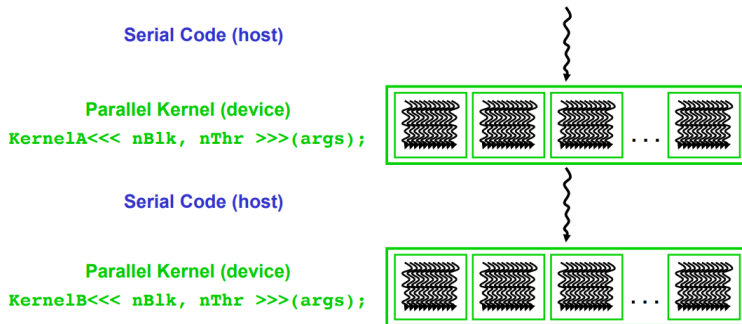
Traditional Program Structure

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

Hardware is free to schedule thread blocks
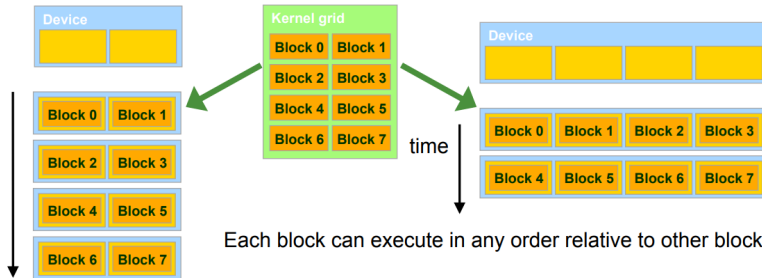
Each block can execute in any order relative to other blocks.

Source: Hwu & Kirk

CUDA/OpenCL Programming Model

Memory hierarchy

Source: https://safari.ethz.ch/architecture/fall2017
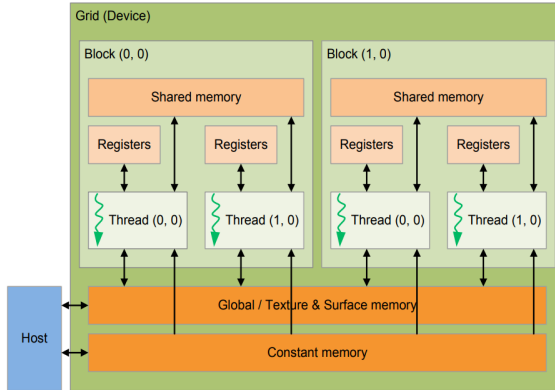
Traditional Program Structure

▶ Function prototypes

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

▶ `main()`

1. Allocate memory space on the device - `cudaMalloc(&d_in, bytes);`
2. Transfer data from host to device - `cudaMemCpy(d_in, h_in, ...);`
3. Execution configuration setup: #blocks and #threads
4. Kernel call - `kernel«execution configuration»(args...);`
5. 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();`

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();
```
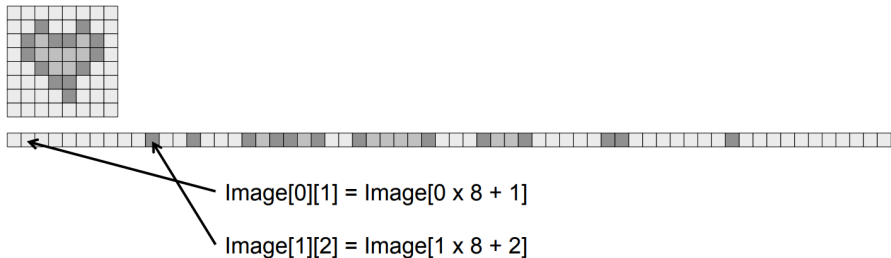
Indexing and Memory Access

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

Image[0][1]

Image[1][2]

- Image layout in memory
  - Row-major layout
  - Image[j][i] = Image[j x width + i]

Image[0][1] = Image[0 x 8 + 1]

Image[1][2] = Image[1 x 8 + 2]

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

Source: https://safari.ethz.ch/architecture/fall2017
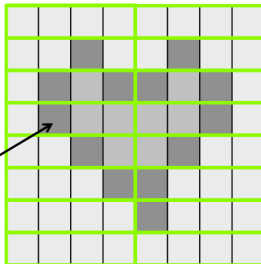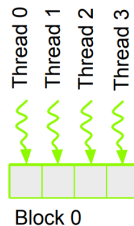
Indexing and Memory Access

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

Block (0, 0)

```
threadIdx.x = 1
threadIdx.y = 0
```

```
Row = blockIdx.y *
blockDim.y + threadIdx.y

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

```
blockIdx.x = 2
blockIdx.y = 1
```

Row = 1 * 2 + 1 = 3

Col = 0 * 2 + 1 = 1

Image[3][1] = Image[3 * 8 + 1]
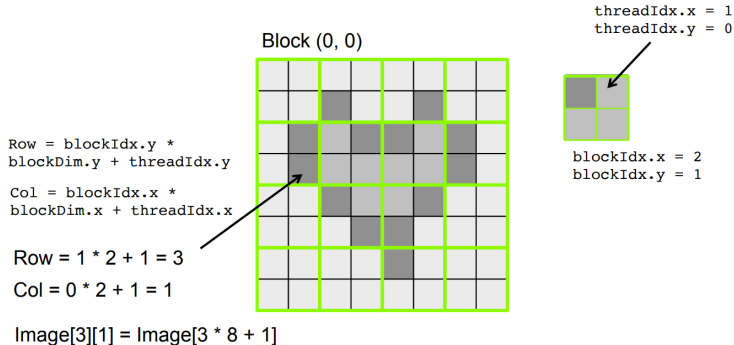
VANDERBILT
UNIVERSITY

Putting it all together

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

Block 0's warps   Block 1's warps   Block 2's warps

t0 t1 t2 ... t31   t0 t1 t2 ... t31   t0 t1 t2 ... t31

Source: https://safari.ethz.ch/architecture/fall2017
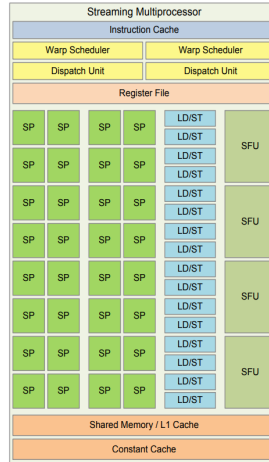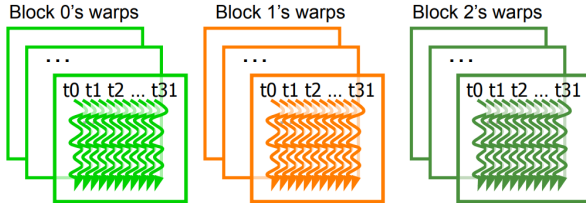
Brief Review of GPU Architecture

- **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

Performance Considerations

Main bottlenecks

- **Global memory access**
- **CPU-GPU data transfers**

- Memory access
  - Memory coalescing
  - Data reuse
  - Shared memory usage
- SIMD Utilization
- Atomic operations

- Data transfers between CPU and GPU
  - Overlap of communication and computation

Memory Coalescing

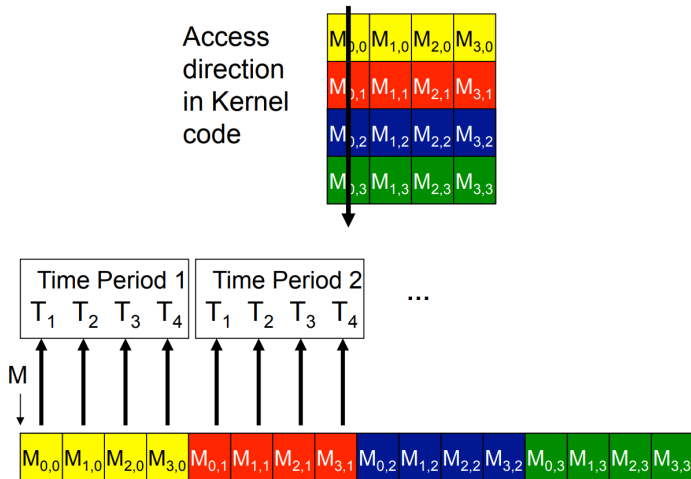When accessing global memory, peak bandwidth utilization occurs when all threads in a warp access the same cache line
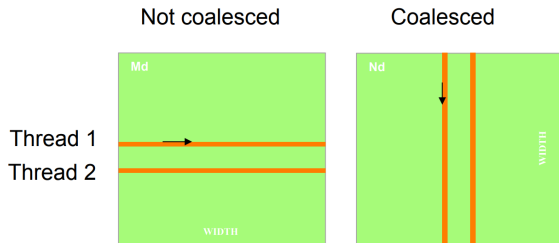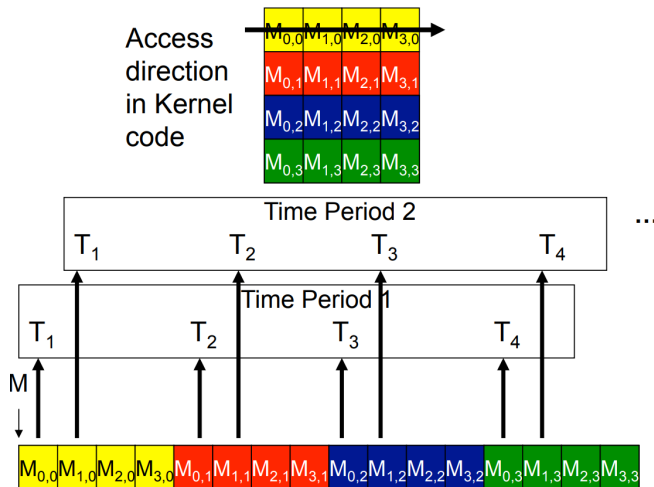

Source: Hwu & Kirk

Coalesced accesses

Source: Hwu & Kirk

Uncoalesced accesses

Source: Hwu & Kirk

Memory Coalescing

AoS vs. SoA

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

Memory Coalescing

Linear and strided accesses
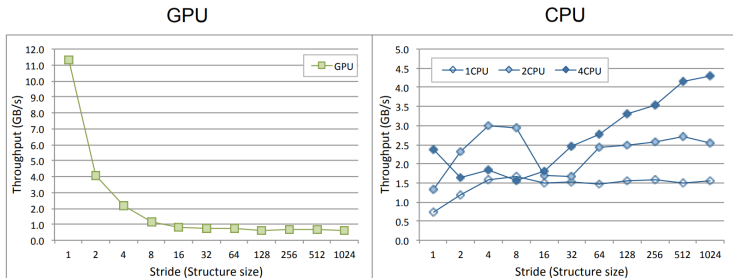
AMD Kaveri A10-7850K

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

Data Reuse
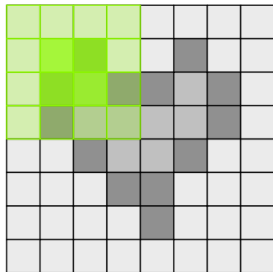
Same memory locations accessed by neighboring threads
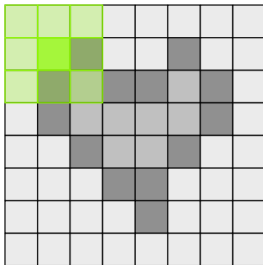

```
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
  }
}
```

Data Reuse

Shared memory tiling


```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
...
// Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
  }
}
```
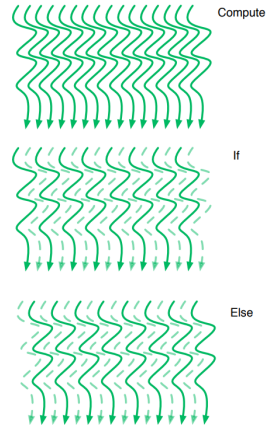
Shared Memory

Bank conflicts are only possible within a warp

- No bank conflicts between different warps

- If strided accesses are needed, some optimization techniques can help
  - Padding
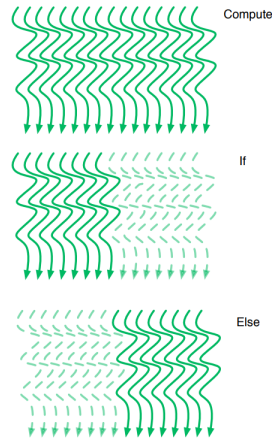 - Hash functions

Intra-warp divergence

```
Compute(threadIdx.x);

if (threadIdx.x % 2 == 0){
    Do_this(threadIdx.x);
}
else{
    Do_that(threadIdx.x);
}
```
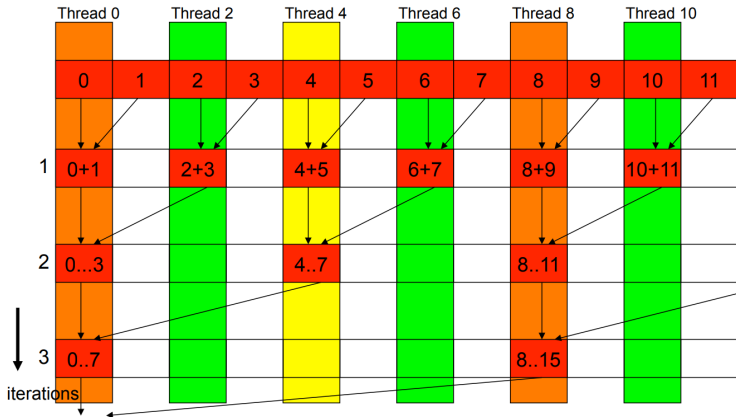
Intra-warp divergence

```
Compute(threadIdx.x);

if (threadIdx.x < 32){
    Do_this(threadIdx.x * 2);
}
else{
    Do_that((threadIdx.x % 32) * 2 + 1);
}
```


Compute

If

Else

Vector Reduction

Naive mapping

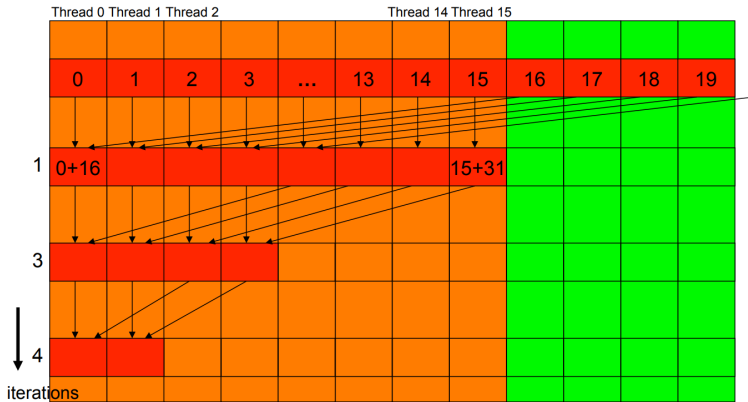Slide credit: Hwu & Kirk

Vector Reduction

Naive mapping

```
__shared__ float partialSum [];

unsigned int t = threadIdx.x;

for (int stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();

    if (t % (2*stride) == 0)
        partialSum[t] += partialSum[t + stride];
}
```

Vector Reduction

Divergence-free mapping

Slide credit: Hwu & Kirk

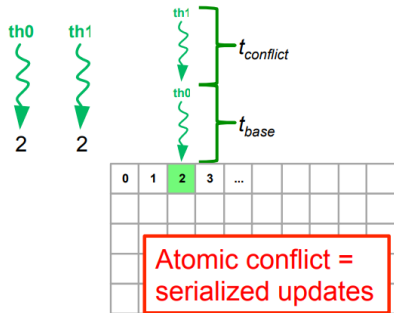# Vector Reduction

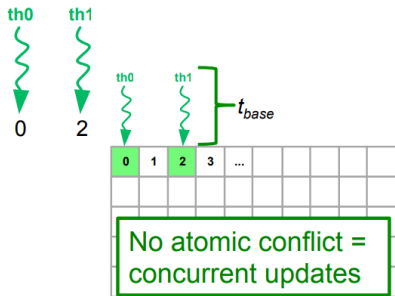## Divergence-free mapping

```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;

for (int stride = blockDim.x; stride > 1; stride >> 1){
    __syncthreads();

    if (t < stride)
        partialSum[t] += partialSum[t + stride];
}
```

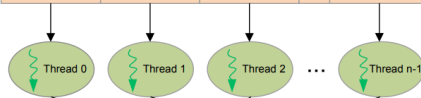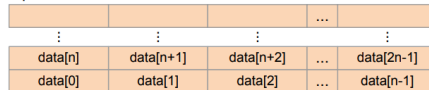Atomic conflicts

▶ Intra-warp **conflict degree** from 1 to 32

Histogram Calculation

Histograms count the number of data instances in disjoint categories (bins)

```
for (each pixel i in image I){
        Pixel = I[i]                          // Read pixel
        Pixel_new = Computation(Pixel)        // Optional computation
        Histogram[Pixel_new]++                // Vote in histogram bin
}
```
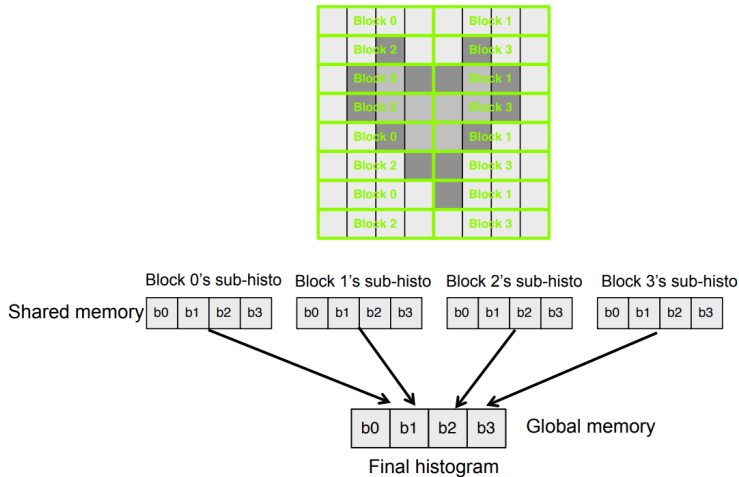
Histogram Calculation

Privatization: Per-block sub-histograms in shared memory

Summary

Traditional accelerator model

- ▶ Program structure
- ▶ Hardware architecture model

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

VANDERBILT
UNIVERSITY

Further Readings

Books for C CUDA

- *Programming Massively Parallel Processors*, David B. KIRK et Wen-mei W. HWU, Morgan Kaufmann, 2010
- *The CUDA handbook*, Nicholas WILT, Addison-Wesley, 2013
- https://docs.nvidia.com/cuda/

Python CUDA

- Nvidia's CUDA parallel computation API from Python
  https://mathema.tician.de/software/pycuda/
- Example of 2D FFT using PyFFT and PyCUDA
  https://wiki.tiker.net/PyCuda/Examples/2DFFT

VANDERBILT
UNIVERSITY