Graphics in 1980
Graphics in 2000

You fragged Ares
2nd place with 28
Graphics in 2015
GPUs in movies

- From Ariel in Little Mermaid to Brave
So ...

- GPUs are a steady market
  - Gaming
  - CAD-like activities
    - Traditional or not …
  - Visualisation
    - Scientific or not …

- GPUs are increasingly used for other types of applications
  - Number crunching in science, finance, image processing
  - (fast) Memory operations in big data processing
Another GPGPU history

Growth of GPU Computing

- 100M CUDA-Capable GPUs
- 150K CUDA Downloads
- 4,000 Academic Papers
- 430M CUDA-Capable GPUs
- 1.6M CUDA Downloads
- 37,000 Academic Papers

Should we use GPUs for all applications?!
1. Briefly on performance
2. GPGPUs
3. CUDA
4. If (time_left && vote)
   advanced CUDA
   else
   talk more about performance
Performance [1]

- **Latency/delay**
  - The time for one operation (instruction) to finish, $L$
  - To improve: minimize $L$
    - Lower is better

- **Throughput**
  - The number of operations (instructions) per time unit, $T$
  - To improve: maximize $T$
    - Higher is better
    - Thus, time per instruction decreases, on average

- **Example: 1 man builds a house in 10 days.**
  - Latency improvement: ...
  - Throughput improvement: ...
Performance [2]

- How do we get faster computers?
  - Faster processors and memory
    - Increase clock frequency → latency boost
  - Better memory techniques
    - Use memory hierarchies → latency boost
    - More memory closer to processor → latency boost
  - Better processing techniques
    - Use pipelining → throughput boost
  - More processing units (cores, threads, …)
    - Use parallelism/concurrency → throughput boost (only?)
  - Accelerators
    - Use specialized functional units → latency+throughput boost
Why multi- and many-cores?

Multi-cores = processors with multiple, homogeneous cores
Many-cores = GPUs & alikes
Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase....” Electronics Magazine 1965
Transistor Counts
Traditionally …

- More transistors = more functionality
- Improved technology = faster clocks = more speed

- Thus, every 18 months, we expect better and faster processors.

- They were all sequential: they execute one operation per clock cycle.
Revolution in Processors

Chip density is continuing to increase about 2x every 2 years

BUT

- Clock speed is not
- Performance per cycle is not
- Power is not
New ways to use transistors

- Parallelism on-chip: multi-core processors
  - Transformed in many-core processors.

- “Multicore revolution”
  - Every machine is a parallel machine.
  - Accelerators start playing an important role.
    - Specialized
    - Energy efficient
    - Used on demand

- Can all applications use this parallelism?
- Can we program all these architectures efficiently?
  - Performance? Productivity?
GPU vs. CPU performance

1 GFLOPs = 10^9 ops / second
GPU vs. CPU performance

1 GB/s = 8 \times 10^9 \text{ bits / second}
Why do we use many-cores?

- **Performance**
  - Large scale parallelism

- **Power Efficiency**
  - Use transistors more efficiently

- **Price (GPUs)**
  - Game market is huge, bigger than Hollywood
    - Gaming pays for our HPC needs!
  - Mass production, economy of scale

- **Prestige**
  - Reach ExaFLOP by 2019/2022 …
GPUs = the hardware
GPGPU = general purpose GPU
(typically related to software/programming)
GPGPU History

- Current generation: NVIDIA Kepler
  - 7.1B transistors
  - More cores, more parallelism, more performance
GPUs @ AMD

AMD Radeon Graphics Roadmap

<table>
<thead>
<tr>
<th>Performance</th>
<th>3DMark Fire Strike</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>7200, $449</td>
<td>HD 7970 GHz Edition</td>
<td></td>
</tr>
<tr>
<td>6700, $399</td>
<td>HD 7970</td>
<td></td>
</tr>
<tr>
<td>6150, $299</td>
<td>HD 7950</td>
<td></td>
</tr>
<tr>
<td>5000, $229</td>
<td>HD 7870 GHz Edition</td>
<td></td>
</tr>
<tr>
<td>4200, $189</td>
<td>HD 7850</td>
<td></td>
</tr>
<tr>
<td>2750, $119</td>
<td>HD 7770 GHz Edition</td>
<td></td>
</tr>
<tr>
<td>2050, $99</td>
<td>HD 7750</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GTX 680</td>
<td>6300, $499</td>
</tr>
<tr>
<td></td>
<td>GTX 670</td>
<td>5650, $399</td>
</tr>
<tr>
<td></td>
<td>GTX 660 Ti</td>
<td>5000, $299</td>
</tr>
<tr>
<td></td>
<td>GTX 660</td>
<td>4350, $229</td>
</tr>
<tr>
<td></td>
<td>GTX 650 Ti</td>
<td>2900, $149</td>
</tr>
<tr>
<td></td>
<td>GTX 650</td>
<td>2000, $109</td>
</tr>
<tr>
<td></td>
<td>GT 640</td>
<td>1200, $89</td>
</tr>
</tbody>
</table>

Performance, Price
GPUs @ ARM

ARM Mali
“Graphics Performance Leadership”

- Innovative Tri-architecture for performance and flexibility
- GPGPU computing with OpenCL 1.1 up to 68GFLOPS
- State of the art bandwidth reduction
- DirectX11 and next generation Khronos graphics standards up to 2Gpix/s

Mali-T604
- World’s first multicore embedded GPU
- High-performance graphics to beyond 1080p
- Leading power efficiency and reduced bandwidth

Mali-400 MP
- Ideal configuration for mid-range use-cases
- Efficient energy and bandwidth usage

Mali-200
- Entry level
- Leading anti-aliasing for superior image quality

All Mali GPUs support the Khronos APIs
OpenVG 1.1 and OpenGL ES 2.0 plus roadmap
(NVIDIA) GPUs

- **Architecture**
  - Many (100s) slim cores
  - Sets of (32 or 192) cores grouped into “multiprocessors” with shared memory
    - SM(X) = stream multiprocessors
  - Work as accelerators

- **Memory**
  - Shared L2 cache
  - Per-core caches + shared caches
  - Off-chip global memory

- **Programming**
  - Symmetric multi-threading
  - Hardware scheduler
NVIDIA’s GPU Architecture
Parallelism

- Data parallelism (fine-grain)
  - Restricted forms of task parallelism possible with newest generation of NVIDIA GPUs

- **SIMT** (Single Instruction Multiple Thread) execution
  - Many threads execute concurrently
    - Same instruction
    - Different data elements
    - HW automatically handles divergence
  - Not same as SIMD because of multiple register sets, addresses, and flow paths*

- Hardware multithreading
  - HW resource allocation & thread scheduling
    - Excess of threads to hide latency
    - Context switching is (basically) free

Integration into host system

- Typically PCI Express 2.0
- Theoretical speed 8 GB/s
  - Effective $\leq 6$ GB/s
  - In reality: 4 – 6 GB/s
- V3.0 recently available
  - Double bandwidth
  - Less protocol overhead
CPU vs. GPU

CPU

GPU
Different goals produce different designs!
- CPU must be good at everything
- GPUs focus on massive parallelism
  - Less flexible, more specialized

CPU: minimize latency experienced by 1 thread
- big on-chip caches
- sophisticated control logic

GPU: maximize throughput of all threads
- # threads in flight limited by resources => lots of resources (registers, etc.)
- multithreading can hide latency => no big caches
- share control logic across many threads
CPU vs. GPU

- Movie
- The Mythbusters
  - Jamie Hyneman & Adam Savage
  - Discovery Channel
- Appearance at NVIDIA’s NVISION 2008
GPU Hardware: NVIDIA
Fermi

- **Consumer:** GTX 480, 580
- **HPC:** Tesla C2050
  - More memory, ECC
  - 1.0 Tlop SP
  - 515 GFlop SP
- **16 streaming multiprocessors (SM)**
  - GTX 580: 16
  - GTX 480: 15
  - C2050: 14
- **SMs are independent**
- **768 KB L2 cache**
Fermi Streaming Multiprocessor (SM)

- 32 cores per SM (512 cores total)
- 64KB configurable L1 cache / shared memory
- 32,768 32-bit registers
CUDA Core Architecture

- Decoupled floating point and integer data paths
- Double precision throughput is 50% of single precision
- Integer operations optimized for extended precision
  - 64 bit and wider data element size
- Predication field for all instructions
- Fused-multiply-add
Memory architecture (since Fermi)

- Configurable L1 cache per SM
  - 16KB L1 cache / 48KB Shared
  - 48KB L1 cache / 16KB Shared

- Shared L2 cache
Kepler: the new SMX

- **Consumer:**
  - GTX680, GTX780, GTX-Titan

- **HPC**
  - Tesla K10..K40

- **SMX features**
  - 192 CUDA cores
    - 32 in Fermi
  - 32 Special Function Units (SFU)
    - 4 for Fermi
  - 32 Load/Store units (LD/ST)
    - 16 for Fermi
  - 3x Perf/Watt improvement
## A comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>FERMI GF100</th>
<th>FERMI GF104</th>
<th>KEPLER GK104</th>
<th>KEPLER GK110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Warps / Multiprocessor</td>
<td>48</td>
<td>48</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Threads / Multiprocessor</td>
<td>1536</td>
<td>1536</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Thread Blocks / Multiprocessor</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32-bit Registers / Multiprocessor</td>
<td>32768</td>
<td>32768</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Thread</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>255</td>
</tr>
<tr>
<td>Max Threads / Thread Block</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Shared Memory Size Configurations (bytes)</td>
<td>16K</td>
<td>16K</td>
<td>16K</td>
<td>16K</td>
</tr>
<tr>
<td></td>
<td>48K</td>
<td>48K</td>
<td>32K</td>
<td>32K</td>
</tr>
<tr>
<td></td>
<td>48K</td>
<td>48K</td>
<td>48K</td>
<td>48K</td>
</tr>
<tr>
<td>Max X Grid Dimension</td>
<td>$2^{16}-1$</td>
<td>$2^{16}-1$</td>
<td>$2^{32}-1$</td>
<td>$2^{32}-1$</td>
</tr>
<tr>
<td>Hyper-Q</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic Parallelism</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Maxwell: the newest SMM

- **Consumer:**
  - GTX 970, GTX 980, …

- **HPC:**
  - ?

- **SMM Features:**
  - 4 subblocks of 32 cores
  - Dedicated L1/LM per 64 cores
  - Dispatch/decode/registers per 32 cores
  - L2 cache: 2MB (~3x vs. Kepler)
  - 40 texture units
  - Lower power consumption
Programming many-cores
Parallelism

- **Threads**
  - Independent units of computation
  - Expected to execute in parallel
  - Write once, instantiate many times

- **Concurrent execution**
  - Threads execute in the same time if there are sufficient resources

- Assume a processor P with 10 cores and an application A with:
  - 10 threads: how long does A take?
  - 20 threads: how long does A take?
  - 33 threads: how long does A take?
Parallelism

- Synchronization = a thread’s execution must depend on other threads
  - Barrier = all threads wait to get to barrier before they continue
  - Shared variables = more threads RD/WR them
    - Locks = threads can use locks to protect the WR sections
  - Atomic operation = operation completed by a single thread at a time

- Thread scheduling = the order in which the threads are executed on the machine
  - User-based: programmer decides
  - OS-based: OS decides (e.g., Linux, Windows)
  - Hardware-based: hardware decides (e.g., GPUs)
Programming many-cores

= parallel programming:

- Choose/design algorithm
- Parallelize algorithm
  - Expose enough layers of parallelism
  - Minimize communication, synchronization, dependencies
  - Overlap computation and communication
- Implement parallel algorithm
  - Choose parallel programming model
  - (?) Choose many-core platform
- Tune/optimize application
  - Understand performance bottlenecks & expectations
  - Apply platform specific optimizations
  - (?) Apply application & data specific optimizations
Programming GPUs in CUDA
CUDA

- CUDA: Scalable parallel programming
  - C/C++ extensions
    - Other wrappers exist

- Straightforward mapping onto hardware
  - Hierarchy of threads (to map to cores)
    - Configurable at logical level
  - Various memory spaces (to map to physical spaces)
    - Usable via variable scopes

- Scale to 1000s of cores & 100,000s of threads
  - GPU threads are lightweight
  - GPUs need 1000s of threads for full utilization
CUDA Model of Parallelism

- CUDA virtualizes the physical hardware
  - A block is a virtualized streaming multiprocessor
    - threads, shared memory
  - A thread is a virtualized scalar processor
    - registers, PC, state
- Threads are scheduled onto physical hardware without pre-emption
  - threads/blocks launch & run to completion
  - blocks must be independent
CUDA Model of Parallelism

Software

Thread

Thread Block

Thread Grid

GPU

Thread Processor

Multi-processor

Device
Hierarchy of threads

Thread → Block → Grid
Using CUDA

- Two parts of the code:
  - Device code = GPU code = kernel(s)
    - Sequential program
    - Write for 1 thread, execute for all
  - Host code = CPU code
    - Instantiate grid + run the kernel
    - Memory allocation, management, deallocation
    - C/C++/Java/Python/…

- Host-device communication
  - Explicit / implicit via PCI/e
  - Minimum: data input/output
All this happens from the host code.
Grids, Thread Blocks and Threads

Grid

Thread Block 0, 0

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3

Thread Block 0, 1

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3

Thread Block 0, 2

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3

Thread Block 1, 0

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3

Thread Block 1, 1

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3

Thread Block 1, 2

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3
## Kernels and grids

- **Launch kernel** \(12 \times 6 = 72\) instances

```cpp
myKernel<<<numBlocks,threadsPerBlock>>>(...);
```

- `dim3 threadsPerBlock(3,4);
  - `threadsPerBlock.x = 3`
  - `threadsPerBlock.y = 4`
  - Each thread:
    - `(threadIdx.x, threadIdx.y)`

- `dim3 numBlocks(2,3);
  - `blockDim.x = 2`
  - `blockDim.y = 3`
  - Each block:
    - `(blockIdx.x, blockIdx.y)`

### Grid

<table>
<thead>
<tr>
<th>Thread Block 0, 0</th>
<th>Thread Block 0, 1</th>
<th>Thread Block 0, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
</tr>
<tr>
<td>2.0</td>
<td>2.1</td>
<td>2.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread Block 1, 0</th>
<th>Thread Block 1, 1</th>
<th>Thread Block 1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
</tr>
<tr>
<td>2.0</td>
<td>2.1</td>
<td>2.2</td>
</tr>
</tbody>
</table>
Multiple Device Memory Scopes

- **Per-thread private memory**
  - Each thread has its own local memory
  - Stacks, other private data, *registers*

- **Per-SM shared memory**
  - Small memory close to the processor, low latency

- **Device memory**
  - GPU frame buffer
  - Can be accessed by any thread in any SM
Memory Spaces in CUDA

Host

- Shared data (per block)
- Private data
- Global data

Grid

- Block (0, 0)
  - Shared Memory
  - Registers
  - Thread (0, 0)
  - Thread (1, 0)

- Block (1, 0)
  - Shared Memory
  - Registers
  - Thread (0, 0)
  - Thread (1, 0)

Device Memory

Constant Memory

Texture Memory
Device Memory

- CPU and GPU have separate memory spaces
  - Data is moved across PCI-e bus
  - Use functions to allocate/set/copy memory on GPU
  - Very similar to corresponding C functions

- Pointers are just addresses
  - Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa
Additional memories

- **Textures**
  - Read-only
  - Data resides in device memory
  - Different read path, includes specialized caches

- **Constant memory**
  - Data resides in device memory
  - Manually managed
  - Small (e.g., 64KB)
  - Assumes all threads in a block read the same addresses
    - Serializes otherwise
C for CUDA

- Philosophy: provide minimal set of extensions necessary

- Function qualifiers:
  ```
  __global__ void my_kernel() { }
  __device__ float my_device_func() { }
  ```

- Execution configuration:
  ```
  dim3 gridDim(100, 50); // 5000 thread blocks
  dim3 blockDim(4, 8, 8); // 256 threads per block (1.3M total)
  my_kernel <<< gridDim, blockDim >>> (...) // Launch kernel
  ```

- Built-in variables and functions valid in device code:
  ```
  dim3 gridDim; // Grid dimension
  dim3 blockDim; // Block dimension
  dim3 blockIdx; // Block index
  dim3 threadIdx; // Thread index
  void syncthreads(); // Thread synchronization
Our first CUDA program
First CUDA program

- Determine mapping of operations and data to threads
- Write kernel(s)
  - Sequential code
  - Written per-thread
- Determine block geometry
  - Threads per block, blocks per grid
  - Number of grids (>= number of kernels)
- Write host code
  - Memory initialization and copying to device
  - Kernel(s) launch(es)
  - Results copying to host
- Optimize the kernels
Vector add: sequential

```c
void vector_add(int size, float* a, float* b, float* c) {
    for(int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}
```
How do we parallelize this?

- What does each thread compute?
  - One addition per thread
  - Each thread deals with *different* elements
  - How do we know which element?
    - Compute a mapping of the grid to the data
      - Any mapping will do!
All this happens from the host code.
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = ?
    C[i] = A[i] + B[i];
}
Calculating the global thread index

“global” thread index:

\[ \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x}; \]
Calculating the global thread index

“global” thread index:

\[ \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x} \]

\[ 4 \times 2 + 1 = 9 \]
// compute vector sum \( c = a + b \)
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
Processing flow

All this happens from the host code.
Vector add: Launch kernel

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

```c
int main() {
    // initialization code here ...
    N = 5120;
    // launch N/256 blocks of 256 threads each
    vector_add<<<N/256, 256>>>(deviceA, deviceB, deviceC);
    // cleanup code here ...
}
```

(can be in the same file)
// compute vector sum \( c = a + b \)
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

What if \( N = 5000 \)?

```c
int main() {
    // initialization code here ...
    N = 5000;
    // launch \( \frac{N}{256} \) blocks of 256 threads each
    vector_add<<< N/256, 256 >>>(deviceA, deviceB, deviceC);
    // cleanup code here ...
}
```

(host code)

(gpu code)
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i<N) C[i] = A[i] + B[i];
}

int main() {
    // initialization code here ...
    N = 5000;
    // launch N/256 blocks of 256 threads each
    vector_add<<< N/256+1,256 >>>(deviceA, deviceB, deviceC);
    // cleanup code here ...
}

What if N = 5000?
Host (CPU) manages device (GPU) memory:

- `cudaMalloc(void **pointer, size_t nbytes)`
- `cudaMemset(void *pointer, int val, size_t count)`
- `cudaFree(void* pointer)`

```c
int n = 1024;
int nbytes = n * sizeof(int);
int* data = 0;
cudaMalloc(&data, nbytes);
cudaMemset(data, 0, nbytes);
cudaFree(data);
```
Data Copies

- `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn’t start copying until previous CUDA calls complete

- `enum cudaMemcpyKind`
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

- Non-blocking copies are also available
  - DMA transfers, overlap computation and communication
int main(int argc, char** argv) {
    int size = N * sizeof(float);

    // allocate host memory
    hostA = malloc(size);
    hostB = malloc(size);
    hostC = malloc(size);

    // initialize A, B arrays here...

    // allocate device memory
    cudaMalloc(&deviceA, size);
    cudaMalloc(&deviceB, size);
    cudaMalloc(&deviceC, size);
}
// transfer the data from the host to the device
cudaMemcpy(deviceA, hostA, size, cudaMemcpyHostToDevice);
cudaMemcpy(deviceB, hostB, size, cudaMemcpyHostToDevice);

// launch N/256 blocks of 256 threads each
vector_add<<<N/256, 256>>>(deviceA, deviceB, deviceC);

// transfer the result back from the GPU to the host
cudaMemcpy(hostC, deviceC, size, cudaMemcpyDeviceToHost);
}

Done with the host code!
Summary

- Determine mapping of operations and data to threads

- Write kernel(s)
  - Sequential code
  - Written per-thread

- Determine block geometry
  - Threads per block, blocks per grid
  - Number of grids (>= number of kernels)

- Write host code
  - Memory initialization and copying to device
  - Kernel(s) launch(es)
  - Results copying to host

- Optimize the kernels
Let’s try this in practice

Run on DAS4

You need an ssh client

- On Linux/Mac: terminal will do
- On Windows: download putty
  - [http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html](http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html)

Use vim to see the files

Follow the directions in the manual for Assignment 1 and 2
Vim commands:
- i – enables editing
- Esc – finishes editing
- :qw – exit and save changes
- :q! – exit without saving changes

“make” – to compile
Advanced CUDA

Scheduling and synchronization
Thread Scheduling

- Order in which thread blocks are scheduled is undefined!
  - any possible interleaving of blocks should be valid
  - presumed to run to completion without preemption
  - can run in any order
  - can run concurrently OR sequentially

- Order of threads within a block is also undefined!
Q: How do we do global synchronization with these scheduling semantics?
Global synchronization

Q: How do we do global synchronization with these scheduling semantics?

A1: Not possible!
Global synchronization

Q: How do we do global synchronization with these scheduling semantics?

A1: Not possible!

A2: Finish a grid, and start a new one!
Q: How do we do global synchronization with these scheduling semantics?

A1: Not possible!

A2: Finish a grid, and start a new one!

```c
step1<<<grid1,blk1>>>(...);
// CUDA ensures that all writes from step1 are complete.
step2<<<grid2,blk2>>>(...);
```

We don't have to copy the data back and forth!
Atomics

- Guarantee that only a single thread has access to a piece of memory during an operation
  - No loss of data
  - Ordering is still arbitrary

- Different types of atomic instructions
  - Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
  - On device memory and/or shared memory

- Much more expensive than load + operation + store
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;
}
Example: Histogram

// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int c = colors[i];
    buckets[c] += 1;
}
Example: Histogram

// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter atomically

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
CUDA: optimizing your application

1. Occupancy
2. Shared Memory
3. Coalescing
4. Streams
5. Shared Memory Bank Conflicts
SMs implement zero-overhead warp scheduling
- A warp is a group of 32 threads that runs concurrently on an SM
- At any time, the number of warps concurrently executed by an SM is limited by its number of cores.
- Warps whose next instruction has its inputs ready for consumption are eligible for execution
- Eligible Warps are selected for execution on a prioritized scheduling policy
- All threads in a warp execute the same instruction when selected

Instruction:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>TB1</td>
<td>W1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB2</td>
<td>W1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB3</td>
<td>W1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB3</td>
<td>W2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB2</td>
<td>W1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB1</td>
<td>W1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB1</td>
<td>W2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB1</td>
<td>W3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB3</td>
<td>W2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TB = Thread Block, W = Warp
Stalling warps

- What happens if all warps are stalled?
  - No instruction issued → performance lost

- Most common reason for stalling?
  - Waiting on global memory

- If your code reads global memory every couple of instructions
  - You should try to maximize occupancy
Occupancy

- What determines occupancy?
- Limited resources!
  - Register usage per thread
  - Shared memory per thread block
Resource Limits (1)

- Pool of registers and shared memory per SM
  - Each thread block grabs registers & shared memory
  - If one or the other is fully utilized ➔ no more thread blocks
Resource Limits (2)

- Can only have $P$ thread blocks per SM
  - If they’re too small, can’t fill up the SM
  - Need 128 threads / block on gt200 (4 cycles/instruction)
  - Need 192 threads / block on Fermi (6 cycles/instruction)

- Higher occupancy has diminishing returns for hiding latency
Hiding Latency with more threads

Throughput, 32-bit words
How do you know what you’re using?

- Use compiler flags to get register and shared memory usage
  - “nvcc -Xptxas -v”
- Use the NVIDIA Profiler
- Plug those numbers into CUDA Occupancy Calculator
- Maximize occupancy for improved performance
  - Empirical rule! Don’t overuse!
1. Select Compute Capability (click): 1.3

2. Enter your resource usage:
   - Number of Threads Per Block: 128
   - Registers Per Thread: 3584
   - Shared Memory Per Block (bytes): 1024

3. GPU Occupancy Data is displayed here and in the graphs:
   - Active Threads per Multithread: 512
   - Active Warps per Multithread: 128
   - Active Thread Blocks per Multithread: 4
   - Occupancy of each Multithread: 50%

Physical Limits for GPU Compute Capability: 1.3

- Threads per Warp: 32
- Warps per Multithread: 32
- Threads per Multithread: 1024
- Thread Blocks per Multithread: 8
- Total # of 32-bit registers per Multithread: 16384
- Register allocation unit size: 512
- Register allocation granularity: block
- Shared Memory per Multithread (bytes): 16384
- Shared Memory Allocation unit size: 512
- Warp allocation granularity (for register allocation): 2

Allocation Per Thread Block:
- Warps: 4
- Registers: 3584
- Shared Memory: 1024

These data are used in computing the occupancy data in blue.

Maximum Thread Blocks Per Multithread: Blocks Limited by Max Warps/Blocks per Multithread: 8
Limited by Registers per Multithread: 4
Limited by Shared Memory per Multithread: 16
Thread Block Limit Per Multithread highlighted: RED

CUDA Occupancy Calculator
Version: 2.0

Copyright and License
“I heard GPU branching is expensive. Is this true?”

```c
__global__ void Divergence(float* dst, float* src)
{
    float value = 0.0f;

    if ( threadIdx.x % 2 == 0 )
        // active threads : 50%
        value = src[0] + 5.0f;
    else
        // active threads : 50%
        value = src[0] - 5.0f;

    dst[index] = value;
}
```
Worst case performance loss:
50% compared with the non divergent case.
Another example

Time (clocks)

ALU 1  ALU 2  ...  ...  ALU 8

T  T  F  T  F  F  F  F  F

Not all ALUs do useful work!
Worst case: 1/8 peak performance

(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

<unconditional code>

float x = A[i];
if (x > 0) {
    float tmp = exp(x,5.f);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}
<resume unconditional code>

result[i] = x;
Performance penalty?

- Depends on the amount of divergence
  - Worst case: 1/32 performance
    - When each thread does something different

- Depends on whether branching is data- or ID-dependent
  - If ID – consider grouping threads differently
  - If data – consider sorting

- Non-diverging warps => NO performance penalty
  - In this case, branches are not expensive …
CUDA: optimizing your application

1. Occupancy
2. Shared Memory
3. Coalescing
4. Streams
5. Shared Memory Bank Conflicts
Matrix multiplication example

- \( C = A \times B \)
- Each element \( C_{i,j} \)
  
  \[ = \text{dot} (\text{row}(A,i), \text{col}(B,j)) \]
- Parallelization strategy
  - Each thread computes element in \( C \)
  - 2D kernel
Matrix multiplication implementation

```c
__global__ void mat_mul(float *a, float *b,
                        float *c, int width)
{
    // calc row & column index of output element
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;

    float result = 0;

    // do dot product between row of a and column of b
    for(int k = 0; k < width; k++) {
        result += a[row*width+k] * b[k*width+col];
    }

    c[row*width+col] = result;
}
```
# Matrix multiplication performance

<table>
<thead>
<tr>
<th>Loads per dot product term</th>
<th>2 (a and b) = 8 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOPS</td>
<td>2 (multiply and add)</td>
</tr>
<tr>
<td>AI</td>
<td>2 / 8 = 0.25</td>
</tr>
<tr>
<td>Performance GTX 580</td>
<td>1581 GFLOPs</td>
</tr>
<tr>
<td>Memory bandwidth GTX 580</td>
<td>192 GB/s</td>
</tr>
<tr>
<td>Attainable performance</td>
<td>192 * 0.25 = 48 GFLOPS</td>
</tr>
<tr>
<td>Maximum efficiency</td>
<td>3.0 % of theoretical peak</td>
</tr>
</tbody>
</table>
Data reuse

- Each input element in A and B is read WIDTH times
- Load elements into shared memory
- Have several threads use local version to reduce the memory bandwidth
Using shared memory

- Partition kernel loop into phases
- In each thread block, load a tile of both matrices into shared memory each phase
- Each phase, each thread computes a partial result
Matrix multiply with shared memory

```c
__global__ void mat_mul(float *a, float *b, float *c, int width) {

    // shorthand
    int tx = threadIdx.x, ty = threadIdx.y;
    int bx = blockIdx.x, by = blockIdx.y;

    // allocate tiles in shared memory
    __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

    // calculate the row & column index
    int row = by*blockDim.y + ty;
    int col = bx*blockDim.x + tx;

    float result = 0;
```
Matrix multiply with shared memory

// loop over input tiles in phases
for(int p = 0; p < width/TILE_WIDTH; p++) {
    // collaboratively load tiles into shared memory
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
    __syncthreads();

    // dot product between row of s_a and col of s_b
    for(int k = 0; k < TILE_WIDTH; k++) {
        result += s_a[ty][k] * s_b[k][tx];
    }
    __syncthreads();
}

c[row*width+col] = result;
Use of Barriers in mat_mul

- Two barriers per phase:
  - `__syncthreads` after all data is loaded into shared memory
  - `__syncthreads` after all data is read from shared memory
  - Second `__syncthreads` in phase p guards the load in phase p+1

- Use barriers to guard data
  - Guard against using uninitialized data
  - Guard against corrupting live data
Matrix multiplication performance

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>shared memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global loads</td>
<td>$2N^3 \times 4 \text{ bytes}$</td>
<td>$(2N^3 / \text{TILE WIDTH}) \times 4 \text{ bytes}$</td>
</tr>
<tr>
<td>Total ops</td>
<td>$2N^3$</td>
<td>$2N^3$</td>
</tr>
<tr>
<td>AI</td>
<td>0.25</td>
<td>$0.25 \times \text{TILE WIDTH}$</td>
</tr>
</tbody>
</table>

Performance GTX 580

<table>
<thead>
<tr>
<th></th>
<th>1581 GFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory bandwidth GTX 580</td>
<td>192 GB/s</td>
</tr>
<tr>
<td>AI needed for peak</td>
<td>$1581 / 192 = 8.23$</td>
</tr>
<tr>
<td>TILE_WIDTH required to achieve peak</td>
<td>$0.25 \times \text{TILE WIDTH} = 8.23$, \text{TILE_WIDTH} = 32.9</td>
</tr>
</tbody>
</table>
CUDA: optimizing your application

1. Occupancy
2. Shared Memory
3. Coalescing
4. Streams
5. Shared Memory Bank Conflicts
Coalescing

traditional multi-core optimal memory access pattern

<table>
<thead>
<tr>
<th>thread 0</th>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0</td>
<td>t = 0</td>
<td>t = 0</td>
<td>t = 0</td>
</tr>
<tr>
<td>t = 1</td>
<td>t = 1</td>
<td>t = 1</td>
<td>t = 1</td>
</tr>
</tbody>
</table>

address 0
address 1
address 2
address 3
address 4
address 5
address 6
address 7

many-core GPU optimal memory access pattern

<table>
<thead>
<tr>
<th>thread 0</th>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0</td>
<td>t = 0</td>
<td>t = 0</td>
<td>t = 0</td>
</tr>
<tr>
<td>t = 1</td>
<td>t = 1</td>
<td>t = 0</td>
<td>t = 1</td>
</tr>
<tr>
<td>t = 1</td>
<td>t = 0</td>
<td>t = 1</td>
<td>t = 1</td>
</tr>
</tbody>
</table>

address 0
address 1
address 2
address 3
address 4
address 5
address 6
address 7
Consider the stride of your accesses

```c
__global__ void foo(int* input, float3* input2) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    // Stride 1, OK!
    int a = input[i];

    // Stride 2, half the bandwidth is wasted
    int b = input[2*i];

    // Stride 3, 2/3 of the bandwidth wasted
    float c = input2[i].x;
}
```
Example: Array of Structures (AoS)

```c
struct record {
    int key;
    int value;
    int flag;
};

record *d_records;
cudaMalloc((void**)&d_records, ...);
```
Example: Structure of Arrays (SoA)

Struct SoA {
    int* keys;
    int* values;
    int* flags;
};

SoA d_SoA_data;
cudaMalloc((void**) &d_SoA_data.keys, ...);
cudaMalloc((void**) &d_SoA_data.values, ...);
cudaMalloc((void**) &d_SoA_data.flags, ...);
__global__ void kernel(record* AoS_data,
                     SoA   SoA_data) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    // AoS wastes bandwidth
    int key1 = AoS_data[i].key;

    // SoA efficient use of bandwidth
    int key2 = SoA_data.keys[i];
}
Memory Coalescing

- Structure of arrays is often better than array of structures
- Very clear win on regular, stride 1 access patterns
- Unpredictable or irregular access patterns are case-by-case
- Can lose a factor of 10x – 30x!
CUDA: Streams

1. Occupancy
2. Shared Memory
3. Coalescing
4. Streams
5. Shared Memory Bank Conflicts
What are streams?

- Stream = a sequence of operations that execute on the device in the order in which they are issued by the host code.
- Same stream: In-Order execution
- Different streams: Out-of-Order execution

- Default stream = Synchronizing stream
  - No operation in the default stream can begin until all previously issued operations in any stream on the device have completed.
  - An operation in the default stream must complete before any other operation in any stream on the device can begin.
CUDA Memory Copy Operations Example

```c
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a);
CpuFunction(b);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- **All operations happen in the same stream**
- **Device (GPU)**
  - Synchronous execution
    - all operations execute (in order), one after the previous has finished
  - Unaware of CpuFunction()
- **Host (CPU)**
  - Launches increment and regains control
  - *May* execute CpuFunction *before* increment has finished
  - Final copy starts *after* both increment and CpuFunction() have finished
Non-default streams

- Enable asynchronous execution and overlaps
  - Require special creation/deletion of streams
    - `cudaStreamCreate(&stream1)`
    - `cudaStreamDestroy(stream1)`
  - Special memory operations
    - `cudaMemcpyAsync(deviceMem, hostMem, size, cudaMemcpyHostToDevice, stream1)`
  - Special kernel parameter (the 4th one)
    - `increment<<<1, N, 0, stream1>>>(d_a)`

- Synchronization
  - All streams
    - `cudaDeviceSynchronize()`
  - Specific stream:
    - `cudaStreamSynchronize(stream1)`
Computation vs. communication

//Single stream, numBytes = 16M, numElements = 4M
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
kernel<<blocks,threads>>(d_a, firstElement);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);

C1060 (pre-Fermi): 12.9ms

C2050 (Fermi): 9.9ms
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, stream[i]);
    kernel<<<blocks,threads,0,stream[i]>>>(d_a, offset);
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, stream[i]);
}

C1060 (pre-Fermi): 13.63 ms (worse than sequential)

C2050 (Fermi): 5.73 ms (better than sequential)
for (int i = 0; i < nStreams; ++i) offset[i]=i * streamSize;
for (int i = 0; i < nStreams; ++i)
cudaMemcpyAsync(&d_a[offset[i]], &a[offset[i]], streamBytes, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < nStreams; ++i)
kernel<<<blocks,threads,0,stream[i]>>>(d_a, offset);
for (int i = 0; i < nStreams; ++i)
cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost, stream[i]);

C1060 (pre-Fermi): 8.84 ms (better than sequential)

<table>
<thead>
<tr>
<th>Copy Engine</th>
<th>H2D - 1</th>
<th>H2D - 2</th>
<th>H2D - 3</th>
<th>H2D - 4</th>
<th>D2H - 1</th>
<th>D2H - 2</th>
<th>D2H - 3</th>
<th>D2H - 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Engine</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C2050 (Fermi): 7.59 ms (better than sequential, worse than v1)

| H2D Engine | 1 | 2 | 3 | 4 |
| Kernel Engine | 1 | 2 | 3 | 4 |
| D2H Engine | 1 | 2 | 3 | 4 |

CUDA: optimizing your application

1. Occupancy
2. Shared Memory
3. Coalescing
4. Streams
5. Shared Memory Bank Conflicts
Shared Memory Banks

- Shared memory is banked
  - Only matters for threads within a warp
  - Full performance with some restrictions
    - Threads can each access different banks
    - Or can all access the same value

- Consecutive words are in different banks

- If two or more threads access the same bank but different value, we get bank conflicts
Bank Addressing Examples: OK

- No Bank Conflicts

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Bank 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Bank 1</td>
</tr>
<tr>
<td>Thread 2</td>
<td>Bank 2</td>
</tr>
<tr>
<td>Thread 3</td>
<td>Bank 3</td>
</tr>
<tr>
<td>Thread 4</td>
<td>Bank 4</td>
</tr>
<tr>
<td>Thread 5</td>
<td>Bank 5</td>
</tr>
<tr>
<td>Thread 6</td>
<td>Bank 6</td>
</tr>
<tr>
<td>Thread 7</td>
<td>Bank 7</td>
</tr>
</tbody>
</table>

- No Bank Conflicts

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Bank 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Bank 1</td>
</tr>
<tr>
<td>Thread 2</td>
<td>Bank 2</td>
</tr>
<tr>
<td>Thread 3</td>
<td>Bank 3</td>
</tr>
<tr>
<td>Thread 4</td>
<td>Bank 4</td>
</tr>
<tr>
<td>Thread 5</td>
<td>Bank 5</td>
</tr>
<tr>
<td>Thread 6</td>
<td>Bank 6</td>
</tr>
<tr>
<td>Thread 7</td>
<td>Bank 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 15</th>
<th>Bank 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 15</td>
<td>Bank 15</td>
</tr>
</tbody>
</table>
Bank Addressing Examples: BAD

- 2-way Bank Conflicts
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 8
  - Thread 9
  - Thread 10
  - Thread 11
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 15

- 8-way Bank Conflicts
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 5
  - Thread 6
  - Thread 7
  - Thread 15
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 8
  - Bank 9
  - Bank 15
Trick to Assess Performance Impact

- Change all shared memory reads to the same value
- All broadcasts = no conflicts
- Will show how much performance could be improved by eliminating bank conflicts

- The same doesn’t work for shared memory writes
  - So, replace shared memory array indices with threadIdx.x
  - (Could also be done for the reads)
OpenCL: Programming GPUs, CPUs, APU
Portability

- Inter-family vs inter-vendor
  - NVIDIA Cuda runs on all NVIDIA GPU families
  - OpenCL runs on all GPUs, Cell, CPUs

- Parallelism portability
  - Different architecture requires different granularity
  - Task vs data parallel

- Performance portability
  - Can we express platform-specific optimizations?
The Khronos group
OpenCL: Open Compute Language

- Architecture independent
- Explicit support for many-cores
- Low-level host API
  - Uses C library, no language extensions
- Separate high-level kernel language
  - Explicit support for vectorization
- Run-time compilation
- Architecture-dependent optimizations
  - Still needed
  - Possible
# Cuda vs OpenCL Terminology

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Work item</td>
</tr>
<tr>
<td>Thread block</td>
<td>Work group</td>
</tr>
<tr>
<td>Device memory</td>
<td>Global memory</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Constant memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Local memory</td>
</tr>
<tr>
<td>Local memory</td>
<td>Private memory</td>
</tr>
</tbody>
</table>
## Cuda vs OpenCL Qualifiers

### Functions

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong></td>
<td>__kernel</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>(no qualifier needed)</td>
</tr>
</tbody>
</table>

### Variables

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>constant</strong></td>
<td>__constant</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>__global</td>
</tr>
<tr>
<td><strong>shared</strong></td>
<td>__local</td>
</tr>
</tbody>
</table>
### Cuda vs OpenCL Indexing

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>gridDim</td>
<td>get_num_groups()</td>
</tr>
<tr>
<td>blockDim</td>
<td>get_local_size()</td>
</tr>
<tr>
<td>blockIdx</td>
<td>get_group_id()</td>
</tr>
<tr>
<td>threadIdx</td>
<td>get_local_id()</td>
</tr>
<tr>
<td>Calculate manually</td>
<td>get_global_id()</td>
</tr>
<tr>
<td>Calculate manually</td>
<td>get_global_size()</td>
</tr>
</tbody>
</table>

```cpp
__syncthreads() → barrier()
```
Vector add: Cuda vs OpenCL kernel

**CUDA**

```c
__global__ void vectorAdd(float* a, float* b, float* c) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  c[index] = a[index] + b[index];
}
```

**OpenCL**

```c
__kernel void vectorAdd(__global float* a, __global float* b, __global float* c) {
  int index = get_global_id(0);
  c[index] = a[index] + b[index];
}
```
const size_t workGroupSize = 256;
const size_t nrWorkGroups = 3;
const size_t totalSize = nrWorkGroups * workGroupSize;

cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

// create properties list of key/values, 0-terminated.
cl_context_properties props[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0
};

cl_context context = clCreateContextFromType(props,
                                              CL_DEVICE_TYPE_GPU, 0, 0, 0);
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 1,
                &device, NULL);

// create command queue on 1st device the context reported
cl_command_queue commandQueue =
    clCreateCommandQueue(context, device, 0, 0);

// create & compile program
cl_program program = clCreateProgramWithSource(context, 1,
                                                &programSource, 0, 0);
clBuildProgram(program, 0, 0, 0, 0, 0, 0, 0);

// create kernel
cl_kernel kernel = clCreateKernel(program, "vectorAdd", 0);
float* A, B, C = new float[totalSize]; // alloc host vecs
// initialize host memory here...

// allocate device memory
cl_mem deviceA = clCreateBuffer(context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    totalSize * sizeof(cl_float), A, 0);

cl_mem deviceB = clCreateBuffer(context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    totalSize * sizeof(cl_float), B, 0);

cl_mem deviceC = clCreateBuffer(context,
    CL_MEM_WRITE_ONLY, totalSize * sizeof(cl_float), 0, 0);
// setup parameter values
clSetKernelArg(kernel, 0, sizeof(cl_mem), &deviceA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &deviceB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &deviceC);

clEnqueueNDRangeKernel(commandQueue, kernel, 1, 0,
    &totalSize, &workGroupSize, 0,0,0); // execute kernel

// copy results from device back to host, blocking
clEnqueueReadBuffer(commandQueue, deviceC, CL_TRUE, 0,
    totalSize * sizeof(cl_float), C, 0, 0, 0);

delete[] A, B, C; // cleanup
clReleaseMemObject(deviceA); clReleaseMemObject(deviceB);
clReleaseMemObject(deviceC);
Summary and Conclusions
Higher performance cannot be reached by increasing clock frequencies anymore.

Solution: introduction of large-scale parallelism

Multiple cores on a chip

Today:
- Up to 48 CPU cores in a node
- Up to 3200 compute elements on a single GPU

Host system can contain multiple GPUs: 10,000+ cores

We can build clusters of these nodes!

Future: 100,000s – millions of cores?
Many different types of many-core hardware

Very different properties
- Performance
- Programmability
- Portability

It's all about the memory

Choose the right platform for your application
- Arithmetic intensity / Operational intensity
- Roofline model
Open questions

- New application domains – e.g., signal processing, graph processing.
  - Performance analysis
  - Performance prediction
  - Modeling
- Memory patterns understanding, description, detection, automatic improvement
  - Local memory usage
- Heterogeneous computing
  - Using both the host and the device
- Application-device fitting
Questions?

- Slides are/will be available
- If you are interested in working with us on using GPUs for new applications, let us know!

A.L.Varbanescu@uva.nl
Backup slides
// For algorithms where the amount of work per item // is highly non-uniform, it often makes sense to // continuously grab work from a queue.

__global__
void workq(int* work_q, int* q_counter, 
            int queue_max, int* output)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int q_index = atomicInc(q_counter, queue_max);
    int result = do_work(work_q[q_index]);
    output[q_index] = result;
}
//Adjacent Difference application:
//compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    //compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        //each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
Using shared memory

// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread’s global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}

How do we use device memory bandwidth?
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread’s global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}

The next thread also reads input[i]
Using shared memory

__global__ void adj_diff(int *result, int *input) {
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
  // each thread reads 1 device memory elt, stores it in s_data
  s_data[threadIdx.x] = input[i];

  // avoid race condition: ensure all loads are complete
  __syncthreads();

  if(threadIdx.x > 0) {
    result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
  } else if(i > 0) {
    // I am thread 0 in this block: handle thread block boundary
    result[i] = s_data[threadIdx.x] - input[i-1];
  }
}
Using shared memory: coalescing

```c
__global__ void adj_diff(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
    // each thread reads 1 device memory elt, stores it in s_data
    s_data[threadIdx.x] = input[i];   // COALESCED ACCESS!

    // avoid race condition: ensure all loads are complete
    __syncthreads();

    if(threadIdx.x > 0) {
        result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
    } else if(i > 0) {
        // I am thread 0 in this block: handle thread block boundary
        result[i] = s_data[threadIdx.x] - input[i-1];
    }
}
```
Backup slides
CPU vs GPU Chip

AMD Magny-Cours (6 cores)  ATI 4870 (800 cores)

2 billion transistors 1 billion transistors
346 mm² 256 mm²
ATI GPUs
Latest generation ATI

- Southern Islands
  - 1 chip: HD 7970
    - 2048 cores
    - 264 GB/sec memory bandwidth
    - 3.8 tflops single, 947 gflops double precision
    - Maximum power: 250 Watts
    - 399 euros!
  - 2 chips: HD 7990
    - 4096 cores, 7.6 tflops
- Comparison: entire 72-node DAS-4 VU cluster has 4.4 tflops
ATI 5870 architecture overview
Each of the 20 SIMD engines has:

- 16 thread processors x 5 stream cores = 80 scalar stream processing units
- 20 * 16 * 5 = 1600 cores total
- 32KB Local Data Share
- its own control logic and runs from a shared set of threads
- a dedicated fetch unit with 8KB L1 cache
- a 64KB global data share to communicate with other SIMD engines
Each thread processor includes:
- 4 stream cores + 1 special function stream core
- general purpose registers
- FMA in a single clock
ATI 5870 Memory Hierarchy

- EDC (Error Detection Code)
  - CRC Checks on Data Transfers for Improved Reliability at High Clock Speeds

- Bandwidths
  - Up to 1 TB/sec L1 texture fetch bandwidth
  - Up to 435 GB/sec between L1 & L2
  - 153.6 GB/s to device memory
  - PCI-e 2.0, 16x: 8GB/s to main memory
Unified Load/Store Addressing

Non-unified Address Space

Local

* p_local

Shared

* p_shared

Device

0

* p_device

32-bit

Unified Address Space

Local

Shared

Device

0

*p

40-bit
A Common Programming Strategy

- Partition data into subsets that fit into shared memory
A Common Programming Strategy

- Handle each data subset with one thread block
Load the subset from device memory to shared memory, using multiple threads to exploit memory-level parallelism.
A Common Programming Strategy

- Perform the computation on the subset from shared memory
- Copy the result from shared memory back to device memory