GPU COMPUTING

Part 1

Ana Lucia Varbanescu (UvA)
HPC computing

• Big Data, Big Simulation, Big Science

• Challenges
  • Compute and storage
  • Efficiency
    • Performance vs. Energy

• HPC focuses traditionally on performance, but now moving towards efficiency
  • Traditional HPC: complex machines, on-demand
  • Modern HPC: more and more based on existing machines, put together in dense clusters/datacenters

• HPC is expanding to new application domains.
SAR imaging
Sound ray tracing

• A collaboration with Dutch NLR
• Simulate the sound propagation
  • from an aircraft to receivers
• Assess aircraft flyover noise during the aircraft take-off and approach procedures
Sound ray tracing

(a) Aircraft altitude vs. sound speed
(b) Relative distance from source vs. workload

Launch Angle [deg]
Workload (# iterations)
Ray ID (parallelization dimension)
BRIEFLY ON HARDWARE
Moore’s Law

- 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
Evolution of processors

Chip density can increase about 2x every 2 years

BUT
- Clock speed is not
- Power is not
- Instruction Level Parallelism is not

What does this mean in practice?
New ways to use transistors

Improve PERFORMANCE by using parallelism on-chip: multi-core (CPUs) and many-core processors (GPUs).
Parallelism ↔ HPC

• Parallelism is **mandatory** for high performance
  • Yesterday: clusters (and grids)
  • Today: multi-/many-core processors
  • Tomorrow: massive multi-scale heterogeneous parallelism = clusters using different types of multi-/many-cores

>95% of computing systems today are parallel!

**Main challenges:** learn to program parallel machines and learn to use them efficiently!
Why talk about GPUs?

- 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
GPGPU ?!

Massive parallelism => massive performance
Graphics in 1980
Graphics in 2015
GPUs in movies

- From Ariel in Little Mermaid to Brave
Why GPUs?

- Promise of performance beyond most other architectures
  - CPUs
  - Multi-core CPUs
  - FPGAs
  - ...
- They are power efficient
  - 2-5x better than a CPU
- What took us so long?
  - These things are not easy to program ...
GPUs vs. supercomputers

The graph compares the computational power of supercomputers and GPUs over time. The y-axis represents the number of floating operations per second (flops) ranging from 100 mflops to 100 pflops. The x-axis represents the years from 1993 to 2019.

- **Fastest supercomputer in the world** is marked with red squares and shows a steady increase in computational power.
- **nr. 500 supercomputer in the world** is marked with purple squares and also shows an increase, but at a lower rate than the fastest supercomputer.
- **1 single Graphics Processing Unit** is marked with green squares and demonstrates a dramatic increase, especially towards the end of the timeline, indicating significant advancements in GPU technology.

The graph illustrates the exponential growth in computational capabilities, with GPUs outpacing supercomputers in terms of sheer computational power by the end of the timeline.
TODO List

1. Introduction
2. GPGPU & hardware performance
3. CUDA & application performance
4. Advanced CUDA
INTRODUCTION TO GPUs

GPU = the processor
GPGPU = general purpose computing on GPUs
(typically refers to non-graphics stuff)
GPUs @ NVIDIA
GPUs @ ATI/AMD

Celebrating 30 Years of Graphics Innovation:
The Evolution of AMD Radeon GPUs
GPUs @ ATI/AMD

- **1999**: Radeon™ 9700 Pre-launch: the world’s first Radeon™ X800 Series GPU.
- **2001**: Radeon™ 8500 Launches, the first Radeon™ High Per Formance (HPU) and our first GPU with programmable pixel and vertex processors.
- **2002**: Radeon™ 9700 Pre-launch: the world’s first Radeon™ X800 Series GPU.
- **2003**: Introduction of the Radeon™ 9600 XT, the world’s first high volume 3.1bmb low-K chip.
- **2004**: CrossFire Technology first introduced.
- **2005**: Radeon HD 2000 Series launches, featuring our first unified shader product.
- **2006**: AMD acquires ATI to create a new, innovative processing powerhouse.
- **2007**: Launch of Radeon™ HD 5970, the world’s first DirectX® 11 card and the fastest graphics card in the world to date.
- **2009**: AMD increases the Radeon HD 5970 GItE Edition, the world’s fastest and most versatile graphics card.
- **2011**: Launch of Radeon™ HD 6990, the fastest discrete graphics card in the world.
- **2013**: Arrival of Mantle, a groundbreaking graphics API that promises to transform the world of game development to help bring better, faster games to the PC.
- **2013**: Launch of Radeon HD 7970, the fastest desktop graphics solution in the world, designed for enthusiasts.
- **2013**: Introduction of the Radeon R7 and R9 Series graphics cards, first to support a new era of 4K gaming.

**What’s Next?**
Find out on 08.23.14
NVIDIA vs AMD

---

Radeon HD 5000  |  Radeon HD 6000  |  Radeon HD 7000  |  Radeon R9 290

2009  |  2010  |  2011  |  2012  |  2013  |  2014

GeForce GTX 400  |  GeForce GTX 500  |  GeForce GTX 600  |  GeForce GTX 700  |  GeForce GTX 900
NVIDIA vs AMD

AMD Radeon Graphics Roadmap

<table>
<thead>
<tr>
<th>Performance</th>
<th>NVIDIA</th>
<th>AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>7200, $449</td>
<td>HD 7970 GHz Edition</td>
<td>HD 7970</td>
</tr>
<tr>
<td>6700, $399</td>
<td>HD 7970</td>
<td>GTX 680</td>
</tr>
<tr>
<td>6150, $299</td>
<td>HD 7950</td>
<td>GTX 670</td>
</tr>
<tr>
<td>5000, $229</td>
<td>HD 7870 GHz Edition</td>
<td>GTX 660 Ti</td>
</tr>
<tr>
<td>4200, $189</td>
<td>HD 7850</td>
<td>GTX 660</td>
</tr>
<tr>
<td>2750, $119</td>
<td>HD 7770 GHz Edition</td>
<td>GTX 650 Ti</td>
</tr>
<tr>
<td>2050, $99</td>
<td>HD 7750</td>
<td>GTX 650</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GT 640</td>
</tr>
</tbody>
</table>

3DMark Fire Strike
Performance, Price

http://www.4game.net
NVIDIA vs. AMD

**Radeon HD 7950**

- Clock speed: 1,050 MHz
- Shading units: 1,664
- Pixel rate: 58.8 GPixel/s
- Floating-point performance: 3,494 GFLOPS
- Price: $320

**GeForce GTX 970**

- Clock speed: 800 MHz
- Shading units: 1,792
- Pixel rate: 25.6 GPixel/s
- Floating-point performance: 2,867 GFLOPS
- Price: $311

As you can see, NVIDIA is a great choice under $311.
GPUs @ ARM

ARM Mali Graphics Processor Generations

BIFROST
- Mali-G71 GPU
- Unified shader cores, scalar ISA, clause execution, full coherency, Vulkan, OpenCL

MIDGARD
- Mali-T600 GPU series
- Mali-T700 GPU series
- Mali-T800 GPU series
- Unified shader cores, SIMD ISA, OpenGL ES 3.x, OpenCL, Vulkan

UTGARD
- Mali-200 GPU
- Mali-300 GPU
- Mali-400 GPU
- Mali-450 GPU
- Mali-470 GPU
- Separate shader cores, SIMD ISA, OpenGL ES 2.x
ON 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 \(\rightarrow\) latency boost
  - Better memory techniques
    - Use memory hierarchies \(\rightarrow\) latency boost
    - More memory closer to processor \(\rightarrow\) latency boost
  - Better processing techniques
    - Use pipelining \(\rightarrow\) throughput boost
  - More processing units (cores, threads, …)
    - Use parallelism/concurrency \(\rightarrow\) throughput boost (only?)
  - Accelerators
    - Use specialized functional units \(\rightarrow\) latency+throughput boost
Hardware Performance metrics

- Clock frequency [GHz] = absolute hardware speed
  - Memories, CPUs, interconnects

- **Operational speed [GFLOPs]**
  - Operations per second
  - *single* AND *double* precision

- **Memory bandwidth [GB/s]**
  - Memory operations per second
    - Can differ for read and write operations!
    - Differs a lot between different memories on chip

- Power [Watt]
  - The rate of consumption of energy

- Derived metrics
  - FLOP/Byte, FLOP/Watt
Theoretical peak performance

Peak = \text{chips} \times \text{cores} \times \text{vectorWidth} \times \text{FLOPs/cycle} \times \text{clockFrequency}

• Examples
  • Intel Core i7 CPU
    2 chips \times 4 \text{ cores} \times 4\text{-way vectors} \times 2 \text{ FLOPs/cycle} \times 2.4 \text{ GHz} = 154 \text{ GFLOPs}
  • NVIDIA GTX 580 GPU
    1 \text{ chip} \times 16 \text{ SMs} \times 32 \text{ cores} \times 2 \text{ FLOPs/cycle} \times 1.544 \text{ GhZ} = 1581 \text{ GFLOPs}
  • AMD HD 6970
    1 \text{ chip} \times 24 \text{ SIMD engines} \times 16 \text{ cores} \times 4\text{-way vectors} \times 2 \text{ FLOPs/cycle} \times 0.880 \text{ GhZ} = 2703 \text{ GFLOPs}
GPU vs. CPU performance

Single vs. double precision?
Main Memory bandwidth

Throughput = memory bus frequency * bits per cycle * bus width

• Memory clock != CPU clock
• In bits, divide by 8 for GB/s

• Examples:
  • Intel Core i7 DDR3: $1.333 \times 2 \times 64 = 21\ \text{GB/s}$
  • NVIDIA GTX 580 GDDR5: $1.002 \times 4 \times 384 = 192\ \text{GB/s}$
  • ATI HD 6970 GDDR5: $1.375 \times 4 \times 256 = 176\ \text{GB/s}$
Memory bandwidths

• On-chip memory can be orders of magnitude faster
  • Registers, shared memory, caches, …
  • E.g., AMD HD 7970 L1 cache achieves 2 TB/s (vs. 176GB/s for main memory)

• Other memories: depends on the interconnect
  • Intel’s technology: QPI (Quick Path Interconnect)
    • 25.6 GB/s
  • AMD’s technology: HT3 (Hyper Transport 3)
    • 19.2 GB/s
  • Accelerators: PCI-e 2.0
    • 8 GB/s
GPU vs. CPU performance

Theoretical GB/s

- GPU
  - GeForce GPU
  - Tesla-GPU
- CPU
  - CPU

Graph showing performance comparison between GPUs and CPUs over time.
Power

• Chip manufactures specify Thermal Design Power (TDP)
• We can measure dissipated power
  • Whole system
  • Typically (much) lower than TDP
• Power efficiency
  • FLOPS / Watt

• Examples (with theoretical peak and TDP)
  • Intel Core i7: \[ \frac{154}{160} = 1.0 \text{ GFLOPs/W} \]
  • NVIDIA GTX 580: \[ \frac{1581}{244} = 6.3 \text{ GFLOPs/W} \]
  • ATI HD 6970: \[ \frac{2703}{250} = 10.8 \text{ GFLOPs/W} \]
Absolute hardware performance

• Only achieved in the optimal conditions:
  • Processing units 100% used
  • All parallelism 100% exploited
  • All data transfers at maximum bandwidth

• In real life
  • No application is like this
  • Can we reason about “real” performance?
HIGH-LEVEL OPERATIONAL VIEW
A GPU Architecture
Integration into host system

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

CPU
Few complex cores
Lots of on-chip memory
Lots of control logic

GPU
many simple cores,
little memory,
little control
Why so different?

- 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
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
Kepler: SMX

- Consumer:
  - GTX680, GTX780, GTX-Titan
- HPC
  - Tesla K10..K40, K80
- 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
  - 4x more texture memory
Memory architecture (since Fermi)

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

- Shared L2 cache
Maxwell: SMM

- Consumer:
  - GTX 970, GTX 980, …
- HPC:
  - Tesla M40
- 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
Pascal: SMP

- 64 single-precision (FP32) CUDA Cores.
  - Maxwell = 128
  - Kepler = 192
- Focus on DP
- Energy efficiency
# Evolution in numbers

<table>
<thead>
<tr>
<th>GPU / Form Factor</th>
<th>Kepler GK110 / PCIe</th>
<th>Maxwell GM200 / PCIe</th>
<th>Pascal GP100 / SXM2</th>
<th>Pascal GP100 / PCIe</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMs</td>
<td>15</td>
<td>24</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>FP32 CUDA Cores / SM</td>
<td>192</td>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>FP32 CUDA Cores / GPU</td>
<td>2880</td>
<td>3072</td>
<td>3584</td>
<td>3584</td>
</tr>
<tr>
<td>FP64 CUDA Cores / SM</td>
<td>64</td>
<td>4</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>FP64 CUDA Cores / GPU</td>
<td>960</td>
<td>96</td>
<td>1792</td>
<td>1792</td>
</tr>
<tr>
<td>Base Clock</td>
<td>745 MHz</td>
<td>948 MHz</td>
<td>1328 MHz</td>
<td>1126 MHz</td>
</tr>
<tr>
<td>GPU Boost Clock</td>
<td>810/875 MHz</td>
<td>1114 MHz</td>
<td>1480 MHz</td>
<td>1303 MHz</td>
</tr>
<tr>
<td>Single precision GFLOPS</td>
<td>5040</td>
<td>6844</td>
<td>10608</td>
<td>9340</td>
</tr>
<tr>
<td>Double precision GFLOPS</td>
<td>1680</td>
<td>213</td>
<td>5304</td>
<td>4670</td>
</tr>
</tbody>
</table>
## Evolution in numbers

<table>
<thead>
<tr>
<th>GPU / Form Factor</th>
<th>Kepler</th>
<th>Maxwell</th>
<th>Pascal (SXM2)</th>
<th>Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GK110 / PCIe</td>
<td>GM200 / PCIe</td>
<td>GP100 / SXM2</td>
<td>GP100 / PCIe</td>
</tr>
<tr>
<td>Texture Units</td>
<td>240</td>
<td>192</td>
<td>224</td>
<td>224</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>384-bit GDDR5</td>
<td>384-bit GDDR5</td>
<td>4096-bit HBM2</td>
<td>3072-bit HBM2 (12GB)</td>
</tr>
<tr>
<td></td>
<td>288 GB/s</td>
<td>288 GB/s</td>
<td>732 GB/s</td>
<td>732 GB/s (16GB)</td>
</tr>
<tr>
<td>Memory Size</td>
<td>Up to 12 GB</td>
<td>Up to 24 GB</td>
<td>16 GB</td>
<td>12 GB or 16 GB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>1536 KB</td>
<td>3072 KB</td>
<td>4096 KB</td>
<td>4096 KB</td>
</tr>
<tr>
<td>Register File Size / SM</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>Register File Size / GPU</td>
<td>3840 KB</td>
<td>6144 KB</td>
<td>14336 KB</td>
<td>14336 KB</td>
</tr>
<tr>
<td>TDP</td>
<td>235 Watts</td>
<td>250 Watts</td>
<td>300 Watts</td>
<td>250 Watts</td>
</tr>
<tr>
<td>Transistors</td>
<td>7.1 billion</td>
<td>8 billion</td>
<td>15.3 billion</td>
<td>15.3 billion</td>
</tr>
<tr>
<td>GPU Die Size</td>
<td>551 mm²</td>
<td>601 mm²</td>
<td>610 mm²</td>
<td>610 mm²</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td>28-nm</td>
<td>28-nm</td>
<td>16-nm</td>
<td>16-nm</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**GPU / Form Factor**
- **Kepler**
  - GK110 / PCIe
- **Maxwell**
  - GM200 / PCIe
- **Pascal (SXM2)**
  - GP100 / SXM2
- **Pascal**
  - GP100 / PCIe

**Texture Units**
- Kepler: 240
- Maxwell: 192
- Pascal (SXM2): 224
- Pascal: 224

**Memory Interface**
- Kepler: 384-bit GDDR5
- Maxwell: 384-bit GDDR5
- Pascal (SXM2): 4096-bit HBM2
- Pascal 3072-bit HBM2 (12GB) 4096-bit HBM2 (16GB)

**Memory Bandwidth**
- Kepler: 288 GB/s
- Maxwell: 288 GB/s
- Pascal (SXM2): 732 GB/s
- Pascal: 549 GB/s (12GB) 732 GB/s (16GB)

**Memory Size**
- Kepler: Up to 12 GB
- Maxwell: Up to 24 GB
- Pascal (SXM2): 16 GB
- Pascal: 12 GB or 16 GB

**L2 Cache Size**
- Kepler: 1536 KB
- Maxwell: 3072 KB
- Pascal (SXM2): 4096 KB
- Pascal: 4096 KB

**Register File Size / SM**
- Kepler: 256 KB
- Maxwell: 256 KB
- Pascal (SXM2): 256 KB
- Pascal: 256 KB

**Register File Size / GPU**
- Kepler: 3840 KB
- Maxwell: 6144 KB
- Pascal (SXM2): 14336 KB
- Pascal: 14336 KB

**TDP**
- Kepler: 235 Watts
- Maxwell: 250 Watts
- Pascal (SXM2): 300 Watts
- Pascal: 250 Watts

**Transistors**
- Kepler: 7.1 billion
- Maxwell: 8 billion
- Pascal (SXM2): 15.3 billion
- Pascal: 15.3 billion

**GPU Die Size**
- Kepler: 551 mm²
- Maxwell: 601 mm²
- Pascal (SXM2): 610 mm²
- Pascal: 610 mm²

**Manufacturing Process**
- Kepler: 28-nm
- Maxwell: 28-nm
- Pascal (SXM2): 16-nm
- Pascal: 16-nm
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

Kernel = the parallel program
Device code = manage the parallel program
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
Hierarchy of threads
Grids, Thread Blocks and Threads

Grid

Thread Block 0, 0

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

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

\[
\text{myKernel} \lll \text{numBlocks}, \text{threadsPerBlock} \rrr (\ldots); \\
\]

- \(\text{dim3} \ \text{threadsPerBlock}(3,4)\);
  - \(\text{threadsPerBlock}.x = 3\)
  - \(\text{threadsPerBlock}.y = 4\)
  - Each thread: \((\text{threadIdx}.x, \text{threadIdx}.y)\)

- \(\text{dim3} \ \text{numBlocks}(2,3)\);
  - \(\text{blockDim}.x = 2\)
  - \(\text{blockDim}.y = 3\)
  - Each block: \((\text{blockIdx}.x, \text{blockIdx}.y)\)
Multiple Device Memory Scopes

- **Per-thread private memory**
  - Each thread has its own local memory
  - Stacks, other private data, **registers**
  - Accessible to a single thread only

- **Per-SM shared memory**
  - Small memory close to the processor, low latency
  - Accessible to threads in the same block.

- **Device memory**
  - GPU frame buffer
  - Accessible to any thread
Memory spaces: Registers

Example:

```c
__global__ void aKernel(float *C, float *A, float *B) {
    int tx = threadIdx.x; // local variable in registers
    float local_sum[4]; // small compile-time sized array in registers
}
```

Registers:

- Thread-local scalars or small constant size arrays are stored as registers
- Implicit in the programming model
- Behavior is very similar with local variables
- Not persistent: kernel ends, data is lost
Memory spaces: global memory

Example:
```
__global__ void matmul_kernel(float *C, //C points to global memory
    float *A, //A points to global memory
    float *B) //B points to global memory
```

Global memory

- **Allocated by the host program using** `cudaMalloc()`
- **Initialized by the host program using** `cudaMemcpy()` or previous kernels
- Persistent = the values are retained between kernels
- Not coherent, writes by other threads might not be visible until kernel has finished
Memory spaces: Constant

Example

```c
__constant__ float speed_of_light = 0.299792458; // scalars can be initialized directly
__constant__ float2 vertices[NUM_VERTICES]; // initialized by a host function
__global__ void cn_pnpoly(uint8_t* bitmap, float2* points, int n) {
...
for (int j = 0; j < NUM_VERTICES; k = j++) {
    float2 vj = vertices[j]; // index j does not depend on threadIdx
    ...
}
```

Constant memory:

- **Statically defined by the host program using `__constant__` qualifier**
- Defined as a global variable, visible only within the same translation unit
- **Initialized by the host program using `cudaMemcpyToSymbol()`**
- Read-only to the GPU, cannot be accessed directly by the host
- Values are cached in a special cache optimized for broadcast access by multiple threads simultaneously, access should not depend on threadIdx
Memory spaces: Shared

Example:

```c
__global__ void matmul_kernel(float *C, float *A, float *B) {
    __shared__ float sh_A[.tile_size][tile_size]; //2D array in shared memory
    for (k = 0; k < WIDTH; k += .tile_size) {
        __syncthreads(); //wait for all threads in the block
        sA[ty][tx] = A[y*WIDTH + k + tx]; //fill shared memory with values
        __syncthreads(); //wait again
    }
}
```

Shared memory

- Variables have to be declared using __shared__ qualifier, size known at compile time
- In the scope of thread block, all threads in a thread block see the same piece of memory
- Not initialized, threads have to fill shared memory with meaningful values
- Not persistent, after the kernel has finished, values in shared memory are lost
- Not coherent, __syncthreads() is required to make writes visible to other threads within the thread block
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
Processing flow

All this happens from the host code.

Image courtesy of Wikipedia
Compiling CUDA

- `nvcc` is a compiler driver
- Separates source code into:
  - device code (runs on GPU)
    - further processed by NVIDIA compiler
  - host code (runs on CPU)
    - further processed by host compiler

![Diagram of CUDA compilation process]

- C/C++ CUDA source code
- nvcc
- PTX code
  - ptx
  - G80, G200, ...
  - GPU code
- CPU code
CUDA: kernels and launch

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

• Execution configuration:
  ```c
  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:
  ```c
  dim3 gridDim; // Grid dimension
  dim3 blockDim; // Block dimension
  dim3 blockIdx; // Block index
  dim3 threadIdx; // Thread index
  ```

  ```c
  void syncthreads(); // Thread synchronization
  ```
CUDA: Memory Allocation/Release

- All memory buffers – CPU and GPU must be allocated
- Host (CPU) manages device (GPU) memory:
  - `cudaMalloc(void **pointer, size_t nbytes)`
  - `cudaMemset(void *pointer, int val, size_t count)`
  - `cudaFree(void* pointer)`
CUDA: Data Copies

cudaMemcpy(void *dst, void *src,
    size_t nbytes,
    enum cudaMemcpyKind direction);

• blocks CPU thread until all bytes have been copied
• doesn’t start copying until previous CUDA calls complete

• enum {
    cudaMemcpyHostToDevice,
    cudaMemcpyDeviceToHost,
    cudaMemcpyDeviceToDevice
} cudaMemcpyKind

• Non-blocking copies are also available
  • cudaMemcpyAsync
  • DMA transfers, overlap computation and communication
CUDA: dummy example

```c
int n = 1024;
int nbytes = n * sizeof(int);
int* dataCPU = (int *)malloc(nbytes);
int* dataGPU;

cudaMalloc(&dataGPU, nbytes);
cudaMemset(dataGPU, 0, nbytes);

cudaMemcpy(dataGPU, dataCPU, nbytes, cudaMemcpyHostToDevice);
myKernel<<<n/128,128>>>(n, dataGPU);
cudaMemcpy(dataCPU, dataGPU, nbytes, cudaMemcpyDeviceToHost);
cudaFree(dataGPU);
free(dataCPU);
```
EXAMPLE: VECTOR-ADD
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
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!
Vector add: Kernel

// 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:

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

\[
4 \times 2 + 1 = 9
\]
Vector add: Kernel

// 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];
}
Vector add: Launch kernel

```c
// 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];
}
```

```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 ...
}
```

(in the same file)
Vector add: Launch kernel

// 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];
}

int main() {
    // initialization code here ...
    N = 5000;  // <- what happens?
    // launch N/256 blocks of 256 threads each
    vector_add<<< N/256, 256 >>>(deviceA, deviceB, deviceC);
    // cleanup code here ...
}
Vector add: Launch kernel

// 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;  // <- what happens?
    // launch N/256 blocks of 256 threads each
    vector_add<<<N/256+1, 256 >>>(deviceA, deviceB, deviceC);
    // cleanup code here ... 
}

(in the same file)
Vector add: Host

```c
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);
}```
Vector add: Host

// 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);
}
ADVANCED CONCEPTS
Thread Scheduling

- Order of threads within a block is undefined!
  - Threads are grouped in warps (32 threads/warp)
    - AMD calls it “a wavefront” (64 threads/wavefront)
- Order in which thread blocks are mapped and scheduled is undefined!
  - Blocks run to completion on one SM without preemption
  - Can run in any order
    - Any possible interleaving of blocks should be valid
  - Can run concurrently OR sequentially
Global synchronization

- We launch many more blocks than physical SM’s.
- Each block might/should have more threads than the SM’s cores

```c
__global__ void my_kernel() {
    step1; // compute some values in a global array
    // wait for *all* threads to finish
    __my_global_barrier();
    step2; // use the array
}

int main() {
    dim3 blockSize(32, 32);
    dim3 gridSize(100, 100, 100);
    my_kernel<<<gridDim, blockDim>>>();
}
```
An example: parallel reduction

- Given an array with data, “reduce” it to a single value
  - The sum of all elements
  - The min/max of all elements
- Sequentially: $O(n)$
- In parallel?
  - Tree-based algo.
    - $O(\log n)$
  - Requires a barrier after each step
Parallel reduction in CUDA*

- One element per thread
- We need to use multiple blocks
  - Large arrays
  - Good GPU utilization
- We need global synchronization
  - Synchronization inside blocks is possible.
  - Synchronization between blocks is not possible!
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

Parallel reduction in CUDA*

• Other optimizations
  • Use shared memory
  • Increase granularity
  • Avoid branching
  • Improve data access patterns

Memory consistency

• Device (global) memory is not serially consistent
  • No ordering guarantees in shared/global memory Rd/Wr
• Share data between streaming multiprocessors
  • Potential write hazards!
• Use **atomics** to avoid data races for global (and shared) memory variables!

• Evolution:
  • Fermi has reasonable atomics for both shared and global memory
  • Kepler increases *global memory atomics* performance vs. Fermi
  • Maxwell uses native support for shared memory atomics
    • Much faster than Fermi and Kepler
Atomics

- Guarantee that only a single thread has access to a piece of memory during an operation
  - Ordering is still arbitrary
- Different types of atomic instructions
  - Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
- Both for device memory and shared memory
- Much more expensive than load + operation + store
An example: image histogram

- The histogram of an image: the distribution of the pixels in the image.
  - In practice: count the pixels of each color
  - Useful image feature detection for image recognition.
An example: image histogram

```c
// 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;  // incorrect!
}
```
An example: image 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: OCCUPANCY
Thread Scheduling

• Order of threads within a block is undefined!
  • Threads are grouped in warps (32 threads/warp)
    • AMD calls it “a wavefront” (64 threads/wavefront)

• Order in which thread blocks are mapped and scheduled is undefined!
  • Blocks run to completion on one SM without preemption
  • Can run in any order
    • Any possible interleaving of blocks should be valid
  • Can run concurrently OR sequentially
Warps = 32 threads

- Threads are scheduled in warps
  - AMD calls them “wavefronts”

- One warp => on one SM
  - Same SM till completion

- Scheduling
  - GigaThread Unit : schedules blocks per SM’s
  - Inside SM: warp scheduler(s) + instruction dispatcher
    - Replace warps that are stalled by warps waiting to compute
    - Very fast context switching
Thread Scheduling

- 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:

\[
\begin{array}{ccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 1 & 2 & 1 & 2 & 3 & 4 & 7 & 8 & 1 & 2 & 1 & 2 & 3 & 4
\end{array}
\]

Time

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

Occupancy = Active Warps / Maximum Active Warps

• Remember: resources are allocated for the entire block!
• Resources are finite
  • Utilizing too many resources per thread may limit the occupancy
• Potential occupancy limiters:
  • Register usage
  • Shared memory usage
  • Block size
• 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
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!
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.
Thread divergence - 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: THREAD DIVERGENCE
Thread divergence

“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

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')

```plaintext
<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: MEMORY COALESCING
Memory Coalescing

- Memory coalescing refers to combining multiple memory accesses into a single transaction.
Caching vs. Coalescing

traditional multi-core
optimal memory access pattern

thread 0

$t = 0$ -> address 0
$t = 1$ -> address 1

thread 1

$t = 0$ -> address 2
$t = 1$ -> address 3

thread 2

$t = 0$ -> address 4
$t = 1$ -> address 5

thread 3

$t = 0$ -> address 6
$t = 1$ -> address 7

Caching
Caching vs. Coalescing

Caching

thread 0
- $t = 0$ → address 0
- $t = 1$ → address 1

thread 1
- $t = 0$ → address 2
- $t = 1$ → address 3

thread 2
- $t = 0$ → address 4
- $t = 1$ → address 5

thread 3
- $t = 0$ → address 6
- $t = 1$ → address 7

Coalescing

thread 0
- $t = 0$ → address 0
- $t = 1$ → address 1

thread 1
- $t = 0$ → address 2
- $t = 1$ → address 3

thread 2
- $t = 0$ → address 4
- $t = 1$ → address 5

thread 3
- $t = 0$ → address 6
- $t = 1$ → address 7

optimal memory access pattern

traditional multi-core

many-core GPU
Consider the stride of your accesses

```c
__global__ void foo(int* input, float3* input2) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  // Stride 1, full bandwidth used!
  int a = input[i];
  // Stride 2, 50% of the bandwidth is wasted
  int b = input[2*i+1];
  // "Random" stride - ?? up to 7/8 bandwidth wasted
  int c = input[f(i)];
}
```
Example: Array of Structures (AoS)

Struct AoS{
    int key;
    int value;
    int flag;
};

record *d_AoS_data;
cudaMalloc((void**) &d_AoS_data, ...);

kernel {
    threadID = blockDim.x * blockIdx.x + threadIdx.x;
    // ...
    d_AoS_data[threadID].value += i; // wastes bandwidth!
    // ...
}
Example: Structure of Arrays (SoA)

```c
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, ...);

kernel {
    threadID = blockDim.x * blockIdx.x + threadIdx.x;
...
    d_SoA_data.values[threadID] += i; // full bandwidth!
... }
```
Memory Coalescing*

- Group memory accesses in as few memory transactions as possible.
  - 128-byte or 32-byte long lines
- Stride 1 access patterns are preferred!
  - Other patterns can still get benefits
- Structure of arrays is often better than array of structures
- Unpredictable/irregular access patterns
  - Case-by-case performance impact
- No coalescing => performance loss ~10x or more!
  - Caching might improve this impact …

*http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz3nJ0kBsWe
CUDA: MEMORY COALEScing
Memory Coalescing

- Memory coalescing refers to combining multiple memory accesses into a single transaction.
Caching vs. Coalescing

traditional multi-core
optimal memory access pattern

Caching
Caching vs. Coalescing

Caching vs. Coalescing

- **Caching:**
  - **traditional multi-core**
  - **optimal memory access pattern**

- **Thread 0**:
  - $t = 0$: address 0
  - $t = 1$: address 1

- **Thread 1**:
  - $t = 0$: address 2
  - $t = 1$: address 3

- **Thread 2**:
  - $t = 0$: address 4
  - $t = 1$: address 5

- **Thread 3**:
  - $t = 0$: address 6
  - $t = 1$: address 7

- **Coalescing:**
  - **many-core GPU**
  - **optimal memory access pattern**

- **Thread 0**:
  - $t = 0$: address 0
  - $t = 1$: address 1

- **Thread 1**:
  - $t = 0$: address 2
  - $t = 1$: address 3

- **Thread 2**:
  - $t = 0$: address 4
  - $t = 1$: address 5

- **Thread 3**:
  - $t = 0$: address 6
  - $t = 1$: address 7
Consider the stride of your accesses

<table>
<thead>
<tr>
<th>Stride1</th>
<th>Stride2</th>
<th>“random”</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>T0</td>
<td>T1</td>
<td>T1</td>
</tr>
</tbody>
</table>

```c
__global__ void foo(int* input, float3* input2) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    // Stride 1, full bandwidth used!
    int a = input[i];
    // Stride 2, 50% of the bandwidth is wasted
    int b = input[2*i+1];
    // “Random” stride – ?? up to 7/8 bandwidth wasted
    int c = input[f(i)];
}
```
Example: Array of Structures (AoS)

Struct AoS{
    int key;
    int value;
    int flag;
};

record *d_AoS_data;
cudaMalloc((void**)&d_AoS_data, ...);

kernel {
    threadID = blockDim.x * blockIdx.x + threadIdx.x;
    // ...
    d_AoS_data[threadID].value += i; // wastes bandwidth!
    // ...
}
Example: Structure of Arrays (SoA)

```c
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, ...);

kernel {
    threadID = blockDim.x * blockIdx.x + threadIdx.x;
    ...
    d_SoA_data.values[threadID] += i; // full bandwidth!
    ...
}
```
Memory Coalescing*

- Group memory accesses in as few memory transactions as possible.
  - 128-byte or 32-byte long lines
- Stride 1 access patterns are preferred!
  - Other patterns can still get benefits
- Structure of arrays is often better than array of structures
- Unpredictable/irregular access patterns
  - Case-by-case performance impact
- No coalescing => performance loss 10 – 30x!

*http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz3nJ0kB
CUDA: USING SHARED MEMORY
Using shared memory

• Equivalent with providing software caching
  • Explicit: Load data to be re-used in shared memory
  • Use it for computation
  • Explicit: Store results back to global memory

• All threads in a block share memory
  • Load/Store: using all threads
  • Barrier: __syncthreads
    • Guard against using uninitialized data – not all threads have finished loading data to shared memory
    • Guard against corrupting live data – not all threads have finished computing
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
A Common Programming Strategy

- 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
A Common Programming Strategy

• Copy the result from shared memory back to device memory
Caches vs. Shared Memory

- Since Fermi, NVIDIA GPUs feature BOTH hardware **L1 caches** and **shared memory** per SM
  - They share the same space
    - \( \frac{3}{4} \) Cache + \( \frac{1}{4} \) Shared Memory OR
    - \( \frac{1}{4} \) Cache + \( \frac{3}{4} \) Shared Memory

- **L1 Cache**
  - Hardware caching enabled
    - The HW decides what goes in or out and when

- **Shared memory**
  - Software manages what goes in/out
  - Allows more complex access patterns to be cached
Example: Matrix multiplication

- \( C = A \times B \)
  - \( C(i,j) = \text{sum}(\text{dot(row}(A,i),\text{col}(B,j))) \)

- Parallelization strategy
  - Each thread computes one \( C \) element
  - 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

IDEA:
- Load elements into shared memory
- Have several threads use local version to improve 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 from A,B
    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, p = crt. phase
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];
    // barrier: ALL writes to shared memory finished
    __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];
    }
    // barrier: ALL reads of shared memory finished
    __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

• Formally, __synchthreads is a barrier for shared memory for a block of threads:

  “void __synchthreads();
  waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to __synchthreads() are visible to all threads in the block.”
## 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

- 1581 GFLOPs

### Memory bandwidth GTX 580

- 192 GB/s

### AI needed for peak

- $1581 / 192 = 8.23$

### TILE_WIDTH required to achieve peak

- $0.25 \times \text{TILE WIDTH} = 8.23$, 
  $\text{TILE WIDTH} = 32.9$
CUDA: STREAMS
Overlap Computation and Communication

- Main idea: while executing a kernel, bring data in for the next kernel:
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.
Default stream: example

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);
kernell<<blocks,threads>>>(d_a, firstElement);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);

C1060 (pre-Fermi): 12.9ms

C2050 (Fermi): 9.9ms

Sequential Version
Computation-communication overlap[1]*

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)
Computation-communication overlap[2]*

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)

C2050 (Fermi): 7.59 ms (better than sequential, worse than v1)
CUDA: MANY OTHER FEATURES
Kepler: Dynamic parallelism
Kepler: Hyper-Q

Fermi:
1 MPI* Task at a Time

NVIDIA Hyper-Q

Kepler:
32 Simultaneous MPI Tasks
Kepler GK110: GPU Direct
Unified memory
SUMMARY
Take home message

- GPUs are massively parallel architectures with limited flexibility, but very high throughput
- Pro’s:
  - Much higher compute capabilities
  - Higher bandwidth
- Con’s
  - Limited on-card memory
  - Low-bandwidth communication with host
- Debate-able
  - Programmability & productivity
Open research questions

• Shall we port all applications on GPUs?
  • If yes – can we automate the process?
  • If not – can we decide how to select?

• Shall we use GPUs in large-scale systems?

• Shall we use heterogeneous CPU+GPU systems?

• Can we improve the GPU design …
  • For HPC?
  • For other application domains?
Questions? Comments? Suggestions?

• A.L.Varbanescu@uva.nl

… also if you want to work on GPU-related projects OR in a team that works on heterogeneous computing.

… All you have to do is ask 😊