What we’ve covered so far…

- What: overview of issues that will come up later on
  - From C code to machine instructions
  - ISA: RISC vs. CISC architectures
  - Microarchitecture issues
  - The transistor, and its role in enabling logical & math operations
  - Registers
  - Pipelining
  - Memory aspects: caches and the virtual memory space
  - Three walls to sequential computing
Before we get started

- Use the link below to register and get an account of the Euler cluster supercomputer

https://docs.google.com/forms/d/1bTD8qKDcnuW0INlgoC63xV4EMRR_C9ecK1gsEDurpqo/viewform?fbzx=-1500101795554835256

- Slides available at

http://outreach.sbel.wisc.edu/Workshops/GPUworkshop/
From Simple to Complex: Part 1

- The von Neumann architecture
  - Red arrow: instructions
  - Green arrow: data
From Simple to Complex: Part 2

- The architecture of the early to mid 1990s
  - Pipelining was king
From Simple to Complex: Part 3

- The architecture of late 1990s, early 2000s
  - ILP galore
Conventional Wisdom in Computer Architecture

- Old: Power is free, Transistors expensive
- New: Power expensive, Transistors free
  (lots of power means heat to be dissipated and short battery lifespan)

- Old: Multiplies are slow, Memory access is fast
- New: Memory slow, multiplies fast [“Memory wall”]
  (400-600 cycles for DRAM memory access, 1 clock for FMA)

- Old: Increasing Instruction Level Parallelism via compilers, innovation (Out-of-order, speculation, VLIW, …)
- New: “ILP wall” diminishing returns on more ILP

- New: Power Wall + Memory Wall + ILP Wall = Brick Wall
  - Old: Uniprocessor performance 2X / 1.5 yrs
  - New: Uniprocessor performance only 2X / 5 yrs?
Old School

- Increasing clock frequency is primary method of performance improvement
- Don’t bother parallelizing an application, just wait and run on much faster sequential computer
- Less than linear scaling for a multiprocessor is failure

New School

- Processors parallelism is primary method of performance improvement
- Nobody is building one processor per chip. This marks the end of the “raised by the tide” era
- Given the switch to parallel hardware, even sub-linear speedups are beneficial as long as you beat the sequential on a watt-to-watt basis
Summarizing It All…

- The sequential execution model is losing steam
- The bright spot: number of transistors per unit area going up and up
● A bunch of bad news…

● … with only one bright spot
Moore’s Law

- 1965 paper: Doubling of the number of transistors on integrated circuits every two years
  - Moore himself wrote only about the density of components (or transistors) at minimum cost

- Increase in transistor count is also a rough measure of computer processing performance

Moore’s Law (1965)

• “The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (see graph on next page). Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.”

“Cramming more components onto integrated circuits” by Gordon E. Moore, Electronics, Volume 38, Number 8, April 19, 1965
Intel’s Vision: Evolutionary Configurable Architecture

**Large, Scalar cores for high single-thread performance**

**Scalar plus many core for highly threaded workloads**

**Multi-core array**
- CMP with ~10 cores

**Many-core array**
- CMP with 10s-100s low power cores
- Scalar cores
- Capable of TFLOPS+
- Full System-on-Chip
- Servers, workstations, embedded...

**Dual core**
- Symmetric multithreading

CMP = “chip multi-processor”

Parallel Computing: Some Black Spots

- More transistors = More computational units

- November 2013:
  - Intel Xeon w/ 12 cores – 3 billion transistors

- Projecting ahead, this would seem feasible:
  - 2015: 24 cores
  - 2017: about 50 cores
  - 2019: about 100 cores
  - 2021: about 200 cores

- Black silicon: because of high density and power leaks, we might not be able to power these chips…
  - Black silicon: transistors that today don’t get used and are dead weight
  - Dennard’s scaling started to break down at the end of last decade
  - Dennard’s law is the secret sauce for Moore’s law
Getting into specifics...
Two Examples of Parallel HW

- Intel Haswell
  - Multicore architecture

- NVIDIA Fermi
  - Large number of scalar processors ("shaders")
Intel Haswell

- June 2013
- 22 nm technology
- 1.4 billion transistors
- 4 cores, hyperthreaded
- Integrated GPU
- System-on-a-chip (SoC) design
Fermi: 30,000 Feet Perspective

- Lots of ALU (green), not much of CU (orange)
- Explains why GPUs are fast for high arithmetic intensity applications
- Arithmetic intensity: high when many operations performed per word of memory
The Fermi Architecture

- Late 2009, early 2010
- 40 nm technology
- Three billion transistors
- 512 Scalar Processors (SP, “shaders”)
- L1 cache
- L2 cache
- 6 GB of global memory
- Operates at low clock rate
- High bandwidth (close to 200 GB/s)
Why is the GPU Fast?

- **The GPU is specialized** for compute-intensive, highly data parallel computation (owing to its graphics rendering origin)
  - More transistors devoted to **data processing** rather than data caching and control flow
  - Where are GPUs good: **high arithmetic intensity** (the ratio between arithmetic operations and memory operations)

![Diagram of CPU and GPU architecture]

- The large video game industry exerts strong **economic pressure** that forces constant innovation in GPU computing
# Key Parameters

## GPU, CPU

<table>
<thead>
<tr>
<th></th>
<th>GPU – NVIDIA Tesla C2050</th>
<th>CPU – Intel core i7 975 Extreme</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing Cores</strong></td>
<td>448</td>
<td>4 (8 threads)</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>48 KB L1/SM</td>
<td>- 32 KB L1 cache / core</td>
</tr>
<tr>
<td></td>
<td>768 KB (all SMs)</td>
<td>- 256 KB L2 (I&amp;D) cache / core</td>
</tr>
<tr>
<td></td>
<td>3 GB (global mem)</td>
<td>- 8 MB L3 (I&amp;D) shared by all cores</td>
</tr>
<tr>
<td><strong>Clock speed</strong></td>
<td>1.15 GHz</td>
<td>3.20 GHz</td>
</tr>
<tr>
<td><strong>Memory bandwidth</strong></td>
<td>140 GB/s</td>
<td>25.6 GB/s</td>
</tr>
<tr>
<td><strong>Floating point operations/s</strong></td>
<td>$515 \times 10^9$ Double Precision</td>
<td>$70 \times 10^9$ Double Precision</td>
</tr>
</tbody>
</table>
“Big Iron” Parallel Computing
Euler: CPU/GPU Heterogeneous Cluster
~ Hardware Configuration ~

Legend, Connection Type:
- Gigabit Ethernet
- 4x QDR Infiniband

File Server Architecture
- CPU Intel Xeon 5620
- RAM 16 GB DDR3
- Infiniband HCA
- RAID 6
- 24x 2TB Hard Disks

CPU/GPU Node Architecture
- CPU 0
  - Intel Xeon 5520
  - Hard Disk
- Infiniband HCA
  - GPU 0
  - GPU 1
  - GPU 2
  - GPU 3
  - GTX480
  - 1.5GB RAM
  - 448 Cores
  - PCIeX16 2.0

AMD Node Architecture
- CPU 0
  - AMD Opteron 6276
- CPU 1
  - AMD Opteron 6276
- CPU 2
  - AMD Opteron 6276
- CPU 3
  - AMD Opteron 6276
- RAM 128 GB DDR3
- Infiniband HCA
- SSD

Internal Users
- Gigabit Ethernet Switch
- Head Node
- 4x QDR Infiniband Switch
- File Server
- CPU/GPU Node 1
- CPU/GPU Node 2
- CPU/GPU Node 14
- AMD Node 1
- Remote Collaborators
- Internet
Background: Lab’s Research Cluster

EULER - Heterogeneous Research Cluster.
Overview of Large Multiprocessor Hardware Configurations (“Big Iron”)

- Larger multiprocessors
  - Shared address space
    - Symmetric shared memory (SMP)
      - Examples: IBM eserver, SUN Sunfire
    - Distributed shared memory (DSM)
  - Distributed address space
    - Commodity clusters: Beowulf and others
    - Custom cluster
      - Cache coherent: ccNUMA: SGI Origin/Altix
        - Non-cache coherent: Cray T3E, X1
      - Uniform cluster: IBM BlueGene
        - Constellation cluster of DSMs or SMPs
          - SGI Altix, ASC Purple

© 2007 Elsevier, Inc. All rights reserved.

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Some Nomenclature…

- **Shared addressed space:** when you invoke address “0x0043fc6f” on one machine and then invoke “0x0043fc6f” on a different machine they actually point to the same global memory space
  - Issues: memory coherence
    - Fix: software-based or hardware-based

- **Distributed addressed space:** the opposite of the above

- **Symmetric Multiprocessor (SMP):** you have one machine that shares amongst all its processing units a certain amount of memory (same address space)
  - Mechanisms should be in place to prevent data hazards (RAW, WAR, WAW). Brings back the issue of memory coherence

- **Distributed shared memory (DSM) — aka distributed global address space (DGAS):**
  - Although physically memory is distributed, it shows as one uniform memory
  - Memory latency is highly unpredictable
Example

- Distributed-memory multiprocessor architecture (Euler, for instance)
Comments, distributed-memory multiprocessor architecture

- Basic architecture consists of nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all nodes.

- Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology, which is less scalable than the global interconnection network.

- Popular interconnection network: Mellanox and Qlogic InfiniBand
  - Bandwidth range: 1 through 50 Gb/sec (about 6 GB/s)
  - Latency: in the microsecond range (approx. 1E-6 seconds)
  - Requires special network cards: HCA – “Host Channel Adaptor”
Example, SMP
[This is not “Big Iron”, rather a desktop nowadays]

- Shared-Memory Multiprocessor Architecture

![Diagram of Shared-Memory Multiprocessor Architecture]

Usually SRAM

Usually DRAM

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Comments, SMP Architecture

- Multiple processor-cache subsystems share the same physical off-chip memory

- Typically connected to this off-chip memory by one or more buses or a switch

- Key architectural property: uniform memory access (UMA) time to all of memory from all the processors
  - This is why it’s called symmetric
Examples…

- **Shared-Memory**
  - Intel Xeon Phi available as of 2012
    - Packs 61 cores, which are on the basic (unsophisticated) side
  - AMD Opteron 6200 Series (16 cores: Opteron 6276) – Bulldozer architecture
  - Sun Niagara

- **Distributed-Memory**
  - IBM BlueGene/L
  - Cell (see [http://users.ece.utexas.edu/~adnan/vlsi-07/hofstee-cell.ppt](http://users.ece.utexas.edu/~adnan/vlsi-07/hofstee-cell.ppt))
### Big Iron: Where Are We Today?

[Info lifted from Top500 website: http://www.top500.org/]

<table>
<thead>
<tr>
<th>Name</th>
<th>Specs</th>
<th>Site</th>
<th>Country</th>
<th>Cores</th>
<th>R_max (Pflops)</th>
<th>Power (MW)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Tianhe-2 (Milkyway-2)</strong></td>
<td>NUDT, Intel Ivy Bridge (12C, 2.2 GHz) &amp; Xeon Phi (57C, 1.1 GHz), Custom interconnect</td>
<td>NSCC Guangzhou</td>
<td>China</td>
<td>3,120,000</td>
<td>33.9</td>
<td>17.8</td>
</tr>
<tr>
<td><strong>2. Titan</strong></td>
<td>Cray XK7, Operon 6274 (16C 2.2 GHz) + Nvidia Kepler GPU, Custom interconnect</td>
<td>DOE/SC/ORNL</td>
<td>USA</td>
<td>560,640</td>
<td>12.6</td>
<td>8.2</td>
</tr>
<tr>
<td><strong>3. Sequoia</strong></td>
<td>IBM BlueGene/Q, Power BGC (16C 1.60 GHz), Custom interconnect</td>
<td>DOE/NNSA/LLNL</td>
<td>USA</td>
<td>1,572,864</td>
<td>12.2</td>
<td>7.9</td>
</tr>
<tr>
<td><strong>4. K computer</strong></td>
<td>Fujitsu SPARC64 VIIIfx (8C, 2.0GHz), Custom interconnect</td>
<td>RIKEN AICS</td>
<td>Japan</td>
<td>706,024</td>
<td>10.5</td>
<td>12.7</td>
</tr>
<tr>
<td><strong>5. Mira</strong></td>
<td>IBM BlueGene/Q, Power BGC (16C, 1.60 GHz), Custom interconnect</td>
<td>DOE/SC/ANL</td>
<td>USA</td>
<td>786,432</td>
<td>8.59</td>
<td>3.95</td>
</tr>
</tbody>
</table>

**Performance Development**

- **Sum N=1**: 2.74 Pflops
- **Sum N=500**: 33.9 Pflops
- **Projected**: 124 Pflops
Abbreviations/Nomenclature

- **MPP** – Massively Parallel Processing
- **Constellation** – subclass of cluster architecture envisioned to capitalize on data locality
- **MIPS** – “Microprocessor without Interlocked Pipeline Stages”, a chip design of the MIPS Computer Systems of Sunnyvale, California
- **SPARC** – “Scalable Processor Architecture” is a RISC instruction set architecture developed by Sun Microsystems (now Oracle) and introduced in mid-1987
- **Alpha** - a 64-bit reduced instruction set computer (RISC) instruction set architecture developed by DEC (Digital Equipment Corporation was sold to Compaq, which was sold to HP) – adopted by Chinese chip manufacturer (see primer)
Short Digression: Massively Parallel Processing

What is a MPP?

- A very large-scale computing system with commodity processing nodes interconnected with a high-speed low-latency interconnect
- Memories are physically distributed
- Nodes often run a microkernel
- Contains one host monolithic OS
- Rather blurred line between MPPs and clusters
How is the speed measured to put together the Top500?
- Basically reports how fast you can solve a dense linear system
Flynn’s Taxonomy of Architectures

- There are several ways to classify architectures (we just saw on based on how memory is organized/accessed)

- Below, classified based on how instructions are executed in relation to data

  - SISD - Single Instruction/Single Data
  - SIMD - Single Instruction/Multiple Data
  - MISD - Multiple Instruction/Single Data
  - MIMD - Multiple Instruction/Multiple Data
Single Instruction/Single Data Architectures

Your desktop, before the spread of dual core CPUs

Flavors of SISD

Instructions:

Pipelining

Instruction-Level Parallelism (ILP)
Single Instruction/Multiple Data Architectures

Processors that execute same instruction on multiple pieces of data: NVIDIA GPUs

Single Instruction/Multiple Data [Cntd.]

- Each core runs the same set of instructions on different data
- Examples:
  - Graphics Processing Unit (GPU): processes pixels of an image in parallel
  - CRAY’s vector processor, see image below

Slide Source: Klimovitski & Macri, Intel
SISD versus SIMD

Writing a compiler for SIMD architectures is difficult (inter-thread communication complicates the picture...)

Slide Source: ars technica, Peakstream article
Multiple Instruction/Single Data

Not useful, not aware of any commercial implementation...

Multiple Instruction/Multiple Data

Almost all our desktop/laptop chips are MIMD systems

Multiple Instruction/Multiple Data

- The sky is the limit: each PU is free to do as it pleases
- Can be of either shared memory or distributed memory categories

Time

Instructions:

Thread-Level Parallelism (TLP)
Amdahl's Law


“A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude”

- Let $r_s$ capture the amount of time that a program spends in components that can only be run sequentially
- Let $r_p$ capture the amount of time spent in those parts of the code that can be parallelized.
- Assume that $r_s$ and $r_p$ are normalized, so that $r_s + r_p = 1$
- Let $n$ be the number of threads used to parallelize the part of the program that can be executed in parallel
- The “best case scenario” speedup $S$ is

$$S = \frac{T_{old}}{T_{new}} = \frac{r_s + \frac{r_p}{n}}{r_s + \frac{r_p}{n}} = \frac{1}{r_s + \frac{r_p}{n}}$$
Amdahl’s Law

[Cntd.]

- Sometimes called the law of diminishing returns

- In the context of parallel computing used to illustrate how going parallel with a part of your code is going to lead to overall speedups

- The art is to find for the same problem an algorithm that has a large $r_p$
  - Sometimes requires a completely different angle of approach for a solution

- Nomenclature
  - Algorithms for which $r_p=1$ are called “embarrassingly parallel”
Example: Amdahl's Law

- Suppose that a program spends 60% of its time in I/O operations, pre and post-processing.
- The rest of 40% is spent on computation, most of which can be parallelized.
- Assume that you buy a multicore chip and can throw 6 parallel threads at this problem. What is the maximum amount of speedup that you can expect given this investment?
- Asymptotically, what is the maximum speedup that you can ever hope for?
A Word on “Scaling”
[important to understand]

- **Algorithmic Scaling** of a solution algorithm
  - You only have a mathematical solution algorithm at this point
  - Refers to how the effort required by the solution algorithm scales with the size of the problem
  - Examples:
    - Naïve implementation of the N-body problem scales like $O(N^2)$, where $N$ is the number of bodies
    - Sophisticated algorithms scale like $O(N \log N)$
    - Gauss elimination scales like the cube of the number of unknowns in your linear system

- **Implementation Scaling** of a solution on a certain architecture
  - **Intrinsic Scaling**: how the wall-clock run time changes with an increase in the size of the problem
  - **Strong Scaling**: how the wall-clock run time changes when you increase the processing resources
  - **Weak Scaling**: how the wall-clock run time changes when you increase the problem size but also the processing resources in a way that basically keeps the ration of problem size/processor constant
A Word on “Scaling”
[important to understand]

- Two follow up comments

1. Worry about this: Is the Intrinsic Scaling similar to the Algorithmic Scaling?
   - If Intrinsic Scaling significantly worse than Algorithmic Scaling:
     - You might have an algorithm that thrashes the memory badly, or
     - You might have a sloppy implementation of the algorithm

2. If the problem doesn’t scale can be an interplay of several factors
   - The intrinsic nature of the problem at hand
   - The cleverness of the implementation
   - The attributes (organization) of the underlying hardware
   - The algorithm used to solve the problem
End: Intro Part

Beginning: GPU Computing, CUDA Programming Model
Where Does the GPU Hide?

Right here, there’s four of them little fellows…
Parallel Computing on a GPU

- NVIDIA GPU Computing Architecture
  - Connected to the motherboard via a separate HW interface
  - In laptops, desktops, workstations, servers

- Kepler K20X delivers 1.515 Tflops in double precision

- Multithreaded SIMT model uses application data parallelism and thread parallelism

- Programmable in C with CUDA tools
  - "Extended C"
Bandwidth in a CPU-GPU System

NOTE: The width of the black lines is proportional to the bandwidth.
Bandwidth of Common Interfaces

Internal
- RAM DDR3-1600
- PCI-E 3.0
- DisplayPort 1.3
- HDMI 2.0
- SATA 3.2

Home Network
- USB 3.1
- eSATA
- Gigabit Ethernet
- Typical HDD
- WiFi 802.11n
- USB 2.0

Internet Link
- LTE 4G
- Cable Modem
- HSPA+ 3.5G
- EV-DO 3G

One Super Important Point

- Almost all our applications are bound by memory speed
  - Most often, the cores (or SPs) idle waiting for data
  - What's important is how fast you can move data
    - You want high memory bandwidth
The Need for High Bandwidth

- Assume you have a 1 Tflops card:
  - Assume that you want to add different numbers and reach 1 Tflops: 1E12 ops/second
  - You need to feed 2E12 operands per second…
  - If each number is stored using 4 bytes (float), then you need to fetch 2E12*4 bytes in a second. This is 8E12 B/s, which is 8 TB/s…
  - The memory bandwidth on the GPU is in the neighborhood of 0.2 TB/s, about 40 times less than what you need (and you haven’t taken into account that you probably want to save the result of the operation that you carry out)
When Are GPUs Good?

- Ideally suited for data-parallel computing (SIMD)

- Moreover, you want to have high arithmetic intensity
  - Arithmetic intensity: ratio of arithmetic operations to memory operations

- You are off to a good start with GPU computing if you can do this…
  - GET THE DATA ON THE GPU AND KEEP IT THERE
  - GIVE THE GPU ENOUGH WORK TO DO
  - FOCUS ON DATA REUSE WITHIN THE GPU TO AVOID MEMORY BANDWIDTH LIMITATIONS
GPU Computing – The Basic Idea

- GPU, going beyond graphics:
  - The GPU is connected to the CPU by a reasonable fast bus
    - PCI 3.0 - 12 GB/s is typical today
  - The idea is to use the GPU as a co-processor
    - Farm out big parallel jobs to the GPU
    - CPU stays busy with the execution of “corner” tasks
    - You have to copy data down into the GPU, and then fetch results back
      - Ok if this data transfer is overshadowed by the number crunching done using that data (remember Amdahl’s law…)
“Compute Unified Device Architecture” – freely distributed by NVIDIA

When introduced it eliminated the constraints associated with GPGPU

It enables a general purpose programming model
  - User kicks off batches of threads on the GPU to execute a function (kernel)

Targeted software stack
  - Scientific computing oriented drivers, language, and tools

Driver for loading computation programs into GPU
  - Interface designed for compute, graphics-free API
  - Explicit GPU memory management
CUDA Programming Model: GPU as a Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a co-processor to the CPU or host
  - Has its own DRAM (global memory in CUDA parlance)
  - Runs many threads in parallel

- Data-parallel portions of an application run on the device as kernels which are executed in parallel by many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few heavy ones
Fermi: We’ve Seen This Slide...

- Lots of ALU (green), not much of CU
- GPUs are fast for high arithmetic intensity applications
- High arithmetic intensity: many operations performed per word of memory
The Fermi Architecture

- Late 2009, early 2010
- 40 nm technology
- Three billion transistors
- 512 Scalar Processors (SP, “shaders”)
- 64 KB L1 cache
- 768 KB L2 uniform cache (shared by all SMs)
- Up to 6 GB of global memory
- Operates at several clock rates
  - Memory
  - Scheduler
  - Shader (SP)
- High memory bandwidth
  - Close to 200 GB/s
GPU Processor Terminology

- GPU is a SIMD device → it works on “streams” of data
  - Each “GPU thread” executes one general instruction on the stream of data that the GPU is assigned to process
  - The NVIDIA calls this model SIMT (single instruction multiple thread)

- The number crunching power comes from a vertical hierarchy:
  - A collection of Streaming Multiprocessors (SMs)
  - Each SM has a set of 32 Scalar Processors (SPs)
    - Kepler has 196, Maxwell has 128 SPs, Fermi 2.1 had 48 SPs

- The quantum of scalability is the SM
  - The more $ you pay, the more SMs you get inside your GPU
  - Fermi can have up to 16 SMs on one GPU card
<table>
<thead>
<tr>
<th></th>
<th>NVS 5400M</th>
<th>NVS 5200M</th>
<th>NVS 4200M</th>
<th>NVS 3100M</th>
<th>NVS 2100M</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Cores</td>
<td>96</td>
<td>96</td>
<td>48</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>PhysX capable</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>OpenCL support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DirectX 11 support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DirectCompute support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OpenGL 2.1 support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Graphics Clock (MHz)</td>
<td>Up to 660</td>
<td>Up to 625</td>
<td>Up to 810</td>
<td>600</td>
<td>535</td>
</tr>
<tr>
<td>Processor Clock (MHz)</td>
<td>Up to 1320</td>
<td>Up to 1250</td>
<td>Up to 1620</td>
<td>1470</td>
<td>1230</td>
</tr>
<tr>
<td>Memory Amount</td>
<td>Up to 2 GB</td>
<td>1 GB</td>
<td>Up to 1 GB</td>
<td>Up to 512MB</td>
<td>Up to 512MB</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>128 bit</td>
<td>64 bit</td>
<td>64 bit</td>
<td>64 bit</td>
<td>64 bit</td>
</tr>
<tr>
<td>CUDA compute capability</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>PCI Express 2.0 support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

My laptop’s card: NVIDIA MVS 5200M (2 SMs w/ 48 SPs each)
Compute Capability [of a Device] vs. CUDA Version

- “Compute Capability of a Device” refers to hardware:
  - Defined by a major revision number and a minor revision number.
  
  - Example:
    - Tesla C1060 is compute capability 1.3
    - Tesla C2050 is compute capability 2.0
    - Fermi architecture is capability 2 (on Euler now)
    - Kepler architecture is capability 3 (the highest, on Euler now)
    - The minor revision number indicates incremental changes within an architecture class.

- A higher compute capability indicates a more able piece of hardware.

- The “CUDA Version” indicates what version of the software you are using to run on the hardware:
  - Right now, the most recent version of CUDA is 6.5.

- In a perfect world:
  - You would run the most recent CUDA (version 6.5) software release.
  - You would use the most recent architecture (compute capability 5.X).
Compatibility Issues

- The basic rule: the CUDA Driver API is backward, but not forward compatible
  - Makes sense, the functionality in later versions increased, was not there in previous versions
NVIDIA CUDA Devices

CUDA-Enabled Devices with Compute Capability, Number of Multiprocessors, and Number of CUDA Cores

<table>
<thead>
<tr>
<th>Card</th>
<th>Compute Capability</th>
<th>Number of Multiprocessors</th>
<th>Number of CUDA Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 690</td>
<td>3.0</td>
<td>2x8</td>
<td>2x1536</td>
</tr>
<tr>
<td>GTX 680</td>
<td>3.0</td>
<td>8</td>
<td>1536</td>
</tr>
<tr>
<td>GTX 670</td>
<td>2.1</td>
<td>7</td>
<td>1344</td>
</tr>
<tr>
<td>GTX 590</td>
<td>2.1</td>
<td>2x16</td>
<td>2x512</td>
</tr>
<tr>
<td>GTX 560TI</td>
<td>2.1</td>
<td>8</td>
<td>384</td>
</tr>
<tr>
<td>GTX 460</td>
<td>2.1</td>
<td>7</td>
<td>336</td>
</tr>
<tr>
<td>GTX 470M</td>
<td>2.1</td>
<td>6</td>
<td>288</td>
</tr>
<tr>
<td>GTX 460</td>
<td>2.1</td>
<td>4</td>
<td>192</td>
</tr>
<tr>
<td>GTX 445M</td>
<td>2.1</td>
<td>3</td>
<td>144</td>
</tr>
<tr>
<td>GTX 435M, GTX 425M, GTX 420M</td>
<td>2.1</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>GTX 415M</td>
<td>2.1</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>GTX 490</td>
<td>2.0</td>
<td>2x15</td>
<td>2x480</td>
</tr>
<tr>
<td>GTX 580</td>
<td>2.0</td>
<td>16</td>
<td>512</td>
</tr>
<tr>
<td>GTX 570, GTX 480</td>
<td>2.0</td>
<td>15</td>
<td>480</td>
</tr>
<tr>
<td>GTX 470</td>
<td>2.0</td>
<td>14</td>
<td>448</td>
</tr>
<tr>
<td>GTX 465, GTX 480</td>
<td>2.0</td>
<td>11</td>
<td>352</td>
</tr>
<tr>
<td>GTX 295</td>
<td>1.3</td>
<td>2x30</td>
<td>2x240</td>
</tr>
<tr>
<td>GTX 285, GTX 280, GTX 275</td>
<td>1.3</td>
<td>30</td>
<td>240</td>
</tr>
<tr>
<td>GTX 260</td>
<td>1.3</td>
<td>24</td>
<td>192</td>
</tr>
<tr>
<td>9800 GX2</td>
<td>1.1</td>
<td>2x16</td>
<td>2x128</td>
</tr>
<tr>
<td>GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512, GTX 285M, GTX 280M</td>
<td>1.1</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>8800 Ultra, 8800 GTX</td>
<td>1.0</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>9800 GT, 8800 GT</td>
<td>1.1</td>
<td>14</td>
<td>112</td>
</tr>
</tbody>
</table>
The CUDA Execution Model
GPU Computing – The Basic Idea
[We’ve seen this before…]

- The GPU is linked to the CPU by a reasonably fast connection

- The idea is to use the GPU as a co-processor
  - Farm out big parallel tasks to the GPU
  - Keep the CPU busy with the control of the execution and “corner” tasks
The CUDA Execution Model is Asynchronous

This is how your C code looks like

This is how the code gets executed on the hardware in heterogeneous computing. GPU calls are asynchronous...
Languages Supported in CUDA

- Note that everything is done in C
  - Yet minor extensions are needed to flag the fact that a function actually represents a kernel, that there are functions that will only run on the device, etc.
    - You end up working in “C with extensions”

- FOTRAN is supported too

- There is [limited] support for C++ programming (operator overload, for instance)
### CUDA Function Declarations
(the “C with extensions” part)

<table>
<thead>
<tr>
<th></th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float myDeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void myKernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float myHostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- __global__ defines a kernel function, launched by host, executed on the device
  - Must return **void**
- For a full list, see CUDA Reference Manual
CUDA, First Example

```cpp
#include <cutil_inline.h>
#include <iostream>

__global__ void simpleKernel(int* data)
{
    // write something trivial to the global memory...
    data[threadIdx.x] = blockIdx.x + threadIdx.x;
}

int main()
{
    int hostArray[4], *devArray;
    // allocate memory on the device (GPU)
    cudaMalloc((void**)&devArray, sizeof(int)*4);

    // invoke GPU kernel, with one block that has four threads
    simpleKernel<<<1,4>>>(devArray);

    // bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int)*4, cudaMemcpyDeviceToHost);

    // print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: ";
    std::cout << hostArray[0] << ", ";
    std::cout << hostArray[1] << ", ";
    std::cout << hostArray[2] << ", ";
    std::cout << hostArray[3] << std::endl;

    // release the memory allocated on the GPU
    cudaFree(devArray);

    return 0;
}
```
First Example, Nuts and Bolts

● Here’s how you compile on euler

```
[negrut@euler CodeBits]$ qsub -I -l nodes=1:gpus=1:default -X
[negrut@euler01 CodeBits]$ nvcc -gencode arch=compute_20, code=sm_20 test.cu
[negrut@euler01 CodeBits]$ ./test
```

● Instructions on how to access Euler:

http://wacc.wisc.edu/docs/

● Here’s what you’d get as output
  ● Shown on my machine, which ran Windows

![Image of a command prompt showing output](image.png)
CUDA, Second Example

- Multiply, pairwise, two arrays of 3 million integers

```c
1. int main(int argc, char* argv[])  
2. {  
3.   const int arraySize = 3000000;  
4.   int *hA, *hB, *hC;  
5.   setupHost(&hA, &hB, &hC, arraySize);  
6.  
7.   int *dA, *dB, *dC;  
8.   setupDevice(&dA, &dB, &dC, arraySize);  
9.  
10.  cudaMemcpy(dA, hA, sizeof(int) * arraySize, cudaMemcpyHostToDevice);  
11.  cudaMemcpy(dB, hB, sizeof(int) * arraySize, cudaMemcpyHostToDevice);  
12.  
13.   const int threadsPerBlock = 512;  
14.   const int blockSizeMultiplication = arraySize/threadsPerBlock + 1;  
15.   multiply_ab<<<blockSizeMultiplication,threadsPerBlock>>>(dA,dB,dC,arraySize);  
16.   cudaMemcpy(hC, dC, sizeof(int) * arraySize, cudaMemcpyDeviceToHost);  
17.   cleanupHost(hA, hB, hC);  
18.   cleanupDevice(dA, dB, dC);  
19.   return 0;  
20. }
```
CUDA, Second Example

[Cntd.]

```c
__global__ void multiply_ab(int* a, int* b, int* c, int size) {
    int whichEntry = threadIdx.x + blockIdx.x*blockDim.x;
    if (whichEntry<size)
        c[whichEntry] = a[whichEntry]*b[whichEntry];
}
```

```c
void setupDevice(int** pdA, int** pdB, int** pdC, int arraySize) {
    cudaMalloc((void**) pdA, sizeof(int) * arraySize);
    cudaMalloc((void**) pdB, sizeof(int) * arraySize);
    cudaMalloc((void**) pdC, sizeof(int) * arraySize);
}
```

```c
void cleanupDevice(int *dA, int *dB, int *dC) {
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
}
```
Review of Nomenclature…

- **The HOST**
  - This is your CPU executing the “master” thread

- **The DEVICE**
  - This is the GPU card, connected to the HOST through a PCIe connection

- **The HOST** (the master CPU thread) calls **DEVICE** to execute **KERNEL**

- When calling the **KERNEL**, the **HOST** also has to inform the **DEVICE** how many threads should each execute the **KERNEL**
  - This is called “defining the execution configuration”
The Concept of Execution Configuration

- A kernel function must be called with an execution configuration:

```c
__global__ void kernelFoo(...); // declaration

dim3 DimGrid(100, 50); // 5000 thread blocks
dim3 DimBlock(4, 8, 8); // 256 threads per block

kernelFoo<<< DimGrid, DimBlock>>>(...your arg list comes here…);
```

- Recall that any call to a kernel function is **asynchronous**
  - By default, execution on host doesn’t wait for kernel to finish
The host call below instructs the GPU to execute the function (kernel) “foo” using 25,600 threads
- Two arguments are passed down to each thread executing the kernel “foo”

foo<<<100,256>>>(pMyMatrixD, pMyVecD)

In this execution configuration, the host instructs the device that it is supposed to run 100 blocks each having 256 threads in it

The concept of block is important since it represents the entity that gets executed by an SM (stream multiprocessor)
More on the Execution Model
[Some Constraints]

- There is a limitation on the number of blocks in a grid:
  - The grid of blocks can be organized as a 3D structure: max of 65,535 by 65,535 by 65,535 grid of blocks (about 280,000 billion blocks)

- Threads in each block:
  - The threads can be organized as a 3D structure \((x,y,z)\)
  - The total number of threads in each block cannot be larger than 1024
    - More on this 1024 number later
Matrix Multiplication Example
Simple Example: Matrix Multiplication

- **Purpose:** Illustrate the basic features of memory and thread management in CUDA programs

- **Quick remarks**
  - Use only global memory (don’t bring shared memory into picture yet)
  - Matrix will be of small dimension, job can be done using one block
  - Concentrate on
    - Thread ID usage
  - Memory data transfer API between host and device
Matrix Data Structure

- The following data structure will come in handy
  - Purpose: store info related to a matrix
  - Note that the matrix is stored in row-major order in a one dimensional array pointed to by “elements”

```c
// IMPORTANT - Matrices are stored in row-major order:
// M(row, col) = M.elements[row * M.width + col]

typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```
Square Matrix Multiplication Example

- Compute $P = M \times N$
  - The matrices $P$, $M$, $N$ are of size $\text{WIDTH} \times \text{WIDTH}$
  - Assume $\text{WIDTH}$ was defined to be 32

- Software Design Decisions:
  - One thread handles one element of $P$
  - Each thread will access all the entries in one row of $M$ and one column of $N$
    - $2\times\text{WIDTH}$ read accesses to global memory
    - One write access to global memory
Multiply Using One Thread Block

- One Block of threads computes matrix P
  - Each thread computes one element of P

- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1
    - Not that good, acceptable for now…

- Size of matrix limited by the number of threads allowed in a thread block
Matrix Multiplication:
Sequential Approach, Coded in C

// Matrix multiplication on the (CPU) host in double precision;

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i) {
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];  //march along a row of M
                double b = N.elements[k * N.width + j];  //march along a column of N
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
    }
}
int main(void) {
    // Allocate and initialize the matrices.
    // The last argument in AllocateMatrix: should an initialization with
    // random numbers be done? Yes: 1. No: 0 (everything is set to zero)
    Matrix M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix P  = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);

    return 0;
}
Step 2: Matrix Multiplication

[host-side code]

```c
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);

    // Setup the execution configuration
    dim3 dimGrid(1, 1, 1);
    dim3 dimBlock(WIDTH, WIDTH);

    // Launch the kernel on the device
    MatrixMulKernel<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```
Step 4: Matrix Multiplication - Device-side Kernel Function

```c
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P) {
    // 2D Thread Index; computing P[ty][tx]...
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue will end up storing the value of P[ty][tx].
    // That is, P.elements[ty * P.width + tx] = Pvalue
    float Pvalue = 0;

    for (int k = 0; k < M.width; ++k) {
        float Melement = M.elements[ty * M.width + k];
        float Nelement = N.elements[k * N.width + tx];
        Pvalue += Melement * Nelement;
    }

    // Write matrix to device memory; each thread one element
    P.elements[ty * P.width + tx] = Pvalue;
}
```
Step 4: Some Loose Ends

// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M) {
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMemcpy((void**)&Mdevice.elements, M.elements, size, cudaMemcpyHostToDevice);
    return Mdevice;
}

// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost) {
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size, cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice) {
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size, cudaMemcpyDeviceToHost);
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
Block and Thread Index (Idx)

- Threads and blocks have indices
  - Used by each thread to decide what data to work on (more later)
  - Block Index: a triplet of uint
  - Thread Index: a triplet of uint

- Why this 3D layout?
  - Simplifies memory addressing when processing multidimensional data
    - Handling matrices
    - Solving PDEs on subdomains
    - ...
A Couple of Built-In Variables
[Critical in supporting the SIMD parallel computing paradigm]

- It’s essential for each thread to be able to find out the grid and block dimensions and its block index and thread index.

- Each thread when executing a kernel has access to the following read-only built-in variables:
  - `threadIdx` (uint3) – contains the thread index within a block
  - `blockDim` (dim3) – contains the dimension of the block
  - `blockIdx` (uint3) – contains the block index within the grid
  - `gridDim` (dim3) – contains the dimension of the grid
  - `warpSize` (uint) – provides warp size, we’ll talk about this later…]
Thread Index vs. Thread ID

[critical in (i) understanding how SIMD is supported in CUDA, and (ii) understanding the concept of “warp”]

- Each block organizes its threads in a 3D structure defined by its three dimensions: \( D_x \), \( D_y \), and \( D_z \) that you specify.

- A block cannot have more than 1024 threads \( \Rightarrow \) \( D_x \times D_y \times D_z \leq 1024 \).

- Each thread in a block can be identified by a unique index \((x, y, z)\), and

\[
0 \leq x < D_x \quad 0 \leq y < D_y \quad 0 \leq z < D_z
\]

- A triplet \((x, y, z)\), called the thread index, is a high-level representation of a thread in the economy of a block. Under the hood, the same thread has a simplified and unique id, which is computed as \( t_{id} = x + y \cdot D_x + z \cdot D_x \cdot D_y \). You can regard this as a ”projection” to a 1D representation. The concept of thread id is important in understanding how threads are grouped together in warps (more on ”warps” later).

- In general, operating for vectors typically results in you choosing \( D_y = D_z = 1 \). Handling matrices typically goes well with \( D_z = 1 \). For handling PDEs in 3D you might want to have all three block dimensions nonzero.
Revisit - Execution Configuration: Grids and Blocks

- A kernel is executed as a **grid of blocks of threads**
  - All threads executing a kernel can access several device data memory spaces

- A **block [of threads]** is a collection of threads that can **cooperate** with each other by:
  - Synchronizing their execution
  - Efficiently sharing data through a low latency **shared memory**

- Check your understanding:
  - How was the grid defined for this pic? I.e., how many blocks in X and Y directions?
  - How was a block defined in this pic?
Example: Array Indexing

- Purpose of Example: see a scenario of how multiple blocks are used to index entries in an array

- First, recall this: there is a limit on the number of threads you can squeeze in a block (up to 1024 of them)

- Note: In the vast majority of applications you need to use many blocks (each of which contains the same number N of threads) to get a job done. This example puts things in perspective
With $M$ threads per block a unique index for each thread is given by:

$$
\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} \times M;
$$
What will be the array entry that thread of index 5 in block of index 2 will work on?

```
int index = threadIdx.x + blockIdx.x * M;
=      5      +     2      * 8;
= 21;
```
A Recurring Theme in CUDA Programming
[and in SIMD in general]

- Imagine you are one of many threads, and you have your thread index and block index

- You need to figure out what the task you need to do
  - Just like we did on previous slide where thread 5 in block 2 mapped into 21

- One caveat: You have to make sure you actually need to do that work
  - In many cases there are threads, typically of large id, that need to do no work
  - Example: you launch two blocks with 512 threads but your array is only 1000 elements long. Then 24 threads at the end do nothing
Before Moving On…
[Some Words of Wisdom]

- In GPU computing you use as many threads as data items (tasks, jobs) you have to perform
  - This replaces the purpose in life of the “for” loop
  - Number of threads & blocks is established at run-time

- Number of threads = Number of data items (tasks)
  - It means that you’ll have to come up with a rule to match a thread to a data item (task) that this thread needs to process
  - Major source of errors and frustration in GPU computing
    - It never fails to deliver (frustration)
      :-(
Timing Your Application

- Timing support – part of the CUDA API
  - You pick it up as soon as you include `<cuda.h>`

- Why it is good to use
  - Provides cross-platform compatibility
  - Deals with the asynchronous nature of the device calls by relying on events and forced synchronization

- Reports time in milliseconds, accurate within 0.5 microseconds
  - From NVIDIA CUDA Library Documentation:
    - Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). If either event has not been recorded yet, this function returns `cudaErrorInvalidValue`. If either event has been recorded with a non-zero stream, the result is undefined.
Timing Example
~ Timing a GPU call ~

```cpp
#include<iostream>
#include<cuda.h>

int main() {
    cudaEvent_t startEvent, stopEvent;
    cudaEventCreate(&startEvent);
    cudaEventCreate(&stopEvent);

    cudaEventRecord(startEvent, 0);

    yourKernelCallHere<<<NumBlk,NumThrds>>>(args);

    cudaEventRecord(stopEvent, 0);
    cudaEventSynchronize(stopEvent);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);
    std::cout << "Time to get device properties: " << elapsedTime << " ms\n";

    cudaEventDestroy(startEvent);
    cudaEventDestroy(stopEvent);
    return 0;
}
```
The CUDA API
What is an API?

- Application Programming Interface (API)
  - “A set of functions, procedures or classes that an operating system, library, or service provides to support requests made by computer programs” (from Wikipedia)
  - Example: OpenGL, a graphics library, has its own API that allows one to draw a line, rotate it, resize it, etc.

- In this context, CUDA provides an API that enables you to tap into the computational resources of the NVIDIA’s GPUs
  - This replaced the old GPGPU way of programming the hardware
  - CUDA API exposed to you through a collection of header files that you include in your program

105
On the CUDA API

- Reading the CUDA Programming Guide you’ll run into numerous references to the CUDA Runtime API and CUDA Driver API
  - Many time they talk about “CUDA runtime” and “CUDA driver”. What they mean is CUDA Runtime API and CUDA Driver API

- CUDA Runtime API – is the friendly face that you can choose to see when interacting with the GPU. This is what gets identified with “C CUDA”
  - Needs nvcc compiler to generate an executable

- CUDA Driver API – low level way of interacting with the GPU
  - You have significantly more control over the host-device interaction
  - Significantly more clunky way to dialogue with the GPU, typically only needs a C compiler

- I don’t anticipate any reason to use the CUDA Driver API
Talking about the API:
The C CUDA Software Stack

- Image at right indicates where the API fits in the picture

An API layer is indicated by a thick red line:

- NOTE: any CUDA runtime function has a name that starts with “cuda”
  - Examples: cudaMalloc, cudaFree, cudaMemcpy, etc.
- Examples of CUDA Libraries: CUFFT, CUBLAS, CUSP, thrust, etc.
Application Programming Interface (API) ~Taking a Step Back~

- CUDA runtime API: exposes a set of extensions to the C language
  - Spelled out in an appendix of “NVIDIA CUDA C Programming Guide”
  - There is many of them → Keep in mind the 20/80 rule

- CUDA runtime API:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library, which is split into:
    - A common component providing built-in vector types and a subset of the C runtime library available in both host and device codes
      - Callable both from device and host
    - A host component to control and access devices from the host
      - Callable from the host only
    - A device component providing device-specific functions
      - Callable from the device only
## Language Extensions: Variable Type Qualifiers

<table>
<thead>
<tr>
<th><strong>device</strong> <strong>local</strong></th>
<th>int LocalVar;</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>shared</strong></td>
<td>int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong></td>
<td>int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- **__device__** is optional when used with __local__, __shared__, or __constant__

- **Automatic variables** without any qualifier reside in a register
  - Except arrays, which reside in local memory (unless they are small and of known constant size)
Common Runtime Component

- “Common” above refers to functionality that is provided by the CUDA API and is common both to the device and host.

- Provides:
  - Built-in vector types
  - A subset of the C runtime library supported in both host and device codes
**Common Runtime Component: Built-in Vector Types**

- `[u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4], double[1..2]
  - Structures accessed with \( x, y, z, w \) fields:
    ```cpp
    uint4 param;
    int dummy = param.y;
    ```

- `dim3`
  - Based on `uint3`
  - Used to specify dimensions
  - You see a lot of it when defining the execution configuration of a kernel (any component left uninitialized assumes default value 1)

See Appendix B in
“NVIDIA CUDA C Programming Guide”
Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- etc.

- When executed on the host, a given function uses the C runtime implementation if available
- These functions only supported for scalar types, not vector types
Host Runtime Component

- Provides functions available only to the host to deal with:
  - Device management (including multi-device systems)
  - Memory management
  - Error handling

- Examples
  - Device memory allocation
    - `cudaMalloc()`, `cudaFree()`
  - Memory copy from host to device, device to host, device to device
    - `cudaMemcpy()`, `cudaMemcpy2D()`, `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`
  - Memory addressing – returns the address of a device variable
    - `cudaGetSymbolAddress()`
CUDA Device Memory Space Overview

[Quick Overview of GPU Memory Ecosystem]

- Image shows the memory hierarchy that a block sees while running on an SM

- The host can R/W global, constant, and texture memory
CUDA API: Device Memory Allocation
[Note: picture assumes two blocks, each with two threads]

- **cudaMalloc()**
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object

- **cudaFree()**
  - Frees object from device **Global Memory**
  - Pointer to freed object
Example Use: A Matrix Data Type

- NOT part of CUDA API
- Used in several code examples
  - 2 D matrix
  - Single precision float elements
  - width * height entries
  - Matrix entries attached to the pointer-to-float member called "elements"
  - Matrix is stored row-wise

```c
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```
Example
CUDA Device Memory Allocation (cont.)

- Code example:
  - Allocate a $64 \times 64$ single precision float array
  - Attach the allocated storage to \texttt{Md.elements}
  - “\texttt{d}” in “\texttt{Md}” is often used to indicate a device data structure

```c
BLOCK_SIZE = 64;
Matrix Md;
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**)Md.elements, size);
...
//use it for what you need, then free the device memory
cudaFree(Md.elements);
```

\textbf{Question}: why is the type of the first argument \texttt{(void **)}?
CUDA Host-Device Data Transfer

- `cudaMemcpy()`
  - memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device

CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - \texttt{M} is in host memory and \texttt{Md} is in device memory
  - \texttt{cudaMemcpyHostToDevice} and \texttt{cudaMemcpyDeviceToHost} are symbolic constants

\begin{verbatim}
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);
\end{verbatim}
Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. \( \sin(x) \)) have a less accurate, but faster device-only version (e.g. \_\_\_\sin(x)\)
  - \_\_\_pow
  - \_\_\_log, \_\_\_log2, \_\_\_log10
  - \_\_\_exp
  - \_\_\_sin, \_\_\_cos, \_\_\_tan

- Some of these have hardware implementations

- By using the "-use_fast_math" flag, \( \sin(x) \) is substituted at compile time by \_\_\_\sin(x)

>> nvcc -arch=sm_20 -use_fast_math foo.cu
CPU vs. GPU – Flop Rate (GFlops)