Computational Accelerators: GPU
Evolution of high-performance computing

Long-standing forces governing HPC systems
– constructed using *commodity* CPUs (mostly)

Recent market forces
– Server farms
  • large memory, more cores, more I/O
– Gaming
  • GPUs for real-time graphics
– Cell phones
  • Signal processing hardware:
    – compression, computational photography

Computational accelerators emerge from GPUs
– 2007: Nvidia Compute Unified Device Architecture GPU (CUDA)
– 2009: IBM/Toshiba/Sony Cell Broadband Engine (Cell BE) PlayStation 3
– 2010: Intel Larrabee (DOA) → Many Integrated Cores (MIC) → Xeon Phi
Revolution

- Nvidia Tesla V100
  - 2017
  - 7 TF/s with 5120 ALUs and 16GB memory on a single die

- ASCI white
  - 2001 top linpack machine on the planet
  - 4.9TF/s with 8000 processors, occupying the space of 2 basketball courts and weighing over 100 tons.
CPU and GPU are designed very differently

**CPU**
- Low latency cores
- Core
  - Local Cache
  - Registers
  - SIMD Unit
  - Control

**GPU**
- High throughput cores
- Compute Unit
  - Cache/Local Mem
  - Registers
  - SIMD Unit
  - Threading
CPUs: Latency Oriented Design

- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Instruction dependency analysis and superscalar operation
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
GPUs: Throughput Oriented Design

- Small caches
  - High bandwidth main memory
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, high latency, ALUs heavily pipelined for high throughput
- Requires massive number of threads to tolerate latencies
  - Threading logic
  - Thread state
Performance scaling has encountered major limitations
- cannot increase clock frequency
- cannot increase power
- can increase transistor count
Using accelerators in HPC systems

• Barriers
  – not general purpose, only good for some problems
  – difficult to program
  – interface to host system can be a bottleneck
  – low precision arithmetic (this is now a feature!)

• Incentives
  – cheap
  – becoming more general-purpose and easier to program
  – improving host interfaces and performance
  – IEEE double precision
  – very high compute and local memory performance

• They are being used!
  – NSC China Tianhe-2 48,000 Intel Xeon Phi
  – ORNL USA Summit 27,600 Nvidia Tesla V100

• Current trends
  – Simplified access from host
  – Improved integration of multiple GPUs
  – Low- and mixed-precision FP arithmetic
Host and accelerator interface

(dual socket Intel Xeon E5 v3)

Host system diagram

accelerators

Intel Xeon Phi 5110P

Nvidia Titan V100

16 GB/s bidirectional
Nvidia GPU organization

- **GPU**
  - device is a set of $N$ (1 - 84) streaming multiprocessors (SM)
  - each SM executes one or more blocks of threads
  - each SM has $M$ (1 - 4) sets of 32 SIMD processors
  - at each clock cycle, a SIMD processor executes a single instruction on a group of 32 threads called a warp
  - total of $N \times M \times 32$ arithmetic operations per clock

- **Volta V100** $N=80$, $M=2$ up to 5120 SP floating point operations per clock
Volta V100 chip organization

- up to 84 SMs
- shared L2 cache (6MB)
- interfaces: 8 memory controllers, 6 NVLink intfc, PCIe host intfc
Volta V100 SM organization

- 64 single-precision FP32 arithmetic units
- 32 double-precision FP64 arithmetic units
- 64 integer arithmetic units
- 16 special function units
- 32 load/store units

- 64K registers
  - allocated across threads

- 128KB data cache / shared memory
  - L1 cache
  - user-allocated shared memory

- 4 warps can be running concurrently
  - up to 2 instructions per warp concurrently
CUDA memory hierarchy

- **Host memory**
  - **Device memory**
    - shared between N multiprocessors
    - global, constant, and texture memory (4-32 GB total)
    - can be accessed by host
  - **Shared Memory**
    - shared by SIMD processors
    - R/W shared memory and L1 cache
    - R/O constant/texture cache
  - **SIMD register memory**
    - set of 32-bit registers
CUDA Control Hierarchy

- **A CUDA context consists of streams**
  - A **stream** is a sequence of kernels
    - kernels execute in sequence
    - kernels share device memory
    - different streams may run concurrently
  
  - A **kernel** is a grid of blocks
    - blocks share device memory
    - blocks are scheduled across SMs and run concurrently

- **A block** is a collection of threads that
  - may access shared memory
  - can synchronize execution
  - are executed as a set of warps

- A **warp** of threads runs SIMD style
  - Multiple warps may run concurrently
Execution Model

- A grid consists of multiple blocks
  - each block has a 1D, 2D, or 3D Block ID
  - a block is assigned to an SM
  - multiple blocks are required to fully utilize all SMs
    - more blocks per grid are better

- Each block consists of multiple threads
  - each thread has a 1D, 2D, or 3D Thread ID
  - threads are executed concurrently SIMD style one warp at a time
  - hardware switches between warps on any stall (e.g. load)
  - multiple threads are required to keep hardware busy
    - 64 - 1024 threads can be used to hide latency

- Each warp consists of 32 threads
  - execution of a warp is like the synchronous CRCW PRAM model
## Compute capability

<table>
<thead>
<tr>
<th>Feature</th>
<th>Kepler GK180</th>
<th>Maxwell GM200</th>
<th>Pascal GP100</th>
<th>Volta GV100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>3.5</td>
<td>5.2</td>
<td>6.0</td>
<td>7.0</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Warps / SM</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Threads / SM</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Thread Blocks / SM</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max 32-bit Registers / SM</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Block</td>
<td>65536</td>
<td>32768</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Thread</td>
<td>255</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>Max Thread Block Size</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>FP32 Cores / SM</td>
<td>192</td>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Ratio of SM Regs to FP32 Cores</td>
<td>341</td>
<td>512</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Shared Memory Size / SM</td>
<td>16/32/48 KB</td>
<td>96KB</td>
<td>64KB</td>
<td>config 96KB</td>
</tr>
</tbody>
</table>

CUDA GPU programming
# Comparison of Nvidia Tesla GPUs

<table>
<thead>
<tr>
<th>Tesla Product</th>
<th>Tesla K40</th>
<th>Tesla M40</th>
<th>Tesla P100</th>
<th>Tesla V100</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>GK108 (Kepler)</td>
<td>GM200 (Maxwell)</td>
<td>GP100 (Pascal)</td>
<td>GV100 (Volta)</td>
</tr>
<tr>
<td>SMs</td>
<td>15</td>
<td>24</td>
<td>56</td>
<td>80</td>
</tr>
<tr>
<td>TPCs</td>
<td>15</td>
<td>24</td>
<td>28</td>
<td>40</td>
</tr>
<tr>
<td>FP32 Cores / SM</td>
<td>192</td>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>FP32 Cores / GPU</td>
<td>2880</td>
<td>3072</td>
<td>3584</td>
<td>5120</td>
</tr>
<tr>
<td>FP64 Cores / SM</td>
<td>64</td>
<td>4</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>FP64 Cores / GPU</td>
<td>960</td>
<td>96</td>
<td>1792</td>
<td>2560</td>
</tr>
<tr>
<td>Tensor Cores / SM</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>8</td>
</tr>
<tr>
<td>Tensor Cores / GPU</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>640</td>
</tr>
<tr>
<td>GPU Boost Clock</td>
<td>810/875 MHz</td>
<td>1114 MHz</td>
<td>1480 MHz</td>
<td>1530 MHz</td>
</tr>
<tr>
<td>Peak FP32 TFLOPS(^1)</td>
<td>5</td>
<td>6.8</td>
<td>10.6</td>
<td>15.7</td>
</tr>
<tr>
<td>Peak FP64 TFLOPS(^1)</td>
<td>1.7</td>
<td>.21</td>
<td>5.3</td>
<td>7.8</td>
</tr>
<tr>
<td>Peak Tensor TFLOPS(^1)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>125</td>
</tr>
<tr>
<td>Texture Units</td>
<td>240</td>
<td>192</td>
<td>224</td>
<td>320</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>384-bit GDDR5</td>
<td>384-bit GDDR5</td>
<td>4096-bit HBM2</td>
<td>4096-bit HBM2</td>
</tr>
<tr>
<td>Memory Size</td>
<td>Up to 12 GB</td>
<td>Up to 24 GB</td>
<td>16 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>1536 KB</td>
<td>3072 KB</td>
<td>4096 KB</td>
<td>6144 KB</td>
</tr>
<tr>
<td>Shared Memory Size / SM</td>
<td>16 KB/32 KB/48 KB</td>
<td>96 KB</td>
<td>64 KB</td>
<td>Configurable up to 96 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>20480 KB</td>
</tr>
<tr>
<td>TDP</td>
<td>235 Watts</td>
<td>250 Watts</td>
<td>300 Watts</td>
<td>300 Watts</td>
</tr>
<tr>
<td>Transistors</td>
<td>7.1 billion</td>
<td>8 billion</td>
<td>15.3 billion</td>
<td>21.1 billion</td>
</tr>
<tr>
<td>GPU Die Size</td>
<td>551 mm(^2)</td>
<td>601 mm(^2)</td>
<td>610 mm(^2)</td>
<td>815 mm(^2)</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td>28 nm</td>
<td>28 nm</td>
<td>16 nm FinFET+</td>
<td>12 nm FinFET+</td>
</tr>
</tbody>
</table>

\(^1\) Peak TFLOPS rates are based on GPU Boost Clock
CUDA Application Programming Interface

- The `cuda` API is an extension to the C programming language
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library split into:
    - A common component for host and device codes providing
      - built-in vector types and a
      - subset of the C runtime library
    - A host component to control and access CUDA devices
    - A device component providing device-specific functions

- Tools for `cuda`
  - `nvcc` compiler
    - runs cuda compiler on .cu files, and gcc on other files
  - `nvprof` profiler
    - reports on device performance including host-device transfers
### CUDA C Language Extensions: Type Qualifiers

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

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1
Language Extensions: Built-in Variables

- `dim3 gridDim;`  
  - Dimensions of the grid in blocks
- `dim3 blockDim;`  
  - Dimensions of the block in # threads
- `dim3 blockIdx;`  
  - Block index within the grid
- `dim3 threadIdx;`  
  - Thread index within the block

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1
## CUDA Function Declarations

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- __global__ defines a kernel function
  - Must return void

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1
Calling a Kernel Function

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

```c
__global__ void KernelFunc(...);
dim3 DimGrid(100, 50);  // 5000 thread blocks
dim3 DimBlock(4, 8, 8);  // 256 threads per block
size_t SharedMemBytes = 64;  // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

• Any call to a kernel function is asynchronous in the host from CUDA 1.0 on, explicit synchronization needed to await completion

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1
A simple example

```c
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

- single block, with N threads
  - also need to allocate and move A, B, C

- How large can the vectors be?

- What kind of performance could we expect?
Host and device memory

- Separate address spaces (compute capability < 6.0)
  - cudaMemCopy to move data back and forth

- Unified address space (compute capability >= 6.0)
  - host and device “page” out of a single address space

Example: vector addition