524 GPUs
**CPU**
1690 pJ/flop
Optimized for Latency
Caches

**GPU**
140 pJ/flop
Optimized for Throughput
Explicit Management of On-chip Memory

Westmere
32 nm

Kepler
28 nm
1\textsuperscript{ST} ERA: Fixed Function

3D Geometry Transformation

\[
V_{ev} = M\cdot V_{in}
\]

\[
V_{ev} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}
\]

\[
V_{in} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}
\]

Lighting

\[
C_p = k_dL_o + \sum_{n=lights} A_t\cdot\left(k_a(\hat{L}_a\cdot\hat{N}) + k_s(\hat{R}_a\cdot\hat{V})^\gamma\right)
\]

2\textsuperscript{ND} ERA: Simple Shaders

Memory Interface

8 Vertex Pipes

Setup Engine

Pixel Shader Core

16 Pixel Pipes

3\textsuperscript{RD} ERA: Graphics Parallel Core

VLIW5

VLIW4

General Purpose Registers

Stream Processing Units

Branch Unit

FMADD Special Functions
GPUs have enormous power that is enormously difficult to use

- Nvidia GP100 - 5.3TFlops of double precision
  - This is equivalent to the fastest super computer in the world in 2001; put a single rack together of these and you would be in the top 500 (you could get there for under $1MM)
- Gianormous: 600mm^2
- 3,840 CUDA cores
- 720GB/s die-stacked DRAM (16GB total)
  - This is equivalent to about 11 DDR4 channels
- **It’s a process**
  - ~ 2001 first hints of programmability
  - ~ mid 2000’s useful 32 bit math, sometime later 64 bit
  - ~ 2013 unified virtual addressing
    - ~ 2015/6 and products that don’t totally suck at it
  - Active areas of research: GPU accesses to the filesystem network, etc.

- partitioned address space
  - (although this is changing!)
  - CPU dispatches work to GPU
  - GPU is not a first class compute device
  - ongoing research on this!
Some terminology...

- MIMD = Multiple Instruction, Multiple Data
  - Multicore
- SIMD = Single Instruction, Multiple Data
  - Vector
- FGMT = Fine-Grained Multithreading
- VLIW = Very Long Instruction Word
- Vector = what you think it means from Mathematics
- Bandwidth
- Injection Rate or Packet Rate or Message Rate
  - Peak Injection Rate = Peak Bandwidth / Smallest Packet Size
  - Peak Bandwidth = Largest Packet Size X Injection Rate (usually not peak)
- Channel or Bus
  - as in DDR3 channel, which for a typical rate has peak BW of ~18GB/s and Injection Rate of 280M Msg/S.
## AMD / NVidia lingua franca

<table>
<thead>
<tr>
<th></th>
<th>AMD</th>
<th>NVidia</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workitem</td>
<td>Workitem</td>
<td>Thread</td>
<td>a single task</td>
</tr>
<tr>
<td>Workgroup</td>
<td>Thread-block / CTA</td>
<td>Workgroup</td>
<td>a group of threads that can share data and synchronize locally</td>
</tr>
<tr>
<td>Wavefront</td>
<td>Thread-block / CTA</td>
<td>Wavefront</td>
<td>unit of hardware SIMT scheduling</td>
</tr>
<tr>
<td>VGPR (64x32 bit)</td>
<td>VGPR (32x32 bit)</td>
<td>VGPR (64x32 bit)</td>
<td>Vector general purpose register</td>
</tr>
<tr>
<td>SGPR (32 bit)</td>
<td>N/A</td>
<td>SGPR (32 bit)</td>
<td>Scalar general purpose register</td>
</tr>
<tr>
<td>Global memory</td>
<td>Global memory</td>
<td>Global memory</td>
<td>Globally shared memory across all workitems/threads</td>
</tr>
<tr>
<td>Local memory (LDS)</td>
<td>Shared memory</td>
<td>Shared memory</td>
<td>Software managed scratch-pad shared between a workgroup or thread-block. Per SM/CU</td>
</tr>
<tr>
<td>Private memory</td>
<td>Local memory</td>
<td>Local memory</td>
<td>Workitem/thread private memory</td>
</tr>
</tbody>
</table>

Mark

Emily

The person to ask…
A simplified perspective on GPU architecture
A simplified perspective on GPU architecture

must have lots of threads

main memory
A simplified perspective on GPU architecture

threads must not branch diverge

main memory
A simplified perspective on GPU architecture

Must explicitly copy memory to LDS for good performance.
A simplified perspective on GPU architecture

GPUs will try hard to use global memory
A simplified perspective on GPU architecture

but they will fail at it when push comes to shove
After much consternation I decided to focus class lecture on OpenCL. This is because it is supported by AMD, NVidia and Intel and works on Mac OS X, Windows and Linux. Personally, I like this. But it’s a work in progress still and doesn’t work with NVidia.

Like C++Amp but I don’t have time/ infrastructure

To first order, no one cares

No one cares
Core concepts

• Memory, Memory, Memory
  • Memory hierarchy
• SIMD execution
• Threads for latency tolerance
An important mindset

Execution is free, data access is not
## Energy Shopping List

<table>
<thead>
<tr>
<th>Processor Technology</th>
<th>40 nm</th>
<th>10 nm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vdd (nominal)</td>
<td>0.9 V</td>
<td>0.7 V</td>
</tr>
<tr>
<td>DFMA energy</td>
<td>50 pJ</td>
<td>7.6 pJ</td>
</tr>
<tr>
<td>64b 8 KB SRAM Rd</td>
<td>14 pJ</td>
<td>2.1 pJ</td>
</tr>
<tr>
<td>Wire energy (256 bits, 10mm)</td>
<td>310 pJ</td>
<td>174 pJ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Technology</th>
<th>45 nm</th>
<th>16 nm</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM interface pin bandwidth</td>
<td>4 Gbps</td>
<td>50 Gbps</td>
</tr>
<tr>
<td>DRAM interface energy</td>
<td>20-30 pJ/bit</td>
<td>2 pJ/bit</td>
</tr>
<tr>
<td>DRAM access energy</td>
<td>8-15 pJ/bit</td>
<td>2.5 pJ/bit</td>
</tr>
</tbody>
</table>

FP Op lower bound = 4 pJ

---

Keckler [Micro 2011], Vogelsang [Micro 2010]
Memory Hierarchy

ALUs
Registers
Local

ALUs
Registers
Local

Global

PCI Express

10,000 GB/s
1,000 GB/s
100 GB/s
10 GB/s
Let’s talk about caches....
Let’s talk about DRAM
"KAVERI" GPU – GRAPHICS CORE NEXT ARCHITECTURE

47% of "Kaveri" is dedicated for GPU

- 8 compute units (512 IEEE 2008-compliant shaders)
- Masked Quad Sum of Absolute Difference (MQSAD) with 32b accumulation and saturation
- Device flat (generic) addressing support
- Precision improvement for native LOG/EXP ops to 1ULP

L2 Cache

Branch & Message Unit
Scheduler
Vector Units (4x SIMD-16)
Scalar Unit
Texture Filter Units (4)
Texture Fetch Load / Store Units (16)

Vector Registers (4x 64KB)
Local Data Share (64KB)
Scalar Registers (4KB)
L1 Cache (15KB)
Some core thoughts to keep in mind

• every access to a cache or DRAM accesses a block.

• In front of the L2 on a GPU is a structure that coalesces accesses to the same block.

  • For good L2 performance it is key that you use this

  • Easiest to use it by accessing the same block in different work items.

• The L1 is fine-grained interleaved, but the net/net conceptually is the same for you as a developer.
Temporal SIMT

Spatial SIMT (current GPUs)

32-wide datapath

Pure Temporal SIMT

1-wide

1 warp instruction = 32 threads
Thread scheduling

• There’s nothing to say. It is implementation dependent

  • Do not write code that assumes anything being scheduled or completed across workgroups.

• Within a workgroup:

  • barrier(…)

Example
A.2 High-Priority Recommendations

- To get the maximum benefit from OpenCL, focus first on finding ways to parallelize sequential code. (Section 1.1.3)
- Use the effective bandwidth of your computation as a metric when measuring performance and optimization benefits. (Section 2.2)
- Minimize data transfer between the host and the device, even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU. (Section 3.1)
- Ensure global memory accesses are coalesced whenever possible. (Section 3.2.1)
- Minimize the use of global memory. Prefer shared memory access where possible. (Section 5.2)
- Avoid different execution paths within the same warp. (Section 6.1)
- Use the -cl-mad-enable build option. (Chapter 5)
A.3 Medium-Priority Recommendations

- Judiciously use “pinned” memory for host buffers (Section 3.1.1)
- Where feasible and for applications where it is effective, overlap host – device memory transfers with device computations and asynchronous host activities (Sections 3.1.2 and 3.1.3)
- For applications where the destination of computational results is the display, use OpenCL-OpenGL or OpenCL-D3D interop.
- Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts. (Section 3.2.2.1)
- Use shared memory to avoid redundant transfers from global memory. (Section 3.2.2.2)
- To hide latency arising from register dependencies, maintain at least 25 percent occupancy on devices with compute capability 1.1 and lower, and 18.75 percent occupancy on later devices. (Section 4.3)
- The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency and facilitates coalescing. (Section 4.4)
- Use the native math library whenever speed trumps precision. (Section 5.1.4)
A.4 Low-Priority Recommendations

- For kernels with long argument lists, place some arguments into constant memory to save shared memory. (Section 3.2.2.4)

- Use shift operations to avoid expensive division and modulo calculations. (Section 5.1.1)

- Avoid automatic conversion of doubles to floats. (Section 5.1.3)

- Make it easy for the compiler to use branch predication in lieu of loops or control statements. (Section 6.2)
Some closing thoughts

- GPU’s are in a sweet-enough spot between efficiency and pain
  - FPGAs = more pain, CPUS = less efficient