GPU Architectures
A CPU Perspective

Derek Hower  AMD Research  5/21/2013
With updates by David Wood

Goals

Data Parallelism: What is it, and how to exploit it?
- Workload characteristics

Execution Models / GPU Architectures
- MIMD (SPMD), SIMD, SIMT

GPU Programming Models
- Terminology translations: CPU → AMD GPU → Nvidia GPU
- Intro to OpenCL

Modern GPU Microarchitectures
- i.e., programmable GPU pipelines, not their fixed-function predecessors

Advanced Topics: (Time permitting)
- The Limits of GPUs: What they can and cannot do
- The Future of GPUs: Where do we go from here?

Data Parallel Execution on GPUs

Data Parallelism, Programming Models, SIMT

Graphics Workloads

Streaming computation

GPU
Graphics Workloads

Streaming computation on pixels

Graphics Workloads

Identical, Streaming computation on pixels

Graphics Workloads

Identical, Independent, Streaming computation on pixels

Architecture Spelling Bee

Spill ‘Independent’
Generalize: Data Parallel Workloads

Identical, Independent computation on multiple data inputs

Data Parallelism: A MIMD Approach

Multiple Instruction Multiple Data
Split independent work over multiple processors

When work is identical (same program):
Single Program Multiple Data (SPMD) (Subcategory of MIMD)
Data Parallelism: An SPMD Approach

**Single Program Multiple Data**
Split identical, independent work over multiple processors

Program color\(\downarrow\)out = f(color\(\downarrow\)in)

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data Parallelism: A SIMD Approach

**Single Instruction Multiple Data**
Split identical, independent work over multiple execution units (lanes)
More efficient: Eliminate redundant fetch/decode

Program color\(\downarrow\)out = f(color\(\downarrow\)in)

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SIMD: A Closer Look

One Thread + Data Parallel Ops → Single PC, single register file

Program color\(\downarrow\)out = f(color\(\downarrow\)in)

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data Parallelism: A SIMT Approach

**Single Instruction Multiple Thread**
Split identical, independent work over multiple lockstep threads
Multiple Threads + Scalar Ops → One PC, Multiple register files

Program color\(\downarrow\)out = f(color\(\downarrow\)in)

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

WFD

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Terminology Headache #1

It’s common to interchange ‘SIMD’ and ‘SIMT’

Data Parallel Execution Models

<table>
<thead>
<tr>
<th>Model</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIMD/SPMD</td>
<td>Easier to program</td>
<td>Inefficient for data parallelism</td>
</tr>
<tr>
<td>SIMD/Vector</td>
<td>Can mix sequential &amp; parallel code</td>
<td>Gather/Scatter can be awkward</td>
</tr>
<tr>
<td>SIMT</td>
<td>Multiple independent threads</td>
<td>Divergence kills performance</td>
</tr>
</tbody>
</table>

Execution Model Comparison

<table>
<thead>
<tr>
<th>Example Architecture</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 SSE/AVX</td>
<td>More general supports TLP</td>
<td>Inefficient for data parallelism</td>
</tr>
<tr>
<td>Cray-1</td>
<td>Can mix sequential &amp; parallel code</td>
<td>Gather/Scatter can be awkward</td>
</tr>
<tr>
<td>GPUs</td>
<td>Easier to program Gather/Scatter operations</td>
<td>Divergence kills performance</td>
</tr>
</tbody>
</table>

GPUs and Memory

Recall: GPUs perform Streaming computation $\rightarrow$ Streaming memory access

DRAM latency: 100s of GPU cycles

How do we keep the GPU busy (hide memory latency)?
Hiding Memory Latency

Options from the CPU world:

- Need spatial/temporal locality
- Need ILP
- Multicore/Multithreading/SMT

Multicore Multithreaded SIMT

Many SIMT "threads" grouped together into GPU "Core"

SIMT threads in a group = SMT threads in a CPU core
- Unlike CPU, groups are exposed to programmers
Multiple GPU "Cores"

This is a GPU Architecture (Whew!)

GPU Component Names

- AMD/OpenCL
- Processing Element
- SIMD Unit
- Compute Unit
- GPU Device

Derek's CPU Analogy

- Lane
- Pipeline
- Core
- Device
GPU Programming Models

OpenCL

OpenCL – Open Computing Language
- Developed by Khronos Group
- Wide industry support: AMD, Apple, Qualcomm, Nvidia (begrudgingly), etc.

C++ AMP – C++ Accelerated Massive Parallelism
- Developed by Microsoft
- Much higher abstraction than CUDA/OpenCL

OpenACC – Open Accelerator
- Like OpenMP for GPUs (semi-auto-parallelize serial code)
- Much higher abstraction than CUDA/OpenCL

GPU Programming Models

CUDA – Compute Unified Device Architecture
- Developed by Nvidia – proprietary
- First serious GPGPU language/environment

OpenCL – Open Computing Language
- From makers of OpenGL
- Wide industry support: AMD, Apple, Qualcomm, Nvidia (begrudgingly), etc.

C++ AMP – C++ Accelerated Massive Parallelism
- Microsoft
- Much higher abstraction than CUDA/OpenCL

OpenACC – Open Accelerator
- Like OpenMP for GPUs (semi-auto-parallelize serial code)
- Much higher abstraction than CUDA/OpenCL

Early CPU languages were light abstractions of physical hardware
- E.g., C

Early GPU languages are light abstractions of physical hardware
- OpenCL + CUDA
OpenCL

Early CPU languages were light abstractions of physical hardware
- E.g., C
Early GPU languages are light abstractions of physical hardware
- OpenCL + CUDA

GPU Architecture

NDRange

N-Dimensional (N = 1, 2, or 3) index space
- Partitioned into workgroups, wavefronts, and work-items

Kernel

Run an NDRange on a kernel (i.e., a function)
Same kernel executes for each work-item
- Smells like MIMD/SPMD
Kernel

Run an NDRange on a kernel (i.e., a function)
- Same kernel executes for each work-item
  - Smells like MIMD/SPMD… but beware, it’s not!

OpenCL Code

```c
__kernel void flip_and_recolor(__global float3 **in_image,
                               __global float3 **out_image,
                               int img_dim_x, int img_dim_y)
{
    int x = get_global_id(1); // get work-item id in dim 1
    int y = get_global_id(2); // get work-item id in dim 2
    out_image[(img_dim_x - x)][(img_dim_y - y)] =
        recolor(in_image[x][y]);
}
```

GPU Hardware Overview
Compute Unit – A GPU Core

Compute Unit (CU) – Runs Workgroups
- Contains 4 SIMT Units
- Picks one SIMT Unit per cycle for scheduling

SIMT Unit – Runs Workfronts
- Each SIMT Unit has 10 wavefront instruction buffer
- Takes 4 cycles to execute one wavefront

10 Wavefront x 4 SIMT Units = 40 Active Wavefronts / CU
64 work-items / wavefront x 40 active wavefronts = 2560 Active Work-items / CU

SIMT Unit – A GPU Pipeline
Like a wide CPU pipeline – except one fetch for entire width
16-wide physical ALU
- Executes 64-wavefront over 4 cycles. Why??

64KB register state / SIMT Unit
- Compare to x86 (Bulldozer): ~1KB of physical register file state (~1/64 size)

Address Coalescing Unit
- A key to good memory performance

Address Coalescing Unit

Compute Unit Timing Diagram
On average: fetch & commit one wavefront / cycle

SIMT0 SIMT1 SIMT2 SIMT3

Time

Address Coalescing
Wavefront: Issue 64 memory requests

NDRange
Workgroup
Workgroup
Address Coalescing

Wavefront: Issue 64 memory requests

Common case:
- work-items in same wavefront touch same cache block

Coalescing:
- Merge many work-items requests into single cache block request

Important for performance:
- Reduces bandwidth to DRAM

GPU Memory

GPUs have caches.

Not Your CPU’s Cache

By the numbers: Bulldozer – FX-8170 vs. GCN – Radeon HD 7970

<table>
<thead>
<tr>
<th></th>
<th>CPU (Bulldozer)</th>
<th>GPU (GCN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 data cache capacity</td>
<td>16KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>Active threads [work-items] sharing L1 D Cache</td>
<td>1</td>
<td>2560</td>
</tr>
<tr>
<td>L1 dcache capacity / thread</td>
<td>16KB</td>
<td>6.4 bytes</td>
</tr>
<tr>
<td>Last level cache (LLC) capacity</td>
<td>8MB</td>
<td>768KB</td>
</tr>
<tr>
<td>Active threads [work-items] sharing LLC</td>
<td>8</td>
<td>81,920</td>
</tr>
<tr>
<td>LLC capacity / thread</td>
<td>1MB</td>
<td>9.6 bytes</td>
</tr>
</tbody>
</table>

GPU Caches

Maximize throughput, not hide latency
- Not there for temporal locality
- Not much spatial locality either, since coalescing logic catches most of that

L1 Cache: Coalesce requests to same cache block by different work-items
- i.e., streaming thread locality?
- Keep block around just long enough for each work-item to hit once
- Ultimate goal: Reduce bandwidth to DRAM

L2 Cache: DRAM staging buffer + some instruction reuse
- Ultimate goal: Tolerate spikes in DRAM bandwidth

If there is any temporal locality:
- Use local memory (scratchpad)
### Scratchpad Memory

GPUs have scratchpads (Local Memory)
- Separate address space
- Managed by software
- Rename address
- Manage capacity – manual fill/eviction

Allocated to a workgroup
- i.e., shared by wavefronts in workgroup

### Example System: Radeon HD 7970

High-end part

- **32 Compute Units:**
  - 81,920 Active work-items
  - 32 CUs * 4 SMT Units * 16 ALUs * 2048 Max FP ops/cycle
  - 264 GB/s Max memory bandwidth
- **925 MHz engine clock**
  - 3.79 TFLOPS single precision (accounting trickery: FMA)
- **210W Max Power (Chip)**
  - >350W Max Power (card)
  - 100W idle power (card)

### Radeon HD 7990 - Cooking

Two 7970s on one card:
- 375W (AMD Official) – 450W (OEM)

### A Rose by Any Other Name...
Terminology Headaches #2-5

<table>
<thead>
<tr>
<th>Nvidia/CUDA</th>
<th>AMD/OpenCL</th>
<th>Derek’s CPU Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Processor</td>
<td>Processing Element</td>
<td>Lane</td>
</tr>
<tr>
<td>CUDA Core</td>
<td>SIMD Unit</td>
<td>Pipeline</td>
</tr>
<tr>
<td>Streaming Multiprocessor</td>
<td>Compute Unit</td>
<td>Core</td>
</tr>
<tr>
<td>GPU “Core”</td>
<td>GPU Device</td>
<td>Device</td>
</tr>
</tbody>
</table>

Terminology Headaches #6-9

<table>
<thead>
<tr>
<th>CUDA/Nvidia</th>
<th>OpenCL/AMD</th>
<th>Henne&amp;Patt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Wave-item</td>
<td>Sequence of SIMD Lane Operations</td>
</tr>
<tr>
<td>Warp</td>
<td>Wavefront</td>
<td>Thread of SIMD Instructions</td>
</tr>
<tr>
<td>Block</td>
<td>Workgroup</td>
<td>Body of vectorized loop</td>
</tr>
<tr>
<td>Grid</td>
<td>NDRange</td>
<td>Vectorized loop</td>
</tr>
</tbody>
</table>

Terminology Headache #10

GPUs have scratchpads (Local Memory)
- Separate address space
- Managed by software
- Requires address
- Manage capacity – manual fill/eviction

Allocated to a workgroup
- i.e., shared by wavefronts in workgroup

Nvidia calls ‘Local Memory’ ‘Shared Memory’. AMD sometimes calls it ‘Group Memory’.

Recap

Data Parallelism: Identical, Independent work over multiple data inputs
- GPU version: Add streaming access pattern

Data Parallel Execution Models: MIMD, SIMD, SIMT

GPU Execution Model: Multicore Multithreaded SIMT

OpenCL Programming Model
- NDRange over workgroup/wavefront

Modern GPU Microarchitecture: AMD Graphics Core Next (GCN)
- Compute Unit (“GPU Core”): 4 SIMT Units
- SIMT Unit (“GPU Pipeline”): 16-wide ALU pipe (16x4 execution)
- Memory: designed to stream

GPUs: Great for data parallelism. Bad for everything else.
Advanced Topics

GPU Limitations, Future of GPGPU

Choose Your Own Adventure!

SIMT Control Flow & Branch Divergence
Memory Divergence
When GPUs talk
- Wavefront communication
- GPU “coherence”
- GPU consistency
Future of GPUs: What’s next?

SIMT Control Flow
Consider SIMT conditional branch:
- One PC
- Multiple data (i.e., multiple conditions)

```
if (x <= 0)
  y = 0;
else
  y = x;
```

Branching through predication
- Active lane: commit result
- Inactive lane: throw away result

- All lanes active at start: 1111
  - Branch → set execution mask: 1000
  - Else → invert execution mask: 0111
  - Converge → reset execution mask: 1111

- Work items in wavefront run in lockstep
  - Don’t all have to commit

SIMT Control Flow

Active lane: commit result
Inactive lane: throw away result

All lanes active at start: 1111
Branch → set execution mask: 1000
Else → invert execution mask: 0111
Converge → reset execution mask: 1111
SIMT Control Flow

Work-items in wavefront run in lockstep
- Don't all have to commit

Branching through predication

Branch divergence

if (x <= 0)
  y = 0;
else
  y = x;

Branch divergence

Beware!

Divergence isn't just a performance problem:

```c
__global int lock = 0;
void mutex_lock(...)
{
  ...
  // acquire lock
  while (test&set(lock, 1) == false) {
    // spin
  }
  return;
}
```

Branch Divergence

When control flow diverges, all lanes take all paths

Divergence Kills Performance

Beware!

Divergence isn't just a performance problem:

```c
__global int lock = 0;
void mutex_lock()
{
  ...
  // acquire lock
  while (test&set(lock, 1) == false) {
    // spin
  }
  return;
}
```

Deadlock: Work-items can’t enter mutex together!
Memory Bandwidth

**✓ -- Parallel Access**

Memory Bandwidth

**✗ -- Sequential Access**

Memory Bandwidth

**✗ -- Sequential Access**

Memory Bandwidth

**Memory divergence**

- **✗ -- Sequential Access**

Memory Divergence

- One work-item stalls → entire wavefront must stall
- Cause: Bank conflicts, cache misses
- Data layout & partitioning is important
Memory Divergence

One work-item stalls \(\rightarrow\) entire wavefront must stall
- Cause: Bank conflicts, cache misses
Data layout & partitioning is important

Divergence Kills Performance

Communication and Synchronization

Work-items can communicate with:
- Work-items in same wavefront
- Work-items in different wavefront, same workgroup (local)
- Work-items in different wavefront, different workgroup (global)
- No special sync needed…they are lockstep!

GPU Consistency Models

Very weak guarantee:
- Program order respected within single work-item
- All other bets are off

Safety net:
- Fence – “make sure all previous accesses are visible before proceeding”
- Built-in barriers are also fences

A wrench:
- GPU fences are scoped – only apply to subset of work-items in system
- E.g., local barrier

Take-away: Area of active research

GPU Coherence?

Notice: GPU consistency model does not require coherence
- I.e., Single Writer, Multiple Reader

Marketing claims they are coherent...

GPU “Coherence”:
- Nvidia: disable private caches
- AMD: flush/invalid entire cache at fences
GPU Architecture Research

- Blending with CPU architecture:
  - Dynamic scheduling / dynamic wavefront re-org
  - Work items have more locality than we think

- Tighter integration with CPU on SOC:
  - Fast kernel launch
  - Explicit fine-grained parallel region: Remember Amdahl’s law
  - Common shared memory

- Reliability:
  - Historically: Who notices a bad pixel?
  - Future: GPU compute demands correctness

- Power:
  - Mobile, mobile mobile!!

Computer Economics 101

- GPU Compute is cool + gaining steam, but...
  - Is a $0 billion dollar industry (to quote Mark Hill)

- GPU design priorities:
  1. Graphics
  2. Graphics
  3. GPU Compute

- Moral of the story:
  - GPU won’t become a CPU (nor should it)