GPU Computing Architecture

HiPEAC Summer School, July 2015
Tor M. Aamodt
aamodt@ece.ubc.ca
University of British Columbia
What is a GPU?

- GPU = Graphics Processing Unit
  - Accelerator for raster based graphics (OpenGL, DirectX)
  - Highly programmable (Turing complete)
  - Commodity hardware
  - 100’s of ALUs; 10’s of 1000s of concurrent threads
The GPU is Ubiquitous

THE FUTURE BELONGS TO THE APU:
BETTER GRAPHICS, EFFICIENCY AND COMPUTE

"SANDY BRIDGE"
17% GPU*

"IVY BRIDGE"
27% GPU*

"HASWELL"
31% GPU*

(Estimated)

2014 AMD A-SERIES/CODENAMED "KAVERI"
47% GPU

DELIVERS BREAKTHROUGHS IN APU-BASED:

- Compute
  - (OpenCL™, Direct Compute)
- Gaming
  - (DirectX®, OpenGL, Mantle)
- Experiences
  - (Audio, Ultra HD, Devices, New Interactivity)
“Early” GPU History

– 1981: IBM PC Monochrome Display Adapter (2D)
– 1996: 3D graphics (e.g., 3dfx Voodoo)
– 1999: register combiner (NVIDIA GeForce 256)
– 2001: programmable shaders (NVIDIA GeForce 3)
– 2002: floating-point (ATI Radeon 9700)
– 2005: unified shaders (ATI R520 in Xbox 360)
– 2006: compute (NVIDIA GeForce 8800)
GPU: The Life of a Triangle

process commands

Vertex Processing

Host / Front End / Vertex Fetch

transform vertices to screen-space

Primitive Assembly, Setup

generate per-triangle equations

Rasterize & Zcull

generate pixels, delete pixels that cannot be seen

Pixel Shader

determine the color, transparencies and depth of the pixel

Texture

do final hidden surface test, blend and write out color and new depth

Pixel Engines (ROP)

Frame Buffer Controller

[David Kirk / Wen-mei Hwu]
pixel color result of running “shader” program
Why use a GPU for computing?

- GPU uses larger fraction of silicon for computation than CPU.
- At peak performance GPU uses order of magnitude less energy per operation than CPU.

CPU
2nJ/op

Rewrite Application

GPU
200pJ/op

Order of Magnitude More Energy Efficient

However.... Application must perform well
GPU uses larger fraction of silicon for computation than CPU?
Growing Interest in GPGPU

• Supercomputing – Green500.org Nov 2014

  “the top three slots of the Green500 were powered by three different accelerators with number one, L-CSC, being powered by AMD FirePro™ S9150 GPUs; number two, Suiren, powered by PEZY-SC many-core accelerators; and number three, TSUBAME-KFC, powered by NVIDIA K20x GPUs. Beyond these top three, the next 20 supercomputers were also accelerator-based.”

• Deep Belief Networks map very well to GPUs (e.g., Google keynote at 2015 GPU Tech Conf.)

  http://blogs.nvidia.com/blog/2015/03/18/google-gpu/
  http://www.ustream.tv/recorded/60071572
GPGPUs vs. Vector Processors

- Similarities at hardware level between GPU and vector processors.

- (I like to argue) SIMT programming model moves hardest parallelism detection problem from compiler to programmer.
Course Learning Objectives

After course you should be able to:

1. Explain motivation for investigating novel GPU-like computing architectures
2. Understand basic CUDA / PTX programs
3. Describe features of a generic GPU architecture representative of contemporary GPGPUs
4. Describe selected research on improving GPU computing programming models and hardware efficiency
Further Reading?

The following title is under development:


Other resources (primarily research papers) will be mentioned throughout the lectures.
Course Outline

• Part 1: Introduction to GPGPU Programming Model
• Part 2: Generic GPGPU Architecture
• Part 3: Research Directions
  – Mitigating SIMT Control Divergence
  – Mitigating High GPGPU Memory Bandwidth Demands
  – Coherent Memory for Accelerators
  – Easier Programming with Synchronization
Part 1: Introduction to GPGPU Programming Model
GPGPU Programming Resources

• 9 week MOOC covering CUDA, OpenCL, C++AMP and OpenACC
  https://www.coursera.org/course/hetero

• Kirk and Hwu, Programming Massively Parallel Processors, Morgan Kaufmann, 2nd edition, 2014 (NOTE: 2nd edition includes coverage of OpenCL, C++AMP, and OpenACC)
GPU Compute Programming Model

How is this system programmed (today)?
GPGPU Programming Model

- CPU “Off-load” parallel kernels to GPU
  - Transfer data to GPU memory
  - GPU HW spawns threads
  - Need to transfer result data back to CPU main memory
CUDA/OpenCL Threading Model

CPU spawns fork-join style “grid” of parallel threads

- Spawns more threads than GPU can run (some may wait)
- Organize threads into “blocks” (up to 1024 threads per block)
- Threads can communicate/synchronize with other threads in block
- Threads/Blocks have an identifier (can be 1, 2 or 3 dimensional)
- Each kernel spawns a “grid” containing 1 or more thread blocks.
- **Motivation:** *Write parallel software once and run on future hardware*
SIMT Execution Model

• Programmers sees **MIMD threads** (scalar)
• GPU bundles threads into **warps** (wavefronts) and runs them in lockstep on **SIMD hardware**
• An NVIDIA warp groups 32 consecutive threads together (AMD wavefronts group 64 threads together)
• Aside: Why “Warp”? In the textile industry, the term “warp” refers to “the threads stretched lengthwise in a loom to be crossed by the weft” [Oxford Dictionary].
• Jacquard Loom => Babbage’s Analytical Engine => ... => GPU.

[https://en.wikipedia.org/wiki/Warp_and_woof]
SIMT Execution Model

• Challenge: How to handle branch operations when different threads in a warp follow a different path through program?
• Solution: Serialize different paths.

foo[] = {4,8,12,16};

A: v = foo[threadIdx.x];

B: if (v < 10)

C: v = 0;

else

D: v = 10;

E: w = bar[threadIdx.x]+v;
CUDA Syntax Extensions

• Declaration specifiers
  __global__ void foo(...); // kernel entry point (runs on GPU)
  __device__ void bar(...); // function callable from a GPU thread

• Syntax for kernel launch
  foo<<<500, 128>>>(...); // 500 thread blocks, 128 threads each

• Built in variables for thread identification
  dim3 threadIdx; dim3 blockIdx; dim3 blockDim;
Example: Original C Code

```c
void saxpy_serial(int n, float a, float *x, float *y) {
   for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}

int main() {
   // omitted: allocate and initialize memory
   saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY kernel
   // omitted: using result
}
```
__global__ void saxpy(int n, float a, float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i<n) y[i]=a*x[i]+y[i];
}

int main() {
    // omitted: allocate and initialize memory
    int nbblocks = (n + 255) / 256;

cudaMalloc((void**) &d_x, n);
cudaMalloc((void**) &d_y, n);
cudaMemcpy(d_x,h_x,n*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_y,h_y,n*sizeof(float),cudaMemcpyHostToDevice);
saxpy<<<nbblocks, 256>>>(n, 2.0, d_x, d_y);
cudaMemcpy(h_y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
    // omitted: using result
}
__kernel void saxpy(int n, float a, __global float *x, __global float *y) {
    int i = get_global_id(0);
    if(i<n) y[i]=a*x[i]+y[i];
}

int main() {
    // omitted: allocate and initialize memory on host, variable declarations

    int nblocks = (n + 255) / 256;
    int blocksize = 256;

    clGetPlatformIDs(1, &cpPlatform, NULL);
    clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
    cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
    cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr1);
    dx = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float) * n, NULL, &ciErr1);
    dy = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE, sizeof(cl_float) * n, NULL, &ciErr1);

    // omitted: loading program into char string cSourceCL
    cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char **)cSourceCL, &szKernelLength, &ciErr1);
    clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);
    ckKernel = clCreateKernel(cpProgram, "saxpy_serial", &ciErr1);

    clSetKernelArg(ckKernel, 0, sizeof(cl_int), (void*)&n);
    clSetKernelArg(ckKernel, 1, sizeof(cl_float), (void*)&a);
    clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&dx);
    clSetKernelArg(ckKernel, 3, sizeof(cl_mem), (void*)&dy);

    clEnqueueWriteBuffer(cqCommandQueue, dx, CL_FALSE, 0, sizeof(cl_float) * n, x, 0, NULL, NULL);
    clEnqueueWriteBuffer(cqCommandQueue, dy, CL_FALSE, 0, sizeof(cl_float) * n, y, 0, NULL, NULL);
    clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL, &nblocks, &blocksize, 0, NULL, NULL);
    clEnqueueReadBuffer(cqCommandQueue, dy, CL_TRUE, 0, sizeof(cl_float) * n, y, 0, NULL, NULL);

    // omitted: using result
C++AMP Example Code

```cpp
#include <amp.h>
using namespace concurrency;

int main() {
    // omitted: allocation and initialization of y and x
    array_view<int> xv(n, x);
    array_view<int> yv(n, y);
    parallel_for_each(yv.get_extent(), [=](index<1> i) restrict(amp) {
        yv[i] = a * xv[i] + yv[i];
    });
    yv.synchronize();
    // omitted: using result
}
```

Runs on GPU
void saxpy_serial(int n, float a, float *x, float *y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
Review: Memory

• E.g., use to save state between steps in a computation.

• Each memory location has an associated *address* which identifies the location. The location contains a value:

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xFF</td>
</tr>
<tr>
<td>1</td>
<td>0x42</td>
</tr>
<tr>
<td>2</td>
<td>0x00</td>
</tr>
<tr>
<td>3</td>
<td>0x01</td>
</tr>
</tbody>
</table>

Example: Memory with 4 one byte locations. Location with address 1 contains value 0x42.
GPU Memory Address Spaces

• GPU has three *address spaces* to support increasing visibility of data between threads: local, shared, global

• In addition two more (read-only) address spaces: Constant and texture.
Local (Private) Address Space

Each thread has own “local memory” (CUDA) “private memory” (OpenCL).

Note: Location at address 100 for thread 0 is different from location at address 100 for thread 1.

Contains local variables private to a thread.
Global Address Spaces

Each thread in the different thread blocks (even from different kernels) can access a region called “global memory” (CUDA/OpenCL).

Commonly in GPGPU workloads threads write their own portion of global memory. Avoids need for synchronization—slow; also unpredictable thread block scheduling.
History of “global memory”

• Prior to NVIDIA GeForce 8800 and CUDA 1.0, access to memory was through texture reads and raster operations for writing.

• Problem: Address of memory access was highly constrained function of thread ID.

• CUDA 1.0 enabled access to arbitrary memory location in a flat memory space called “global”
Example: Transpose (CUDA SDK)

```c
__global__ void transposeNaive(float *odata, float* idata, int width, int height)
{
    int xIndex = (blockIdx.x * TILE_DIM) + threadIdx.x;  // TILE_DIM = 16
    int yIndex = (blockIdx.y * TILE_DIM) + threadIdx.y;

    int index_in  = xIndex + (width * yIndex);
    int index_out = yIndex + (height * xIndex);
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {  // BLOCK_ROWS = 16
        odata[index_out+i] = idata[index_in+(i*width)];
    }
}
```

NOTE: “xIndex”, “yIndex”, “index_in”, “index_out”, and “i” are in local memory (local variables are register allocated but stack lives in local memory)

“odata” and “idata” are pointers to global memory (both allocated using calls to cudaMalloc -- not shown above)
“Coalescing” global accesses

- **Not** same as CPU write combining/buffering:
- Aligned accesses request single 128B cache blk

\[
\text{ld.global } r1,0(r2)
\]

- **Memory Divergence:**

\[
\text{ld.global } r1,0(r2)
\]
Example: Transpose (CUDA SDK)

```c
__global__ void transposeNaive(float *odata, float* idata, int width, int height)
{
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

    int index_in  = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
        odata[index_out+i] = idata[index_in+i*width];
    }
}
```

Assume height=16 and consider i=0:

Thread x=0,y=0 has xIndex=0, yIndex=0 so accesses odata[0]
Thread x=1,y=0 has xIndex=1, yIndex=0 so accesses odata[16]

Write to global memory highlighted above is not “coalesced”.
Redundant Global Memory Accesses

```c
__global__ void matrixMul (float *C, float *A, float *B, int N)
{
    int xIndex = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int yIndex = blockIdx.y * BLOCK_SIZE + threadIdx.y;

    float sum = 0;

    for (int k=0; k<N; i++)
        sum += A[yIndex][k] * B[k][xIndex];

    C[yIndex][xIndex] = sum;
}
```

E.g., both thread x=0, y=0 and thread x=32, y=0 access A[0][0] potentially causing two accesses to off-chip DRAM. In general, each element of A and B is redundantly fetched O(N) times.
Tiled Multiply Using Thread Blocks

- One block computes one square sub-matrix $P_{sub}$ of size BLOCK_SIZE
- One thread computes one element of $P_{sub}$
- Assume that the dimensions of M and N are multiples of BLOCK_SIZE and square shape
History of “shared memory”

• Prior to NVIDIA GeForce 8800 and CUDA 1.0, threads could not communicate with each other through on-chip memory.

• “Solution”: small (16-48KB) programmer managed scratchpad memory shared between threads within a thread block.
Shared (Local) Address Space

Each thread in the same thread block (work group) can access a memory region called “shared memory” (CUDA) “local memory” (OpenCL).

Shared memory address space is limited in size (16 to 48 KB).

Used as a software managed “cache” to avoid off-chip memory accesses.

Synchronize threads in a thread block using __syncthreads();
Optimizing Transpose for Coalescing

**Step 1:** Read block of data into shared memory

<table>
<thead>
<tr>
<th>idata</th>
<th>1 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 4</td>
</tr>
</tbody>
</table>

**Step 2:** Copy from shared memory into global memory using coalesce write

<table>
<thead>
<tr>
<th>odata</th>
<th>1 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 4</td>
</tr>
</tbody>
</table>
Optimizing Transpose for Coalescing

```c
__global__ void transposeCoalesced(float *odata, float *idata, int width, int height) {
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int xIndex = (blockIdx.x * TILE_DIM) + threadIdx.x;
    int yIndex = (blockIdx.y * TILE_DIM) + threadIdx.y;
    int index_in = xIndex + (width * yIndex);

    xIndex = (blockIdx.y * TILE_DIM) + threadIdx.x;
    yIndex = (blockIdx.x * TILE_DIM) + threadIdx.y;
    int index_out = xIndex + (yIndex*height);

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
        tile[threadIdx.y+i][threadIdx.x] = idata[index_in+(i*width)];
    }

    __syncthreads(); // wait for all threads in block to finish above for loop

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
        odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
    }
}
```

GOOD: Coalesced write
BAD: Shared memory bank conflicts
Review: Bank Conflicts

- To increase bandwidth common to organize memory into multiple banks.
- Independent accesses to different banks can proceed in parallel.

Example 1: Read 0, Read 1 (can proceed in parallel)

Example 2: Read 0, Read 3 (can proceed in parallel)

Example 3: Read 0, Read 2 (bank conflict)
Shared Memory Bank Conflicts

__shared__ int A[BSIZE];

...

A[threadIdx.x] = ... // no conflicts
Shared Memory Bank Conflicts

```c
__shared__ int A[BSIZE];
...
A[2*threadIdx.x] = // 2-way conflict
```
Optimizing Transpose for Coalescing

**Step 1:**  Read block of data into shared memory

![Diagram showing data transpose]

**Step 2:**  Copy from shared memory into global memory using coalesce write

Problem: Access two locations in same shared memory bank.
Eliminate Bank Conflicts

```c
__global__ void transposeNoBankConflicts (float *odata, float *idata, int width, int height)
{

    __shared__ float tile[TILE_DIM][TILE_DIM+1];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*width;

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*height;

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
        tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
        odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
    }
}
```
Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory

Step 2: Copy from shared memory into global memory using coalesce write
CUDA Streams

• CUDA (and OpenCL) provide the capability to overlap computation on GPU with memory transfers using “Streams” (Command Queues)
• A Stream orders a sequence of kernels and memory copy “operations”.
• Operations in one stream can overlap with operations in a different stream.
How Can Streams Help?

Serial:
- cudaMemcpy(H2D)
- kernel
- cudaMemcpy(D2H)
  - GPU idle
  - GPU busy
  - GPU idle

Streams:
- cudaMemcpy(H2D)
- K0 DH0
- K1 DH1
- K2 DH2
  - Savings
  - Time

CUDA Streams

cudaStream_t streams[3];
for(i=0; i<3; i++)
    cudaStreamCreate(&streams[i]); // initialize streams

for(i=0; i<3; i++) {
    cudaMemcpyAsync(pD+i*size,pH+i*size,size,
                    cudaMemcpyHostToDevice,stream[i]);  // H2D
    MyKernel<<<grid,block,0,stream[i]>>>(pD+i,size); // compute
    cudaMemcpyAsync(pD+i*size,pH+i*size,size,
                    cudaMemcpyDeviceToHost,stream[i]); // D2H
}
Recent Features in CUDA

• Dynamic Parallelism (CUDA 5): Launch kernels from within a kernel. Reduce work for e.g., adaptive mesh refinement.

• Unified Memory (CUDA 6): Avoid need for explicit memory copies between CPU and GPU

See also, Gelado, et al. ASPLOS 2010.

GPU Instruction Set Architecture (ISA)

- NVIDIA defines a virtual ISA, called “PTX” (Parallel Thread eXecution)
- More recently, Heterogeneous System Architecture (HSA) Foundation (AMD, ARM, Imagination, Mediatek, Samsung, Qualcomm, TI) defined the HSAIL virtual ISA.
- PTX is Reduced Instruction Set Architecture (e.g., load/store architecture)
- Virtual: infinite set of registers (much like a compiler intermediate representation)
- PTX translated to hardware ISA by backend compiler (“ptxas”). Either at compile time (nvcc) or at runtime (GPU driver).
Some Example PTX Syntax

- Registers declared with a type:
  .reg .pred p, q, r;
  .reg .u16 r1, r2;
  .reg .f64 f1, f2;

- ALU operations
  add.u32 x, y, z; // x = y + z
  mad.lo.s32 d, a, b, c; // d = a*b + c

- Memory operations:
  ld.global.f32 f, [a];
  ld.shared.u32 g, [b];
  st.local.f64 [c], h

- Compare and branch operations:
  setp.eq.f32 p, y, 0; // is y equal to zero?
  @p bra L1 // branch to L1 if y equal to zero
Part 2: Generic GPGPU Architecture
Extra resources

GPGPU-Sim 3.x Manual
GPU Microarchitecture Overview

Single-Instruction, Multiple-Threads

SIMT Core Cluster
SIMT Core
SIMT Core

SIMT Core Cluster
SIMT Core
SIMT Core

SIMT Core Cluster
SIMT Core
SIMT Core

Interconnection Network

Memory Partition

Memory Partition

Memory Partition

GDDR5
GDDR5
GDDR5

Off-chip DRAM
GPU Microarchitecture

• Companies tight lipped about details of GPU microarchitecture.

• Several reasons:
  – Competitive advantage
  – Fear of being sued by “non-practicing entities”
  – The people that know the details too busy building the next chip

• Model described next, embodied in GPGPU-Sim, developed from: white papers, programming manuals, IEEE Micro articles, patents.
GPGPU-Sim v3.x w/ SASS

HW - GPGPU-Sim Comparison

Correlation
~0.976
GPU Microarchitecture Overview

- **SIMT Core Cluster**
  - SIMT Core
  - SIMT Core

- **Interconnection Network**

- **Memory Partition**
  - GDDR3/GDDR5
  - GDDR3/GDDR5
  - Off-chip DRAM
  - GDDR3/GDDR5
Inside a SIMT Core

- SIMT front end / SIMD backend
- Fine-grained multithreading
  - Interleave warp execution to hide latency
  - Register values of all threads stays in core
Inside an “NVIDIA-style” SIMT Core

- Three decoupled warp schedulers
- Scoreboard
- Large register file
- Multiple SIMD functional units
Fetch + Decode

- Arbitrate the I-cache among warps
  - Cache miss handled by fetching again later
- Fetched instruction is decoded and then stored in the I-Buffer
  - 1 or more entries / warp
  - Only warp with vacant entries are considered in fetch
Instruction Issue

• Select a warp and issue an instruction from its I-Buffer for execution
  – Scheduling: Greedy-Then-Oldest (GTO)
    • run a warp until it stalls (greedy), then pick the oldest warp to run next
  – GT200/later Fermi/Kepler: Allow dual issue (superscalar)
  – To avoid stalling pipeline might keep instruction in I-buffer until know it can complete (replay)
Review: **In-order Scoreboard**

- **Scoreboard**: a bit-array, 1-bit for each register
  - If the bit is *not* set: the register has valid data
  - If the bit is set: the register has stale data
    i.e., some outstanding instruction is going to change it

- **Issue in-order**: $RD \leftarrow Fn (RS, RT)$
  - If $SB[RS]$ or $SB[RT]$ is set $\rightarrow$ RAW, stall
  - If $SB[RD]$ is set $\rightarrow$ WAW, stall
  - Else, dispatch to FU ($Fn$) and set $SB[RD]$

- **Complete out-of-order**
  - Update $GPR[RD]$, clear $SB[RD]$
In-Order Scoreboard for GPUs?

- **Problem 1**: 32 warps, each with up to 128 (vector) registers per warp means scoreboard is 4096 bits.
- **Problem 2**: Warps waiting in I-buffer needs to have dependency updated every cycle.

**Solution?**
- Flag instructions with hazards as *not ready* in I-Buffer so not considered by scheduler
- Track up to 6 registers per warp (out of 128)
- I-buffer 6-entry bitvector: 1b per register dependency
- Lookup source operands, set bitvector in I-buffer. As results written per warp, clear corresponding bit
Example

Code

\[\text{ld } r7 \leftarrow [r0]\]
\[\text{mul } r6 \leftarrow r2, r5\]
\[\text{add } r8 \leftarrow r6, r7\]

Scoreboard

<table>
<thead>
<tr>
<th>Index 0</th>
<th>Index 1</th>
<th>Index 2</th>
<th>Index 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 0</td>
<td>-</td>
<td>-</td>
<td>r8</td>
</tr>
<tr>
<td>Warp 1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Instruction Buffer

<table>
<thead>
<tr>
<th>i0</th>
<th>i1</th>
<th>i2</th>
<th>i3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\text{add } r8, r6, r7 \quad 0 \quad 0 \quad 0 \quad 0\]
SIMT Using a Hardware Stack

Stack approach invented at Lucasfilm, Ltd in early 1980’s

Version here from [Fung et al., MICRO 2007]

Stack

<table>
<thead>
<tr>
<th></th>
<th>Reconv. PC</th>
<th>Next PC</th>
<th>Active Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOS</td>
<td>-</td>
<td>E</td>
<td>1111</td>
</tr>
<tr>
<td>TOS</td>
<td>E</td>
<td>D</td>
<td>0110</td>
</tr>
<tr>
<td>TOS</td>
<td>E</td>
<td>E</td>
<td>1001</td>
</tr>
</tbody>
</table>

Thread Warp

<table>
<thead>
<tr>
<th>Thread</th>
<th>Thread</th>
<th>Thread</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Common PC

SIMT = SIMD Execution of Scalar Threads
SIMT Notes

• Execution mask stack implemented with special instructions to push/pop. descriptions can be found in AMD ISA manual and NVIDIA patents.

• In practice augment stack with predication (lower overhead).
SIMT outside of GPUs?

- ARM Research looking at SIMT-ized ARM ISA.

- Intel MIC implements SIMT on top of vector hardware via compiler (ISPC)

- Possibly other industry players in future
Register File

- 32 warps, 32 threads per warp, 16 x 32-bit registers per thread = 64KB register file.
- Need “4 ports” (e.g., FMA) greatly increase area.
- Alternative: banked single ported register file. How to avoid bank conflicts?
Banked Register File

Strawman microarchitecture:

Register layout:

<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>w1:r4</td>
<td>w1:r5</td>
<td>w1:r6</td>
<td>w1:r7</td>
</tr>
<tr>
<td>w1:r0</td>
<td>w1:r1</td>
<td>w1:r2</td>
<td>w1:r3</td>
</tr>
<tr>
<td>w0:r4</td>
<td>w0:r5</td>
<td>w0:r6</td>
<td>w0:r7</td>
</tr>
<tr>
<td>w0:r0</td>
<td>w0:r1</td>
<td>w0:r2</td>
<td>w0:r3</td>
</tr>
</tbody>
</table>
Register Bank Conflicts

<table>
<thead>
<tr>
<th>i1: mad r2, r5, r4, r6</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2: add r5, r5, r1</td>
</tr>
</tbody>
</table>

- warp 0, instruction 2 has two source operands in bank 1: takes two cycles to read.
- Also, warp 1 instruction 2 is same and is also stalled.
- Can use warp ID as part of register layout to help.
Term “Operand Collector” appears in figure in NVIDIA Fermi Whitepaper

Operand Collector Architecture (US Patent: 7834881)
  – Interleave operand fetch from different threads to achieve full utilization
Operand Collector (1)

- Issue instruction to collector unit.
- Collector unit similar to reservation station in tomasulo’s algorithm.
- Stores source register identifiers.
- Arbiter selects operand accesses that do not conflict on a given cycle.
- Arbiter needs to also consider writeback (or need read+write port)
Operand Collector (2)

- Combining swizzling and access scheduling can give up to ~2x improvement in throughput.
• SIMT processing often includes redundant computation across threads.

thread 0...31:
for( i=0; i < runtime_constant_N; i++ {
    /* do something with “i” */
}

AMD Southern Islands SIMT-Core

ISA visible scalar unit executes computation identical across SIMT threads in a wavefront
Example

```c
float fn0(float a, float b) {
    if (a > b)
        return (a * a - b);
    else
        return (b * b - a);
}
```

// Registers r0 contains “a”, r1 contains “b”
// Value is returned in r2
```
    v cmp gt f32 r0, r1  // a > b
    s mov b64 s0, exec  // Save current exec mask
    s and b64 exec, vcc, exec  // Do “if”
    s cbranch vccz label0  // Branch if all lanes fail
    v mul f32 r2, r0, r0  // result = a * a
    v sub f32 r2, r2, r1  // result = result - b
label0:
    s not b64 exec, exec  // Do “else”
    s and b64 exec, s0, exec  // Do “else”
    s cbranch execz label1  // Branch if all lanes fail
    v mul f32 r2, r1, r1  // result = b * b
    v sub f32 r2, r2, r0  // result = result - a
label1:
    s mov b64 exec, s0  // Restore exec mask
```

[Southern Islands Series Instruction Set Architecture, Aug. 2012]
Southern Islands SIMT Stack?

- Instructions: S_CBRANCH_*_FORK; S_CBRANCH_JOIN
- Use for arbitrary (e.g., irreducible) control flow
- 3-bit control stack pointer
- Six 128-bit stack entries; stored in scalar general purpose registers holding \{exec[63:0], PC[47:2]\}
- S_CBRANCH_*_FORK executes path with fewer active threads first
A Modern GPU: Nvidia GTX 1080
<table>
<thead>
<tr>
<th>Feature</th>
<th>GTX 1080</th>
<th>GTX 1070</th>
<th>GTX 980</th>
<th>GTX 970</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Cores</td>
<td>2560</td>
<td>1920</td>
<td>2048</td>
<td>1664</td>
</tr>
<tr>
<td>Texture Units</td>
<td>160</td>
<td>120</td>
<td>128</td>
<td>104</td>
</tr>
<tr>
<td>ROPs</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>56</td>
</tr>
<tr>
<td>Core Clock</td>
<td>1607MHz</td>
<td>1506MHz</td>
<td>1126MHz</td>
<td>1050MHz</td>
</tr>
<tr>
<td>Boost Clock</td>
<td>1733MHz</td>
<td>1683MHz</td>
<td>1216MHz</td>
<td>1178MHz</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>10Gbps GDDR5X</td>
<td>8Gbps GDDR5</td>
<td>7Gbps GDDR5</td>
<td>7Gbps GDDR5</td>
</tr>
<tr>
<td>Memory Bus Width</td>
<td>256-bit</td>
<td>256-bit</td>
<td>256-bit</td>
<td>256-bit</td>
</tr>
<tr>
<td>VRAM</td>
<td>8GB</td>
<td>8GB</td>
<td>4GB</td>
<td>4GB</td>
</tr>
<tr>
<td>FP64</td>
<td>1/32</td>
<td>1/32</td>
<td>1/32</td>
<td>1/32</td>
</tr>
<tr>
<td>TDP</td>
<td>180W</td>
<td>150W</td>
<td>165W</td>
<td>145W</td>
</tr>
<tr>
<td>GPU</td>
<td>GP104</td>
<td>GP104</td>
<td>GM204</td>
<td>GM204</td>
</tr>
<tr>
<td>Transistor Count</td>
<td>7.2B</td>
<td>7.2B</td>
<td>5.2B</td>
<td>5.2B</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td>TSMC 16nm</td>
<td>TSMC 16nm</td>
<td>TSMC 28nm</td>
<td>TSMC 28nm</td>
</tr>
<tr>
<td>Launch Date</td>
<td>05/27/2016</td>
<td>06/10/2016</td>
<td>09/18/14</td>
<td>09/18/14</td>
</tr>
<tr>
<td>Launch Price</td>
<td>MSRP: $599, Founder $699</td>
<td>MSRP: $379, Founder $449</td>
<td>$549</td>
<td>$329</td>
</tr>
</tbody>
</table>
GP104

- GDDR5/GDDR5X
- SM Size: 128 Cores
- 64KB Register File
- 1:32 FP64 Perf
- 1:64 FP16 Perf

GP100

- Compute Preemption
- Dynamic Scheduling
- 16nm FinFET
- HEVC Encode/Decode

- HBM2 w/ECC
- SM Size: 64 Cores
- 128KB Register File
- 1:2 FP64 Perf
- 2:1 FP16 Perf
- NVLInk
Part 3: Research Directions
Decreasing cost per unit computation

- 1971: Intel 4004
- 1981: IBM 5150
- 2007: iPhone
- 2012: Google Datacenter
Ease of Programming vs Hardware Efficiency

- Single Core OoO Superscalar CPU
- Brawny (OoO) Multicore
- Wimpy (In-order) Multicore
- 16K thread, SIMT Accelerator
- ASIC

Better

(how to get here?)
Start by using right tool for each job…
Amdahl’s Law Limits this Approach

Improvement_{overall} = \frac{1}{\text{Fraction}_{\text{hard}}} + \frac{1 - \text{Fraction}_{\text{hard}}}{\text{Improvement}_{\text{easy}}}

Hard to accelerate

Easy to accelerate
Question: Can dividing line be moved?

- Easy to accelerate (Acc. Arch1)
- Easy to accelerate (Acc. Arch2)
Forward-Looking GPU Software

• Still Massively Parallel
• Less Structured
  – Memory access and control flow patterns are less predictable
Two Routes to “Better”

Ease of Programming

Energy Efficiency
Research Direction 1:  
Mitigating SIMT Control Divergence
Recall: SIMT Hardware Stack

Potential for significant loss of throughput when control flow diverged!
Performance vs. Warp Size

- 165 Applications

Convergent Applications

Warp-Size Insensitive Applications

Divergent Applications

Rogers et al., A Variable Warp-Size Architecture, ISCA 2015
Dynamic Warp Formation
(Fung MICRO’07)

How to pick threads to pack into warps?
Dynamic Warp Formation: Hardware Implementation

Thread Scheduler

Warp Update Register T

<table>
<thead>
<tr>
<th>5</th>
<th>2</th>
<th>3</th>
<th>8</th>
<th>0110</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>N</td>
<td>4</td>
<td>1001</td>
<td>C</td>
</tr>
</tbody>
</table>

Warp Update Register NT

PC-Warp LUT

<table>
<thead>
<tr>
<th>B</th>
<th>0010</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1101</td>
<td>1</td>
</tr>
</tbody>
</table>

Warp Pool

<table>
<thead>
<tr>
<th>B</th>
<th>5</th>
<th>2</th>
<th>3</th>
<th>8</th>
<th>Prio</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>Prio</td>
</tr>
<tr>
<td>B</td>
<td>TID</td>
<td>7</td>
<td>Prio</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Warp Allocator

No Lane Conflict

Issue Logic

Commit/Writeback

ALU 1

ALU 2

ALU 3

ALU 4

RF 1

RF 2

RF 3

RF 4

A: BEQ R2, B
C: ...

Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow
DWF Pathologies: Starvation

- **Majority Scheduling**
  - Best Performing
  - Prioritize largest group of threads with same PC

- **Starvation**
  - LOWER SIMD Efficiency!

- Other Warp Scheduler?
  - Tricky: Variable Memory Latency

```c
B: if (K > 10)
  C: K = 10;
else
  D: K = 0;
E: B = C[tid.x] + K;
```
DWF Pathologies: Extra Uncoalesced Accesses

- Coalesced Memory Access = Memory SIMD – 1\textsuperscript{st} Order CUDA Programmer Optimization
- Not preserved by DWF

\[
E: B = C[tid.x] + K;
\]

<table>
<thead>
<tr>
<th>No DWF</th>
<th>With DWF</th>
<th>#Acc = 3</th>
<th>#Acc = 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>1 2 3 4</td>
<td>5 6 7 8</td>
<td>9 10 11 12</td>
<td>1 2 7 12</td>
</tr>
<tr>
<td>5 6 7 8</td>
<td></td>
<td></td>
<td>9 6 3 8</td>
</tr>
<tr>
<td>9 10 11 12</td>
<td></td>
<td></td>
<td>5 10 11 4</td>
</tr>
</tbody>
</table>

Memory:
- \(0x100\)
- \(0x140\)
- \(0x180\)

L1 Cache Absorbs Redundant Memory Traffic
L1\$ Port Conflict
DWF Pathologies: Implicit Warp Sync.

• Some CUDA applications depend on the lockstep execution of “static warps”

<table>
<thead>
<tr>
<th>Warp 0</th>
<th>Thread 0 ... 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>Thread 32 ... 63</td>
</tr>
<tr>
<td>Warp 2</td>
<td>Thread 64 ... 95</td>
</tr>
</tbody>
</table>

— E.g. Task Queue in Ray Tracing

```c
int wid = tid.x / 32;
if (tid.x % 32 == 0) {
    sharedTaskID[wid] = atomicAdd(g_TaskID, 32);
}
my_TaskID = sharedTaskID[wid] + tid.x % 32;
ProcessTask(my_TaskID);
```
Observation

- Compute kernels usually contain divergent and non-divergent (coherent) code segments
- Coalesced memory access usually in coherent code segments
  - DWF no benefit there
Thread Block Compaction

• Run a thread block like a warp
  – Whole block move between coherent/divergent code
  – Block-wide stack to track exec. paths reconvg.
• Barrier @ Branch/reconverge pt.
  – All avail. threads arrive at branch
  – Insensitive to warp scheduling
• Warp compaction
  – Regrouping with all avail. threads
  – If no divergence, gives static warp arrangement
### Thread Block Compaction

<table>
<thead>
<tr>
<th>PC</th>
<th>RPC</th>
<th>Active Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>-</td>
<td>1  2  3  4  5  6  7  8  9  10 11 12</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>-- -- -- -- -- -- -- -- -- -- -- --</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>-- -- -- -- -- -- -- -- -- -- -- --</td>
</tr>
</tbody>
</table>

A: $K = A[tid.x]$;
B: if ($K > 10$)
C: $K = 10$;
else
D: $K = 0$;
E: $B = C[tid.x] + K$;

```
<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
```
Thread Compactor

- Convert *activemask* from block-wide stack to *thread IDs* in warp buffer
- Array of Priority-Encoder

![Diagram of Priority-Encoder and Warp Buffer]
Experimental Results

• 2 Benchmark Groups:
  – COHE = Non-Divergent CUDA applications
  – DIVG = Divergent CUDA applications

 Serious Slowdown from pathologies
 No Penalty for COHE
 22% Speedup on DIVG

Per-Warp Stack
Recent work on warp divergence

- Intel [MICRO 2011]: Thread Frontiers – early reconvergence for unstructured control flow.

- UT-Austin/NVIDIA [MICRO 2011]: Large Warps – similar to TBC except decouple size of thread stack from thread block size.

- NVIDIA [ISCA 2012]: Simultaneous branch and warp interweaving. Enable SIMD to execute two paths at once.

- Intel [ISCA 2013]: Intra-warp compaction – extends Xeon Phi uarch to enable compaction.

- NVIDIA: Temporal SIMT [described briefly in IEEE Micro article and in more detail in CGO 2013 paper]

- NVIDIA [ISCA 2015]: Variable Warp-Size Architecture – merge small warps (4 threads) into “gangs”.
Figure 1: An example of an application with unstructured control flow leading to dynamic code expansion.
Temporal SIMT

Spatial SIMT (current GPUs)

32-wide datapath

Pure Temporal SIMT

1-wide

1 warp instruction = 32 threads

[slide courtesy of Bill Dally]
Temporal SIMT Optimizations

Control divergence — hybrid MIMD/SIMT

Scalarization
Factor common instructions from multiple threads
Execute once – place results in common registers

[See: SIMT Affine Value Structure (ISCA 2013)]

[slide courtesy of Bill Dally]
Scalar Instructions in SIMT Lanes

Scalar instruction spanning warp

Scalar register visible to all threads

Temporal execution of Warp

Multiple lanes/warps

T: thread
R: thread registers
S: scalar registers

[slide courtesy of Bill Dally]
Variable Warp-Size Architecture

- Most recent work by NVIDIA [ISCA 2015]
- Split the SM datapath into narrow slices.
  - Extensively studied 4-thread slices
- Gang slice execution to gain efficiencies of wider warp.

Slices share an L1 I-Cache and Memory Unit

Slices can execute independently

Frontend
L1 I-Cache

Memory Unit

Ganging Unit

Slice
Slice Datapath
4-wide

Slice Front End

Warp Data Path
32-wide

Slice
Slice Datapath
4-wide

Slice Front End

Memory Unit
Divergent Application Performance

<table>
<thead>
<tr>
<th>Divergent Applications</th>
<th>IPC normalized to warp size 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoMD</td>
<td>1.8</td>
</tr>
<tr>
<td>Lighting</td>
<td>1.6</td>
</tr>
<tr>
<td>GamePhysics</td>
<td>1.4</td>
</tr>
<tr>
<td>ObjClassifier</td>
<td>1.2</td>
</tr>
<tr>
<td>Raytracing</td>
<td>1.0</td>
</tr>
<tr>
<td>HMEAN-DIV</td>
<td>0.8</td>
</tr>
</tbody>
</table>

- **WS 32**: Red bars
- **WS 4**: Blue bars
- **I-VWS**: Yellow bars
- **E-VWS**: Diagonal bars

**E-VWS: Break + Reform**

Tim Rogers

A Variable Warp-Size Architecture
Convergent Application Performance

IPC normalized to warp size 32

<table>
<thead>
<tr>
<th>Convergent Applications</th>
<th>Game 1</th>
<th>MatrixMultiply</th>
<th>Game 2</th>
<th>FeatureDetect</th>
<th>Radix Sort</th>
<th>HMEAN-CON</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS 32</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>WS 4</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
</tr>
<tr>
<td>I-VWS</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
</tr>
<tr>
<td>E-VWS</td>
<td>0.70</td>
<td>0.70</td>
<td>0.70</td>
<td>0.70</td>
<td>0.70</td>
<td>0.70</td>
</tr>
</tbody>
</table>

E-VWS: Break + Reform

Warp-Size Insensitive Applications Unaffected
Research Direction 2: Mitigating High GPGPU Memory Bandwidth Demands
Reducing Off-Chip Access / Divergence

• Re-writing software to use “shared memory” and avoid uncoalesced global accesses is bane of GPU programmer existence.

• Recent GPUs introduce caches, but large number of warps/wavefronts lead to thrashing.
• NVIDIA: Register file cache (ISCA 2011, MICRO)
  – Register file burns significant energy
  – Many values read once soon after written
  – Small register file cache captures locality and saves energy but does not help performance
  – Recent follow on work from academia
• Prefetching (Kim, MICRO 2010)
• Interconnect (Bakhoda, MICRO 2010)
• Lee & Kim (HPCA 2012) CPU/GPU cache sharing
Thread Scheduling Analogy

[MICRO 2012]

• Human Multitasking
  – Humans have limited **attention capacity**
  – GPUs have limited **cache capacity**

![Diagram showing the analogy between human multitasking and GPU scheduling](image)

**Graph showing productivity vs. tasks at once:**
- As the number of tasks increases, productivity also increases until a peak is reached, after which it starts to decrease.

**Graph showing performance vs. threads actively scheduled:**
- Performance remains relatively constant as the number of threads increases, indicating efficient scheduling.

**Legend:**
- GPU Core
  - Processor
  - Cache
Use Memory System Feedback

[MICRO 2012]
Sparse Vector-Matrix Multiply

**Example 1 Highly Divergent SPMV-Scalar Kernel**

```c
__global__ void
spmv_csr_scalar_kernel(const float* val,
const int* cols,
const int* rowDelimiters,
const int dim,
float* out)
{
  int myRow = blockDim.x * blockIdx.x + threadIdx.x;
texReader vecTexReader;
  if (myRow < dim)
  {
    float t = 0.0f;
    int start = rowDelimiters[myRow];
    int end = rowDelimiters[myRow+1];
    // Divergent Branch
    for (int j = start; j <= end; j++)
    {
      // Uncoalesced Load
      int col = cols[j];
      t += val[j] * vecTexReader(col);
    }
  out[myRow] = t;
}
```
Sources of Locality

Intra-wavefront locality

Wave_0
LD $line (X)
LD $line (X)

Data Cache

Inter-wavefront locality

Wave_0
Wave_1
LD $line (X)
LD $line (X)

Data Cache
Scheduler affects access pattern

Round Robin Scheduler

Greedy then Oldest Scheduler
Use scheduler to *shape* access pattern

Greedy then Oldest Scheduler

Cache-Conscious Wavefront Scheduling
[MICRO 2012 best paper runner up, Top Picks 2013, CACM Research Highlight]
Wave Scheduler

Locality Scoring System

Memory Unit

Cache

Y 2 Data

Tag WID Data

W0, X

W0, X

W0 detected lost locality

Victim Tags

W0 X Tag

W1 Tag Tag

W2 Tag Tag

No W2 loads

Score

Time

W0

W1

W2

W0

W1

W2

...
The diagram illustrates the Speedup for three methods: LRR, GTO, and CCWS. The x-axis represents the HMEAN-Highly Cache-Sensitive workload. The y-axis shows the Speedup ranging from 0 to 2. The CCWS method shows the highest Speedup, followed by GTO, and then LRR.
Static Wavefront Limiting
[Rogers et al., MICRO 2012]

• Profiling an application we can find an optimal number of wavefronts to execute
• Does a little better than CCWS.
• Limitations: Requires profiling, input dependent, does not exploit phase behavior.
Improve upon CCWS?

- CCWS detects bad scheduling decisions and avoids them in future.

- Would be better if we could “think ahead” / “be proactive” instead of “being reactive”
Observations
[Rogers et al., MICRO 2013]

• Memory divergence in static instructions is predictable

• Data touched by divergent loads dependent on active mask
Footprint Prediction

1. Detect loops with locality

Some loops have locality

Some don’t

Limit multithreading here

2. Classify loads in the loop

Loop with locality

while(...) {
  load 1
  ...
  load 2
}

Diverged
Not Diverged

3. Compute footprint from active mask

Loop with locality

while(...) {
  load 1
  ...
  load 2
}

Diverged
Not Diverged

4 accesses + 1 access

Warp 0’s Footprint = 5 cache lines
Example Compressed Sparse Row Kernel

```c
int C[]={0,64,96,128,160,160,192,224,256};

void sum_row_csr(float* A, ...)
{
    float sum = 0;
    int i = C[tid];

    while (i < C[tid+1])
    {
        sum += A[i];
        ++i;
    }
}
```

### DAWS Operation Example

- **Cache**:
  - A[0], A[64], A[96], A[128]
  - Hit, Hit, Hit, Hit

- **Example Compressed Sparse Row Kernel**
  - `int C[]={0,64,96,128,160,160,192,224,256};`
  - `void sum_row_csr(float* A, ...) {
        float sum = 0;
        int i = C[tid];

        while (i < C[tid+1])
        {
            sum += A[i];
            ++i;
        }
    }
  `

- **Loop**:
  - While loop iteration

<table>
<thead>
<tr>
<th>Warp 0</th>
<th>Warp 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

- **Memory Divergence**
  - Divergent Branch
  - Go

- **Warp 0** has branch divergence
  - Loop for later warps

- **Both warps capture locality together**
  - Footprint decreased = 4x1
Sparse MM Case Study Results

- Performance (normalized to optimized version)

```
Within 4% of optimized with no programmer input
```

![Bar chart showing performance comparison](chart.png)
Memory Request Prioritization Buffer
[Jia et al., HPCA 2014]

- Reorder requests by sorting by Warp ID.
- Bypass when too many accesses to same cache set.

Reorder requests by warp ID
Bypass accesses to hot set
Priority-Based Cache Allocation in Throughput Processors [Li et al., HPCA 2015]

- CCWS leaves L2 and DRAM underutilized.
- Allow some additional warps to execute but do not allow them to allocate space in cache:

  - **Normal Warps**: Warp 0 and Warp 1 are scheduled and allocated in L1.
  - **Non-Polluting Warps**: Warp 2, Warp 3, and Warp 4 are scheduled and bypass L1.
  - **Throttled Warps**: Warp 5 and Warp n-1 are not scheduled.
Coordinated criticality-Aware Warp Acceleration (CAWA) [Lee et al., ISCA 2015]

• Some warps execute longer than others due to lack of uniformity in underlying workload.

• Give these warps more space in cache and more scheduling slots.

• Estimate critical path by observing amount of branch divergence and memory stalls.

• Also, predict if line inserted in line will be used by a warp that is critical using modified version of SHiP cache replacement algorithm.
Other Memory System Performance Considerations

• TLB Design for GPUs.
  – Current GPUs have translation look aside buffers (makes managing multiple graphics application surfaces easier; does not support paging)
  – How does large number of threads impact TLB design?
  – E.g., Power et al., *Supporting x86-64 Address Translation for 100s of GPU Lanes*, HPCA 2014. Importance of multithreaded page table walker + page walk cache.
Research Direction 3: Coherent Memory for Accelerators
Why GPU Coding Difficult?

• Manual data movement CPU $\leftrightarrow$ GPU
• Lack of generic I/O, system support on GPU
• Need for performance tuning to reduce
  – off-chip accesses
  – memory divergence
  – control divergence
• For complex algorithms, synchronization
• Non-deterministic behavior for buggy code
• Lack of good performance analysis tools
Manual CPU ↔ GPU Data Movement

• **Problem #1:** Programmer needs to identify data needed in a kernel and insert calls to move it to GPU
• **Problem #2:** Pointer on CPU does not work on GPU since different address spaces
• **Problem #3:** Bandwidth connecting CPU and GPU is order of magnitude smaller than GPU off-chip
• **Problem #4:** Latency to transfer data from CPU to GPU is order of magnitude higher than GPU off-chip
• **Problem #5:** Size of GPU DRAM memory much smaller than size of CPU main memory
Identifying data to move CPU ⇔ GPU

• CUDA/OpenCL: Job of programmer 😞

• C++AMP passes job to compiler.

• OpenACC uses pragmas to indicate loops that should be offloaded to GPU.
Memory Model

Rapid change (making programming easier)

• Late 1990’s: fixed function graphics only
• 2003: programmable graphics shaders
• 2006: + global/local/shared (GeForce 8)
• 2009: + caching of global/local
• 2011: + unified virtual addressing
• 2014: + unified memory / coherence
Caching

• Scratchpad uses explicit data movement. Extra work. Beneficial when reuse pattern statically predictable.

• NVIDIA Fermi / AMD Southern Island add caches for accesses to global memory space.
CPU memory vs. GPU global memory

• Prior to CUDA: input data is texture map.
• CUDA 1.0 introduces cudaMemcpy
  – Allows copy of data between CPU memory space to global memory on GPU
• Still has problems:
  – #1: Programmer still has to think about it!
  – #2: Communicate only at kernel grid boundaries
  – #3: Different virtual address space
    • pointer on CPU not a pointer on GPU => cannot easily share complex data structures between CPU and GPU
Fusion / Integrated GPUs

• Why integrate?
  – One chip versus two (cf. Moore’s Law, VLSI)
  – Latency and bandwidth of communication: shared physical address space, even if off-chip, eliminates copy: AMD Fusion. 1st iteration 2011. Same DRAM
  – Shared virtual address space? (AMD Kavari 2014)
  – Reduce latency to spawn kernel means kernel needs to do less to justify cost of launching
CPU Pointer not a GPU Pointer

- NVIDIA Unified Virtual Memory partially solves the problem but in a bad way:
  - GPU kernel reads from CPU memory space
- NVIDIA Uniform Memory (CUDA 6) improves by enabling automatic migration of data
CPU ↔ GPU Bandwidth

• Shared DRAM as found in AMD Fusion (recent Core i7) enables the elimination of copies from CPU to GPU. Painful coding as of 2013.

• One question how much benefit versus good coding. Our limit study (WDDD 2008) found only ~50% gain. Lustig & Martonosi HPCA 2013.

• Algorithm design—MummerGPU++
CPU ↔ GPU Latency

- NVIDIA’s solution: **CUDA Streams**. Overlap GPU kernel computation with memory transfer. Stream = ordered sequence of data movement commands and kernels. Streams scheduled independently. Very painful programming.

GPU Memory Size

• CUDA Streams

• Academic work: Treat GPU memory as cache on CPU memory (Kim et al., ScaleGPU, IEEE CAL early access).
Solution to all these sub-issues?

• Heterogeneous System Architecture: Integrated CPU and GPU with coherence memory address space.

• Need to figure out how to provide coherence between CPU and GPU.

• Really two problems: Coherence within GPU and then between CPU and GPU.
Review: Cache Coherence Problem

- Processors see different values for $u$ after event 3
- With write back caches, value written back to memory depends on order of which cache writes back value first
- Unacceptable situation for programmers
1. **Single-Writer, Multiple-Reader (SWMR) Invariant**

![Diagram of SWMR Invariant]

2. **Data-Value Invariant.** The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.
Coherence States

• How to design system satisfying invariants?

• Track “state” of memory block copies and ensure states changes satisfy invariants.

• Typical states: “modified”, “shared”, “invalid”.

• Mechanism for updating block state called a coherence protocol.
Intra-GPU Coherence

[Singh et al., HPCA 2013, IEEE Micro Top Picks 2014]

Coherent memory space
- Efficient critical sections
- Load balancing

lock shared structure
... computation
... unlock
GPU Coherence Challenges

• Challenge 1: Coherence traffic

- No coherence
- MESI
- GPU-VI

![Graph showing interconnect traffic distribution with recall labels for different loads: C1, C2, C3, C4. Each load has an associated recall label (rcl A, rcl B, ack). The graph illustrates the traffic distribution across different coherence levels.]
GPU Coherence Challenges

• Challenge 2: Tracking in-flight requests
  • Significant % of L2
GPU Coherence Challenges

- Challenge 3: Complexity

Non-coherent L1

Non-coherent L2

MESI L2 States
Coherence Challenges

• Challenges of introducing coherence messages on a GPU
  1. Traffic: transferring messages
  2. Storage: tracking message
  3. Complexity: managing races between messages

• GPU cache coherence without coherence messages?
  • YES – using global time
Temporal Coherence

Related: Library Cache Coherence

Global Timestamp

> Global Time $\rightarrow$ VALID

Local Timestamp

< Global Time $\rightarrow$

NO L1 COPIES

Global time
Temporal Coherence Example

Core 1
L1D

Core 2
L1D

L2 Bank

No coherence messages

A=0
A=0
A=0
A=0
A=1
A=1
A=1
A=1

T=0
T=11
T=15
Performance

- TC-Weak with simple predictor performs 85% better than disabling L1 caches
CPU-GPU Coherence?

• Many vendors have introduced chips with both CPU and GPU (e.g., AMD Fusion, Intel Core i7, NVIDIA Tegra, etc…)

• What are the challenges with maintaining coherence across CPU and GPU?

• One important one: GPU has higher cache miss rate than CPU. Can place pressure on directory impacting performance.

• Power et al., *Heterogeneous System Coherence for Integrated CPU-GPU Systems*, ISCA 2013: Use “region coherence” to reduce number of GPU requests that need to access directory.
Review: Consistency Model

• Memory consistency model specifies **allowable** orderings of loads and stores to **different locations**

• The number of **allowable** execution orderings generally far greater than one.

• Ordering of operations from different processors is non-deterministic. Software must use synchronization (mutexes, semaphores, etc...) to provide determinism.
Sequential Consistency

• Sequential consistency is basically a “naïve” programmer’s intuition of allowable orderings:

sequential processors issuing memory references as per program order

switch is randomly set after each memory reference

Memory
Total Store Order (TSO/x86)
Memory Model

Use of write (store) buffer considered very important by Intel and AMD for x86.

Leads to total store order memory model supported by x86.

In general, memory model on multicore processors is not sequential consistency.
Example, TSO/x86 ordering

Program order of core C1

\[
S1: x = \text{NEW} /* \text{NEW} */ \\
L1: r1 = y /* 0 */
\]

Memory order

\[
L1: r1 = y /* 0 */ \\
S2: y = \text{NEW} /* \text{NEW} */ \\
L2: r2 = x /* 0 */
\]

Program order of core C2

(r1, r2) = (0, 0) is legal outcome under TSO/x86 (!)
Current GPU Memory Consistency Models?

• NVIDIA Fermi: No coherence. Can have stale data in first level data cache (e.g., Barnes Hut example from GPU Gems). “Consistency”: Write from kernel N guaranteed to be visible to load from kernel N+1.

• NVIDIA Kepler restricts caching in L1D to global data compiler can prove is read only.

• See also: Alglave et al., “GPU Concurrency: Weak Behaviours and Programming Assumptions”, ASPLOS 2015.
Impact of Consistency Model on Performance of GPU Coherence?

• [Singh HPCA 2013] Assumes release consistency as do more recent AMD/Wisconsin papers on CPU-GPU coherence.

• Hechtman and Sorin [ISCA 2013]: large number of threads on GPU may enable one to implement sequential consistency with same performance as more relaxed consistency models.

• One caveat: Write back caches in their study versus write through in existing GPUs.
Research Direction 4:
Easier Programming with Synchronization
Synchronization

- Locks are not encouraged in current GPGPU programming manuals.
- Interaction with SIMT stack can easily cause deadlocks:

```c
while( atomicCAS(&lock[a[tid]],0,1) != 0 )
    ; // deadlock here if a[i] = a[j] for any i,j = tid in warp

// critical section goes here

atomicExch (&lock[a[tid]], 0) ;
```
Correct way to write critical section for GPGPU:

done = false;
while( !done ) {
    if( atomicCAS (&lock[a[tid]], 0, 1) == 0 ) {
        // critical section goes here

        atomicExch(&lock[a[tid]], 0);
    }
}

Most current GPGPU programs use barriers within thread blocks and/or lock-free data structures.

This leads to the following picture...
• Lifetime of GPU Application Development

Functionality
Performance

E.g. N-Body with 5M bodies
CUDA SDK: $O(n^2)$ – 1640 s (barrier)
Barnes Hut: $O(n \log n)$ – 5.2 s (locks)

Fine-Grained Locking/Lock-Free

Transactionable Memory
Transactional Memory

• Programmer specifies atomic code blocks called transactions [Herlihy’93]

Lock Version:

Lock(X[a]);
Lock(X[b]);
Lock(X[c]);

X[c] = X[a]+X[b];
Unlock(X[c]);
Unlock(X[b]);
Unlock(X[a]);

TM Version:

atomic {
    X[c] = X[a]+X[b];
}

Potential Deadlock!
Transactional Memory

Programmers’ View:

Non-conflicting transactions may run in parallel

Conflicting transactions automatically serialized
Are TM and GPUs Incompatible?

GPU uarch very different from multicore CPU...

KILO TM [MICRO’11, IEEE Micro Top Picks]

- Hardware TM for GPUs
- Half performance of fine grained locking
- Chip area overhead of 0.5%
Hardware TM for GPUs
Challenge #1: SIMD Hardware

- On GPUs, scalar threads in a warp/wavefront execute in lockstep

```
... 
TxBegin
LD r2,[B]
ADD r2,r2,2
ST r2,[A]
TxCommit
... Committed
```

A Warp with 4 Scalar Threads

Branch Divergence!

Aborted
KILO TM – Solution to Challenge #1: SIMD Hardware

• Transaction Abort
  – Like a Loop
  – Extend SIMT Stack

...  
TxBegin  
LD r2,[B]  
ADD r2,r2,2  
ST r2,[A]  
TxCommit  
...
Hardware TM for GPUs
Challenge #2: Transaction Rollback

CPU Core

Register File

@ TX Abort

@ TX Entry

Checkpoint Register File

10s of Registers

GPU Core (SM)

Warp

32k Registers

Register File

2MB Total On-Chip Storage

Checkpoint?
KILO TM – Solution to Challenge #2: Transaction Rollback

• SW Register Checkpoint
  – Most TX: Reg overwritten first appearance (idempotent)
  – TX in Barnes Hut: Checkpoint 2 registers

```assembly
TxBegin
LD r2, [B]
ADD r2, r2, 2
ST r2, [A]
TxCommit
```

Overwritten

Abort
Hardware TM for GPUs
Challenge #3: Conflict Detection

Existing HTMs use Cache Coherence Protocol
- Not Available on (current) GPUs
- No Private Data Cache per Thread

Signatures?
- 1024-bit / Thread
- 3.8MB / 30k Threads
Hardware TM for GPUs
Challenge #4: Write Buffer

Problem: 384 lines / 1536 threads < 1 line per thread!

(48kB)

= 384 X 128B Lines
KILO TM: Value-Based Conflict Detection

- Self-Validation + AI:
  - Only detects **existence** of conflict (not identity)
Parallel Validation? Data Race!??

Private Memory

Read-Log
A=1

Write-Log
B=2

Global Memory

TX1
atomic
{B=A+1}

TX2
atomic
{A=B+2}

Private Memory

Read-Log
B=0

Write-Log
A=2

OR

Tx1 then Tx2:
A=4, B=2

OR

Tx2 then Tx1:
A=2, B=3

A=1
B=0

Serialize Validation?

- Benefit #1: No Data Race
- Benefit #2: No Live Lock
- Drawback: Serializes **Non-Conflicting** Transactions ("collateral damage")
Solution: Speculative Validation

Key Idea: Split Conflict Detection into two parts

1. Recently Committed TX in Parallel
2. Concurrently Committing TX in Commit Order

Approximate

Conflict Rare → Good Commit Parallelism
Efficiency Concerns?

• Scalar Transaction Management
  – Scalar Transaction fits SIMT Model
  – Simple Design
  – Poor Use of SIMD Memory Subsystem

• Rereading every memory location
  – Memory access takes energy

- 128X Speedup over CG-Locks
- 40% FG-Locks Performance
- 2X Energy Usage
Inefficiency from Scalar Transaction Management

- Kilo TM ignores GPU thread hierarchy
  - Excessive Control Message Traffic
  - Scalar Validation and Commit → Poor L2 Bandwidth Utilization

- Simplify HW Design, but **Cost Energy**
Intra-Warp Conflict

- Potential existence of intra-warp conflict introduces complex corner cases:

Read Set
- X=9
- Y=8
- Z=7
- W=6

Write Set
- Y=9
- Z=8
- W=7
- X=6

Correct Outcomes
- Global Memory
  - X = 6
  - Y = 8
  - Z = 8
  - W = 6

OR

Global Memory
- X = 9
- Y = 9
- Z = 7
- W = 7
Intra-Warp Conflict Resolution

- Kilo TM stores read-set and write-set in logs
  - Compact, fits in caches
  - Inefficient for search
- Naive, pair-wise resolution too slow
  - T threads/warp, R+W words/thread
  - \( O(T^2 \times (R+W)^2) \), \( T \geq 32 \)
Fung, MICRO 2013
Intra-Warp Conflict Resolution:
2-Phase Parallel Conflict Resolution

• Insight: Fixed priority for conflict resolution enables parallel resolution
• $O(R+W)$
• Two Phases
  – Ownership Table Construction
  – Parallel Match
Results

40% → 66% FG-Lock Performance

2X → 1.3X Energy Usage

Low Contention Workload: Kilo TM w/ SW Optimizations on par with FG Lock
Other Research Directions....

• Non-deterministic behavior for buggy code
  – GPUDet ASPLOS 2013

• Lack of good performance analysis tools
  – NVIDIA Profiler/Parallel NSight
  – AerialVision [ISPASS 2010]
  – GPU analytical perf/power models (Hyesoon Kim)
Lack of I/O and System Support...

- Support for printf, malloc from kernel in CUDA
- File system I/O?
- GPUfs (ASPLOS 2013):
  - POSIX-like file system API
  - One file per warp to avoid control divergence
  - Weak file system consistency model (close->open)
  - Performance API: O_GWRONCE, O_GWRONCE
  - Eliminate seek pointer
- GPUnet (OSDI 2014): Posix like API for sockets programming on GPGPU.
Conclusions

• GPU Computing is growing in importance due to energy efficiency concerns
• GPU architecture has evolved quickly and likely to continue to do so
• We discussed some of the important microarchitecture bottlenecks and recent research.
• Also discussed some directions for improving programming model