CIS 631
Parallel Processing

Lecture 13: GPUs and Heterogeneous Computing

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon
Acknowledgements

- Portions of the lectures slides were adopted from:
  - Programming Massively Parallel Processors: A Hands-on Approach
    - David B. Kirk, NVIDIA, and Wen-mei Hwu, University of Illinois, Urbana-Champaign
    - Morgan Kaufmann, 2010
  - Rick Vuduc, CSE 6230, Fall 2011, Georgia Institute of Technology
  - Bryan Cantanzaro, Introduction to CUDA/OpenCL and Manycore Graphics Processors, NVIDIA
  - Jeff Vetter, Heterogeneous Computing with GPUs
  - NVIDIA
Thinking about Parallelism

Core
- Assembler
- SIMD, AVX
- Compiler
- Libraries, Frameworks

Socket: Multicore
- Threads – Pthreads, OpenMP
- Distributed memory model like MPI, or GAS Languages
- Libraries, Frameworks

Node
- Threads – Pthreads, OpenMP
- Distributed memory model like MPI, or GAS Languages
- Memory-Thread affinity becomes much more important
- Libraries, Frameworks

System
- Distributed memory model like MPI, GAS Languages
- Libraries, Frameworks
Thinking about Parallelism (2)

Core
- Assembler
- SIMD, AVX
- Compiler
- Libraries, Frameworks

Socket: Multicore
- Threads – Pthreads, OpenMP
- Distributed memory model like MPI, or GAS
  Languages
- Libraries, Frameworks

Node
- Threads – Pthreads, OpenMP
- Distributed memory model like MPI, or GAS
  Languages
- Memory-Thread affinity becomes much more important
- Libraries, Frameworks

System
- Distributed memory model like MPI, GAS
  Languages
- Libraries, Frameworks

Heterogeneity can exist at all levels

Existing GPU, FPGA Archs
#2: Tianhe-1A uses 7,000 NVIDIA GPUs (2011)

- Tianhe-1A uses
  - 7,168 NVIDIA Tesla M2050 GPUs
  - 14,336 Intel Westmeres
- Performance
  - 4.7 PF peak
  - 2.5 PF sustained on HPL
- 4.04 MW
  - If Tesla GPU’s were not used in the system, the whole machine could have needed 12 megawatts of energy to run with the same performance, which is equivalent to 5000 homes
- Custom fat-tree interconnect
  - 2x bandwidth of Infiniband QDR
ORNL “Titan” System

- Upgrade of existing Jaguar Cray XT5
- Cray Linux Environment operating system
- Gemini interconnect
  - 3-D Torus
  - Globally addressable memory
  - Advanced synchronization features
- AMD Opteron 6200 processor (Interlagos)
- New accelerated node design using NVIDIA multi-core accelerators
  - 2011: 960 NVIDIA M2090 “Fermi” GPUs
  - 2012: 10-20 PF NVIDIA “Kepler” GPUs
- 10-20 PFlops peak performance
- Performance based on available funds
- 600 TB DDR3 memory (2x that of Jaguar)

Titan Specs

<table>
<thead>
<tr>
<th>Compute Nodes</th>
<th>18,688</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login &amp; I/O Nodes</td>
<td>512</td>
</tr>
<tr>
<td>Memory per node</td>
<td>32 GB + 6 GB</td>
</tr>
<tr>
<td>NVIDIA “Fermi” (2011)</td>
<td>665 GFlops</td>
</tr>
<tr>
<td># of Fermi chips</td>
<td>960</td>
</tr>
<tr>
<td>NVIDIA “Kepler” (2012)</td>
<td>&gt;1 TFlops</td>
</tr>
<tr>
<td>Opteron</td>
<td>2.2 GHz</td>
</tr>
<tr>
<td>Opteron performance</td>
<td>141 GFlops</td>
</tr>
<tr>
<td>Total Opteron Flops</td>
<td>2.6 PFlops</td>
</tr>
<tr>
<td>Disk Bandwidth</td>
<td>~ 1 TB/s</td>
</tr>
</tbody>
</table>
## Contemporary Architectures

<table>
<thead>
<tr>
<th>Date</th>
<th>System</th>
<th>Location</th>
<th>Comp</th>
<th>Comm</th>
<th>Peak (PF)</th>
<th>Power (MW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>Jaguar; Cray XT5</td>
<td>ORNL</td>
<td>AMD 6c</td>
<td>Seastar2</td>
<td>2.3</td>
<td>7.0</td>
</tr>
<tr>
<td>2010</td>
<td>Tianhe-1A</td>
<td>NSC Tianjin</td>
<td>Intel + NVIDIA</td>
<td>Proprietary</td>
<td>4.7</td>
<td>4.0</td>
</tr>
<tr>
<td>2010</td>
<td>Nebulae</td>
<td>NSCS Shenzhen</td>
<td>Intel + NVIDIA</td>
<td>IB</td>
<td>2.9</td>
<td>2.6</td>
</tr>
<tr>
<td>2010</td>
<td>Tsubame 2</td>
<td>TiTech</td>
<td>Intel + NVIDIA</td>
<td>IB</td>
<td>2.4</td>
<td>1.4</td>
</tr>
<tr>
<td>2011</td>
<td>K Computer</td>
<td>RIKEN/Kobe</td>
<td>SPARC64 VIIIfx</td>
<td>Tofu</td>
<td>10.5</td>
<td>12.7</td>
</tr>
<tr>
<td>2012</td>
<td>Titan; Cray XK6</td>
<td>ORNL</td>
<td>AMD + NVIDIA</td>
<td>Gemini</td>
<td>10-20</td>
<td>7?</td>
</tr>
<tr>
<td>2012</td>
<td>Mira; BlueGeneQ</td>
<td>ANL</td>
<td>SoC</td>
<td>Proprietary</td>
<td>10</td>
<td>?</td>
</tr>
<tr>
<td>2012</td>
<td>Sequoia; BlueGeneQ</td>
<td>LLNL</td>
<td>SoC</td>
<td>Proprietary</td>
<td>20</td>
<td>?</td>
</tr>
<tr>
<td>2012</td>
<td>Blue Waters; Cray</td>
<td>NCSA/UIUC</td>
<td>AMD + (partial) NVIDIA</td>
<td>Gemini</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>2013</td>
<td>Stampede</td>
<td>TACC</td>
<td>Intel + MIC</td>
<td>IB</td>
<td>?</td>
<td>10</td>
</tr>
</tbody>
</table>
Trend #1: Facilities and Power
Trend #2: Dark Silicon (Heterogeneity, Specialization)
AMD’s Llano: A-Series APU

- Combines
  - 4 x86 cores
  - Array of Radeon cores
  - Multimedia accelerators
  - Dual channel DDR3
- 32nm
- Up to 29 GB/s memory bandwidth
- Up to 500 Gflops SP
- 45W TDP

Source: AMD
Multicore and Manycore

- Multicore: yoke of oxen
  - Each core optimized for executing a single thread
- Manycore: flock of chickens
  - Cores optimized for aggregate throughput, deemphasizing individual performance
Heterogeneous Parallel Computing

Multicore CPU
Fast Serial Processing

Manycore GPU
Scalable Parallel Processing
**Multicore CPUs Connected to Manycore GPUs**

- Currently, GPU devices are connected to CPUs over I/O.
## Multicore and Manycore – Real Chips

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Westmere-EP</th>
<th>Fermi (Tesla C2050)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Elements</td>
<td>6 cores, 2 issue, 4 way SIMD @ 3.46 GHz</td>
<td>14 SMs, 2 issue, 16 way SIMD @ 1.15 GHz</td>
</tr>
<tr>
<td>Resident Strands/Threads (max)</td>
<td>6 cores, 2 threads, 4 way SIMD: 48 strands</td>
<td>14 SMs, 48 SIMD vectors, 32 way SIMD: 21504 threads</td>
</tr>
<tr>
<td>SP GFLOP/s</td>
<td>166</td>
<td>1030</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>32 GB/s</td>
<td>144 GB/s</td>
</tr>
<tr>
<td>Register File</td>
<td>6 kB (?)</td>
<td>1.75 MB</td>
</tr>
<tr>
<td>Local Store/L1 Cache</td>
<td>192 kB</td>
<td>896 kB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1536 kB</td>
<td>0.75 MB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>12 MB</td>
<td>-</td>
</tr>
</tbody>
</table>

# transistors & area:           | 1.2 B, 240 mm$^2$                               | 3 B, 520 mm$^2$                             |
thermal design power:           | 130 Watts                                       | 160+ Watts? (240 W/card)                    |
NVIDIA Fermi

- 3B transistors in 40nm
- 512 CUDA Cores
  - New IEEE 754-2008 floating-point standard
    - FMA
    - 8× the peak double precision arithmetic performance over NVIDIA’s last generation GPU
  - 32 cores per SM, 21k threads per chip
- 384b GDDR5, 6 GB capacity
  - 178 GB/s memory BW
- C/M2090
  - 665 GigaFLOPS DP, 6GB
  - ECC Register files, L1/L2 caches, shared memory and DRAM
Why Heterogeneity?

- Different goals produce different designs
  - Manycore assumes work load is highly parallel
  - Multicore must be good at everything, parallel or not

- Multicore: *minimize latency* experienced by 1 thread
  - lots of big on-chip caches
  - extremely sophisticated control

- Manycore: *maximize throughput* of all threads
  - lots of big ALUs
  - multithreading can hide latency ... so skip the big caches
  - simpler control, cost amortized over ALUs via SIMD
**SIMD**

- Single Instruction Multiple Data architectures make use of data parallelism
- We care about SIMD because of area and power efficiency concerns
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler
SIMD – Neglected Parallelism

- It is difficult for a compiler to exploit SIMD
- How do you deal with sparse data & branches?
  - Many languages (like C) are difficult to vectorize
  - Fortran is somewhat better
- Most common solution:
  - Either forget about SIMD
    - Pray the autovectorizer likes you
  - Or instantiate intrinsics (assembly language)
  - Requires a new code version for every SIMD extension

Arguably much better, with recent extensions.
A Brief History of x86 SIMD Extensions

- **2*64 bit FP**
  - Horizontal ops
    - **SSE3**
    - SSSE3
    - **SSE4.1**
  - 8*32 bit FP
    - **SSE2**
  - 4*32 bit FP
    - **SSE**
  - 8*8 bit Int
    - **MMX**

- **AVX**
  - 3 operand
    - **AVX+FMA**
      - 256 bit Int ops, Gather
        - **AVX2**
      - 8*32 bit FP
        - **SSE4.2**
  - **LRB**
    - 512 bit
      - **SSE4.A**
      - **SSE5**
      - **3dNow!**
What to do with SIMD?

- Neglecting SIMD is becoming more expensive
  - AVX: 8 way SIMD, Larrabee: 16 way SIMD, Nvidia: 32 way SIMD, ATI: 64 way SIMD
- This problem composes with thread level parallelism
- We need a programming model which addresses both problems
Manycore GPU Performance

8x Higher Linpack

CPU 1U Server: 2x Intel Xeon X5550 (Nehalem) 2.66 GHz, 48 GB memory, $7K, 0.55 kw

GPU-CPU 1U Server: 2x Tesla C2050 + 2x Intel Xeon X5550, 48 GB memory, $11K, 1.0 kw

Lecture 12

CIS 631 - Parallel Processing
Manycore GPU Performance (2)

Performance Summary

- MIDG: Discontinuous Galerkin Solvers for PDEs
- AMBER Molecular Dynamics (Mixed Precision)
- Lock Exchange Problem OpenCurrent
- OpenEye ROCS Virtual Drug Screening
- Radix Sort CUDA SDK

Speed-up

- Intel Xeon X5550 CPU
- Tesla C1060
- Tesla C2050
Manycore GPU Performance (3)

Standard FFT Library: cuFFT 3.2

Transform Size

Transform Size

cuFFT 3.2: NVIDIA Tesla C1060, Tesla C2050 (Fermi)
MKL 10.2.4.32: Quad-Core Intel Xeon 5550, 2.67 GHz
Manycore GPU Performance (4)

Standard BLAS Library: cuBLAS 3.2

Gflops Single Precision BLAS: SGEMM

Gflops Double Precision BLAS: DGEMM

Matrix Size

cuBLAS 3.2: NVIDIA Tesla C1060, Tesla C2050 (Fermi)
MKL 10.2.4.32: Quad-Core Intel Xeon 5550, 2.67 GHz
Manycore GPU Performance (5)

Matrix Size for Best CUBLAS3.2 Performance

SGEMM: Multiples of 96

DGEMM: Multiples of 64

Gflops

cuBLAS 3.2: NVIDIA Tesla C1060, Tesla C2050 (Fermi)
MKL 10.2.4.32: Quad-Core Intel Xeon 5550, 2.67 GHz
The CUDA Programming Model

- CUDA is a recent programming model, designed for
  - Manycore architectures
  - Wide SIMD parallelism
  - Scalability

- CUDA provides:
  - A thread abstraction to deal with SIMD
  - Synchronization & data sharing between small groups of threads

- CUDA programs are written in C++ with minimal extensions

- OpenCL is inspired by CUDA, but HW & SW vendor neutral
  - Similar programming model, C only for device code
Hierarchy of Concurrent Threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program

- Threads are grouped into thread blocks
  - threads in the same block can cooperate

- Threads/blocks have unique IDs
What is a CUDA Thread?

- Independent thread of execution
  - has its own PC, variables (registers), processor state, etc.
  - no implication about how threads are scheduled

- CUDA threads might be physical threads
  - as mapped onto NVIDIA GPUs

- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU
What is a CUDA Thread Block?

- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want

- Thread blocks of kernel must be independent tasks
  - program valid for any interleaving of block executions
What CUDA Supports

- Thread parallelism
  - each thread is an independent thread of execution

- Data parallelism
  - across threads in a block
  - across blocks in a kernel

- Task parallelism
  - different blocks are independent
  - independent kernels executing in separate streams
### SIMD (SSE) View versus SIMT (CUDA) View

#### SIMD (SSE) View

- **a**: 1 2 3 4
- **b**: 5 6 7 8
- **c**: __m128 \( a = _\text{mm\_set\_ps} (4, 3, 2, 1); \)
  \( b = _\text{mm\_set\_ps} (8, 7, 6, 5); \)
  \( c = _\text{mm\_add\_ps} (a, b); \)

#### SIMT (CUDA) View

- **a**: 1 2 3 4
- **b**: 5 6 7 8
- **c**: float \( a[4] = \{1, 2, 3, 4\}, \)
  \( b[4] = \{5, 6, 7, 8\}, c[4]; \)

// …

// Define a compute kernel, which
// a fine-grained thread executes.
{
    int \text{id} = … ; // my thread ID
    c[\text{id}] = a[\text{id}] + b[\text{id}];
}
CUDA is Extended C

- **Declspecs**
  - global, device, shared, local, constant

- **Keywords**
  - threadIdx, blockIdx

- **Intrinsics**
  - __syncthreads

- **Runtime API**
  - Memory, symbol, execution management

- **Function launch**

```c
__device__ float filter[N];
__global__ void convolve (float *image) {
  __shared__ float region[M];
  ...
  region[threadIdx] = image[i];
  ...
  __syncthreads()
  ...
  image[j] = result;
}

// Allocate GPU memory
global kernel convolve<<<100, 10>>>(myimage);
```
CUDA Compilation

- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

\[\text{C/C++ CUDA Application} \rightarrow \text{NVCC} \rightarrow \text{PTX Code} \rightarrow \text{CPU Code} \rightarrow \text{PTX to Target Compiler} \rightarrow \text{Target code} \rightarrow \text{G80} \rightarrow \text{GPU}\]
CUDA Compilation (2)

• Any source file containing CUDA language extensions must be compiled with NVCC

• NVCC is a compiler driver
  – Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...

• NVCC outputs:
  – C code (host CPU Code)
    • Must then be compiled with the rest of the application using another tool
  – PTX
    • Object code directly
    • Or, PTX source, interpreted at runtime
**Array of Parallel Threads**

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

```c
// Example code snippet
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

```
threadID  0 1 2 3 4 5 6 7
```
Thread Blocks – Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks cannot cooperate
CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to 512 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocs!
Transparent Scalability

- Hardware is free to assigns blocks to any processor at any time
  - A kernel scales across any number of parallel processors

Each block can execute in any order relative to other blocks.
Hello World – Vector Addition

//Compute vector sum C=A+B
//Each thread performs one pairwise addition
__global__ void vecAdd(float* a, float* b, float* c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

int main() {
    //Run N/256 blocks of 256 threads each
    vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);
}
Synchronization

- Threads within a block may synchronize with **barriers**
  ```
  ... Step 1 ...
  __syncthreads();
  ... Step 2 ...
  ```

- Blocks **coordinate** via atomic memory operations
  - e.g., increment shared queue pointer with **atomicInc()**

- Implicit barrier between **dependent kernels**
  ```
  vec_minus<<<nbblocks, blksize>>>(a, b, c);
  -------------------------------
  vec_dot<<<nbblocks, blksize>>>(c, c);
  ```
Blocks Must Be Independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially

- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD ... can easily deadlock

- Independence requirement gives scalability
Scalability

- Manycore chips exist in a diverse set of configurations

![Number of SMs](chart.png)

- CUDA allows one binary to target all these chips
- Thread blocks bring scalability!
Memory Model

- Thread
  - Per-thread Local Memory

- Block
  - Per-block Shared Memory

Sequential Kernels
- Kernel 0
- Kernel 1
  - ...
Memory Model – CPU to/from GPU Device

Host Memory

cudaMemcpy()

Device 0 Memory

Device 1 Memory
Hello World – Managing Data

```c
int main() {
    int N = 256 * 1024;
    float* h_a = malloc(sizeof(float) * N);
    //Similarly for h_b, h_c. Initialize h_a, h_b

    float *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, sizeof(float) * N);
    //Similarly for d_b, d_c

    cudaMemcpy(d_a, h_a, sizeof(float) * N, cudaMemcpyHostToDevice);
    //Similarly for d_b

    //Run N/256 blocks of 256 threads each
    vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);

    cudaMemcpy(h_c, d_c, sizeof(float) * N, cudaMemcpyDeviceToHost);
}
```
Using per-Block Shared Memory

- Variables shared across block
  ```c
  __shared__ int *begin, *end;
  ```

- Scratchpad memory
  ```c
  __shared__ int scratch[BLOCKSIZE];
  scratch[threadIdx.x] = begin[threadIdx.x];
  // ... compute on scratch values ...
  begin[threadIdx.x] = scratch[threadIdx.x];
  ```

- Communicating values between threads
  ```c
  scratch[threadIdx.x] = begin[threadIdx.x];
  __syncthreads();
  int left = scratch[threadIdx.x - 1];
  ```

- Per-block shared memory is faster than L1 cache, slower than register file
- It is relatively small: register file is 2-4x larger
CUDA – Minimal Extensions to C/C++

- Declaration specifiers to indicate where things live
  __global__ void KernelFunc(...);  // kernel callable from host
  __device__ void DeviceFunc(...);  // function callable on device
  __device__ int GlobalVar;        // variable in device memory
  __shared__ int SharedVar;        // in per-block shared memory

- Extend function invocation syntax for parallel kernel launch
  KernelFunc<<<500, 128>>>(...);   // 500 blocks, 128 threads each

- Special variables for thread identification in kernels
  dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;

- Intrinsics that expose specific operations in kernel code
  __syncthreads();                  // barrier synchronization
CUDA – Features Available on GPU

- Double and single precision (IEEE compliant)

- Standard mathematical functions
  - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtyf`, etc.

- Atomic memory operations
  - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.

- These work on both global and shared memory
CUDA – Runtime Support

- Explicit memory allocation returns pointers to GPU memory
  - `cudaMalloc()`, `cudaFree()`

- Explicit memory copy for host ↔ device, device ↔ device
  - `cudaMemcpy()`, `cudaMemcpy2D()`, ...

- Texture management
  - `cudaBindTexture()`, `cudaBindTextureToArray()`, ...

- OpenGL & DirectX interoperability
  - `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, ...
Imperatives for Efficient CUDA Code

- Expose abundant fine-grained parallelism
  - need 1000’s of threads for full utilization

- Maximize on-chip work
  - on-chip memory orders of magnitude faster

- Minimize execution divergence
  - SIMT execution of threads in 32-thread warps

- Minimize memory divergence
  - warp loads and consumes complete 128-byte cache line
Mapping CUDA to NVIDIA GPUs

- CUDA is designed to be functionally forgiving

- However, to get good performance, one must understand how CUDA is mapped to Nvidia GPUs

- Threads: each thread is a SIMD vector lane

- Warps: A SIMD instruction acts on a “warp”
  - Warp width is 32 elements: LOGICAL SIMD width

- Thread blocks: Each thread block is scheduled onto an SM
  - Peak efficiency requires multiple thread blocks per SM
Mapping CUDA to NVIDIA GPUs (2)

- The GPU is very deeply pipelined to maximize throughput.
- This means that performance depends on the number of thread blocks which can be allocated on a processor.
- Therefore, resource usage costs performance:
  - More registers => Fewer thread blocks
  - More shared memory usage => Fewer thread blocks
- It is often worth trying to reduce register count in order to get more thread blocks to fit on the chip:
  - For Fermi, target 20 registers or less per thread for full occupancy.
Occupyancy (Constants for Fermi)

- The Runtime tries to fit as many thread blocks simultaneously as possible on to an SM
  - The number of simultaneous thread blocks (B) is ≤ 8
  - The number of warps per thread block (T) ≤ 32
  - B * T ≤ 48 (Each SM has scheduler space for 48 warps)
  - The number of threads per warp (V) is 32
  - B * T * V * Registers per thread ≤ 32768
  - B * Shared memory (bytes) per block ≤ 49152/16384
    - Depending on Shared memory/L1 cache configuration

- Occupancy is reported as B * T / 48
SIMD and Control Flow

- Nvidia GPU hardware handles control flow divergence and reconvergence
- Write scalar SIMD code, the hardware schedules the SIMD execution
- One caveat: `__syncthreads()` can’t appear in a divergent path
  - This will cause programs to hang
- Good performing code will try to keep the execution convergent within a warp
  - Warp divergence only costs because of a finite instruction cache
**Memory, Memory, Memory**

- A many core processor ≡ A device for turning a compute bound problem into a memory bound problem

- Lots of processors, only one socket
- Memory concerns dominate performance tuning
Memory is SIMD Too!

- Virtually all processors have SIMD memory subsystems

![Cache Line Width Diagram]

- This has two effects:
  - Sparse access wastes bandwidth
    - 2 words used, 8 words loaded: \(\frac{1}{4}\) effective bandwidth
  - Unaligned access wastes bandwidth
    - 4 words used, 8 words loaded: \(\frac{1}{2}\) effective bandwidth
Coalescing

- GPUs and CPUs both perform memory transactions at a larger granularity than the program requests ("cache line")
- GPUs have a "coalescer", which examines memory requests dynamically and coalesces them
- To use bandwidth effectively, when threads load, they should:
  - Present a set of unit strided loads (dense accesses)
  - Keep sets of loads aligned to vector boundaries
Data Structure Padding

- Multidimensional arrays are usually stored as monolithic vectors in memory.
- Care should be taken to assure aligned memory accesses for the necessary access pattern.
OpenCL

- OpenCL is supported by AMD {CPUs, GPUs} and Nvidia
  - Intel, Imagination Technologies (purveyor of GPUs for iPhone/OMAP/etc.) are also on board
- OpenCL’s data parallel execution model mirrors CUDA, but with different terminology
- OpenCL has rich task parallelism model
- Runtime walks a dataflow DAG of kernels/memory transfers
Thrust

- There exist many tools and libraries for GPU programming
- Thrust is now part of the CUDA SDK
- C++ libraries for CUDA programming, inspired by STL
- Many important algorithms:
  - reduce, sort, reduce_by_key, scan, ...
- Dramatically reduces overhead of managing heterogeneous memory spaces
- Includes OpenMP backend for multicore programming
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
Thrust saxpy

// C++ functor replaces __global__ function
struct saxpy
{
    float a;

    saxpy(float _a) : a(_a) {}

    __host__ __device__
    float operator()(float x, float y)
    {
        return a * x + y;
    }
};

transform(x.begin(), x.end(), y.begin(), y.begin(), saxpy(a));
Summary

- Manycore processors provide useful parallelism
- Programming models like CUDA and OpenCL enable productive parallel programming
- They abstract SIMD, making it easy to use wide SIMD vectors
- CUDA and OpenCL encourages SIMD friendly, highly scalable algorithm design and implementation
- Thrust is a productive C++ library for CUDA development
Next Class

- Preparation for midterm