CIS 410/510
Parallel Computing
Manycore Computing and GPUs

Prof. Allen D. Malony
Department of Computer and Information Science
Spring 2016
Logistics

- Midterm handed back last week
  - See email for midterm answers
- No final exam
- Please plan to present your class project results in the final period
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

<table>
<thead>
<tr>
<th>Hardware level</th>
<th>Software (programming) level</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Core</strong></td>
<td>• Assembler</td>
</tr>
<tr>
<td></td>
<td>• SIMD, AVX</td>
</tr>
<tr>
<td></td>
<td>• Compiler</td>
</tr>
<tr>
<td></td>
<td>• Libraries, Frameworks</td>
</tr>
<tr>
<td><strong>Socket: Multicore</strong></td>
<td>• Threads – Pthreads, OpenMP, TBB, Cilk Plus, ...</td>
</tr>
<tr>
<td></td>
<td>• Distributed memory model like MPI or GAS Languages</td>
</tr>
<tr>
<td></td>
<td>• Libraries, Frameworks</td>
</tr>
<tr>
<td><strong>Node</strong></td>
<td>• Threads – Pthreads, OpenMP, TBB, Cilk Plus, ...</td>
</tr>
<tr>
<td></td>
<td>• Distributed memory model like MPI or GAS Languages</td>
</tr>
<tr>
<td></td>
<td>• Memory-Thread affinity becomes much more important</td>
</tr>
<tr>
<td></td>
<td>• Libraries, Frameworks</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>• Distributed memory model like MPI or GAS Languages</td>
</tr>
<tr>
<td></td>
<td>• Libraries, Frameworks</td>
</tr>
</tbody>
</table>
Heterogeneity and Parallelism

- Different parallel hardware
- Different ways for programming that hardware

![Diagram showing heterogeneity at different levels](image-url)
Trend #1: Facilities and Power
## Trend #2: Dark Silicon (Heterogeneity, Specialization)

<table>
<thead>
<tr>
<th>Year</th>
<th>Node</th>
<th>Area $^2$</th>
<th>Peak freq</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>45nm</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2014</td>
<td>22nm</td>
<td>4</td>
<td>1.6</td>
<td>1</td>
</tr>
<tr>
<td>2020</td>
<td>11nm</td>
<td>16</td>
<td>2.4</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Exploitable Si (In 45nm power budget):

- $(4 \times 1)^2 = 25\%$
- $(16 \times 0.6)^2 = 10\%$

Source: ISSCC 2016

---

*The Architecture for the Digital World*

ARM
Multicore versus 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
Memory, Memory, Memory

- A many core processor $\equiv$ A device for turning a compute bound problem into a memory bound problem

- Lots of processors, only one socket
- Memory concerns dominate performance tuning
Tianhe-1A uses 7,000 NVIDIA GPUs (2010)

- 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
### Top 10 (November 2014)

<table>
<thead>
<tr>
<th>RANK</th>
<th>SITE</th>
<th>SYSTEM</th>
<th>CORES</th>
<th>RMAX (TFLOP/S)</th>
<th>RPeak (TFLOP/S)</th>
<th>Power (KW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Super Computer Center in Guangzhou, China</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.20GHz, TH Express-2, Intel Xeon Phi 3151P NUDT</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>2</td>
<td>DOE/SC/Oak Ridge National Laboratory, United States</td>
<td>Titan - Cray XK7, Opteron 6374 16C 2.20GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>3</td>
<td>DOE/NNSA/LLNL, United States</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM</td>
<td>1,572,864</td>
<td>17,173.2</td>
<td>20,132.7</td>
<td>7,890</td>
</tr>
<tr>
<td>4</td>
<td>RIKEN Advanced Institute for Computational Science (AICS), Japan</td>
<td>K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu</td>
<td>705,024</td>
<td>10,510.0</td>
<td>11,280.4</td>
<td>12,660</td>
</tr>
<tr>
<td>5</td>
<td>DOE/SC/Argonne National Laboratory, United States</td>
<td>Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM</td>
<td>786,432</td>
<td>8,586.6</td>
<td>10,066.3</td>
<td>3,945</td>
</tr>
<tr>
<td>6</td>
<td>Swiss National Supercomputing Centre (CSCS), Switzerland</td>
<td>Piz Daint - Cray XC30, Xeon E5-2670 8C 2.60GHz, Aries interconnect, NVIDIA K20x Cray Inc.</td>
<td>115,984</td>
<td>6,271.0</td>
<td>7,788.9</td>
<td>2,325</td>
</tr>
<tr>
<td>7</td>
<td>Texas Advanced Computing Center/Univ. of Texas, United States</td>
<td>Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.70GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell</td>
<td>462,462</td>
<td>5,168.1</td>
<td>8,520.1</td>
<td>4,510</td>
</tr>
<tr>
<td>8</td>
<td>Forschungszentrum Juelich (FZJ), Germany</td>
<td>JUQUEEN - BlueGene/Q, Power BQC 16C 1.60Ghz, Custom Interconnect IBM</td>
<td>458,752</td>
<td>5,008.9</td>
<td>5,872.0</td>
<td>2,301</td>
</tr>
<tr>
<td>9</td>
<td>DOE/NNSA/LLNL, United States</td>
<td>Vulcan - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM</td>
<td>393,216</td>
<td>4,293.3</td>
<td>5,033.2</td>
<td>1,972</td>
</tr>
<tr>
<td>10</td>
<td>Government, United States</td>
<td>Cray CS-Storm, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, Nvidia K40 Cray Inc.</td>
<td>72,800</td>
<td>3,577.0</td>
<td>6,131.8</td>
<td>1,499</td>
</tr>
</tbody>
</table>

**Different architectures**

- **GPU-based machines**
- **MIC-based machines**
Top 500 – Performance (November 2014)
#1: NUDT Tiahne-2 (Milkyway-2)

- **Compute Nodes** have 3.432 Tflop/s per node
  - 16,000 nodes
  - 32000 Intel Xeon CPU
  - 48000 Intel Xeon Phi

- **Operations Nodes**
  - 4096 FT CPUs

- **Proprietary interconnect**
  - TH2 express

- **1PB memory**
  - Host memory only

- **Global shared parallel storage** is 12.4 PB

- **Cabinets**: 125+13+24 = 162
  - Compute, communication, storage
  - ~750 m²
#2: ORNL Titan Hybrid System (Cray XK7)

- Peak performance of 27.1 PF
  - 24.5 GPU + 2.6 CPU
- 18,688 Compute Nodes each with:
  - 16-Core AMD Opteron CPU
  - NVIDIA Tesla “K20x” GPU
  - 32 + 6 GB memory
- 512 Service and I/O nodes
- 200 Cabinets
- 710 TB total system memory
- Cray Gemini 3D Torus Interconnect
- 8.9 MW peak power

4,352 ft²
## Top 10 (November 2015)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System Description</th>
<th>Cores</th>
<th>Rmax (TFLOP/S)</th>
<th>Rpeak (TFLOP/S)</th>
<th>Power (KW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Super Computer Center in Guangzhou, China</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>2</td>
<td>DOE/SC/Oak Ridge National Laboratory, United States</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>3</td>
<td>DOE/NNSA/LNL, United States</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM</td>
<td>1,572,864</td>
<td>17,173.2</td>
<td>20,132.7</td>
<td>7,890</td>
</tr>
<tr>
<td>4</td>
<td>RIKEN Advanced Institute for Computational Science (AICS), Japan</td>
<td>K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu</td>
<td>705,024</td>
<td>10,510.0</td>
<td>11,280.4</td>
<td>12,660</td>
</tr>
<tr>
<td>5</td>
<td>DOE/SC/Argonne National Laboratory, United States</td>
<td>Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM</td>
<td>786,432</td>
<td>8,586.6</td>
<td>10,066.3</td>
<td>3,945</td>
</tr>
<tr>
<td>6</td>
<td>DOE/NNSA/LANL, United States</td>
<td>Trinity - Cray XC40, Xeon E5-2698v3 16C 2.30GHz, Aries interconnect Cray Inc.</td>
<td>301,056</td>
<td>8,100.9</td>
<td>11,078.9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Swiss National Supercomputing Centre (CScS), Switzerland</td>
<td>Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect, NVIDIA K20x Cray Inc.</td>
<td>115,984</td>
<td>6,271.0</td>
<td>7,788.9</td>
<td>2,325</td>
</tr>
<tr>
<td>8</td>
<td>HLRS - Höchstleistungsrechenzentrum, Stuttgart, Germany</td>
<td>Hazel Hen - Cray XC40, Xeon E5-2680v3 12C 2.5GHz, Aries interconnect Cray Inc.</td>
<td>185,088</td>
<td>5,640.2</td>
<td>7,403.5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>King Abdullah University of Science and Technology, Saudi Arabia</td>
<td>Shaheen II - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.</td>
<td>196,608</td>
<td>5,537.0</td>
<td>7,235.2</td>
<td>2,834</td>
</tr>
<tr>
<td>10</td>
<td>Texas Advanced Computing Center/University of Texas, United States</td>
<td>Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell</td>
<td>462,462</td>
<td>5,168.1</td>
<td>8,520.1</td>
<td>4,510</td>
</tr>
</tbody>
</table>

### Different Architectures
- **GPU-based machines**
- **MIC-based machines**
Heterogeneous Parallel Computing

(GPGPU – general purpose graphics processing unit)
# Multicore versus 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 mm²
- 3 B, 520 mm²
- Thermal Design Power: 130 Watts
- 160+ Watts? (240 W/card)
Multicore CPUs Connected to Manycore GPUs

- Typically, GPU devices are accessed over PCI
Integrated CPU and GPU

- AMD APU
  - x86 cores
  - Array of Radeon cores
  - Multimedia accelerators
  - Dual channel DDR3

- Intel Ivy Bridge and Haswell
  - x86 cores
  - Symmetric execution units
NVIDIA Fermi

- 3B transistors in 40nm
- 512 CUDA Cores
  - New IEEE 754-2008 floating-point standard
    - FMA
    - 8x the peak double precision arithmetic performance over NVIDIA's last generation
  - 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
- 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

8*8 bit Int
8*32 bit FP
2*64 bit FP
Horizontal ops

MMX
SSE
SSE2
SSE3
SSSE3
SSE4.1
SSE4.2
AVX
AVX+FMA
AVX2
LRB
3dNow!
SSE4.A
SSE5

512 bit
256 bit Int ops, Gather
3 operand
8*32 bit FP
What to do with SIMD?

4 way SIMD (SSE)       16 way SIMD (LRB)

- 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

8x
Performance
Gflops

5x
Performance / $
Gflops / $K

4.5x
Performance / watt
Gflops / kwatt

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

CIS 410/510: Parallel Computing, University of Oregon, Spring 2016
Lecture 15 – Manycore Computing and GPUs
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

Graphs showing speed-up for different applications and GPUs:
- Mid Xeon X5550 CPU
- Tesla C1060
- Tesla C2050
Manycore GPU Performance (3)
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
Matrix Size for Best CUBLAS3.2 Performance

SGEMM: Multiples of 96

DGEMM: Multiples of 64

Gflops

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
Using CPU+GPU Architecture

- Heterogeneous system architecture
- Use the right processor and memory for each task
- CPU excels at executing a few serial threads
  - Fast sequential execution
  - Low latency cached memory access
- GPU excels at executing many parallel threads
  - Scalable parallel execution
  - High bandwidth parallel memory access
CUDA Parallelism

- CUDA virtualizes the physical hardware
  - Thread is a virtualized scalar processor (registers, PC, state)
  - Block is a virtualized multiprocessor (threads, shared memory)
- Scheduled onto physical hardware without pre-emption
  - Threads/blocks launch & run to completion
  - Blocks execute independently
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**

\[
\begin{array}{c}
\text{a} & 1 & 2 & 3 & 4 \\
+ & + & + & + \\
\text{b} & 5 & 6 & 7 & 8 \\
= & = & = & = \\
\text{c} & & & & \\
\end{array}
\]

\[
\begin{array}{c}
\_m128 \text{a} = \_mm\_set\_ps (4, 3, 2, 1); \\
\_m128 \text{b} = \_mm\_set\_ps (8, 7, 6, 5); \\
\_m128 \text{c} = \_mm\_add\_ps (\text{a}, \text{b}); \\
\end{array}
\]

**Example Code**

```c
float a[4] = {1, 2, 3, 4}, b[4] = {5, 6, 7, 8}, c[4];
```

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

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

- **Keywords**
  - threadIdx, blockIdx

- **Intrinsics**
  - __syncthreads

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

- **Function launch**

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

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>>(myimage);
```
CUDA Compilation

C/C++ CUDA Application → NVCC → PTX Code → PTX to Target Compiler → Target code

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

G80, ..., GPU

Virtual

Physical

Target code
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 code to be executed on GPU 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
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 of threads
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks cannot cooperate except through global memory

```c
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```
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 in 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 blocks!
**Transparent Scalability**

- Hardware is free to assign blocks to any processor at any time
  - A kernel can thus scale 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);
}

N/256 blocks
256 threads per block
CUDA kernel maps to Grid of Blocks

```
kernel_func<<<nblk, nthread>>>(param, ...);
```
Thread blocks execute on an SM

- Thread instructions execute on a core

```c
float myVar;
__shared__ float shVar;
__device__ float glVar;
```
Synchronization

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

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

- Implicit barrier between dependent kernels
  
  ```c
  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
Manycore chips exist in a diverse set of configurations

CUDA allows one binary to target all these chips

Thread blocks bring scalability!
Memory Model

Thread

Per-thread Local Memory

float LocalVar;

Block

Per-block Shared Memory

__shared__ float SharedVar;

Sequential Kernels

Kernel 0

Kernel 1

Per Device Global Memory

__device__ float GlobalVar;
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
  
  ```c
  __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
  
  ```c
  KernelFunc<<<500, 128>>>(...); // 500 blocks, 128 threads each
  ```

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

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

- Double and single precision (IEEE compliant)

- Standard mathematical functions
  - `sinf, powf, atanf, ceil, min, sqrtf`, 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
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.
Occupancy (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 \(\leq 8\)
  - The number of warps per thread block \((T)\) \(\leq 32\)
  - \(B \times T \leq 48\) (Each SM has scheduler space for 48 warps)
  - The number of threads per warp \((V)\) is 32
  - \(B \times T \times V \times \text{Registers per thread} \leq 32768\)
  - \(B \times \text{Shared memory (bytes) per block} \leq 49152/16384\)
    - Depending on Shared memory/L1 cache configuration

- Occupancy is reported as \(B \times T / 48\)
**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

  \[ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \]

  cache line width

- This has two effects:
  - Sparse access wastes bandwidth
    \[ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \]
    2 words used, 8 words loaded: \( \frac{1}{4} \) effective bandwidth
  - Unaligned access wastes bandwidth
    \[ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \]
    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
Thrust Hello World

```c
#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
Optimizing Parallel Performance

- Understand how software maps to architecture
- Use heterogeneous CPU+GPU computing
- Use massive amounts of parallelism
- Understand SIMT instruction execution
- Enable global memory coalescing
- Understand cache behavior
- Use shared memory
- Optimize memory copies
- Understand PTX instructions
Using CPU+GPU Architecture

- Heterogeneous system architecture
- Use the right processor and memory for each task
- CPU excels at executing a few serial threads
  - Fast sequential execution
  - Low latency cached memory access
- GPU excels at executing many parallel threads
  - Scalable parallel execution
  - High bandwidth parallel memory access
CUDA Parallelism

- CUDA virtualizes the physical hardware
  - Thread is a virtualized scalar processor (registers, PC, state)
  - Block is a virtualized multiprocessor (threads, shared memory)

- Scheduled onto physical hardware without pre-emption
  - Threads/blocks launch & run to completion
  - Blocks execute independently

```
**CUDA Parallelism**

- CUDA virtualizes the physical hardware
  - Thread is a virtualized scalar processor (registers, PC, state)
  - Block is a virtualized multiprocessor (threads, shared memory)

- Scheduled onto physical hardware without pre-emption
  - Threads/blocks launch & run to completion
  - Blocks execute independently

---

**Diagram:**

- Host Thread
- Grid of Thread Blocks
- CPU
- Cache
- Bridge
- Device Memory
- PCIe Bridge
- Host Memory
- GPU SMs
- Device Memory Cache
- SMem
```
Exposé Massive Parallelism

- Use hundreds to thousands of thread blocks
  - A thread block executes on one SM
  - Need many blocks to use 10s of SMs
  - SM executes 2 to 8 concurrent blocks efficiently
  - Need many blocks to scale to different GPUs
  - Coarse-grained data parallelism, task parallelism

- Use hundreds of threads per thread block
  - A thread instruction executes on one core
  - Need 384 – 512 threads/SM to use all the cores all the time
  - Use multiple of 32 threads (warp) per thread block
  - Fine-grained data parallelism, vector parallelism, thread parallelism, instruction-level parallelism
Scalable Parallel Architectures

- Run thousands of concurrent threads
  - 32 SP cores: 3,072 threads
  - 128 SP cores: 12,288 threads
  - 240 SP cores: 30,720 threads
Fermi Streaming Multiprocessor (SM)

- Increases instruction-level parallelism

512 CUDA Cores
24,576 threads
SM parallel instruction execution

- **SIMT** (Single Instruction Multiple Thread) execution
  - Threads run in groups of 32 called **warps**
  - Threads in a warp share instruction unit (IU)
  - HW automatically handles branch divergence

- Hardware multithreading
  - HW resource allocation & thread scheduling
  - HW relies on threads to hide latency

- Threads have all resources needed to run
  - Any warp not waiting for something can run
  - Warp context switches are zero overhead
Use per-Block Shared Memory

- Latency is an order of magnitude lower than L2 or DRAM
- Bandwidth is 4x – 8x higher than L2 or DRAM

- Place data blocks or tiles in shared memory when the data is accessed multiple times
- Communicate among threads in a block using Shared memory
- Use synchronization barriers between communication steps
  - __syncthreads() is single bar.sync instruction – very fast

- Threads of warp access shared memory banks in parallel via fast crossbar network
- Bank conflicts can occur – incur a minor performance impact
- Pad 2D tiles with extra column for parallel column access if tile width == # of banks (16 or 32)
Using cudaMemcpy()

- cudaMemcpy() invokes a DMA copy engine
- Minimize the number of copies
- Use data as long as possible in a given place
- PCIe gen2 peak bandwidth = 6 GB/s
- GPU load/store DRAM peak bandwidth = 150 GB/s
Overlap computing & CPU ↔ GPU transfers

- `cudaMemcpy()` invokes data transfer engines
  - CPU → GPU and GPU → CPU data transfers
  - Overlap with CPU and GPU processing

- Pipeline Snapshot:
Fermi runs independent kernels in parallel

Concurrent Kernel Execution + Faster Context Switch

Serial Kernel Execution

Parallel Kernel Execution
Minimize thread runtime variance

Long running warp

Warps executing kernel with variable run time
Optimizing Parallel GPU Performance

- Understand the parallel architecture
- Understand how application maps to architecture
- Use LOTS of parallel threads and blocks
- Often better to redundantly compute in parallel
- Access memory in local regions
- Leverage high memory bandwidth
- Keep data in GPU device memory
- Experiment and measure