CIS 631
Advanced Parallel Computing
Parallel Programming Methods:
GPU (CUDA)

Prof. Allen D. Malony
Department of Computer and Information Science
Winter 2022
Logistics

- All videos should now be available
- OpenMP exercise due on Friday
Thinking about Parallelism

Hardware level

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

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

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

System
- Distributed memory model like MPI or GAS Languages
- Libraries, Frameworks
Heterogeneity and Parallelism

- Different parallel hardware
- Different ways for programming that hardware

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

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

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

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

Heterogeneity can exist at all levels.
Multicore versus Manycore

- **Multicore**: yoke of oxen
  - Each core optimized for executing single thread

- **Manycore**: flock of chickens
  - Cores optimized for aggregated throughput
deemphasize individual performance
CPU Multicore versus GPU Manycore

- Manycore processor turns a compute-bound problem into a memory-bound problem

- Lots of CPUs in a single processor
- 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 2.200GHz, 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 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/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 VIII fx 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 2.700GHz, 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>
<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 1.60GHz, 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>
Top 500 – Performance (November 2014)
#1: NUDT Tiahne-2 (Milkyway-2) (2014)

- 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) (2014)

- 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</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 3151P NUDT</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>
</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>
</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>
</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>
</tr>
<tr>
<td>6</td>
<td>DOE/NNSA/LANL/SNL United States</td>
<td>Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.</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>
</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>
</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>
</tr>
<tr>
<td>10</td>
<td>Texas Advanced Computing Center/Univ. of Texas United States</td>
<td>Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RANK</th>
<th>SITE</th>
<th>SYSTEM</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 3151P NUDT</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>
</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>
</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>
</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>
</tr>
<tr>
<td>6</td>
<td>DOE/NNSA/LANL/SNL United States</td>
<td>Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.</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>
</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>
</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>
</tr>
<tr>
<td>10</td>
<td>Texas Advanced Computing Center/Univ. of Texas United States</td>
<td>Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell</td>
</tr>
</tbody>
</table>

Different architectures

GPU-based machines

MIC-based machines
Heterogeneous Parallel Computing

Multicore CPU

Fast serial processing

Manycore GPU

Scalable parallel processing
### 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:**
  - Westmere-EP: 1.2 B, 240 mm²
  - Fermi: 3 B, 520 mm²

- **thermal design power:**
  - Westmere-EP: 130 Watts
  - Fermi: 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
  - 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

SM: streaming multiprocessor
Why Heterogeneity?

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

- Multicore: *minimize latency* in 1 thread
  - Lots of big on-chip caches
  - Extremely sophisticated control

- Manycore: *maximize throughput* of all threads
  - Lots of ALUs … skip the big (per core) caches
  - Multithreading can hide latency
  - Simple control with SIMD
A Brief History of x86 SIMD Extensions

- **8*8 bit Int**: MMX
- **4*32 bit FP**: SSE
- **2*64 bit FP**: SSE2
- **Horizontal ops**: SSE3
- **SSSE3**: SSE4.1
- **AVX**: SSE4.2
- **AVX+FMA**: AVX
- **AVX2**: 8*32 bit FP
- **256 bit Int ops, Gather**: 3 operand
- **LRB**: 512 bit

- **3dNow!**
- **SSE4.A**
- **SSE5**
**SIMD (SSE) View  versus  SIMT (CUDA) View**

\[ \begin{align*}
\text{a} & \quad 1 \quad 2 \quad 3 \quad 4 \\
\text{b} & \quad 5 \quad 6 \quad 7 \quad 8 \\
\text{c} & \quad \text{empty}
\end{align*} \]

\[ \begin{align*}
\_\text{m128} \ a & = \_\text{mm_set_ps} (4, 3, 2, 1); \\
\_\text{m128} \ b & = \_\text{mm_set_ps} (8, 7, 6, 5); \\
\_\text{m128} \ c & = \_\text{mm_add_ps} (a, b);
\end{align*} \]

\[ \begin{align*}
\text{float} \ a[4] & = \{1, 2, 3, 4\}, \\
b[4] & = \{5, 6, 7, 8\}, \ c[4];
\end{align*} \]

```c
// …
// Define a compute kernel, which
// a fine-grained thread executes.
{
    int id = … ;  // my thread ID
    c[id] = a[id] + b[id];
}
```
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
GPU Parallelism

- GPU parallelism 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 and run to completion
  - Blocks execute independently
SM parallel instruction execution

- **SIMT** (Single Instruction Multiple Thread)
  - 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
Expose 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
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
CUDA Programming Model

- CUDA is a programming model for:
  - Manycore architectures
  - Wide SIMD parallelism
  - Scalability

- CUDA provides:
  - A thread abstraction to deal with SIMD
  - Synchronization and data sharing
  - Groups of threads

- CUDA programs are written in C++ with extensions

- OpenCL is inspired by CUDA
  - HW and SW vendor neutral
  - Similar programming model with C used for device code
Hierarchical 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), CPU state, …
  - No implication about how threads are scheduled

- CUDA threads are mapped onto GPUs
What is a CUDA Thread Block?

- A thread block is a (data) parallel task
  - All blocks in a kernel have the same entry point
  - Can execute any code they want

- Thread blocks of kernel must be independent tasks
- Execution of blocks are interleaved
- Program must be valid for any interleaving
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 execute in separate streams
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
void *myimage = cudaMalloc(bytes)

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

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

C/C++ CUDA Application → NVCC → CPU Code → PTX Code → PTX to Target Compiler
→ Target code (G80, ...) → 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 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 assigns 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

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

- Threads within a block may synchronize with barriers
  - __syncthreads();

- Blocks coordinate via atomic memory operations
  - atomicInc()

- Implicit barrier between dependent kernels
Memory Model

Thread

Per-thread Local Memory

float LocalVar;

Block

Per-block Shared Memory

__shared__ float SharedVar;

Sequential Kernels

Kernel 0

Kernel 1

...
Memory Model – CPU to/from GPU Device

Host Memory

Device 0 Memory

Device 1 Memory

cudaMemcpy()
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`, `sqrdf`, 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()`, `cudaD3D9MapVertexBuffers()`, ...

---

Tuesday, October 11, 11
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
  - First priority: make things work
  - Second priority: get performance
- However, to get good performance, one must understand how CUDA is mapped to 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 is scheduled onto a SM
  - Peak efficiency requires multiple thread blocks per SM
Mapping CUDA to NVIDIA GPUs (2)

- GPU is deeply pipelined to maximize throughput
- This means that performance depends on the #thread blocks which can be allocated
- 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
  - Look at the minimum # registers for full occupancy
SIMD and Control Flow

- NVIDIA GPU hardware handles control flow divergence and reconvergence
  - Write scale SIMD code
  - Hardware schedules SIMD execution
- Each thread execution the instruction
- What if it is a conditional statement?
  - Some threads might take TRUE branch
  - Some thread might take FALSE branch
  - This is called divergence
- Good performing code will try to keep the execution divergence within a warp
Memory is SIMD Too!

- Virtually all processors have SIMD memory subsystems

```
0 1 2 3 4 5 6 7
```

- cache line width

- This has two effects:
  - Sparse access wastes bandwidth
    ```
    0 1 2 3 4 5 6 7
    ```
    2 words used, 8 words loaded: 
    \( \frac{1}{4} \) effective bandwidth
  - Unaligned access wastes bandwidth
    ```
    0 1 2 3 4 5 6 7
    ```
    4 words used, 8 words loaded: 
    \( \frac{1}{2} \) effective bandwidth
Coalescing

- CPUs and GPUs both perform memory transactions at a larger granularity than data access
  - Each has caches
- GPUs have a “coalesce” 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 accesss)
  - 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
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
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

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

Pipeline Snapshot:
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
Typically, GPU devices are accessed over PCI.