Demo

- Performance gap between GPUs and multicore CPUs continues to widen
Flynn’s Taxonomy for Parallel Computers

<table>
<thead>
<tr>
<th></th>
<th>Single Instruction</th>
<th>Multiple Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

**Single Instruction, Single Data stream (SISD)**
A sequential computer which exploits no parallelism in either the instruction or data streams. E.g., old single processor PC

**Single Instruction, Multiple Data streams (SIMD)**
A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. E.g. graphics processing unit

**Multiple Instruction, Single Data stream (MISD)**
Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. E.g. the Space Shuttle flight control computer.

**Multiple Instruction, Multiple Data streams (MIMD)**
Multiple autonomous processors simultaneously executing different instructions on different data. E.g. a PC cluster memory space.

Multicore Processors are examples of MIMD systems

- Memory hierarchy for a single Intel Xeon Quad-core E5530 processor chip

![Memory hierarchy diagram for Intel Xeon Quad-core E5530 processor chip]
SIMD computers

- Definition: A single instruction stream is applied to multiple data elements.
  - One program text
  - One instruction counter
  - Distinct data streams per Processing Element (PE)
- Examples: Vector Processors, GPUs

“CPU-Style” Cores

The “CPU-Style” core is designed to make individual threads speedy.

“Execution context” == memory and hardware associated to a specific stream of instructions (e.g. a thread)
Multiple cores lead to MIMD computers
GPU Design Idea #1: more slow cores

The first big idea that differentiates GPU and CPU core design: slim down the footprint of each core.

Idea #1:
Remove the modules that help a single instruction execute fast.

Slides and graphics based on presentations from Andreas Klöckner and Kayvon Fatahalian

GPU Design Idea #1: more slow cores

See: Andreas Klöckner and Kayvon Fatahalian
GPU Design Idea #2: lock stepping

In the GPU rendering context, the instruction streams are typically very similar.

Design for a “single instruction multiple data” SIMD model: share the cost of the instruction stream across many ALUs.

See: Andreas Klöckner and Kayvon Fatahalian

---

GPU Design Idea #2: branching?

Question:

What happens when the instruction streams include branching?

How can they execute in lock step?

See: Andreas Klöckner and Kayvon Fatahalian
GPU Design Idea #2: lock stepping w/ branching

Non branching code;
if(flag > 0){ /* branch */
    x = exp(y);
    y = 2.3*x;
} 
else{
    x = sin(y);
    y = 2.1*x;
}
Non branching code;
The cheap branching approach means that some ALUs are idle as all ALUs traverse all branches [executing NOPs if necessary]
In the worst possible case we could see 1/8 of maximum performance.

GPU Design Idea #3: latency hiding

work on registers; work on registers; work on registers; 
load registers from main memory; 
It takes O(1000) cycles to load data from off chip memory into the SM registers file
These ALUs are idled (stalled) after a load

See: Andreas Klöckner and Kayvon Fatahalian
**GPU Design Idea #3: latency hiding**

Idea #3: enable fast context switching so the ALUs can efficiently alternate between different tasks.

See: Andreas Klöckner and Kayvon Fatahalian

---

**GPU Design Idea #3: context switching**

<table>
<thead>
<tr>
<th>Ctx1: work on registers;</th>
<th>Ctx1: load request, switch context;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctx1: work on registers;</td>
<td></td>
</tr>
<tr>
<td>Ctx1: work on registers;</td>
<td></td>
</tr>
<tr>
<td>Ctx1: load request, switch context;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ctx3: work on registers;</th>
<th>Ctx3: work on registers;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctx3: load request, switch context;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ctx2: work on registers;</th>
<th>Ctx2: load request, switch context;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctx2: work on registers;</td>
<td></td>
</tr>
<tr>
<td>Ctx2: work on registers;</td>
<td></td>
</tr>
</tbody>
</table>

See: Andreas Klöckner and Kayvon Fatahalian
Summary: CPUs and GPUs have fundamentally different design

**GPU = Graphics Processing Unit**

<table>
<thead>
<tr>
<th>Single CPU core</th>
<th>Multiple GPU processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Streaming Multiprocessor</td>
</tr>
<tr>
<td>ALU</td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td></td>
</tr>
<tr>
<td>Cache</td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>DRAM</td>
</tr>
</tbody>
</table>

GPUs are provided to accelerate graphics, but they can also be used for non-graphics applications that exhibit large amounts of data parallelism and require large amounts of “streaming” throughput ⇒ SIMD parallelism within an SM, and SPMD parallelism across SMs

**Host vs. Device**

- CPU (aka host)
- GPU (aka device)
Host vs. Device

- The GPU has its own independent memory space.
- The GPU brick is a separate compute sidecar.
- We refer to:
  - the GPU as a “DEVICE”
  - the CPU as the “HOST”
- An array that is in HOST-attached memory is not directly visible to the DEVICE, and vice versa.
- To load data onto the DEVICE from the HOST:
  - We allocate memory on the DEVICE for the array
  - We then copy data from the HOST array to the DEVICE array
- To retrieve results from the DEVICE they have to be copied from the DEVICE array to the HOST array.

Execution of a CUDA program

- Host Code (small number of threads)
- Explicit host-device communication
- Device Kernel (large number of threads)
- Explicit host-device communication
- Host Code (small number of threads)
- Explicit host-device communication
- Device Kernel (large number of threads)
- Explicit host-device communication
- Host Code (small number of threads)
Outline of a CUDA main program

pseudo_cuda_code.cu:

```c
__global__ void kernel(arguments) {
    instructions for a single GPU thread;
}
...
main(){
    set up GPU arrays;
    copy CPU data to GPU;
    kernel <<< # thread blocks, # threads per block >>> (arguments);
    copy GPU data to CPU;
}
```

CUDA Storage Classes + Thread Hierarchy

- **Local Memory**: per-thread
  - Private per thread
  - Auto variables, register spill
- **Shared Memory**: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- **Global Memory**: per-application
  - Shared by all threads
  - Inter-Grid communication

Sequential Grids in Time
Matrix multiplication kernel code in CUDA --- SPMD model with 2D index

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Width + k];
        float Ndelement = Nd[k * Width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

Host Code in C for Matrix Multiplication

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
    int size = Width*Width*sizeof(float); // matrix size
    float* Md, Nd, Pd; // pointers to device arrays
    cudaMalloc((void**)&Md, size); // allocate Md on device
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice); // copy M to Md
    cudaMalloc((void**)&Nd, size); // allocate Nd on device
    cudaMemcpy(Nd, M, size, cudaMemcpyHostToDevice); // copy N to Nd
    cudaMalloc((void**)&Pd, size); // allocate Pd on device
    dim3 dimBlock(Width,Width); dim3 dimGrid(1,1);
    // launch kernel (equivalent to “async at(GPU), forall, forall”
    MatrixMulKernel<<<dimGrid,dimBlock>>>(Md, Nd, Pd, Width);
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost); // copy Pd to P
    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
CUDA Host-Device Data Transfer

- cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)
- copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind is one of
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- The memory areas may not overlap
- Calling cudaMemcpy() with dst and src pointers that do not match the direction of the copy results in an undefined behavior.

Summary of key features in CUDA

<table>
<thead>
<tr>
<th>CUDA construct</th>
<th>Related HJ/Java constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel invocation, [ \ldots ]</td>
<td>async at(gpu-place)</td>
</tr>
<tr>
<td>1D/2D grid with 1D/2D/3D blocks of threads</td>
<td>Outer 1D/2D forall with inner 1D/2D/3D forall</td>
</tr>
<tr>
<td>Intra-block barrier, __syncthreads()</td>
<td>HJ forall-next on implicit phaser for inner forall</td>
</tr>
<tr>
<td>cudaMemcpy()</td>
<td>No direct equivalent in HJ/Java (can use System.arraycopy() if needed)</td>
</tr>
<tr>
<td>Storage classes: local, shared, global</td>
<td>No direct equivalent in HJ/Java (method-local variables are scalars)</td>
</tr>
</tbody>
</table>
Worksheet #35: Branching in SIMD code

Consider SIMD execution of the following pseudocode with 8 threads. Assume that each call to doWork(x) takes x units of time, and ignore all other costs. How long will this program take when executed on 8 GPU cores, taking into consideration the branching issues discussed in Slide 9?

1. int tx = threadIdx.x; // ranges from 0 to 7
2. if (tx % 2 = 0) {
3.   S1: doWork(1); // Computation S1 takes 1 unit of time
4. }
5. else {
6.   S2: doWork(2); // Computation S2 takes 2 units of time
7. }
HJ abstraction of a CUDA kernel invocation: async at + forall + forall