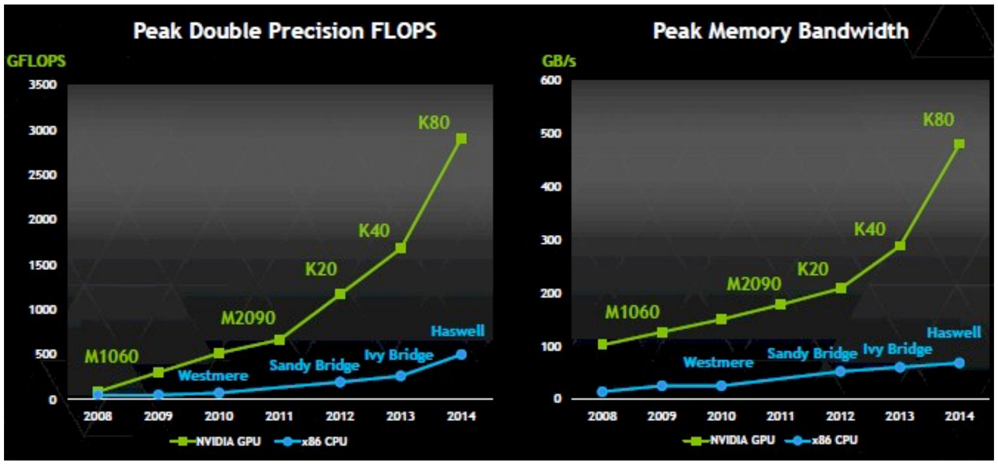
COMP 322: Fundamentals of Parallel Programming

Lecture 35: General-Purpose GPU (GPGPU) Computing

Max Grossman, Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University
max.grossman@rice.edu, vsarkar@rice.edu, shams@rice.edu

comp322.rice.edu

Demo



- Performance gap between GPUs and multicore CPUs continues to widen

Flynn's Taxonomy for Parallel Computers

	Single Instruction	Multiple Instructions
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

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

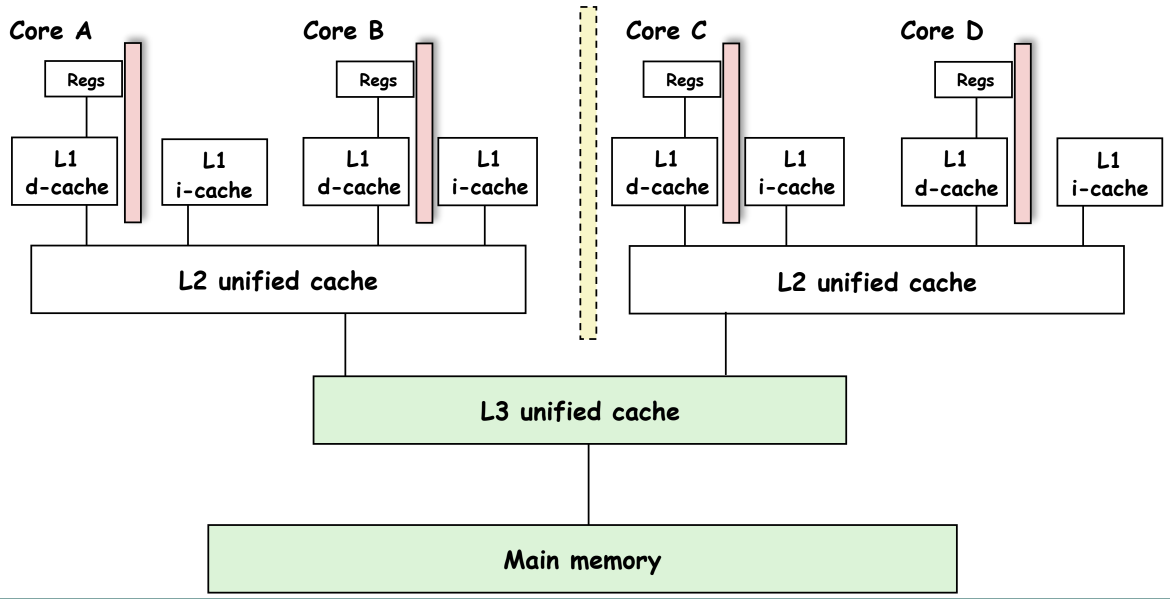
Single Instruction, Multiple Data streams (SIMD)
A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. e.g. graphics processing unit

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.

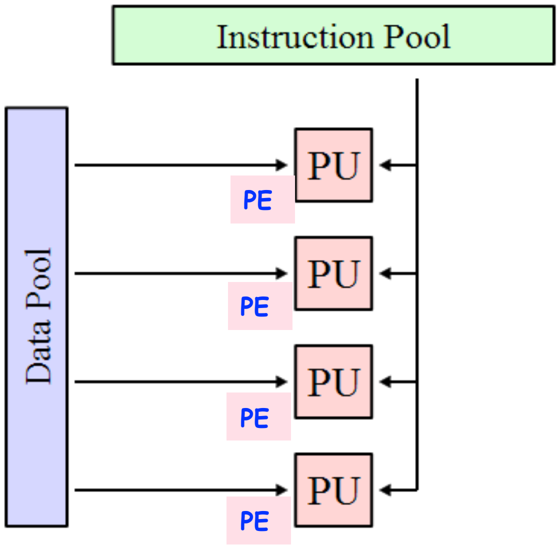
Multicore Processors are examples of MIMD systems

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



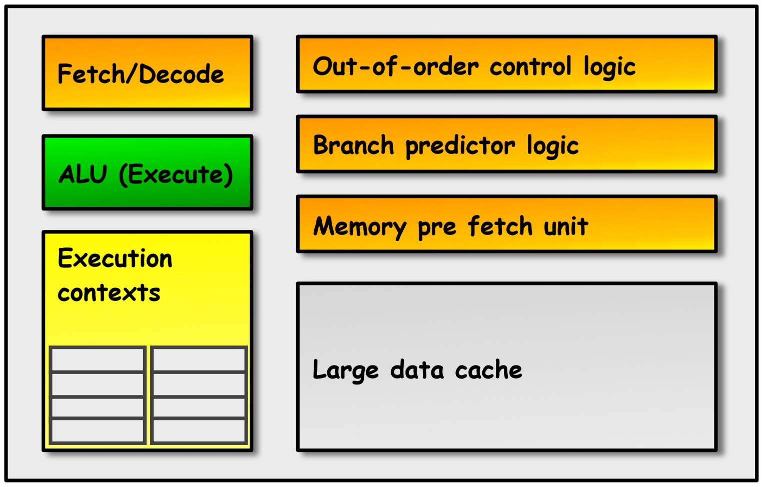
SIMD computers

- Definition: A single instruction stream is applied to multiple data elements.
  - One program text
  - One instruction counter
  - Distinct data streams per Proc
- Examples: Vector Processors, GPU



"CPU-Style" Cores

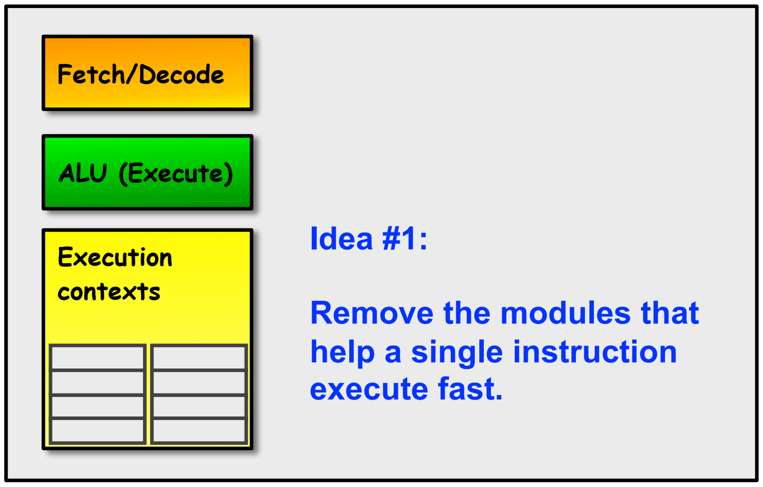
The "CPU-Style" core is designed to make individual threads speedy.



"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.

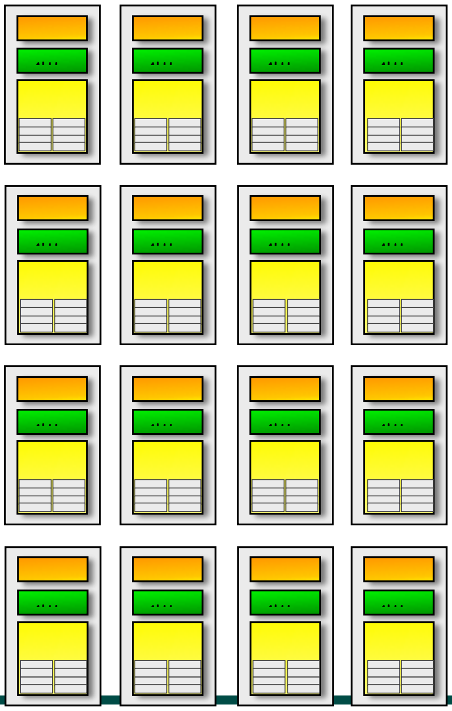


Idea #1:

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

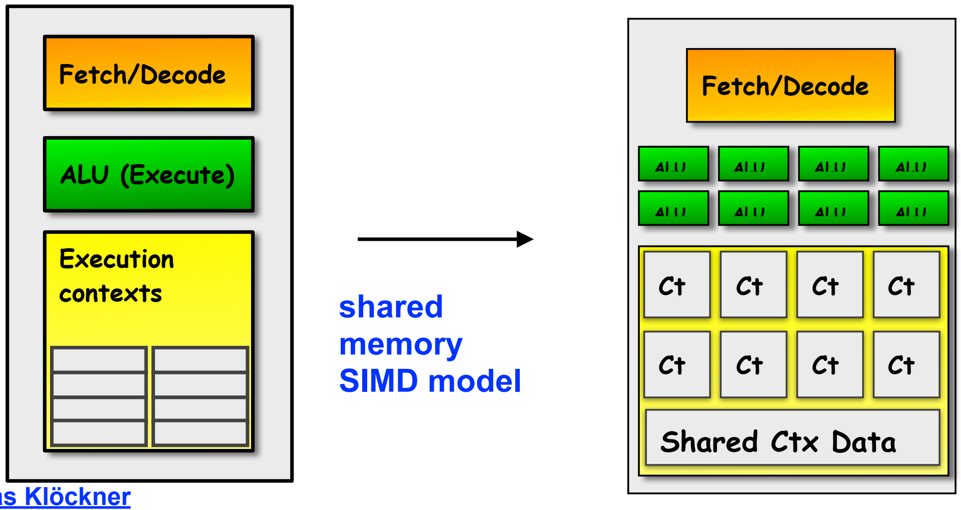


See: Andreas Klöckner
and Kayvon Fatahalian

GPU Design Idea #2: lock stepping

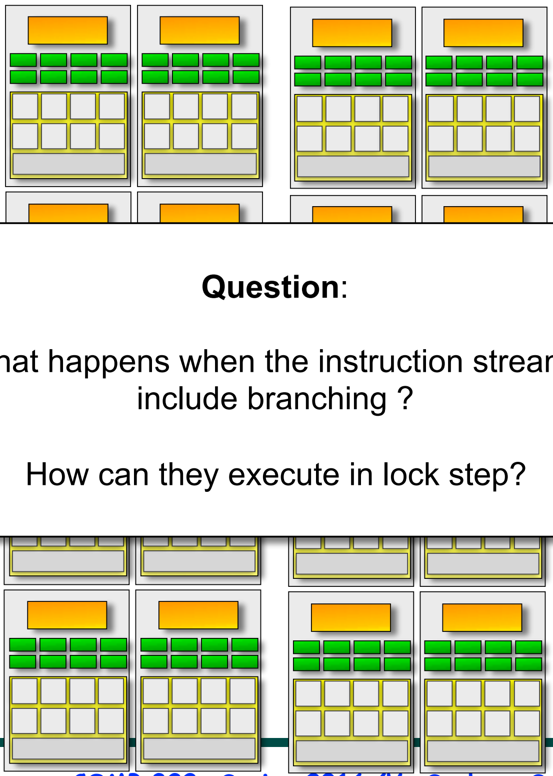
In the GPU rendering context, the instruction streams are typically very similar.

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 ?



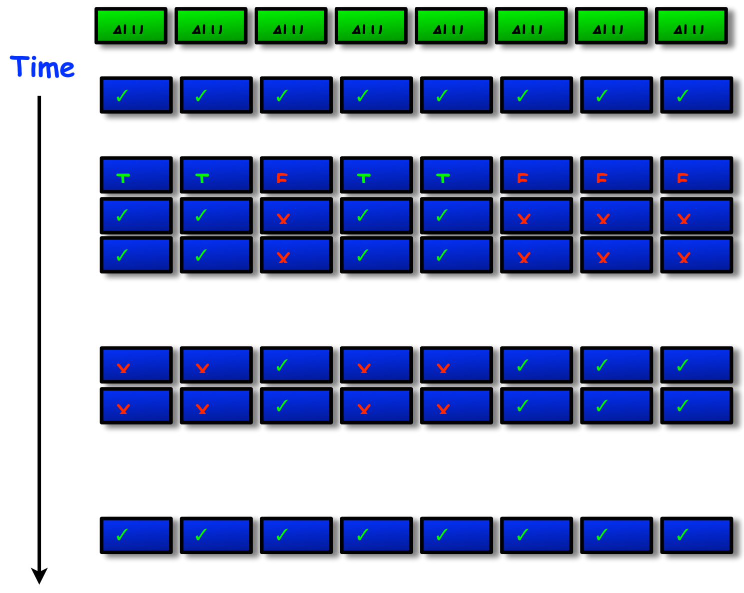
Question:

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;

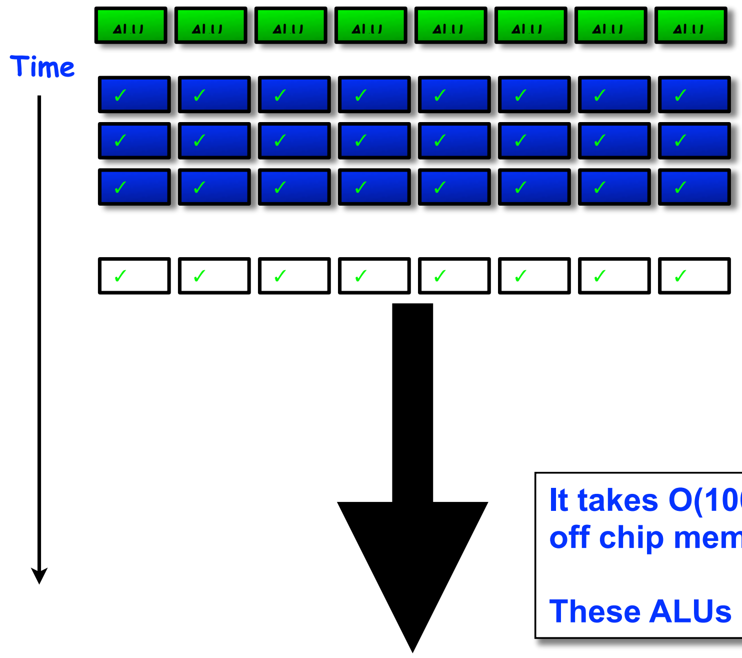
```
if(flag > 0){ /* branch */
  x = exp(y);
  y = 2.3*x;
}
else{
  x = sin(y);
  y = 2.1*x;
}
```

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;

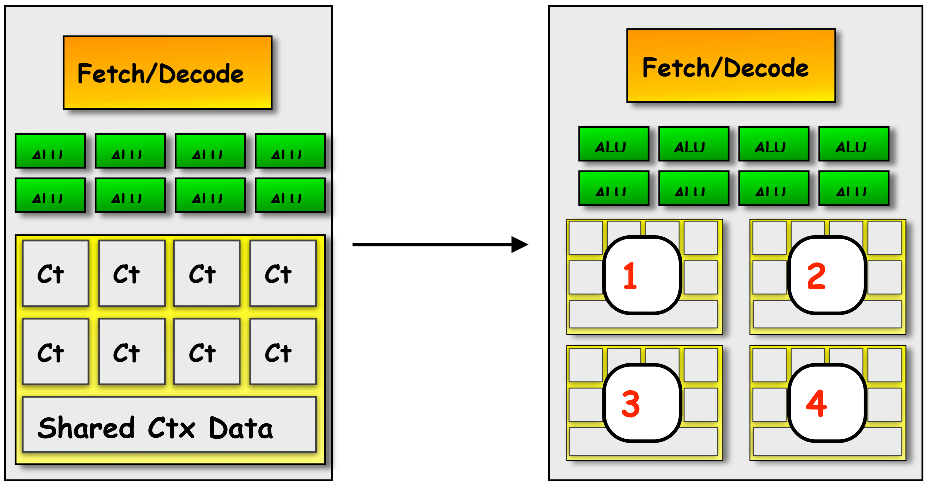
It takes O(1000) cycles to load data from off chip memory into the SM registers file

These ALUs are idled (stalled) after a load

See: Andreas Klöckner and Kayvon Fatahalian

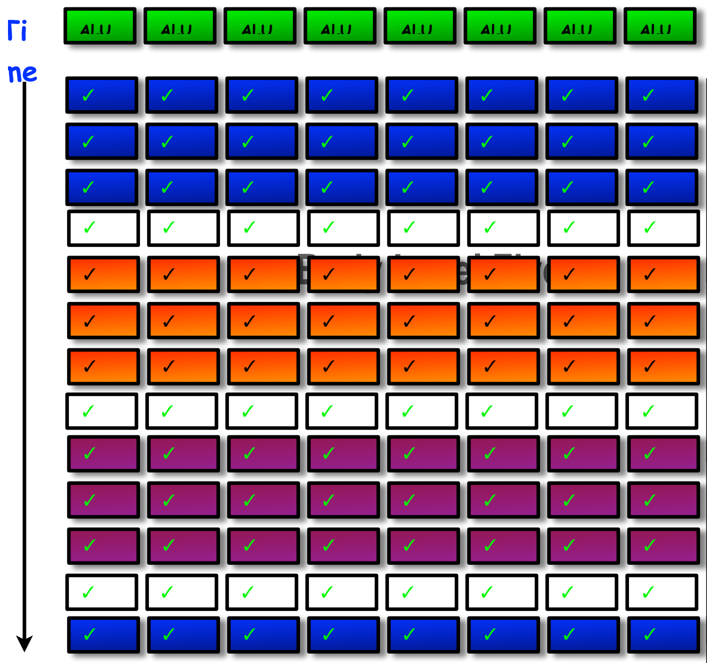
GPU Design Idea #3: latency hiding

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



Ctx1: work on registers;
Ctx1: work on registers;
Ctx1: work on registers;
Ctx1: load request, switch context;

Ctx3: work on registers;
Ctx3: work on registers;
Ctx3: work on registers;
Ctx3: load request, switch context;

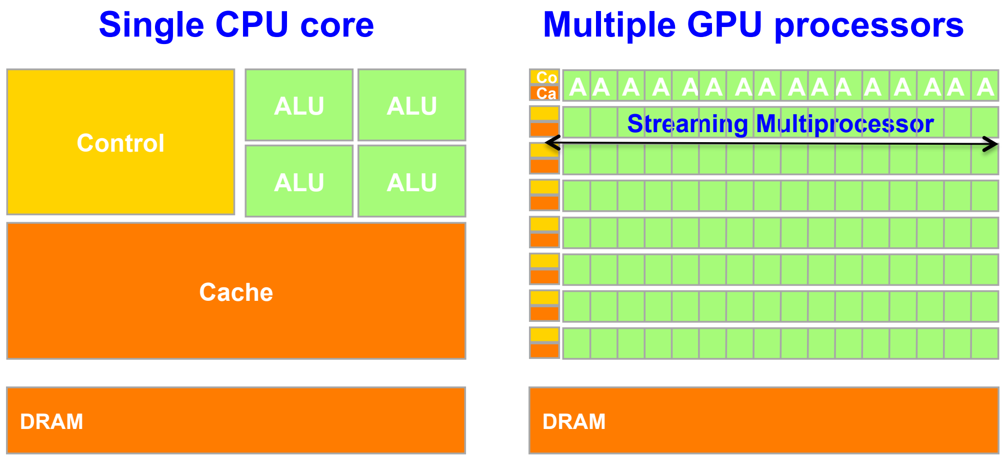
Ctx2: work on registers;
Ctx2: work on registers;
Ctx2: work on registers;
Ctx2: load request, switch context;

Ctx1: load done so continue

See: Andreas Klöckner and Kayvon Fatahalian

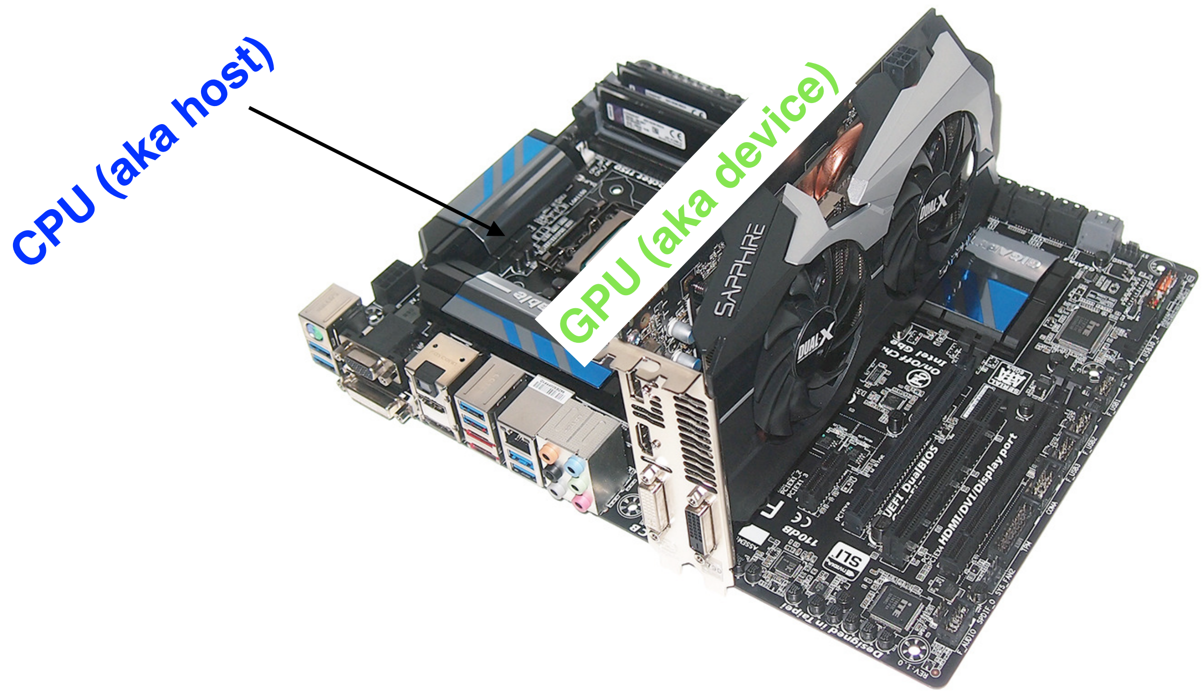
Summary: CPUs and GPUs have fundamentally different design

GPU = Graphics Processing Unit



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

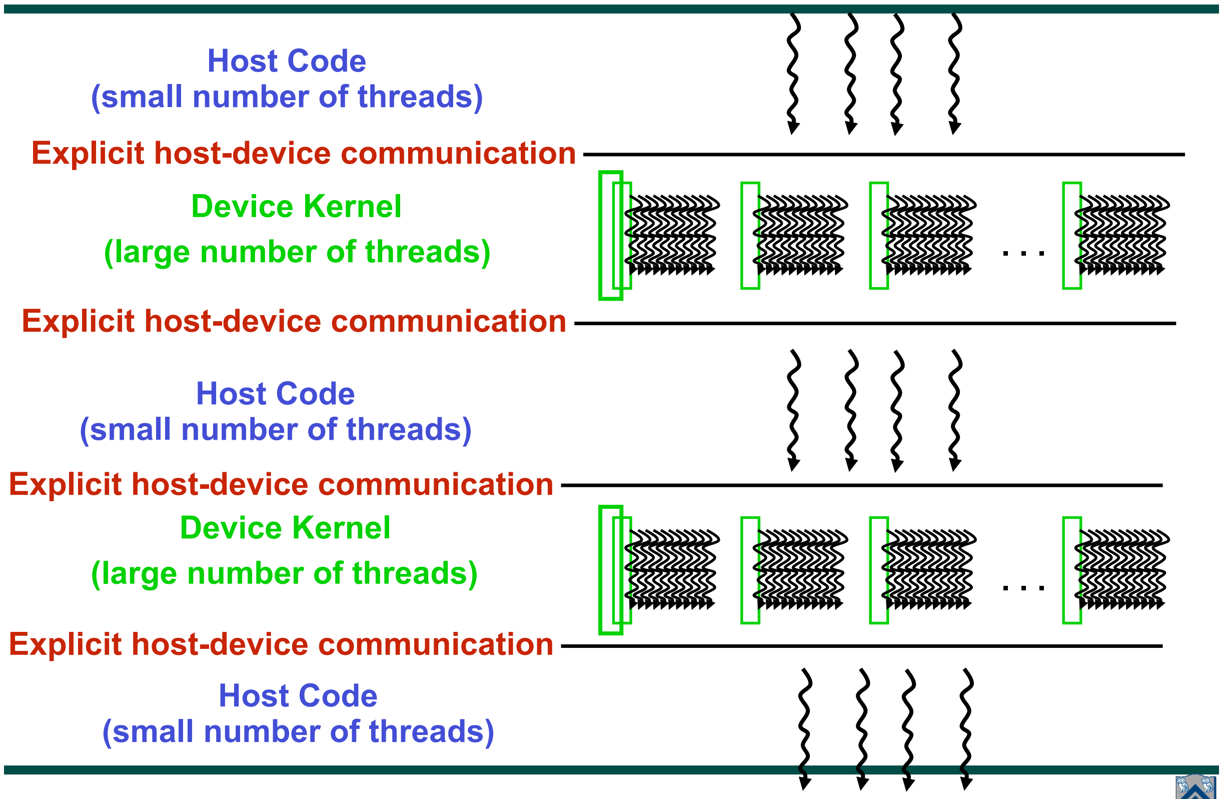
Host vs. Device



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.

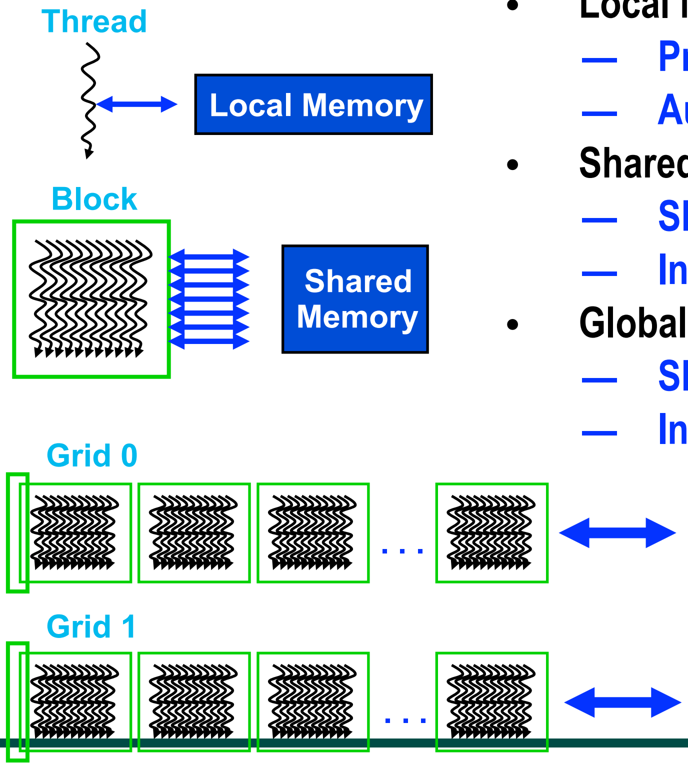
Execution of a CUDA program


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:

__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 bl
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication

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

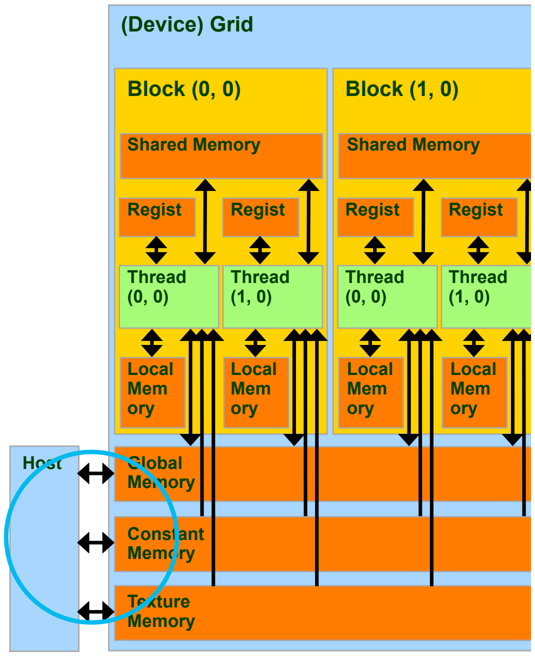
```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   // 2D Thread ID
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // 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

- cudaError_t 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

CUDA construct	Related HJ/Java constructs
Kernel invocation, <<<. . .>>>	async at(gpu-place)
1D/2D grid with 1D/2D/3D blocks of threads	Outer 1D/2D forall with inner 1D/2D/3D forall
Intra-block barrier, __syncthreads()	HJ forall-next on implicit phaser for inner forall
cudaMemcpy()	No direct equivalent in HJ/Java (can use System.arraycopy() if needed)
Storage classes: local, shared, global	No direct equivalent in HJ/Java (method-local variables are scalars)

Worksheet #35: Branching in SIMD code

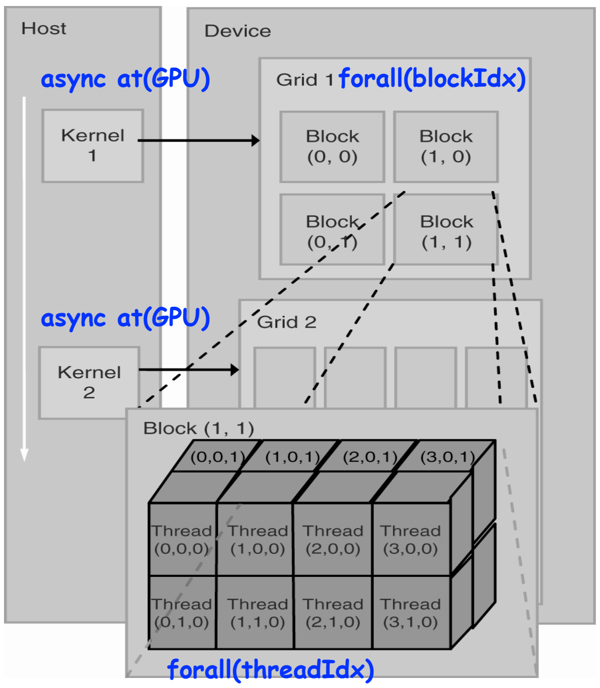
Name: ______________________ Netid: ______________________

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?

```
int tx = threadIdx.x; // ranges from 0 to 7
if (tx % 2 = 0) {
  S1: doWork(1); // Computation S1 takes 1 unit of time
}
else {
  S2: doWork(2); // Computation S2 takes 2 units of time
}
```

BACKUP SLIDES START HERE

HJ abstraction of a CUDA kernel invocation: async at + forall + forall



Host
Device
async at(GPU)
Kernel 1
Grid 1
forall(blockIdx)
Block (0, 0)
Block (1, 0)
Block (0, 1)
Block (1, 1)
async at(GPU)
Kernel 2
Grid 2
Block (1, 1)
(0,0,1)
(1,0,1)
(2,0,1)
(3,0,1)
Thread (0,0,0)
Thread (1,0,0)
Thread (2,0,0)
Thread (3,0,0)
Thread (0,1,0)
Thread (1,1,0)
Thread (2,1,0)
Thread (3,1,0)
forall(threadIdx)