COMP 322: Fundamentals of Parallel Programming

Lecture 34: General-Purpose GPU (GPGPU) Computing

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Block Distribution (Recap)

- A block distribution splits a region into contiguous subregions, one per place, while trying to keep the subregions as close to equal in size as possible.
- Block distributions can improve the performance of parallel loops that exhibit spatial locality across contiguous iterations.
- Example: block distribution of [0:15] across 4 places

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Place id	0				1				2				3			

Cyclic Distribution (Recap)

- A cyclic distribution "cycles" through places 0 … place.MAX PLACES − 1 when spanning the input region
- Cyclic distributions can improve the performance of parallel loops that exhibit load imbalance
- Example: cyclic distributions of [0:15] and [0:1,0:7] across 4 places

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Place id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

Index	[0,0]	[0,1]	[1,0]	[1,1]	[2,0]	[2,1]	[3,0]	[3,1]	[4,0]	[4,1]	[5,0]	[5,1]	[6,0]	[6,1]	[7,0]	[7,1]
Place id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

Worksheet #33 solution: impact of distribution on parallel completion time (instead of locality)

```
1.  public void sampleKernel(
2.      int iterations, int numChunks, Dist d) {
3.    for (int iter = 0; iter < iterations; iter++) {
4.      finish(() -> {
5.        forseq (0, numChunks - 1, (jj) -> {
6.          asyncAt(dist.get(jj), () -> {
7.            perf.doWork(jj);
8.            // Assume that time to process chunk jj = jj units
9.          });
10.       });
11.     });
12.      double[] temp = myNew; myNew = myVal; myVal = temp;
13.   } // for iter
14. } // sample kernel
```

- Assume an execution with n places, each place with one worker thread
- Will a block or cyclic distribution for d have a smaller abstract completion time, assuming that all tasks on the same place are serialized?

Answer: Cyclic distribution because it leads to better load balance (locality was not a consideration in this problem)

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

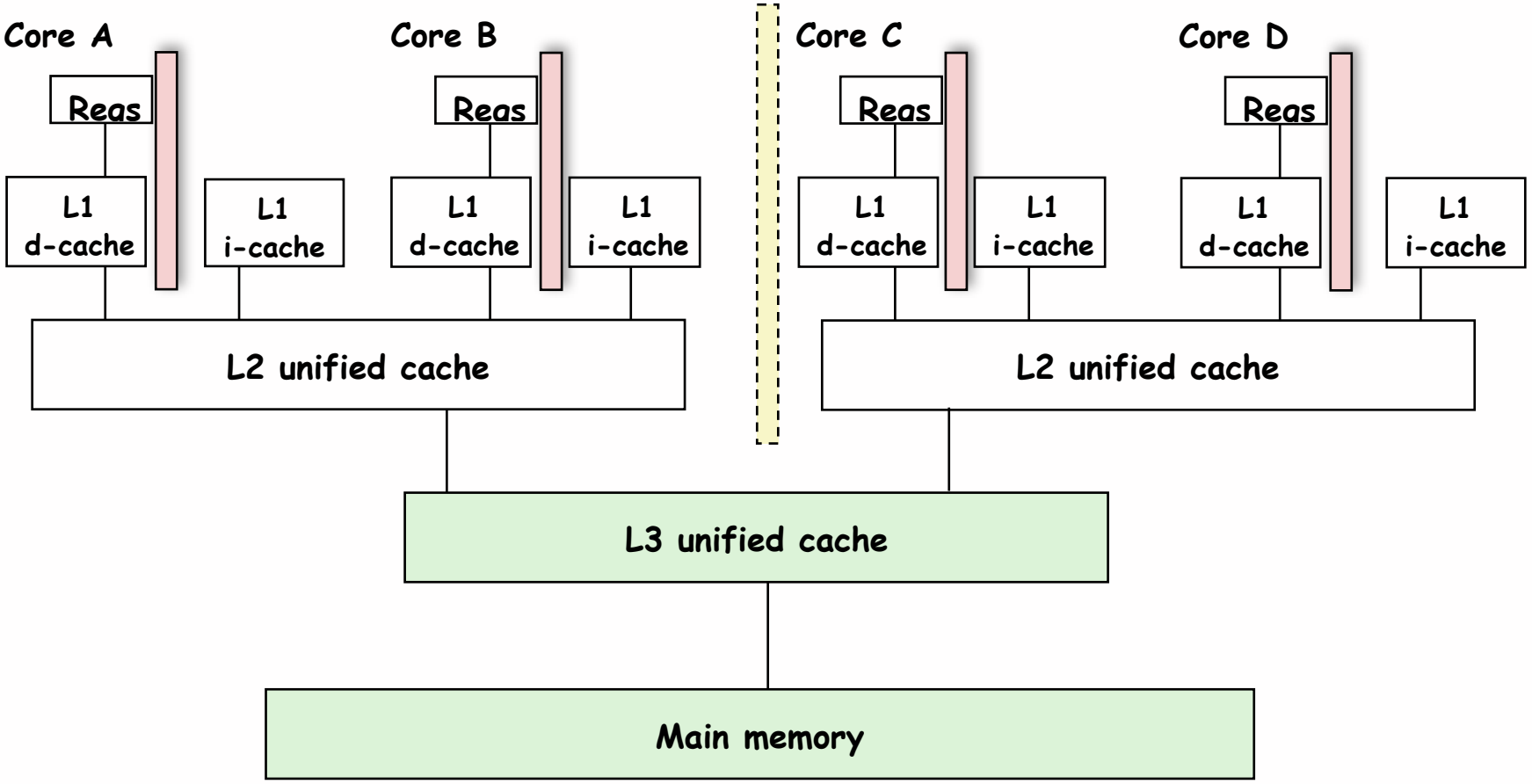
Multiple Instruction, Single Data stream (MISD)

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.

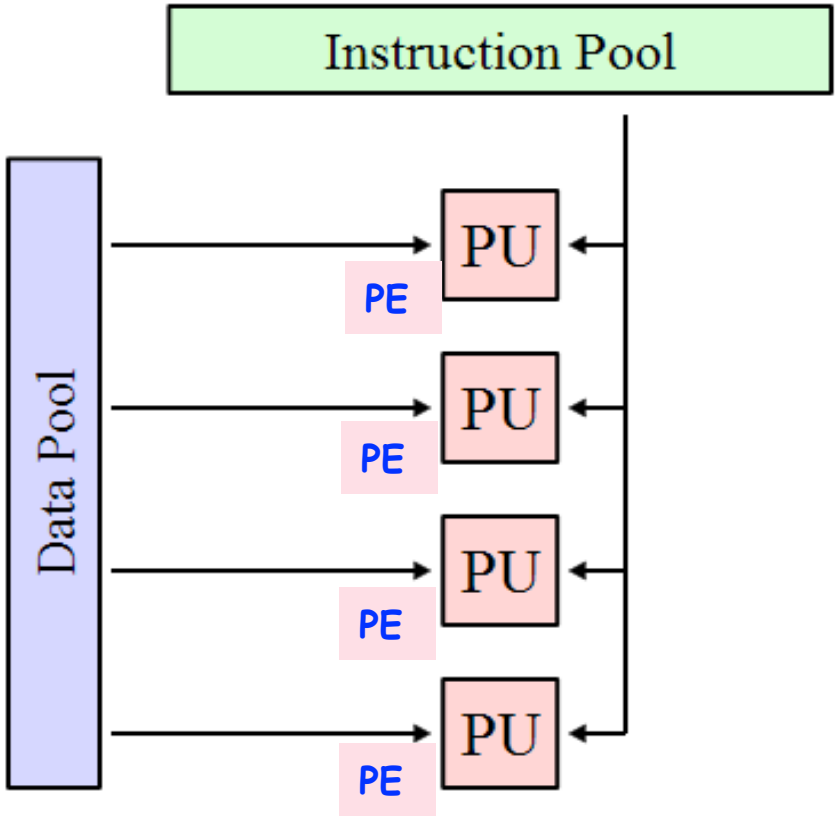
Multicore Processors are examples of MIMD systems



- Memory hierarchy for a single Intel Xeon Quad-core E5440 HarperTown processor chip
  - —A SUG@R node contains TWO such chips, for a total of 8 cores

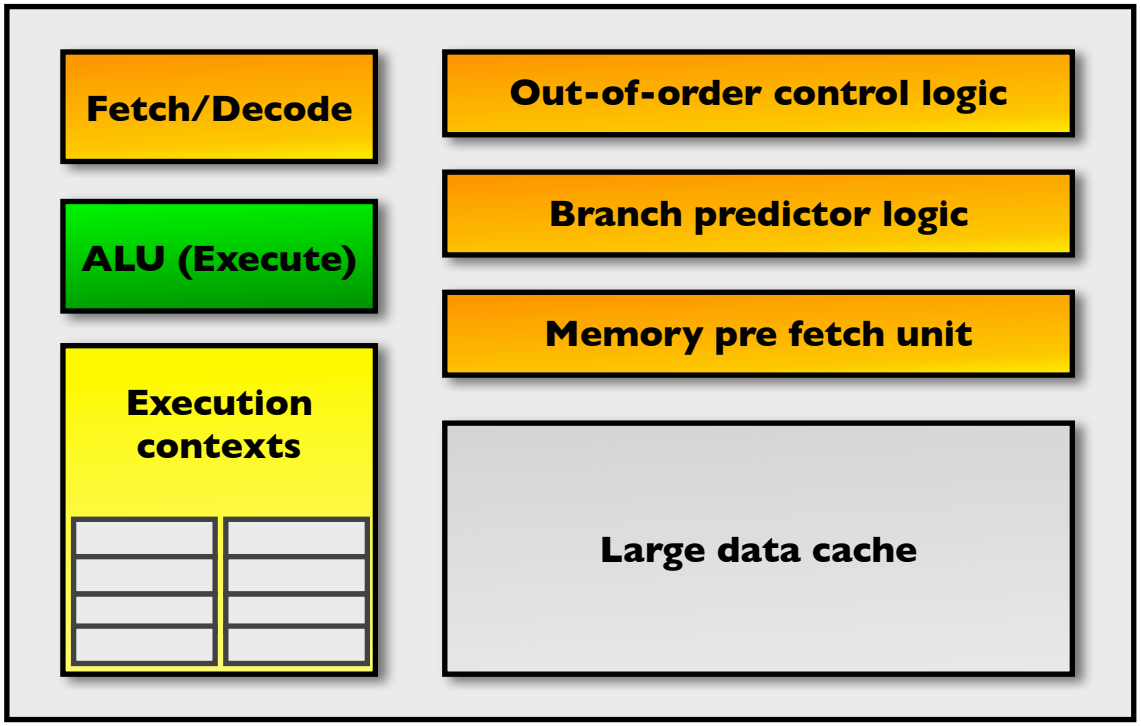
SIMD computers

- **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.



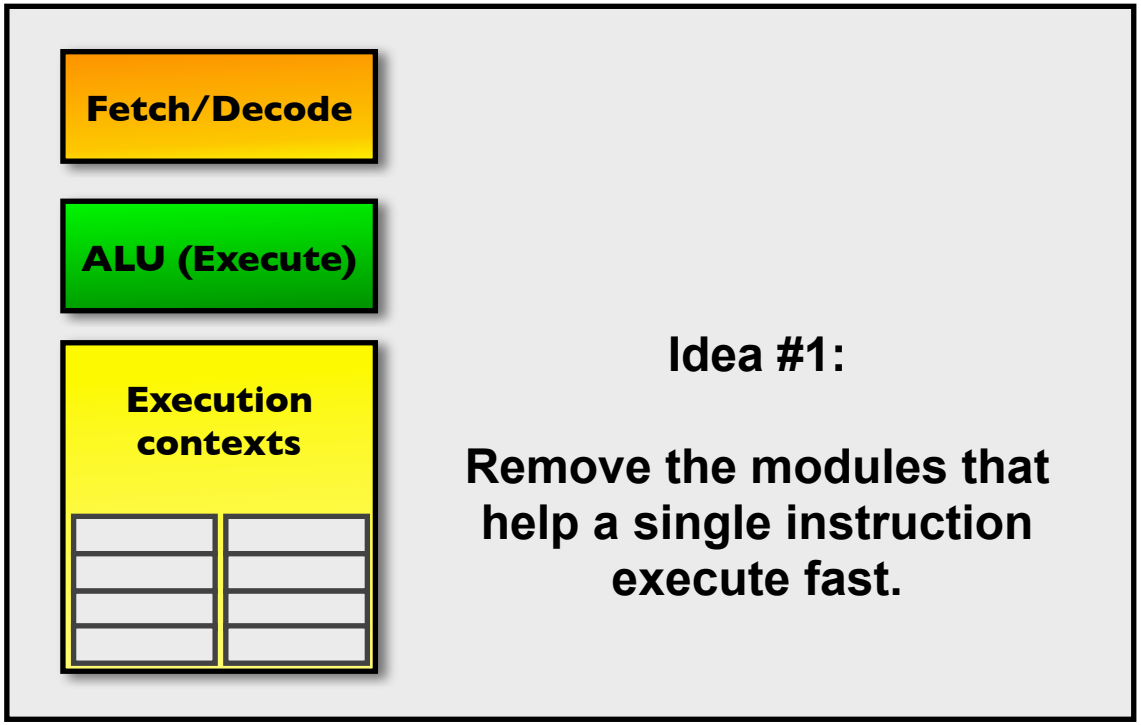
“Execution context” == memory and hardware associated
to a specific stream of instructions (e.g. a thread)
Multiple cores lead to MIMD computers

RICE

Graphics adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

GPU Design Idea #1: more slow cores

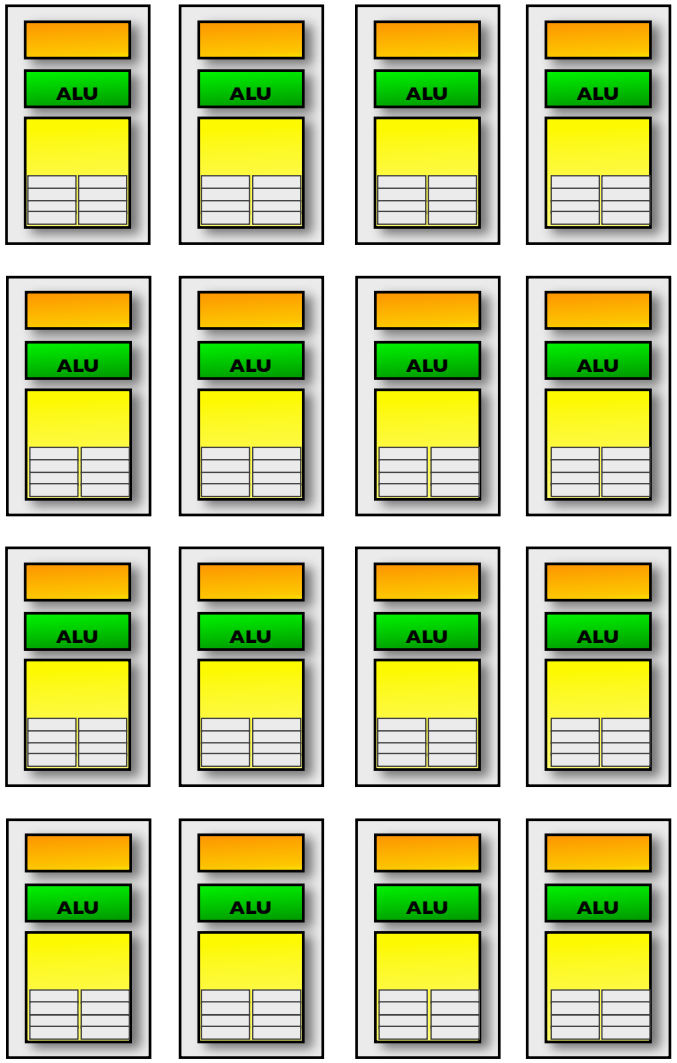
The first big idea that differentiates GPU and CPU core design:
slim down the footprint of each core.



RICE

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

GPU Design Idea #1: more slow cores



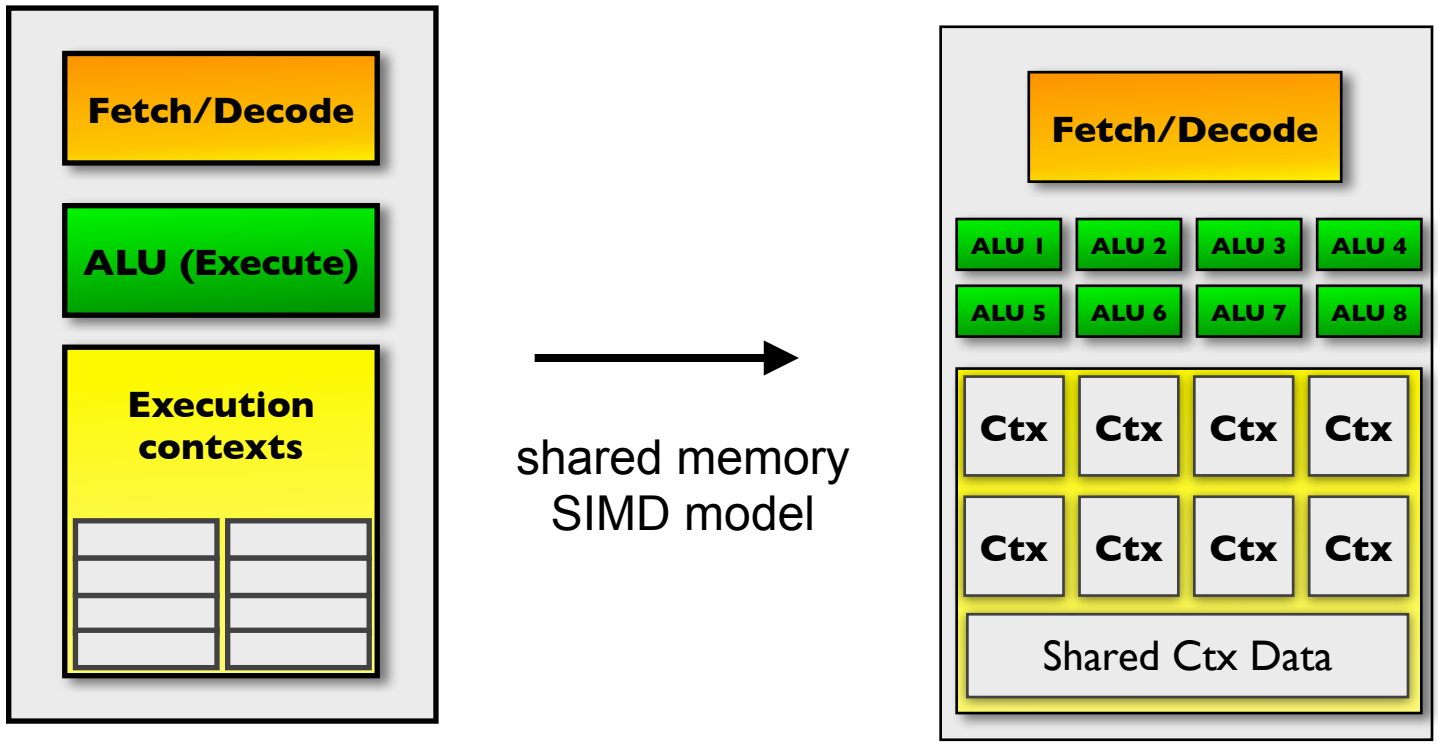
RICE

See: Andreas Klöckner and Kayvon Fatahalian

GPU Design Idea #2: lock stepping

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

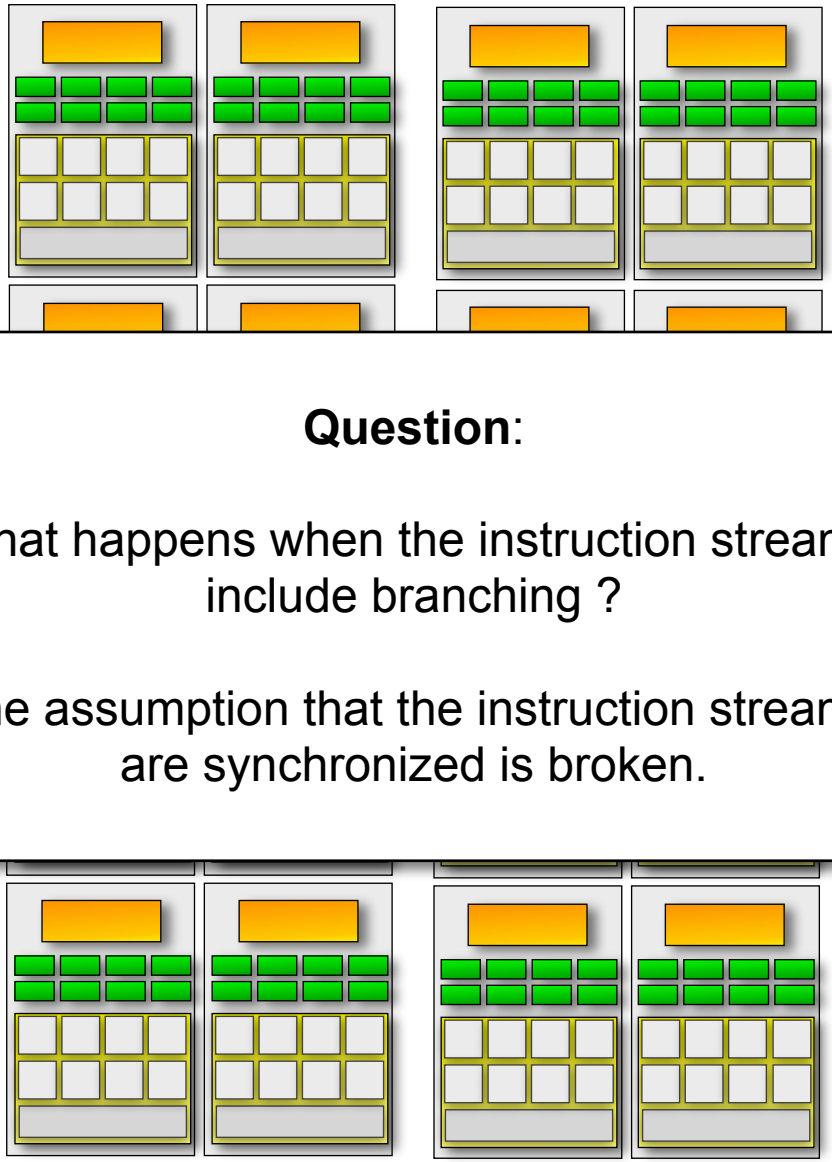
Design for a "single instruction multiple data" **SIMD** model: share the cost of the instruction stream across many ALUs



RICE

See: Andreas Klöckner and Kayvon Fatahalian

GPU Design Idea #2: branching ?



Question:

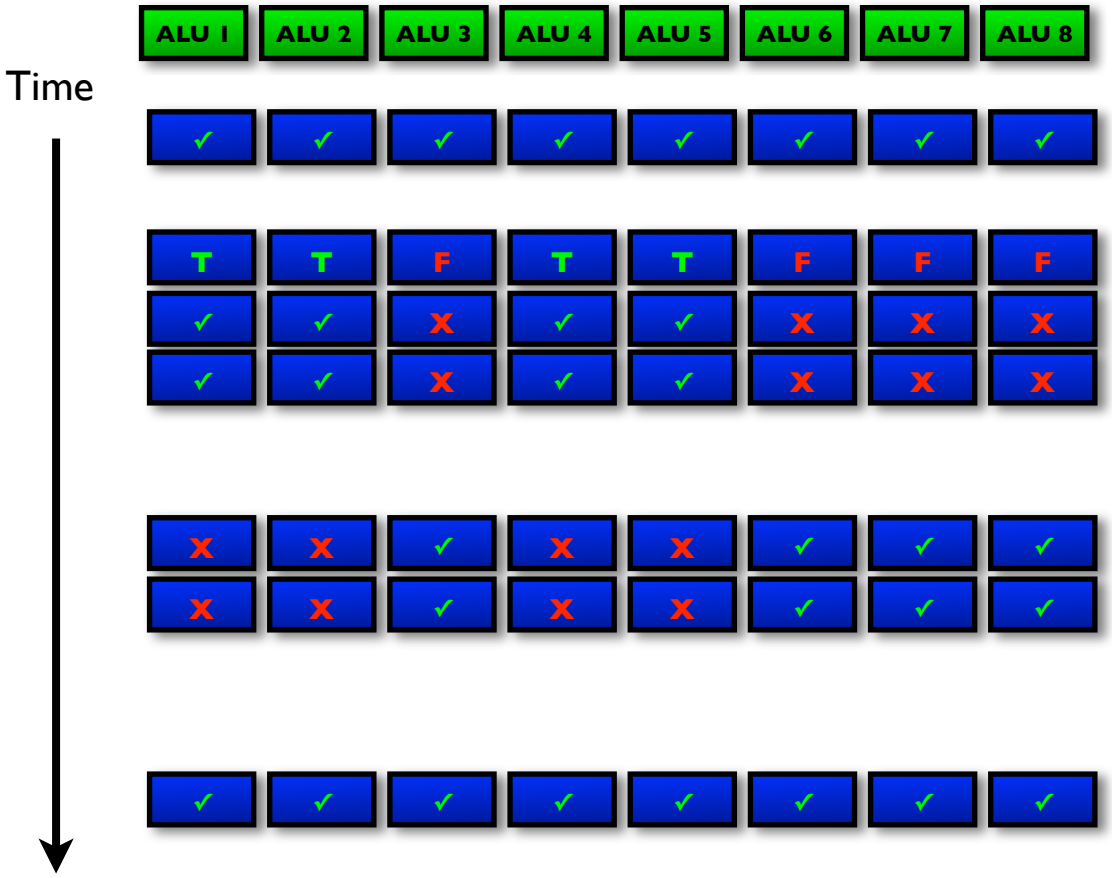
What happens when the instruction streams include branching ?

The assumption that the instruction streams are synchronized is broken.

RICE

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;
```

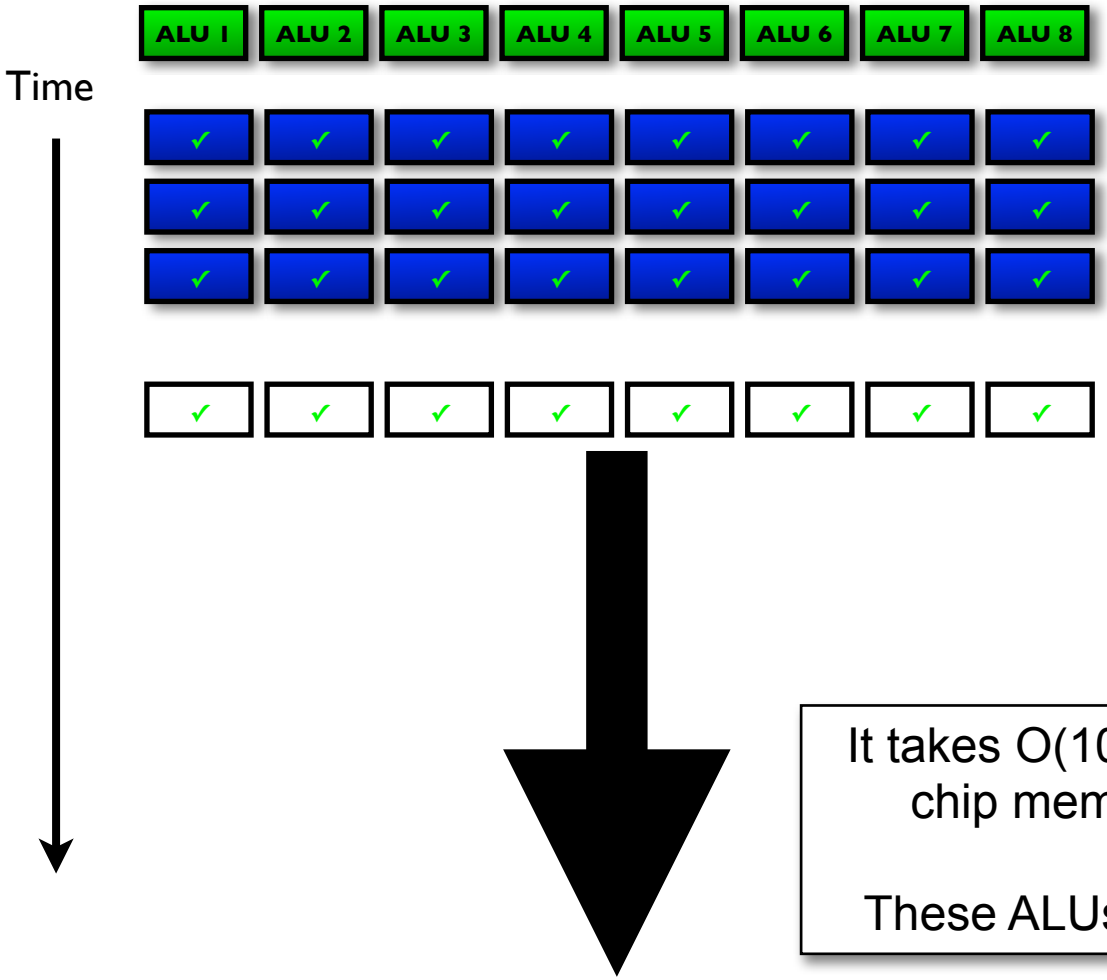
The cheap branching approach means that some ALUs are idle as all ALUs traverse all branches [executing NOPs if necessary]

In the worst possible case we could see 1/8 of maximum performance.

RICE

See: Andreas Klöckner and Kayvon Fatahalian

GPU Design Idea #3: stalls



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

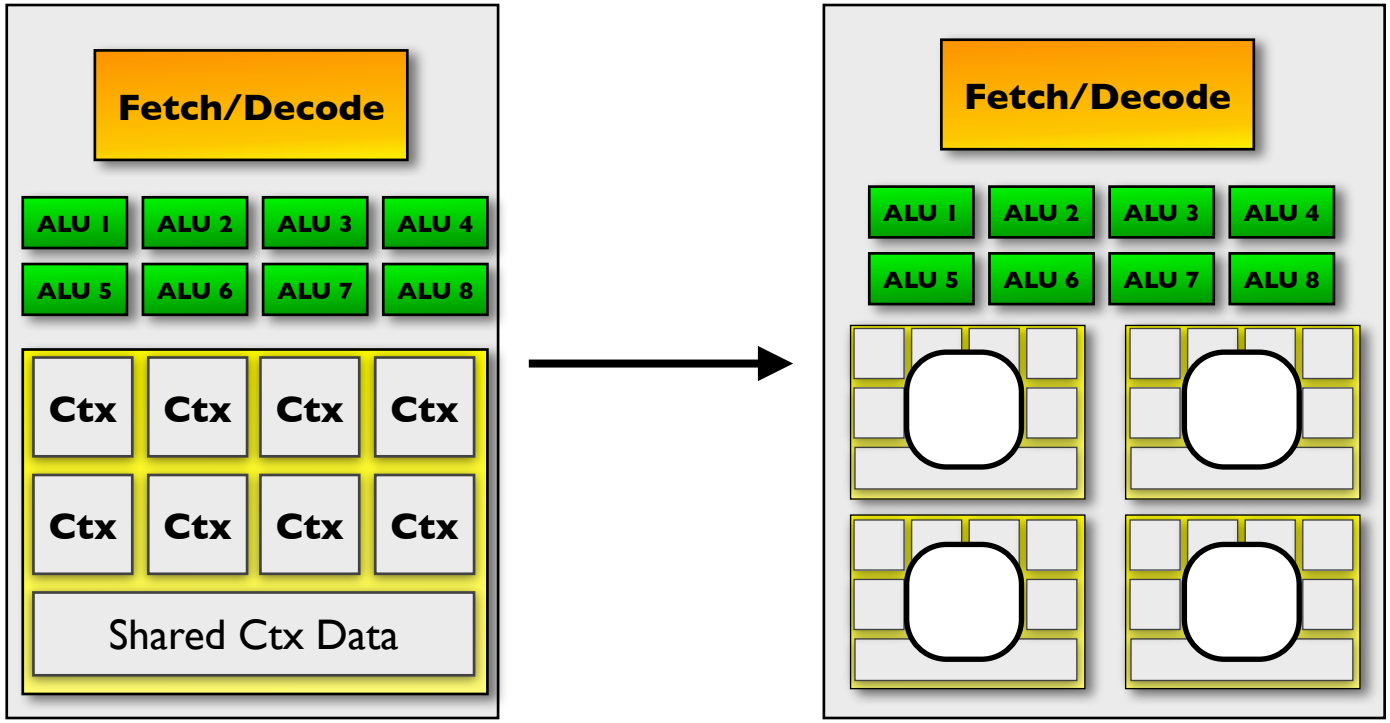
These ALUs are idled (stalled) after a load

RICE

See: Andreas Klöckner and Kayvon Fatahalian

GPU Design Idea #3: context switching

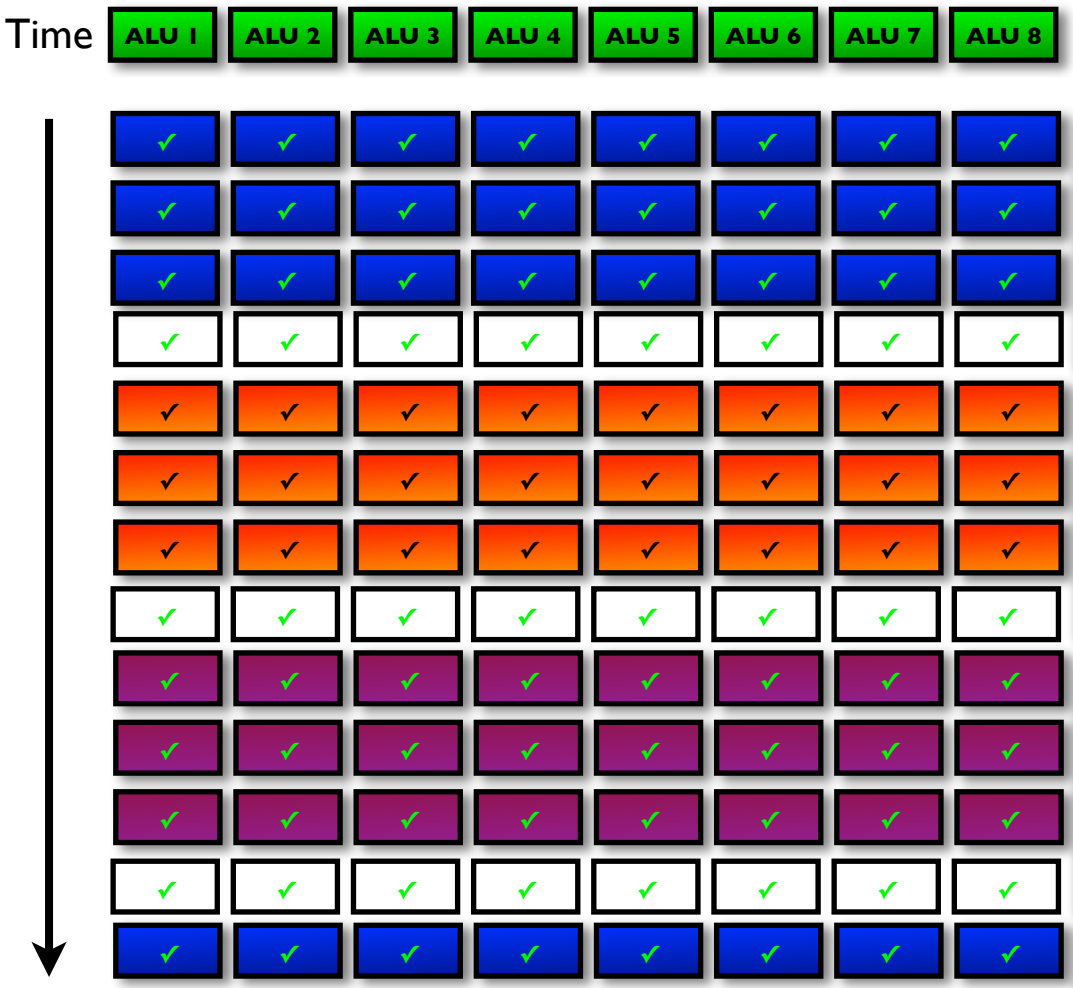
Idea #3: enable fast context switching so the ALUs can efficiently alternate between different tasks.



RICE

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;

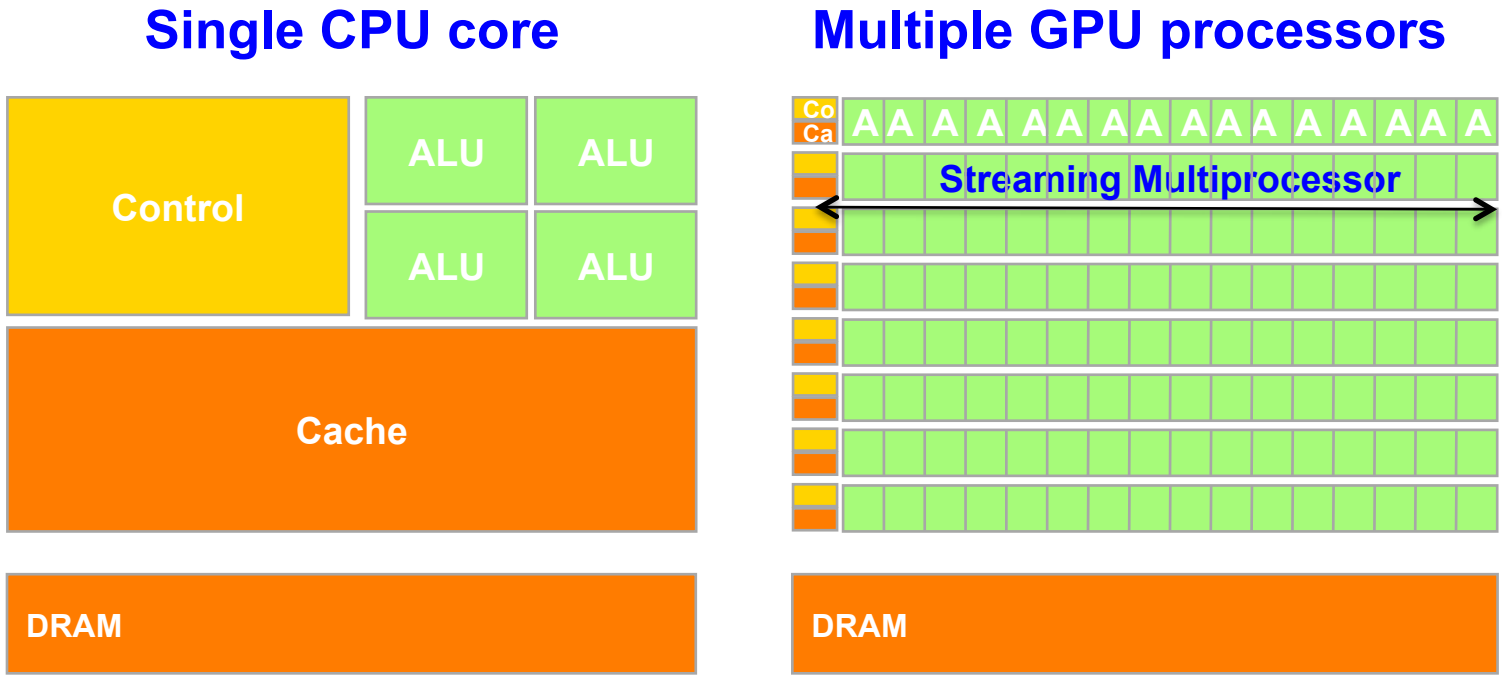
Ctx1: load done so continue

RICE

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

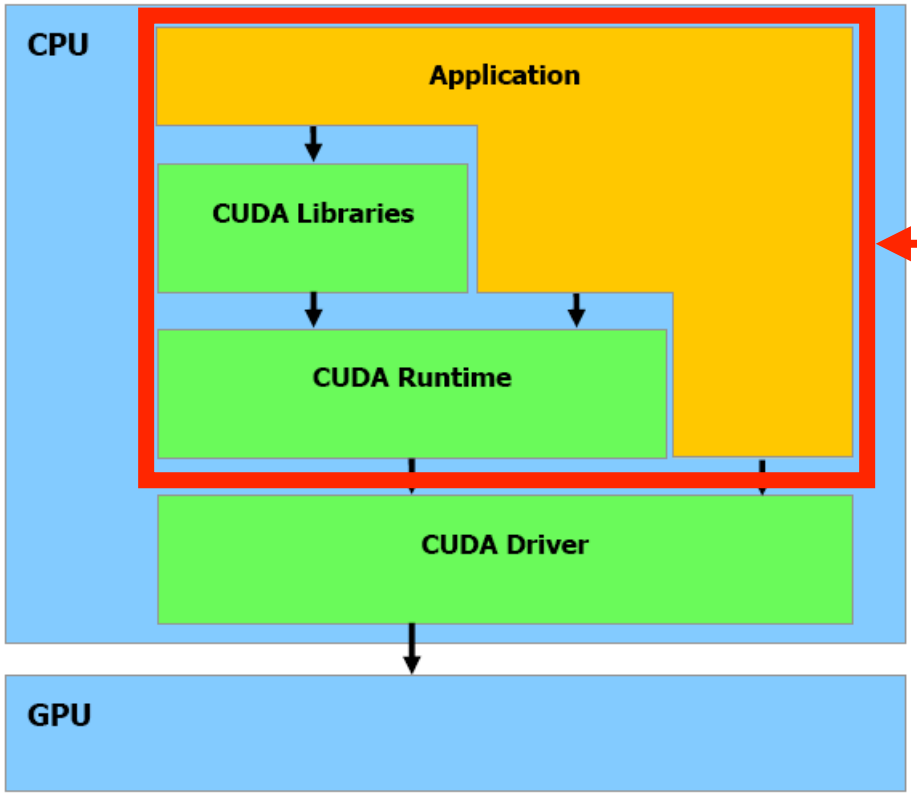
GPU Nomenclature

- 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.

CUDA Software Stack

CUDA = Common Unified Device Architecture

OpenCL is an alternative language for programming GPUs



Today we will focus mostly on the CUDA Runtime level

Figure Credit: NVIDIA CUDA Compute Unified Device Architecture Programming Guide 1.1

CUDA and OpenCL are based on C. More recently, the APARAPI project was created to support GPU programming from Java

RICE

Outline of a CUDA Code

```
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;

}
```

RICE

Process Flow of a CUDA Kernel Call (Compute Unified Device Architecture)

- Data parallel programming architecture from NVIDIA
  - —Execute programmer-defined kernels on extremely parallel GPUs
  - —CUDA program flow:
    1. Push data on device
    2. Launch kernel
    3. Execute kernel and memory accesses in parallel
    4. Pull data off device
- Device threads are launched in batches
  - —Blocks of Threads, Grid of Blocks
- Explicit device memory management
  - —cudaMalloc, cudaMemcpy, cudaFree, etc.
- NOTE: OpenCL is a newer standard for GPU programming that is more portable than CUDA

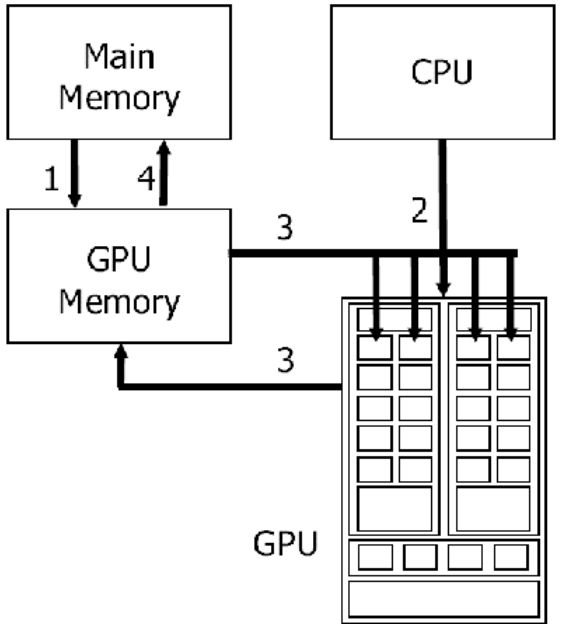


Figure source: Y. Yan et. al "JCUDA: a Programmer Friendly Interface for Accelerating Java Programs with CUDA." Euro-Par 2009.

Execution of a CUDA program

- Integrated host+device application
  - Serial or modestly parallel parts on CPU host
  - Highly parallel kernels on GPU device

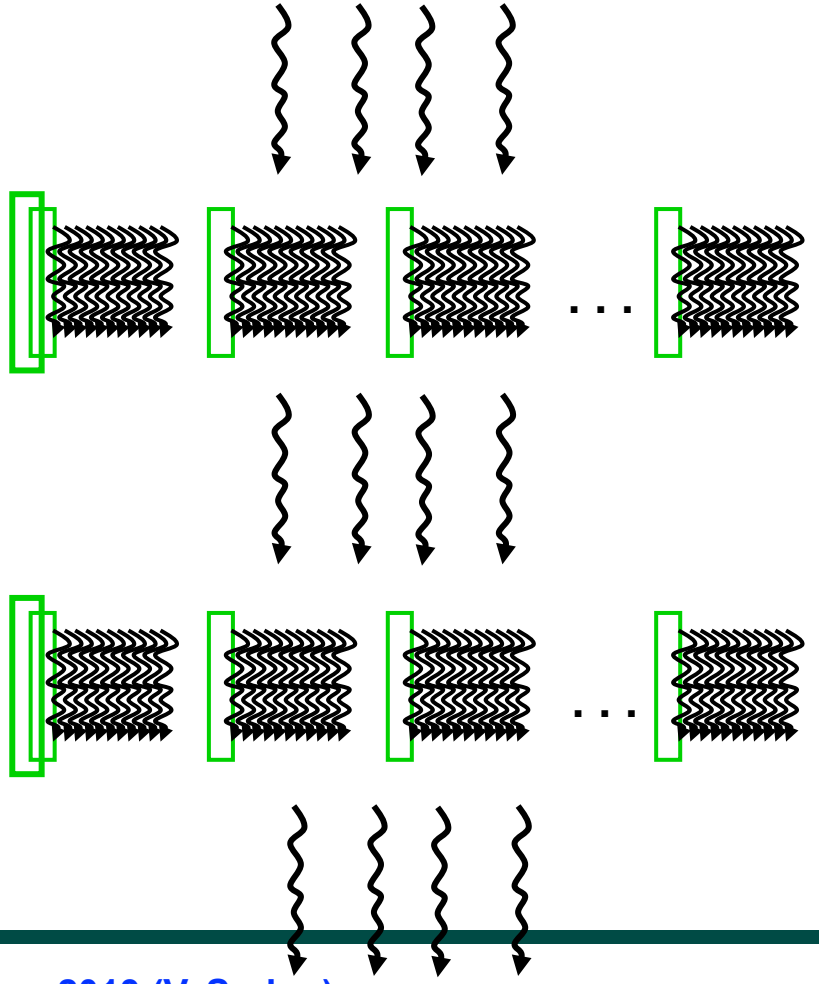
Host Code
(small number of threads)

Device Kernel
(large number of threads)

Host Code
(small number of threads)

Device Kernel
(large number of threads)

Host Code
(small number of threads)



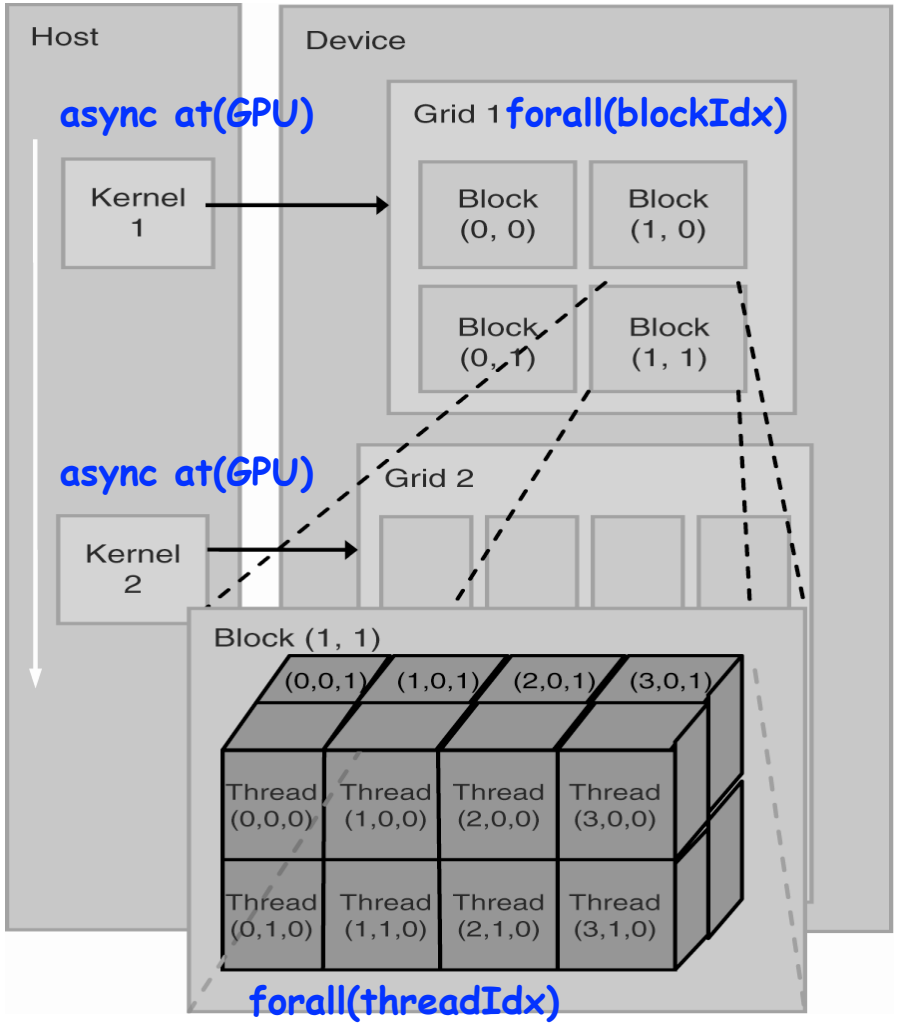
Matrix multiplication kernel code in CUDA --- SPMD model with 2D index (threadIdx)

```
// 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

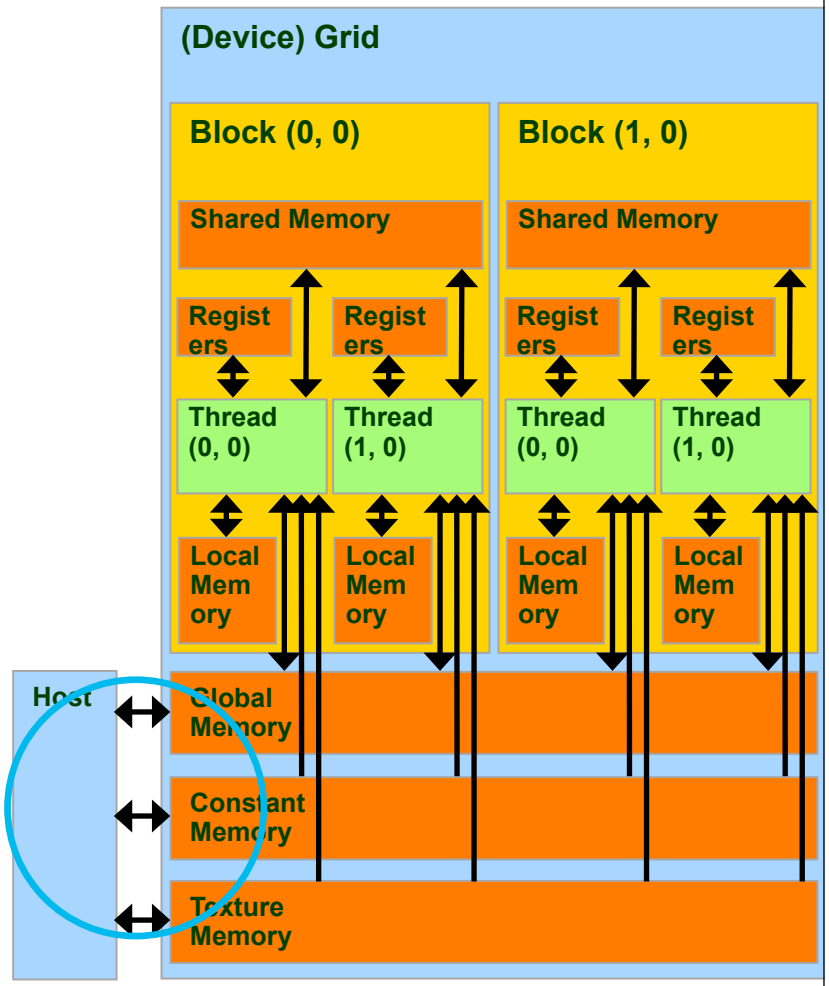
```
1.  void MatrixMultiplication(float* M, float* N, float* P, int Width)
    {
2.    int size = Width*Width*sizeof(float); // matrix size
3.    float* Md, Nd, Pd; // pointers to device arrays
4.    cudaMalloc((void**)&Md, size); // allocate Md on device
5.    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice); // copy M to Md
6.    cudaMalloc((void**)&Nd, size); // allocate Nd on device
7.    cudaMemcpy(Nd, M, size, cudaMemcpyHostToDevice); // copy N to Nd
8.    cudaMalloc((void**)&Pd, size); // allocate Pd on device
9.    dim3 dimBlock(Width,Width); dim3 dimGrid(1,1);
10.   // launch kernel (equivalent to "async at(GPU), forall, forall"
11.   MatrixMulKernel<<<dimGrid,dimBlock>>>(Md, Nd, Pd, Width);
12.   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost); // copy Pd to P
13.   // Free device matrices
14.   cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
15. }
```


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

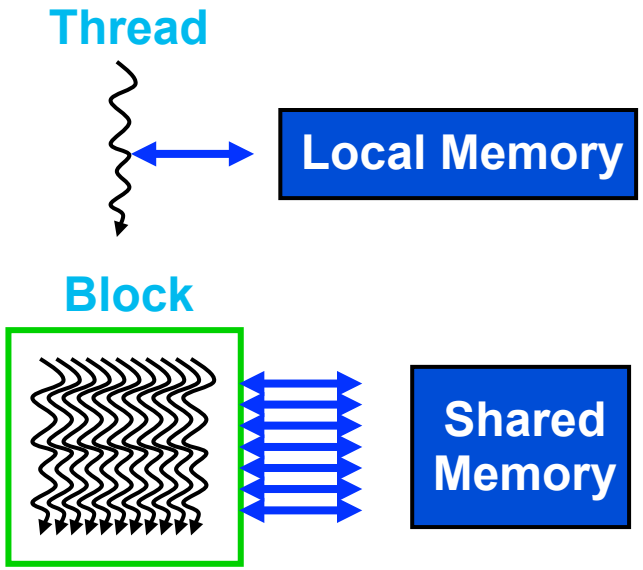


CUDA Host-Device Data Transfer

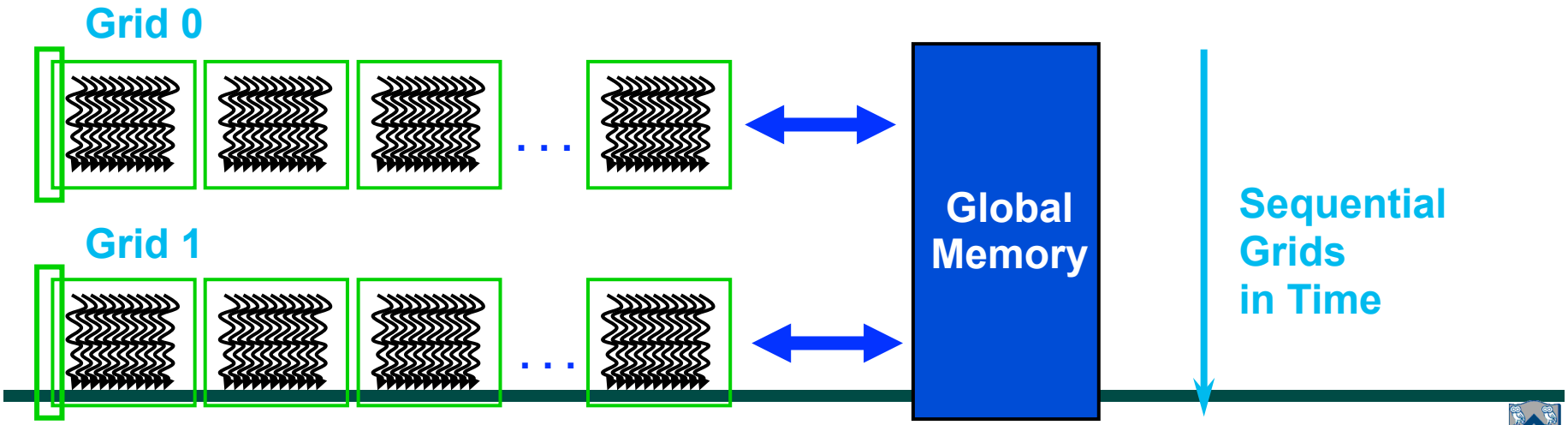
- cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)
- copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind is one of
  - —cudaMemcpyHostToHost
  - —cudaMemcpyHostToDevice
  - —cudaMemcpyDeviceToHost
  - —cudaMemcpyDeviceToDevice
- 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.



CUDA Storage Classes



- 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



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 #34: 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 13?

```
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. }
```