COMP 322: Fundamentals of Parallel Programming

Lecture 36: General-Purpose GPU (GPGPU) Computing

Guest Lecturer: Max Grossman
Founder and Principal @ 7pod Technologies
Research Scientist @ Rice U
Author, Professional CUDA C Programming

http://comp322.rice.edu/

Worksheet #35: Solution
Finding maximal index of goal in matrix

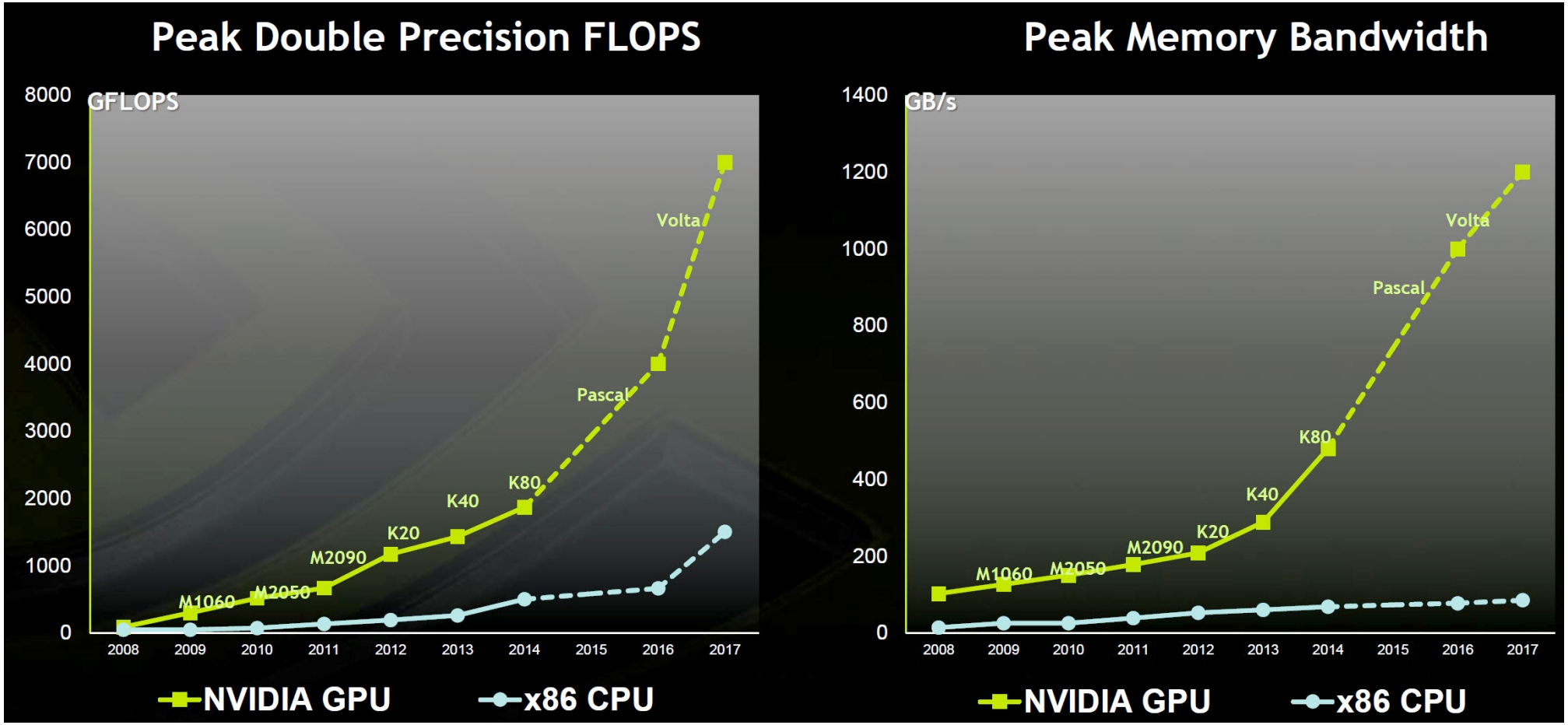
Below is a code fragment intended to find the maxi
occurs multiple times in the input matrix. What log

```
1.   class AsyncFinishEurekaSearchMaxIndexO
2.     HjEureka eurekaFactory() {
3.       comparator = (cur, newVal) -> { //
           (cur.x==newVal.x) ? (newVal.y –
4.       return new MaximaEureka([-1, -1],
5.     }
6.     int[] doWork(matrix, goal) {
7.       val eu = eurekaFactory()
8.       finish (eu, () -> { // eureka registration
9.         forasync (0, matrix.length - 1, (r) ->
10.          procRow(matrix(r), r, goal));
11.      });
12.      return eu.get()
13.    }
14.    void procRow(array, r, goal) {
15.      for (int c = array.length() - 1; c >= 0; c--)
16.        check([r, c]) // terminate if comparator returns negative
17.        if goal.match(array(c)) offer([r, c]) // updates cur in eureka
18.    } }
```

	0	…	10	…	15	…
…						
5			M			
…						
10	M				M	

The task terminates when check([r,c]) is called and the comparator has cur smaller than [r,c]. We need to ensure the iteration order in our code is such that the comparator returning negative means we cannot produce an offer([r',c']) where [r', c'] is greater than the value of cur.

Why GPUs?



- Performance gap between GPUs and multicore CPUs continues to widen

Applications of GPUs

- *Google* - Use GPUs internally to train deep learning models
- *DoE and DoD* – two of the next three supercomputers deployed by USA Department of Energy will be GPU based
- *Mayo Clinic* - GPUs to improve tumor identification
- *Audi* - GPUs for self-driving cars
- *SpaceX* - GPUs for combustion modeling of the methane-based Raptor rocket (system to be used for Mars missions)
- …

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)

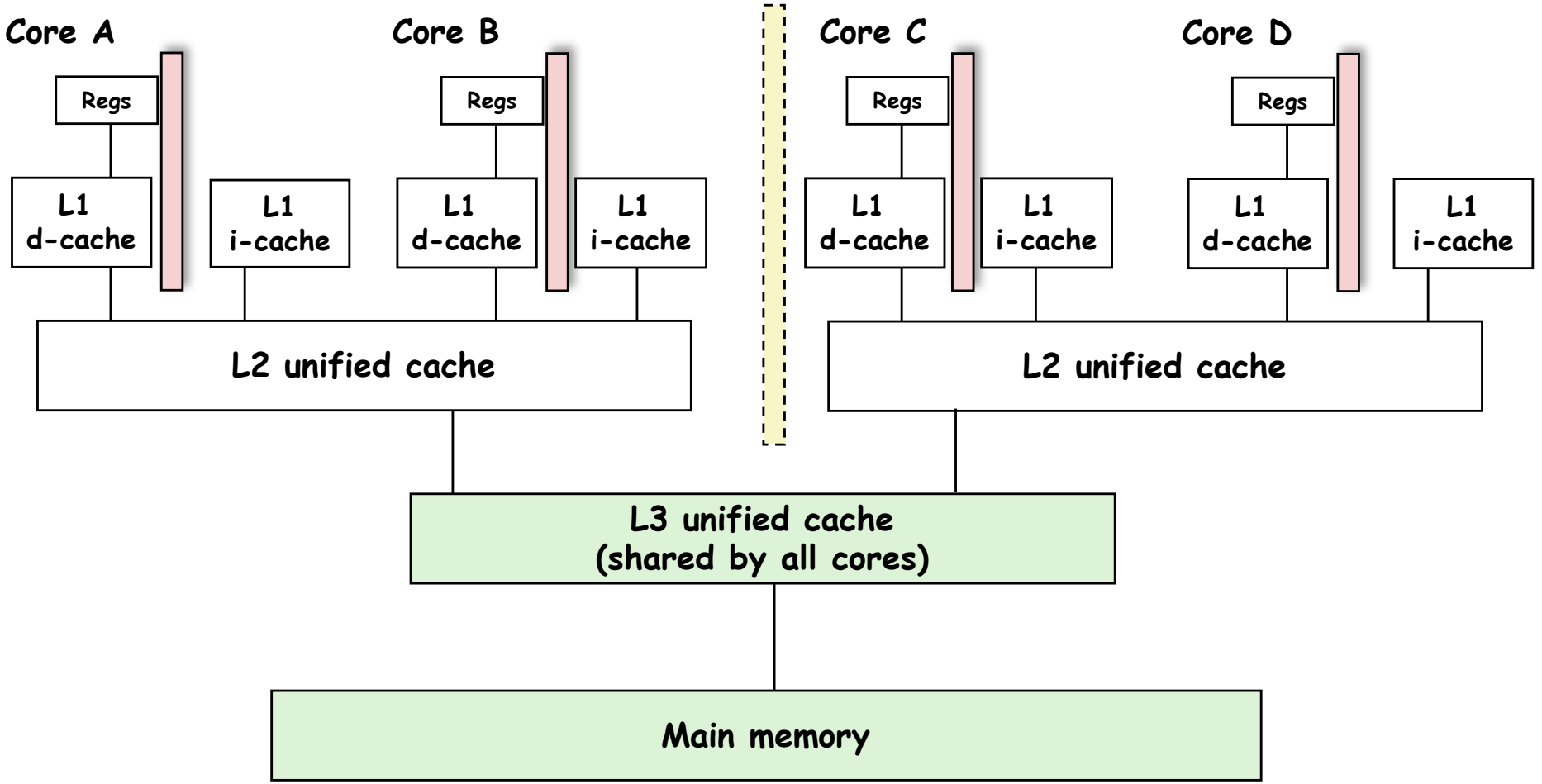
Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. e.g. the Space Shuttle flight control computer.

Multiple Instruction, Multiple Data streams (MIMD)

Multiple autonomous processors simultaneously executing different instructions on different data. e.g. a PC cluster memory space.

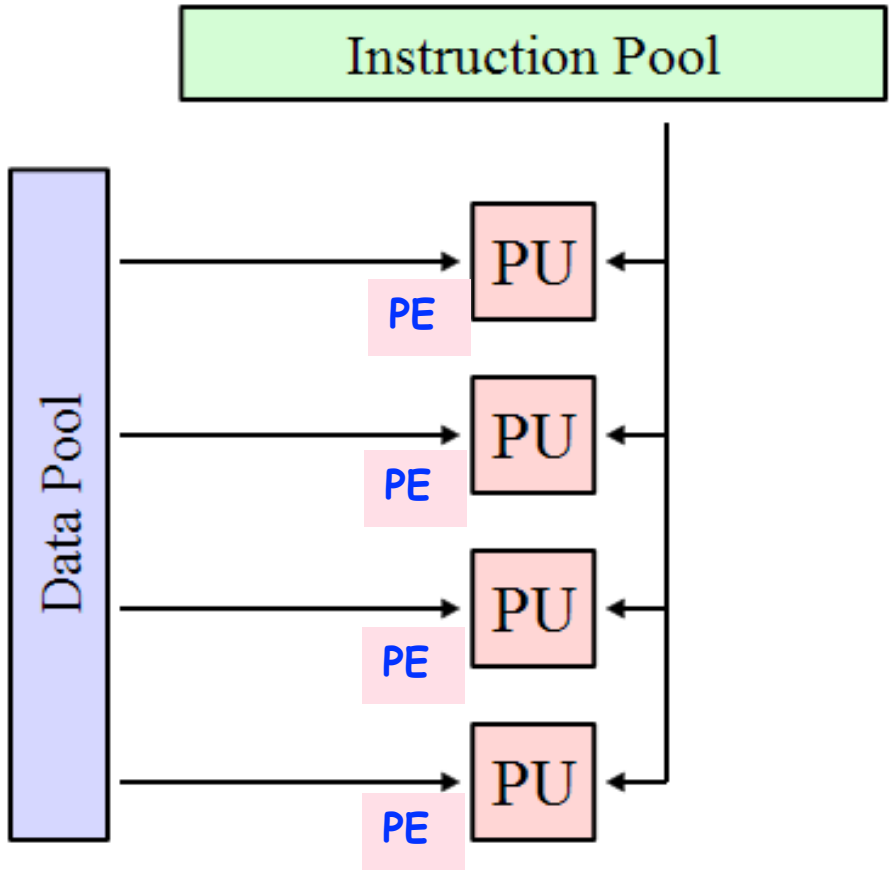
Multicore Processors are examples of MIMD systems

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



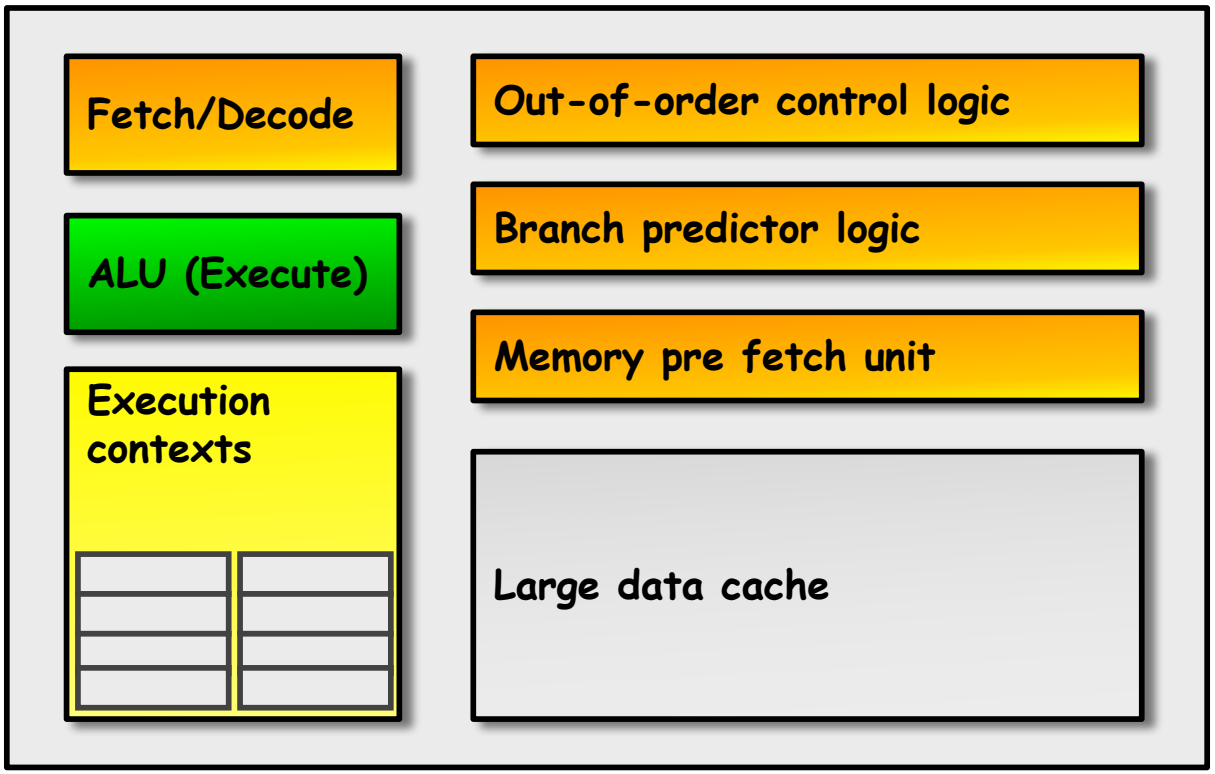
SIMD computers

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



"CPU-Style" Cores

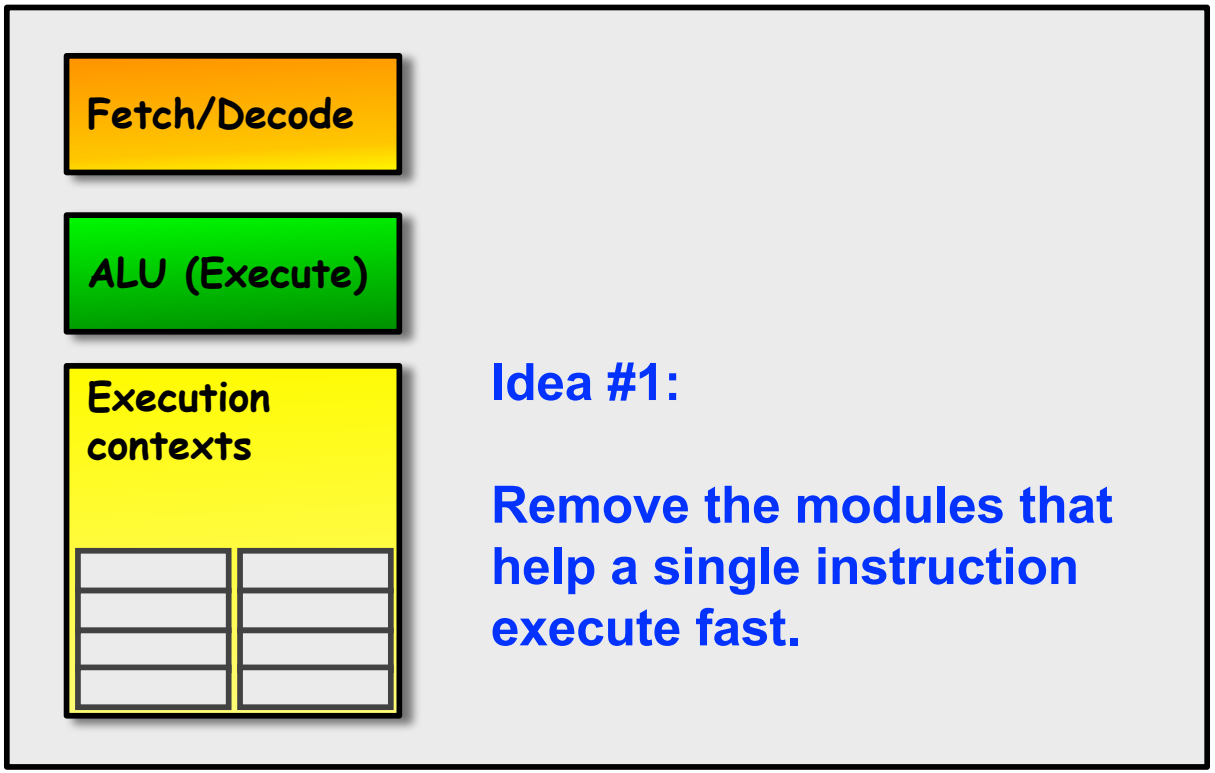
The "CPU-Style" core is designed to make individual threads speedy.



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

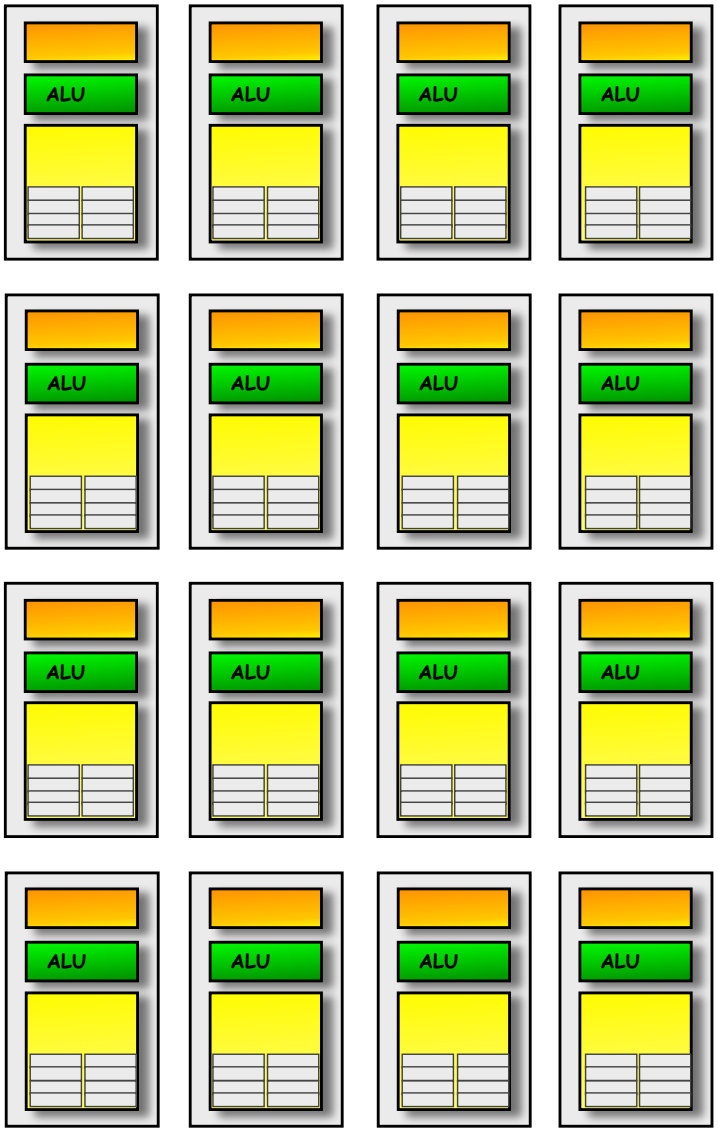
GPU Design Idea #1: more slow cores

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



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

GPU Design Idea #1: more slow cores



Idea #1:

Remove the modules that help a single instruction execute fast.

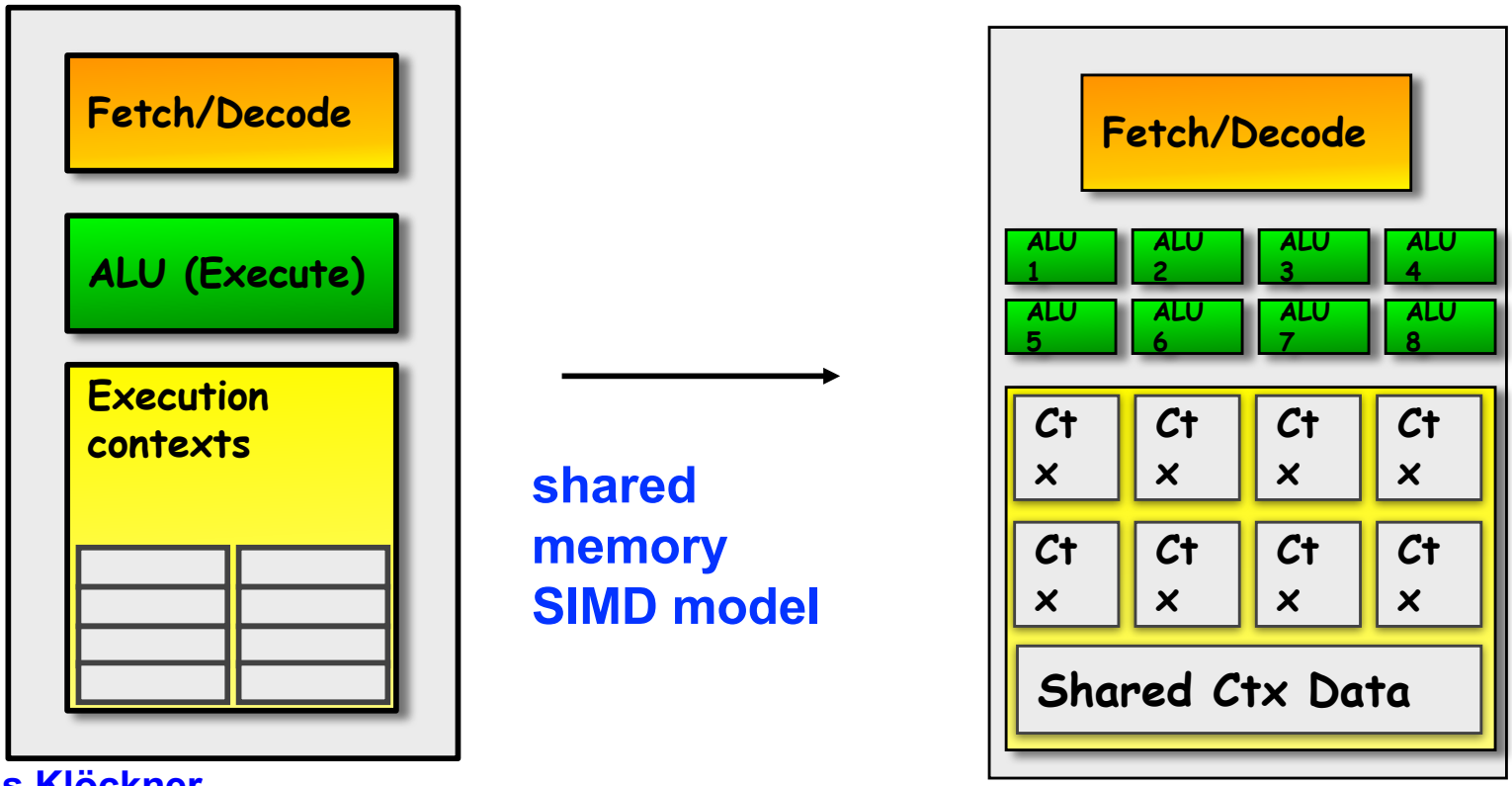
And then replicate at a massive scale.

See: Andreas Klöckner and Kayvon Fatahalian

GPU Design Idea #2: lock stepping

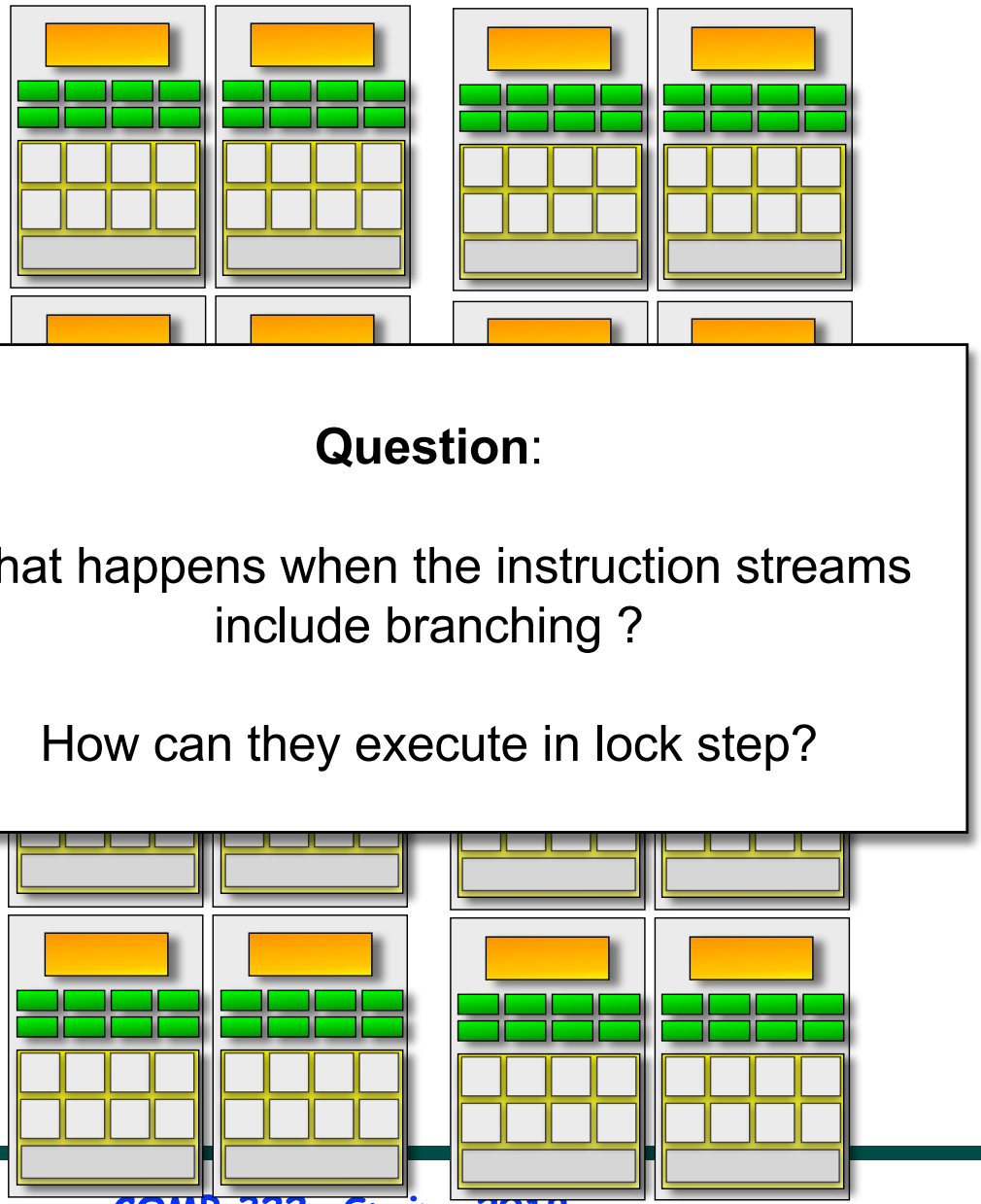
In the GPU rendering context, instruction streams are typically very similar.

Design for a "single instruction multiple data" SIMD model:
share the cost of the instruction stream across many ALUs (i.e. single program counter for multiple "cores")



See: Andreas Klöckner and Kayvon Fatahalian

GPU Design Idea #2: branching ?



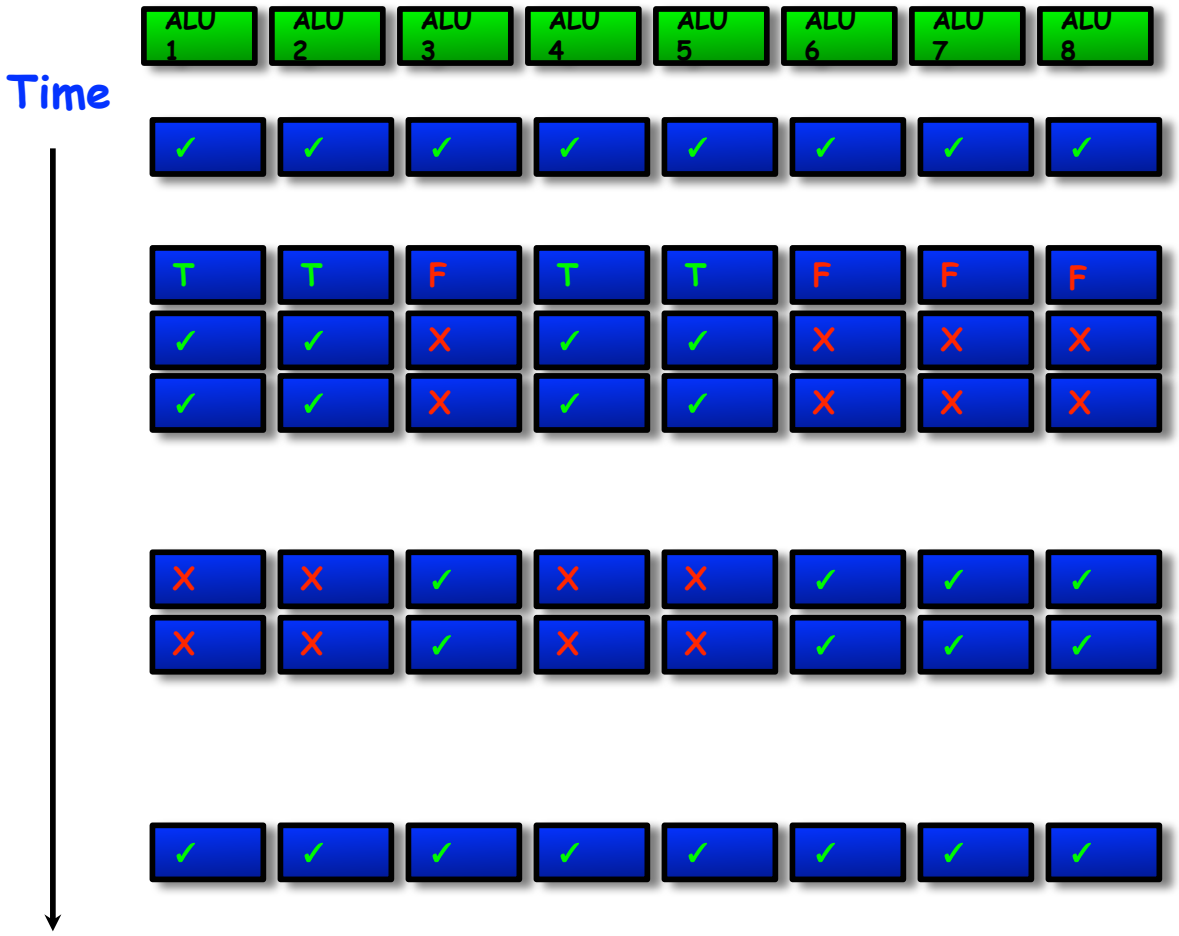
Question:

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

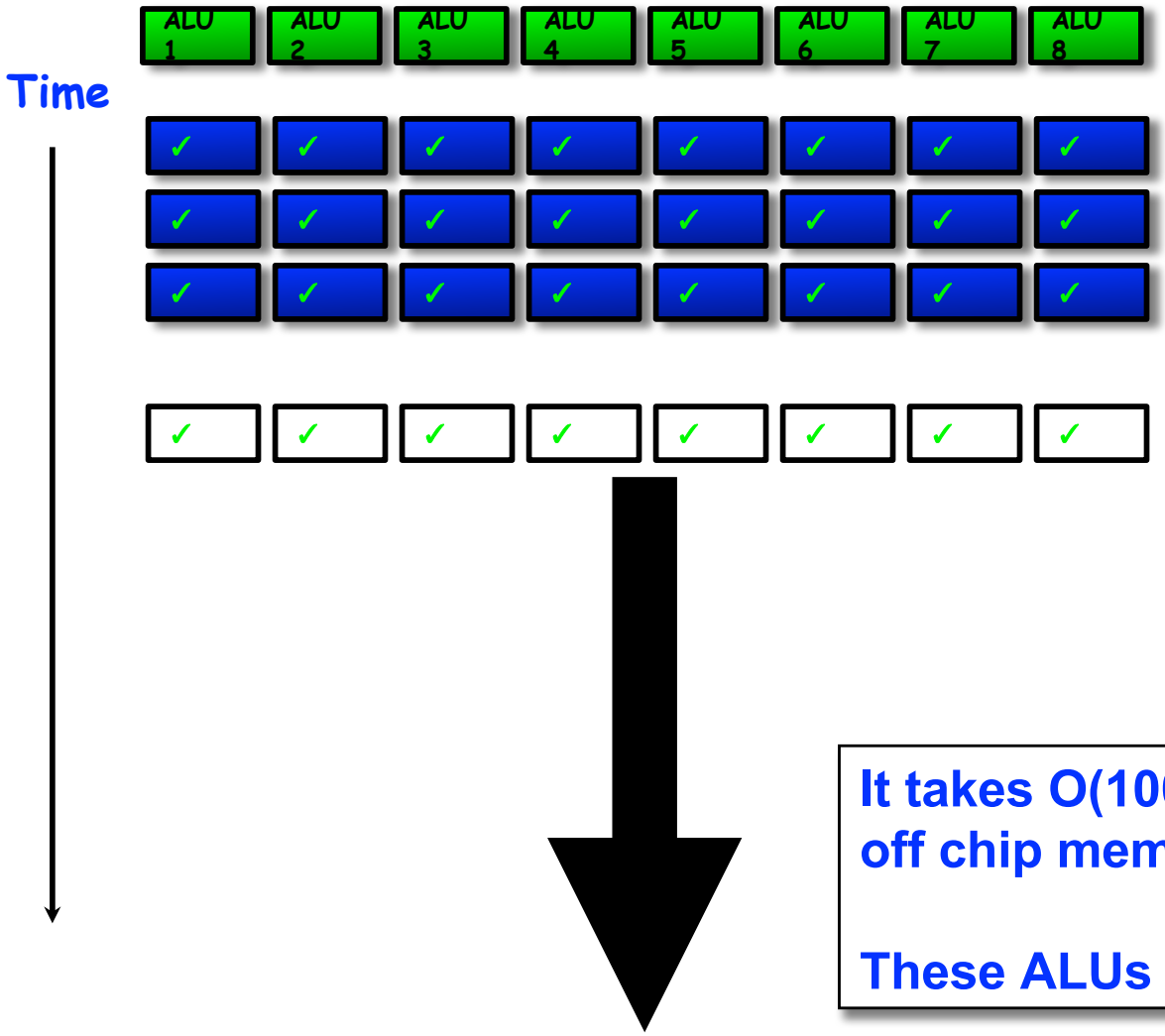



```
Non branching code;

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;

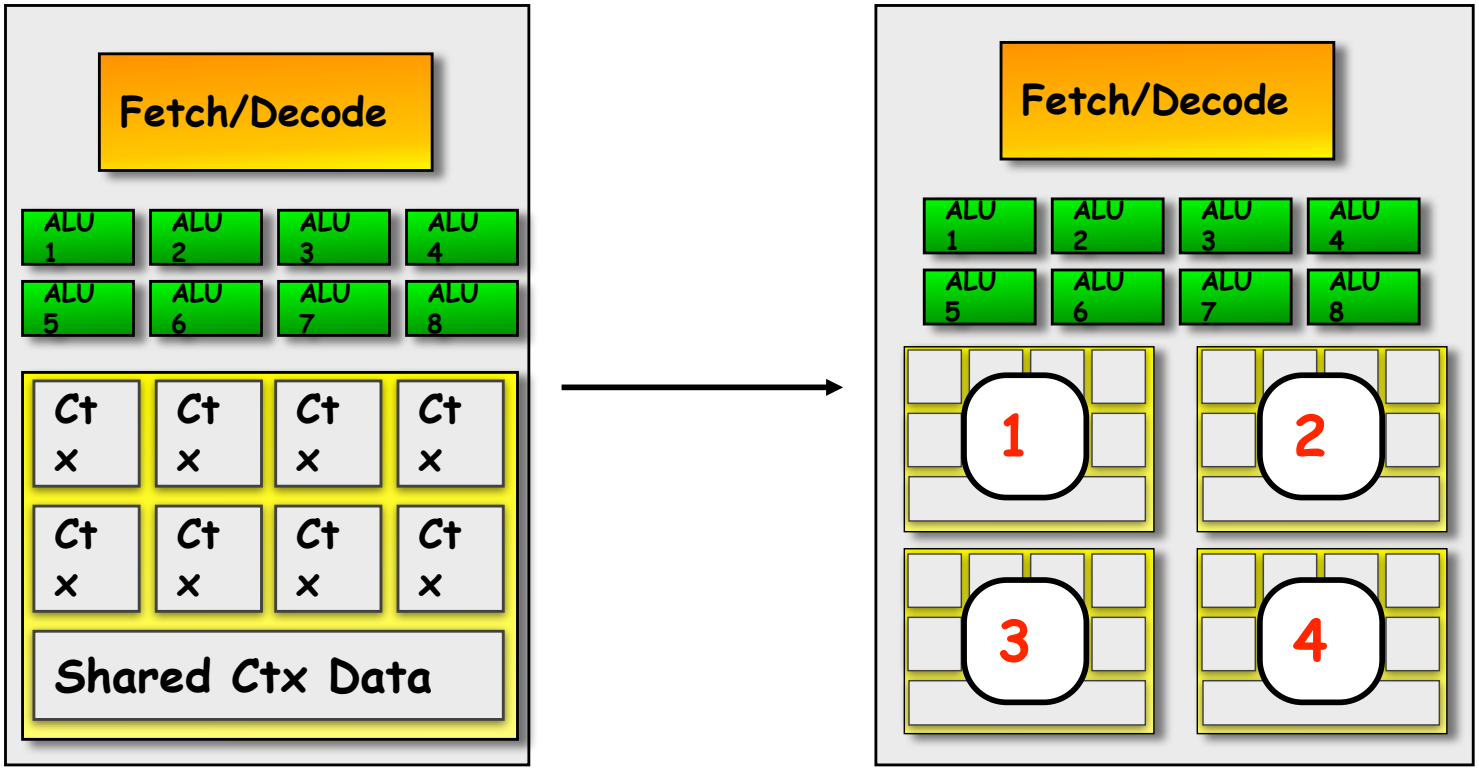
It takes O(1000) cycles to load data from off chip memory into the SM registers file

These ALUs are idled (stalled) after a load

See: Andreas Klöckner and Kayvon Fatahalian

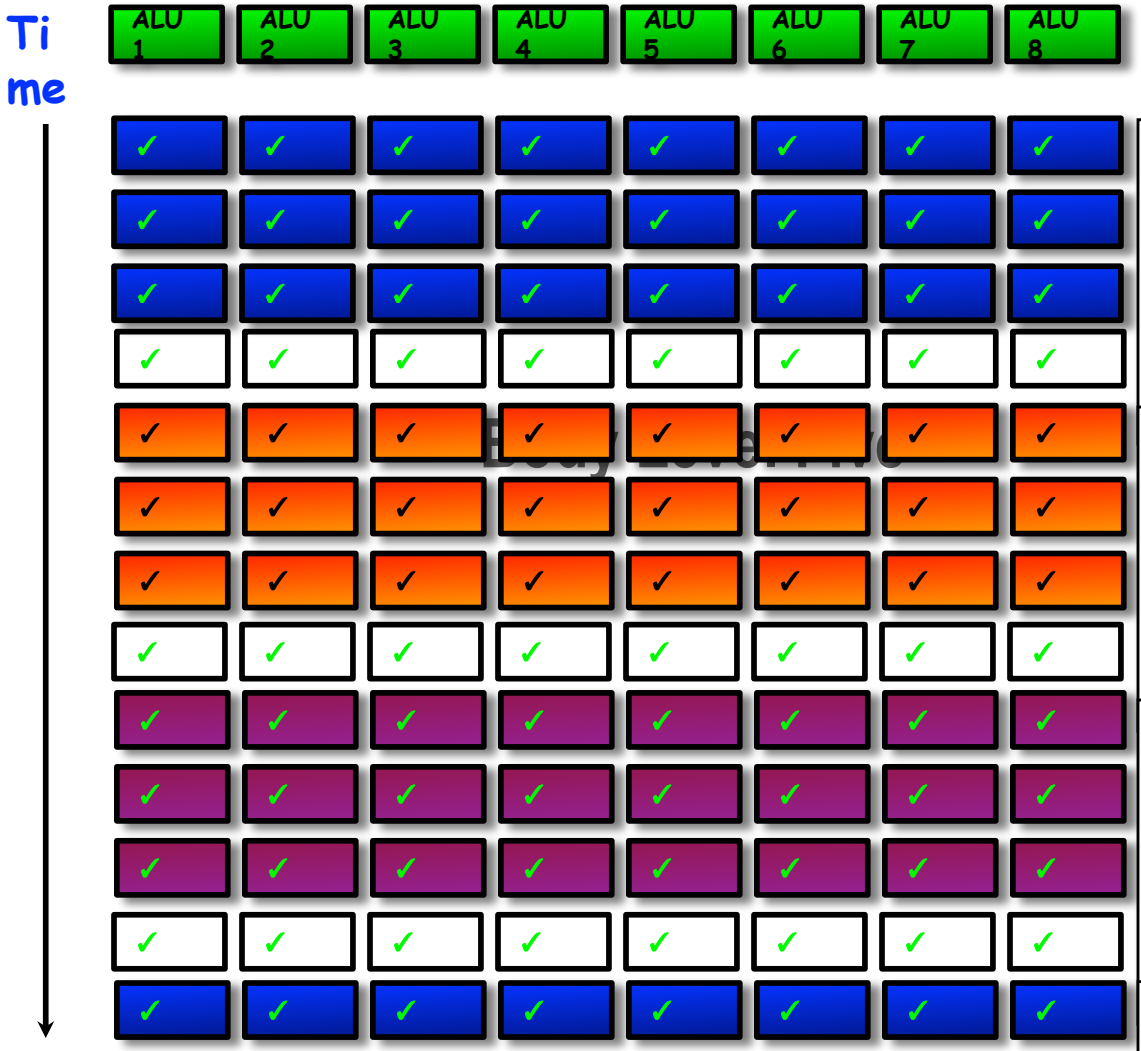
GPU Design Idea #3: latency hiding

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;

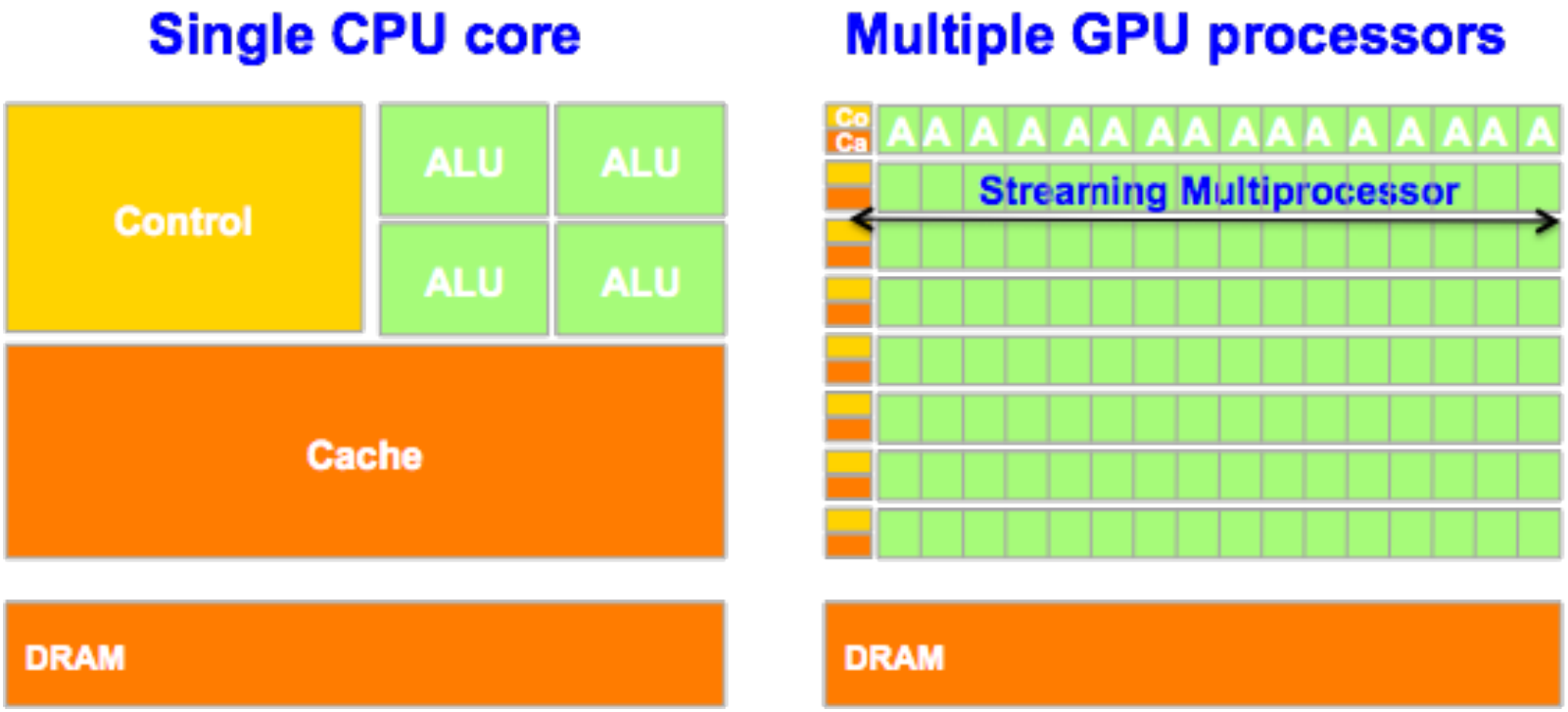
Ctx2: work on registers;
Ctx2: work on registers;
Ctx2: work on registers;
Ctx2: load request, switch context;

Ctx1: load done so continue

See: Andreas Klöckner and Kayvon Fatahalian

Summary: CPUs and GPUs have fundamentally different design philosophies

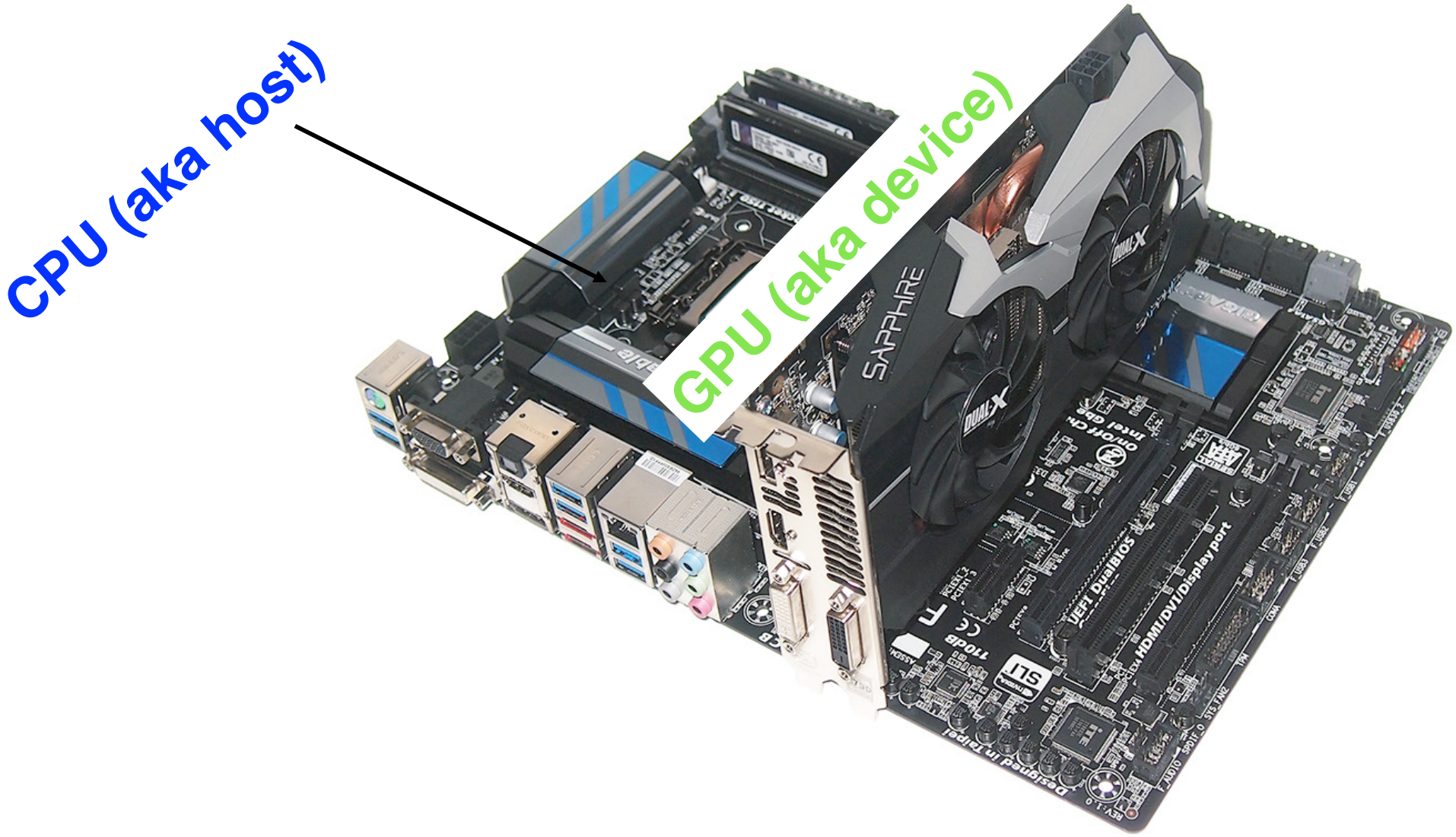
GPU = Graphics Processing Unit



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

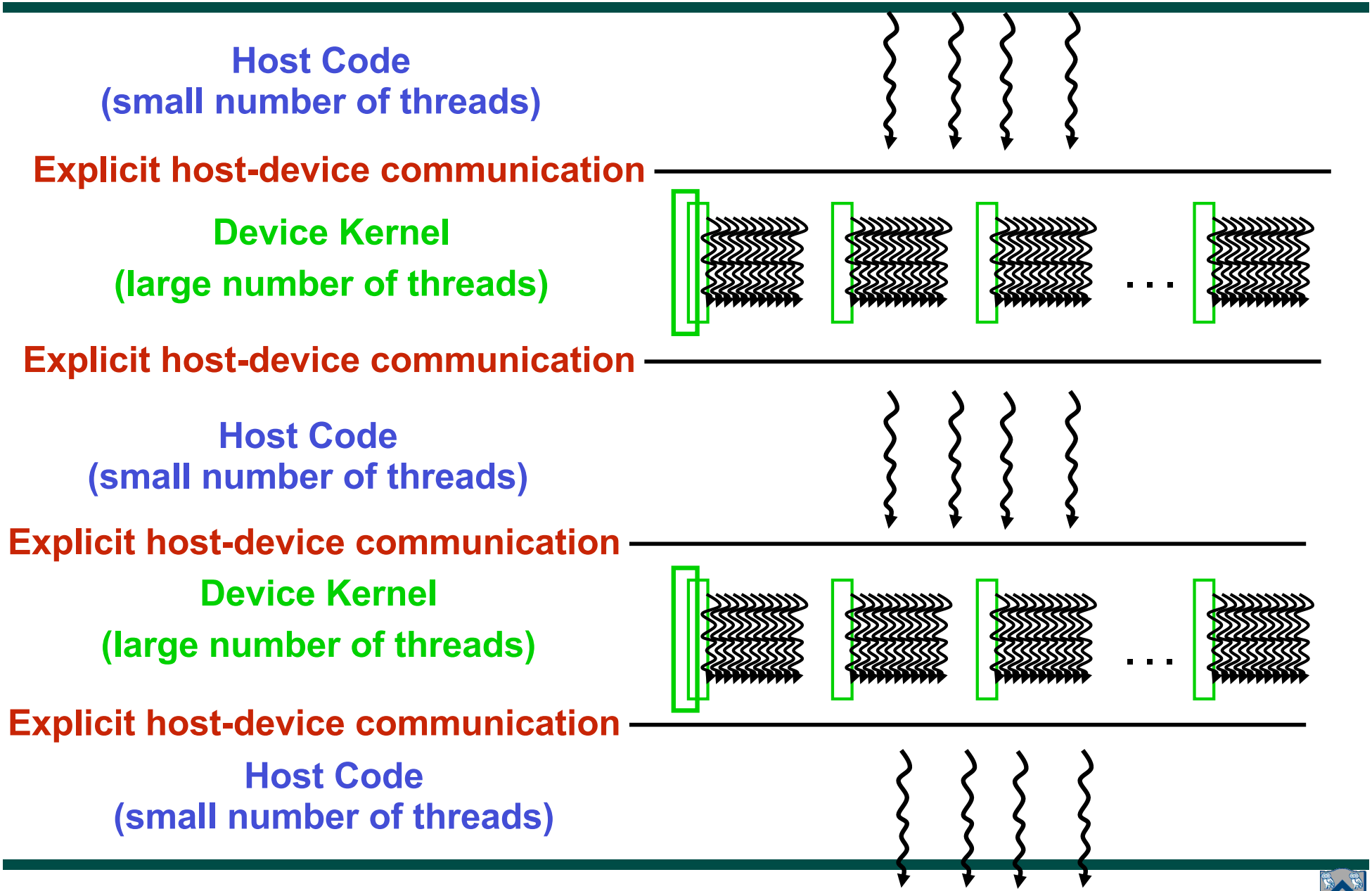
Host vs. Device



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



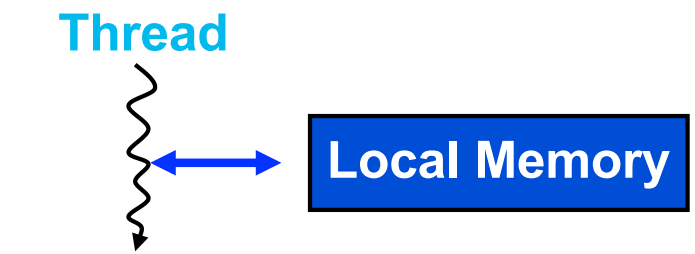
Outline of a CUDA main program

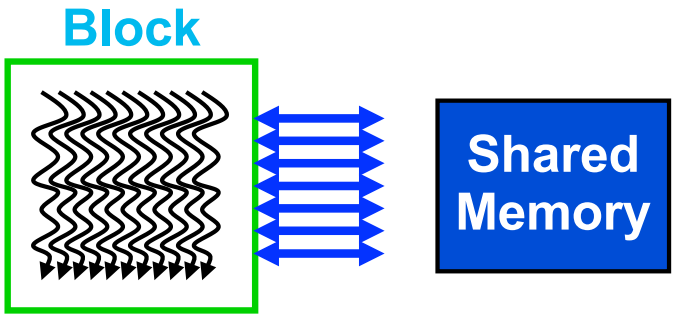
Runs on device

Runs on host

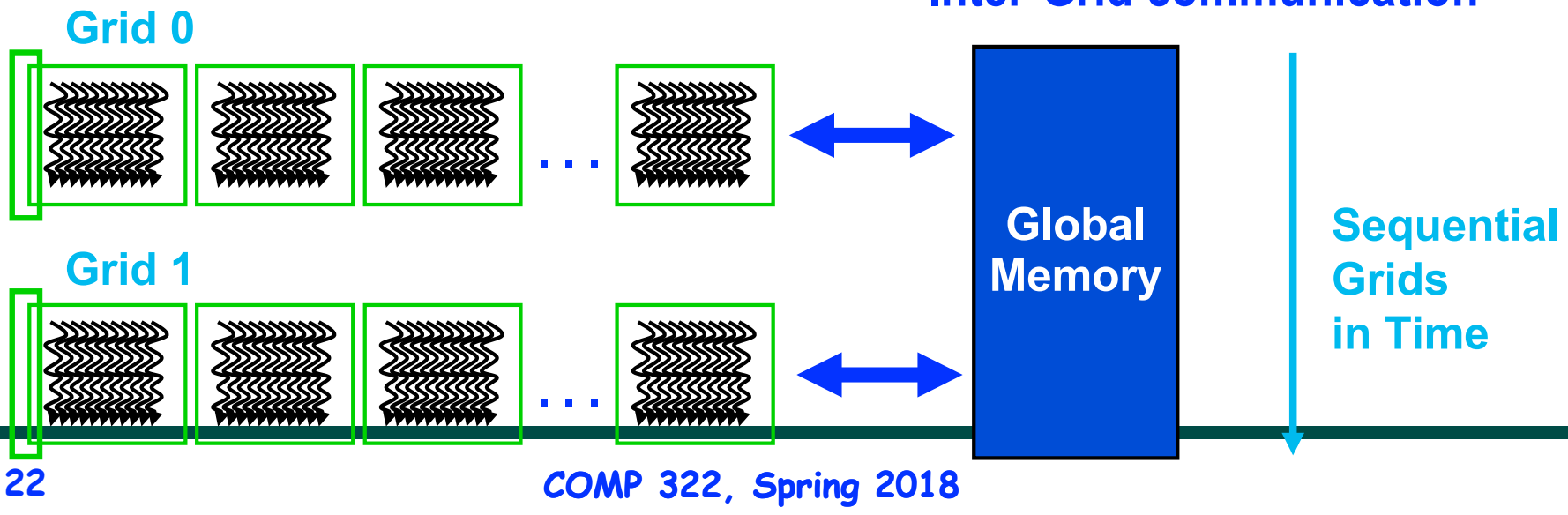
```
__global__ void kernel(arguments) {

  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





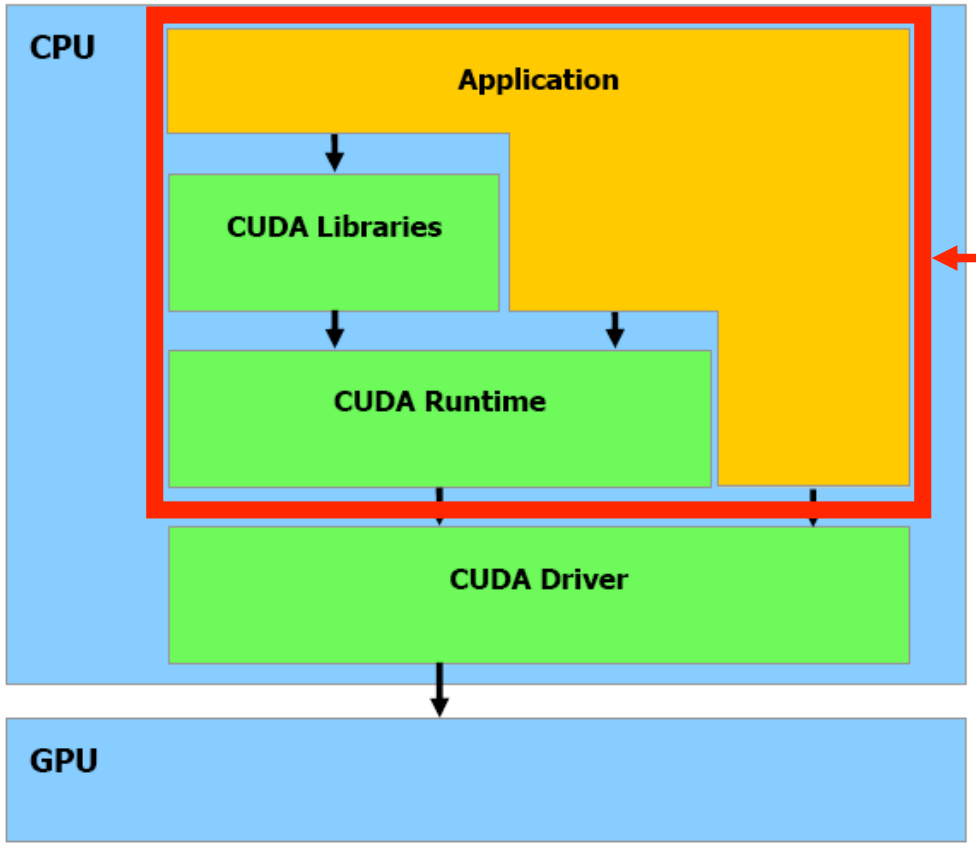
- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-block
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication



CUDA Software Stack

CUDA = Common Unified Device Architecture

Note: OpenCL is another framework 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

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

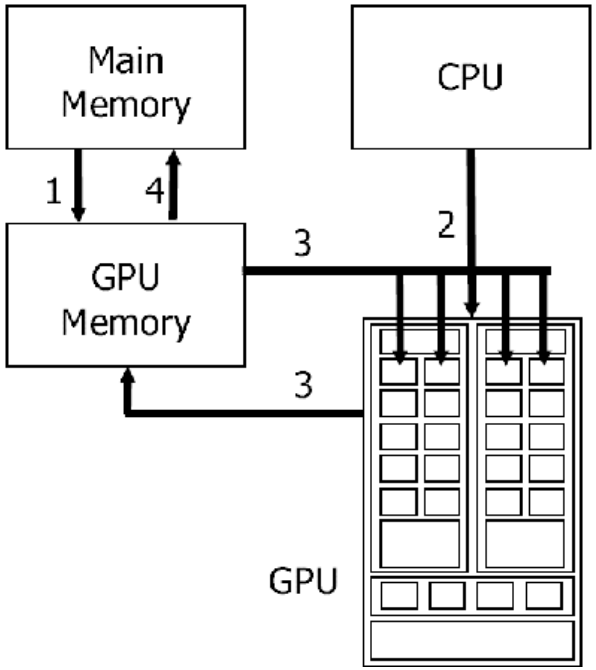


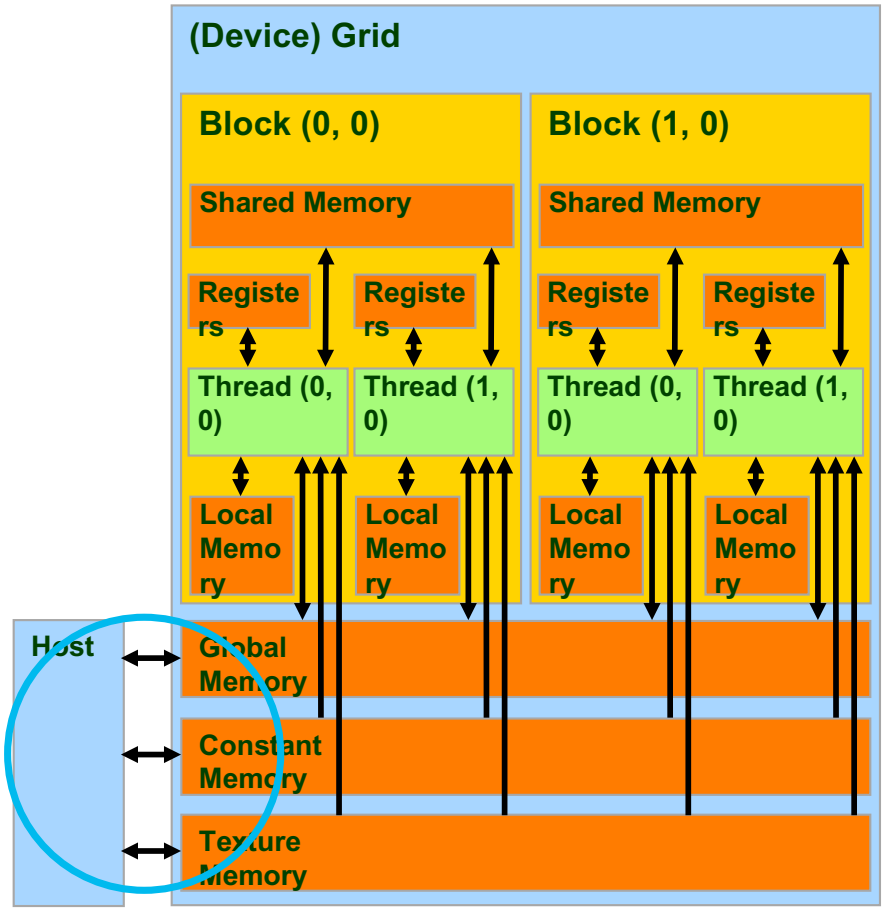
Figure source: Y. Yan et. al "JCUDA: a Programmer Friendly Interface for Accelerating Java Programs with CUDA." Euro-Par 2009.

CUDA Host-Device Data Transfer

cudaError_t **cudaMemcpy**(void* dst, const void* src, size_t count,
enum cudaMemcpyKind kind)

Copies 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



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 this 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 on 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.}
```


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)