

GPGPUs: Overview

Pedagogically precursor concepts

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Precursor Concepts: Pedagogical

- Architectural elements that, at least pedagogically, are precursors to understanding GPGPUs
 - SIMD and vector units.
 - We have already seen those
 - Large scale “Hyperthreading” for latency tolerance
 - Scratchpad memories
 - High Bandwidth memory

Tera MTA, Sun UltraSPARC T1 (Niagara)

- Tera computers, with Burton Smith as a co-founder (1987)
 - Precursor: HEP processor (Denelcor Inc.), 1982
- First machine was MTA
 - MTA-1, MTA-2, MTA-3
- Basic idea:
 - Support a huge number of hardware threads (128)
 - Each with its own registers
 - No Cache!
 - Switch among threads on every cycle, thus tolerating DRAM latency
 - These threads could be running different processes
 - Such processors are called “barrel processors” in literature
 - But they switched to the “next thread” always.. So your turn is 127 clocks away, **always**
- Was especially good for highly irregular accesses

Scratchpad memory

- Caches are complex and can cause unpredictable impact on performance
- Scratchpad memories are made from SRAM on chip
 - Separate part of the address space
 - As fast or faster than caches, because no tag-matching or associative search
 - Need explicit instructions to bring data into scratchpad from memory
 - Load/store instructions exist between registers and scratchpad
- Example: IBM/Toshiba/Sony cell processor used in PS/3
 - 1 PPE, and 8 SPE cores
 - Each SPE core has 256 KiB scratchpad
 - DMA is mechanism for moving data between scratchpad and external DRAM

High Bandwidth Memory

- As you increase compute capacity of processor chips, the bandwidth to DRAM comes under pressure
 - Many past improvements (e.g. Intel's Nehalem) were the result of improving bandwidth, including integration of memory related hardware into processor chip
 - Recall: Bandwidth is a matter of resources (here with some inherent limits on number of pins per chip)
 - GPUs have historically used higher bandwidth DRAM configurations
 - GDDR3 SDRAM, GDDR4, GGDR5, GGDR5X (10-ish Gbits/s per pin)
- Recent advance: High Bandwidth memory via 3D stacking
 - Multiple DRAM dies are stacked vertically and connected *through-silicon vias (tsv)*
 - *NVIDIA*,
 - MCDRAM (Intel), Hybrid Memory Cube (Micron)

GPGPUs

General Purpose Graphics Processing Units (GPUs)

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

GPUs and General Purposing of GPUs : I

- Graphics Processing Unit (GPU)
 - Drive the displays connected to a computer
 - Wikipedia: “[...] designed to manipulate .. memory to accelerate creation of images in a frame buffer.”
 - Original purpose: high speed rendering i.e. video games, etc.
- Due to need for speed in usages like video games, there was pressure to increase their speed and capabilities
 - Especially in the 1990’s with the rise of 3D graphics
 - Helped along by APIs of OpenGL, DirectX, Direct3D
 - Nvidia’s GeForce 3 had enough hardware support to do programmable shading

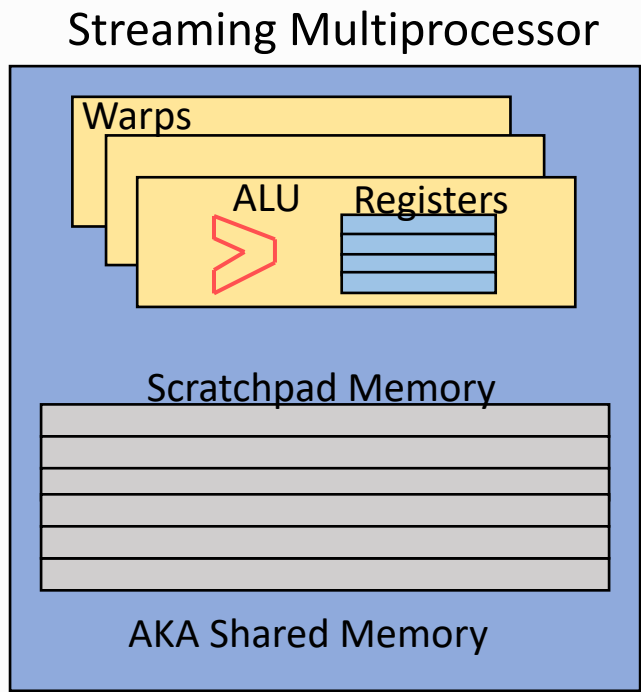
GPUs and General Purposing of GPUs : II

- GPGPUs were dedicated units, with a specialized function, but
 - They were getting faster and faster
 - They were getting more programmable in order to support graphics capabilities
- Many individuals and researchers in high performance computing started noticing that these devices could be used for computations
 - At least for a few, but commonly needed, patterns, such as data parallel loops!

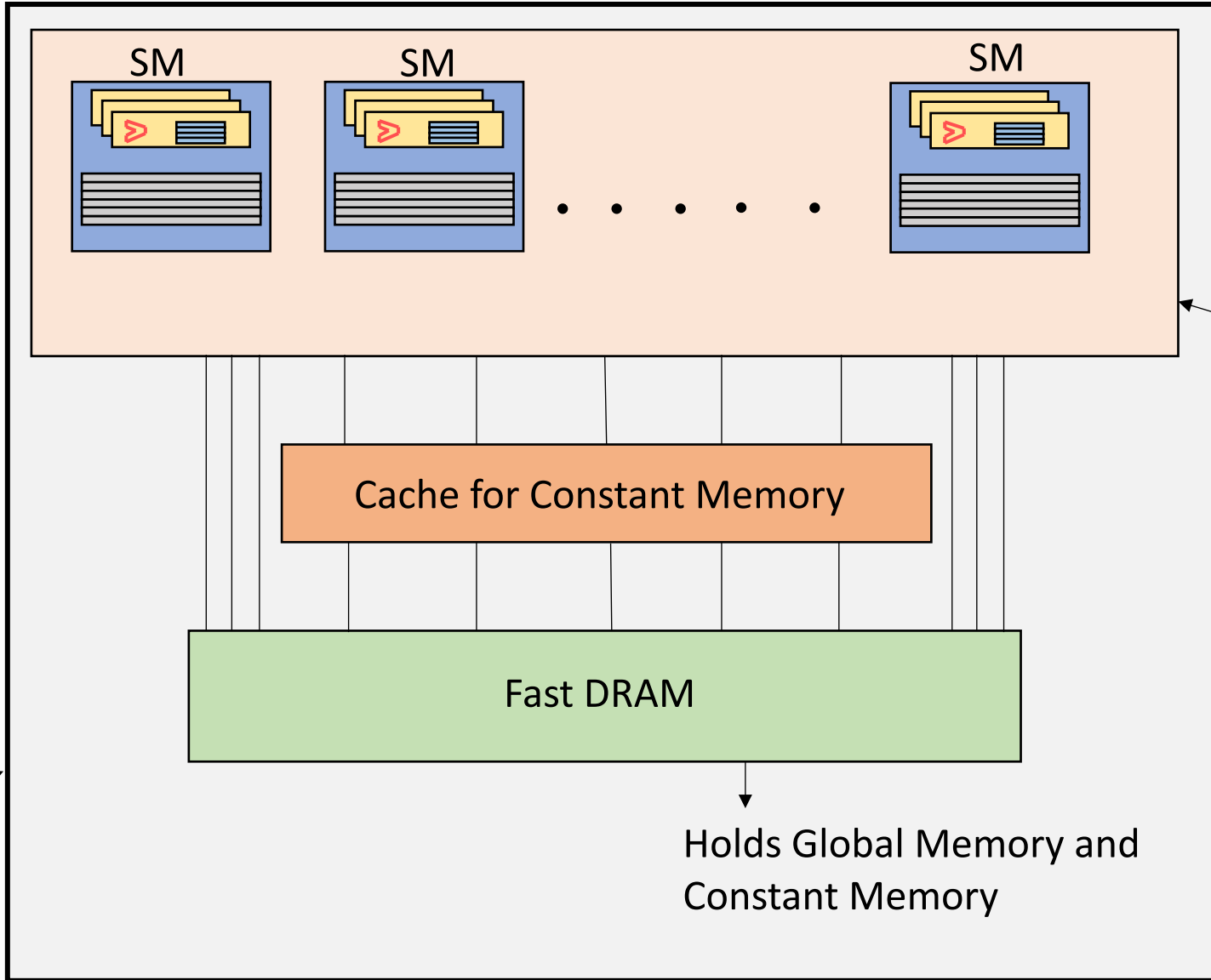
GPUs and General Purposing of GPUs : III

- Graphics Processing Unit (GPU)
- Original purpose: high speed rendering(?) i.e. video games, etc
- Optimized for being good at math
- Result: High memory BW and many “cores”
- Brook Streaming Language from Stanford
 - Ian Buck et al paper is worth a read
 - The idea of specialized kernels
 - Running on specialized devices
- NVIDIA and AMD (and Intel’s integrated graphics)
- Programming: CUDA, OpenCL, and OpenMP

In this paper, we present Brook for GPUs, a system for general-purpose computation on programmable graphics hardware. Brook extends C to include simple data-parallel constructs, enabling the use of the GPU as a streaming coprocessor.



The Device



Each SM is like a Vector Core

GPGPU Chip

Holds Global Memory and Constant Memory

Schematic GPGPUs

CPU vs GPGPU Comparison

CPU (Host)

- Latency optimized
- 10s of Cores
- 10s of Threads
- Memory
- Cache hierarchy
- Perf. via fast memory/large caches

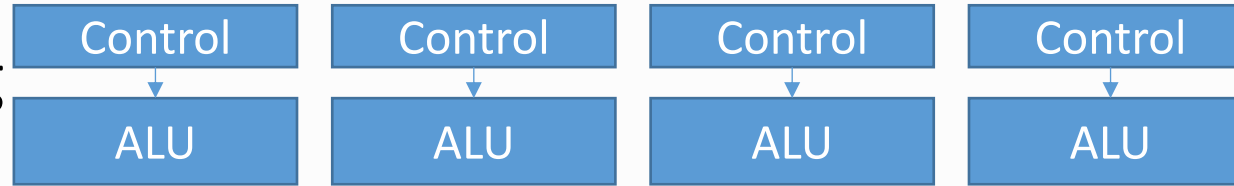
GPU (Device)

- Throughput optimized
- 1000s of “Cores” (CUDA cores)
- 10,000s of Threads
- Memory (on board)
- Simplified cache
 - No cpu-like consistency support
- Perf. via massive (regular) parallelism

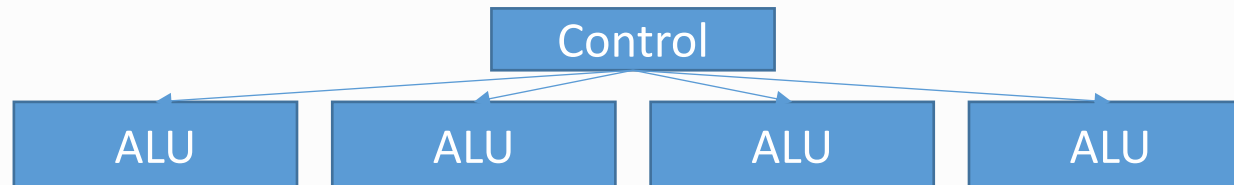
SIMT

- Single Instruction Multiple Threads

- Programming Model



- Execution Model



GPGPU: Programming with CUDA

GPGPU Software Architecture

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

CUDA

- We will present a very simple, over-simplified, overview
- Explicit resource-aware programming
- What you specify
 - Data transfers
 - Data parallel kernel/s, expressed in form of threads
 - Each thread does the action specified by the kernel
 - The total number of threads are grouped into teams called “blocks”
 - Kernel calls specify the number of blocks , and number of threads per block

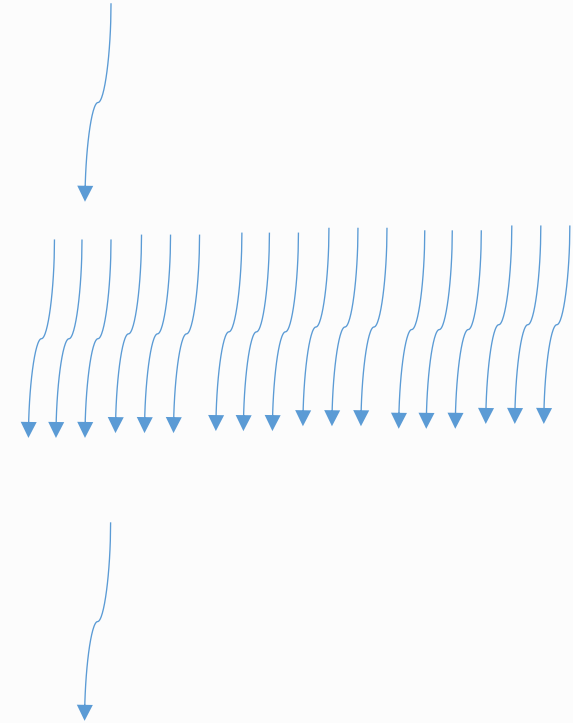
Programming Model Overview

- Host (serial)
- Launches device functions (parallel)
- Control can return asynchronously
- Memory?
 - Device memory
 - “Unified” memory
- Overlap
 - It is possible to overlap data transfer of one kernel with computation of another

- Serial

- Parallel

- Serial



Simple CUDA Program

```
#include <stdio.h>

void hello() {
    printf("Hello, world!\n");
}

int main() {
    hello();
}
```

```
$ gcc hello.c
$ ./a.out
Hello, world!
```


Simple CUDA Program

```
#include <stdio.h>

__global__
void hello() {
    printf("Hello, world!\n");
}

int main() {
    hello<<<1,1>>>();
}
```

```
$ gcc hello.c
```

```
$ ./a.out
```

```
Hello, world!
```

```
$ nvcc hello.cu
```

```
$ ./a.out
```

```
Hello, world!
```

Blocks

- Basic parallel unit
- Threads in a block can assume access to a common shared memory region (scratchpad).
- Analogous to processes
- Blocks grouped into grid
- Asynchronous

```
int main() {  
    hello<<<128,1>>>();  
}
```

```
$ ./a.out  
Hello, world!  
Hello, world!  
...  
Hello, world!
```

Threads

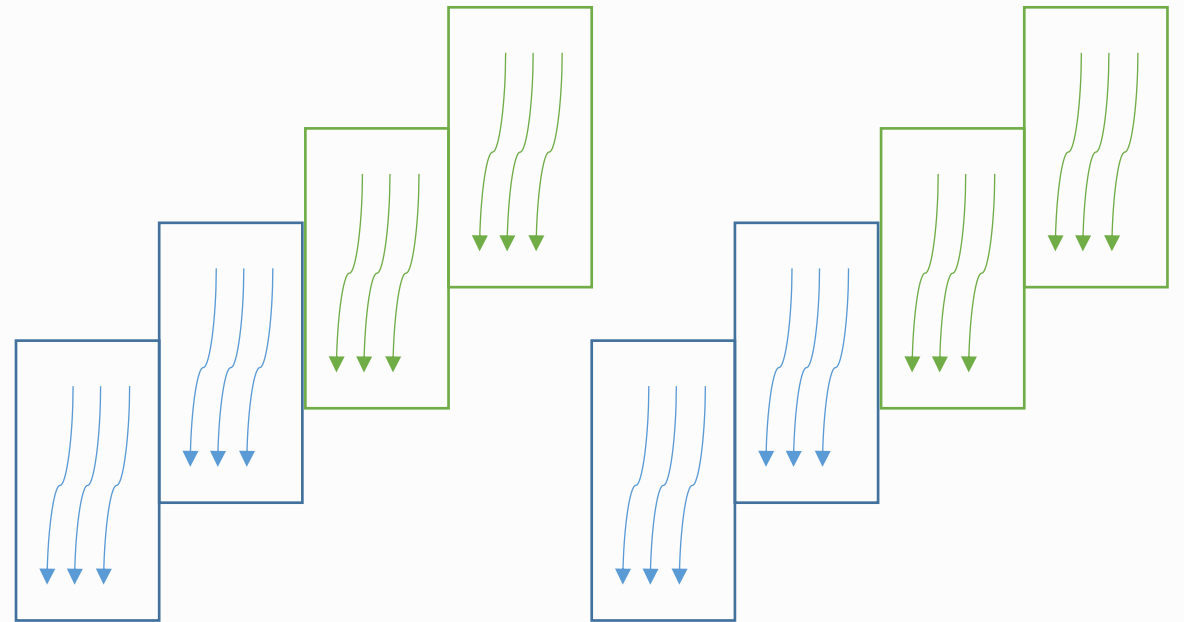
- Sub-division of a block (shared memory)
- Analogous to OpenMP threads
- Grouped into warps (shared execution)
- Level of synchronization and communication

```
int main() {  
    hello<<<1,128>>>();  
}
```

```
$ a./out  
Hello, world!  
Hello, world!  
...  
Hello, world!
```

Warps

- Groupings of threads
- All execute same instruction (SIMT)
- One miss, all miss
- Thread divergence, No-Ops
- Analogous to vector instructions
- Scheduling unit



Combining Blocks, Warps, and Threads

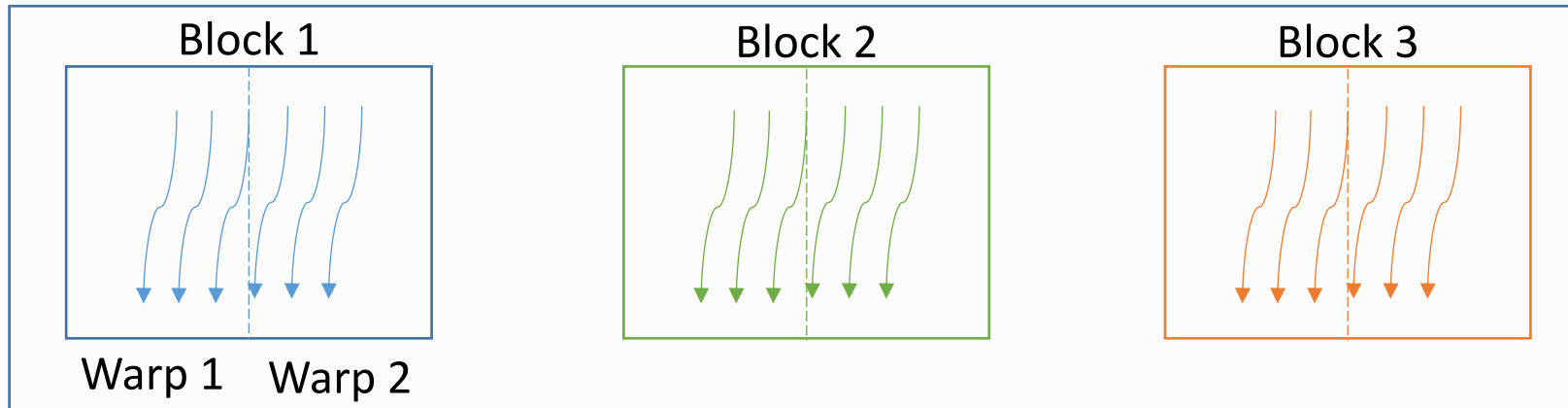
Number of Blocks

Number of Threads per Block

KernelFunc<<<3,6>>>(...);

Block Dimension = 6

For this picture, assume a warp has 3 threads.. (in reality, its almost always 32.. It's a device dependent parameter)



Thread Index	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Global Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

If you specify blocksize that's not a multiple of warpsize, the system will leave some cuda cores in a warp idle)

Illustrative Example

```
__global__
void vecAdd(int* A, int* B, int* C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
...
int main() {
    // Unified memory allocation
    vecAdd<<<VEC_SZ/512, 512>>>>(A, B, C);
}
```

blockIdx.x is my block's serial number

blockDim.x is the number of threads per block

threadIdx.x is my thread's id in my block

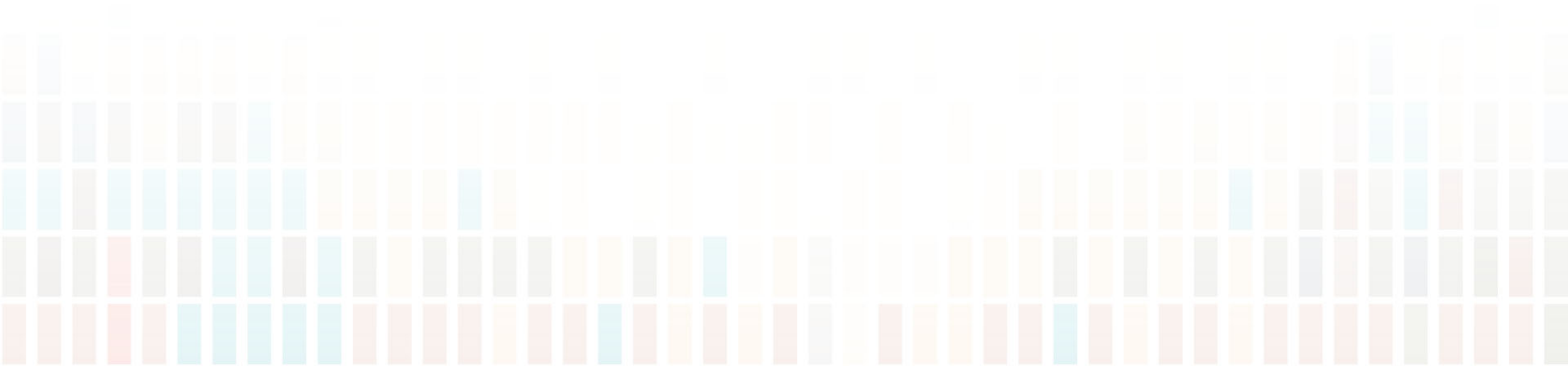
Number of Blocks

Number of Threads per Block

Using OpenMP for GPU programming

A Simpler way of using GPGPUs

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN



Background: OpenMP support for GPGPUs

- Traditional solution: use specialized languages (CUDA/OpenCL)
 - Need to rewrite lots of code
 - Target only subset of device types
 - e.g.: CUDA code can't run on AMD GPUs, OpenCL is slow on Nvidia
- OpenMP 4.0+ has support for offloading computation to accelerators
 - Lots of overlap with (earlier) OpenACC standard
 - OpenMP already widely used for multicore parallelization
 - Mixing OpenACC and OpenMP is difficult
 - Can target different types of devices (Nvidia GPUs, AMD GPUs, Xeon Phi, ...)
 - OpenMP standard only describes **interface**, not implementation
 - Each compiler needs to implement support for different devices

General overview – ZAXPY in OpenMP

Multicore

```
double x[N], y[N], z[N], a;  
//calculate z[i]=a*x[i]+y[i]  
  
#pragma omp parallel for  
for (int i=0; i<N; i++)  
    z[i] = a*x[i] + y[i];
```

General overview – ZAXPY in OpenMP

Offloading

```
double x[N], y[N], z[N], a;  
  
#pragma omp target  
{  
  for (int i=0; i<N; i++)  
    z[i] = a*x[i] + y[i];  
}
```

Compiler: Generate code
for GPU

Runtime: Run code on
device if possible, copy
data from/to GPU

- Code is unmodified except for the pragma
- Data (x, y, z, a) is **implicitly** copied
- Calculation done on device **if available**
 - Runs on CPU otherwise

OpenMP offloading – distributing work

- By default, code in target region runs sequentially on accelerator

```
double x[N], y[N], z[N], a; //calculate z=a*x+y

#pragma omp target
{
#pragma omp teams distribute parallel for
for (int i=0; i<N; i++)
    z[i] = a*x[i] + y[i];
}
```

- **teams** directive – create thread teams to perform work
 - e.g., one team per SM of GPU
- **distribute parallel for**:
 - distribute for loop iterations among threads
- Can be combined, as in the example above

OpenMP offloading – data movement


- Data movement can be made explicit
 - Data can be reused by multiple regions

```
double x[N], y[N], z[N], a;
```

```
#pragma omp target data map(tofrom: x[:N], y[:N], z[:N], a)
{
    #pragma omp target
    // [...]

    #pragma omp target
    // [...]
}
```

Copy data to device on entering region, copy it back to host on leaving region



Programming GPGPUs:

- CUDA
 - High performance
 - Control
 - Can automate data movement
- OpenMP Target
 - Automates Data Movement
 - “Easy”
- cuLibraries : cuBLAS, cuFFT, etc
 - Very high performance/tuned
 - Drop in replacement
- Others: OpenCL, OpenACC, etc