Exebit 2018

CUDA Workshop

Somesh Singh

Department of Computer Science and Engineering

Indian Institute of Technology Madras

April 14, 2018

Outline

- GPU Hardware
- Deep Dive into CUDA
  - Thread Organization
  - Memory Hierarchy
  - Data Transfer
  - Writing Kernels
- Streams: Overlapping kernel execution and data transfers
- Synchronization:
  - barrier: __syncthreads()
  - atomic operations
- memory coalescing
- using shared memory
- thread divergence
- CUDA register pressure
- Occupancy

Expectations

At the end of the workshop

- You will be able to write parallel code in CUDA
- Know how to optimize code to extract performance in CUDA code.

Why Parallelism? Why Efficiency?

- Speedup
  - A major motivation of using parallel processing
  - for a given problem:

$$speedup(using\ P\ processors) = \frac{execution\ time(using\ 1\ processor)}{execution\ time(using\ P\ processors)}$$

- Efficiency
  - Use the provided machine hardware efficiently.

Why Parallelism? Why Efficiency?

- Speedup
  - A major motivation of using parallel processing
  - for a given problem:

$$speedup(using\ P\ processors) = \frac{execution\ time(using\ 1\ processor)}{execution\ time(using\ P\ processors)}$$

- Efficiency
  - Use the provided machine hardware efficiently.
  - Is $2\times$ speedup on a computer with 10 processors a good result?

Why Parallelism? Why Efficiency?

- Speedup
  - A major motivation of using parallel processing
  - for a given problem:

$$speedup(using\ P\ processors) = \frac{execution\ time(using\ 1\ processor)}{execution\ time(using\ P\ processors)}$$

- Efficiency
  - Use the provided machine hardware efficiently.
  - Is $2\times$ speedup on a computer with 10 processors a good result? Arguably NO!

CPU vs GPU

CPU

- 1 (few) thread(s)
- Minimize latency
- Big on-chip cache
- Sophisticated control logic

GPU

- Many threads
- Maximize throughput
- # threads limited by (hardware) resources
- Use multi-threading to hide latency

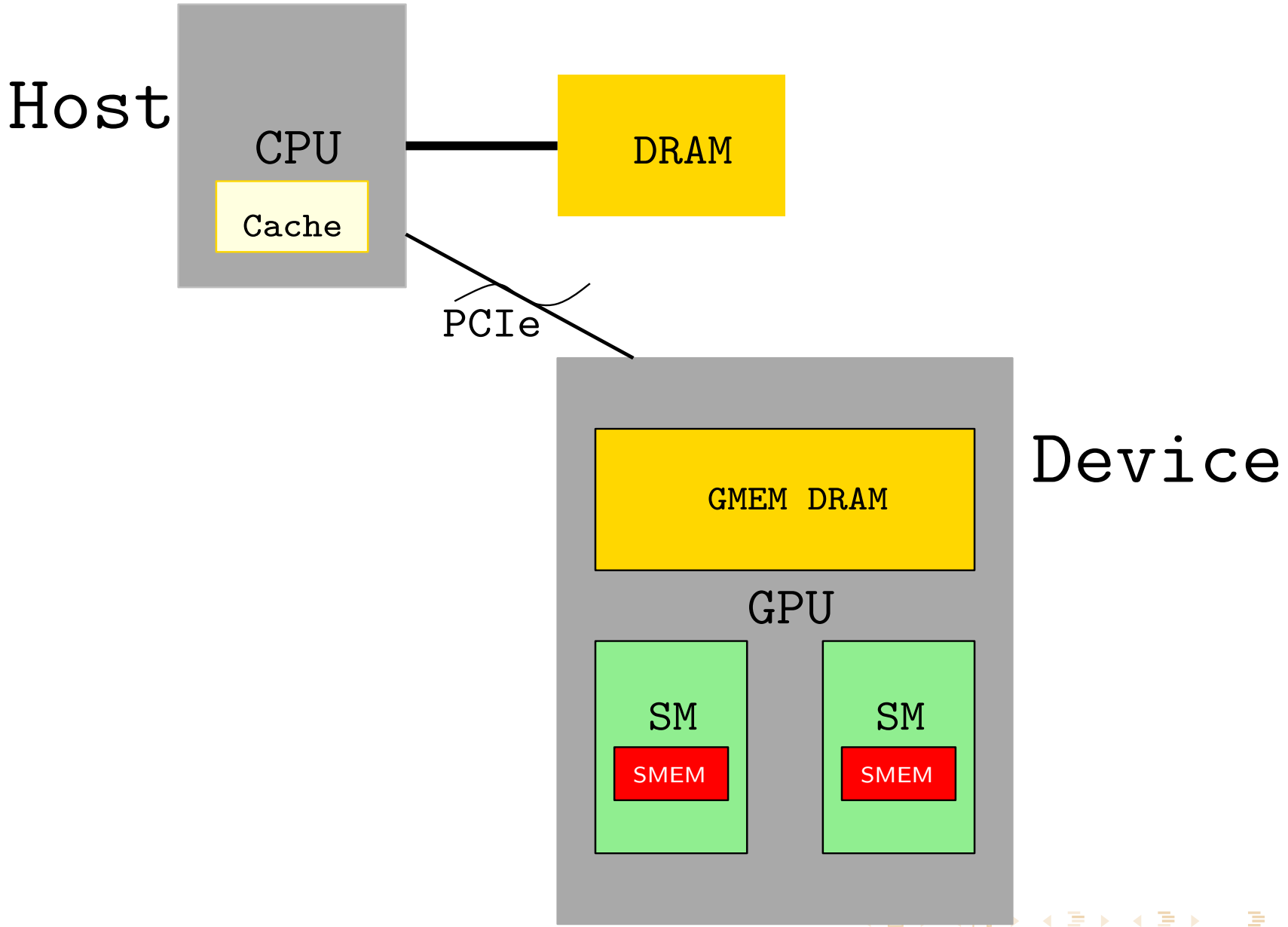
Image Source: Google Images

GPU – A throughput-oriented processor

Key idea of `throughput-oriented systems`:

- Potentially increase time to complete work by any one thread, in order to increase overall system throughput when running multiple threads.

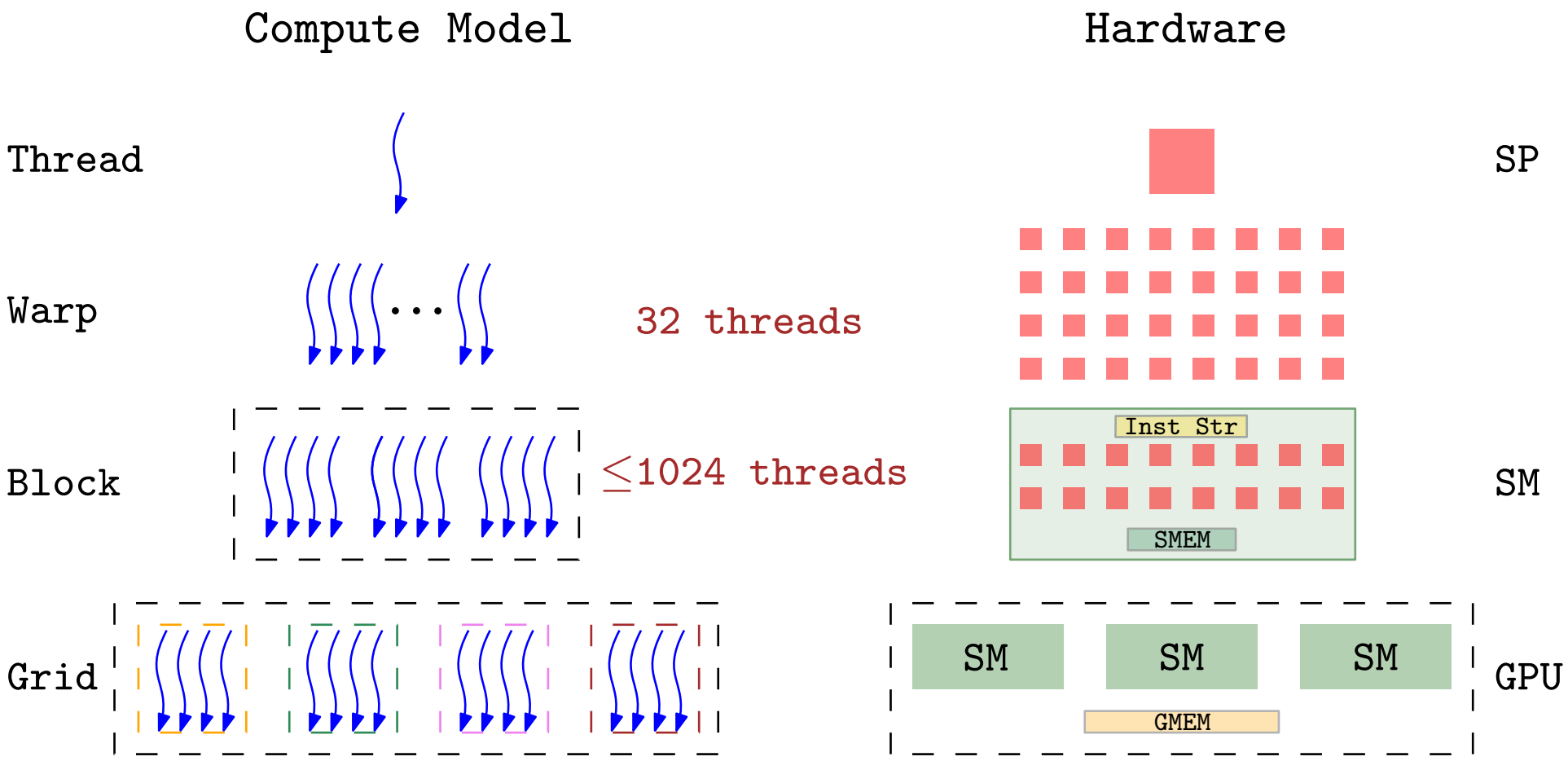
GPU — A Co-processor



CUDA programming language

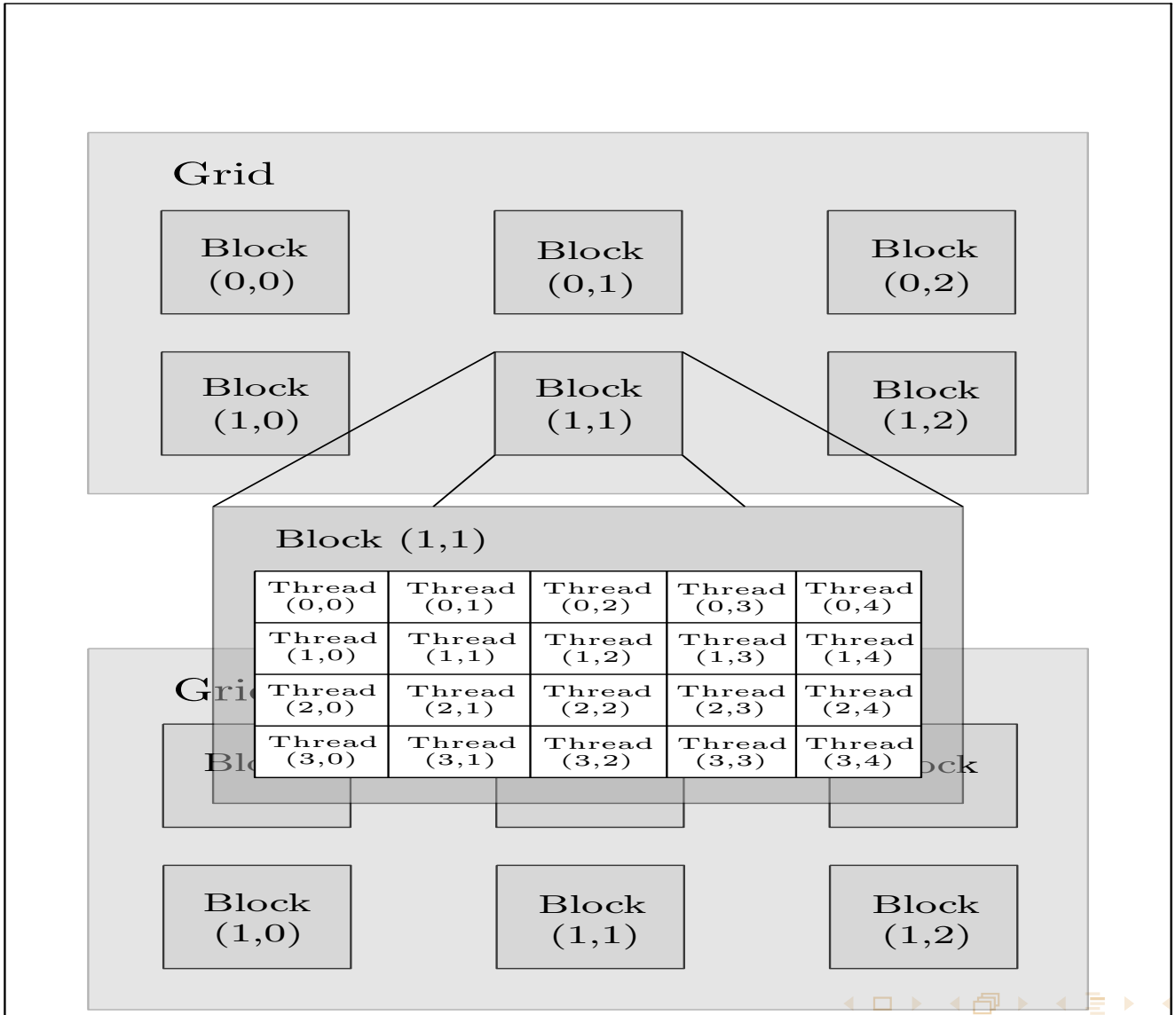
- Introduced in 2007 with NVIDIA Tesla architecture.
- "C-like" language to express programs that run on GPUs using the compute-mode hardware interface.
- Straightforward APIs to manage devices, memory ...

Thread Hierarchy



Thread Hierarchy

Thread IDs can be up to 3-dimensional (2D example shown)



First CUDA program

Goal: Add two vectors and write the result in a third vector.

C code:

```
1  for(int i = 0; i < N; ++i)
2     C[i] = A[i] + B[i];
```

CUDA kernel:

```
1  __global__ void vecAdd(float* d_A, float* d_B, float* d_C, int N) {
2     int idx = threadIdx.x + blockIdx.x * blockDim.x;
3     if(idx < N)
4       d_C[idx] = d_A[idx] + d_B[idx];
5  }
```

Kernel Launch:

```
int main(void) {
  ...
  vecAdd<<<numBlocks,threadsPerBlock>>>(d_A, d_B, d_C, N);
  ...
}
```

CUDA Function Declarations

	Executed on	**Only callable from**
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

- __global__ defines a kernel. It must return void
- A program may have several functions of each kind.

GPU Memory Allocation, Copying, Release

- cudaMalloc(void** pointer, size_t nbytes)
- cudaMemcpy(void *dst, void *src, size_t nbytes, cudaMemcpy direction)
- cudaFree(void *pointer)

- cudaMemcpy(...) blocks execution of CPU code until finished
- Kernel calls are non-blocking

Streams

- cudaMemcpyAsync(...) does not block execution of CPU code
- Possible to interleave kernel launches and memory transfer using streams and cudaMemcpyAsync(...)
  - Launches memory transfer and goes back executing CPU code
- Need to specify on which stream kernel should operate

```
1   cudaStream_t stream1;
2   cudaStreamCreate(&stream1);
3   cudaMemcpyAsync(d_array1, h_array1, ARRAY_BYTES,
        cudaMemcpyHosttoDevice, stream1);
4   kernel<<<numBlocks, numThreadsPerBlock>>>(d_array);
5   cudaMemcpyAsync(h_array1, d_array1, ARRAY_BYTES,
        cudaMemcpyDeviceToHost, stream1);
6   cudaStreamDestroy(stream1);
```

The Warp

- CUDA kernels execute as Single Program Multiple Data (SPMD) programs.
- Groups of **32 threads** share an instruction stream. These are called **warps**.
- The threads of a warp always march together in lock-step fashion. They execute code in SIMD fashion.
- Warp-threads are fully synchronized. There is an implicit barrier after each step / instruction

A Thread Block

- There is no (global) synchronization among thread blocks.
- More than one thread block can run on the same SM at the same time.
- A block is not preempted until it completes its execution
- Number of resident thread blocks — those running on an SM depends on:
  - Maximum number of threads per SM. Generally 1536 or 2048 for the newer GPUs.
  - Shared memory per block and registers

Memory Organization

The different types of memory present on the GPU:

- Registers
- **Global memory**
- **Shared memory**
- Constant memory
- Texture memory

Memory Coalescing

- GPU never reads just single value from global memory
- Reads in chunks of data
- Maximum coalescing when (warp) threads read or write from contiguous memory locations
- Strided memory access is fine, but only if the stride is low. Ideally stride $= 1$

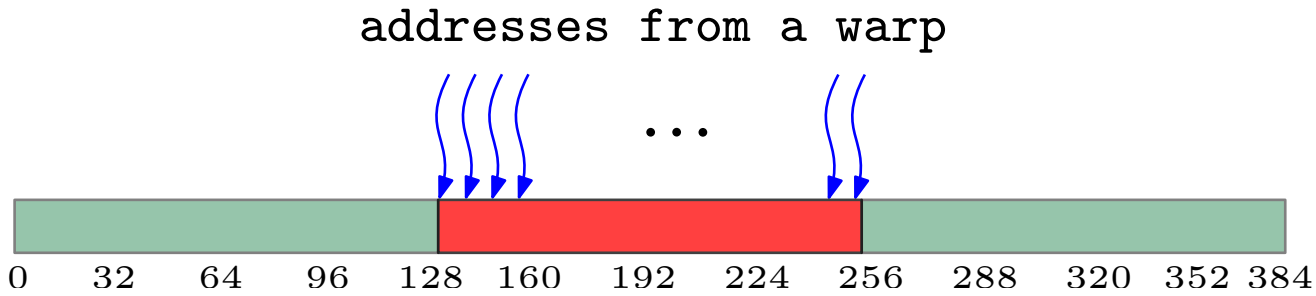


Image Source: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

CUDA Control Flow

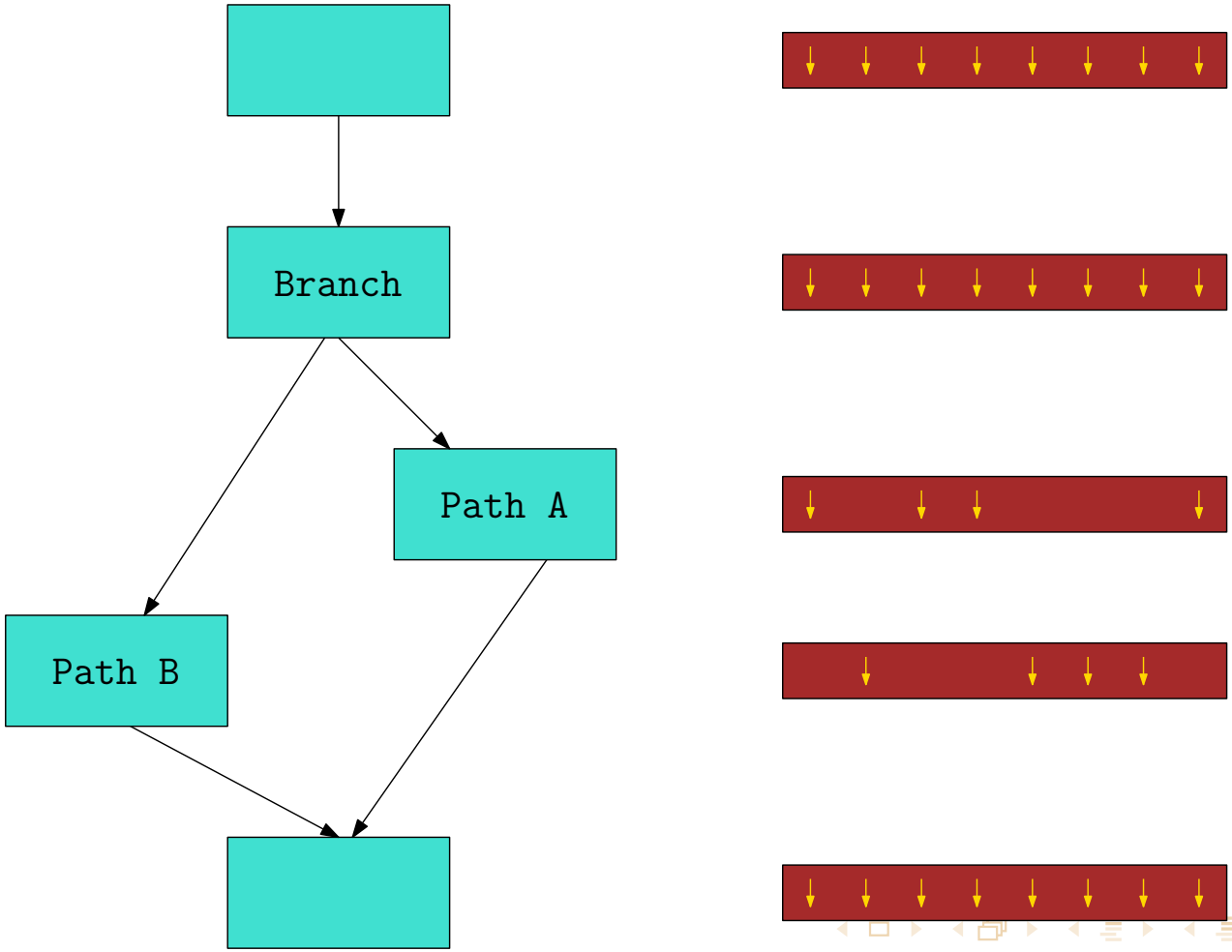
- Recall : All threads of a warp execute the same instruction

CUDA Control Flow

- Recall : All threads of a warp execute the same instruction
- Question: How do *if*-statements work then?

CUDA Control Flow

- Recall : All threads of a warp execute the same instruction
- Question: How do *if*-statements work then?
- Answer: Warp is serialized across the branches.



Control Flow Divergence

- Divergence does not affect correctness

Control Flow Divergence

- Divergence does not affect correctness
- Divergence can affect performance
  - **Condition** evaluating to different truth-values for **warp-threads** is bad

```
1         if(threadIdx.x % 2 == 0) // divergent branch
2             out[threadIdx.x] += in[threadIdx.x + 1];
```

Control Flow Divergence

- Divergence does not affect correctness
- Divergence can affect performance
  - **Condition** evaluating to different truth-values for **warp-threads** is bad

```
1           if(threadIdx.x % 2 == 0) // divergent branch
2               out[threadIdx.x] += in[threadIdx.x + 1];
```

Can be rewritten as:

```
1           // Strided index and non-divergent branch
2               int index = 2 * threadIdx.x;
3               out[index] += in[index+1];
```

Matrix Transpose

For a matrix $A_{m \times n}$, its transpose, $A^T_{n \times m}$ is obtained by interchanging A's rows and columns.

C code:

```
1 for(i = 0; i < n; i++)
2   for(j = 0; j < m; j++)
3     B[j][i] = A[i][j];
```

Matrix Transpose

For a matrix $A_{m \times n}$, its transpose, $A^T_{n \times m}$ is obtained by interchanging A's rows and columns.

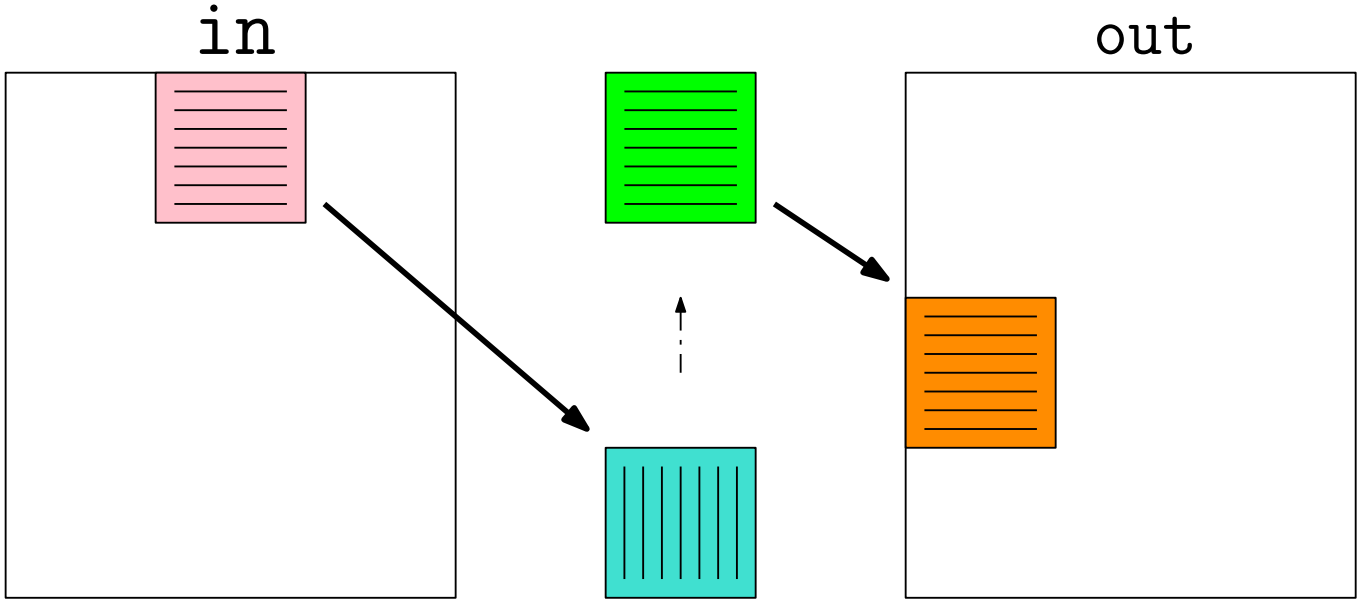
C code:

```
1 for(i = 0; i < n; i++)
2   for(j = 0; j < m; j++)
3     B[j][i] = A[i][j];
```

DEMO !!

Transpose with Shared Memory

- Use shared memory as a *staging area*.



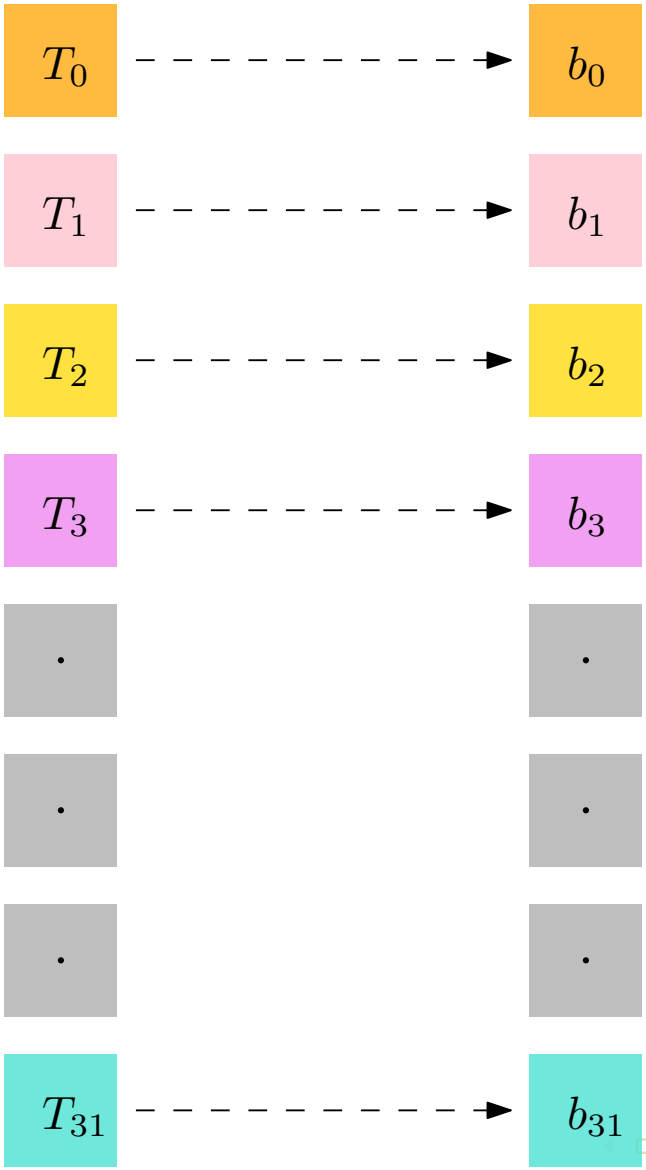
- Need __syncthreads()

Shared Memory Architecture

- Shared memory divided into 32 banks
- Shared memory as fast as registers
- Many threads accessing memory
  - Therefore, shared memory divided into banks.
  - Successive 32-bits assigned to successive banks
- Each bank services one address/cycle
  - As many accesses as banks
- Two threads access the same bank
  - Bank Conflict
  - Accesses are serialized

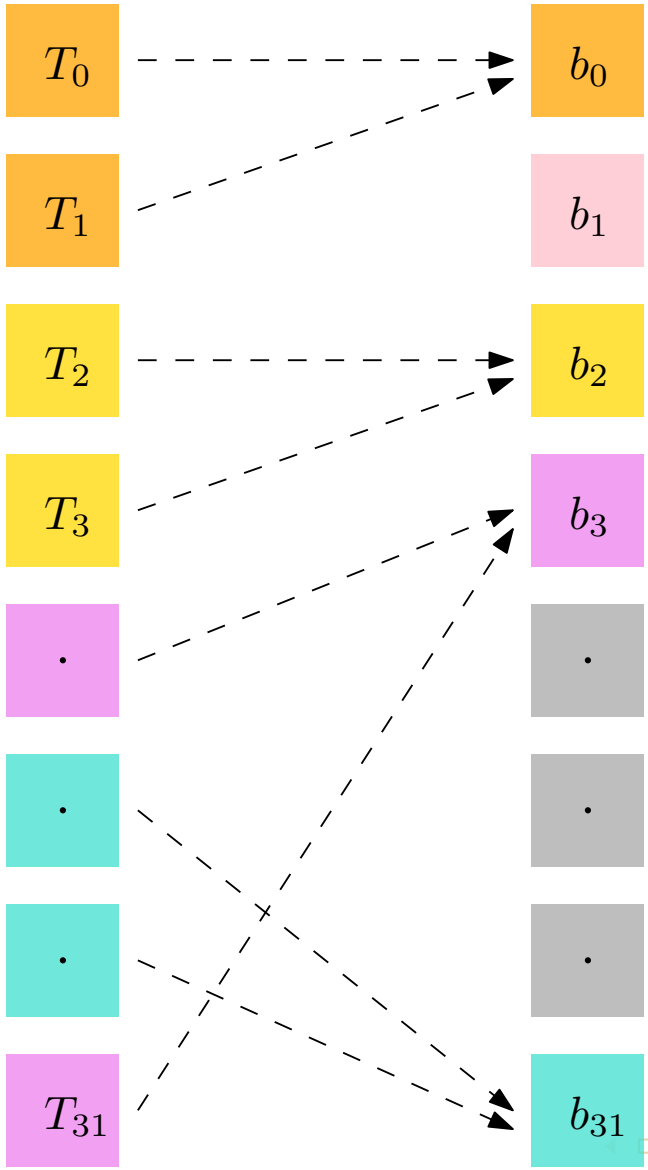
Shared Memory Architecture

Bank-Conflict Free



Shared Memory Architecture

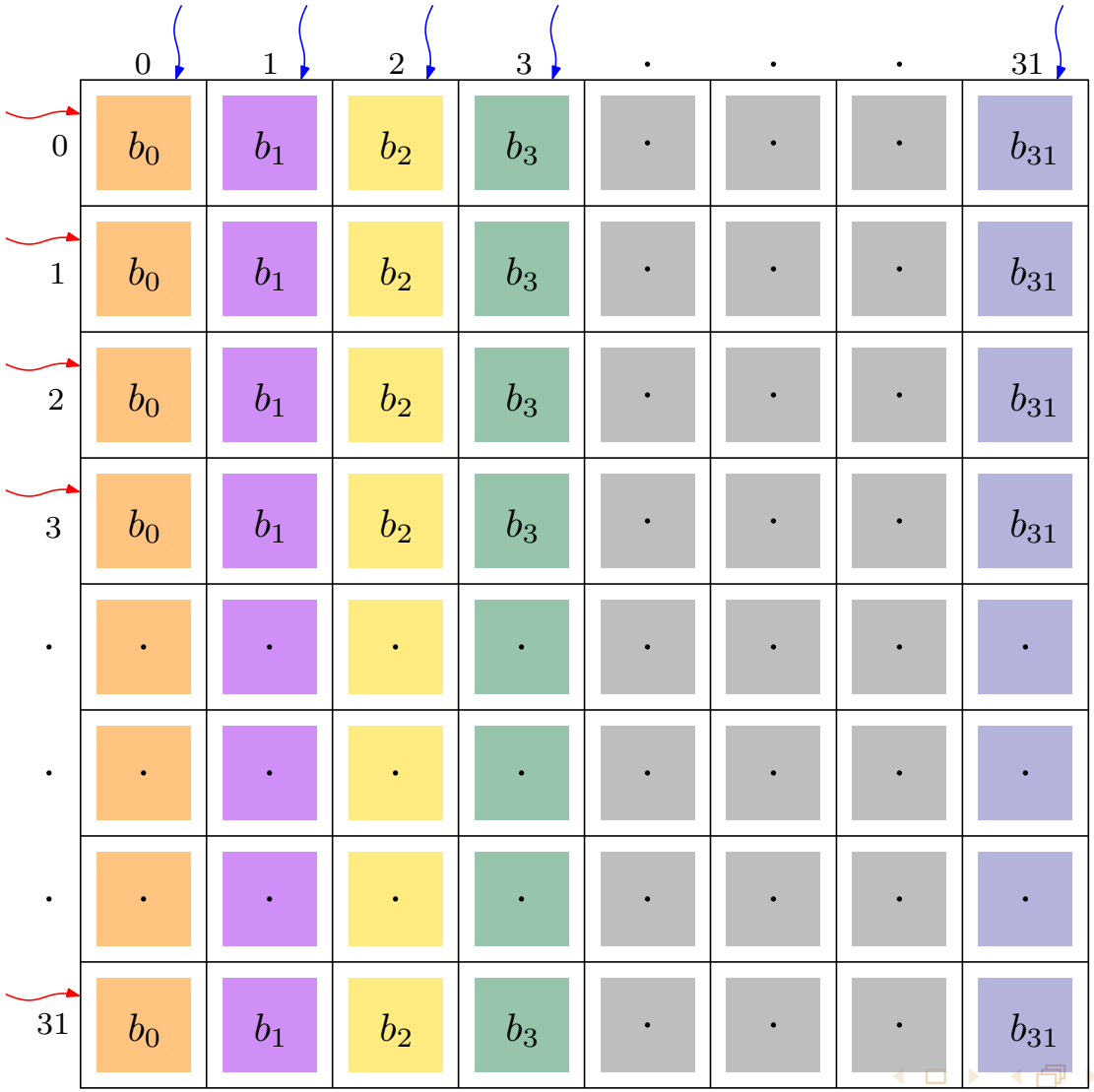
Bank Conflict



Transpose with Shared Memory

Bank Conflicts

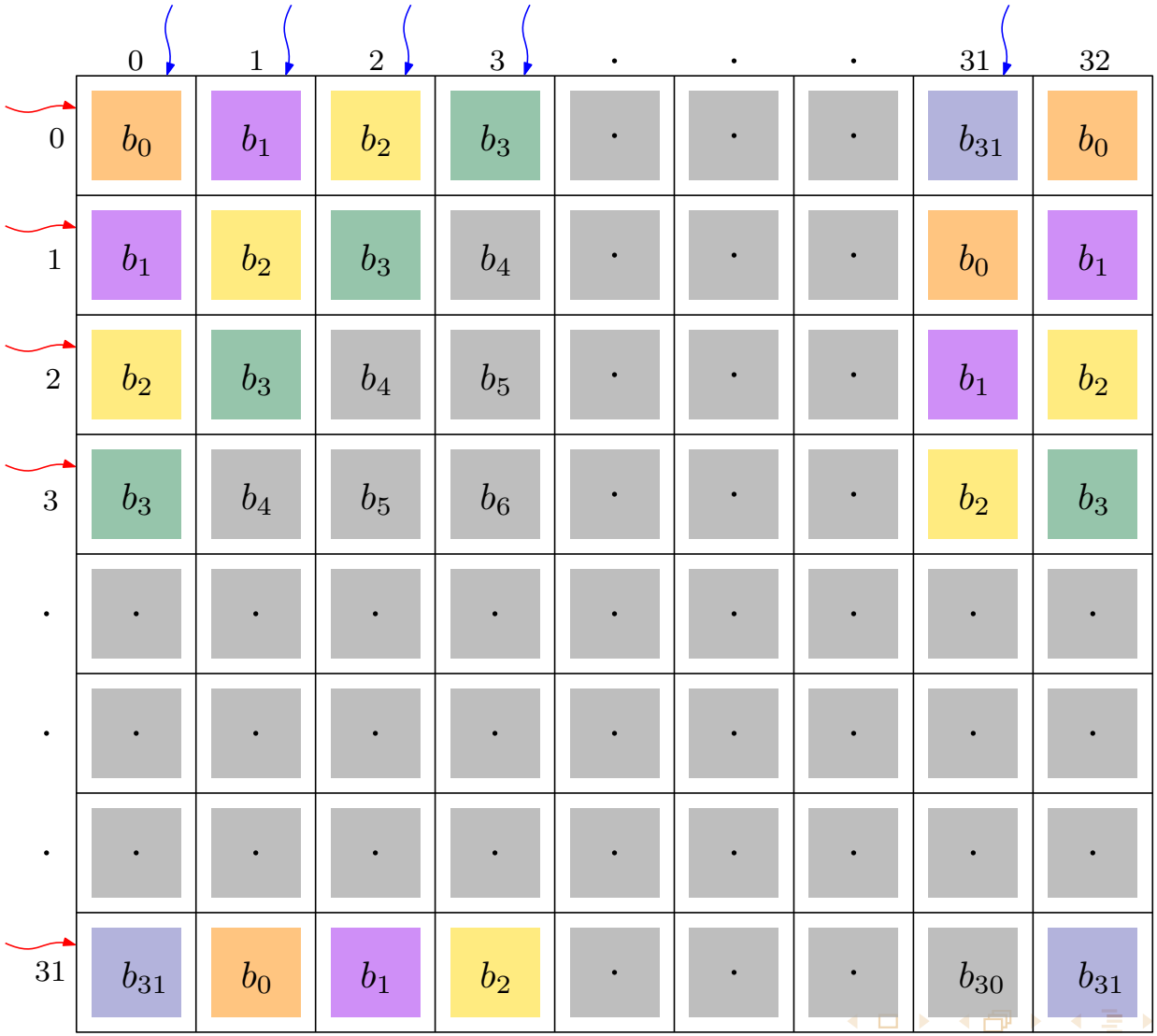
Consider: A tile of size 32×32 ; __shared__ float shm[32][32]



Transpose with Shared Memory

Bank-Conflict free

Solution: __shared__ float shm[32][33]



CUDA synchronization constructs

- __syncthreads()
  - Barrier : wait for all threads in the block to arrive at this point
- Atomic operations
  - Atomic operations serialize memory access, so performance takes a hit
  - Essential (sometimes) to ensure correctness
  - example: `float atomicAdd(float* addr, float val)`
  - Atomic operations on both global memory and shared memory variables

CUDA synchronization constructs

- __syncthreads()
  - Barrier : wait for all threads in the block to arrive at this point
- Atomic operations
  - Atomic operations serialize memory access, so performance takes a hit
  - Essential (sometimes) to ensure correctness
  - example: `float atomicAdd(float* addr, float val)`
  - Atomic operations on both global memory and shared memory variables

DEMO: Histogram

Exercise: 1D convolution

Given an input vector, `input` populate the vector, `output` in the following manner:

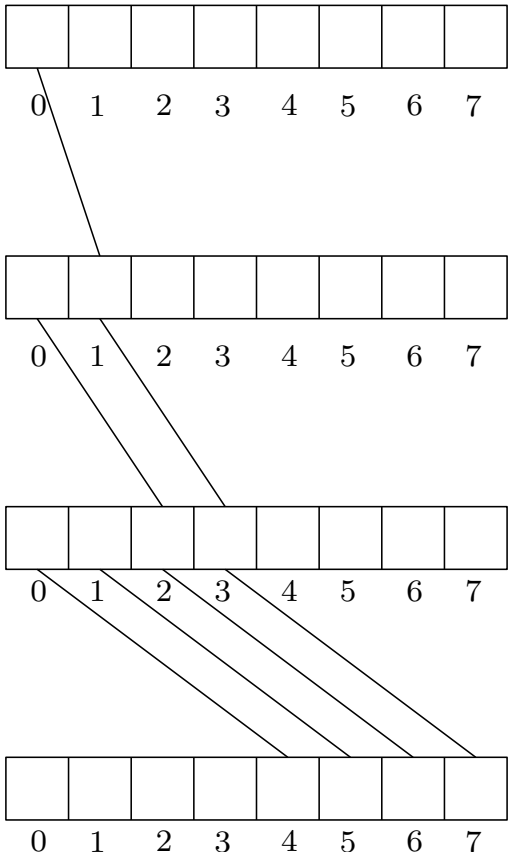
$$output[i] = (input[i] + input[i+1] + input[i+2])/3.f;$$

Hint: `Assign 1 thread per output element`

Reduction

```
1 int x = A[0];
2 for(i = 1; i < N; i++)
3    x += A[i]; // + can be replaced with any associative operator
```

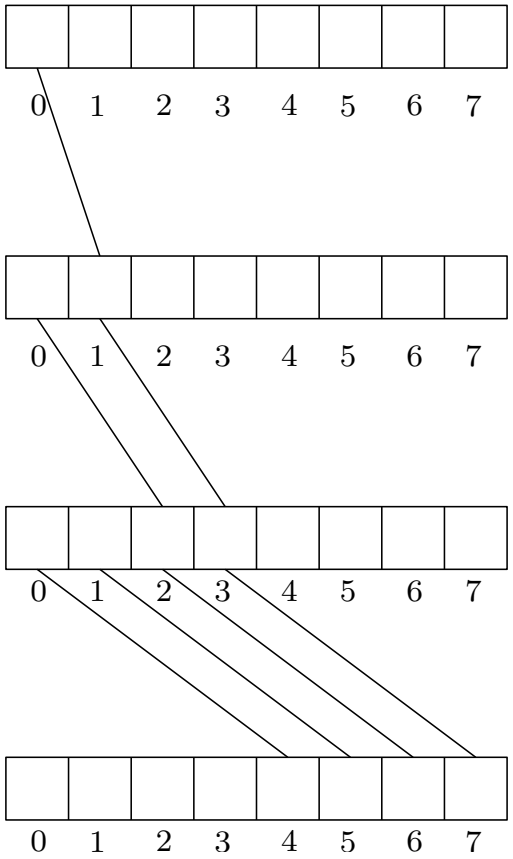
Parallel Reduce



Reduction

```
1 int x = A[0];
2 for(i = 1; i < N; i++)
3    x += A[i]; // + can be replaced with any associative operator
```

Parallel Reduce



CUDA Register Pressure

- If we use too many registers per thread, there is overflow.
- The spilled registers go to global memory – that's BAD.
- From a 20 cycle latency, we get to 800 cycle latency !!
- Solution: Reduce the thread local variables (live at the same time) we are using.

Occupancy

- The number of warps running concurrently vs the maximum number of warps that could
- Want lots of warps to hide latency
- Limited by Registers and Shared Memory

Occupancy

- The number of warps running concurrently vs the maximum number of warps that could
- Want lots of warps to hide latency
- Limited by Registers and Shared Memory
- Increasing occupancy not always translates to performance.
- Look for Instruction Level Parallelism.

Resources and Further Reading

- The CUDA C programming guide (http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)
- You may enjoy the free Udacity Course: **Introduction to Parallel Programming using CUDA** (https://www.udacity.com/course/cs344)
- The **Thrust Library** is a useful collection library for CUDA.
- PyCUDA – for people who love Python
- Kokkos – Write performance portable code for various parallel architectures.

Somesh Singh
somesh.singh1992@gmail.com

Thank You for Attending !