Warps and Reduction Algorithms

1. more on Thread Execution
   - block partitioning into warps
   - single-instruction, multiple-thread, and divergence
2. Parallel Reduction Algorithms
   - computing the sum or the maximum
   - a kernel for the parallel sum
   - a kernel with less thread divergence

MCS 572 Lecture 34
Introduction to Supercomputing
Jan Verschelde, 9 November 2016

Warps and Reduction Algorithms

1. more on Thread Execution
   - block partitioning into warps
   - single-instruction, multiple-thread, and divergence
2. Parallel Reduction Algorithms
   - computing the sum or the maximum
   - a kernel for the parallel sum
   - a kernel with less thread divergence

block partitioning into warps

The grid of threads are organized in a two level hierarchy:

- the grid is 1D, 2D, or 3D array of blocks; and
- each block is 1D, 2D, or 3D array of threads.

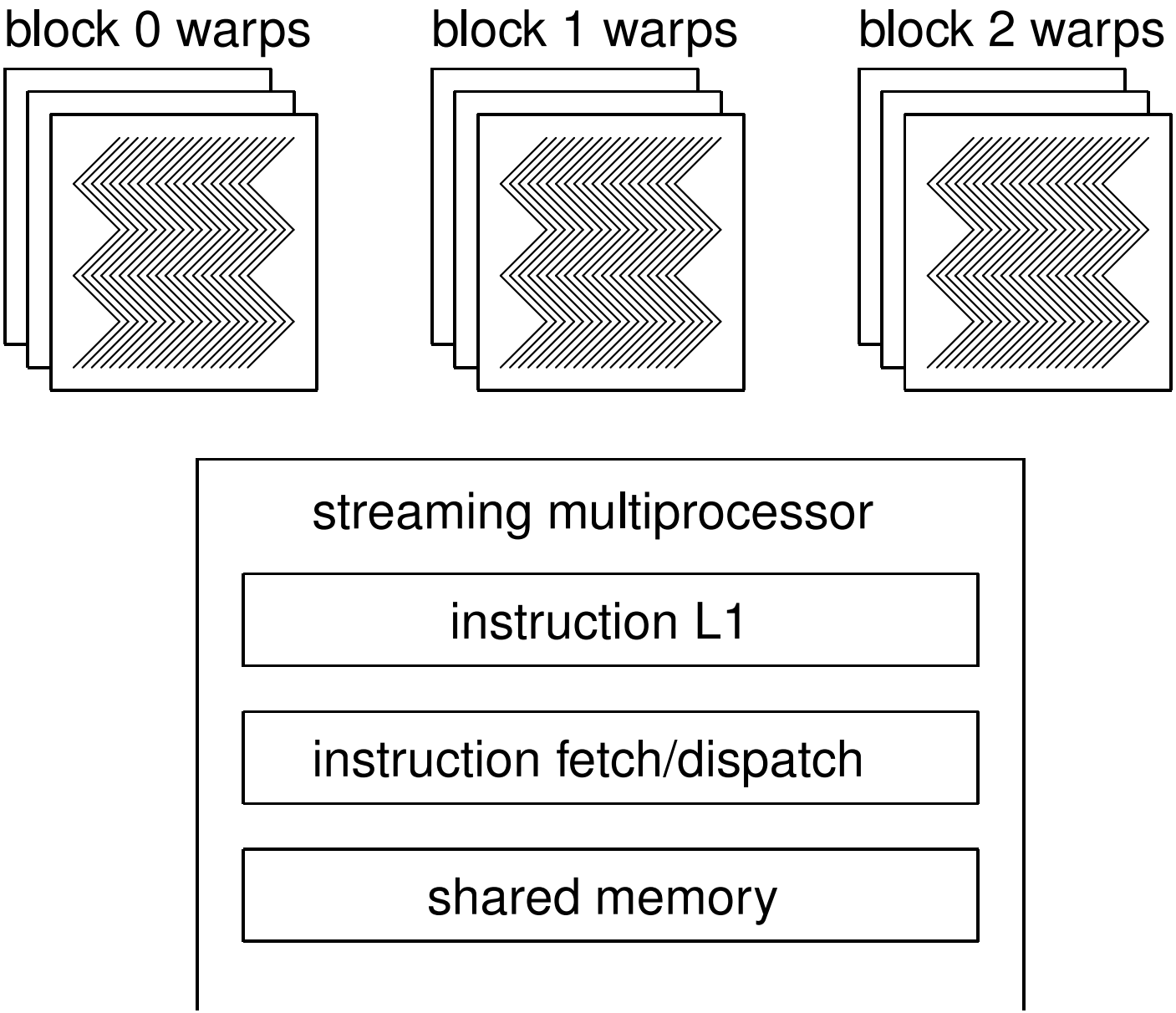
Blocks can execute in any order.

Threads are bundled for execution.

Each block is partitioned into *warps*.

On the Tesla C2050/C2070, K20C, and P100,
each warp consists of 32 threads.

thread scheduling



thread indices of warps

All threads in the same warp run at the same time.

The partitioning of threads in a one dimensional block,
for warps of size 32:

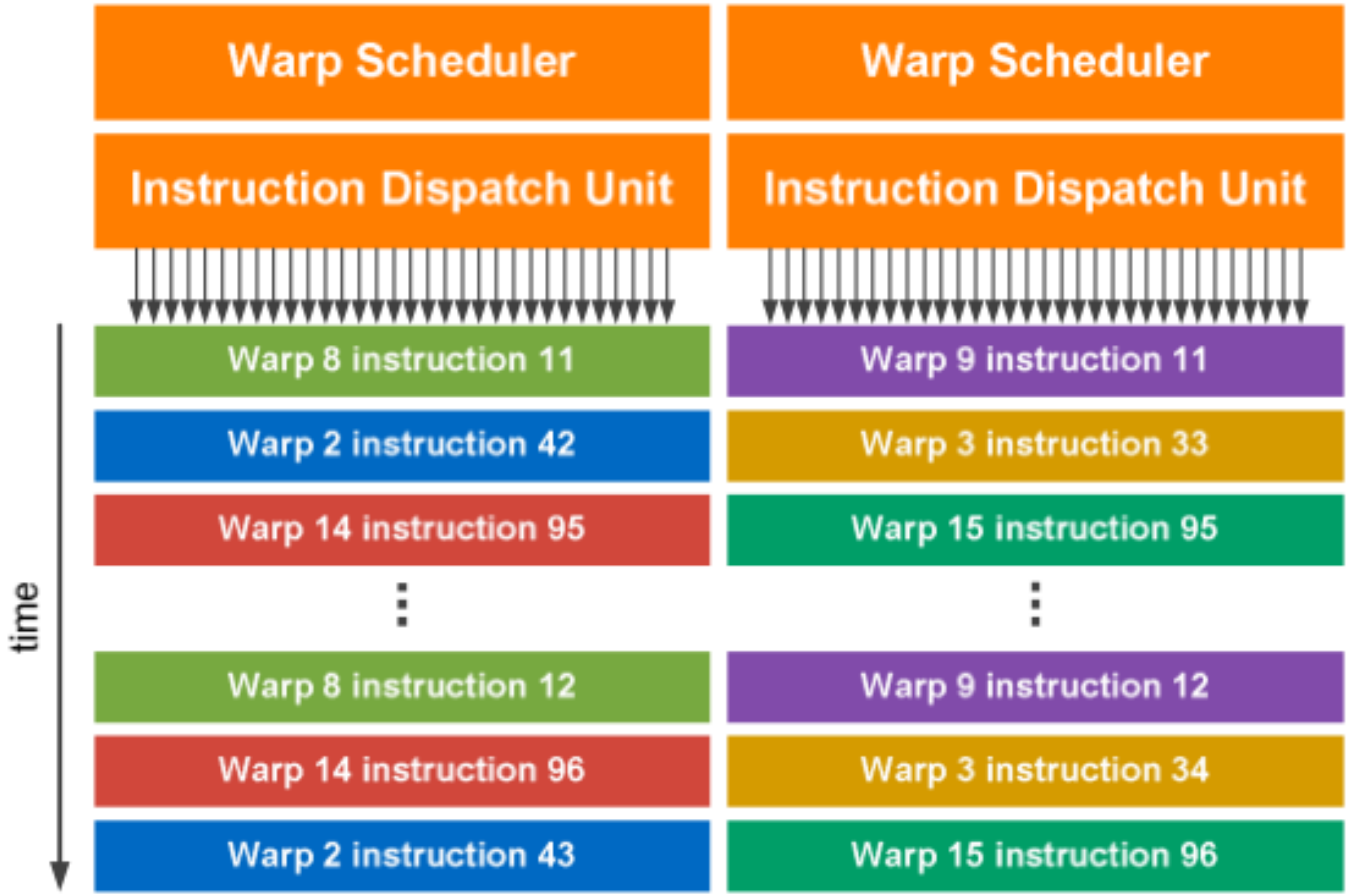
- warp 0 consists of threads 0 to 31 (value of `threadIdx`),
- warp w starts with thread $32w$ and ends with thread $32(w+1)-1$,
- the last warp is padded so it has 32 threads.

In a two dimensional block, threads in a warp are ordered along a lexicographical order of (`threadIdx.x`,`threadIdx.y`).

For example, an 8-by-8 block has 2 warps (of 32 threads):

- warp 0 has threads $(0,0),(0,1),\ldots,(0,7)$, $(1,0),(1,1),\ldots,(1,7)$, $(2,0),(2,1),\ldots,(2,7)$, $(3,0),(3,1),\ldots,(3,7)$; and
- warp 1 has threads $(4,0),(4,1),\ldots,(4,7)$, $(5,0),(5,1),\ldots,(5,7)$, $(6,0),(6,1),\ldots,(6,7)$, $(7,0),(7,1),\ldots,(7,7)$.

dual warp scheduler



from the NVIDIA Whitepaper

NVIDIA Next Generation CUDA Compute Architecture: Fermi.

why so many warps?

Why give so many warps to a streaming multiprocessor
if there only 32 can run at the same time?

Answer: to efficiently execute long latency operations.

What about latency?

- A warp must often wait for the result of a global memory access
  — — an example of a long latency operation — —
  and is therefore not scheduled for execution.
- If another warp is ready for execution,
  then that warp can be selected to execute the next instruction.

The mechanism of filling the latency of expensive operation with work from other threads is known as *latency hiding*.

zero overhead thread scheduling

Warp scheduling is used for other types of latency operations:

- pipelined floating point arithmetic,
- branch instructions.

With enough warps, the hardware will find a warp to execute,
in spite of long latency operations.

The selection of ready warps for execution introduces no idle time and is referred to as *zero overhead thread scheduling*.
The long waiting time of warp instructions is hidden by executing instructions of other warps.

In contrast, CPUs tolerate latency operations with

- cache memories, and
- branch prediction mechanisms.

applied to matrix-matrix multiplication

For matrix-matrix multiplication,
what should the dimensions of the blocks of threads be?

We narrow the choices to three: 8×8, 16×16, or 32×32?

Considering that the C2050/C2070 has 14 streaming multiprocessors:

1. $32 \times 32 = 1,024$ equals the limit of threads per block.
2. $8 \times 8 = 64$ threads per block and $1,024/64 = 12$ blocks.
3. $16 \times 16 = 256$ threads per block and $1,024/256 = 4$ blocks.

Note that we must also take into account the size of shared memory when executing tiled matrix matrix multiplication.

Warps and Reduction Algorithms

1. more on Thread Execution
   - block partitioning into warps
   - single-instruction, multiple-thread, and divergence
2. Parallel Reduction Algorithms
   - computing the sum or the maximum
   - a kernel for the parallel sum
   - a kernel with less thread divergence

Single-Instruction, Multiple-Thread (SIMT)

In multicore CPUs, we use Single-Instruction, Multiple-Data (SIMD): the multiple data elements to be processed by a single instruction must be first collected and packed into a single register.

In SIMT, all threads process data in their own registers.

In SIMT, the hardware executes an instruction for all threads in the same warp, before moving to the next instruction.

This style of execution is motivated by hardware costs constraints.

The cost of fetching and processing an instruction is amortized over a large number of threads.

paths of execution

Single-Instruction, Multiple-Thread works well when all threads within a warp follow the same control flow path.

For example, for an *if-then-else* construct, it works well

- when either all threads execute the *then* part,
- or all execute the *else* part.

If threads within a warp take different control flow paths, then the SIMT execution style no longer works well.

thread divergence

Considering the *if-then-else* example, it may happen that

- some threads in a warp execute the *then* part,
- other threads in the same warp execute the *else* part.

In the SIMT execution style, multiple passes are required:

- one pass for the *then* part of the code, and
- another pass for the *else* part.

These passes are sequential to each other
and thus increase the execution time.

If threads in the same warp follow different paths of control flow,
then we say that these threads *diverge* in their execution.

other examples of thread divergence

Consider an iterative algorithm with a loop

- some threads finish in 6 iterations,
- other threads need 7 iterations.

In this example, two passes are required:

1. one pass for those threads that do the 7th iteration,
2. another pass for those threads that do not.

In some code, decisions are made on the `threadIdx` values:

- For example: `if(threadIdx.x > 2){ ... }`.
- The loop condition may be based on `threadIdx`.

An important class where thread divergence is likely to occur is the class of reduction algorithms.

Warps and Reduction Algorithms

1. more on Thread Execution
   - block partitioning into warps
   - single-instruction, multiple-thread, and divergence

2. Parallel Reduction Algorithms
   - computing the sum or the maximum
   - a kernel for the parallel sum
   - a kernel with less thread divergence

reduction algorithms

A reduction algorithm extracts one value from an array, e.g.:

- the sum of an array of elements,
- the maximum or minimum element in an array.

A reduction algorithm visits every element in the array,
using a current value for the sum or the maximum/minimum.

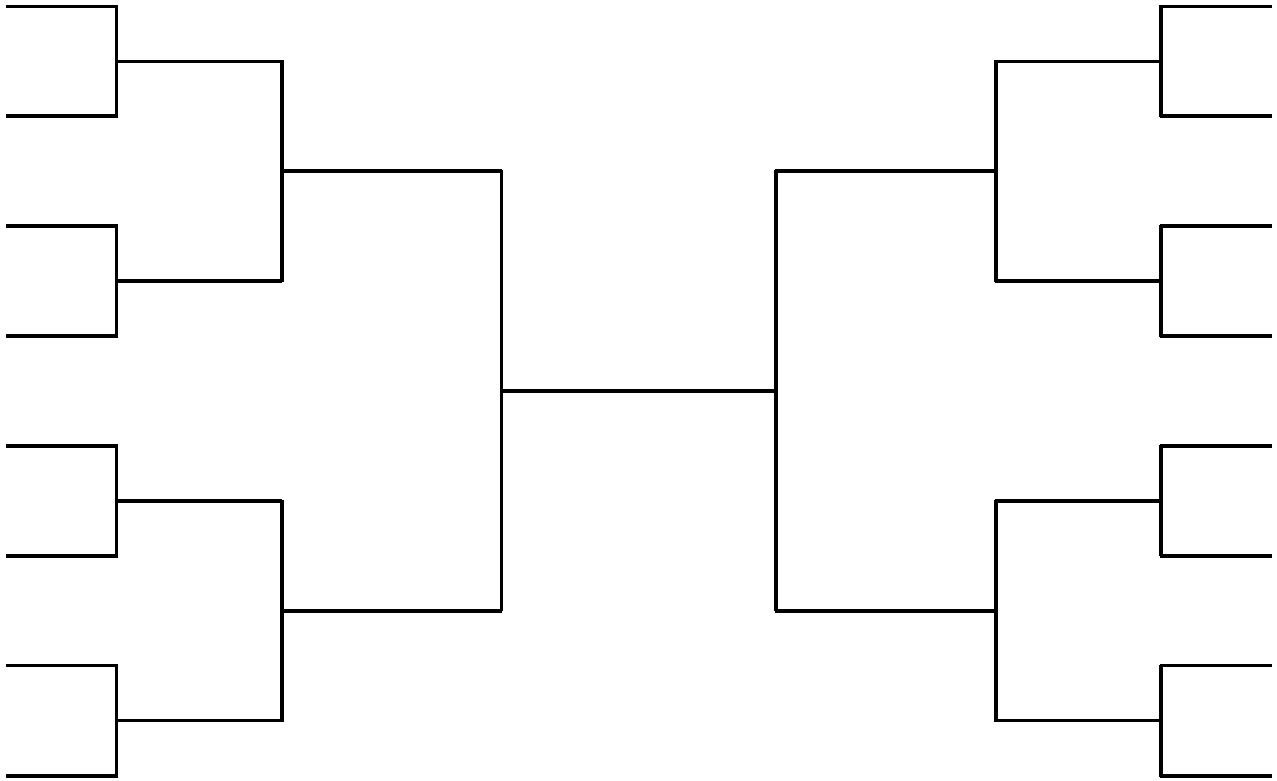
Large enough arrays motivate parallel execution of the reduction.

To reduce n elements, $n/2$ threads take $\log_2(n)$ steps.

Reduction algorithms take only 1 flop per element loaded:

- not *compute bound*, that is: limited by flops performance,
- but *memory bound*, that is: limited by memory bandwidth.

a tournament

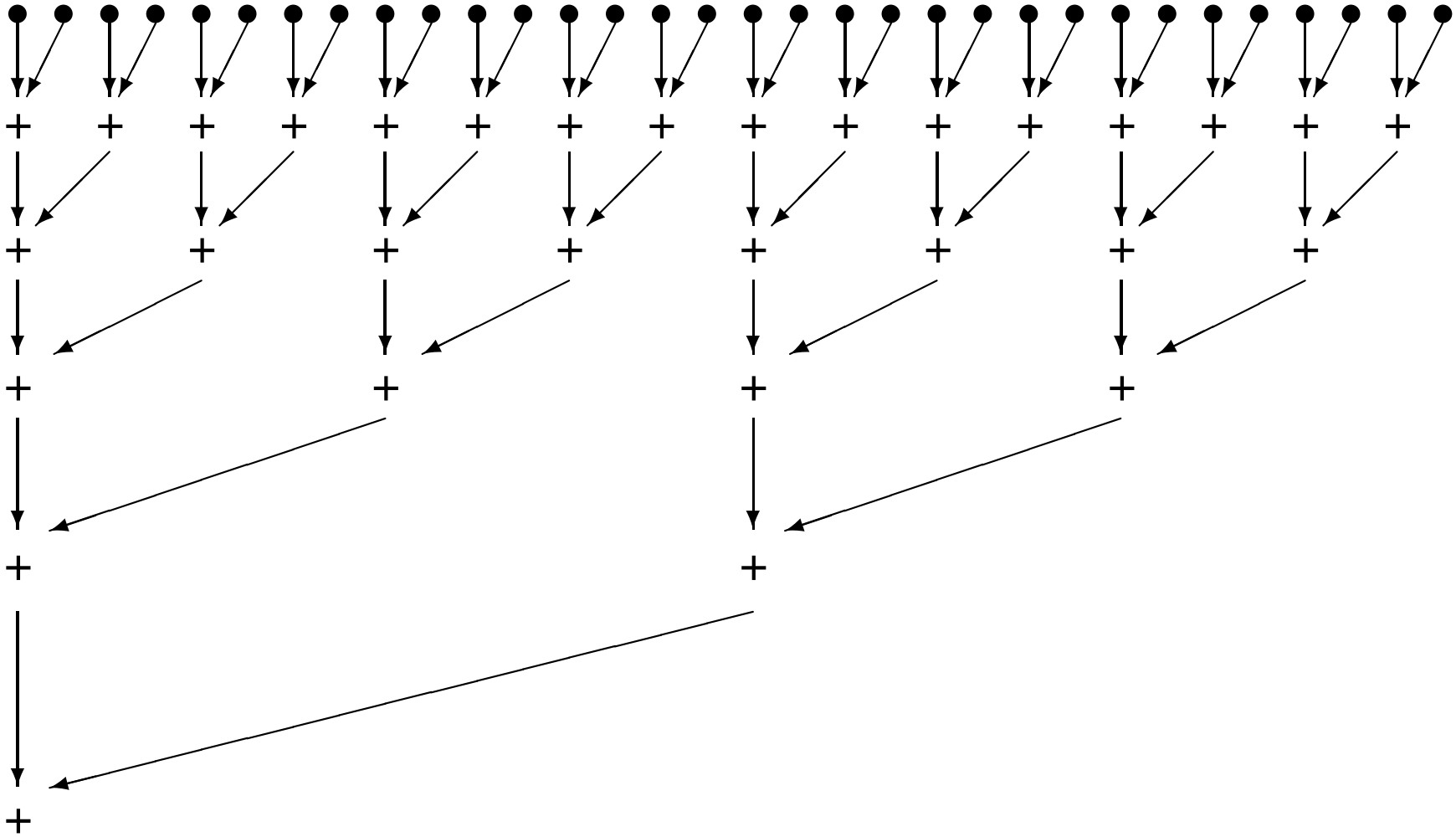


Warps and Reduction Algorithms

1. more on Thread Execution
   - block partitioning into warps
   - single-instruction, multiple-thread, and divergence

2. Parallel Reduction Algorithms
   - computing the sum or the maximum
   - a kernel for the parallel sum
   - a kernel with less thread divergence

summing 32 numbers



code in a kernel to sum numbers

The original array is in the global memory
and copied to shared memory for a thread block to sum.

```
__shared__ float partialSum[];

int t = threadIdx.x;
for(int stride = 1; stride < blockDim.x; stride *= 2)
{
   __syncthreads();
   if(t % (2*stride) == 0)
      partialSum[t] += partialSum[t+stride];
}
```

The reduction is done *in place*, replacing elements.
The `__syncthreads()` ensures that all partial sums from the previous iteration have been computed.

thread divergence in the first kernel

Because of the statement

```
if(t % (2*stride) == 0)
   partialSum[t] += partialSum[t+stride];
```

the kernel clearly has thread divergence.

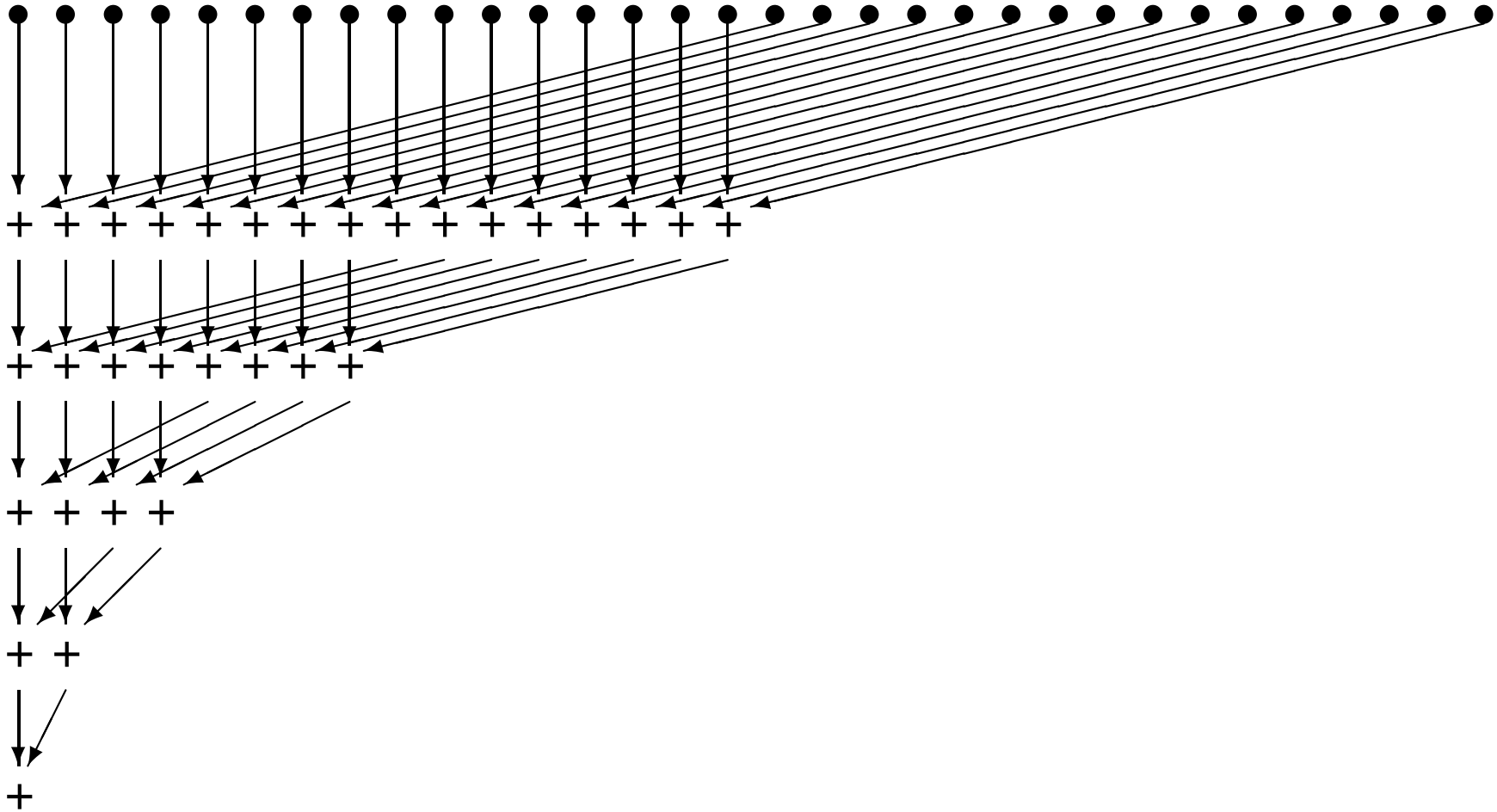
In each iteration, two passes are needed to execute all threads, even though fewer threads will perform an addition.

Warps and Reduction Algorithms

1. more on Thread Execution
   - block partitioning into warps
   - single-instruction, multiple-thread, and divergence

2. Parallel Reduction Algorithms
   - computing the sum or the maximum
   - a kernel for the parallel sum
   - a kernel with less thread divergence

a different summation algorithm



the revised kernel

The original array is in the global memory
and copied to shared memory for a thread block to sum.

```
__shared__ float partialSum[];

int t = threadIdx.x;
for(int stride = blockDim.x >> 1; stride > 0;
    stride >> 1)
{
   __syncthreads();
   if(t < stride)
      partialSum[t] += partialSum[t+stride];
}
```

The division by 2 is done by shifting the stride value to the right by 1 bit.

why less thread divergence?

Af first, there seems no improvement, because of the `if`.

Consider a block of 1,024 threads, partitioned in 32 warps.
A warp consists of 32 threads with consecutive `threadIdx` values:

- all threads in warp 0 to 15 execute the add statement,
- all threads in warp 16 to 31 skip the add statement.

All threads in each warp take the same path $\Rightarrow$ no thread divergence.

If the number of threads that execute the add drops below 32,
then thread divergence still occurs.

Thread divergence occurs in the last 5 iterations.

bibliography

- S. Sengupta, M. Harris, and M. Garland.
  Efficient parallel scan algorithms for GPUs.
  Technical Report NVR-2008-003, NVIDIA, 2008.
- M. Harris. **Optimizing parallel reduction in CUDA.**
  White paper available at `http://docs.nvidia.com`.

summary and exercises

We covered §4.5 and §6.1 in the book of Kirk & Hwu, and discussed warp scheduling, latency hiding, SIMT, and thread divergence.
To illustrate the concepts we discussed two reduction algorithms.

1. Consider the code `matrixMul` of the GPU computing SDK.
   Look up the dimensions of the grid and blocks of threads.
   Can you (experimentally) justify the choices made?
2. Write code for the two summation algorithms we discussed.
   Do experiments to see which algorithm performs better.
3. Apply the summation algorithm to the composite trapezoidal rule.
   Use it to estimate π via $\frac{\pi}{4} = \int_0^1 \sqrt{1-x^2}dx$.