Optimizing the Performance of a CUDA Kernel

Peter Pacheco
University of San Francisco
peter@usfca.edu

March 10, 2016

- High-Performance Computing: widespread use of Graphics Processing Units (GPUs) to improve performance of conventional CPUs.
- This talk: brief introduction to General Purpose Programming on GPUs (GPGPU).
- Focus of talk: programming Nvidia GPUs with CUDA — Nvidia's API for GPGPU.
- Nearly ten years since release of first CUDA SDK.
- Thousands of applications developed using CUDA, and CUDA changed dramatically.
- This talk: look at a couple of simple applications,
- Purpose: give feel for development of CUDA applications.

General Purpose Computing for GPU's

- 1990's: *huge* demand for highly realistic computer games
- Result: huge increase in capabilities of graphics hardware.
- Early 2000's: GPUs so powerful programmers tried to use for *general purpose* programming.
- Result: beginning of GPGPU.

- Initially GPGPU extremely challenging:
- Programmers "tricked" graphics APIs (Direct3D, OpenGL) into doing CPU-like operations on processors designed to manipulate polygons, add shading, etc.
- More tractable APIs for GPGPU soon developed.
- 2003: Stanford *Brook* API.
- Since then several APIs developed:
  - ATI: Stream
  - Nvidia: CUDA,
  - Industry group (led by Apple): OpenCL.
  - Nvidia, Portland Group, Cray: OpenACC

What's Happening Now?

- ATI supporting OpenCL
- Nvidia supports CUDA and OpenACC
- CUDA most widely used GPGPU API
- Has a very sophisticated development environment.
- Many more applications coded in CUDA than any other GPGPU API.
- Main drawback to CUDA: only available for Nvidia GPU's.

Nvidia GPUs

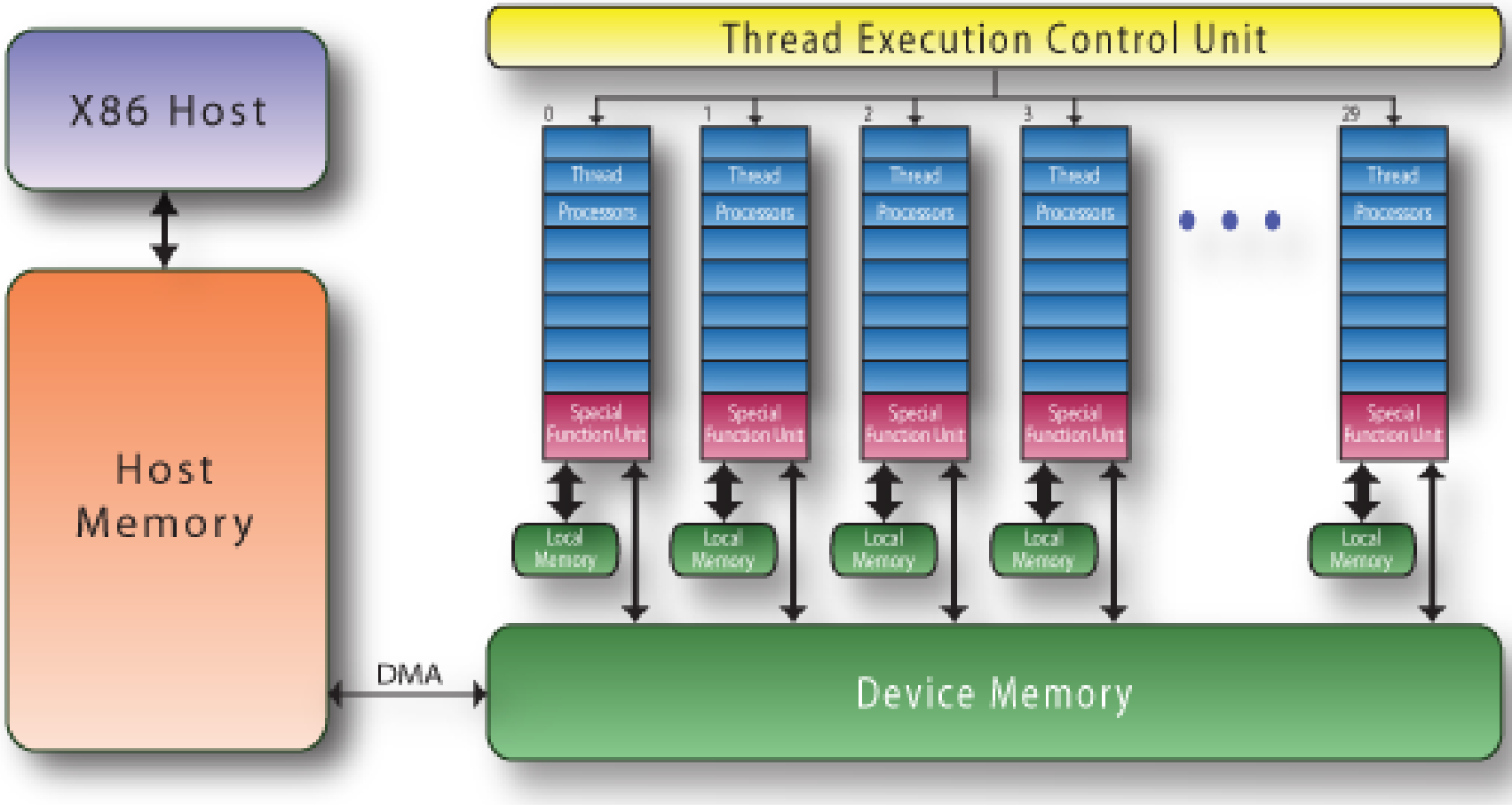
- Design of GPUs not standardized.
- Even *Terminology* not standardized.
- We use CUDA, so talk about Nvidia GPU's and use Nvidia's terminology.
- Nvidia GPUs: composed of "streaming multiprocessors" (SM or SMX's — "SM neXt generation").
- Each SM: 8 or more "cores" or "thread processors"

- Core: CPU without a control unit.
- Single control unit for all cores.
- It can issue different instruction to each SM.
- First approximation: within single multiprocessor cores operate in *SIMD* fashion.
- Recall SIMD: Single-Instruction Multiple-Data
- SIMD execution: each core executes same instruction as other cores on its data; or is idle.

```
if (my_x >= 0)
   my_y = my_x;
else
   my_y = -my_x;
```

- Time 0: All threads execute test.
- Time 1:
  - Threads with `my_x >= 0` assign `my_y = my_x`
  - Threads with `my_x < 0` idle.
- Time 2:
  - Threads with `my_y < 0` assign `my_y = -my_x`
  - Threads with `my_x >= 0` idle.

- Called thread *divergence*.
- *However*, threads running on different multiprocessors *can* execute different instructions.
- Each SM has block of "shared" or "local" memory. (Nvidia terminology: "shared" = shared among the cores in a single SM.)
- Also "global" or "device" memory shared by all SM's (in a single GPU).


X86 Host
Host Memory
DMA
Thread Execution Control Unit
0
1
2
3
29
Thread Processors
Special Function Unit
Local Memory
Device Memory

- Specs for a "Quadro 600" GPU

```
Multiprocessors:                2
Cores per SM:                  48
GPU clockspeed:              1.28 GHz
Shared Memory per SM:          48 KB
Global Memory:                  1 GB
```

- Quadro 600 fairly modest GPU.
- Top-of-the-line processor for GPGPU, Tesla K80, $\sim$ 5000 total cores, $>$ 25 SMs.

How to exploit hundreds or thousands of cores?

- Hardware context switch: when thread is idled (e.g., waiting for data from global memory), almost no delay in starting another thread.
- Program: avoid thread divergence. (More later)
- Program: exploit hardware context switch: run *many* threads on each SM.

CUDA

- CUDA originally "Compute-Unified Device Architecture:" program both CPU and GPU with CUDA program.
- CUDA source stored in ".cu" files.
- Source code very similar to C/C++ code.
- Many C programs (and some C++ programs) can be compiled by CUDA compiler and run on *CPU.*
- Note: CUDA *not* a library — unlike MPI, Pthreads, OpenMP, and many other API's for parallel computing.
- CUDA requires modifications to C/C++ compiler.

- CUDA programs start execution in a main function that runs on CPU or *host*.
- Main function (and other C/C++ functions) execute C/C++ statements in same way as ordinary C/C++ programs.
- Most important difference between CUDA programs and C/C++ programs: "kernels."
- Kernel: a CUDA function called by program running on the host, but run on GPU or *device.*
- Typically quite short.

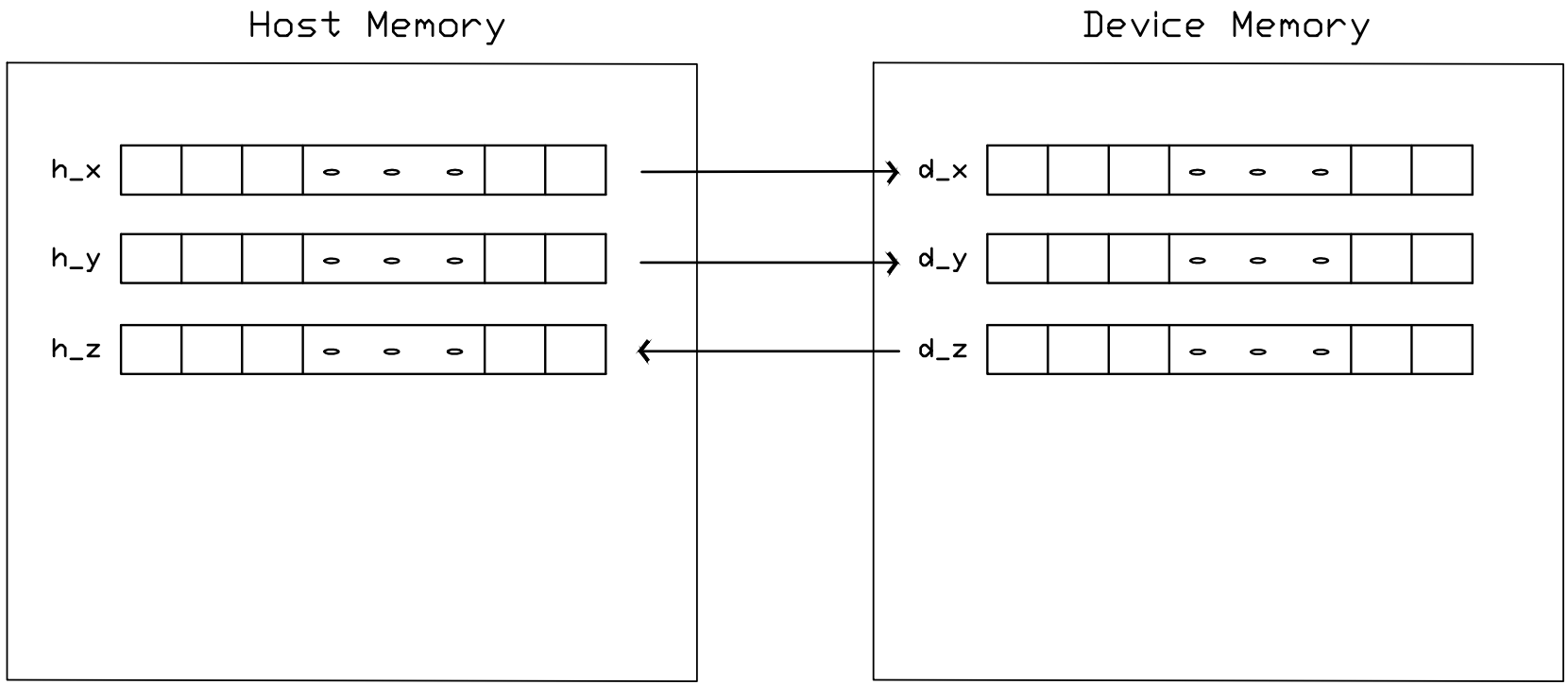
Example: Vector Addition

- Implement CUDA program that does vector addition on the device:

```
void Vec_add(float x[], float y[], float z[],
      int n) {
   for (int i = 0; i < n; i++)
      z[i] = x[i] + y[i];
}
```

- Single source file `vec_add.cu`
- Usual C header files: `stdio.h`, `stdlib.h`, `math.h`
- Main function: ordinary C main function *except* for call to kernel.

- Main: allocate storage for vectors on host and device: `malloc` on host, CUDA library `cudaMalloc` on device.
- (Recall: host and device separate memory.)
- Initialize `x, y` on host.
- Copy `x, y` to device: use `cudaMemcpy`


Host Memory
Device Memory
h_x
h_y
h_z
d_x
d_y
d_z

- CUDA analog of core "thread"
- CUDA analog of SM "thread block"
- Determine: number of threads per block and number of blocks
- Rule of thumb: use *many* more threads per block than cores per SM.
- More blocks than SMs:
  - Plus: SM's can context switch between threads in different blocks when a block is synchronizing (more later)
  - Minus: Limits availability of shared or local memory.

Call kernel and wait for it to complete:

```
Vec_add<<<block_count, threads_per_block>>>
      (d_x, d_y, d_z, n);

cudaThreadSynchronize();
```

Notes:

- Number of blocks and threads per block in triple angle brackets.
- Device memory addresses returned by `cudaMalloc` passed to kernel.

- Copy result, `z`, from device to host.
- Print result
- Free memory: `free` on host, `cudaFree` on device
- Quit

Kernel code:

```
__global__ void Vec_add(float x[], float y[],
      float z[], int n) {
   int threads_per_block = blockDim.x;
   int my_block = blockIdx.x;
   int my_thread = threadIdx.x;
   int i = threads_per_block*my_block + my_thread;

   /* block_count*threads_per_block may be >= n */
   if (i < n) z[i] = x[i] + y[i];
}  /* Vec_add */
```

Notes:

- Single-program multiple data execution of kernel:
  - Call in main starts `block_count` $\times$ `threads_per_block` threads on device.
  - Threads divided into `block_count` thread blocks.
  - Each thread executes kernel code.
- Can have multidimensional blocks.
- Can have multidimensional collections of blocks or *grids*.
- `blockDim, blockIdx, threadIdx` initialized during function invocation.
- `i` determines component of `z` for thread running kernel

Example: Two blocks of four threads and $n = 6$.

```
blockDim.x = 4

blockIdx.x  ThreadIdx.x        i
    0           0          0 = 4*0 + 0
                1          1 = 4*0 + 1
                2          2 = 4*0 + 2
                3          3 = 4*0 + 3

    1           0          4 = 4*1 + 0
                1          5 = 4*1 + 1
                2          6 = 4*1 + 2
                3          7 = 4*1 + 3
```

- Kernel has return type `void`.
- Cannot pass by reference (either using C pointers, or C++ reference arguments): addresses on host are meaningless on device and vice-versa.
- Must "return" results via memory copy.

Performance:

```
n = 1048576, block count = 2048, threads
   per block = 512
Serial:  Intel Xeon E5-2609, 2.40GHz

Serial:                 4.65e-03 seconds
CUDA:                   7.783-04 seconds
```

CUDA version is almost 6 times faster

Thread Synchronization

- Vector addition easy to code: each thread read and wrote its *own* parts of arrays.
- So no need to synchronize the threads: no *race conditions.*
- *Race condition* occurs: two (or more) threads attempt to access shared data structure and *at least one access is a write.*

- Example: x is a shared variable initialized to 2.
- Two threads A and B computing private value and add it to x :

```
   Thread A                    Thread B
my_y = My_func();          my_y = My_func();
x += my_y;                 x += my_y;
```

- Here, both threads have "private" copy of `my_y`.
- Suppose A assigns `my_y = 3` and B assigns `my_y = 5`.
- What's in `x` after both threads are done?

- Maybe 10, but maybe 5 or 7.
- Why? addition may not be *atomic*.
- When one thread starts executing `x += my_y`, another thread can take actions that overlap the addition and change its result.
- Typical implementation of `x += my_y`:

```
Load x from memory into reg1
Load my_y from memory into reg2
Add reg2 to reg1
Store reg1 in x
```

- (Note: each thread has its own, *private*, registers.)

Suppose threads execute:

```
Time      Thread A                  Thread B
 0:   Load x=2 into reg1      Finish computing my_y
 1:   Load my_y=3 into reg2   Load x=2 into reg1
 2:   Add reg2 to reg1        Load my_y=5 into reg2
 3.   Store reg1 in x         Add reg2 to reg1
 4.   ...                     Store reg1 in x
```

What's in x when threads are done?

- Nvidia GPUs have large block of memory (global memory) shared among *all* the threads.
- Nvidia GPUs also have a small block of memory (shared memory) shared among threads in block.
- Problem: control access to shared data structures so no race conditions?

- Several different solutions.
- Example. Compute a dot product of two vectors:

```
float Dot(float x[], float y[], int n) {
   float dot = 0;
   for (i = 0; i < n; i++)
      dot += x[i]*y[i];
   return dot;
}
```

- As in the vector addition one thread for each component of `x, y`.
- Thread *i*: compute `x[i]*y[i]`
- Problem: how to add thread's product into total?

CUDA Atomic Functions

- To control access to dot product, can use CUDA "atomic functions".
- When threads call CUDA's `atomicAdd()` function, CUDA runtime system schedules the executions, so only one thread executes addition at a time.
- Code very similar to vector addition.
- Host code allocates and initializes arrays on host, device.
- Host calls kernel
- Kernel stores result in "one-element array."
- Host copies result from device to host.

Kernel:

```
__global__ void Dev_dot(float x[], float y[], int n,
      float* dot_p) {
   int threads_per_block = blockDim.x;
   int my_block = blockIdx.x;
   int my_thread = threadIdx.x
   int i = threads_per_block*my_block + my_thread;

   if (i < n) {
      float tmp = x[i]*y[i];
      atomicAdd(dot_p, tmp);
   }
}  /* Dev_dot */
```

Performance:

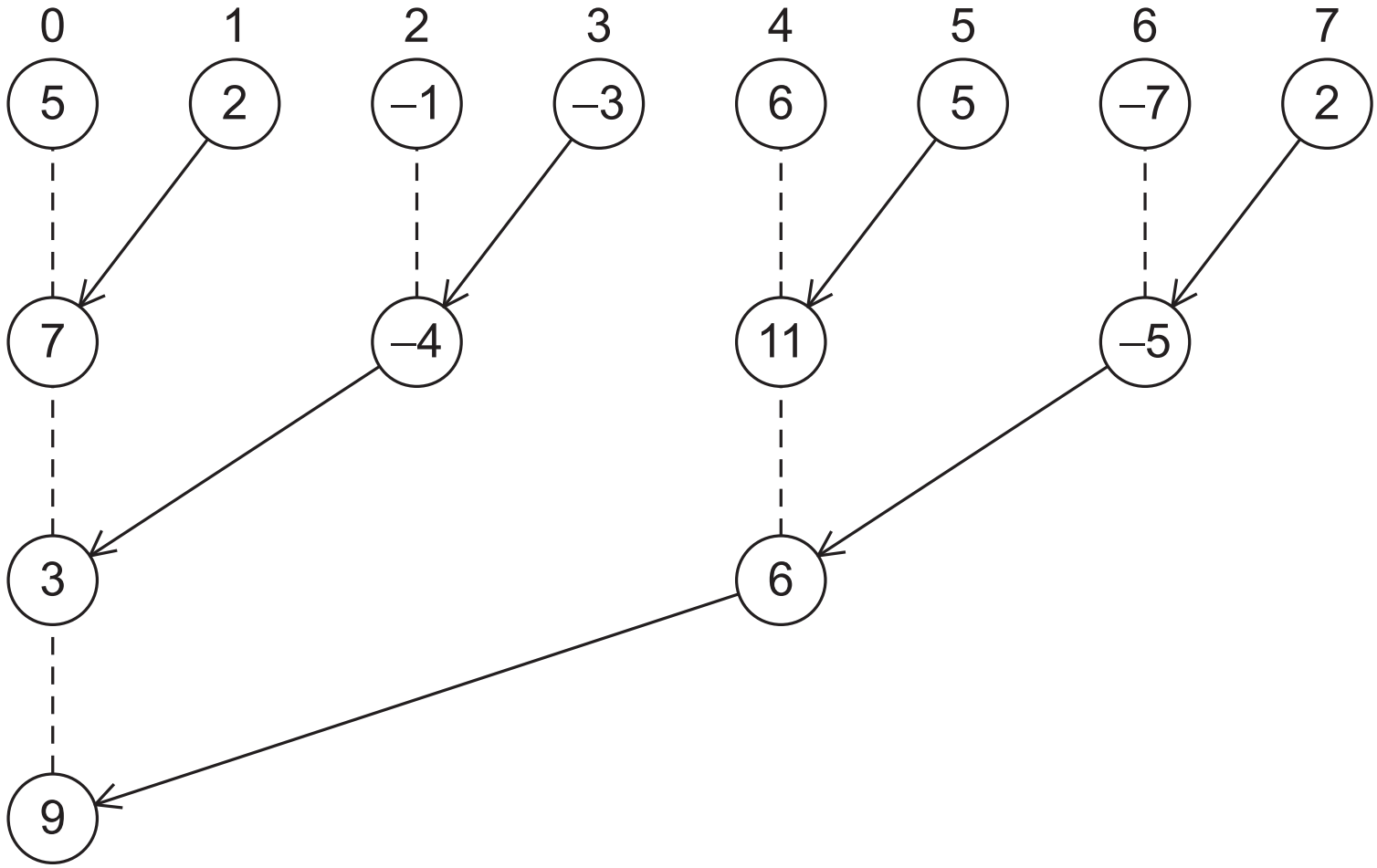
```
n = 1048576, block count = 2048, threads
    per block = 512
Serial:  Intel Xeon E5-2609, 2.40GHz

Serial: 4.05e-03 seconds
CUDA:   1.98e-02 seconds
```

- Serial just under 5 times faster ...
- Not surprising: `atomicAdd` probably serializes access to the dot product.

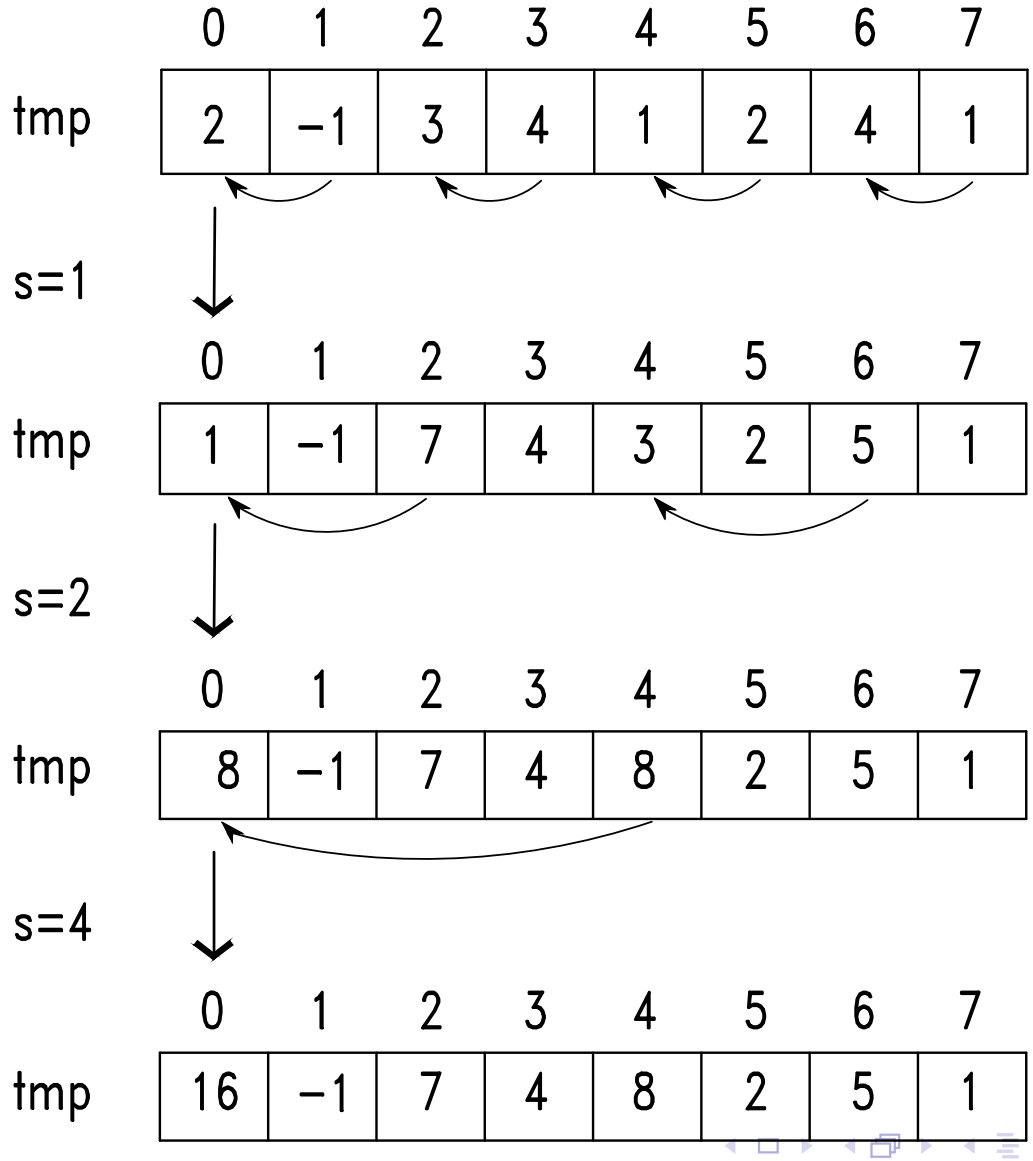
Tree-Structured Global Sum

- OK. Back to the drawing board.
- Example: 8 students, each computes a value.
- How can we add them up?
- Give all numbers to one student and that student does all addition.
- Better alternative: *tree structured* global sum.

Students are 0, 1, 2, . . . , 7



- With shared memory, don't have to send one "student's" result to another: receiver can take result.
- Suppose each thread's product `x[i]*y[i]` stored in a temporary, `tmp[i]`.


0 1 2 3 4 5 6 7
tmp
2 −1 3 4 1 2 4 1
s=1
0 1 2 3 4 5 6 7
tmp
1 −1 7 4 3 2 5 1
s=2
0 1 2 3 4 5 6 7
tmp
8 −1 7 4 8 2 5 1
s=4
0 1 2 3 4 5 6 7
tmp
16 −1 7 4 8 2 5 1

Thread *i* will do something like this:

```
tmp[i] = x[i]*y[i];

for (s = 1; s < thread_count;  s *= 2)
   if (i % (2*s) == 0)
      tmp[i] += tmp[i+s];
```

- Issues?
- Race condition(s)?

- Yes.

```
tmp[i] = x[i]*y[i];

for (s = 1; s < thread_count;  s *= 2)
   if (i % (2*s) == 0)
      tmp[i] += tmp[i+s];
```

- `tmp[i] += tmp[i+1]` can be executed before `tmp[i+1] = x[i+1]*y[i+1];`
- `tmp[i] += tmp[i+s]` in one iteration can be executed before `tmp[i+s] += tmp[(i+s) + s/2]` computed in an earlier iteration.

- Need to synchronize threads with *barrier:* point in code that cannot be passed by any thread, until *all* threads have reached it.
- When thread reaches barrier, it *waits* until all threads have reached barrier before proceeding.
- Good news! CUDA has *very* fast barrier `__syncthreads()`

- Bad news! `__syncthreads()` is barrier *across thread block.*
- Only synchronizes threads in one thread block.
- Threads in different blocks aren't synchronized
- Threads in different blocks can only be synchronized by returning from kernel and either starting new kernel or calling `cudaThreadSynchronize()`.
- So *within block* can do tree-structured "global" sum.
- Then host can add up results computed by individual blocks.

So if z[block] stores results on each block, we have

```
tmp[loc_i] = x[i]*y[i];
__syncthreads();

for (s = 1; s < block_size;  s *= 2) {
   if (loc_i % (2*s) == 0)
      tmp[loc_i] += tmp[loc_i+s];
   __syncthreads();  // Synchronize *all* threads
                     // in block
}

if (loc_i == 0) z[block] = tmp[0];
```

What happens if __syncthreads is called inside if statement?

Performance:

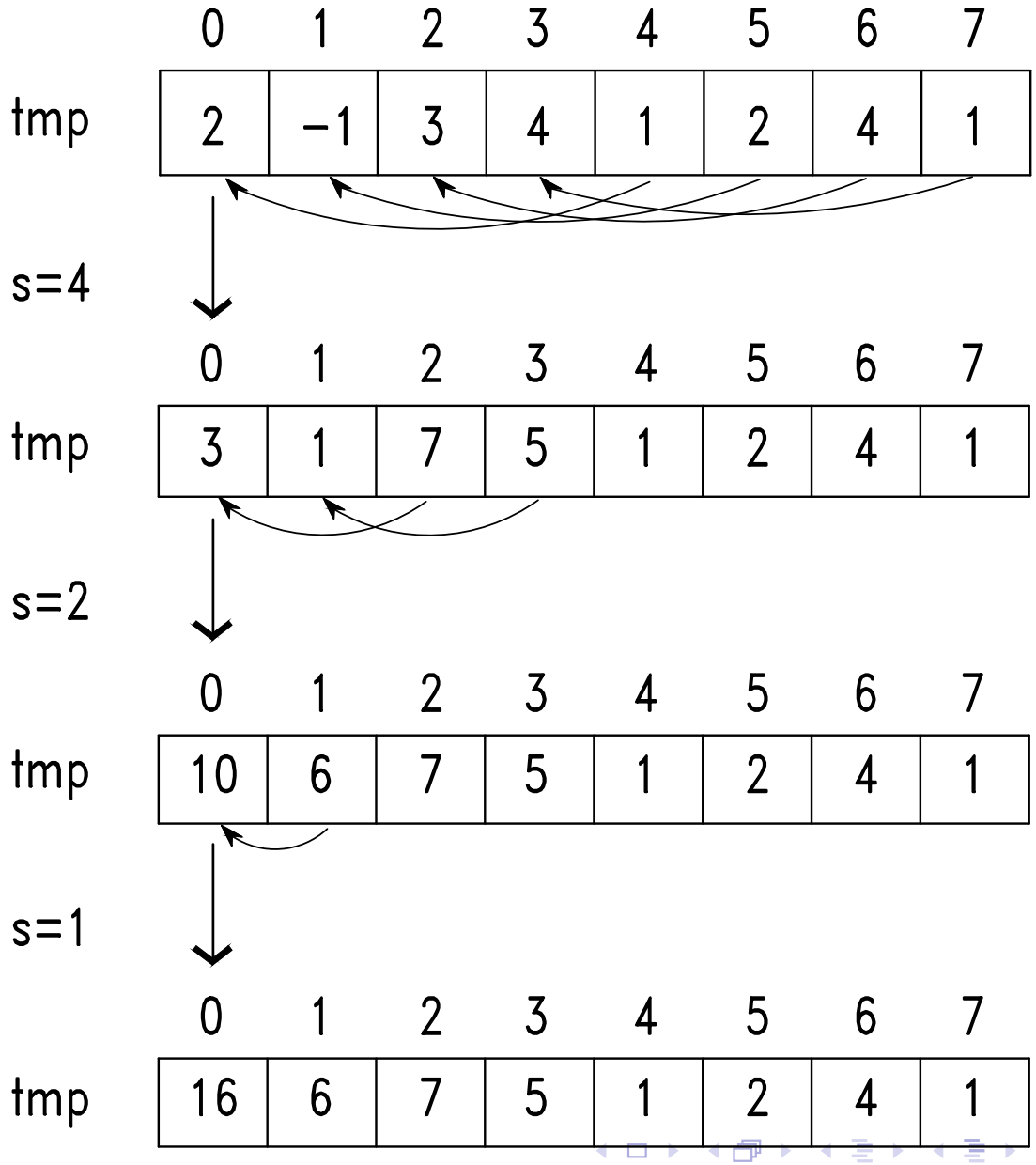
```
n = 1048576, block count = 2048, threads
   per block = 512
Serial:  Intel Xeon E5-2609, 2.40GHz

Serial:              4.03e-03 seconds

CUDA
atomicAdd:           1.98e-02 seconds
Tree structure:      4.00e-03 seconds
```

Much faster than atomicAdd. Roughly same as CPU ...

Alternative Tree Structure

- Possible problem with dot product: thread divergence.
- Earlier: threads in block execute SIMD fashion
- Conditional branch can require considerably more time when threads execute different branches.
- The truth: threads in a *warp* execute SIMD fashion.
- Currently warp has 32 threads.
- Warp is formed from consecutive threads within a block.
- Example: Block has 64 threads. Composed of two warps: threads 0 – 31, and threads 32 – 63.

- Recall tree structure
- If warp size $= 4$, *every* iteration results in thread divergence.
- Structure of tree isn't carved in stone ...
- In alternative structure, if warp size $= 4$, first iteration doesn't result in divergence.


0 1 2 3 4 5 6 7
tmp
2 −1 3 4 1 2 4 1
s=4
0 1 2 3 4 5 6 7
tmp
3 1 7 5 1 2 4 1
s=2
0 1 2 3 4 5 6 7
tmp
10 6 7 5 1 2 4 1
s=1
0 1 2 3 4 5 6 7
tmp
16 6 7 5 1 2 4 1

Code for alternative structure:

```
loc_i = thread rank in block;
i = global thread rank;
tmp[loc_i] = x[i]*y[i];
__syncthreads();

for (s = blocksize/2; s > 0;  s /= 2) {
   if (loc_i < s)
      tmp[loc_i] += tmp[loc_i+s];
   __syncthreads();
}
if (loc_i == 0) z[block] = tmp[0];
```

- Pop Quiz: block size $= 512$, warp size $= 32$
- How many iterations of first tree structure result in divergence?
- How many iterations of second tree structure result in divergence?

- 9 of 9 iterations result in divergence using first structure
- 5 of 9 iterations result in divergence using second structure

Performance:

```
n = 1048576, block count = 2048, threads
   per block = 512
Serial:  Intel Xeon E5-2609, 2.40GHz

Serial:                 4.03e-03 seconds

CUDA
atomicAdd:              1.98e-02 seconds
Tree structure 1:       4.00e-03 seconds
Tree structure 2:       1.94e-03 seconds
```

More than 10 times faster than `atomicAdd` and more than twice as fast as CPU and original tree structure.

Unrolling the Loop

- Within warp, threads operate in lockstep.
- No need to synchronize the threads when they're operating within one warp:
- This will happen when adding the last 64 elements in list and only using threads 0–31 of the block.

Suggests: try replacing

```
for (int s = blocksize/2; s > 0; s /= 2) {
    if (loc_t < s)
        tmp[loc_i] += tmp[loc_i + s];
    __syncthreads();
}
```

With this code:

```
for (int s = blocksize/2; s > 32; s /= 2) {
   if (loc_i < s)
      tmp[loc_i] += tmp[loc_i + s];
   __syncthreads();
}

if (loc_i < 32) {
   if (block_size >= 64) tmp[loc_i] += tmp[loc_i + 32];
   if (block_size >= 32) tmp[loc_i] += tmp[loc_i + 16];
   if (block_size >= 16) tmp[loc_i] += tmp[loc_i + 8];
   if (block_size >= 8) tmp[loc_i] += tmp[loc_i + 4];
   if (block_size >= 4) tmp[loc_i] += tmp[loc_i + 2];
   if (block_size >= 2) tmp[loc_i] += tmp[loc_i + 1];
}
```

Does extra work, but no thread divergence, and result in `tmp[0]` is correct.

Performance:

```
n = 1048576, block count = 2048, threads
    per block = 512
Serial:  Intel Xeon E5-2609, 2.40GHz

Serial:                 4.03e-03 seconds

CUDA
atomicAdd:              1.98e-02 seconds
Tree structure 1:       4.00e-03 seconds
Tree structure 2:       1.94e-03 seconds
Unrolled loop:          1.38e-03 seconds
```

- Thanks to George Ledin and the Computer Science Department at Sonoma State for inviting me.
- Thanks to Karen Parish for preparing the diagrams.
- I'll post the talk and the codes on my webpage

  `http://cs.usfca.edu/~peter/ssu_talk`
- Questions?