CIS 431/531
Intro to Parallel Computing

CUDA

CPU vs. GPU

CPU

- Instruction-level parallelism (ILP)
  - few "brawny" cores
  - general-purpose computing
- Reduce latency
  - large, complex memory hierarchy
  - hardware prefetching

GPU

- Data-level parallelism (DLP)
  - many "wimpy" cores
  - high peak performance
- Increase throughput
  - large number of hardware threads
  - user-managed "scratchpad" cache

Example

Naïve

```
for(i = 0; i < N; i++)
{
    A[i] += 2;
}
```

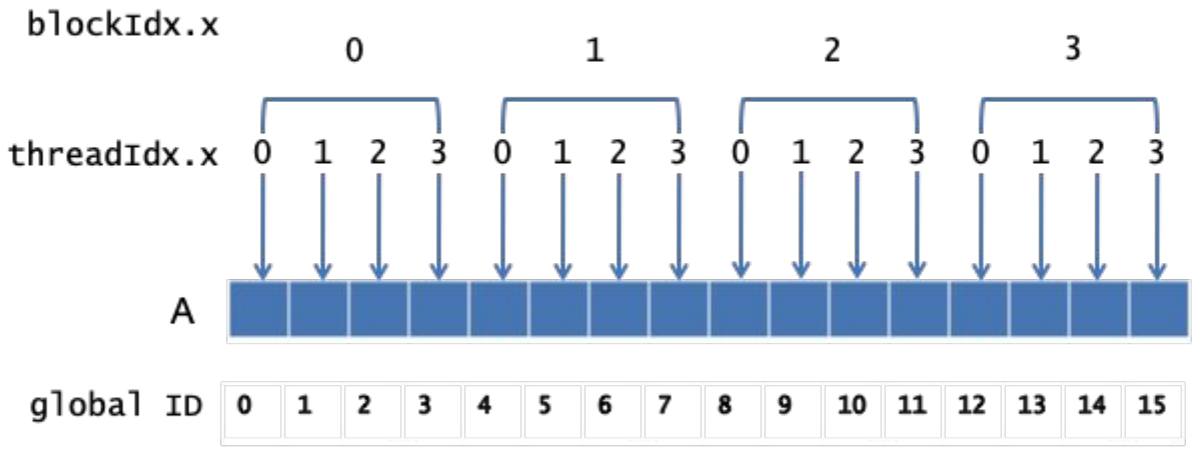
OpenMP

```
#pragma omp parallel for
for(i = 0; i < N; i++) {
    A[i] += 2;
}
```

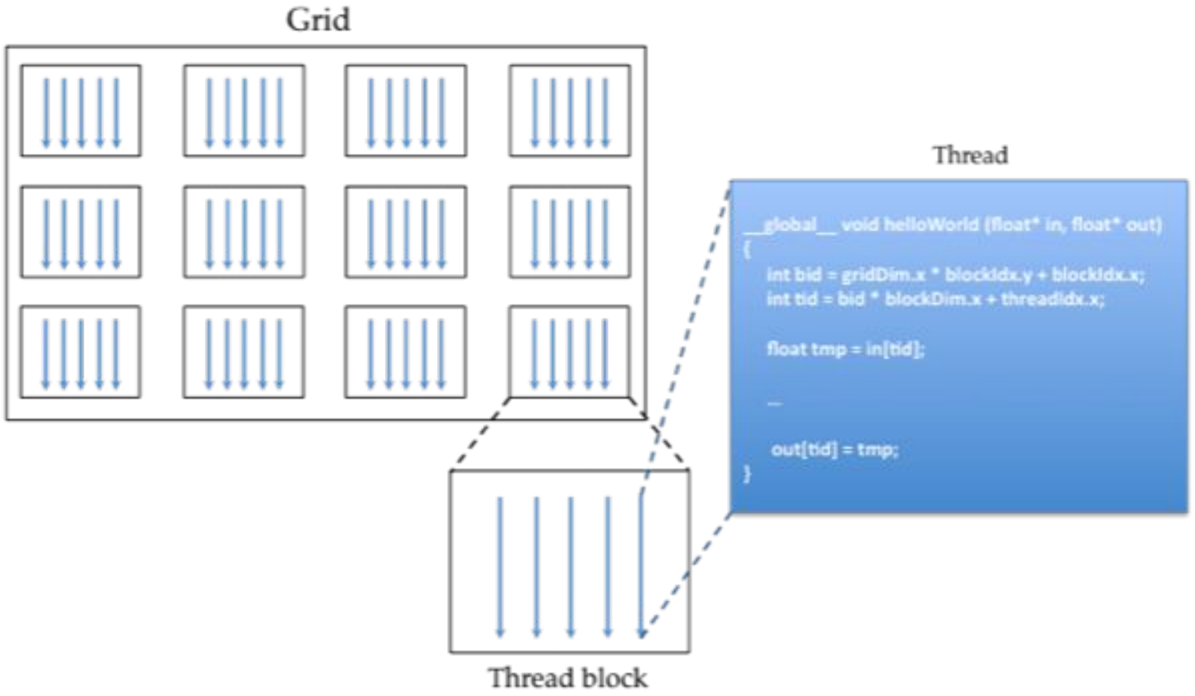
CUDA

```
int threadID = blockIdx.x * blockDim.x + threadIdx.x
A[threadID] += 2;
```

Mapping Threads



Thread Hierarchy



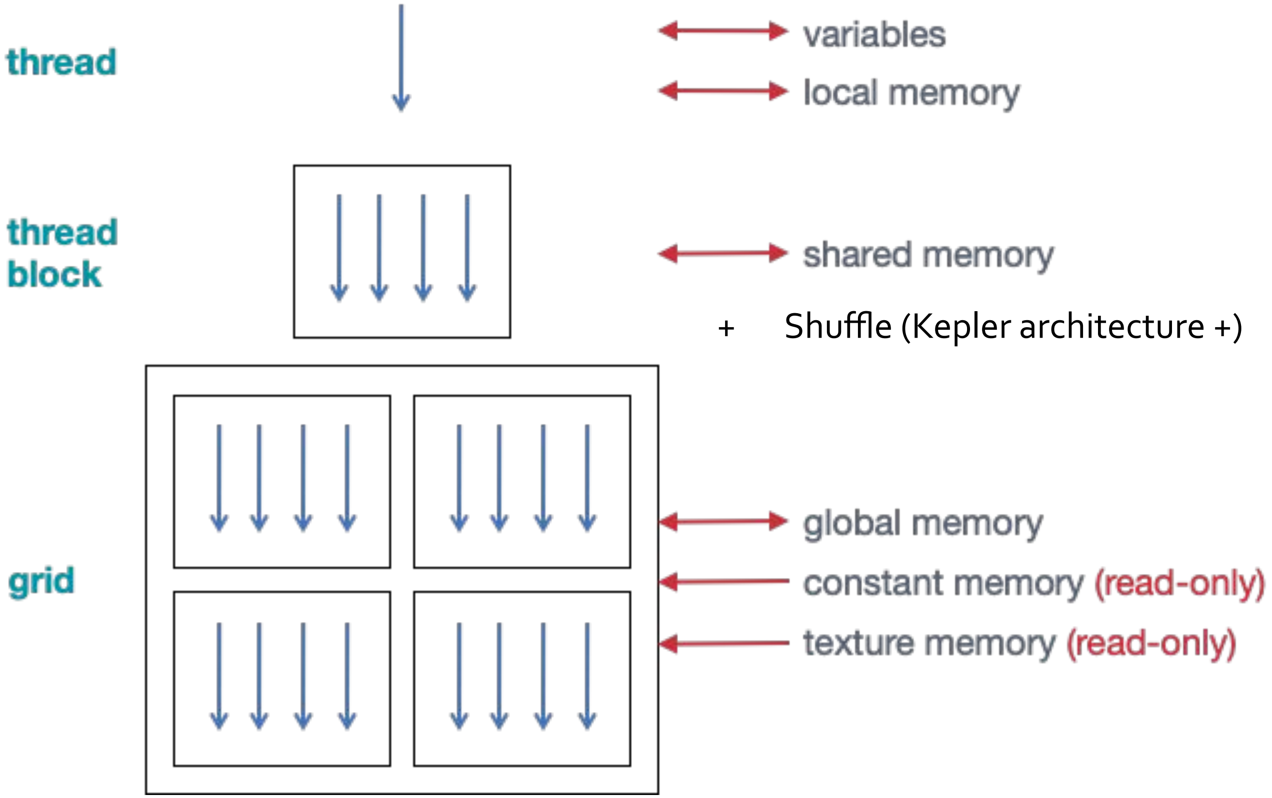
```
__global__ void helloWorld (float* in, float* out)
{
   int bid = gridDim.x * blockIdx.y + blockIdx.x;
   int tid = bid * blockDim.x + threadIdx.x;

   float tmp = in[tid];

   …

   out[tid] = tmp;
}
```

Memory Hierarchy



- Shuffle (Kepler architecture +)

Performance Notes

Reading from the DRAM occurs at the granularity of 128 Byte transactions

- Requests are further decomposed to aligned cache lines
- L1 constant: 64 Bytes (Volta)
- L1 data: 32 Bytes (Volta)
- L2 cache: 64 Bytes (Volta)

Minimize loading redundant cache lines to maximize bandwidth utilization

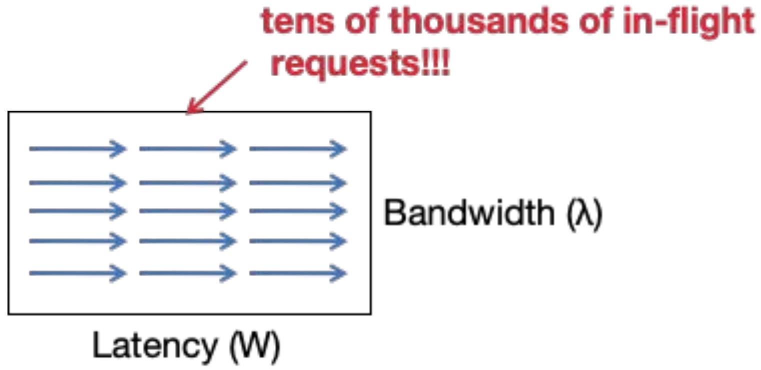
- Aligned access to memory
- Sequential access pattern

Performance Notes

Little's Law

- L = λW
- L = average number of customers in a store
- λ = arrival rate
- W = average time spent

Memory Bandwidth



Questions?

nvcc

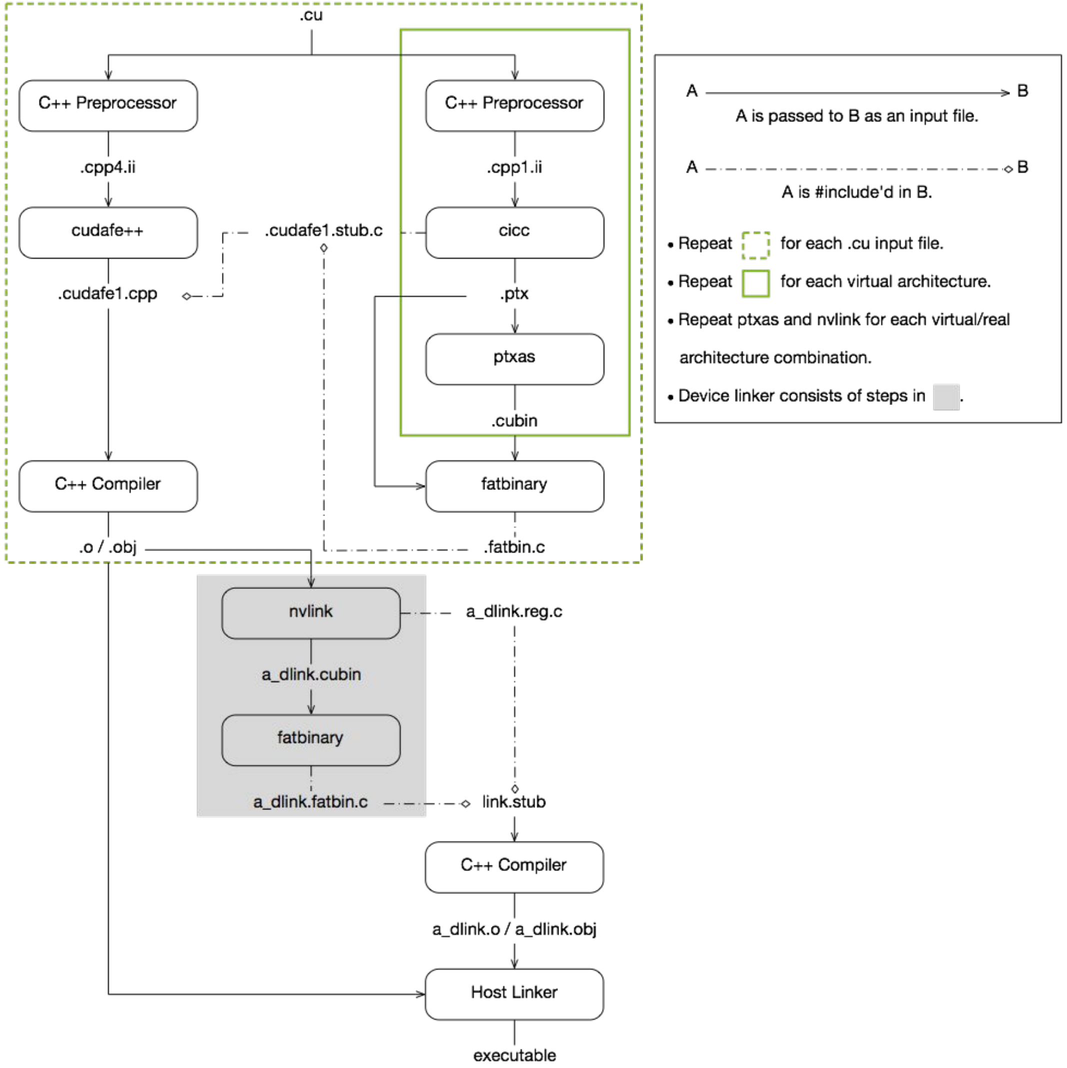
nvcc is a CUDA-C compiler that splits your code into

- Host code - forwarded to gcc/g++
- Device code - forwarded to Nvidia device compiler (nvcc)

nvcc links together the host code and device code into one executable

Convention - CUDA code usually has the `.cu` extension

nvcc


.cu
C++ Preprocessor
C++ Preprocessor
.cpp4.ii
.cpp1.ii
cudafe++
.cudafe1.stub.c
cicc
.cudafe1.cpp
.ptx
ptxas
.cubin
C++ Compiler
fatbinary
.o / .obj
.fatbin.c
nvlink
a_dlink.reg.c
a_dlink.cubin
fatbinary
a_dlink.fatbin.c
link.stub
C++ Compiler
a_dlink.o / a_dlink.obj
Host Linker
executable
A → B
A is passed to B as an input file.
A → B
A is #include'd in B.
• Repeat for each .cu input file.
• Repeat for each virtual architecture.
• Repeat ptxas and nvlink for each virtual/real architecture combination.
• Device linker consists of steps in .

nvcc

PTX (CUDA IR)

- Some instructions are only supported on specific architectures (e.g., shuffle)
- Can be compiled to binary for a newer devices

CUBIN (CUDA Binary)

- ELF formatted binary file
- Typically Embedded into host code by nvcc
- Can also be generated by using the `-cubin` option
- Binary code is specific to a particular architecture

Just-in-time (JIT) compilation

- Any PTX code loaded by an application at runtime is compiled further to binary by the device driver
- Increases application load time
- Allows application to benefit from new compiler improvements specific to new device driver
- Only way to compile code for a device that does not exist at the time of writing the code

Tools

cuobjdump

- Extracts information from CUDA binary files (both standalone and those embedded in host binaries) and presents them in human readable format.
- Includes CUDA assembly code for each kernel, CUDA ELF section headers, string tables, relocators and other CUDA specific sections. It also extracts embedded ptx text from host binaries.

nvidisasm

- Extracts information from standalone cubin files and presents them in human readable format.
- The output includes CUDA assembly code for each kernel, listing of ELF data sections and other CUDA specific sections.
- Also does control flow analysis to annotate jump/branch targets and makes the output easier to read.

CUDA Runtime

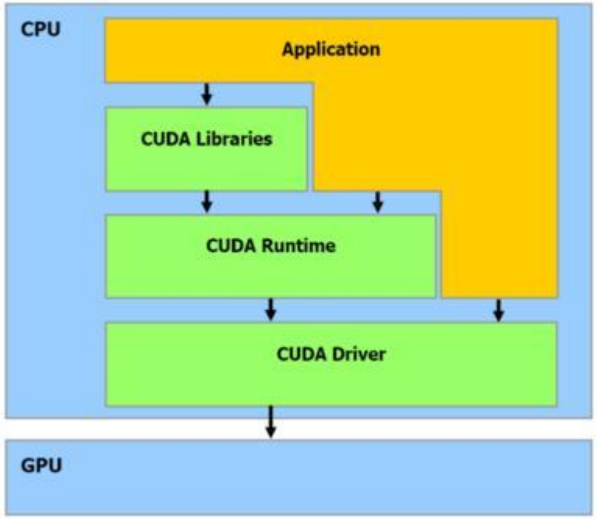
CUDA is composed of two APIs

- Runtime API
- Driver API

Runtime API is implemented on top of the Driver API and eases device code management.

They are mutually exclusive - you can use one or the other, but must use one of them

You will most likely use the Runtime to program CUDA (for now)



Compute Capabilities

General specification and features of a device depends on its compute capability

When compiling code, you must get the compute capability correct for both CUBIN and PTX, otherwise, it may not run (correctly).

Currently up to 9.0

CUDA by Example

```
/usr/local/cuda/bin/nvcc -c bandwidthTest.cu
 -o bandwidthTest.o  -gencode arch=compute_70,code=sm_70
--default-stream per-thread -I/usr/local/cuda/include
-I/usr/local/cuda/samples/common/inc

/usr/bin/g++ -Wall -g  -O3 -fopenmp bandwidthTest.o -o
btest  -L/usr/local/cuda/lib64 -lcuda -lcudart -lcublas
-lcurand -lcusolver
```

CUDA by Example - Data Transfer

```
// Array on the host system
unsigned char* host_array = (unsigned char*) malloc(sizeof(char) * n);
assert(host_array);
memset(host_array, 0xff, n);

// Array on the GPU
unsigned char* device_array;
if(cudaMalloc(&device_array, n) != cudaSuccess) {
    fprintf(stderr, "Error: cudaMalloc at line %d in function %s\n",
            (__LINE__), (__func__));
}

cudaMemcpy(device_array, host_array, n, cudaMemcpyHostToDevice);
```

CUDA by Example - Data Transfer

```
// Timers
cudaEvent_t start, stop;
float tt;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);
cudaMemcpy(device_array, host_array, n, cudaMemcpyHostToDevice);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&tt, start, stop);
fprintf(stdout, "Time for H->D transfer: %g (ms)\n", tt);
fprintf(stdout, "Bandwidth performance: %g\n", (n / tt)/1e6);
```

CUDA by Example - Data Transfer

```
PCIe/NVLink Test
Time for H->D transfer: 73.4372 (ms)
Bandwidth performance: 13.6171
```

CUDA by Example - Data Transfer

This is on a NVLink 2.0 system - up to 75 GB for host to device

Why?

CUDA by Example - Data Transfer

This is on a NVLink 2.0 system - up to 75 GB for host to device

Why?

- **Pinned** memory
- With malloc() GPU is given a virtual memory - for each word of data, CPU has to look up physical address and then copy - **slow**
- Tell the OS to keep a memory at fixed location (i.e., pin). GPU can now directly access the host memory.
- Pinned memory limits OS ability to move data around (i.e., manage the memory) - you will run out of memory much faster (if you're using too much).

CUDA by Example - Data Transfer

```
unsigned char* host_array1;
if(cudaHostAlloc(&host_array1, n, cudaHostAllocPortable)) {
    fprintf(stderr, "Error: cudaHostAlloc at line %d in function %s\n",
            (__LINE__), (__func__));
}
```

CUDA by Example - Data Transfer

```
PCIe/NVLink Test (Pinned)
Time for H->D transfer (Pinned): 15.0076 (ms)
Bandwidth performance (Pinned): 66.6331
```

CUDA by Example - Data Transfer

```
cudaMemcpy(device_array1, device_array, n,cudaMemcpyDeviceToDevice);

Time for D->D transfer: 2.59722 (ms)
Bandwidth performance: 770.055
```

~86% of peak memory bandwidth

CUDA by Example - Data Transfer

```
__global__ void myMemcpy(unsigned char* dst, unsigned char* src, int n)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid < n) {
        dst[tid] = src[tid];
    }
}
```

```
unsigned int tbSize = 256;
unsigned int nTb = (n + tbSize - 1) / tbSize;
myMemcpy<<<nTb, tbSize>>>(device_array1, device_array, n);
```

CUDA by Example - Data Transfer

```
__global__ void myMemcpy(unsigned char* dst, unsigned char* src, int n)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid < n) {
        dst[tid] = src[tid];
    }
}
```

```
unsigned int tbSize = 256;
unsigned int nTb = (n + tbSize - 1) / tbSize;
dim3 dimBlock(tbSize, 1, 1);
dim3 dimGrid(nTb, 1, 1);
myMemcpy<<<dimGrid, dimBlock>>>(device_array1, device_array, n);
```

CUDA by Example - Data Transfer

```
__global__ void myMemcpy(unsigned char* dst, unsigned char* src, int n)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid < n) {
        dst[tid] = src[tid];
    }
}
```

```
Time for D->D transfer (manual): 11.5446 (ms)
Bandwidth performance (manual): 346.483 (GB/s)
```

Less than ½ of what we were getting with cudaMemcpy

CUDA by Example - Data Transfer

```
__global__ void myMemcpy(unsigned char* dst, unsigned char* src, int n)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid < n) {
        dst[tid] = src[tid];
    }
}
```

```
Time for D->D transfer (manual): 11.5446 (ms)
Bandwidth performance (manual): 346.483 (GB/s)
```

```
fprintf(stdout, "Bandwidth performance: %g\n", n( / tt)/1e6);
→ it should be 2 * n
  You are reading n bytes, and then writing n bytes.
```

Free Memory

```
free(host_array); // free for malloc
cudaFreeHost(host_array1); // free for cudaHostAlloc
cudaFree(device_array); // free for cudaMalloc
cudaFree(device_array1);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

CUDA Program Structure

1) Declare memory on GPU to store the data to process
2) Copy data from host to device
3) Call the kernel(s) to process the data
4) Copy result back from device to host
5) Free memory

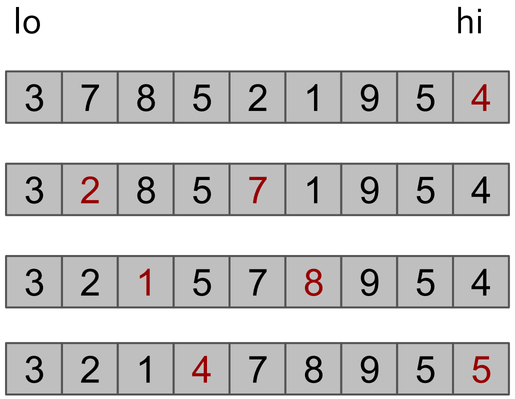
CUDA by Example - Quick Sort

Let’s now consider quicksort on a GPU

Quicksort

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
      return i
```



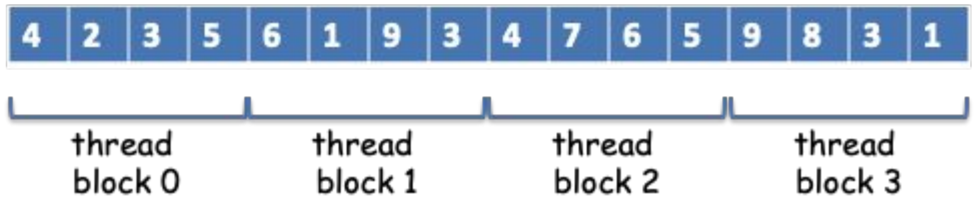
CUDA by Example - Quick Sort

Let's now consider quicksort on a GPU

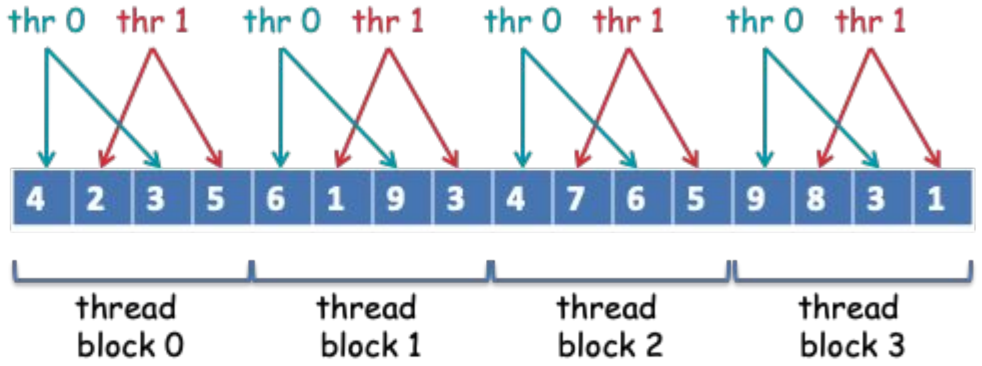
Step 1 Partition the initial list

- How do we partition the list amongst thread blocks?
- Recall that thread blocks CANNOT co-operate and thread blocks can go in ANY order
- However, we need to have MANY threads and thread blocks in order to see good performance

CUDA by Example - Quick Sort

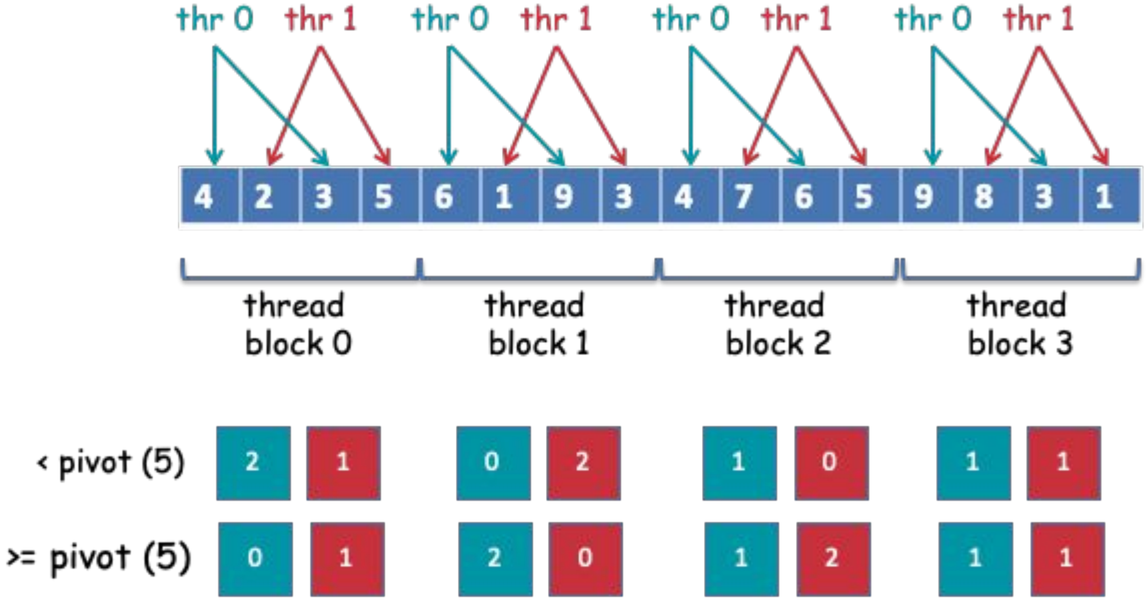


CUDA by Example - Quick Sort



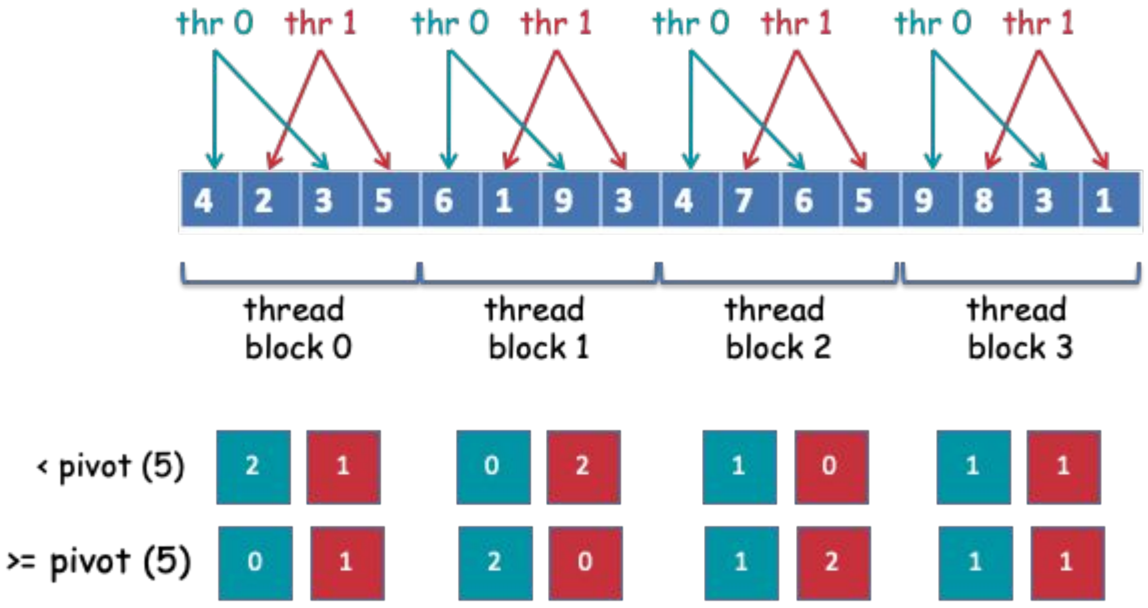
- First each thread block is assigned a chunk of the array, and then the chunk is divided among the threads.
- It would be good to assign consecutive data to consecutive threads - why?

CUDA by Example - Quick Sort



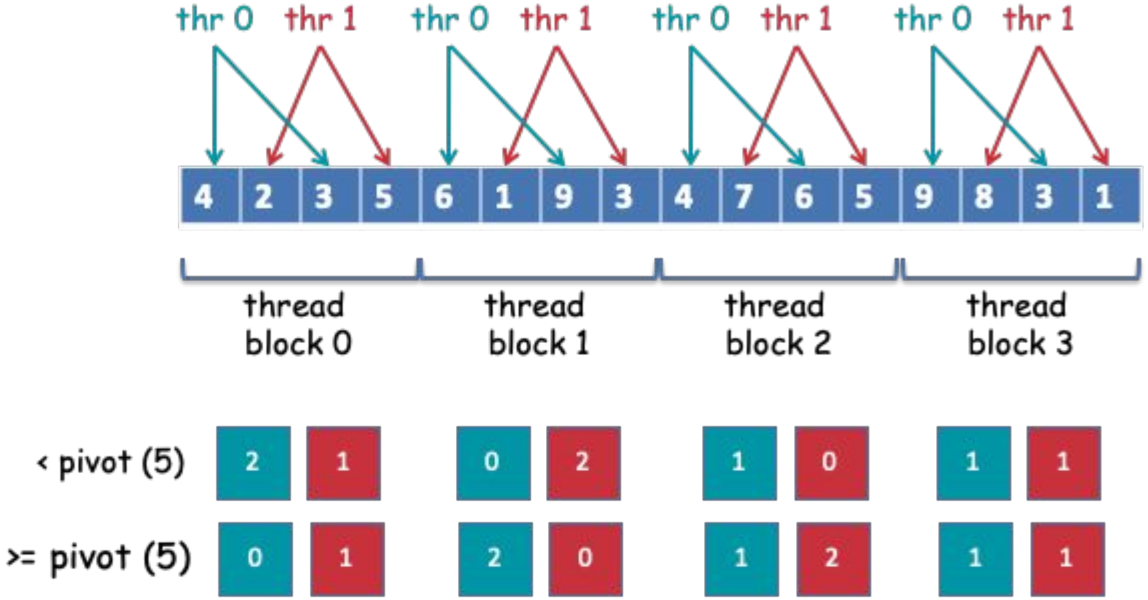
- Each thread counts the number of elements that are below and above the pivot

CUDA by Example - Quick Sort



Do a cumulative sum on < pivot and >= pivot
This should be done in shared memory in parallel

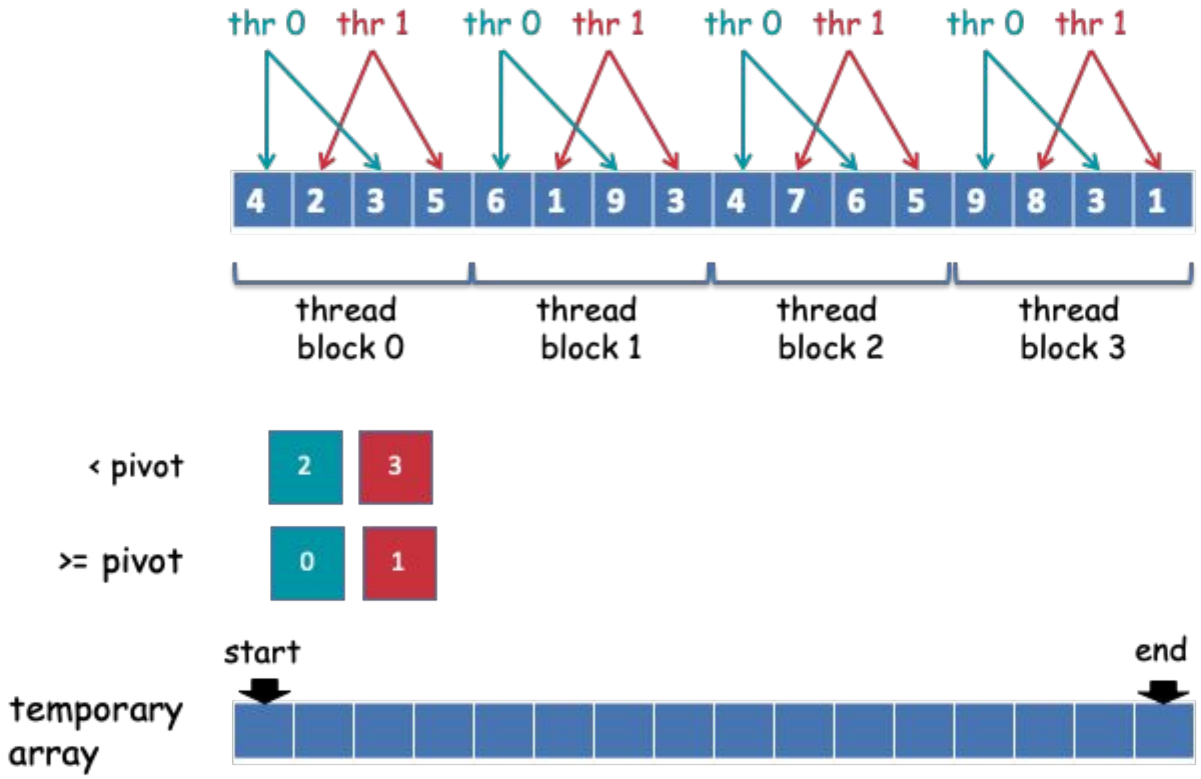
CUDA by Example - Quick Sort



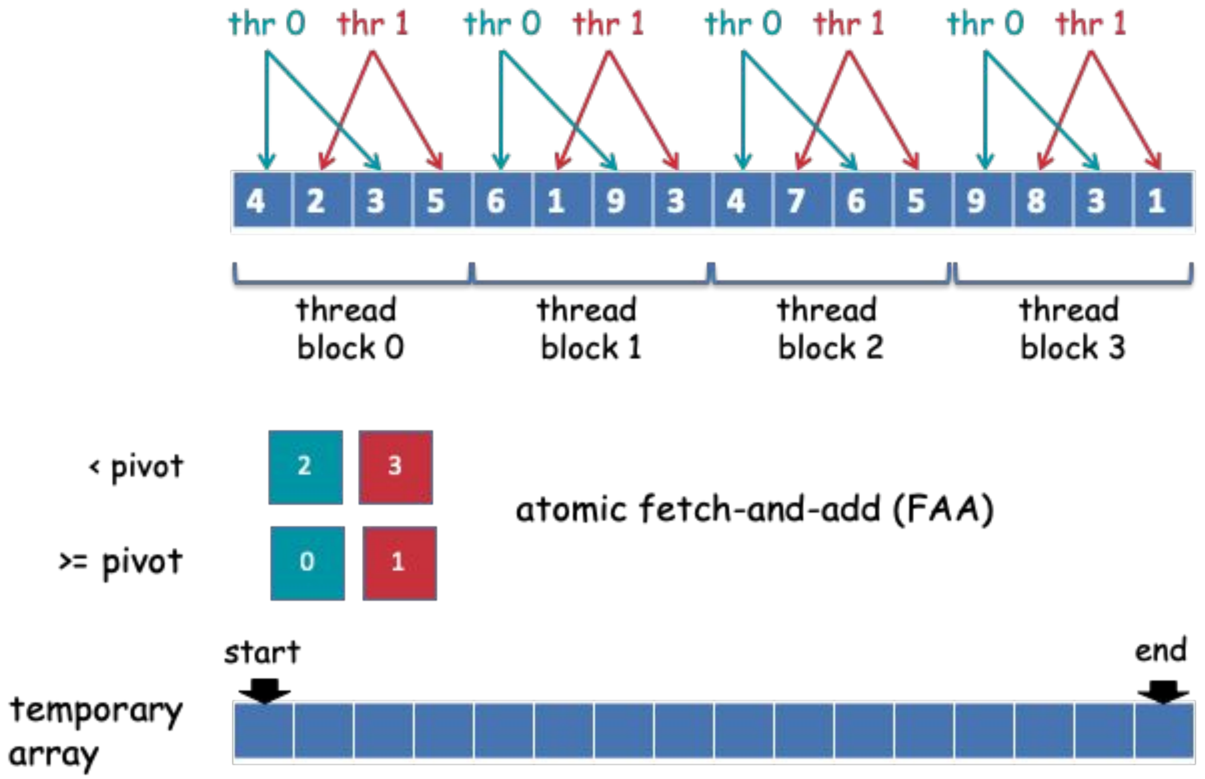
Do a cumulative sum on < pivot and >= pivot
This should be done in shared memory in parallel

This tells us how much space and where each thread block needs to store its values

CUDA by Example - Quick Sort

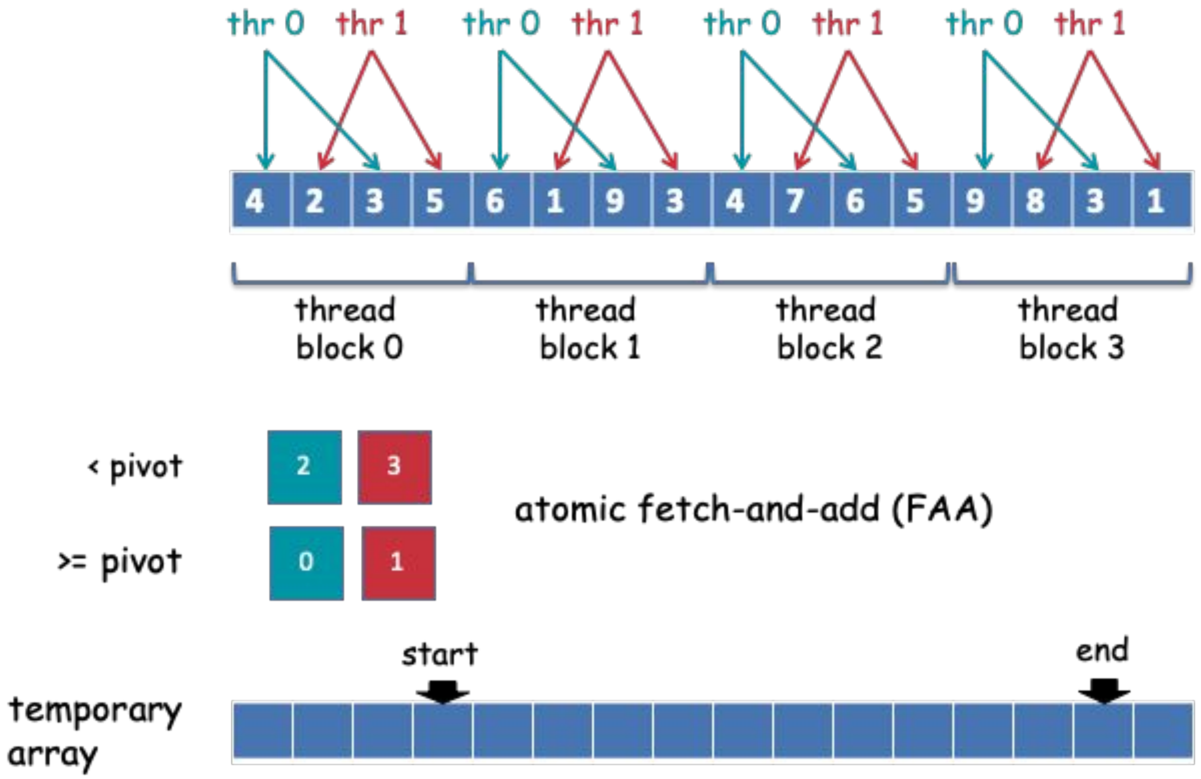


CUDA by Example - Quick Sort



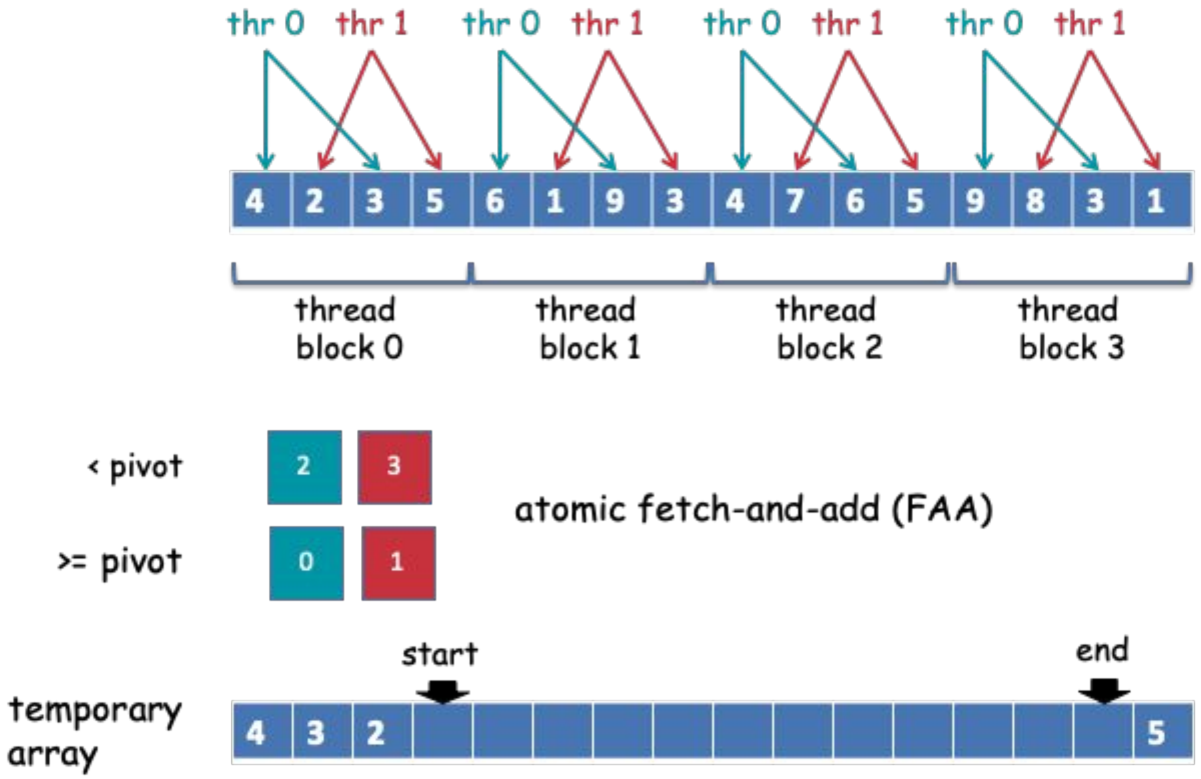
- Without an atomic instruction, different threads might write to the same point

CUDA by Example - Quick Sort



- Now it can safely start writing to the “allocated” space

CUDA by Example - Quick Sort



CUDA by Example - Quick Sort

That was the first part.

- A kernel will be called each for lower and upper half and repeated

This is done until there are enough independent partitions (lower and upper halves) that can be assigned to thread blocks

- Then each thread block will do the same, minus the FAA
- FAA is not needed since each thread block number needs to be sorted within the partition

When sequences become small enough, you can sort it using an alternative sorting algorithm (e.g., bitonic sort), or send it to the CPU to finish off

Reading Recommendations

CUDA Toolkit Documentation (CUDA 101)

http://www.cse.chalmers.se/~tsigas/papers/GPU-Quicksort-jea.pdf

Nvidia Tesla V100 GPU Architecture (Whitepaper)

Dissecting the Volta GPU Architecture via Microbenchmarking (Research paper)

Various Nvidia Tutorials (there are many of them)