CSE 591: GPU Programming

Memories: Global

Klaus Mueller

Computer Science Department

Stony Brook University

Code examples from Shane Cook “CUDA Programming”

A Note on Program Timing

On CPU

- use *get_time()* to obtain *start* and *end* time
- execution time = *end - start*

On GPU

- after executing the kernel
  - **cudaEventRecord**(*kernel_end*, *stream* = 0) records the event
  - **cudaEventSynchronize**(*kernel_end*) synchronizes all devices first
  - **cudaEventElapsedTime**(&*delta*, *kernel_start*, *kernel_end*);
- *kernel_start* and *kernel_end* are of type *cudaEvent_t*
- there can be more than one *stream*
- if called from host must wrap in CUDA_CALL()

- these all use CUDA event management routines

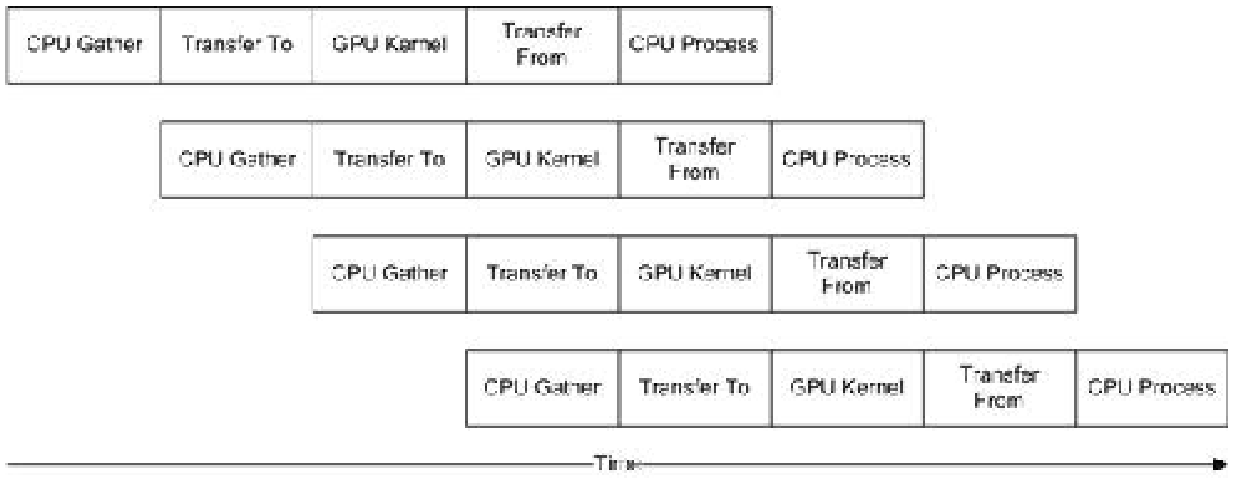
Global Memory: Transfer

How to write to it?

- transfer directly from GPU to GPU (peer to peer, CUDA 4.x SDK)
- explicitly with a blocking transfer
- explicitly with a non-blocking transfer
- implicitly using zero memory copy
- all use the PCI-E bus (8 GB/s)

Transfer can also occur with streams

- overlap transfers and kernels to ensure the GPU is always kept busy



Bandwidth Considerations

GPU compute power on the order of T flops

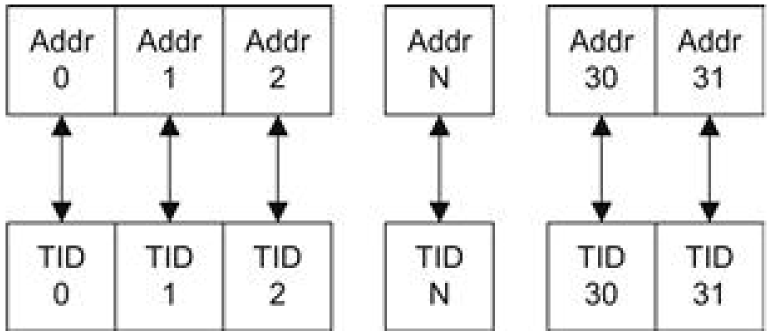
- main memory bandwidth is 190 GB/s or even less: 25 GB/s

Recall difference between latency and bandwidth

- a ratio of 10:1 of threads to number of memory accesses can hide memory latency
- but still need to do access global memory in a coalesced fashion

What is coalescing?

- all threads access a contiguous and aligned memory block
- occurs on a warp-bases (half-warp on G80 hardware)
- accessing floats will get 32 x 4 = 128 bytes



Coalescing

Supported sizes are 32, 64, 128 bytes

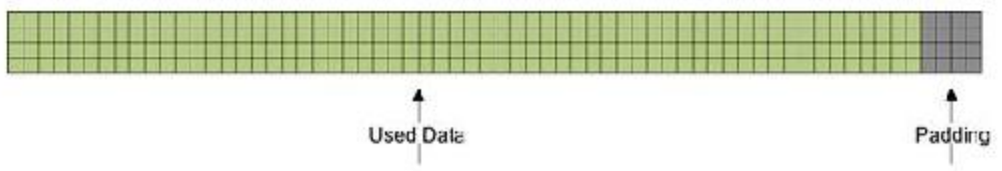
- bytes, 16 and 32 bit data are supported
- must be aligned with 32-byte boundary

Alignment is important

- example: 2D array 100 x 60 floats
- *cudaMalloc()* would allocate 100 x 60 x 4 = 24,000 bytes
- length of a single row would be 240 bytes → not aligned in 32 bytes
- thus an access of element [1][0] would not be coalesced and incur severe delays

How to align?

- use special memory allocation function *cudaMallocPitch()*
- pads the memory for alignment (here, by 4 floats for 256 bytes)

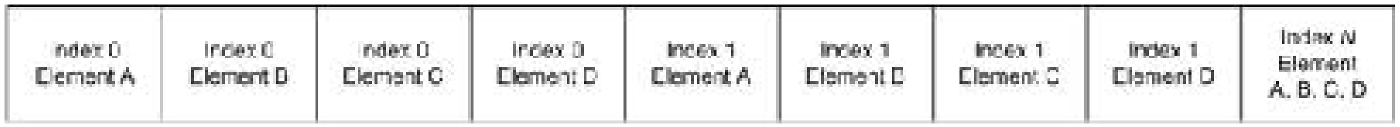


Example

Assume you have a structure

```
typedef struct
{
 unsigned int a;
 unsigned int b;
 unsigned int c;
 unsigned int d;
} MY_TYPE_T;


MY_TYPE_T some_array[1024]; /* 1024 * 4 bytes = 4K */
```

- then data would be stored as follows

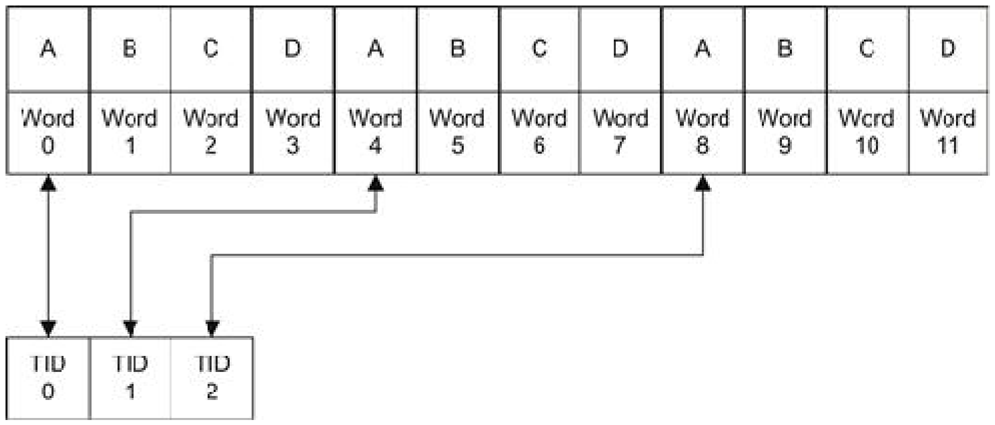


Index 0 Element A	Index 0 Element B	Index 0 Element C	Index 0 Element D	Index 1 Element A	Index 1 Element B	Index 1 Element C	Index 1 Element D	Index N Element A, B, C, D

Access Pattern

Completely un-coalesced

- addresses are not contiguous in memory
- will lead to severe access delays



Two Alternatives

Interleaved vs. non-interleaved

```
// Define the number of elements we'll use
#define NUM_ELEMENTS 4096

// Define an interleaved type
// 16 bytes, 4 bytes per member
typedef struct
{
 u32 a;
 u32 b;
 u32 c;
 u32 d;
} INTERLEAVED_T;
```

```
// Define an array type based on the interleaved structure
typedef INTERLEAVED_T INTERLEAVED_ARRAY_T[NUM_ELEMENTS];

// Alternative - structure of arrays
typedef u32 ARRAY_MEMBER_T[NUM_ELEMENTS];

typedef struct
{
 ARRAY_MEMBER_T a;
 ARRAY_MEMBER_T b;
 ARRAY_MEMBER_T c;
 ARRAY_MEMBER_T d;
} NON_INTERLEAVED_T;
```

CPU Code: Interleaved

```
__host__ float add_test_non_interleaved_cpu(
 NON_INTERLEAVED_T * const host_dest_ptr,
 const NON_INTERLEAVED_T * const host_src_ptr,
 const u32 iter,
 const u32 num_elements)
{
 float start_time = get_time();

 for (u32 tid = 0; tid < num_elements; tid++)
 {
  for (u32 i=0; i<iter; i++)
  {
   host_dest_ptr->a[tid] += host_src_ptr->a[tid];
   host_dest_ptr->b[tid] += host_src_ptr->b[tid];
   host_dest_ptr->c[tid] += host_src_ptr->c[tid];
   host_dest_ptr->d[tid] += host_src_ptr->d[tid];
  }
 }

 const float delta = get_time() - start_time;

 return delta;
}
```

CPU Code: Non-Interleaved

```
__host__ float add_test_interleaved_cpu(
 INTERLEAVED_T * const host_dest_ptr,
 const INTERLEAVED_T * const host_src_ptr,
 const u32 iter,
 const u32 num_elements)
{
 float start_time = get_time();

 for (u32 tid = 0; tid < num_elements; tid++)
 {
  for (u32 i=0; i<iter; i++)
  {
   host_dest_ptr[tid].a += host_src_ptr[tid].a;
   host_dest_ptr[tid].b += host_src_ptr[tid].b;
   host_dest_ptr[tid].c += host_src_ptr[tid].c;
   host_dest_ptr[tid].d += host_src_ptr[tid].d;
   }
  }

  const float delta = get_time() - start_time;
  return delta;
 }
```

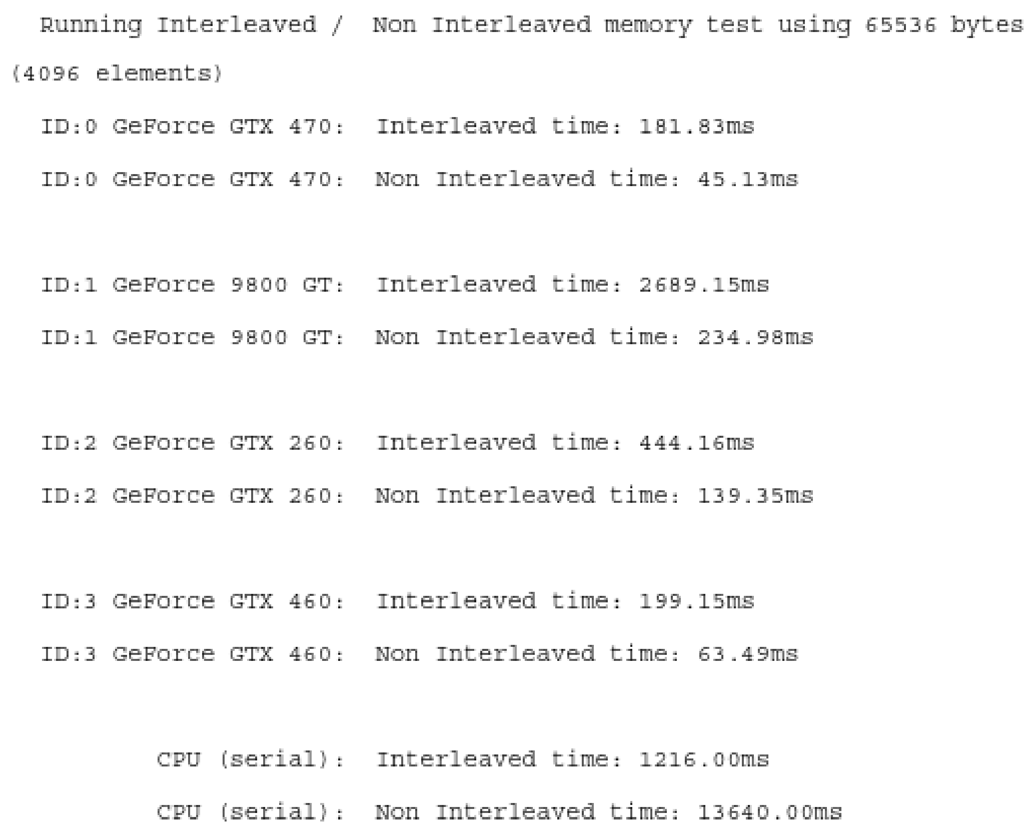
GPU Code: Interleaved

```
__global__ void add_kernel_interleaved(
 INTERLEAVED_T * const dest_ptr,
 const INTERLEAVED_T * const src_ptr,
 const u32 iter,
 const u32 num_elements)
{
 const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;
   if (tid < num_elements)
   {
    for (u32 i=0; i<iter; i++)
    {
     dest_ptr[tid].a += src_ptr[tid].a;
     dest_ptr[tid].b += src_ptr[tid].b;
     dest_ptr[tid].c += src_ptr[tid].c;
     dest_ptr[tid].d += src_ptr[tid].d;
    }
   }
  }
```

GPU Code: Non-Interleaved

```
__global__ void add_kernel_non_interleaved(
 NON_INTERLEAVED_T * const dest_ptr,
 const NON_INTERLEAVED_T * const src_ptr,
 const u32 iter,
 const u32 num_elements)
{
 const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;

 if (tid < num_elements)
 {
  for (u32 i=0; i<iter; i++)
  {
   dest_ptr->a[tid] += src_ptr->a[tid];
   dest_ptr->b[tid] += src_ptr->b[tid];
   dest_ptr->c[tid] += src_ptr->c[tid];
   dest_ptr->d[tid] += src_ptr->d[tid];
  }
 }
}
```

Timing Results



```
  Running Interleaved /  Non Interleaved memory test using 65536 bytes
(4096 elements)
  ID:0 GeForce GTX 470:  Interleaved time: 181.83ms
  ID:0 GeForce GTX 470:  Non Interleaved time: 45.13ms


  ID:1 GeForce 9800 GT:  Interleaved time: 2689.15ms
  ID:1 GeForce 9800 GT:  Non Interleaved time: 234.98ms


  ID:2 GeForce GTX 260:  Interleaved time: 444.16ms
  ID:2 GeForce GTX 260:  Non Interleaved time: 139.35ms


  ID:3 GeForce GTX 460:  Interleaved time: 199.15ms
  ID:3 GeForce GTX 460:  Non Interleaved time: 63.49ms


          CPU (serial):  Interleaved time: 1216.00ms
          CPU (serial):  Non Interleaved time: 13640.00ms
```

Observations:

- non-interleaved has much better performance
- older GPUs more pronounced since coalescing reqs. more stringent
- conversely, on CPU interleaved scheme much better because it favors sequential access

Score Boarding

Helps with latency hiding

Mechanism

- you request a memory item by a statement *a=arr[0]*
- a memory fetch is initiated
- a local variable *a* is listed as having a pending memory transaction
- unlike in CPUs, no stall occurs → no warp is being switched
- only when *a* is actually needed a warp might get switched

How to use it:

- place memory fetches at the start of the kernel
- hide latencies by in-thread computing
- always try to follow memory fetches by unrelated computing
- this works anywhere in a kernel

Is GPU Radix Sort Coalesced?

Assume global memory storage initially

Sorting?

- yes – each thread marches through its own list

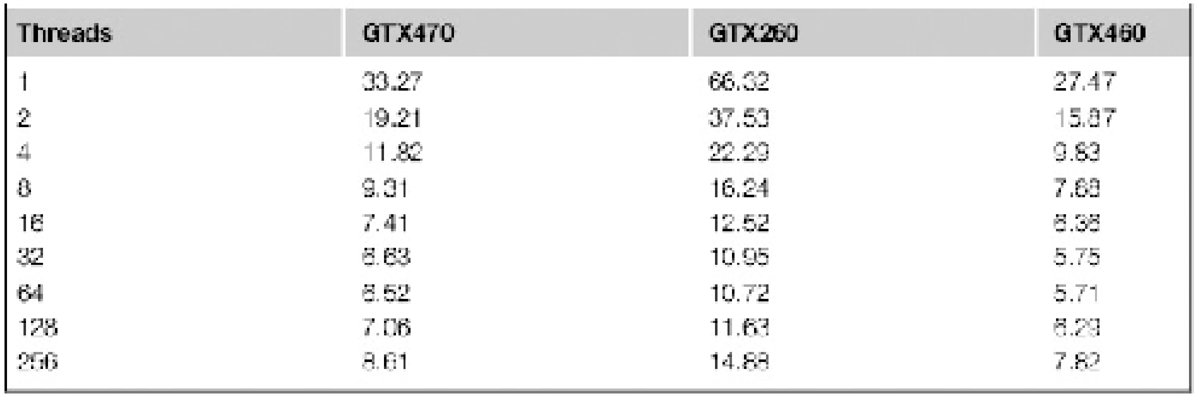
Merging?

- no – the 1-list varies in size

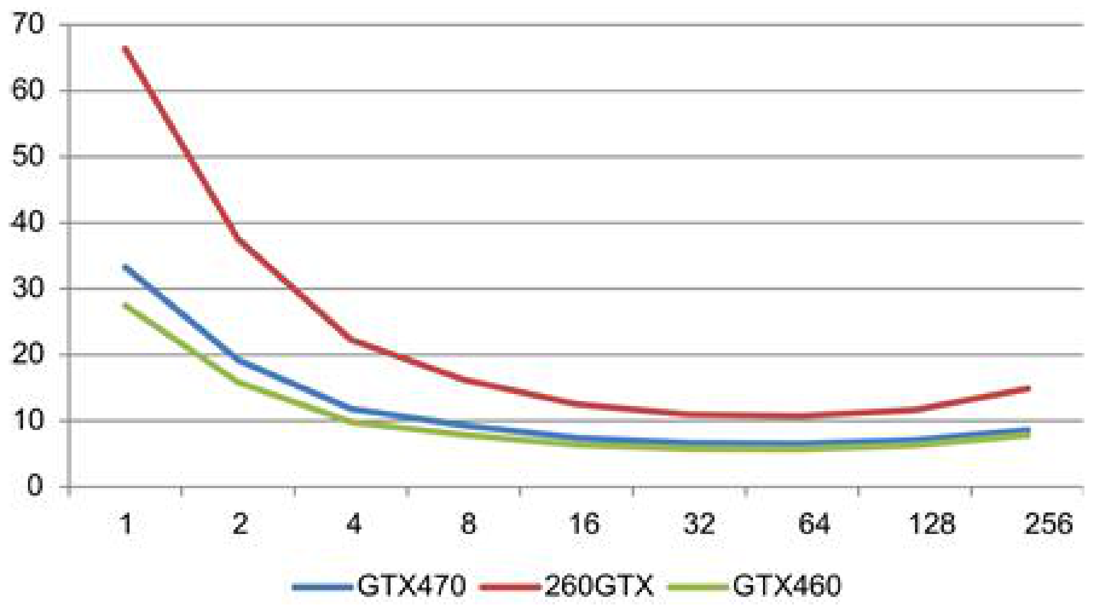
But in each merge

- a single value is written out to global memory
- a single value is read to shared memory (to replace the written value)
- so enough computation to hide the latency

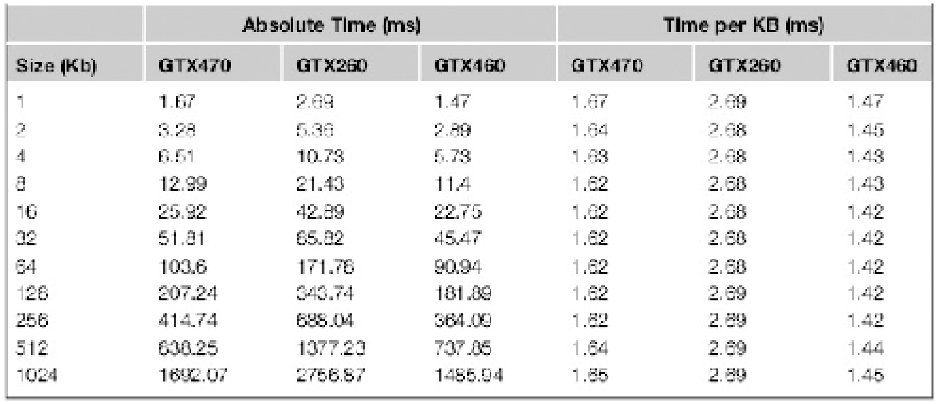
Radix Sort Results



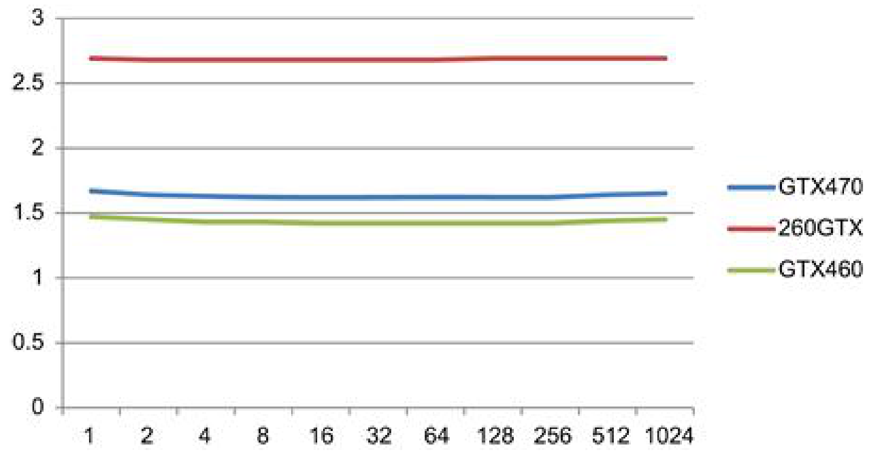
Threads	GTX470	GTX260	GTX460
1	33.27	66.32	27.47
2	19.21	37.53	15.87
4	11.82	22.29	9.83
8	9.31	16.24	7.68
16	7.41	12.52	6.38
32	6.63	10.95	5.75
64	6.52	10.72	5.71
128	7.06	11.63	6.29
256	8.61	14.88	7.82



For 64 Threads



	Absolute Time (ms)			Time per KB (ms)		
Size (Kb)	GTX470	GTX260	GTX460	GTX470	GTX260	GTX460
1	1.67	2.69	1.47	1.67	2.69	1.47
2	3.28	5.36	2.89	1.64	2.68	1.45
4	6.51	10.73	5.73	1.63	2.68	1.43
8	12.99	21.43	11.4	1.62	2.68	1.43
16	25.92	42.89	22.75	1.62	2.68	1.42
32	51.81	85.82	45.47	1.62	2.68	1.42
64	103.6	171.76	90.94	1.62	2.68	1.42
128	207.24	343.74	181.89	1.62	2.69	1.42
256	414.74	688.04	364.09	1.62	2.69	1.42
512	838.25	1377.23	737.85	1.64	2.69	1.44
1024	1692.07	2756.87	1485.94	1.65	2.69	1.45



But still slow since only use one block

- sort 40 MB per minute
- but a 1GB dataset would take 25 minutes which is too slow

Sample Sort

Need a better sorting algorithm

- one that does not need expensive merging
- one that emphasizes parallelism at every step along the way
- Sample Sort is such an algorithm

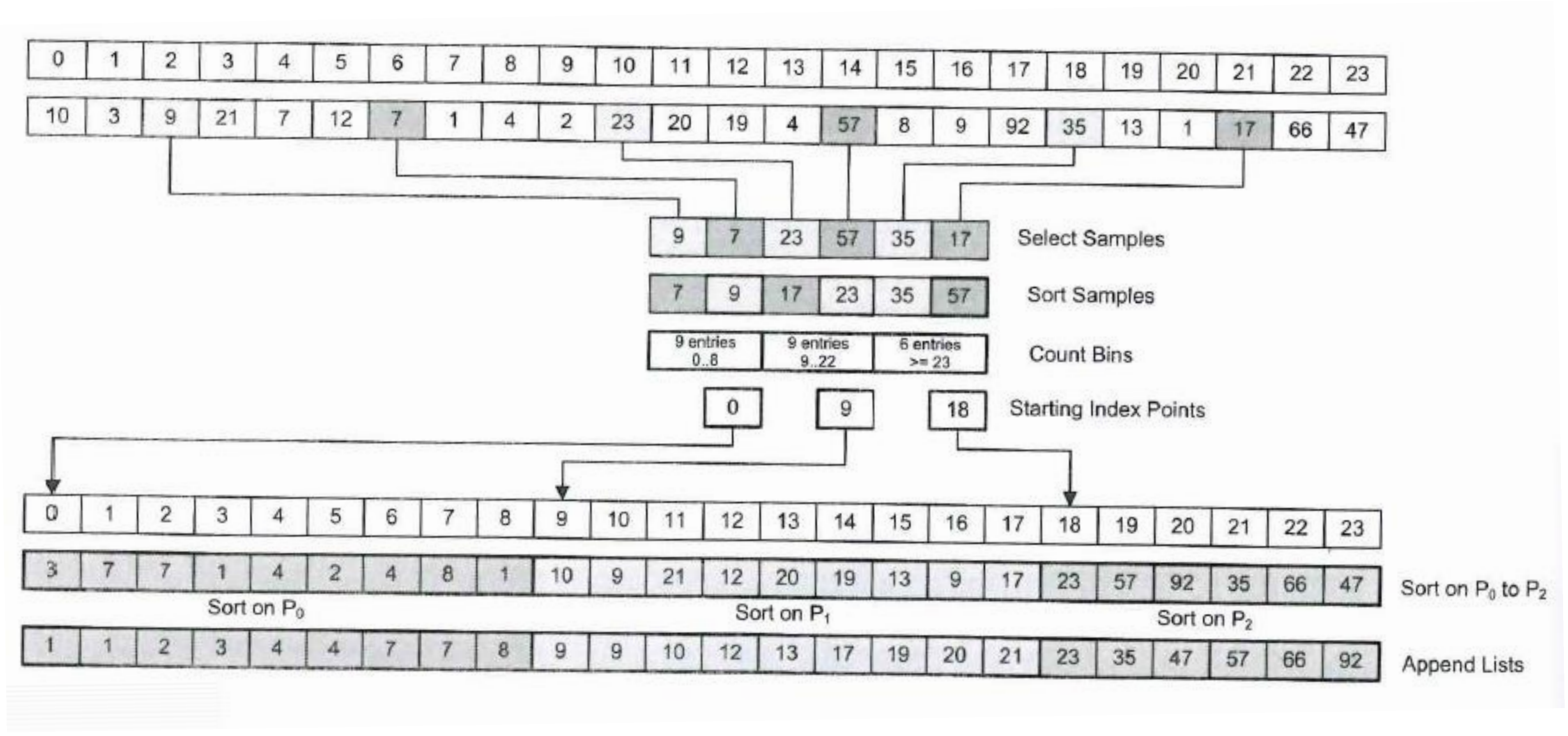
Well suited for parallel implementation

- belongs to the family of randomized algorithms

Sample Sort: Illustration (1)

Using three processors

- size of bins is 9, 9, 6



Sample Sort: Strategy

Randomly pick a set of *S* sample points

- assumes the unsorted list has no major concentration of values
- else must use more samples

Sort the samples in ascending order

From the *S* samples choose *P*-1 pivot points → *P* bins

- number of bins is given by the number of processors

Scan the dataset to see how many samples fit in each bin

Shuffle the dataset and assign the data to the bins

- all data in bin *p* is less than those in bin *p*+1 but greater than those in bin *p*-1

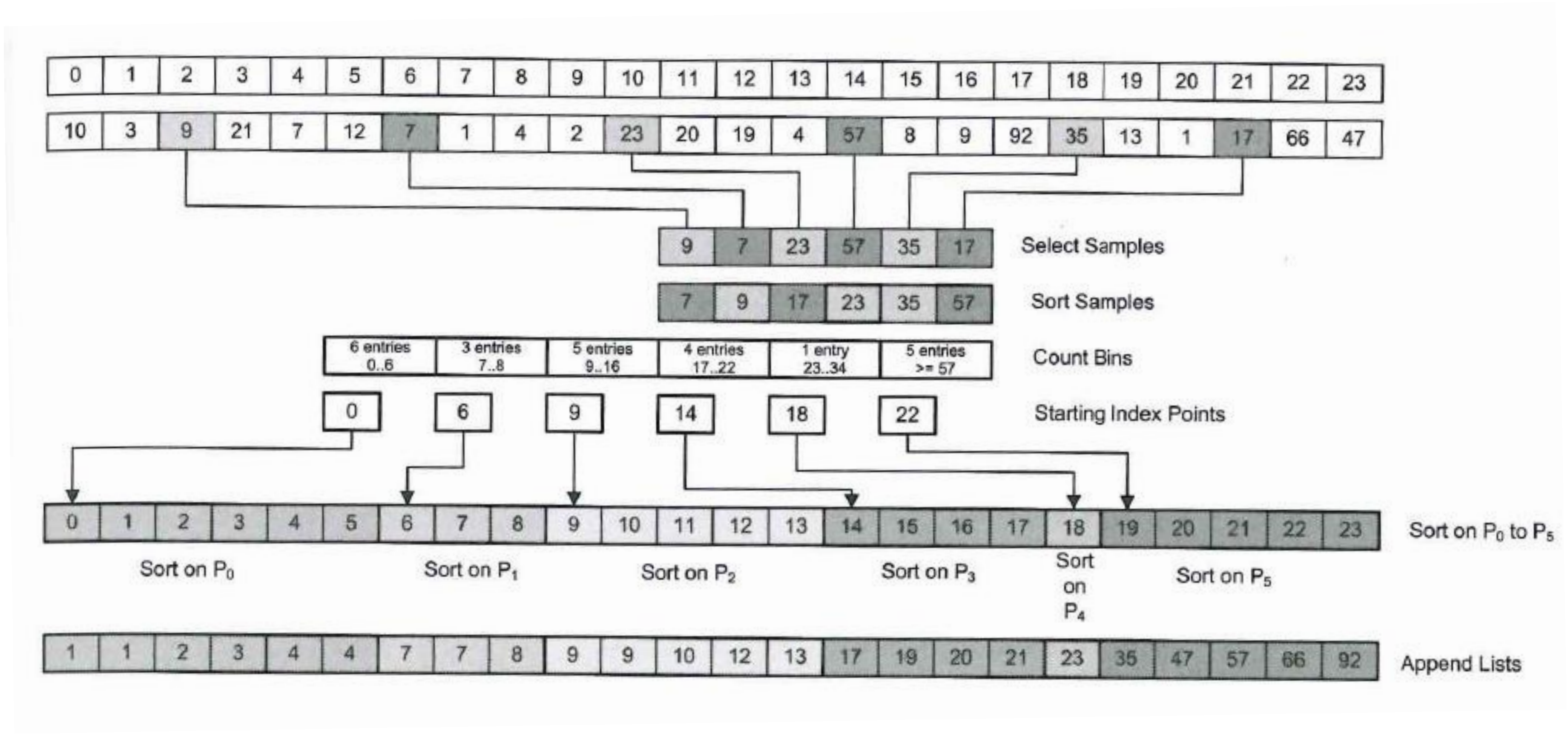
Sort each list separately in parallel

Append the lists

Sample Sort: Illustration (2)

Using six processors

- now size of bins is 6, 3, 5, 4, 1, 5
- largest bin determines speed of the parallel sorting phase
- recall 3-processor case → doubling the processors only reduced bin size by 1/3 → speedup ~ 1.5



Further Thoughts

Actual parallelism will depend on dataset

- best are datasets that are already somewhat sorted
- for example, the case where some new data elements are to be added

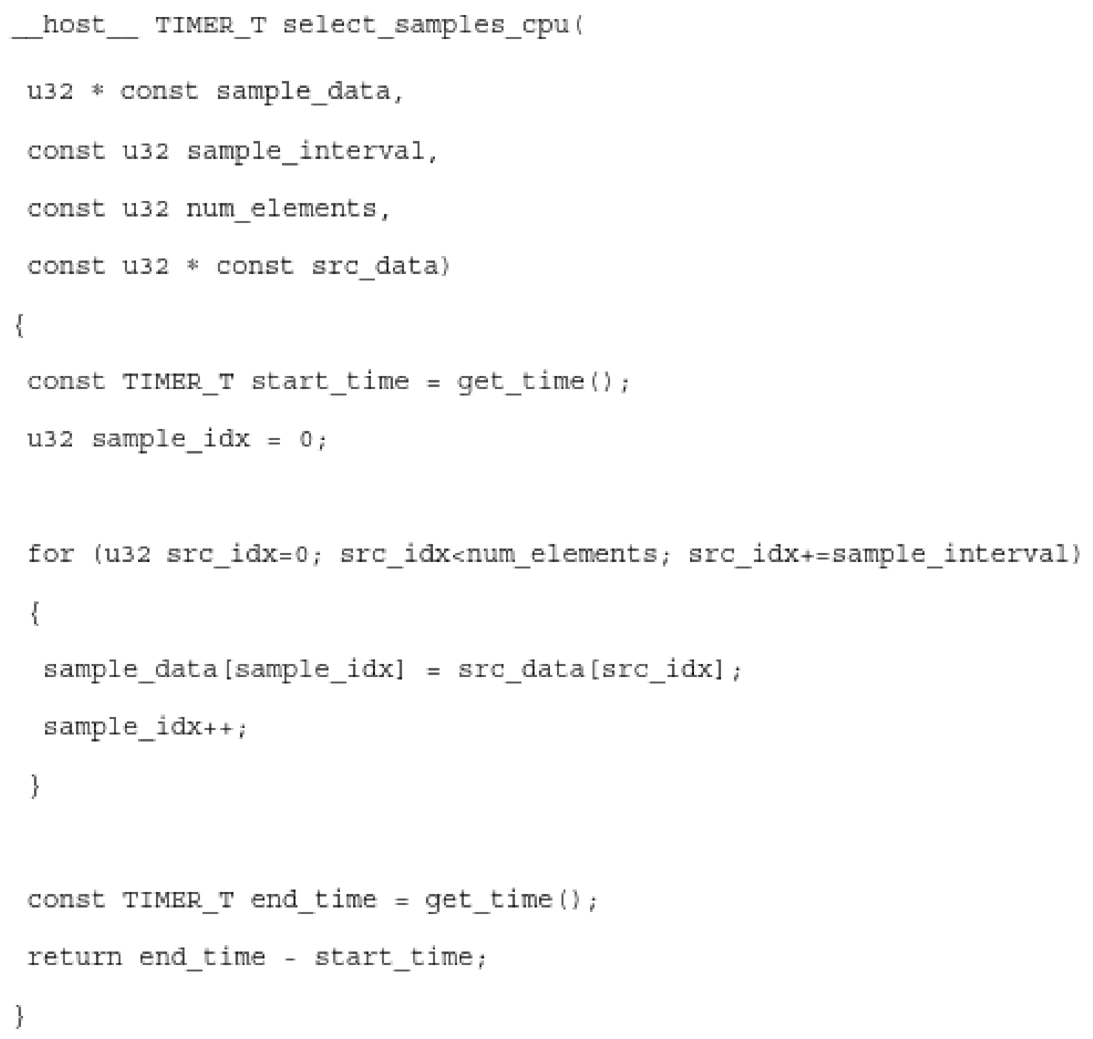
Also, we do not just have *P* processors

- we have *M* SMs
- each needs to run B blocks for latency hiding and so on
- each block would have ideally 256 threads
- if have 8 blocks for each of 14 SMs → 112 blocks in total
- but exact number is matter of optimization

Next:

- shall examine each GPU-accelerated component one by one

Selecting Samples: CPU Code

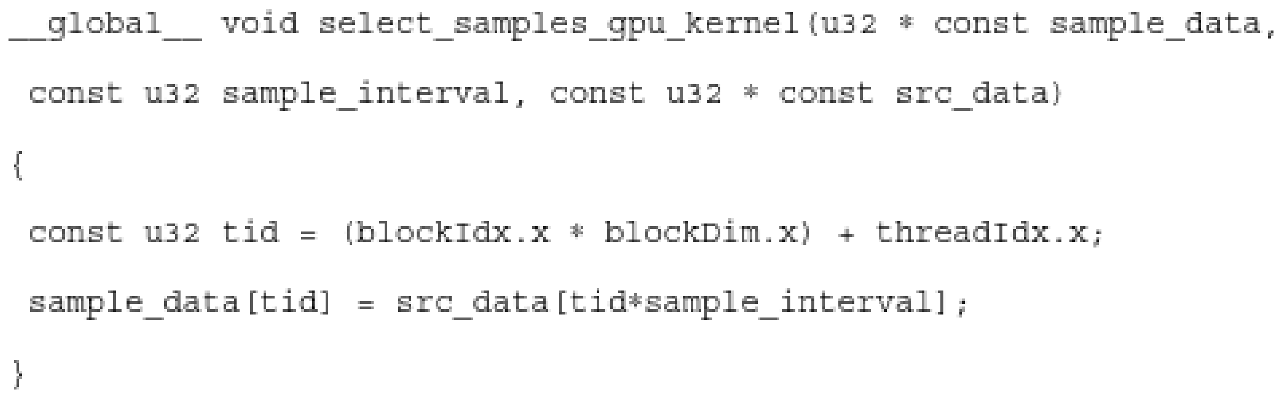


```
__host__ TIMER_T select_samples_cpu(
 u32 * const sample_data,
 const u32 sample_interval,
 const u32 num_elements,
 const u32 * const src_data)
{
 const TIMER_T start_time = get_time();
 u32 sample_idx = 0;

 for (u32 src_idx=0; src_idx<num_elements; src_idx+=sample_interval)
 {
  sample_data[sample_idx] = src_data[src_idx];
  sample_idx++;
 }

 const TIMER_T end_time = get_time();
 return end_time - start_time;
}
```

Selecting Samples: GPU Kernel Code

Now perform the sampling in parallel

- each thread picks a sample spaced apart by *sample_interva*l



```
__global__ void select_samples_gpu_kernel(u32 * const sample_data,
 const u32 sample_interval, const u32 * const src_data)
{
 const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;
 sample_data[tid] = src_data[tid*sample_interval];
}
```

Selecting Samples: GPU Host Code

```
__host__ TIMER_T select_samples_gpu(
 u32 * const sample_data,
 const u32 sample_interval,
 const u32 num_elements,
 const u32 num_samples,
 const u32 * const src_data,
 const u32 num_threads_per_block,
 const char * prefix)
{
 // Invoke one block of N threads per sample
 const u32 num_blocks = num_samples / num_threads_per_block;

 // Check for non equal block size
 assert((num_blocks * num_threads_per_block) == num_samples);
 start_device_timer();

 select_samples_gpu_kernel<<<num_blocks,   num_threads_per_block>>>
(sample_data, sample_interval, src_data);
 cuda_error_check(prefix,              "Error              invoking
select_samples_gpu_kernel");

 const TIMER_T func_time = stop_device_timer();

 return func_time;
}
```

Sorting the Samples

On the CPU:

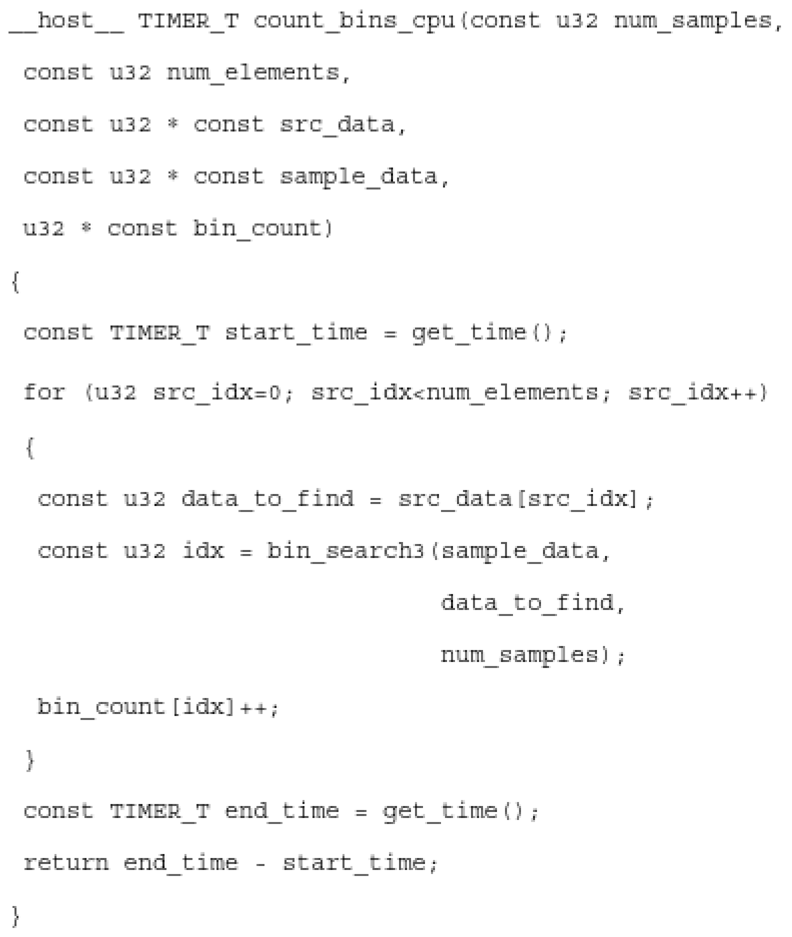
- could just use the *qsort()* routine from the standard C library

```
__host__ TIMER_T sort_samples_cpu(
 u32 * const sample_data,
 const u32 num_samples)
{
 const TIMER_T start_time = get_time();

 qsort(sample_data, num_samples, sizeof(u32),
   &compare_func);

 const TIMER_T end_time = get_time();
 return end_time - start_time;
}
```

On the GPU:

- use RadixSort – either ours of the implementation in the Thrust library
- note that our implementation was for a single SM in shared memory
- it also used shared memory reduction for merge
- more on this later

Counting The Sample Bins: CPU



```
__host__ TIMER_T count_bins_cpu(const u32 num_samples,
 const u32 num_elements,
 const u32 * const src_data,
 const u32 * const sample_data,
 u32 * const bin_count)
{
 const TIMER_T start_time = get_time();
 for (u32 src_idx=0; src_idx<num_elements; src_idx++)
 {
  const u32 data_to_find = src_data[src_idx];
  const u32 idx = bin_search3(sample_data,
                              data_to_find,
                              num_samples);
  bin_count[idx]++;
 }
 const TIMER_T end_time = get_time();
 return end_time - start_time;
}
```

For search have two options:

- binary search
- sequential search

Search Strategy

First note:

- we try to find a data value in a list of S sorted samples
- so in most cases the search will not be successful

Sequential search

- complexity O(S)
- since the list is sorted we will likely have S/2

Binary search

- worst case is O(log(S))
- we will hit it because most of the time we will not find a sample

Numerical example

- S = 1024
- binary would take 10 iterations
- sequential would take 512 iterations
- binary is better
- given N data points total time is $N \times 10$

Performance Issues for Binary Search

Execution

- branch divergence is frequent
- in the worst case need to multiply by the number of iterations
- but sample size not high enough to make this really a factor

Memory access

- not coalesced because of branch divergence
- L1/L2 cache may help here
- could also store all samples in shared memory

Binning: Host Code

Every thread bins one data element

```
  __host__ TIMER_T count_bins_gpu(
   const u32 num_samples,
   const u32 num_elements,
   const u32 * const src_data,
   const u32 * const sample_data,
   u32 * const bin_count,
   const u32 num_threads,
   const char * prefix)
  {
   const u32 num_blocks = num_elements / num_threads;

   start_device_timer();

   count_bins_gpu_kernel5<<<num_blocks,   num_threads>>>(num_samples,
src_data, sample_data, bin_count);
   cuda_error_check(prefix, "Error invoking count_bins_gpu_kernel");

   const TIMER_T func_time = stop_device_timer();
   return func_time;
  }
```

Binning: GPU Kernel

Every thread runs this

```
// Single data point, atomic add to gmem
__global__ void count_bins_gpu_kernel5(
 const u32 num_samples,
 const u32 * const src_data,
 const u32 * const sample_data,
 u32 * const bin_count)
{
 const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;

 // Read the sample point
 const u32 data_to_find = src_data[tid];

 // Obtain the index of the element in the search list
 const u32 idx = bin_search3(sample_data, data_to_find, num_samples);

 atomicAdd(&bin_count[idx],1);
}
```

Binning: Binary Search Function

Can be run on a CPU or GPU implementation (note function qualifier)

```
__host__ __device__ u32 bin_search3(
 const u32 * const src_data,
 const u32 search_value,
 const u32 num_elements)
{
 // Take the middle of the two sections
 u32 size = (num_elements >> 1);

 u32 start_idx = 0;
 bool found = false;
```

```
do
{
   const u32 src_idx = (start_idx+size);
   const u32 test_value = src_data[src_idx];

   if (test_value == search_value)
    found = true;
   else
    if (search_value > test_value)
     start_idx = (start_idx+size);

   if (found == false)
    size >>= 1;

 } while ( (found == false) && (size != 0) );

 return (start_idx + size);
}
```

Discussion

How about parallelism?

- determined by size of data array
- not the number of samples as was the case for the previous sampling and sorting

How about memory access?

- data reads are done in coalesced manner
- using more threads per block will increase read bandwidth

How about thread divergence?

- threads may diverge in the binary search
- but since we assume the data to be almost sorted threads will likely follow the same route
- the prevents divergence in practice

How about atomic writes?

- given that the data are mostly sorted
- will probably hit the same bin for all threads $\rightarrow$ will serialize the writes

Prefix Sum: CPU Code

Needed to generate a variable-size table for indexing the arrays

- size of each bin is variable length

```
__host__ TIMER_T calc_bin_idx_cpu(const u32 num_samples,
      const u32 * const bin_count,
      u32 * const dest_bin_idx)
{
 const TIMER_T start_time = get_time();
 u32 prefix_sum = 0;

 for (u32 i=0; i<num_samples; i++)
 {
  dest_bin_idx[i] = prefix_sum;
  prefix_sum += bin_count[i];
 }

 const TIMER_T end_time = get_time();
 return end_time - start_time;
}
```

Prefix Sum: Parallelism

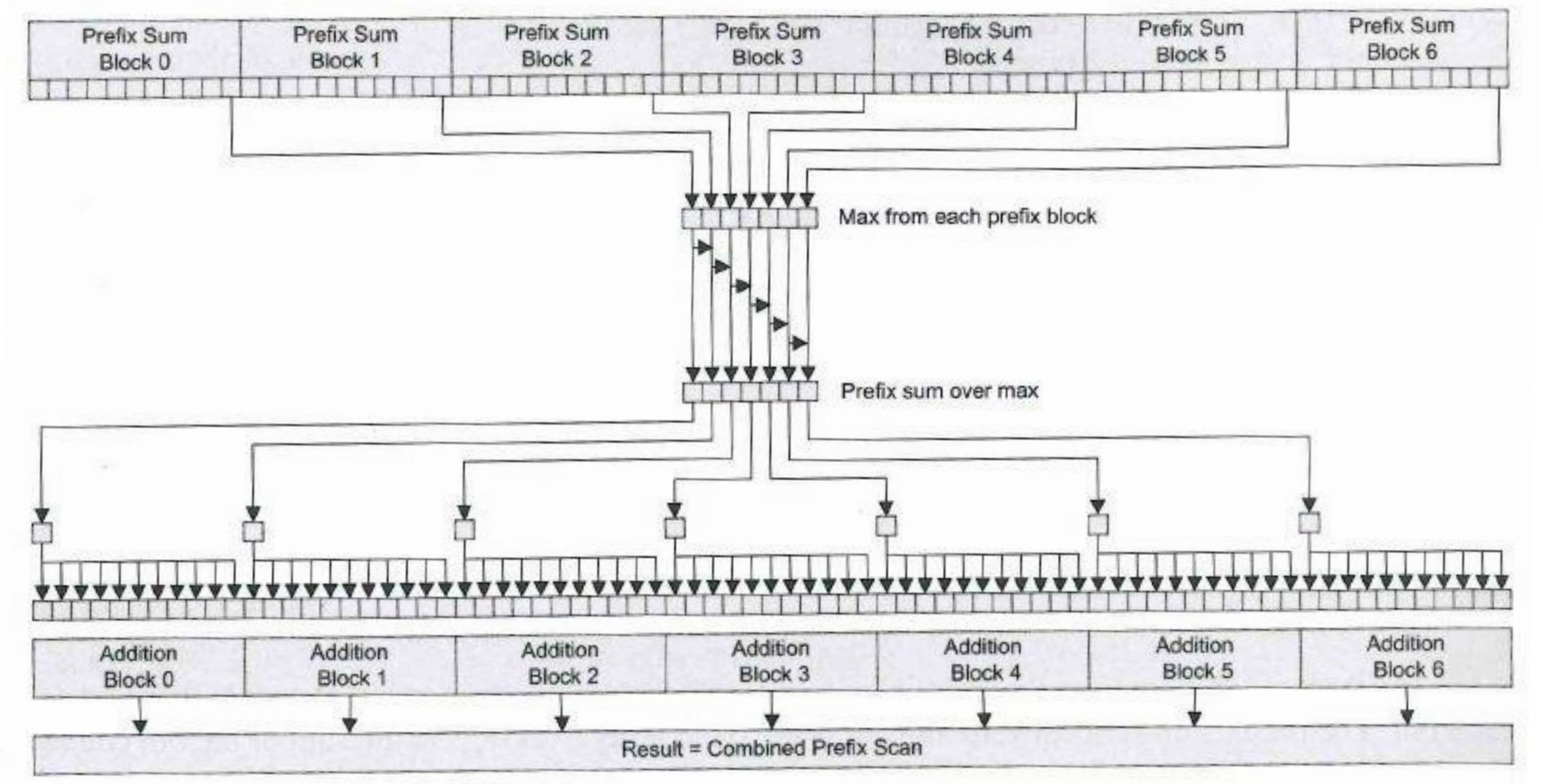
Problematic since it is inherently serial

- can’t compute an element before knowing the previous one
- turns out for small *N* serial prefix sum is quite fast
- need parallel solution when *N* is large

Parallel solution

- split the array into a number of blocks
- calculate the prefix sum on those blocks
- place the end point of each prefix sum block into another array
- compute another prefix sum, in place, on this array
- add the result of this prefix sum to each element in the original prefix sum calculation

Parallel Prefix Sum: Illustration



Considerations

Memory access

- uses a single thread per block
- there is no thread divergence
- however the read memory access is poorly coalesced
- thread 0 will be accessing addresses starting at a zero offset
- thread 1 will be accessing addresses starting at a (NUM_SAMPLES/NUM_BLOCKS) offset

Need multiple synchronization points

- could just use three kernels
- this will enable us to reconfigure block sizes and number of blocks
- could also run small prefix sums on the CPU (when N < 4096)

Parallel Prefix Sum: First Stage

```
__global__ void calc_prefix_sum_kernel(
 const u32 num_samples_per_thread,
 const u32 * const bin_count,
 u32 * const prefix_idx,
 u32 * const block_sum)
{
 const u32 tid = (blockIdx.x * blockDim.x) + threadIdx
   const u32 tid_offset = tid * num_samples_per_thread;
   u32 prefix_sum;

   if (tid == 0)
    prefix_sum = 0;
   else
    prefix_sum = bin_count[tid_offset-1];

   for (u32 i=0; i<num_samples_per_thread; i++)
   {
    prefix_idx[i+tid_offset] = prefix_sum;
    prefix_sum += bin_count[i+tid_offset];
   }

   // Store the block prefix sum as the value from the last element
   block_sum[tid]                    =                    prefix_idx
[(num_samples_per_thread-1uL)+tid_offset];
  }
```

Parallel Prefix Sum: Second and Third Stage

```
__global__ void add_prefix_sum_total_kernel(
 u32 * const prefix_idx,
 const u32 * const total_count)
{
 const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;

 prefix_idx[tid] += total_count[blockIdx.x];
}
```

```
__global__ void calc_prefix_sum_kernel_single(
 const u32 num_samples,
 const u32 * const bin_count,
 u32 * const dest_bin_idx)
{
 u32 prefix_sum = 0;

 for (u32 i=0; i<num_samples; i++)
 {
  dest_bin_idx[i] = prefix_sum;
  prefix_sum += bin_count[i];
 }
}
```

Host Function (1)

```
  __host__ TIMER_T calc_bin_idx_gpu(
   const u32 num_elements,
   const u32 * const bin_count,
   u32 * const dest_bin_idx,
   const u32 num_threads_per_block,
   u32 num_blocks,
   const char * prefix,
   u32 * const block_sum,
   u32 * const block_sum_prefix)
  {
   start_device_timer();

   if (num_elements >= 4096)
   {
    const u32 num_threads_total = num_threads_per_block
                                * num_blocks;

    const    u32    num_elements_per_thread    =    num_elements    /
num_threads_total;
```

Host Function (2)

```
    // Make sure the caller passed arguments which correctly divide the
elements to blocks and threads
    assert( (num_elements_per_thread *
            num_threads_total) == num_elements );

    // First calculate the prefix sum over a block
    calc_prefix_sum_kernel<<<num_blocks,     num_threads_per_block>>>
(num_elements_per_thread, bin_count, dest_bin_idx, block_sum);

    cuda_error_check(prefix, "Error invoking calc_prefix_sum_kernel");

    // Calculate prefix for the block sums
    // Single threaded
           calc_prefix_sum_kernel_single<<<1,1>>>(num_threads_total,
block_sum, block_sum_prefix);
    cuda_error_check(prefix,             "Error             invoking
calc_prefix_sum_kernel_single");

    // Add the prefix sums totals back into the original prefix blocks
    // Switch to N threads per block
    num_blocks = num_elements /
                num_elements_per_thread;
```

Host Function (3)

```
                              add_prefix_sum_total_kernel<<<num_blocks,
num_elements_per_thread>>>(dest_bin_idx, block_sum_prefix);

    cuda_error_check(prefix, "add_prefix_sum_total_kernel");
   }
   else
   {
    // Calculate prefix for the block sums
    // Single threaded
    calc_prefix_sum_kernel_single<<<1,1>>>(num_elements,   bin_count,
dest_bin_idx);

    cuda_error_check(prefix,              "Error              invoking
calc_prefix_sum_kernel_single");
   }
   const TIMER_T func_time = stop_device_timer();
   return func_time;
  }
```

Sorting Into Bins

```
__host__ TIMER_T sort_to_bins_cpu(
 const u32 num_samples,
 const u32 num_elements,
 const u32 * const src_data,
 const u32 * const sample_data,
 const u32 * const bin_count,
 const u32 * const dest_bin_idx,
 u32 * const dest_data)
{
 const TIMER_T start_time = get_time();

 u32 dest_bin_idx_tmp[NUM_SAMPLES];

 // Copy the dest_bin_idx array to temp storage
 for (u32 bin=0;bin<NUM_SAMPLES;bin++)
 {
  dest_bin_idx_tmp[bin] = dest_bin_idx[bin];
 }

 // Iterate over all source data points
 for (u32 src_idx=0; src_idx<num_elements; src_idx++)
 {
  // Read the source data
  const u32 data = src_data[src_idx];

  // Identify the bin in which the source data
  // should reside
  const u32 bin = bin_search3(sample_data,
                              data,
                              num_samples);

  // Fetch the current index for that bin
  const u32 dest_idx = dest_bin_idx_tmp[bin];

  // Write the data using the current index
  // of the correct bin
  dest_data[dest_idx] = data;

  // Increment the bin index
  dest_bin_idx_tmp[bin]++;
 }

 const TIMER_T end_time = get_time();
 return end_time - start_time;
}
```

Parallel Version

```
__global__ void sort_to_bins_gpu_kernel(
 const u32 num_samples,
 const u32 * const src_data,
 const u32 * const sample_data,
 u32 * const dest_bin_idx_tmp,
 u32 * const dest_data)
{
 // Calculate the thread we're using
 const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;

 // Read the sample point
 const u32 data = src_data[tid];

 // Identify the bin in which the
 // source data should reside
 const u32 bin = bin_search3(sample_data,
                             data,
                             num_samples);

 // Increment the current index for that bin
 const u32 dest_idx = atomicAdd(&dest_bin_idx_tmp[bin],1);

 // Write the data using the
 // current index of the correct bin
 dest_data[dest_idx] = data;
}
```

Final operations

Sort the individuals bins

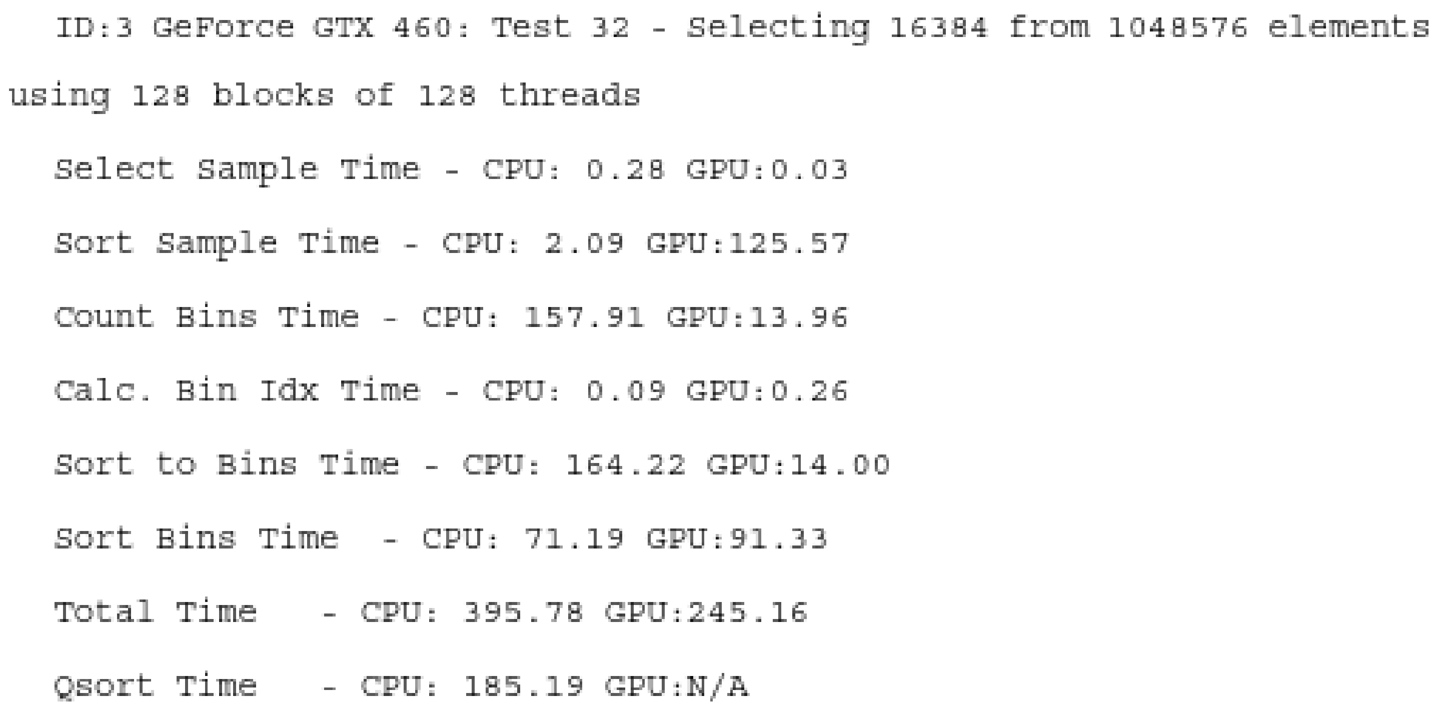
- can use parallel radix sort

Append the lists

See book for details

Performance Breakdown

For GTX 460

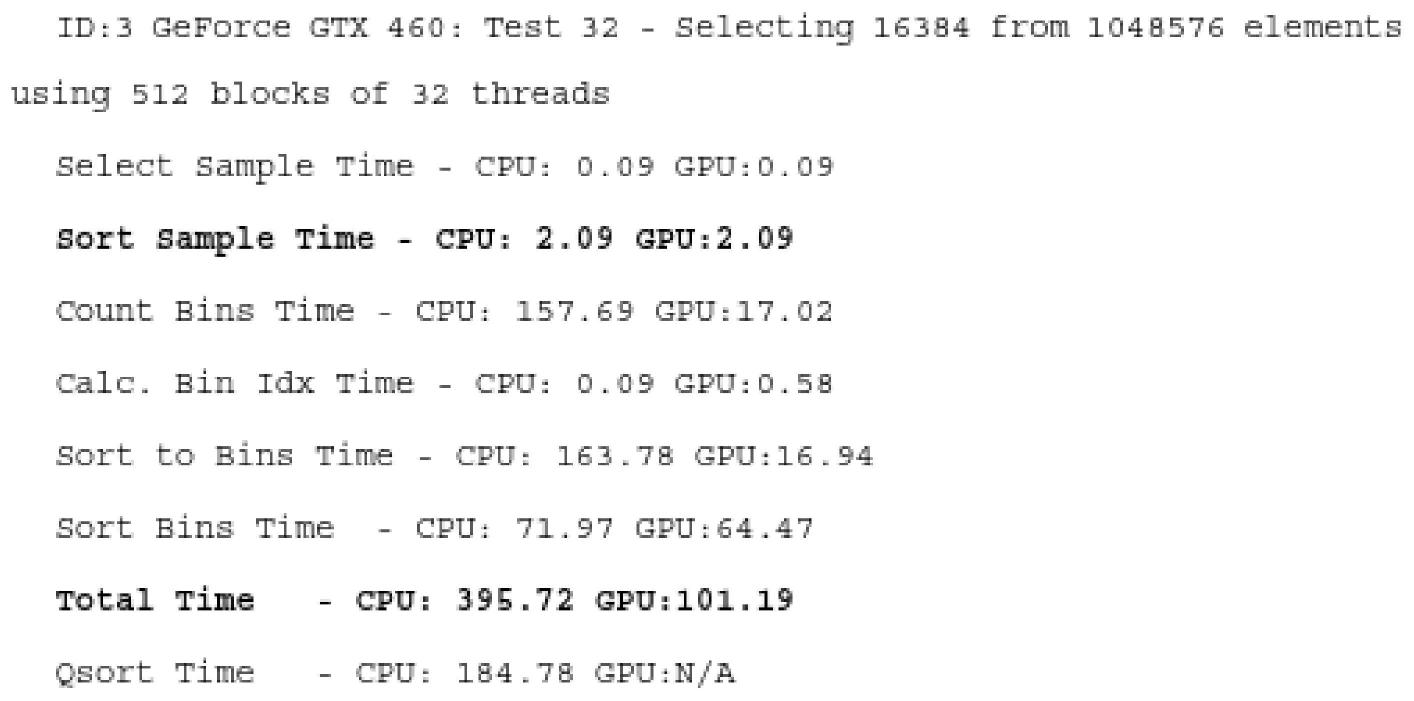


```
   ID:3 GeForce GTX 460: Test 32 - Selecting 16384 from 1048576 elements
using 128 blocks of 128 threads
   Select Sample Time - CPU: 0.28 GPU:0.03
   Sort Sample Time - CPU: 2.09 GPU:125.57
   Count Bins Time - CPU: 157.91 GPU:13.96
   Calc. Bin Idx Time - CPU: 0.09 GPU:0.26
   Sort to Bins Time - CPU: 164.22 GPU:14.00
   Sort Bins Time  - CPU: 71.19 GPU:91.33
   Total Time    - CPU: 395.78 GPU:245.16
   Qsort Time    - CPU: 185.19 GPU:N/A
```

Performance

Performance ratio GPU/CPU improves for larger sizes

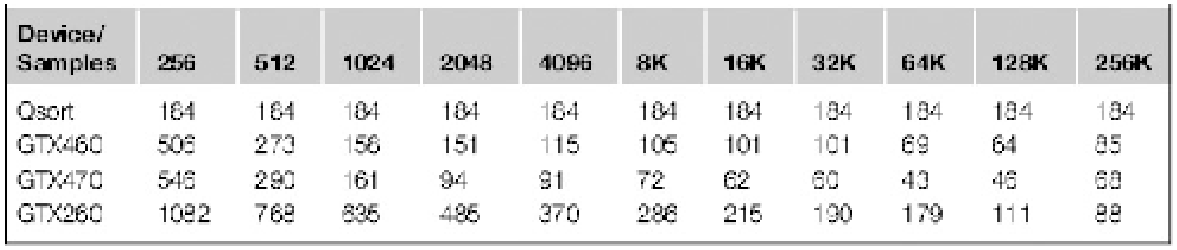
- note that sample sort is about 55% of the qsort CPU time
- so perform sample sorting on CPU and other operations on the GPU
- this changes the performance as follows:



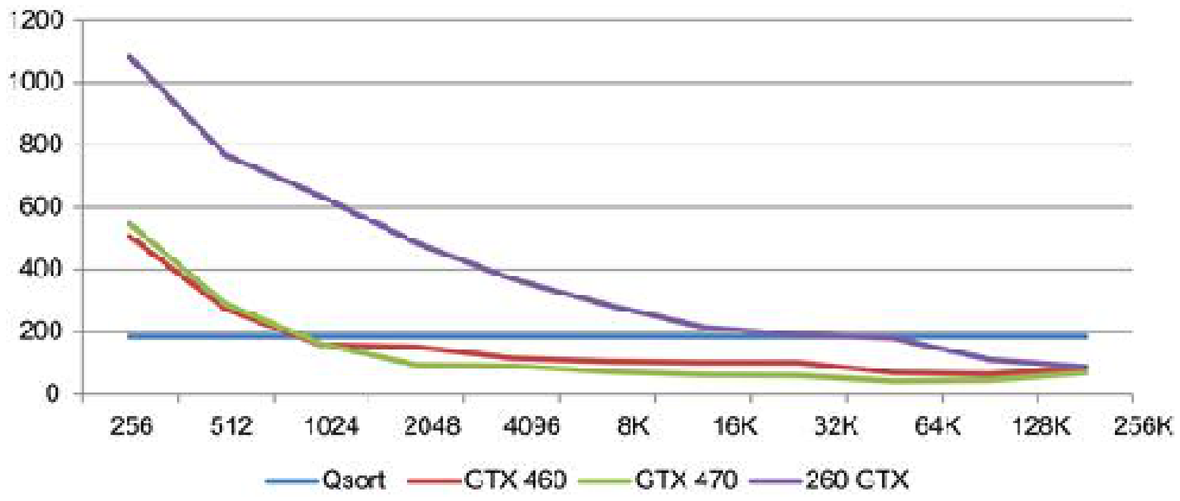
```
  ID:3 GeForce GTX 460: Test 32 - Selecting 16384 from 1048576 elements
using 512 blocks of 32 threads
  Select Sample Time - CPU: 0.09 GPU:0.09
  Sort Sample Time - CPU: 2.09 GPU:2.09
  Count Bins Time - CPU: 157.69 GPU:17.02
  Calc. Bin Idx Time - CPU: 0.09 GPU:0.58
  Sort to Bins Time - CPU: 163.78 GPU:16.94
  Sort Bins Time   - CPU: 71.97 GPU:64.47
  Total Time    - CPU: 395.72 GPU:101.19
  Qsort Time    - CPU: 184.78 GPU:N/A
```

- but for larger sample sizes > 128K CPU becomes a bottleneck again

Performance Charts



Device/ Samples	256	512	1024	2048	4096	8K	16K	32K	64K	128K	256K
Qsort	164	164	184	184	164	184	184	184	184	184	184
GTX460	506	273	156	151	115	105	101	101	69	64	85
GTX470	546	290	161	94	91	72	62	60	43	46	68
GTX260	1082	768	635	486	370	286	215	190	179	111	88



NVISION 08 Highlights: GPU vs CPU Demonstration

- http://www.youtube.com/watch?v=mwDPb3T8bOQ

GPU Technology Conference Keynote

- http://www.youtube.com/watch?v=A84v7lbdcYg