CSE 591: GPU Programming

Memories: Registers and Shared

Klaus Mueller

Computer Science Department

Stony Brook University

Code examples from Shane Cook "CUDA Programming"

Locality

Spatial locality

- if one thread accesses a memory location, another parallel thread will likely access a neighbor

Temporal locality

- most programs will access that same location again within a short time period

DRAMS slower than processors

- 1.6 GHz vs. 3 GHz
- local (fast) caches buffers this discrepancy
- cache overcomes both latency and bandwidth problems

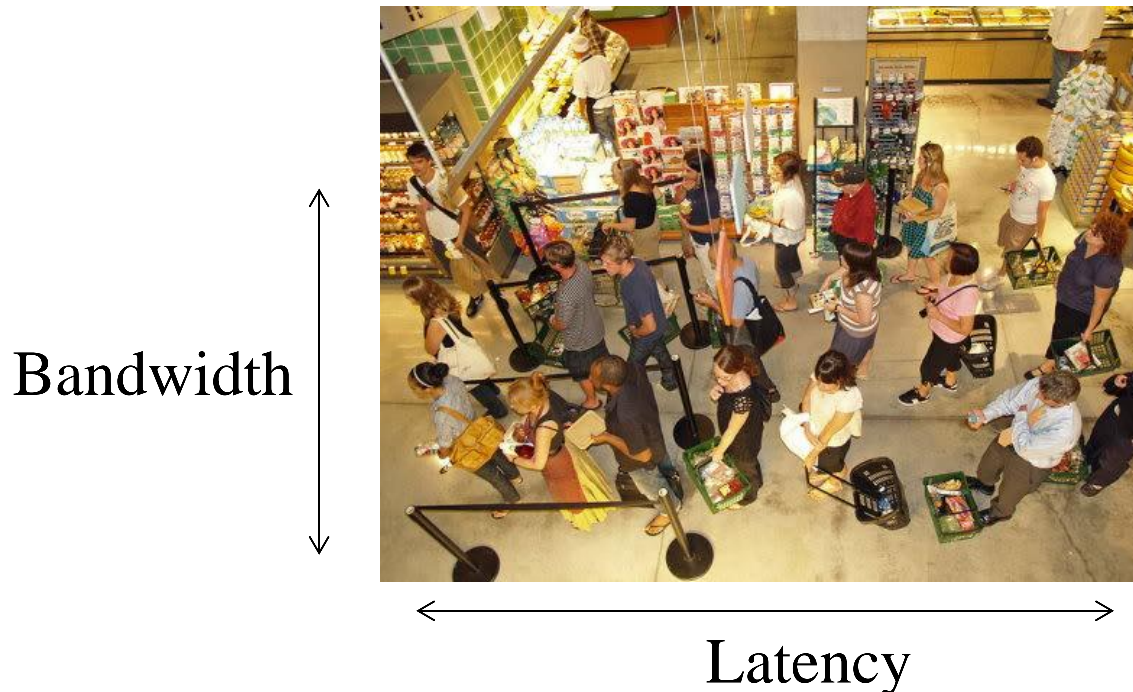
Latency vs. Bandwidth

Latency:

- amount of time it takes to respond to a fetch request
- 100s of clock cycles
- request more than one data item at a time – amortize wait time

Bandwidth:

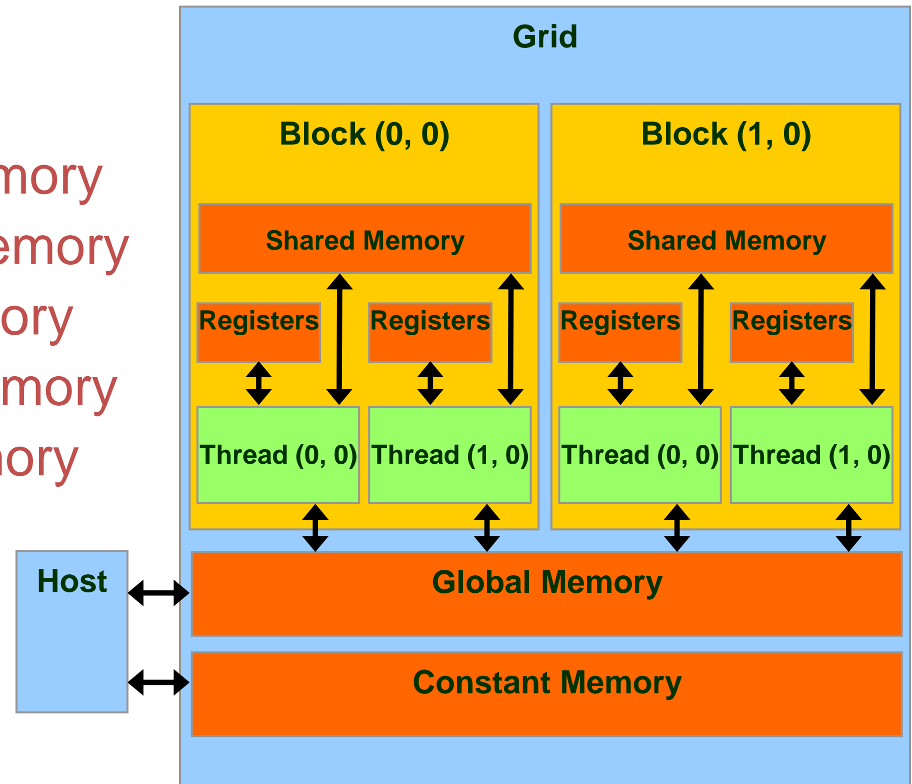
- amount of data you can read/store to DRAM in a given period of time



G80 Implementation of CUDA Memories

Each thread can:

- Read/write per-thread registers
- Read/write per-thread local memory
- Read/write per-block shared memory
- Read/write per-grid global memory
- Read/only per-grid constant memory
- Read/only per-grid texture memory



- Latency and bandwidth

Storage Type	Registers	Shared Memory	Texture Memory	Constant Memory	Global Memory
Bandwidth	~8 TB/s	~1.5 TB/s	~200 MB/s	~200 MB/s	~200 MB/s
Latency	1 cycle	1 to 32 cycles	~400 to 600	~400 to 600	~400 to 600

GPU vs. CPU

A large difference is in context switching

- CPUs take 100s of clock cycles to swap threads
- involves register renaming – registers are saved to the stack
- stack must read back when thread is swapped in

- GPU threads are lightweight
- no register renaming – each thread has its own set of registers
- thread swapping simple moves a pointer
- the limitation is the number of registers available
- this can lead to sudden performance drops if a block cannot be scheduled anymore

Registers

Registers are very fast

- use it for items often read and written (loop vars, accumulators, ..)

Depending on hardware

- 8 K, 16 K, 32 K or 64 K of space per SM for all threads within an SM
- each thread needs at least one register

Register use example – assume Fermi with 32k

- let's say we have 256 threads per block
- float (4 bytes)
- (32,768/4 bytes per register)/256 threads = 32 registers per thread available

More threads will reduce the number of available registers

What then?

- need to move to larger memory transactions
- introduce ILP (Instruction Level Parallelism) – process more than one element of the dataset within a single thread

Register Variables

All variables that are declared without qualifiers

- are called *automatic variables*
- they are allocated to registers

If they do not all fit

- then they get allocated to local memory
- local memory is really global memory (slow)
- so better keep track!

Shared Memory

*User-controlle*d L1 cache

- there is also hardware-controlled L1 cache

L1 cache + shared memory = 64 K memory segment per SM

- can be configured in 16k block for either
- no L1 cache in pre-Fermi cards, just shared memory
- has 1.5 TB/s bandwidth with extremely low latency
- hugely superior to the up to 190 GB/s available from global memory
- but around 1/5 of the speed of registers

GPUs have a load/store architecture

- any operand must be loaded into register prior to any operation

Loading the variable into shared memory must be justified by

- intended re-use
- coalescing of global memory
- data sharing among threads
- else it's more efficient to lead the variable from global memory directly

Shared Memory Organization

Organized into memory banks

- Fermi 32 banks (one warp)
- previously 16 banks (half warp)

Memory bank conflicts

- each bank can only serve one request per cycle
- no need for sequential access
- just need for exclusive access
- fast crossbar switch handles bank-processor communications
- broadcast mechanism in place when all threads access the same memory location

Bank conflicts cause inefficiencies

- serialize the reads/writes at various degrees
- cannot be hidden and stall the SM

Working Example

Let’s have a look at sorting algorithms

- these typically involve recursion and inconsistent execution flow
- quick-sort great serial but not good for parallel computing
- merge sort better but also not optimal

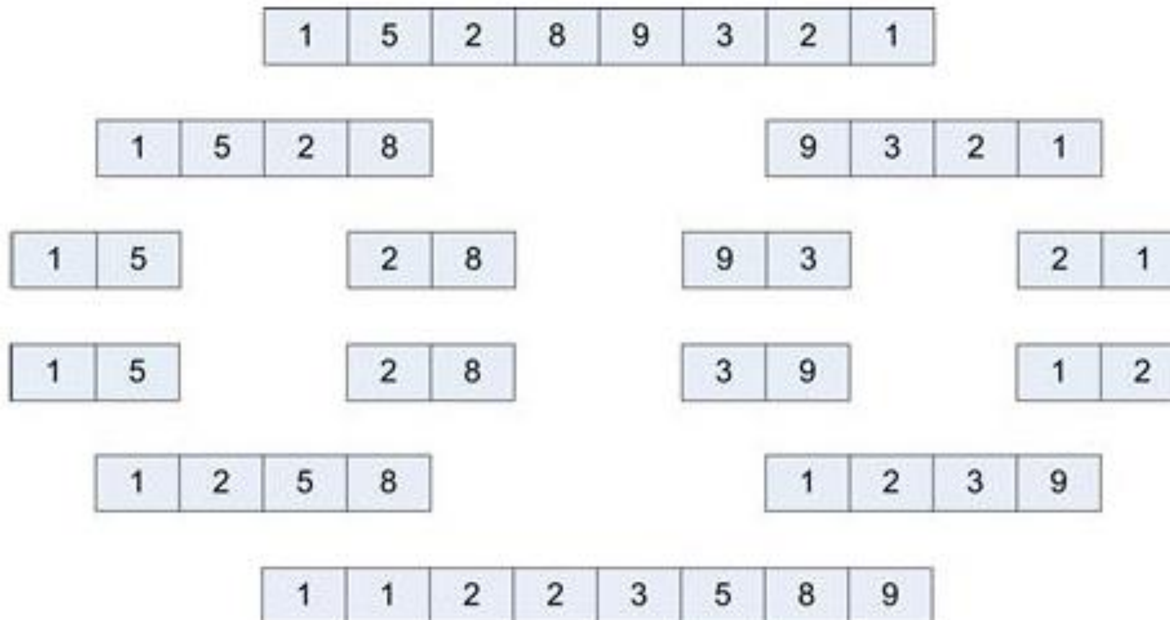
Very good for parallel execution is radix sort

- fixed number of iterations
- consistent execution flow

Merge Sort

Algorithm

- recursively partition the data
- sort the subsets
- recursively merge the subsets



Merge Sort: Parallel Implementation

After down-ward recursion

- have N/2 threads for 2-element sorting

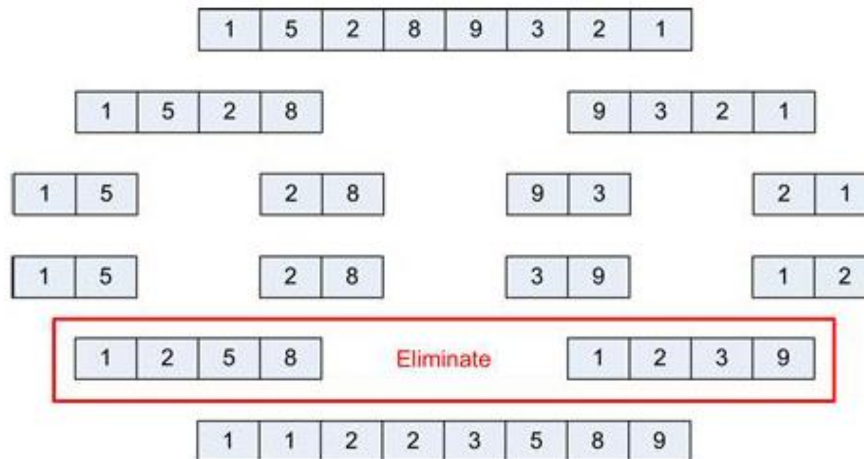
Example:

- sort list of 128K floats
- need 64K threads
- with 16 SMs and 1536 threads each get 24K threads per pass
- need 2.5 passes to sort all pairs

Merge Sort: Parallel Implementation

Next problem: the merging

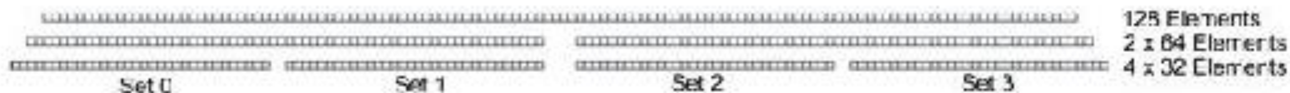
- parallelism halves for each merge step up
- one solution would merge all elements with two threads



- but this is not very efficient because we want full warps

Solution: recursion only down to 32-element sets

- this will also consume less threads in the sorting stage (only 1 pass)



Shared Memory Layout

Merging threads need to cooperate

- shared memory storage is required
- need to assure bank conflict free layout

Radix Sort by Example

Consider this array:

```
{ 122, 10, 2, 1, 2, 22, 12, 9 }
```

The binary representation of each of these would be

```
122 = 01111010

 10 = 00001010

 2 = 00000010

 22 = 00010010

 12 = 00001100

 9 = 00001001
```

Radix Sort by Example

In the first pass of the list, all elements with a 0 in the least significant bit (the right side) would form the first list. Those with a 1 as the least significant bit would form the second list. Thus, the two lists are

```
0 = { 122, 10, 2, 22, 12 }

1 = { 9 }
```

The two lists are appended in this order, becoming

```
{ 122, 10, 2, 22, 12, 9 }
```

The process is then repeated for bit one, generating the next two lists based on the ordering of the previous cycle:

```
0 = { 12, 9 }

1 = { 122, 10, 2, 22 }
```

The combined list is then

```
{ 12, 9, 122, 10, 2, 22 }
```

Scanning the list by bit two, we generate

```
0 = { 9, 122, 10, 2, 22 }

1 = { 12 }

= { 9, 122, 10, 2, 22, 12 }
```

Requires N + 2N memory cells

Serial Radix Sort Code

```
__host__ void cpu_sort(u32 * const data,
                       const u32 num_elements)
{
 static u32 cpu_tmp_0[NUM_ELEM];
 static u32 cpu_tmp_1[NUM_ELEM];

 for (u32 bit=0;bit<32;bit++)
 {
  u32 base_cnt_0 = 0;
  u32 base_cnt_1 = 0;

  for (u32 i=0; i<num_elements; i++)
  {
   const u32 d = data[i];
   const u32 bit_mask = (1 << bit);

   if ( (d & bit_mask) > 0 )
   {
    cpu_tmp_1[base_cnt_1] = d;
    base_cnt_1++;
   }
   else
   {
    cpu_tmp_0[base_cnt_0] = d;
    base_cnt_0++;
   }
  }

  // Copy data back to source - first the zero list
  for (u32 i=0; i<base_cnt_0; i++)
  {
   data[i] = cpu_tmp_0[i];
  }

  // Copy data back to source - then the one list
  for (u32 i=0; i<base_cnt_1; i++)
  {
   data[base_cnt_0+i] = cpu_tmp_1[i];
  }
 }
}
```

GPU Radix Sort

```
__global__ void gpu_sort_array_array(
 u32 * const data,
 const u32 num_lists,
 const u32 num_elements)
{
 const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;
 __shared__ u32 sort_tmp[NUM_ELEM];
 __shared__ u32 sort_tmp_1[NUM_ELEM];

 copy_data_to_shared(data, sort_tmp, num_lists,
                     num_elements, tid);

 radix_sort2(sort_tmp, num_lists, num_elements,
             tid, sort_tmp_1);

 merge_array6(sort_tmp, data, num_lists,
              num_elements, tid);
}
```

GPU Radix Sort Code

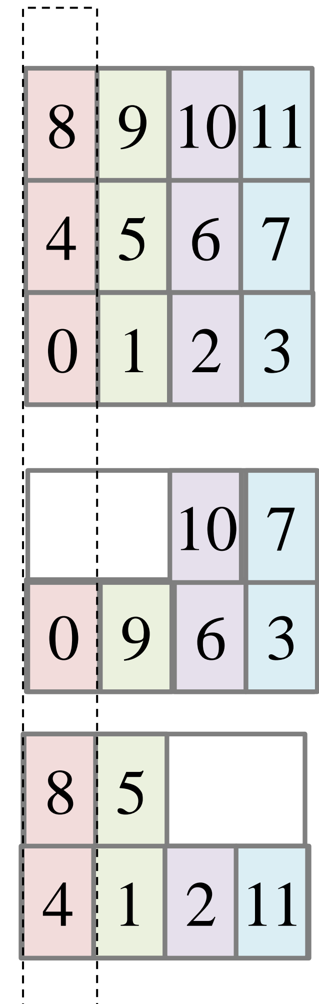
```
__device__ void radix_sort(u32 * const sort_tmp,
                           const u32 num_lists,
                           const u32 num_elements,
                           const u32 tid,
                           u32 * const sort_tmp_0,
                           u32 * const sort_tmp_1)
{
 // Sort into num_list, lists
 // Apply radix sort on 32 bits of data
 for (u32 bit=0;bit<32;bit++)
 {
  u32 base_cnt_0 = 0;
  u32 base_cnt_1 = 0;

  for (u32 i=0; i<num_elements; i+=num_lists)
  {
     const u32 elem = sort_tmp[i+tid];
     const u32 bit_mask = (1 << bit);

     if ( (elem & bit_mask) > 0 )
     {
      sort_tmp_1[base_cnt_1+tid] = elem;
      base_cnt_1+=num_lists;
     }
     else
     {
      sort_tmp_0[base_cnt_0+tid] = elem;
      base_cnt_0+=num_lists;
     }
  }

  // Copy data back to source - first the zero list
  for (u32 i=0; i<base_cnt_0; i+=num_lists)
  {
   sort_tmp[i+tid] = sort_tmp_0[i+tid];
  }

  // Copy data back to source - then the one list
  for (u32 i=0; i<base_cnt_1; i+=num_lists)
  {
   sort_tmp[base_cnt_0+i+tid] = sort_tmp_1[i+tid];
  }
 }
 __syncthreads();
}
```

Memory Layout

Example: 4 threads , 12 numbers

bank 1 thread 1　　bank 1 thread 1　　bank 1 thread 1

sort_tmp	0	1	2	3	4	5	6	7	8	9	10	11
sort_tmp_0	0	9	6	3			10	7				
sort_tmp_1	4	1	2	11	8	5						

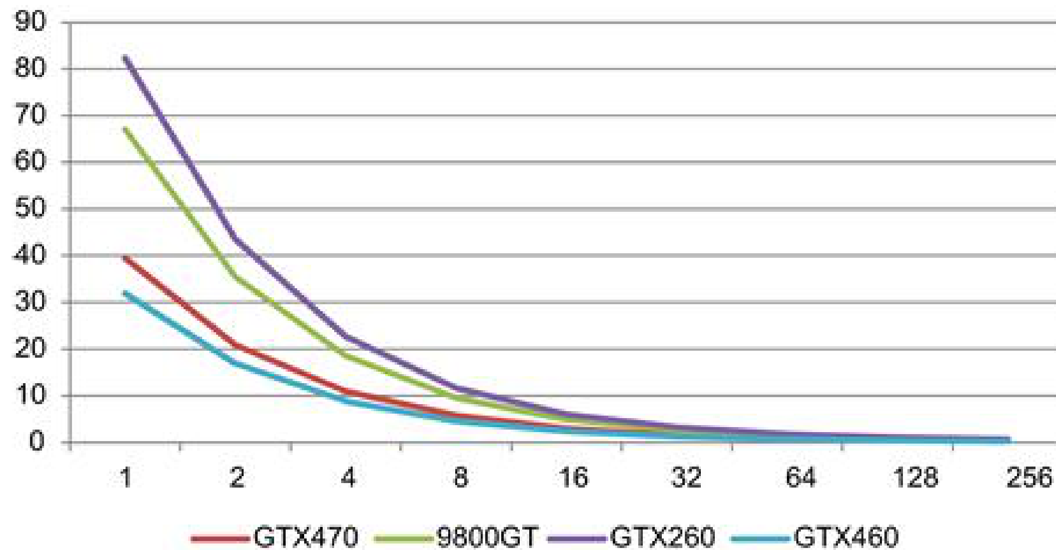


GPU Radix Sort Code

Each thread computes *num_lists*

- ideally num_lists = warp size or a multiple
- will avoid bank conflicts

Device/Threads	1	2	4	8	16	32	64	128	256
GTX470	39.4	20.6	10.9	5.74	2.91	1.55	0.83	0.48	0.3
9800GT	67	35.5	18.6	9.53	4.88	2.66	1.44	0.82	0.56
GTX260	82.4	43.5	22.7	11.7	5.99	3.24	1.77	1.02	0.66
GTX460	31.9	16.9	8.83	4.56	2.38	1.27	0.69	0.4	0.28



Optimizing the Code

```
__device__ void radix_sort2(u32 * const sort_tmp,
                            const u32 num_lists,
                            const u32 num_elements,
       const u32 tid,
       u32 * const sort_tmp_1)
{
 // Sort into num_list, lists
 // Apply radix sort on 32 bits of data
 for (u32 bit=0;bit<32;bit++)
 {
  const u32 bit_mask = (1 << bit);

   u32 base_cnt_0 = 0;
   u32 base_cnt_1 = 0;
 for (u32 i=0; i<num_elements; i+=num_lists)
 {
  const u32 elem = sort_tmp[i+tid];

  if ( (elem & bit_mask) > 0 )
  {
   sort_tmp_1[base_cnt_1+tid] = elem;
   base_cnt_1+=num_lists;
  }
  else
  {
   sort_tmp[base_cnt_0+tid] = elem;
   base_cnt_0+=num_lists;
  }
 }

 // Copy data back to source from the one's list
 for (u32 i=0; i<base_cnt_1; i+=num_lists)
 {
  sort_tmp[base_cnt_0+i+tid] = sort_tmp_1[i+tid];
 }
}
__syncthreads();
```

Optimizing the Code

Can we optimize the code?

- move bit mask operation out of the for-loop
- often done by compiler but not always
- called *variational analysis*

- also can re-use the ‘1’ list for source array
- eliminates the ‘0’ list and saves memory
- removes a copy operation

Device/Threads	1	2	4	8	16	32	64	128	256
GTX470	28.51	14.35	7.65	3.96	2.05	1.09	0.61	0.38	0.24
9800GT	42.8	23.22	12.37	6.41	3.3	1.78	0.98	0.63	0.4
GTX260	52.54	28.46	15.14	7.81	4.01	2.17	1.2	0.7	0.46
GTX460	21.62	11.81	6.34	3.24	1.69	0.91	0.51	0.31	0.21

CPU-Accelerated Merging of Lists

Needed frequently in parallel programming

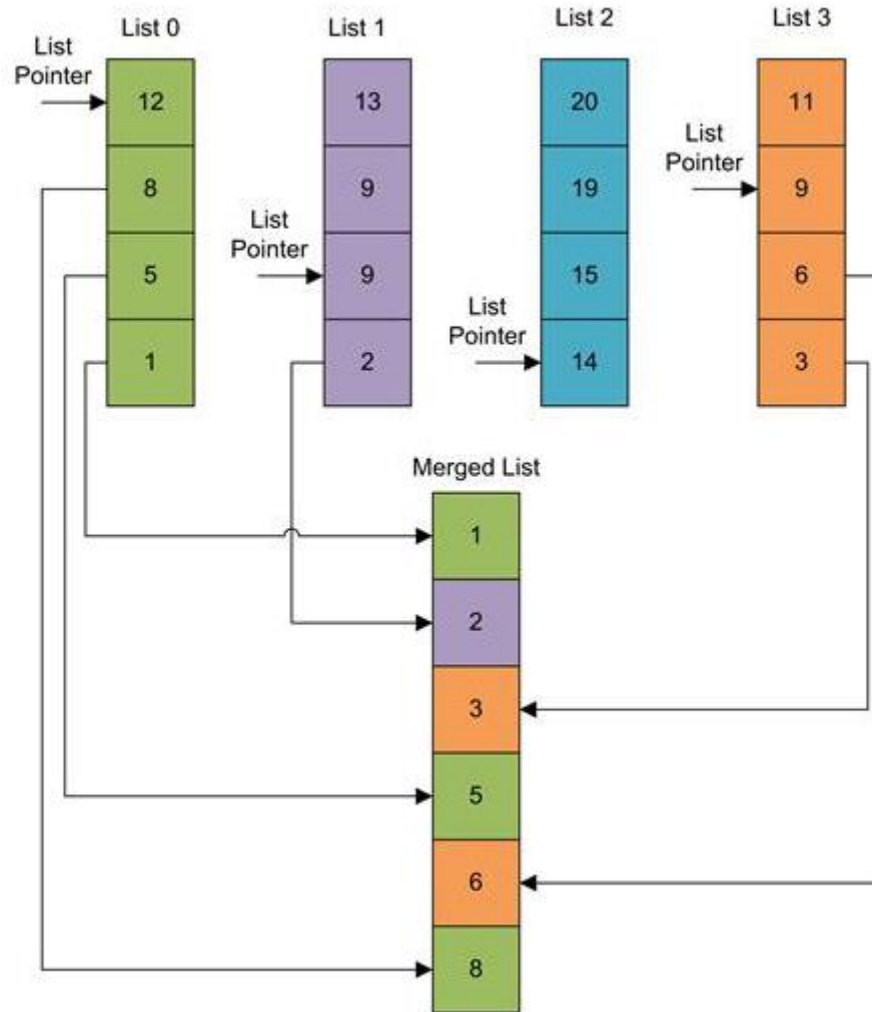
Examples:

- in Merge Sort
- for merging num_lists in Radix Sort

Merging Lists: Serial Code

```
void merge_array(const u32 * const src_array,
                 u32 * const dest_array,
                 const u32 num_lists,
                 const u32 num_elements)
{
       const u32 num_elements_per_list = (num_elements / num_lists);

       u32 list_indexes[MAX_NUM_LISTS];

       for (u32 list=0; list < num_lists; list++)
       {
        list_indexes[list] = 0;
       }

       for (u32 i=0; i<num_elements;i++)
       {
        dest_array[i] = find_min(src_array,
                                 list_indexes,
                                 num_lists,
                                 num_elements_per_list);
       }
      }
```

Merging Lists: Concept



Merging Lists

```
u32 find_min(const u32 * const src_array,
             u32 * const list_indexes,
             const u32 num_lists,
             const u32 num_elements_per_list)
{

   u32 min_val = 0xFFFFFFFF;
   u32 min_idx = 0;
// Iterate over each of the lists
for (u32 i=0; i<num_lists; i++)
{
 // If the current list has already been emptied
 // then ignore it
 if (list_indexes[i] < num_elements_per_list)
 {
  const u32 src_idx = i + (list_indexes[i] * num_lists);

  const u32 data = src_array[src_idx];

  if (data <= min_val)
  {
    min_val = data;
    min_idx = i;
  }
 }
}

list_indexes[min_idx]++;
return min_val;
}
```

Parallel Merge – Host Program

```
__global__ void gpu_sort_array_array(
 u32 * const data,
 const u32 num_lists,
 const u32 num_elements)
{
 const u32 tid = (blockIdx.x * blockDim.x) +
threadIdx.x;
 __shared__ u32 sort_tmp[NUM_ELEM];
 __shared__ u32 sort_tmp_1[NUM_ELEM];

      copy_data_to_shared(data,      sort_tmp,
num_lists,
                         num_elements, tid);

      radix_sort2(sort_tmp,       num_lists,
num_elements,
                 tid, sort_tmp_1);

 merge_array6(sort_tmp, data, num_lists,
              num_elements, tid);
}
```

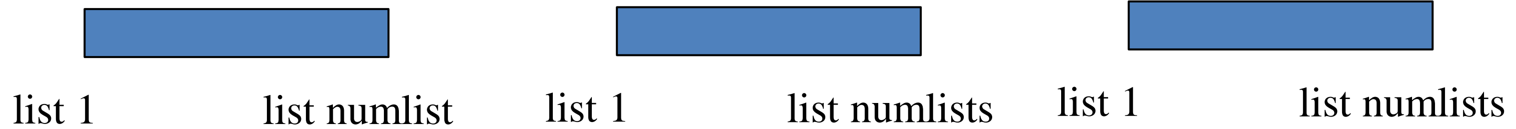
Parallel Merge – Load Data into Shared Memory

```
    __device__ void copy_data_to_shared(const u32
* const data,
                                    u32 *
const sort_tmp,
                                    const u32
num_lists,
                                    const u32
num_elements,
                                    const u32
tid)
    {
     // Copy data into temp store
     for (u32 i=0; i<num_elements; i+=num_lists)
     {
      sort_tmp[i+tid] = data[i+tid];
     }
     __syncthreads();
    }
```

Questions

Remember.

- How is the data laid out in memory?



Single Thread GPU Merge (1)

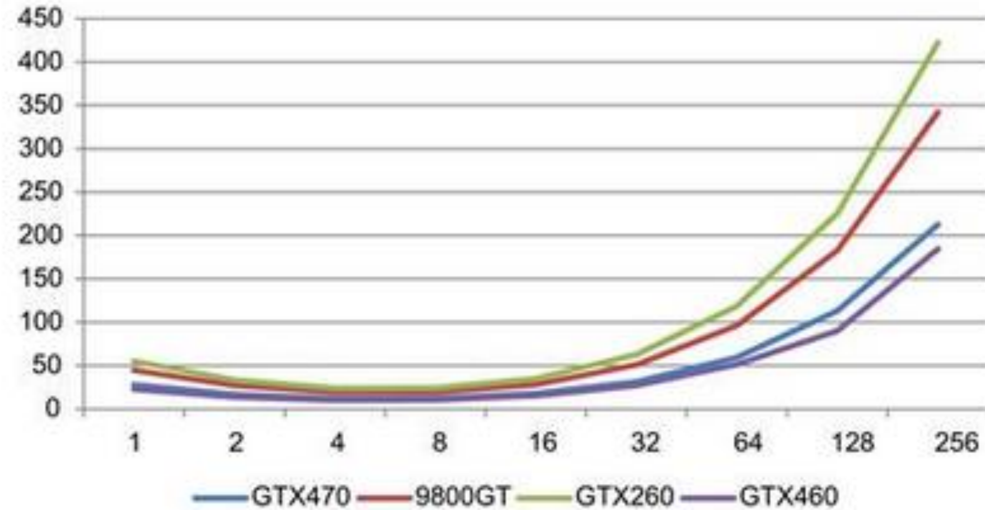
```
// Uses a single thread for merge
__device__ void merge_array1(const u32 * const src_array,
                             u32 * const dest_array,
                             const u32 num_lists,
                             const u32 num_elements,
                             const u32 tid)
{
 __shared__ u32 list_indexes[MAX_NUM_LISTS];

 // Multiple threads
 list_indexes[tid] = 0;
 __syncthreads();

 // Single threaded
 if (tid == 0)
 {
  const u32 num_elements_per_list = (num_elements / num_lists);

  for (u32 i=0; i<num_elements;i++)
  {
   u32 min_val = 0xFFFFFFFF;
   u32 min_idx = 0;
```

Single Thread GPU Merge (2)

```
// Iterate over each of the lists
for (u32 list=0; list<num_lists; list++)
{
    // If the current list has already been
    // emptied then ignore it
    if (list_indexes[list] < num_elements_per_list)
    {
     const u32 src_idx = list + (list_indexes[list] * num_lists);

     const u32 data = src_array[src_idx];
     if (data <= min_val)
     {
      min_val = data;
      min_idx = list;
     }
    }
   }
   list_indexes[min_idx]++;
   dest_array[i] = min_val;
  }
 }
}
```

Performance

Device/ Threads	1	2	4	8	16	32	64	128	256
GTX470	27.9	16.91	12.19	12.31	17.82	31.46	59.42	113.3	212.7
9800GT	44.83	27.21	19.55	19.53	28.07	51.08	96.32	183.08	342.16
GTX260	55.03	33.38	24.05	24.15	34.88	62.9	118.71	225.73	422.55
GTX460	22.76	13.85	10.11	10.41	15.29	27.18	51.46	90.26	184.54



GTX 260 slower than 9800GT, why?

Parallel GPU Merge (1)

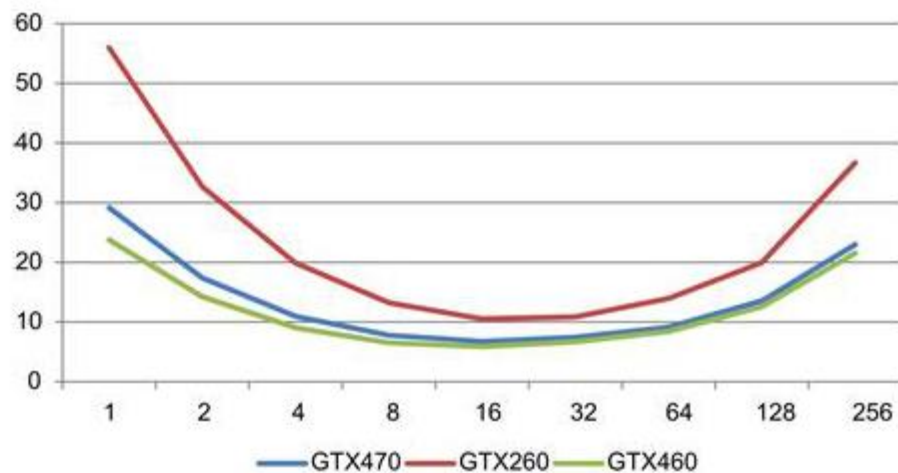
```
// Uses multiple threads for merge
// Deals with multiple identical entries in the data
__device__ void merge_array6(const u32 * const src_array,
                             u32 * const dest_array,
                             const u32 num_lists,
                             const u32 num_elements,
                             const u32 tid)
{
 const u32 num_elements_per_list = (num_elements / num_lists);

 __shared__ u32 list_indexes[MAX_NUM_LISTS];
 list_indexes[tid] = 0;

 // Wait for list_indexes[tid] to be cleared
 __syncthreads();

 // Iterate over all elements
```

```
for (u32 i=0; i<num_elements;i++)
{
 // Create a value shared with the other threads
 __shared__ u32 min_val;
 __shared__ u32 min_tid;

 // Use a temp register for work purposes
 u32 data;

 // If the current list has not already been
 // emptied then read from it, else ignore it
 if (list_indexes[tid] < num_elements_per_list)
 {
  // Work out from the list_index, the index into
  // the linear array
  const u32 src_idx = tid + (list_indexes[tid] * num_lists);

  // Read the data from the list for the given
  // thread
  data = src_array[src_idx];
 }
 else
 {
  data = 0xFFFFFFFF;
 }

 // Have thread zero clear the min values
 if (tid == 0)
 {
  // Write a very large value so the first
  // thread thread wins the min
  min_val = 0xFFFFFFFF;
  min_tid = 0xFFFFFFFF;
 }

 // Wait for all threads
 __syncthreads();

 // Have every thread try to store it's value into
 // min_val. Only the thread with the lowest value
 // will win
 atomicMin(&min_val, data);

 // Make sure all threads have taken their turn.
 __syncthreads();

 // If this thread was the one with the minimum
 if (min_val == data)
 {
  // Check for equal values
  // Lowest tid wins and does the write
  atomicMin(&min_tid, tid);
 }

 // Make sure all threads have taken their turn.
 __syncthreads();

 // If this thread has the lowest tid
 if (tid == min_tid)
 {
  // Incremene the list pointer for this thread
  list_indexes[tid]++;
```

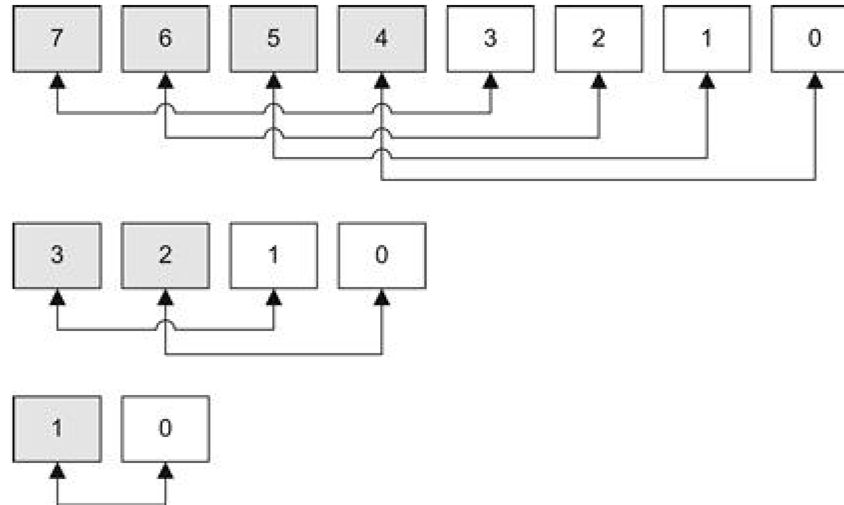
Parallel Merge (3)

```
      // Store the winning value
      dest_array[i] = data;
    }
  }
}
```

Device/Threads	1	2	4	8	16	32	64	128	256
GTX470	29.15	17.36	10.96	7.77	6.74	7.43	9.15	13.55	22.99
GTX260	55.97	32.67	19.87	13.22	10.51	10.86	13.96	19.97	36.68
GTX460	23.78	14.23	9.06	6.54	5.86	6.67	8.41	12.59	21.58



Reduction Approach for Merge



GPU Reduction (1)

```
// Uses multiple threads for reduction type merge
__device__ void merge_array5(const u32 * const src_array,
                             u32 * const dest_array,
                             const u32 num_lists,

                              const u32 num_elements,
                              const u32 tid)
 {
  const u32 num_elements_per_list = (num_elements / num_lists);

  __shared__ u32 list_indexes[MAX_NUM_LISTS];
  __shared__ u32 reduction_val[MAX_NUM_LISTS];
  __shared__ u32 reduction_idx[MAX_NUM_LISTS];

  // Clear the working sets
  list_indexes[tid] = 0;
  reduction_val[tid] = 0;
  reduction_idx[tid] = 0;
  __syncthreads();
```

GPU Reduction (2)

```
for (u32 i=0; i<num_elements;i++)
{
 // We need (num_lists / 2) active threads
 u32 tid_max = num_lists >> 1;

 u32 data;

 // If the current list has already been
 // emptied then ignore it
 if (list_indexes[tid] < num_elements_per_list)
 {
  // Work out from the list_index, the index into
  // the linear array
  const u32 src_idx = tid + (list_indexes[tid] * num_lists);

  // Read the data from the list for the given

  // thread
  data = src_array[src_idx];
 }
 else
 {
  data = 0xFFFFFFFF;
 }
```

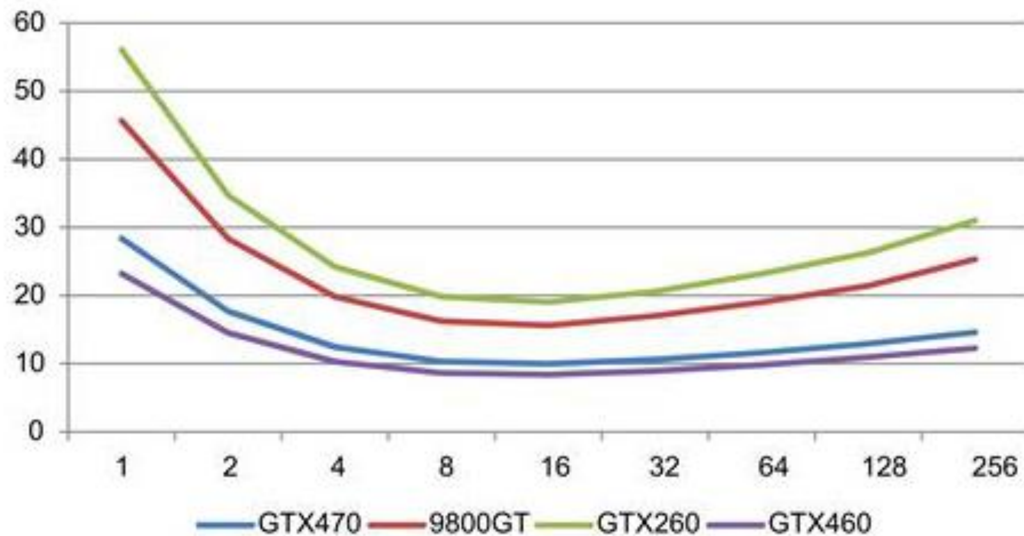
```
// Store the current data value and index
reduction_val[tid] = data;
reduction_idx[tid] = tid;

// Wait for all threads to copy
__syncthreads();
```

GPU Reduction (3)

```
// Reduce from num_lists to one thread zero
while (tid_max != 0)
{
 // Gradually reduce tid_max from
 // num_lists to zero
 if (tid < tid_max)
 {
  // Calculate the index of the other half
  const u32 val2_idx = tid + tid_max;

  // Read in the other half
  const u32 val2 = reduction_val[val2_idx];

  // If this half is bigger
  if (reduction_val[tid] > val2)
  {
   // The store the smaller value
   reduction_val[tid] = val2;
   reduction_idx[tid] = reduction_idx[val2_idx];
  }
 }

  // Divide tid_max by two
  tid_max >>= 1;

  __syncthreads();
 }

 if (tid == 0)
 {
  // Incremenet the list pointer for this thread
  list_indexes[reduction_idx[0]]++;

  // Store the winning value
  dest_array[i] = reduction_val[0];
 }

 // Wait for tid zero
 __syncthreads();
}
}
```

GPU Reduction

Device/Threads	1	2	4	8	16	32	64	128	256
GTX470	28.4	17.67	12.44	10.32	9.98	10.59	11.62	12.94	14.61
9800GT	45.66	28.35	19.82	16.25	15.61	17.03	19.03	21.45	25.33
GTX260	56.07	34.71	24.22	19.84	19.04	20.6	23.2	26.28	31.01
GTX460	23.22	14.52	10.3	8.63	8.4	8.94	9.82	10.98	12.27



Conclusions

AtomicMin code seems to be faster

- but only works for integers
- also only available for compute 1.2 and higher

Reduction is more general

Check out the hybrid atomicMin/reduction code in the book by Cook