CSE 591: GPU Programming

Memories: Registers and Shared

Klaus Mueller

Computer Science Department
Stony Brook University

Code examples from Shane Cook “CUDA Programming”
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
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
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)
- \( \frac{32,768}{4} \) 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-controlled* 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 load the variable from global memory directly
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
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
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
Next problem: the merging

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

Solution: recursion only down to 32-element sets

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

• shared memory storage is required
• need to assure bank conflict free layout
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
__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];
        }
    }
}
__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);
}
```c
__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

<table>
<thead>
<tr>
<th>sort_tmp</th>
<th>bank 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11</td>
<td>8 9 10 11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sort_tmp_0</th>
<th>bank 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 9 6 3</td>
<td>4 5 6 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sort_tmp_1</th>
<th>bank 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 1 2 11 8 5</td>
<td>8 5</td>
</tr>
<tr>
<td></td>
<td>4 1 2 11</td>
</tr>
</tbody>
</table>
Each thread computes *num_lists*

- ideally *num_lists* = warp size or a multiple
- will avoid bank conflicts
__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();
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
CPU-Accelerated Merging of Lists

Needed frequently in parallel programming

Examples:
- in Merge Sort
- for merging num_lists in Radix Sort
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

The diagram illustrates the concept of merging lists. Each list contains nodes with pointers to the next node. The merging process involves comparing the values of the nodes and connecting them in ascending order to create a single merged list.
Merging Lists

```c
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;
}
```
```c
__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);
}
```
__device__ void copy_data_to_shared(const u32 * const data,
const 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();
}
Remember.

- How is the data laid out in memory?
// 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;
// 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

GTX 260 slower than 9800GT, why?
// 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_indexed[tid] = 0;

    // Wait for list_indexed[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 = 0xFFFFFFFFF;
        min_tid = 0xFFFFFFFFF;
    }

    // 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)
    {
        // Increment the list pointer for this thread
        list_indexes[tid]++;
    }
// Store the winning value
dest_array[i] = data;

<table>
<thead>
<tr>
<th>Device/Threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX470</td>
<td>28.15</td>
<td>17.36</td>
<td>10.96</td>
<td>7.77</td>
<td>5.74</td>
<td>7.43</td>
<td>9.15</td>
<td>13.55</td>
<td>22.89</td>
</tr>
<tr>
<td>GTX260</td>
<td>50.87</td>
<td>32.67</td>
<td>19.87</td>
<td>13.22</td>
<td>10.51</td>
<td>10.36</td>
<td>13.96</td>
<td>19.97</td>
<td>38.68</td>
</tr>
<tr>
<td>GTX460</td>
<td>58.78</td>
<td>44.23</td>
<td>29.06</td>
<td>16.54</td>
<td>12.98</td>
<td>12.87</td>
<td>16.41</td>
<td>22.53</td>
<td>41.68</td>
</tr>
</tbody>
</table>

![Graph showing performance comparison for GTX470, GTX260, and GTX460]
Reduction Approach for Merge
// 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();
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();
// 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];
        }
    }
    if (tid == 0)
    {
        // Increment 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();
}
// Divide tid_max by two
tid_max >>= 1;
__syncthreads();
## GPU Reduction

<table>
<thead>
<tr>
<th>Device/Threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>9800GT</td>
<td>48.66</td>
<td>38.36</td>
<td>19.82</td>
<td>16.25</td>
<td>16.61</td>
<td>17.03</td>
<td>19.03</td>
<td>21.45</td>
<td>25.83</td>
</tr>
<tr>
<td>GTX260</td>
<td>56.07</td>
<td>34.71</td>
<td>24.22</td>
<td>19.84</td>
<td>19.04</td>
<td>20.5</td>
<td>23.2</td>
<td>28.28</td>
<td>31.01</td>
</tr>
<tr>
<td>GTX460</td>
<td>23.22</td>
<td>14.62</td>
<td>10.3</td>
<td>8.83</td>
<td>8.4</td>
<td>8.94</td>
<td>9.82</td>
<td>10.98</td>
<td>12.27</td>
</tr>
</tbody>
</table>

- The table above shows the reduction in performance for different GPU models with varying device/thread combinations.
- The graph below visualizes the performance trend across different models and thread counts.

![Graph showing performance trends](image-url)
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