Parallel reduction

This can be applied for many problems, a min operation being just one of them. It works by using half the number of threads of the elements in the dataset. Every thread calculates the minimum of its own element and some other element. The resultant element is forwarded to the next round. The number of threads is then reduced by half and the process repeated until there is just a single element remaining, which is the result of the operation.
With CUDA you must remember that the execution unit for a given SM is a warp. Thus, any amount of threads less than one warp is underutilizing the hardware. Also, while divergent threads must all be executed, divergent warps do not have to be.

When selecting the "other element" for a given thread to work with, you can do so to do a reduction within the warp, thus causing significant branch divergence within it. This will hinder the performance, as each divergent branch doubles the work for the SM. A better approach is to drop whole warps by selecting the other element from the other half of the dataset.
In Figure 6.12 you see the item being compared with one from the other half of the dataset. Shaded cells show the active threads.

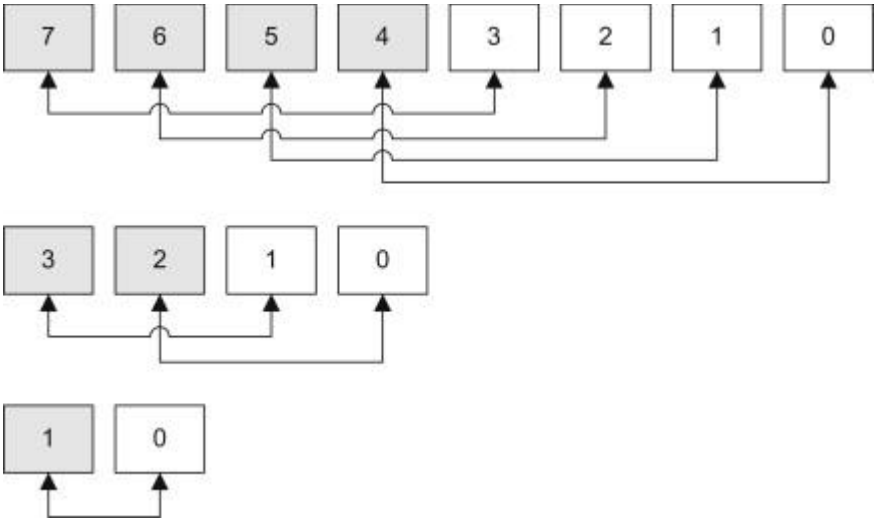

Figure 6.12. Final stages of GPU parallel reduction.

```
// 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];
  }
 }

 // 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();
 }
}
```

This code works by creating a temporary list of data in shared memory, which it populates with a dataset from each cycle from the num_list datasets. Where a list has already been emptied, the dataset is populated with 0xFFFFFFFF, which will exclude the value from the list. The while loop gradually reduces the number of active threads until there is only a single thread active, thread zero. This then copies the data and increments the list indexes to ensure the value is not processed twice.