RADIX SORT

Radix sort

- It has a fixed number of iterations and a consistent execution flow.
- It works by sorting based on the least significant bit and then working up to the most significant bit.
- With a 32-bit integer, using a single radix bit, you will have 32 iterations of the sort, no matter how large the dataset.
- example : { 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

- In the first pass, all elements with a 0 in the LSB would form the first list.
- Those with a 1 as the LSB 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 }

Radix sort

- The program continues until it has processed all 32 bits of the list in 32 passes.
- To build the list you need **N + 2N** memory cells.
- one for the source data, one of the **0 list**, and one of the **1 list**.

Radix sort: Serial 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;
```

Radix sort: Serial code

```
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++;
        }
```

Radix sort: Serial code

```
        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];
        }
```

Radix sort: Serial code

```
// 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];
        }
}
}
```

Radix sort: Serial code

- The code works by being passed two values, a pointer to the data to sort and the number of elements in the dataset.
- It overwrites the unsorted data so the returned set is sorted.
- The outer loop iterates over all 32 bits in a 32-bit integer word and the inner loop iterates over all elements in the list.
- The algorithm requires 32N iterations in which the entire dataset will be read and written 32 times.
- Within the inner loop the data is split into two lists,
  - the 0 list and the 1 list depending on which bit of the word is being processed.
  - The data is then reconstructed from the two lists, the 0 list always being written before the 1 list

Radix sort: Serial code

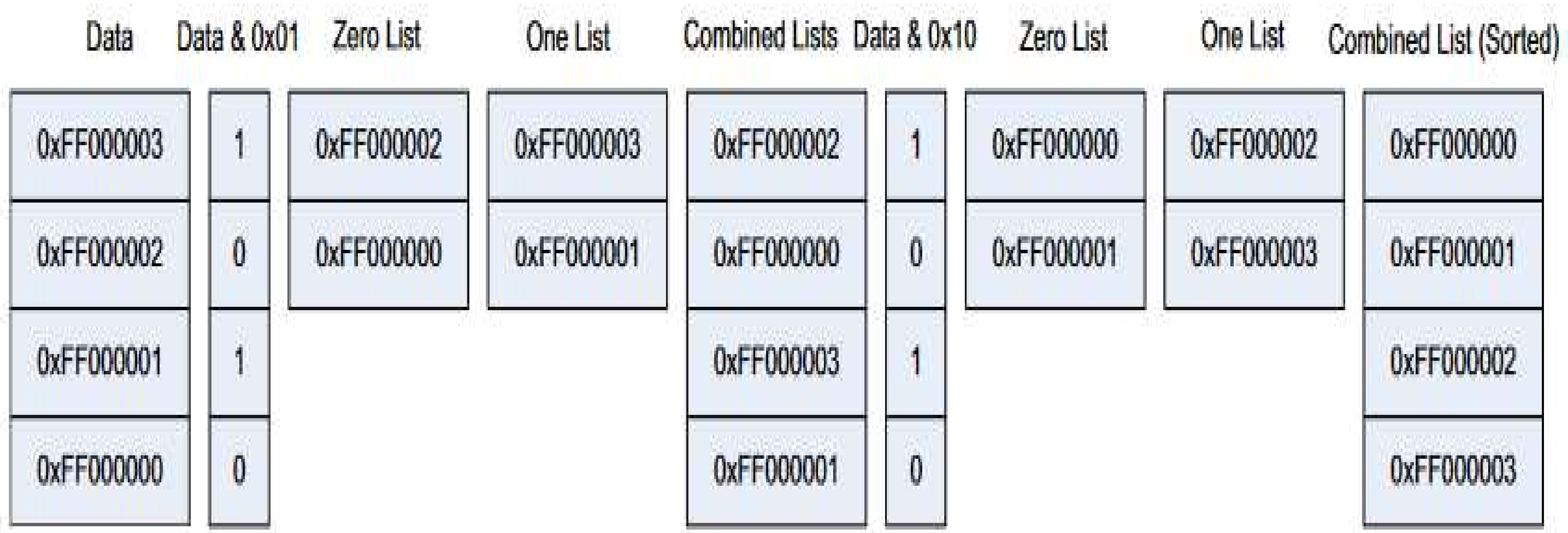

Fig: Radix Sort

Radix Sort: GPU Code

- The GPU version is a little more complex, in that we need to take care of multiple threads.

```
__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);
```

Radix Sort: GPU Code

```
        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
```

Radix Sort: GPU Code

```
        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();
}
```

Radix Sort: GPU Code

- The GPU kernel is written here as a device function, a function only capable of being called within GPU Kernel.
- The inner loop has changed and instead of incrementing by one, the program increments by num_lists a value passed into the function.
- This radix sort will produce is num_lists of independent sorted lists using num_lists threads.
- Since the SM in the GPU can run 32 threads at the same speed as just one thread and it has 32 shared memory banks, you might imagine the ideal value for num_lists would be 32.

Radix Sort: GPU Code

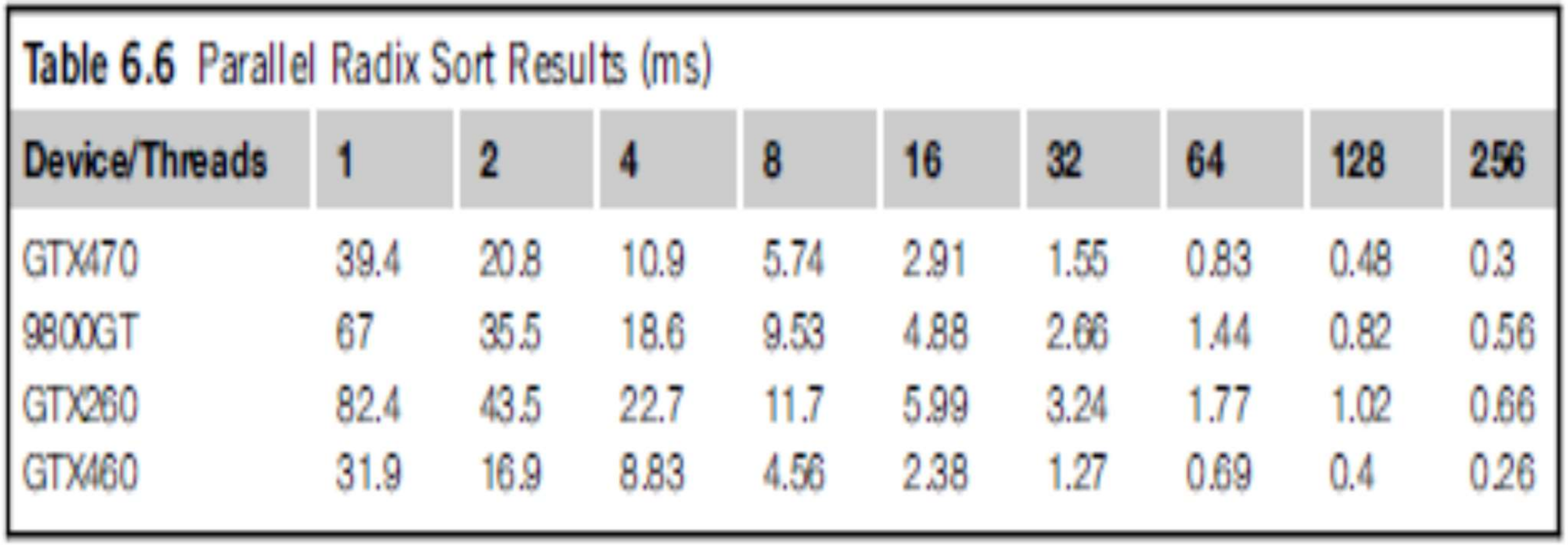

Table 6.6 Parallel Radix Sort Results (ms)

Device/Threads	1	2	4	8	16	32	64	128	256
GTX470	39.4	20.8	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	0.26

Fig: Parallel Radix sort results

Radix Sort: Optimized GPU Code

- Do not need separate 0 and 1 lists.
- The 0 list can be created from reusing the space in the original list.
- This not only allows you to discard the 1 list, but also removes a copy back to the source list.
- This saves a lot of unnecessary work.

Radix Sort: Optimized GPU 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 )
```

Radix Sort: GPU Code

```
{
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;
}
}
```

Radix Sort: Optimized GPU Code

```
// 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();
}
```

Radix Sort: Optimized GPU Code

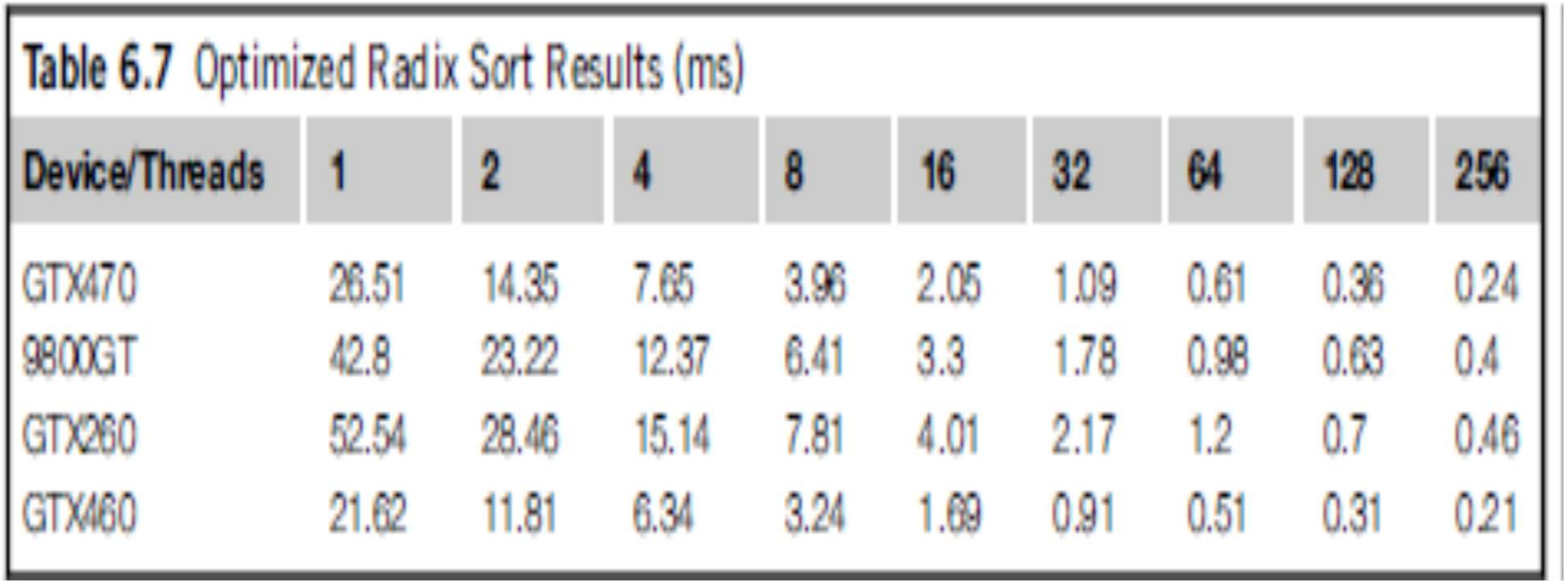

Table 6.7 Optimized Radix Sort Results (ms)

Device/Threads	1	2	4	8	16	32	64	128	256
GTX470	26.51	14.35	7.65	3.96	2.05	1.09	0.61	0.36	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

Fig:Optimized Radix Sort Results