Introduction to GPU Radix Sort

Takahiro Harada
Advanced Micro Devices, Inc.

Lee Howes
Advanced Micro Devices, Inc.

1 Radix Sort

Radix sort is one of the fastest sorting algorithms. It is fast especially for a large problem size. Radix sort is not a comparison sort but a counting sort. When we sort n bit keys, 2^n counters are prepared for each number. Let me explain how radix sort works by showing a simple example sorting 2 bits. Here is the input for the sort.

$$(0, 3, 2, 2, 3, 2, 0, 3, 2, 1)$$

In order to sort them, an offset table for each number is calculated. It first counts the occurrences of each number. Counts are

$$(c_0, c_1, c_2, c_3) = (2, 1, 4, 3)$$

Taking the prefix sum of the counts transforms it to an offset table.

$$(o_0, o_1, o_2, o_3) = (0, 2, 3, 7)$$

The prefix sum is the sum of all values in preceding locations in the sequence: in this case those to the left of the current location. In the case of the radix sort this means that the prefix sum computes the total count of all values less than the current value. For example, the prefix sum of location, and hence value, 2 is $o_2 = 3$. This means there are 3 entries for $0s$ and $1s$ in the sequence. Thus, 3 is the destination address of the first 2 in the data set. The destination address of an element is the sum of the offset computed via the prefix sum and the index of the value in the set of the same value in the original array: the second 2 in the array would be at location $3 + 1$. The elements are shuffled by calculating the destination address to get a sorted array.

$$(0, 0, 1, 2, 2, 2, 2, 3, 3, 3)$$

This is the case for 2 bit numbers. When we sort larger numbers, the table size can be huge. For example, when 32 bit numbers are sorted, 2^{32} entries are necessary for the table. In order to avoid using such a large table, sorting is performed by chunking the numbers into smaller units and applying the sorting algorithm several times. If n bits are sorted at once we have a radix 2^n sort and $32/n$ passes are necessary to complete the sort. It might seem at first that sorting the keys chunk-by-chunk ignores the result of previous passes. However, the sort is stable, which means that any two values that are viewed as the same by a given stage of the sort will be output in unchanged order. If we are sorting bits 0-4 or two different 32-bit integers and only bit 6 differs then the two values are seen as the same and the order will not change. By sorting from the least significant bits to the most significant bits the stability of the algorithm as a whole is maintained.

To summarize the algorithm, each pass over the data (and each new chunk to sort) requires three steps:

1. Count
2. Prefix scan
3. Reorder

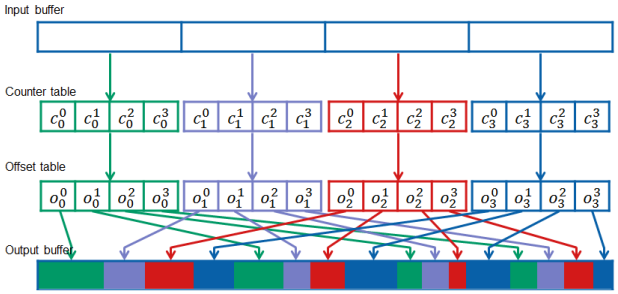

Figure 1: *Parallel radix sort using four threads.*

2 Parallel Radix Sort

When four threads are used for a radix sort, the input buffer is split into four sections, each of which is assigned to one of the threads. At first, each thread counts the occurrence of each value of the subkey in its section. In order to find the destination address of a key in a section it only needs that value, but to find the destination address of a key in the global data set it needs not only the count in its section but also the corresponding values from the other sections. Let c_m^n be the number of instances of value n in section m, the offset o_m^n of value n in section m is

$$o_m^n = \sum_{i=0,i<n} \sum_{j=0,j<4} c_j^i + \sum_{j=0,j<m} c_j^n. \quad (1)$$

If the count values are placed in a one-dimensional array in column major layout then Eqn. 1 translates to a computation of the prefix sum over the array. Then the thread works again for the assigned section to reorder it in the output buffer. Fig. 1 is an illustration of a parallel radix sort using four threads.

3 Implementation on the GPU

Because of the wide vector architecture of the GPU (64 wide SIMD on AMD GPUs), utilizing all the SIMD lanes is important. We use OpenCL's terminology for the following explanation. A work group is the unit of work processed by a SIMD engine and a work item is the unit of work processed by a single SIMD lane (sometimes confusingly termed a "core" by manufacturers). The earlier explanation of the algorithm assumed that it was using a single lane of a SIMD unit, or a single CPU core ignoring its vector instructions. Unfortunately, if a program is written to execute a single work item in each work group it is highly inefficient because it uses only one lane of the SIMD unit. On an AMD GPU whose SIMD width is 64, it is using only $1/64$ of the ALUs. Executing the number of work items equal to the SIMD width is necessary to fully utilize the wide SIMD architecture. Thus we need another level of parallelization in a work group[1].

The basic strategy is the same as that described in Sec. 2. It takes 3 steps to complete a pass. Each step performed by a kernel is detailed in the following subsections. In order to use all of the SIMD

[1] In theory the same approach could be mapped to CPU SIMD units but the lack of scatter/gather operations in current CPUs makes this difficult.

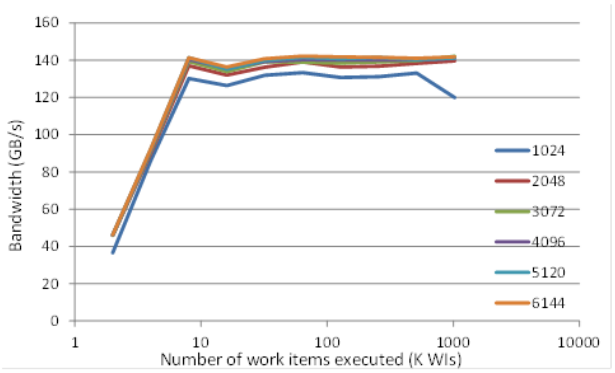

Figure 2: *The effective bandwidth on AMD Radeon HD5780 (Theoretical peak bandwidth is 153.6GB/s).*

engines (each corresponding to cores on the CPU), the input buffer is split into sections and a work group is created for each section. In the following discussion we describe how to split a work group into work items to exploit the wide SIMD architecture.

Another point we have to keep in mind is the number of work groups to execute. On a four core CPU without in-core multithreading executing more than four threads would oversubscribe the system, requiring the operating system to consume resources switching between threads. In the radix sort, one drawback of executing more threads than necessary is that it creates more offset table data. When sorting a large number of keys this means that more memory is consumed and the inter-thread communication phase of the computation takes significantly longer. Thus executing the minimum number of threads on the CPU is the better. The same applies to the GPU but as GPU devices employ in-core multithreading to hide latency (multiple wavefronts are scheduled concurrently on a given SIMD engine) the number is somewhat higher than the number of cores: usually a factor of 4 to 8. We empirically deicide the number of work groups we need to execute. Fig. 2 shows the memory bandwidth and number of work items executed on AMD Radeon HD5870. You can see it can saturate memory read before executing 10K work items. The size of a section to be processed by a work group is calculated from work group size and number of work groups we executes.

There are several works on radix sort on the GPU[Satish et al. 2009][Ha et al. 2009]. Our implementation is based on Merrill *et. al.*[Merrill and Grimshaw 2010].

3.1 Count

The input is an array of keys and the output is a table of counts of each value in the input. A straightforward implementation is to use a work item to read a key from the global memory and increment the counter value of the key on the global memory using an atomic increment operation. Then the memory read window is moved and the process repeated until all keys are consumed. However, it is not efficient because global atomics are expensive compared with other operations.

An obvious optimization is to move the count table into the local data share (LDS) or OpenCL local memory in order to reduce the cost of the atomic operations. We can push it further by preparing a count table for each work item such that multiple work items do not try to access to the same memory locations. After processing all the elements we will have n count tables where n is the number of work items executed. As each count table is a table for a subsection

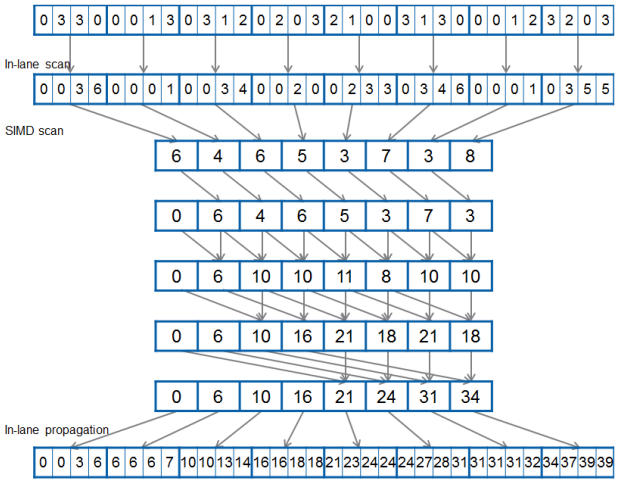

Figure 3: *Parallel scan algorithm.*

of the input, the count table for the entire input is the summation of the individual tables. We use a work item to sum up all the count value of the tables and write it to the global memory.

3.2 Scan

As the number of work groups executed is small, single work group is used to calculate the prefix sum. If the number of items to be scanned is smaller than the vector length, a simple parallel scan (Hillis and Steele) is the fastest as it does not require any work item synchronization although it is not the most work efficient approach. In the case where the scan size is bigger than the SIMD width, we can use an in-lane scan and propagation before and after the vector scan as shown in Fig. 3.

3.3 Reorder

The last step is to write the data back to the appropriate location in global memory. In order to coalesce the memory writes as much as possible the keys are sorted in the LDS. During the sorting the kernel performs scattered writes into LDS, but this memory is designed for efficient random access. Global memory hosted in DRAM is far less suited to scattered writes than LDS. Once the data is sorted, the destination address of value n is calculated by

$$d_i = i - lo^n + o_m^n \tag{2}$$

where i is the local index, lo^n is the local offset, and o_m^n is the global offset of the value n processed by work group m.

For sorting in a work group the radix sort algorithm is used as it is the most efficient sort given a perfect memory system. There are two options for the method to sort n bits: use a 1 bit sort n times, or to use an n bit sort once. Each approach has pros and cons. An advantage of the former option is that it needs 1 counter but it has to repeat it n times. The rank of an element is efficiently calculated by using the vector wide scan as described above when the counter value is 0 or 1. However, because of the serial nature of radix sort, each pass (1 bit sort and data exchange) has to be processed in the order of the bits. A drawback of the latter is that it requires 2^n counters but an advantage is that the scan of each counter is independent so they can be scanned in parallel. If the total number of inputs is less than 256, the counter value does not exceed 256, which means the counter can be encoded with an 8 bit value. We can pack 4 counters in a 32 bit value and still scan

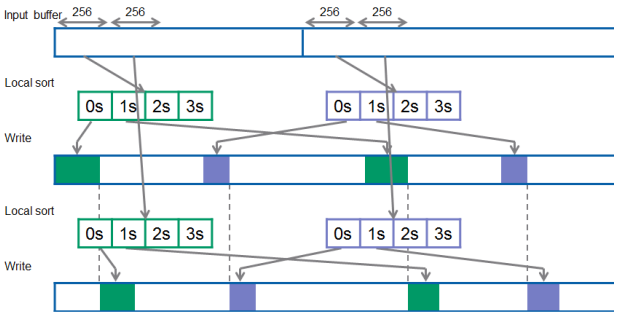

Figure 4: *The reordering step. It illustrates two steps of a local sort and a write. A SIMD engine reads 256 keys and sorts locally and write them back to memory.*

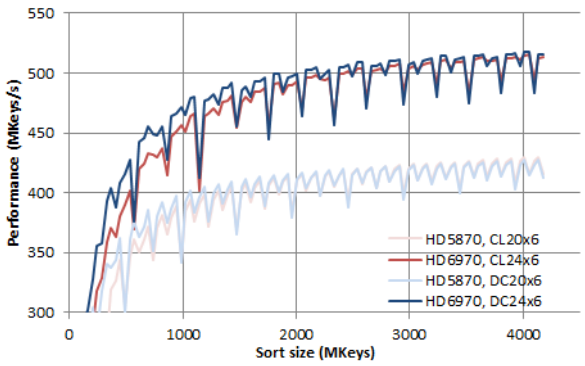

Figure 5: *The radix sort performance on AMD Radeon HD5870 and HD6970 using OpenCL and DirectCompute.*

only once. When a 2 bit radix sort is used, the former requires 2 scans and 2 data exchanges while the latter does 1 scan (4 scans are packed into one) and 1 data exchange. So it is obvious that the latter approach is the better if the cost of packing and unpacking of the data is ignored. Thus for our implementation we used the latter implementation for the local sort.

Because the maximum value of the counter is 256 to enable use of the word packing we need to restrict the number of elements we try to accumulate. We can restrict the size of the work group and the number of elements each work item processes to achieve this restriction. We set the work group size, or the number of work items in each work group, to 64. This size is equal to the vector length of AMD's GPUs. Each work item reads and processes 4 elements from global memory, encodes to 8 bit packed format, and adds them together. The sum is stored in LDS and the vector wide scan is performed once. The scanned value is the local offset of the 4 elements and is used to calculate the local offset of each element.

The input data is shuffled using the local offsets and output at the destination address. As we used 4 bits per pass this process is repeated twice to sort 8 bits. In this way, it sorts and writes 256 elements at a time and this process is repeated until all the elements in the section are reordered. Because it does not sort the entire input dataset at once, we have to keep track of the number of elements in each radix stored to the global memory. It is achieved by counting the occurrence for each radix again for each set of 256 elements and accumulating it in LDS. Fig. 4 illustrates the reordering step.

4 Performance

The sorting kernel was implemented using OpenCL and DirectCompute. Performance of the sort measured on a PC with Phenom IIX6110T CPU and AMD Radeon HD5870 and AMD Radeon HD6970 is shown in Fig. 5. They reach 520MKeys/s when the size of the input is large enough.

References

HA, L., KRUGER, J., AND T.SILVA, C. 2009. Fast 4-way parallel radix sorting on gpus. *Computer Graphics Forum*, 28, 2368–2378.

MERRILL, D., AND GRIMSHAW, A. 2010. Revisiting sorting for gpgpu stream architectures. *Online*, February, 1–17.

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IEEE Computer Society, Washington, DC, USA, 1–10.