Parallel Radix Sort on OpenCL

Hennadiy Yatskov
Nico Mürdter

Karlsruhe Institute of Technology, Karlsruhe, Germany
hennadiy.yatskov@student.kit.edu
nico.muerdter@student.kit.edu

Abstract. Parallel sorting algorithm can increase the performance of lots of applications tremendously. This draft shows a rather simple implementation of Radix-Sort on OpenCL where it can achieve a high speedup providing a good sorting method for integers of any kind.

1 Algorithm

This algorithm is based on an open source implementation of P. Helluy [2]. Before we come to the detailed explanation of the algorithm, we first need to define several values. $K(j)$, represents the non-sorted input values which consist only of integers. The implementation of P. Helluy is only capable of sorting 32-bit unsigned integers. So in his implementation, each of these integers in $K(j)$ for $j = 0 \dots N-1$ is positive and will be called a *key* in the further explanation. To expand the usage of this algorithm, this draft explains the expansion to support also 64-bit unsigned integers, and signed 32 and 64-bit integers. Because of the fact, that Radix-Sort is a non-comparative integer sorting algorithm, this implementation will not support floating point numbers. In order to allow operation with signed integers, an offset is added to the integers. This offset is in case of a signed input list, the absolute value of the minimum possible value of the given integer. In case of a 32-bit signed integer this would be $abs(-2^{31})$. The GPU kernels are still working on the unsigned representation of the given input data type, and the offset incrementation is done on the fly, when the key is first accessed. Also the decrementation at the end of the algorithm is done on the fly to prevent performance losses. So basically from the outside it seems the algorithm is working directly on the signed integer values, but on the inside it is working with a shifted number range to fit into the unsigned number range.

The Radix is represented by $R = 2^r$ where r is the number of bits necessary representing R. We suppose that b, the total number of bits representing the keys is divisible by r and so the number of passes is denoted by $p = b/r$. This assumption can be satisfied by an appropriate definition of r. Each pass $q = 0 \dots p-1$ of the algorithm is sorting the list according to the q^{th} digit (in base R) $K(j)_q$. A pass can be imagined as a reordering of the list by this q^{th} Radix. So after $p = b/r$ passes the list is sorted totally. It is important to say, that each pass sorts the corresponding elements in a stable manner. So the work done

in each pass is not corrupted by previous passes and will not corrupt following passes.

The final algorithm is seperated into three different phases, which are executed consecutively in every pass. These phases are called Histograming, Scanning and Reordering. The explanation is listed below.

1.1 Histograming

This first phase of the algorithm is in charge of calculating key *histograms*. A histogram represents the number of occuring Radix in the given list of elements K. To do this fully parallel, the porcessing units are seperated in Groups G with Items I, where an Item represents a Processor. So based on the GPU Architecture, the total amount of available Processing Units is GI. For the explanation we can suppose, that $N = GI$. If this is (most likely) not the case in a real scenario, the data is extended by adding keys which represent the biggest possible value. Each Item is in charge of a part of the list. It computes the Histogram of its own sublist and adds his histogram to the histogram of the Group. This is simply done by just inspecting every element in the list, computing every q^{th} bit sequence with the length r and then incrementing the corresponding entry in the local Histogram, which represents the computed Radix.

1.2 Scanning

The next step in this algorithm is the so called Scanning. In this part the calculated Histograms are collected over a Prefixsum algorithm. P. Helluy [2] and also the here shown improved implementation is using the Blelloch algorithm for parallel scans [1]. After the Prefixsum, every Processor knows in his own Histogram, how many elements according to the current Radix are smaller. This information given by the Prefixsum, allows a reordering based on the calculated Histograms from the previous section.

1.3 Reordering

To finally reorder the elements, the Prefixsum of the Histograms is needed. As described in [2]: *after the prefix sum, the histogram $H(c, g, i)$ contains the position, in the ordered list, of the first key with radix c that is examined by the item i in the group g.* So the Histogram can than be used to reorder the elements acording to the Radix of the corresponding pass. To do this, every Processor once again looks at the Radix of the current pass of every element, and then moves the element to the new position according to the Histogram. Afterwards the entry in the Historgam is increased by 1 for the next element with the same Radix. This reordering can also be done fully parallel.

2 Implementation Details

In this chapter the focus lies on the implementation. The basecode is written for OpenCL 1.2 and the base features like compiling the OpenCL code is inspired by the code base of the GPGPU practicum at the KIT.

The main routine focuses on instantiating buffers and delegating the computational tasks to the GPU processors. The three main parts, shown in section 1, are represented by so called OpenCL kernels. For each of these described tasks one kernel is in charge. Because a GPU is a PRAM architecture, all processors are working on a common main memory. So the main difficulty here was to effectively seperate the memory each Processor is working on to avoid data corruption. To achieve this the kernels made use of the standard OpenCL routines to get to know what their operating ranges are on the main memory. This restriction also allows to use the OpenCL *restrict* keyword, which gives the compiler a hint, that the marked buffers' adress spaces do not overlap. Therefore, the compiler is able to increase the overall performance by making use of cache optimizations.

To ensure the usage of different datatypes on the same codebase, this implementation is heavily based on the concept of C++ templates. The whole sorting algorithm can be instantiated with one of 4 supported data types: int32 / int64 / uint32 / uint64. This makes it easy to use this implementation for further projects without the need to accept performance losses if only 32-bit integers are of interest.

These type restrictions can easily be handled in C++ but for OpenCL we needed a more special method. Usually, the codebase of the OpenCL kernels is loaded during runtime and then compiled for the local GPU. Right before the compilation the code is imported with appended prepocessor #define directives which set the internal data types to either 32 or 64 bit unsigned integer and the *offset* shown in section 1 to the appropriate value, which depends on whether the input data type is signed or unsigned.

Another notable implementation detail is the usage of local memory and barriers within the OpenCL code. The processors need to be aligned after several operations like copying the histograms to the local memory. To avoid data corruption these actions are controlled by memory barriers. The usage of the local memory of every processor leads once again to an overall significant performance increase as it allows for a considerable boost in access time and can be, broadly speaking, regarded as a cache.

3 Experimental Results

All experimental benchmarks were made on a NVIDIA GeForce GTX 680 and a Intel i7-3770K CPU. GPU Driver Version 361.91 with OpenCL 1.2. For comparison the data was also sorted with std::sort and a custom single threaded Radix-Sort implementation on the CPU. The test data started at an input amount of 2^{13} elements and ended at 2^{25} elements. To test different scenarios every input consisted of 5 different datasets:

1. **Zeros**
   Dataset containing only zeros

2. **Random**
   Dataset containing random generated integers by a mersenne twister

3. **RandomDistributed**
   Dataset containing random generated integers by a uniform distribution random generator

4. **Range**
   Dataset containing a sorted range of integers beginning by the minimum value of the used datatype

5. **InvertedRange**
   Dataset containing the integers from range in inverted order

Below, there are the charts of benchmarks and speedup of the zero values and the uniform distributed random values. All these benchmarks were made with uint64 integers and were executed on the machine mentioned above. The experimental results are illustraded by these two input groups. As can be seen in figure 1 the runtime of the GPU algorithm on only zero values is much higher than the runtime of the standard C++ sorting algorithm `std::sort`. This behavior is explainable by the behaviour of `std::sort` and Radix-Sort itself. While Radix-Sort is always going through every bucket, the quicksort algorithm of `std::sort` can be optimized to recognize such inputs, consisting of identical elements, before starting the whole sorting process. Therefore the overhead of copying the input to the GPU and executing the whole algorithm on this pre-sorted dataset is inefficient. As expected the speedup of GPU Radix-Sort is below 1 which can be seen in figure 2. For other inputs like uniform distributed random integers, the runtime of GPU Radix-Sort increases significantly as shown in figure 3 and reaches a speedup up to a factor of 25, which can be seen in figure 4. The growing curve promises an even better performance for higher problem sizes above 2^{25} elements. All other inputs described above, behave the same way. For more details about the average execution times of the used algorithm parts figure 5 can be regarded.

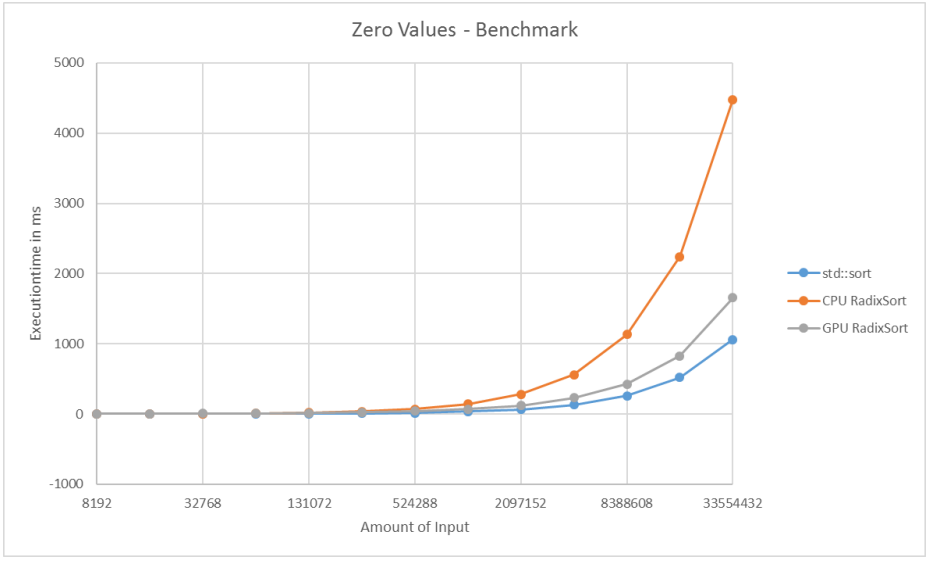

Fig. 1. Benchmark of only zero values.

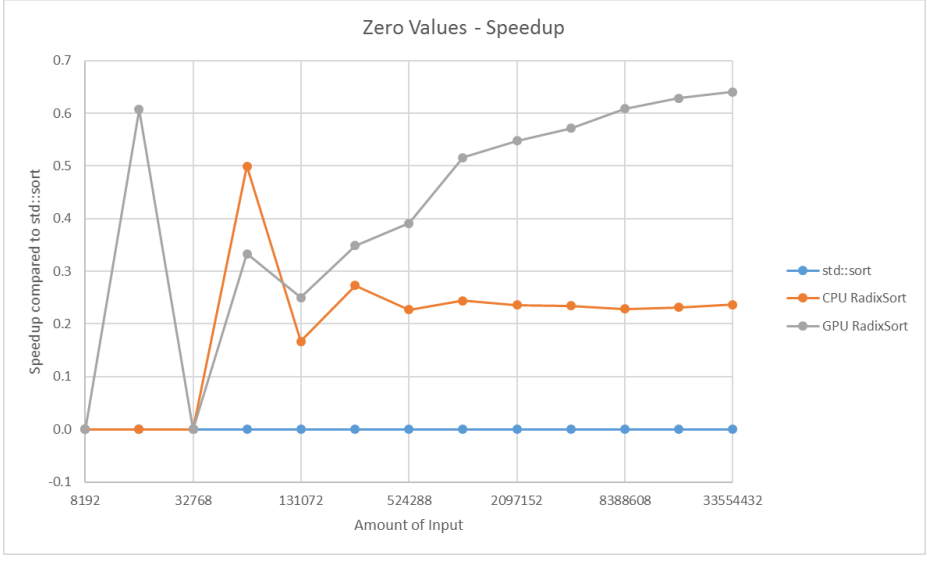

Fig. 2. Speedup of only zero values compared to std::out.

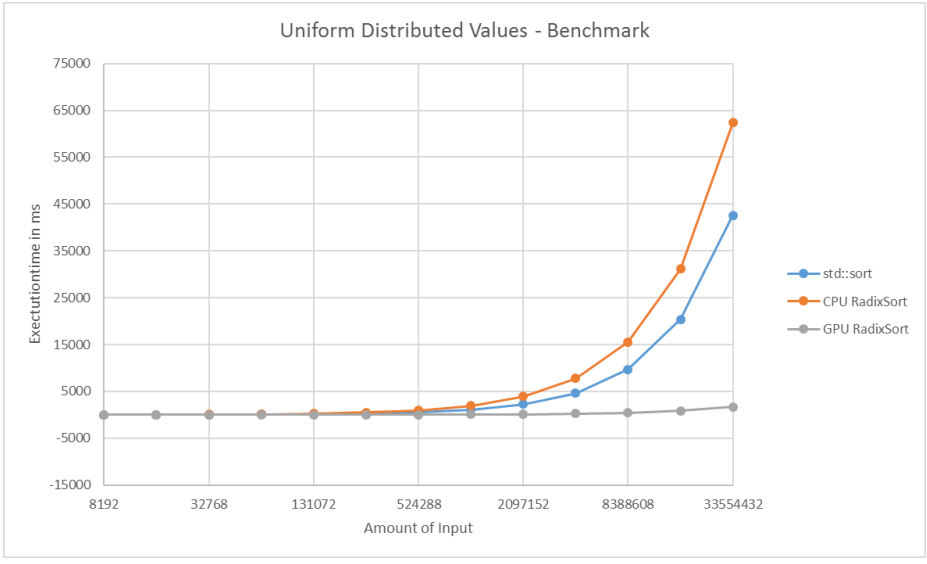

Fig. 3. Benchmark of uniform distributed random values.

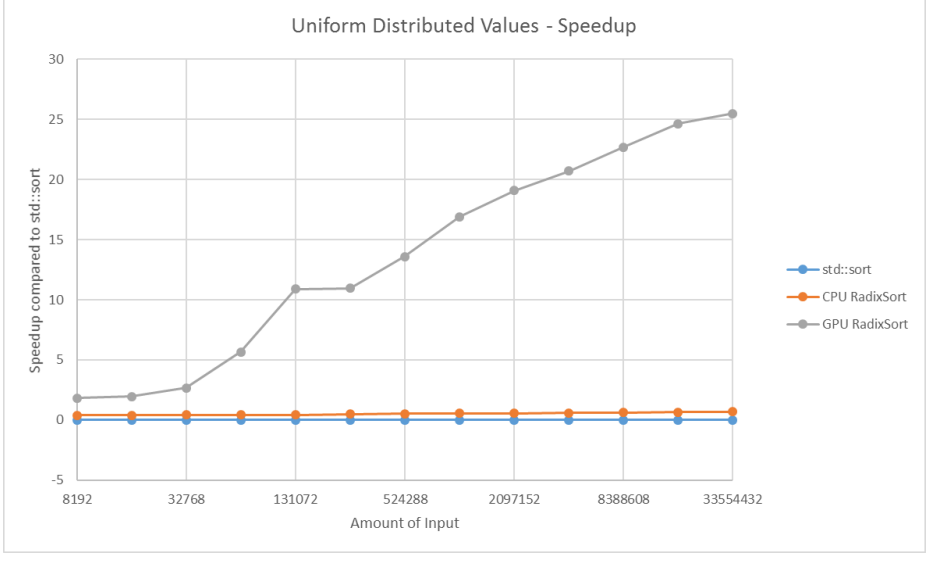

Fig. 4. Speedup of uniform distributed random values compared to std::out.

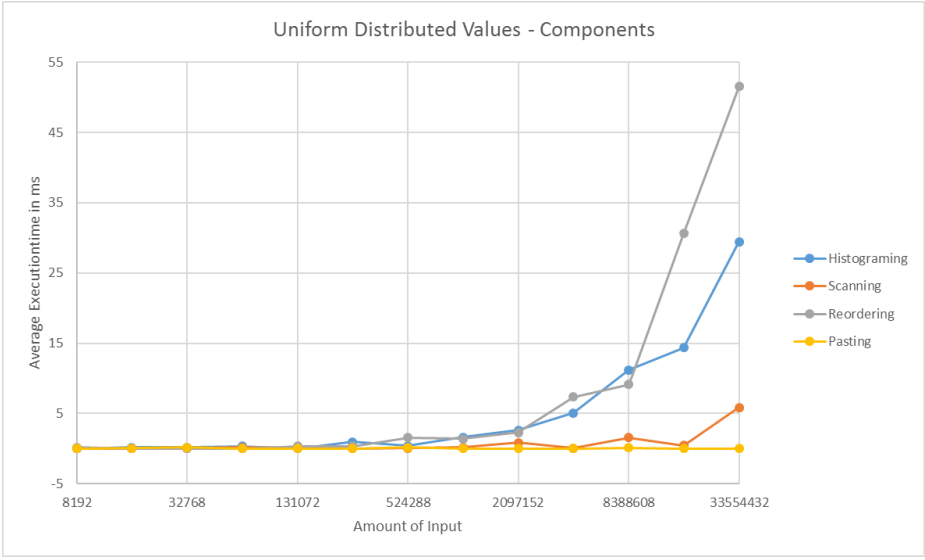

Fig. 5. Average runtimes of the components of the GPU Radix-Sort algorithm.

4 Usage

Below, there is a short list of command line options to run this implementation. All tests are automatically executed on all dataset configurations on all supported data types.

`--num-elements`
Number of elements to be sorted.
`--perf-to-stdout`
Print all measured performances to stdout.
`--perf-to-csv`
Write all measured performances to automatically generated csv files.
`--perf-csv-to-stdout`
Print all measured performances to stdout in a csv format.
`-v or --verbose`
Enable verbose prints for debugging.

References

1. Blelloch, G.E.: Scans as primitive parallel operations. Computers, IEEE Transactions on 38(11), 1526–1538 (1989)
2. Helluy, P.: A portable implementation of the radix sort algorithm in opencl (2011), https://github.com/phelluy/ocl-radix-sort