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. TODO

1 Algorithm

The algorithm is based on an open source implementation of P. Helluy [2]. It is seperated into three different phases, which are executed consecutively.

Before we come to the detailed explanation of the algorithm, we first need to define several values. $K(j)$ for $j = 0 \dots N-1$, 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 $K(j)$ is between 0 and $2^b - 1$ 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 normal 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. To enable also the work with normal integers, the integers are incremented by an offset. 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 decremenrtation at the end of the algorithm is done on the fly to prevent performance losses. So basically from the outside it seemes the algorithm is working directly on the signed integer values, but on the inside it is working with a shifted number range to fit in the unsigned number range.

The Radix is represented by $R = 2^r$ where r is the number of bits necessary representing R. We suppose thath b 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 consists in sorting the list according to the q^{th} digit (in base R) $K(j)_q$. 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.

1.1 Histograming

This first phase of the algorithm is in charge of calculating the so called *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 a 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 can be 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 to an overall Histogram by using a Prefixsum algorithm. P. Helluy [2] and also the here shown improved implementation is using the Blelloch algorithm for parallel scans [1]. The Information given by the Prefixsum, allow 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.*

TODO

2 Implementation Details

2.1 Main Routine

2.2 Kernel

3 Experimental Results

Your hardware.

What do you benchmark.

Running time 1 and speedup plots 2 (for each generator, 64-bit integer and 32-bit floating point (not for non-comparative integer sorting algorithms).

Interpretation.

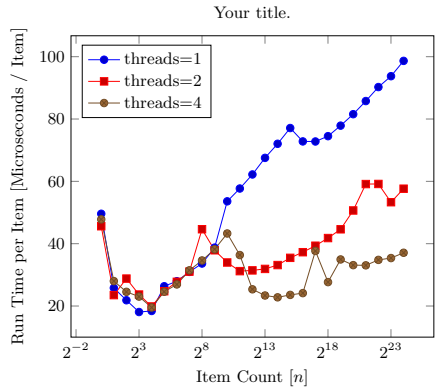

Fig. 1. Running times of `std::sort` with uniform input. Mean of 5 iterations.

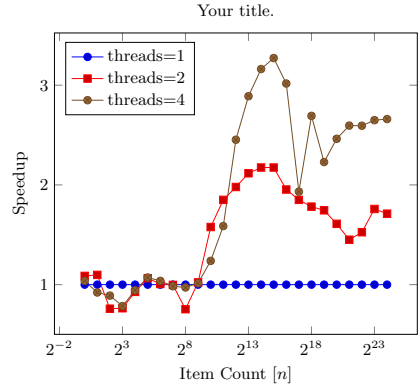

Fig. 2. Speedup of `std::sort` with uniform input. Mean of 5 iterations.

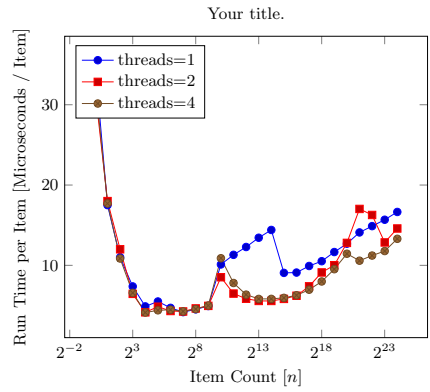

Fig. 3. Running times of `std::sort` with zero input. Mean of 5 iterations.

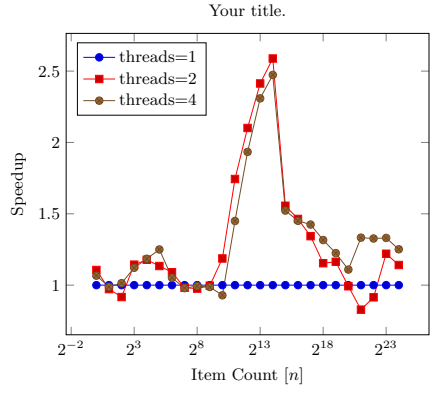

Fig. 4. Speedup of `std::sort` with zero input. Mean of 5 iterations.

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