Parallel Quick Sort in OpenCL

Patrick Bisenius, Manuel Leßmann

Karlsruhe Institute of Technology, Karlsruhe, Germany

Abstract. We have implemented a parallel version of the Quick Sort algorithm using OpenCL to be executed on GPUs. Experiments show, that a significant speedup compared to the sequential CPU version is only possible for very large inputs. A combination with a GPU algorithm that performs better for smaller inputs is advisable to maximize speedup.

1 Algorithm

Since the sequential Quick Sort algorithm is inherently recursive in nature, we used a similar approach for our GPU version. In every recursive call a pivot is selected and the GPU swaps elements such that smaller elements than the pivot are to its left in the array and larger elements are to its right. This is done by computing two prefix sums - one for the lower elements, one for the upper elements. An element adds 1 to the respective prefix sum, if it is lower/higher than the pivot, otherwise 0. As a result the number in the prefix sum array at the position of an element is its target index in the output array. Afterwards we only have to fill the gap between lower and upper numbers with pivots.

Pseudocode for this is shown in Algorithm 1. It is written in SIMD-style such that the function is called on a new thread for every item in the input. The parameter i is the index of the current item. The parameters $offset$ and $count$ describe the bounds of the current recursion; they start at 0 and input-length respectively.

2 Implementation Details

There is no proper way to write templates for OpenCL kernels. Therefore, our algorithm only supports 32-bit integers. Support for floating point numbers and 64-bit integers could be trivially achieved by copying the existing kernels and changing the relevant data types. On top of that, many GPUs only support 64-bit integers by software emulation.

3 Experimental Results

We benchmarked our algorithm on a machine with an i5-4670, 8GB of RAM and an AMD Radeon R290. We used `std::uniform_int_distribution` to generate a random array of numbers.

Algorithm 1 SIMD-pseudocode for the parallel quick sort algorithm on GPUs.

```
function QuickSort(input, output, offset, count, i)
    pivot ← Choose pivot
    lps ← Left prefix sum array
    rps ← Right prefix sum array
    if input[offset + i] < pivot then
        lps[i] ← 1
    else if input[offset + i] > pivot then
        rps[i] ← 1
    end if

    PrefixSum(lps)
    PrefixSum(rps)
    countLeft ← lps[last]
    countRight ← rps[last]
    countPivots ← count − countLeft − countRight

    if input[offset + i] < pivot then
        output[offset + lps[i]] ← input[offset + i]
    else if input[offset + i] > pivot then
        output[offset + countLeft + countPivots + rps[i]] ← input[offset + i]
    end if
    if i < numberPivots then
        output[offset + countLeft + i] ← pivot
    end if
    input ← output
    QuickSort(input, output, offset, countLeft, i)
    QuickSort(input, output, offset + countLeft + countPivot, countRight, i)
end function
```

Table 1 compares the runtime of `std::sort` and our implementation for different input sizes. Our implementation has a large amount of overhead, especially for small input sizes. However, it scales better than `std::sort` for larger input sizes because more and more elements in each step are equal to the pivot and are discarded in further recursion steps.

Table 2 shows the impact of different amounts of unique numbers for a constant input size of 2^{20}. With more unique numbers, less and less elements will be equal to the pivot element in each step. This increases the size of the remaining blocks and therefore the number of necessary recursion steps for our algorithm. The last column shows the runtime of our algorithm if the recursion is stopped if the block size is smaller than 1024.

We did not achieve a speedup with respect to `std::sort` for any input size. As shown in Table 2, a large percentage of the total runtime is spent in recursion steps with block sizes smaller than 1024 elements. For these small blocks, the overhead for the kernel invocations and copy operations on the OpenCL buffers becomes too large. Since OpenCL kernels are invoked from a command queue and only one kernel is executed at a time, the number of active GPU threads becomes smaller and smaller for each recursion step. This problem could be solved by implementing a second sorting algorithm for small block sizes that uses a single workgroup to sort a small block and sorts multiple small blocks at a time, such as bitonic sort.

n	CPU	GPU
2^{20}	73	10151
2^{21}	140	10183
2^{22}	284	11050
2^{23}	548	11134
2^{24}	1091	11093
2^{25}	2133	11809
2^{26}	4263	12909
2^{27}	8620	13764

Table 1. Runtime in milliseconds of the QuickSort algorithm with 2^{14} different numbers.

# numbers	CPU	GPU	rec. aborted
2^{14}	74	10934	393
2^{18}	82	37087	393
2^{20}	84	76149	373

Table 2. Runtime in milliseconds for $n = 2^{20}$ and different amounts of unique numbers. The last column shows the runtime if the recursion is aborted when the block size is smaller than 1024 to show the impact of the last recursion steps.