Introduction to CUDA and OpenCL

Łukasz Gut, Grzegorz Litarowicz

Lab 08, 21.05.2020

Introduction

During eighth laboratories, our task was to implement a histogramming algorithm both on host and device. Then we were supposed to optimize its performance on the device and analyze reads and writes performed by the kernel.

Histogram

A histogram is an approximate representation of the distribution of numerical or categorical data. To construct a histogram, the first step is to divide the domain into "bins" - series of intervals and then count the number of values that fall into each interval.

In our task, we had to construct a 2048 histogram, which means that the number of "bins" equals 2048. The maximum resolution of a bin had to be 16 bits. The input data was to be between 1 and 2 billion.

Single-threaded implementation

Single-threaded implementation is trivial, there is really nothing to it.

```
for(std::size_t i = 0; i < input_size; i++) {
  if(histogram[static_cast<std::size_t>(data[i])] < std::numeric_limits<uint16_t>::max())
    histogram[static_cast<std::size_t>(data[i])] += 1;
}
```

Naive kernel implementation

Implementing naive kernel to calculate the histogram was also fairly simple. The only caveat was to use atomic operations, because of race conditions. Therefore, we use *atomicAdd* to increment the value inside a bin. Also, to deal with the saturation problem, we used *atomicMin*, which takes care of truncating the value inside a bin to some maximum value, which in our case was USHRT_MAX.

Also, not to go fully naive, we actually did make use of striding technique, because the number of samples was really huge.

```
__global__
void naive_histogram_kernel(data_t const * const input_vec, hist_t * const histogram)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;

  while(i < input_size) {
    auto index = static_cast<std::size_t>(input_vec[i]);
    atomicAdd(histogram + index, 1);
    atomicMin(histogram + index, USHRT_MAX); // saturation handling

    i += stride;
  }
}
```

Optimized kernel implementation (128, 256, 512, 1024)

Optimizing the kernel was a bit more tricky. Our main focus was on shared memory. We came up with an idea to let each block in the processing grid calculate its own local histogram and only after this we would push the data back into global memory. This resulted in much less *write* operations into the global memory, thus boosting the kernel's performance.

Unfortunately, this implementation works only if the number of threads per block is the same as the number of bins. In such a scenario, we have a perfect mapping between threadIdx.x, shared memory and global memory.

```
__global__
void shared_histogram_kernel(data_t const * const input_vec, hist_t * const histogram)
{
  __shared__ hist_t localHistogram[n_bins];

  for(std::size_t i = 0; i < n_bins; i++)
    localHistogram[i] = 0;

   __syncthreads();

  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if(i < input_size) {
    const auto index = static_cast<std::size_t>(input_vec[i]);
    atomicAdd(localHistogram + index, 1);
    atomicMin(localHistogram + index, USHRT_MAX); // saturation handling

    __syncthreads();

    atomicAdd(histogram + threadIdx.x, localHistogram[threadIdx.x]);
    atomicMin(histogram + threadIdx.x, USHRT_MAX); // saturation handling
  }
}
```

Optimized kernel implementation (2048)

To make this kernel work for a larger number of bins, as it was specified in an assignment, we had to use a striding technique in order to iterate over whole histogram. However, this solution, as we will see in further benchmarks is not optimal.

To figure out how many threads in a single block will actually have to do the work, we perform this simple calculation:

```
constexpr std::size_t n_bins = 2048;
constexpr std::size_t optimized_threads_per_block = 256;
constexpr std::size_t working_threads = n_bins / optimized_threads_per_block;
```

In our scenario, only 8 threads per block will be working.

Below is the kernel code, which is very similar to its predecessor.

```
__global__
void shared_histogram_kernel(data_t const * const input_vec, hist_t * const histogram)
{
  __shared__ hist_t localHistogram[n_bins];

  for(std::size_t i = 0; i < n_bins; i++)
    localHistogram[i] = 0;

   __syncthreads();

  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if(i < input_size) {
    const auto index = static_cast<std::size_t>(input_vec[i]);
    atomicAdd(localHistogram + index, 1);
    atomicMin(localHistogram + index, USHRT_MAX); // saturation handling

    __syncthreads();

    if(threadIdx.x < working_threads) {
      int localStride = n_bins / blockDim.x;
      for(int j = threadIdx.x; j < n_bins; j += localStride) {
        atomicAdd(histogram + j, localHistogram[j]);
        atomicMin(histogram + j, USHRT_MAX); // saturation handling
      }
    }
  }
}
```

Benchmarks

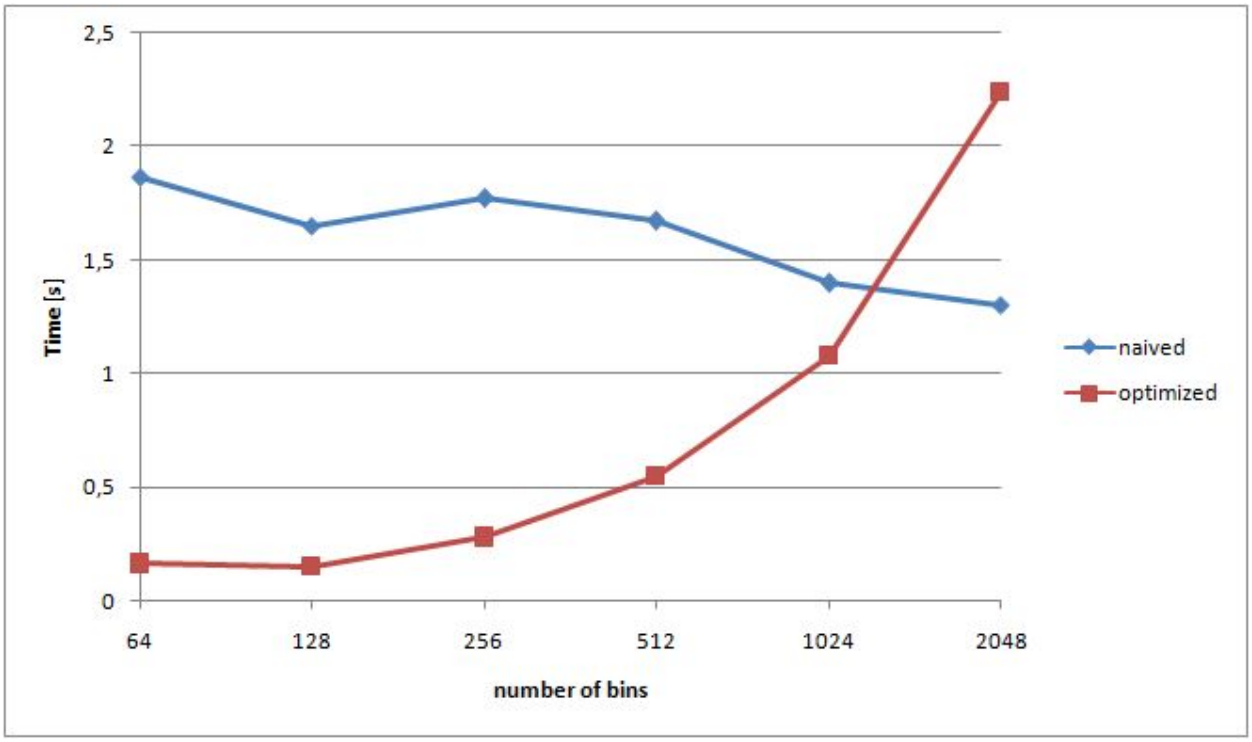

Chart 1

As we can see on the chart 1 optimized kernel is much faster but only up to 1024 bins. This is happening because of the limit of the threads per block (also 1024).

If the number of the bins is greater than the maximum number of threads per block, in order to sum up the results from shared memory to global memory we have to loop through every local table. And, as we know, this operation is very expensive on the GPU.

	Read	Write
Naive	2n	O(2n)
Optimized	2n	n_bins

Where:

n - number of elements in the input vector

n_bins - number of ranges

Conclusion

In this report, we provided three implementations of the histogram algorithm: single-threaded and three kernels. We also proved, that using shared memory, to reduce the number of writes to global memory can have a huge impact on the performance. We think that this was the most important optimization.

However, there surely is much more room for improvement here, even more so with histograms in which number of bins is smaller than the maximum number of threads per block.