Dominik Dziuba

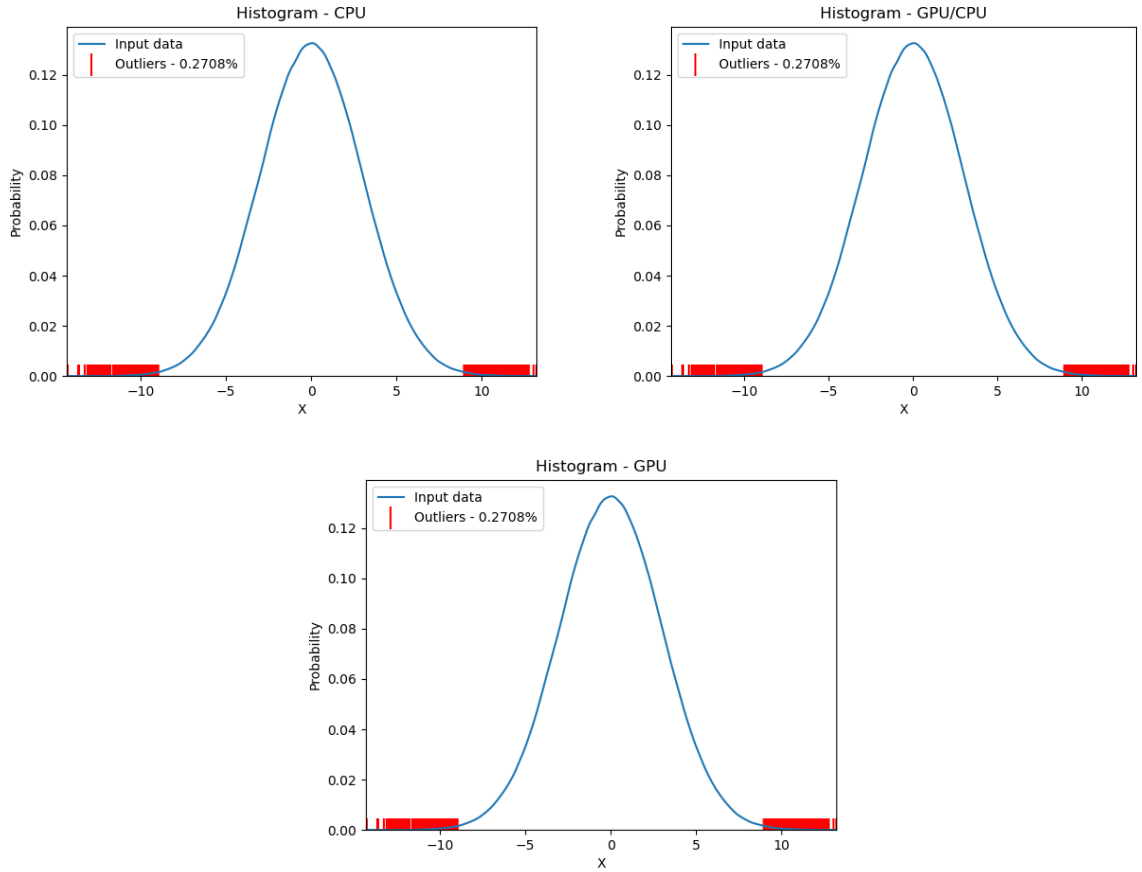
Accelerated algorithm for anomaly detection using statistical approach

Description

Finding outliers in a given dataset is an important step in understanding the nature and meaning of given data. One of the simplest and most common methods for doing so is a 3-sigma method. This method, assuming normal distribution of values in the input data, uses the mean and standard deviation values to determine if a given data point is an outlier in the whole set. Statistically speaking values in the range of $[mean - 3 \cdot \sigma, mean + 3 \cdot \sigma]$ (where σ is standard deviation) cover about 99,7% of values in the dataset. It is a good assumption that values outside of that range are indeed outliers. Naturally for different datasets it is advised to change the amount of sigmas therefore changing the range they cover.

For the purposes of this project three different approaches of finding outliers in a 1D input were devised and implemented so as to be compared:

- Strictly CPU focused approach of finding outliers
- GPU accelerated computation of mean and standard deviation followed by sequential CPU search (same as strictly CPU approach) method of finding outliers
- Mostly GPU approach where both initial mean and standard deviation computation are accelerated as well as GPU accelerated method of finding outliers in the input dataset. Some elements of this approach used CPU to carry out some final reductions and mappings



Pic. 1. Validation of correctness of the different approaches. Testing on random normal distribution with $mean = 0, \sigma = 3.0$. Blue line is probability distribution of input and red lines are singular outliers.

Implementation

On the CPU side of implementations all computation was carried out sequentially. This includes computation of mean and standard deviation as well as final scan to find all of the outliers. Computing in both CPU and GPU case used a single pass method to find mean and standard deviation values, using the following modification of usual variance formula:

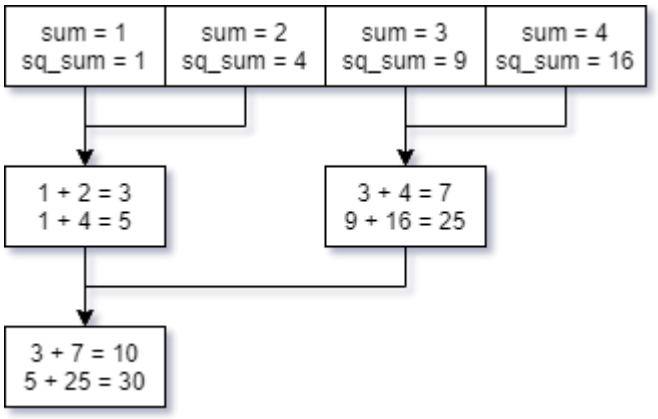
$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x}) = \left(\frac{1}{N}\sum_{i=1}^{N}x_i^2\right) - \bar{x}^2$$

Using this formula, sum of squared values in input as well as sum are computed in a single loop, then all values are computed:

$$\bar{x} = \frac{S_x}{N}$$
$$\sigma^2 = \frac{S_{x^2}}{N} - \bar{x}^2$$
$$\sigma = \sqrt{\sigma^2}$$

Where: N – amount of points in input, S_x – sum of input points, S_{x^2} – sum of input points squared, $\bar{x}$ – mean of input points

CUDA kernel that computes mean and standard deviation uses a privatized approach with a tree based way of reduction. Each block initiates a shared array of partial sums and partial squared sums with appropriately computed input points. Then follows a reduction loop that works as shown on the picture 2. Finally partial values that need to be summed in the end to compute relevant statistics are committed to global memory of the device and then to the CPU to perform the final calculations.



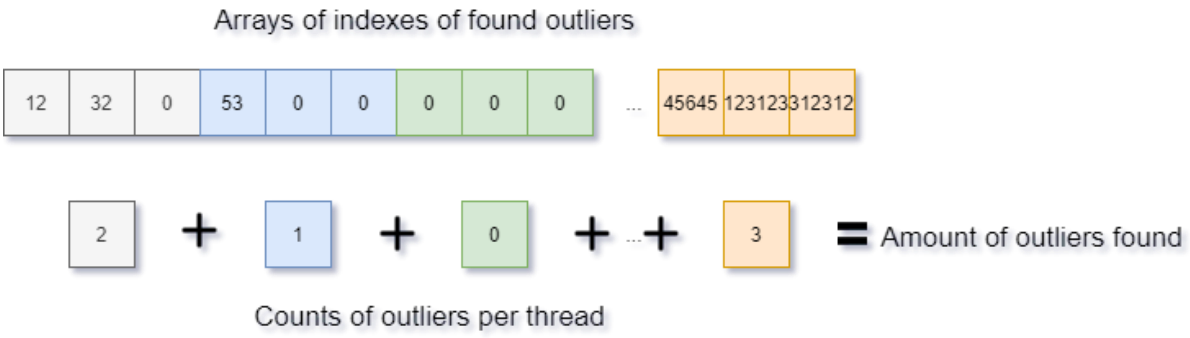
Pic. .2. Reduction loop idea visualization.

And relevant code implementing this kernel:

```
template<typename inputType>
__global__ void cudaReductionPartialStdDev(unsigned long int dataSize,
inputType* dataInput, inputType* partialSums, inputType*
partialSquaredSums) {
	__shared__ inputType localPartialSums[2 * THREADS_PER_BLOCK];
	__shared__ inputType localSquaredPartialSums[2 * THREADS_PER_BLOCK];
	unsigned int th = threadIdx.x;
	unsigned int start = 2 * blockIdx.x * THREADS_PER_BLOCK;
	unsigned int i = start + 2 * th;

	if(i < dataSize) {
		localPartialSums[2 * th] = dataInput[i];
		localSquaredPartialSums[2 * th] = dataInput[i] * dataInput[i];
	}
	else {
		localPartialSums[2 * th] = 0;
		localSquaredPartialSums[2 * th] = 0;
	}

	if(i + 1 < dataSize) {
		localPartialSums[2 * th + 1] = dataInput[i + 1];
		localSquaredPartialSums[2 * th + 1] = dataInput[i + 1] *
dataInput[i + 1];
	}
	else {
		localPartialSums[2 * th + 1] = 0;
		localSquaredPartialSums[2 * th + 1] = 0;
	}
```

```
	for(int stride = 1; stride <= THREADS_PER_BLOCK; stride *= 2) {
		__syncthreads();
		if(th % stride == 0) {
			localPartialSums[2 * th] += localPartialSums[2 * th +
stride];
			localSquaredPartialSums[2 * th] +=
localSquaredPartialSums[2 * th + stride];
		}
	}

	__syncthreads();
	partialSums[blockIdx.x] = localPartialSums[0];
	__syncthreads();
	partialSquaredSums[blockIdx.x] = localSquaredPartialSums[0];
}
```

In the mostly GPU approach another CUDA kernel is utilized. This kernel performs search of outliers given mean and standard deviation values and the amount of sigmas that was passed to the function. Its implementation is just a basic grid-stride to iterate over the whole input array but its output arguments are more complex. This kernel tries to avoid having to sequentially iterate over the whole input array using two arrays. One array contains amount of outliers found by each thread and is of total amount of threads in the kernel launch length. While the other contains sub arrays of indexes of each outlier found in the input, this array is at least the length of the count array but can be multiplied to be able to fit all possible outliers. As exemplified in the picture 3, the upper array is the one holding indexes and the lower is the array of sizes of each sub array.



Pic. 3. GPU kernel output. Indexes in the upper array are for demonstrative purposes, don't reflect actual look.

As the array of indexes need to be at least the length of input array, this approach cannot be fully privatized as it demands too much shared memory. Following is the code implementing this kernel:

```
template<typename inputType>
__global__ void cudaFindOutliers(unsigned long int dataSize, inputType*
dataInput, double sigmaOffset, double mean, unsigned long int*
outlierCounts, unsigned long int* outlierIndexesArray, unsigned int
pointsPerThread) {
	unsigned long int th = threadIdx.x;
	unsigned long int i = blockIdx.x * blockDim.x + th;
```

```
	unsigned long int stride = blockDim.x * gridDim.x;
	unsigned long int globalStart = i * pointsPerThread;
	unsigned long int outlierCount = 0;

	for(unsigned long int j = i; j < dataSize; j += stride) {
		if(dataInput[j] > mean + sigmaOffset || dataInput[j] < mean -
sigmaOffset) {
			outlierIndexesArray[globalStart + outlierCount] = j;

			outlierCount++;
		}
	}

	outlierCounts[i] = outlierCount;
}
```

Finally it is worth noting that the final output of each function that handles finding outliers using all of the described approaches, returns a structure containing the outlier indexes one after another. In case of the CPU based sequential seeking functions this means the returned indexes are sorted in ascending order. The output of the mostly GPU approach on the other hand can be in a more or less random order if there is more input data than the maximum number of threads set in the application.

On this note it needs to be pointed out that following configurations were utilized for kernel launches:

- THREADS_PER_BLOCK – in both kernels – 1024
- Amount of blocks in the mean and standard deviation kernel – appropriate to the input array
- MAX_BLOCKS – for the grid-stride kernel performing finding outliers – 120

The final application takes 4 arguments and can be run as shown:

```
./proj_outliers <input data size> <mean> <stddev> <sigmas>
```

Where: `<input data size>` – size of input array that will be initiated, `<mean>` – is the mean value of the distribution the input will be initiated with, `<stddev>` – is the standard deviation of the initiated input, `<sigmas>` – is the amount of sigmas you want to find outliers for

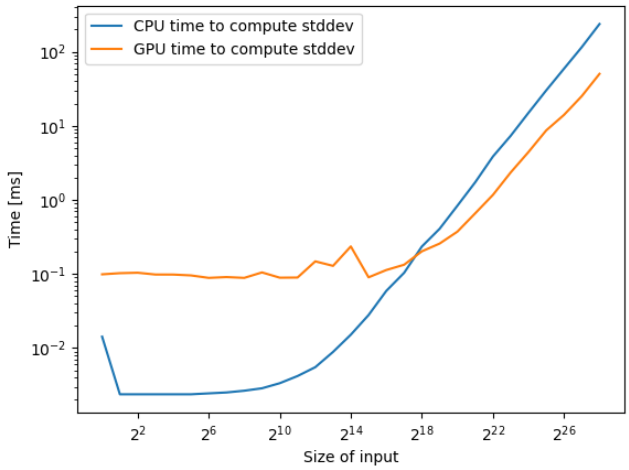
Performance

The final program was prepared so as to make measurements of time taken by each computational step needed to compute the outliers. For each approach each step was performed independently and measured with appropriate CUDA features in case of GPU kernels and convenient tools from the *<chrono>* library available in C++11 for measuring performance of the CPU tasks. The main program launches all three approaches in sequence and takes an input parameter specifying the size of the input array, which is then initiated

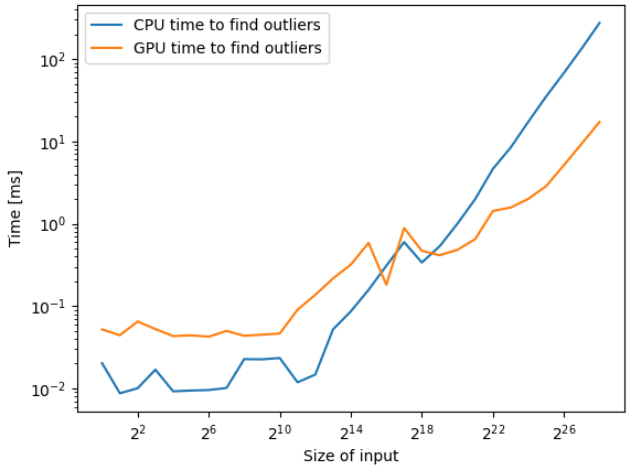
using a normal distribution (from the *<random>* library). Then each step of each approach is measured. Those steps include:

1. Compute mean and standard deviation (in GPU approaches in addition to the time taken by the kernel, the time to perform final reduction on CPU is added)
2. Find outliers in the input array
3. In case of the mostly GPU approach the time taken by CPU to squash the output from the kernel into the final output is also measured.

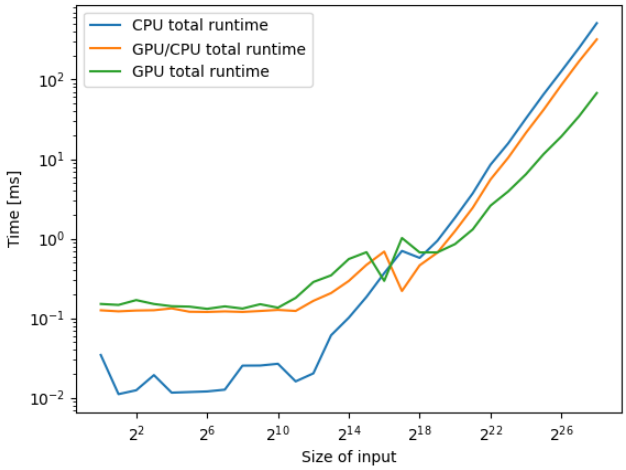
For the following graphs the program was run with varying sizes of input array generated using mentioned normal distribution generator. Again with $mean = 0, \sigma = 3.0$ and the size of the array in range of $[2^0, 2^{28}]$ and the usual amount of sigmas at 3.0.



Pic. 4. Comparison of mean and standard deviation computation times.



Pic. 5. Comparison of time taken by each approach to find outliers.



Pic. 6. Comparison of total run times with each of the three approaches.

Summary

The project provided an interesting way to solving some complex issues while utilizing CUDA framework and hardware. Within a certain range that approach delivers tangible speedup against a simple CPU approach that uses only a single thread. It was also great way to utilize the knowledge gained in the Nvidia DLI course on how to optimize and design the kernel functions. Finally the finished project works properly and results returned by each approach not only match expected ones but also validate against each other. Most importantly GPU approach output matches the CPU one albeit it needs to be sorted first.