Reduction algorithm

Reduction is one of the most common operations on data. It is used to compute means, maximums, minimums, sums and many more useful pieces of information about a given input. As such it often needs to fit certain performance criteria. Reduction operators are often commutative but they don't have to be. In this exercise the sum operator is, which eases the implementation of such algorithm using GPU. To give some idea why parallelization may be important the following graph can be considered. With increasing size of the problem the difference between the time required for even simple sum operation rises greatly on CPU.

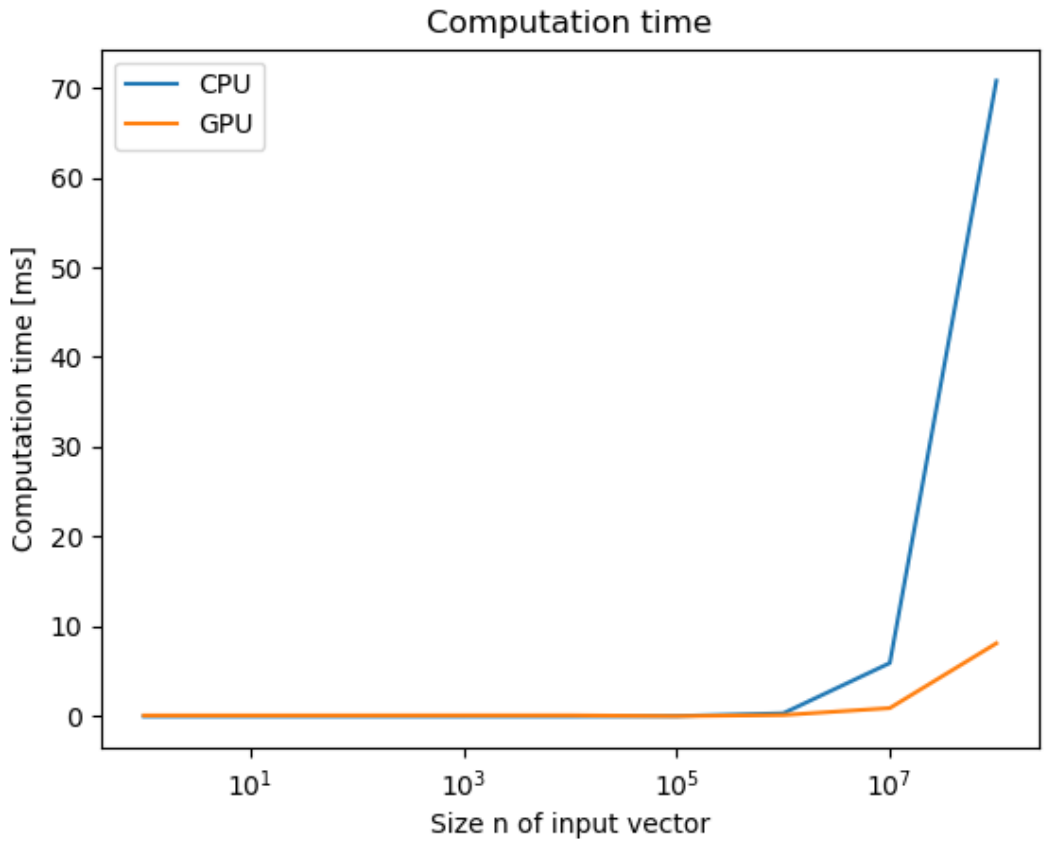

Fig. 1. Performance of CPU and GPU implementations of sum operator.

The sum operator in this exercise is implemented using a tree based method, where appropriate amount of processing blocks is launched each processing 2 times the amount of inputs as the amount of threads in each block. Using a simple divide and conquer approach in a loop summing 2 points and deactivating every other thread with each loop until the final partial sum of initial input partition is reached in each blocks shared memory table at index 0. At the start of GPU code execution each thread initializes 2 elements of shared memory based on its ID and its block's ID corresponding to appropriate elements in input data. Depending on whether an element exists (conforming to the size of the input) its either assigned from global memory to the shared one or is instead zeroed in shared array.

Considering the tree nature of this algorithm it can be approximated that for block size (amount of threads per block) of 1024 at most a thread does 11 summing loops (when actual summation happens) and at the least 1. But this implementation also means that each block also has to synchronize with each loop pass (11 times). Considering the heavy use of synchronization barriers there is no reason to use atomic operations and as a result using them could lead to performance loss.

This being just a basic implementation some optimizations can be made:

- Avoiding use of % operator within the loop
- Avoiding divergences within the main loop
- Replace strided addressing with sequential one
- Considering half the threads do nothing after the first loop, the first iteration could be done explicitly
- Again considering that with each loop the number of active threads decreases, a possible solution is that when only 32 (amounting to a single warp) are active, an exact reduction can be performed

Code

```
#include <stdio.h>
#include <stdint.h>
#include <assert.h>
#include <math.h>
#include <cuda_runtime.h>

#define THREADS_PER_BLOCK 1024

#define checkCudaErrors(code) { cudaAssert((code), __FILE__,
__LINE__); }
inline void cudaAssert(cudaError_t code, const char *file, int line,
bool abort = true)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,"cudaAssert: %s %s %d\n",
cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}

void initInput (unsigned long int size, long int* data, long int
initVal) {
     for(int i = 0; i < size; ++i) {
          data[i] = initVal;
     }
}

__global__ void cudaReductionPartialSums(unsigned long int dataSize,
long int* dataInput, long int* partialSums) {
```

```
     __shared__ long int localData[2 * THREADS_PER_BLOCK];
     unsigned int th = threadIdx.x;
     unsigned int start = 2 * blockIdx.x * THREADS_PER_BLOCK;
     unsigned int i = start + 2 * th;

     // init lokalnej tablicy
     if(i < dataSize) {
          localData[2 * th] = dataInput[i];
     }
     else {
          localData[2 * th] = 0;
     }

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

     // petla redukcji
     for(int stride = 1; stride <= THREADS_PER_BLOCK; stride *= 2) {
          __syncthreads();
          if(th % stride == 0) {
               localData[2 * th] += localData[2 * th + stride];
          }
     }

     // commit do vramu
     partialSums[blockIdx.x] = localData[0];
}

long int reductionGPU(unsigned long int dataSize, long int*
dataInput) {
     long int res = 0;
     long int* partialSums;
     unsigned int partialSumsSize = dataSize / (THREADS_PER_BLOCK *
2);
     partialSumsSize = dataSize % (THREADS_PER_BLOCK * 2) ?
partialSumsSize + 1 : partialSumsSize;

     checkCudaErrors(cudaMallocManaged(&partialSums, partialSumsSize
* sizeof(long int)));

     checkCudaErrors(cudaMemPrefetchAsync(partialSums,
partialSumsSize * sizeof(long int), 0));

     cudaEvent_t start;
     checkCudaErrors(cudaEventCreate(&start));

     cudaEvent_t stop;
     checkCudaErrors(cudaEventCreate(&stop));

     checkCudaErrors(cudaEventRecord(start, NULL));
```

```
     // odpalanie kerneli
     cudaReductionPartialSums<<<partialSumsSize,
THREADS_PER_BLOCK>>>(dataSize, dataInput, partialSums);

     checkCudaErrors(cudaDeviceSynchronize());

     checkCudaErrors(cudaGetLastError());

     checkCudaErrors(cudaEventRecord(stop, NULL));

     checkCudaErrors(cudaEventSynchronize(stop));

     for(unsigned int j = 0; j < partialSumsSize; ++j) {
          res += partialSums[j];
     }

     float msecTotal = 0.0f;
     checkCudaErrors(cudaEventElapsedTime(&msecTotal, start, stop));

     printf("GPU code run time: %f ms\nresult: %d\n", msecTotal,
res);

     checkCudaErrors(cudaEventDestroy(start));
     checkCudaErrors(cudaEventDestroy(stop));
     checkCudaErrors(cudaFree(partialSums));

     return res;
}

long int reductionCPU(unsigned long int dataSize, long int*
dataInput) {
     timespec start, end;
     long int res = 0;

     clock_gettime(CLOCK_MONOTONIC_RAW, &start);

     for(unsigned long int i = 0; i < dataSize; ++i) {
          res += dataInput[i];
     }

     clock_gettime(CLOCK_MONOTONIC_RAW, &end);

     float msecTotal = (float)((end.tv_sec - start.tv_sec) * 1000000
+ (end.tv_nsec - start.tv_nsec) / 1000) / 1000;
     printf("CPU code run time: %f ms\nresult: %d\n", msecTotal,
res);

     return res;
}

int main(int argc, char** argv) {
     cudaDeviceProp prop;
     checkCudaErrors(cudaSetDevice(0));
```

```
	checkCudaErrors(cudaGetDeviceProperties(&prop, 0));

	unsigned long int dataSize;
	char* endptr;

	if(argc != 2) {
		exit(-1);
	}

	dataSize = strtoul(argv[1], &endptr, 0);

	printf("running on: %s\ndata size: %lu\n", prop.name,
dataSize);

	long int* dataInput;

	checkCudaErrors(cudaMallocManaged(&dataInput, dataSize *
sizeof(long int)));

	initInput(dataSize, dataInput, 1);

	long int cpuRes = reductionCPU(dataSize, dataInput);

	checkCudaErrors(cudaMemPrefetchAsync(dataInput, dataSize *
sizeof(long int), 0));

	long int gpuRes = reductionGPU(dataSize, dataInput);

	if(cpuRes != gpuRes) {
		printf("BLAD!!!!!!! %ld != %ld\n", cpuRes, gpuRes);
	}

	printf("end\n");

	checkCudaErrors(cudaFree(dataInput));

	return EXIT_SUCCESS;
}
```

Naturally the code needs to be compiled using Nvidia's dedicated compiler *nvcc* after which example usage can look like this:

./executable_name size_of_input

Where:

- size_of_input – size of input data (number n)

In the current version of the program the input data is initiated with 1s so as to make validation of output easier.