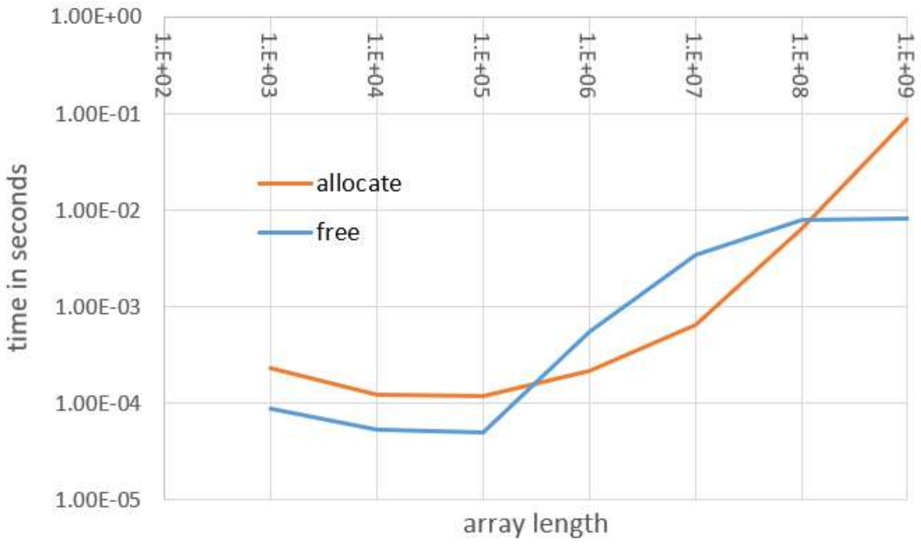
Simon Hinterseer e09925802

Exercise 02

1 Basic CUDA

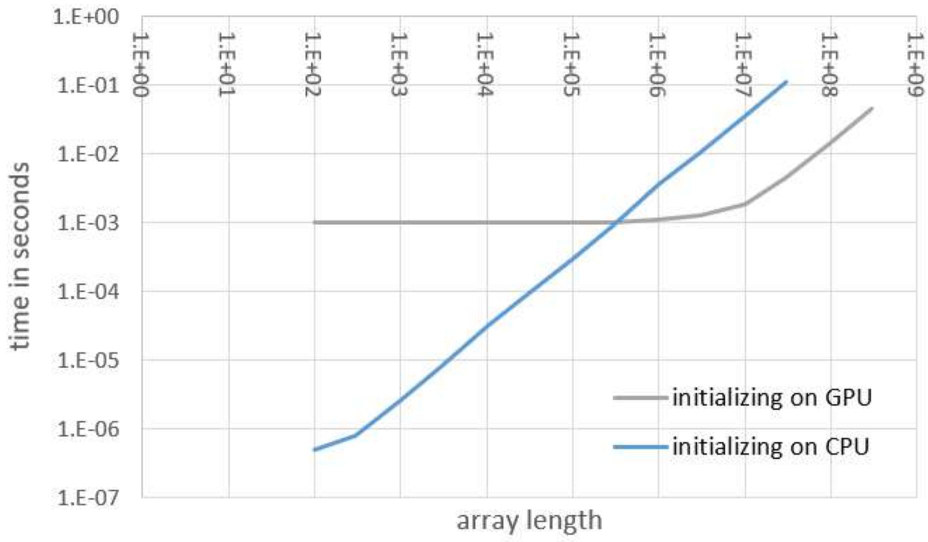
a. Time for cudaMalloc() and cudaFree()



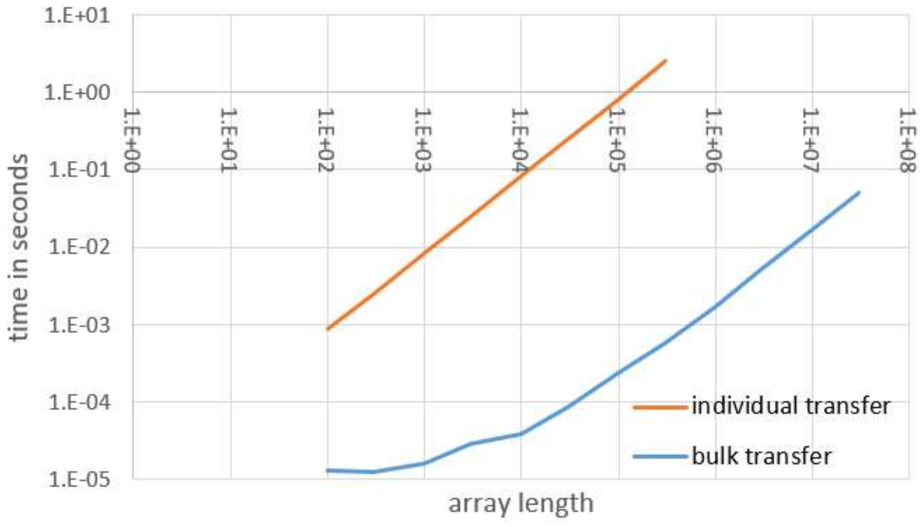
While allocation stabilizes around 30 GB/s, freeing up memory shows a less predictable behaviour.

b. Initialization

Comparing initialization on CPU with initialization on GPU shows, that for small arrays, the CPU is faster (at around 560 MB/s) since there seems to be some overhead, that slows the GPU down. For arrays with over a million entries, the GPU starts to be faster (around 13.5 GB/s).



When it comes to transferring data, the bulk transfer is obviously faster. Individual is stable at $4\ \mu s/entry$. While bulk transfer manages to transport around 1.2 GB/s.



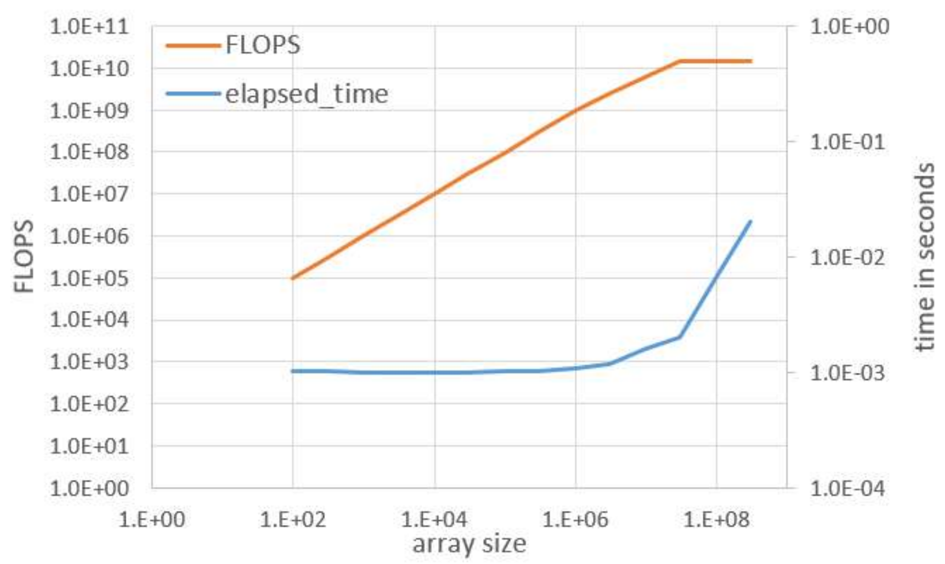
c. Sum of vectors

This is my kernel:

```
__global__
void gpu_sum(const double *gpu_x, const double *gpu_y, double* gpu_result,
             const size_t size){
    size_t thread_id = blockIdx.x*blockDim.x + threadIdx.x;
    for(size_t i=thread_id; i<size; i += blockDim.x*gridDim.x){
        gpu_result[i] = gpu_x[i] + gpu_y[i];
    }
}
```

d. Execution time

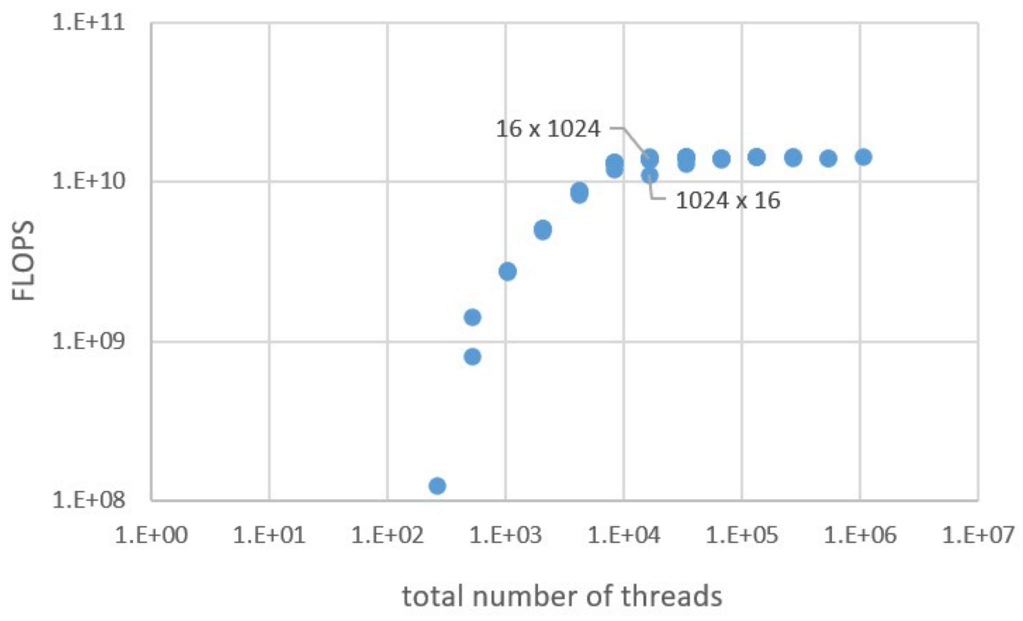
When summing two vectors on the GPU, this performance was observed



Obviously, only for arrays of size over 10^7 entries, the GPU finds its limits. At that point the performance is around 14.8 $GFLOPS$. This is only a fraction of the 277 $GFLOPS$ that is the nominal peak performance of the GTX 1080 (https://www.techpowerup.com/gpu-specs/geforce-gtx-1080.c2839). This low hardware efficiency is probably due to the low arithmetic intensity of summing two vectors. However, the utilized memory bandwidth is here also only 44.6 GB/s which again is a fraction of the promised 320 GB/s.

e. Grid- and block sizes

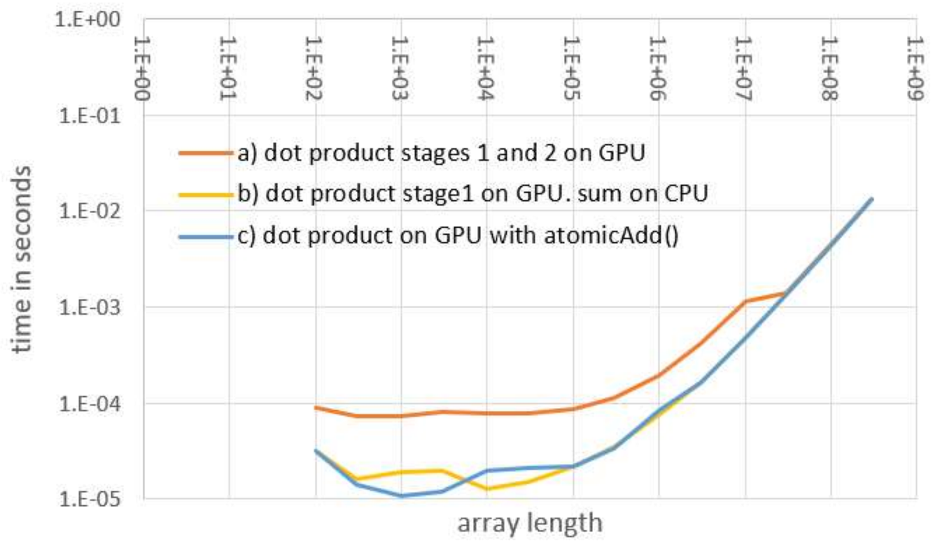
When using different block and grid sizes for an array size of $N = 10^7$, it was observed, that one is mostly safe in terms of performance, as long as the number of threads is sufficiently large.



There is one pair of data points with the same total number of threads, that results in somewhat different performance: there it turns out, that 16 blocks of size 1024 each perform better than 1024 blocks of size 16 each. This seems plausible considering, that there are only 20 multicore units on the RTX 1080.

2 Dot product

All three kernels performed very similarly, when it comes to large arrays ($N > 3 \cdot\ 10^7$). For smaller arrays, the kernel that parallelizes the final sum reduction on the GPU is noticeably slower. This is possibly a consequence of the fact, that two GPU kernels must be called by the CPU. It should be noted, that the summing on the CPU in example (b) was done single-threaded.



Doing all computation on the GPU is certainly more appropriate, if the GPU needs the result for further computation. Transferring data back to the CPU to do computations just to transfer them back to the GPU would be a bad strategy.

If the result is not needed for further computation on the GPU, it is possible to do the final computation on the hosing CPU. Only a few hundred doubles need to be transferred and a few hundred operations will be needed.

The kernel, that uses the `atomicAdd()` function, seems to be the best of both worlds: only one kernel needs to be called and the result is available on the GPU for further computation if necessary.

3 Annex: Dot-Product kernels

```
#define GRID_SIZE 256
#define BLOCK_SIZE 128

…

__global__
void gpu_dotproduct_stage1(const double *gpu_x, const double *gpu_y, size_t
size, double *gpu_result_stage1){

      size_t thread_id_global = blockIdx.x*blockDim.x + threadIdx.x;
      __shared__ double shared_m[BLOCK_SIZE];


      // I think, this is the right way:
      double thread_dp = 0;
      for (unsigned int i = thread_id_global; i<size; i += blockDim.x *
gridDim.x)
            thread_dp += gpu_x[i] * gpu_y[i];
      shared_m[threadIdx.x] = thread_dp;


      // now the reduction
      for(int stride = blockDim.x/2; stride>0; stride/=2){
            __syncthreads();
            if (threadIdx.x < stride){
                  shared_m[threadIdx.x] += shared_m[threadIdx.x + stride];
            }
      }

      // thread 0 writes result
      if (threadIdx.x == 0){
            gpu_result_stage1[blockIdx.x] = shared_m[0];
      }
}
```

```
__global__
void gpu_dotproduct_stage2(double *gpu_result_stage1, double
*gpu_result_stage2){

      // only one block has a job here
      if (blockIdx.x == 0){
            //size_t thread_id_global = blockIdx.x*blockDim.x +
threadIdx.x;
            __shared__ double shared_m[BLOCK_SIZE];

            // this time, the lecture is correct
            double thread_sum = 0;
            for (unsigned int i = threadIdx.x; i<GRID_SIZE; i +=
blockDim.x)
                  thread_sum += gpu_result_stage1[i];
            shared_m[threadIdx.x] = thread_sum;

            // now the reduction
            for(int stride = blockDim.x/2; stride>0; stride/=2){
                  __syncthreads();
                  if (threadIdx.x < stride){
                        shared_m[threadIdx.x] += shared_m[threadIdx.x +
stride];
                  }
            }
            //__syncthreads();
            // thread 0 writes result
            if (threadIdx.x == 0){
                  //printf("hi, im thread 0 and im now writing %1.5e\n",
shared_m[0]);
                  *gpu_result_stage2 = shared_m[0];
            }
      }
}
```

```
__global__
void gpu_dotproduct_atomicAdd(const double *gpu_x, const double *gpu_y,
size_t size, double *gpu_result){

      size_t thread_id_global = blockIdx.x*blockDim.x + threadIdx.x;
      if (thread_id_global == 0)
            *gpu_result = 0;

      __shared__ double shared_m[BLOCK_SIZE];


      // I think, this is the right way:
      double thread_dp = 0;
      for (unsigned int i = thread_id_global; i<size; i += blockDim.x *
gridDim.x)
            thread_dp += gpu_x[i] * gpu_y[i];
      shared_m[threadIdx.x] = thread_dp;


      // now the reduction
      for(int stride = blockDim.x/2; stride>0; stride/=2){
            __syncthreads();
            if (threadIdx.x < stride){
                  shared_m[threadIdx.x] += shared_m[threadIdx.x + stride];
            }
      }

      __syncthreads();
      // thread 0 writes result
      if (threadIdx.x == 0){
            atomicAdd(gpu_result, shared_m[0]);
            //gpu_result_stage1[blockIdx.x] = shared_m[0];
      }
}
```