Computational Science on Multi-Core Architectures: Exercise 2 (Tasks 1.acde and Task 2a):

Viktor Beck, 11713110

18. October 2022

Task1: Basic Cuda

1.a)

In Figure 1 one can see that cudaFree is definetely faster than cudaMalloc - allocation of memory seems to be more computationally expensive than freeing memory. The plot was produced with datapoints obtained from the program "basic_cuda_a" (see appendix). The time was averaged over 10 repeats. Note that the time was measured in each repeat instead of just measuring outside of the for-loop (no time to correct now that but at least I got smarter).

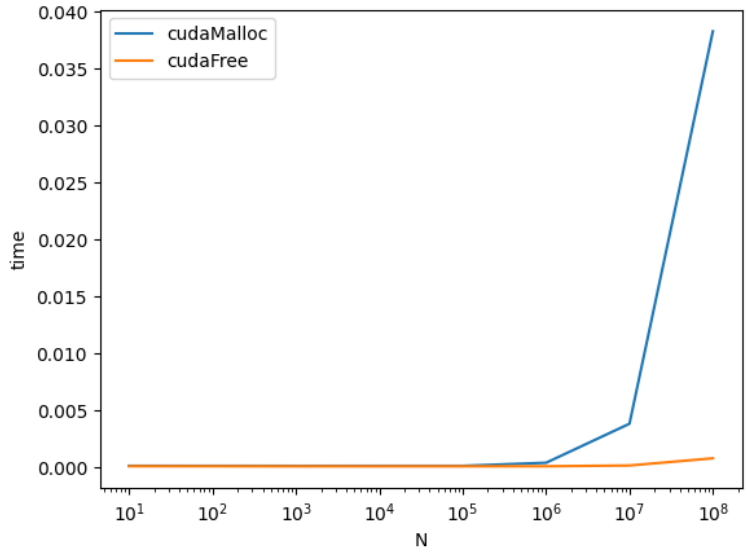

Figure 1: Time it takes to run cudaMalloc and cudaFree for different sizes of N

1.c) and d)

For these tasks a new program, "basic_cuda_cde", was created (similar to the one from 1.a)). The program loops over different values of N (here with 5 repeats) which can be seen in Figure 2. One can see that for N up to 10 to the power of 6 the curve stays almost constant and then

increases rapidly. This is due to the chosen numbers of blocks and threads per block which was 256x256 which equals 65.536 threads in total. At roughly that number of N it starts to increase which means that at that point, some threads have to calculate more than one element in the vector.

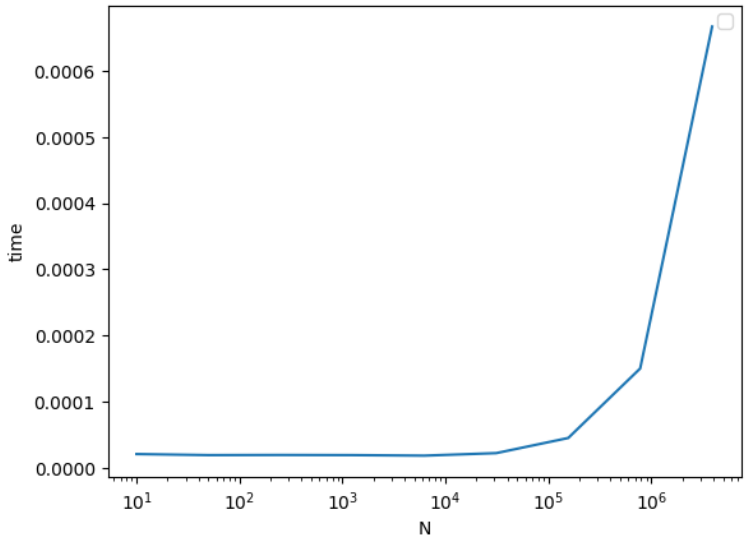

Figure 2: Time it takes to run the vector addition for different sizes of N

1.e)

Here the same code as for c) and d) was used but with fixed N equal to 10 to the power of 7. In Figure 3 one can see that execution time for the vector addition decreases with a higher number of threads per block - as it should be. The number of blocks stayed the same (256) while the number of threads per block went from 16 to 1024. Even for 512x1024 and 1024x1024 (not documented in the Figure) the time did not decrease significantly anymore which means that the curve in Figure 3 would continue on a constant level.

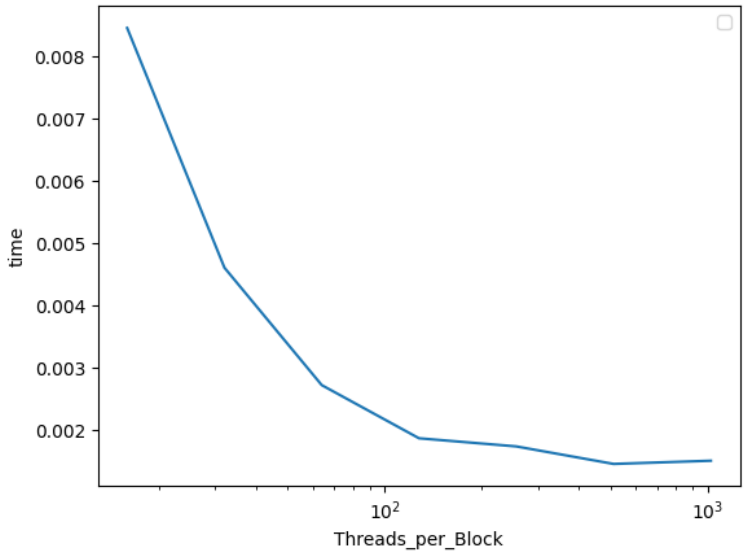

Figure 3: Time it takes to run the vector addition for different numbers of threads per block

Task2: Dot Product

2.a)

One can see the plot in Figure 4 is (unsurprisingly) almost identical to the one of Figure 2. The dot product was calculated by two different kernels, one for multiplication (and summation) and one (only) for summation. In the first kernel the products of the single elements were stored into shared memories which were each shared by a whole block. These products were then summed up in each block. After that the second kernel (function dot_sum) summed up each of the values of a whole block such that the finished dot product is received.

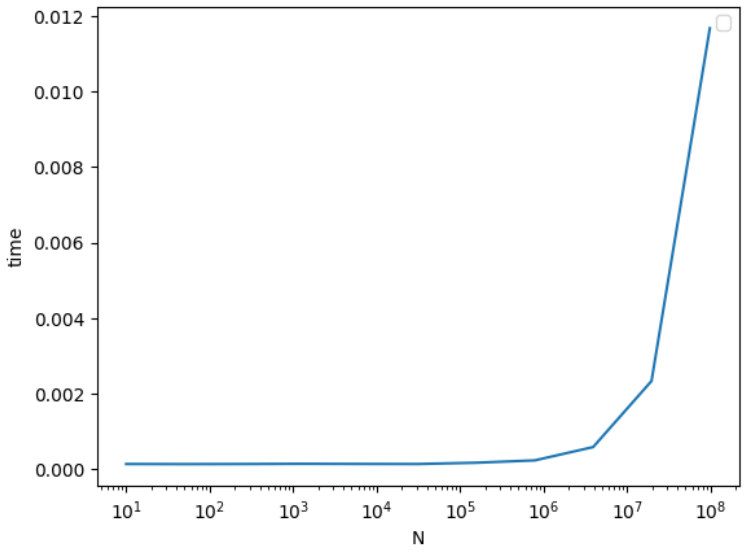

Figure 4: Time it takes to run the dot product (with two kernels) for different sizes of N

Appendix

basic_cuda_1a.cu

```
#include <iostream>
#include <vector>
#include "timer.hpp"

using Vector = std::vector<double>;

void measure_time(int N_min, int N_max, int repeats){

  Vector time_vec_malloc;
  Vector time_vec_free;

  double time_malloc, time_free;
  double *d_x;
  Timer timer;

  for (int i = N_min; i <= N_max; i = i*10){
    std::cout << i << std::endl;
    time_malloc = 0;
    time_free = 0;
    for (int j = 0; j < repeats+1; j++){
        cudaDeviceSynchronize();
        timer.reset();
        cudaMalloc(&d_x, i*sizeof(float));
        time_malloc += timer.get();
        //printf("Elapsed: %g\n", time_malloc);
        cudaDeviceSynchronize();
        timer.reset();
        cudaFree(d_x);
        time_free += timer.get();
        //printf("Elapsed: %g\n", time);

    }
    time_vec_malloc.push_back(time_malloc/repeats);
    time_vec_free.push_back(time_free/repeats);


  }
  printf("malloc\n");
  for (const auto& value : time_vec_malloc){
    printf("%g\n", value);
  }
  printf("\nfree\n");
  for (const auto& value : time_vec_free){
    printf("%g\n", value);
  }

}

int main(){
  measure_time(10, 100000000, 10);

  return EXIT_SUCCESS;
}
```

basic_cuda_1cde.cu

```
1
2  #include <iostream>
3  #include <stdio.h>
4  #include <vector>
5  #include "timer.hpp"
6
7  using Vector = std::vector<double>;
8
9
10 __global__ void addition(int n, double *x, double *y, double *z)
11 {
12   int id = blockIdx.x*blockDim.x + threadIdx.x;
13   for (size_t i = id; i < n; i += blockDim.x*gridDim.x){
14     z[i] = x[i] + y[i];
15   }
16 }
17
18 int main(void)
19 {
20
21     double *x, *y, *z, *d_x, *d_y, *d_z;
22
23     Vector time_vec;
24     double time;
25     Timer timer;
26
27     int N_min = 10000000;
28     int N_max = 10000000;
29     int repeats = 5;
30     for (int N = N_min; N <= N_max; N = N*5){
31         std::cout << N << std::endl;
32         time = 0;
33
34         // REPEAT
35         for (int j = 0; j < repeats; j++){
36
37             // Allocate host memory and initialize
38             x = (double*)malloc(N*sizeof(double));
39             y = (double*)malloc(N*sizeof(double));
40             z = (double*)malloc(N*sizeof(double));
41
42             for (int i = 0; i < N; i++) {
43                 x[i] = i;
44                 y[i] = N-1-i;
45                 z[i] = 0;
46             }
47
48             // Allocate device memory and copy host data over
49             cudaMalloc(&d_x, N*sizeof(double));
50             cudaMalloc(&d_y, N*sizeof(double));
51             cudaMalloc(&d_z, N*sizeof(double));
52
53             cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
54             cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
55             cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);
56
57             cudaDeviceSynchronize();
58             timer.reset();
59
60             // MEASURING TIME FROM HERE
```

```
61
62              addition<<<256, 1024>>>(N, d_x, d_y, d_z);
63
64              // TO HERE
65              cudaDeviceSynchronize();
66              time += timer.get();
67              //printf("Elapsed: %g\n", time);
68
69              cudaMemcpy(z, d_z, N*sizeof(double), cudaMemcpyDeviceToHost);
70
71              // CHECK IF VALUES OF z ARE EQUAL (fast check on first two values)
72              if (z[1] != z[2]){
73                  std::cout << "Values␣of␣z␣are␣not␣equal!!!!!" << std::endl;
74              }
75
76              cudaFree(d_x);
77              cudaFree(d_y);
78              cudaFree(d_z);
79              free(x);
80              free(y);
81              free(z);
82
83          }
84          time_vec.push_back(time/repeats);
85
86
87
88      }
89      printf("Times:\n");
90      for (const auto& value : time_vec){
91          std::cout << value << std::endl;
92      }
93
94
95
96      return EXIT_SUCCESS;
97  }
```

dot_product_2a.cu

```
1   #include <iostream>
2   #include <vector>
3   #include "timer.hpp"
4
5   using Vector = std::vector<double>;
6
7   const int threadsPerBlock = 256;
8
9   // KERNEL 2
10  __global__ void dot_partial(double *x, double *y, double *z_partial, int n)
11  {
12          __shared__ double temp_array[threadsPerBlock]; // shared for both kernels
13          double product = 0;
14          int id = blockIdx.x * blockDim.x + threadIdx.x;
15          for (int i = id; i < n; i += blockDim.x * gridDim.x)
16          {
17                  product += x[i] * y[i];
18          }
19          // add results as long as i < size of vectors
20          temp_array[threadIdx.x] = product;
```

```
21
22          // reduce in each block
23          for (int i = blockDim.x / 2; i > 0; i = i / 2)
24          {
25                  __syncthreads(); // sync threads in block
26
27                  // add elements
28                  if (threadIdx.x < i)
29                  {
30                          temp_array[threadIdx.x] += temp_array[threadIdx.x + i];
31                  }
32          }
33          // let only thread 0 write to memory
34          if (threadIdx.x == 0)
35          {
36                  z_partial[blockIdx.x] = temp_array[0];
37          }
38  }
39  // KERNEL 1
40  __global__ void dot_sum(double *z_partial)
41  {
42          for (int i = blockDim.x / 2; i > 0; i = i / 2)
43          {
44                  __syncthreads();
45
46                  if (threadIdx.x < i)
47                          z_partial[threadIdx.x] += z_partial[threadIdx.x + i];
48          }
49  }
50
51  int main(void)
52  {
53
54          double *x, *y, *z, *d_x, *d_y, *d_z_partial;
55          Vector time_vec;
56          Timer timer;
57
58          int N_min = 10;
59          int N_max = 100000000;
60          int repeats = 10;
61
62          for (int N = N_min; N <= N_max; N = N * 5)
63          {
64                  std::cout << N << std::endl;
65                  // Allocate host memory and initialize
66                  z = (double *)malloc(threadsPerBlock * sizeof(double));
67                  x = (double *)malloc(N * sizeof(double));
68                  y = (double *)malloc(N * sizeof(double));
69
70                  for (int i = 0; i < N; i++)
71                  {
72                          x[i] = 1;
73                          y[i] = 1;
74                  }
75
76                  // Allocate device memory and copy host data over
77                  cudaMalloc(&d_x, N * sizeof(double));
78                  cudaMalloc(&d_y, N * sizeof(double));
79                  cudaMalloc(&d_z_partial, threadsPerBlock * sizeof(double));
80
81                  cudaMemcpy(d_x, x, N * sizeof(double), cudaMemcpyHostToDevice);
82                  cudaMemcpy(d_y, y, N * sizeof(double), cudaMemcpyHostToDevice);
```

```
83                  // cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);
84
85                  // better to measure time over whole loop than inside
86                  //  MEASURING TIME FROM HERE
87                  timer.reset();
88                  for (int n = 0; n < repeats; n++)
89                  {
90                          dot_partial<<<256, threadsPerBlock>>>(d_x, d_y, d_z_partial, N);
91
92                          cudaDeviceSynchronize();
93                          dot_sum<<<1, threadsPerBlock>>>(d_z_partial);
94
95                          cudaMemcpy(z, d_z_partial, threadsPerBlock * sizeof(double), cudaMemcpyDevice
96                  }
97                  time_vec.push_back(timer.get() / repeats);
98                  // TO HERE
99
100                 cudaFree(d_x);
101                 cudaFree(d_y);
102                 cudaFree(d_z_partial);
103                 free(x);
104                 free(y);
105                 free(z);
106         }
107         printf("Times:\n");
108         for (const auto &value : time_vec)
109         {
110                 std::cout << value << std::endl;
111         }
112
113         return EXIT_SUCCESS;
114 }
```