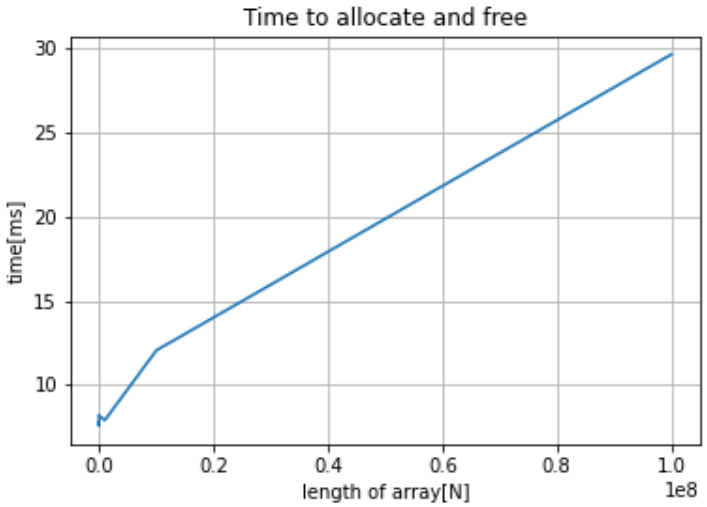
Computational Science on Many-Core Architectures Exercise 2

Example 1 Basic Cuda

a) init array

For the first point I used 8 different array sizes from $N = 10, 100, 1000, ...10^8$ and measures it's time to Malloc and Free them.



I run the code seven times and document the time results.

Listing 1: code for a)

```
#include <stdio.h>
#include "timer.hpp"

int main(void)
{
    int N = 10;
    double *d_x;
    Timer timer;

    timer.reset();
    for (int i = 0;i < 100; i++)
    {
        cudaMalloc(&d_x, N*sizeof(double));
        cudaFree(d_x);
    }
    printf("Malloc_Free_Time: %g[ms] N = %d\n", (1000*timer.get())/100,N);
return EXIT_SUCCESS;
}
```

For this meassure I get a dipendence of $O(1)$ linear behavior.

b) Init by kernel

The calculation the effective bandwidth looks like this:

$$Bw_{ef} = \frac{N \times 8bytes}{t}$$

where N is the number of elements of the array. The $8bytes$ stands for the individual double element.
In total the results for the different methods are:
Init by kernel: : 32.188[ms] for $N = 1e6 \rightarrow 248.53\frac{Mbytes}{s}$
Init by cudaMemcopy: 11.015[ms] for $N = 1e6 \rightarrow 726.28\frac{Mbytes}{s}$
Init by individual: 41729.4[ms] for $N = 1e3 \rightarrow 1.917 * 10^{-4}\frac{Mbytes}{s}$
For the last one I really do not know why it took so long.

Listing 2: code for b)

```
#include <stdio.h>
#include "timer.hpp"

#include <vector>
#include <iostream>

__global__ void init_kernel(int N)
{
    double *x, *y;

    x = new double [N];
    y = new double [N];

    for (int i = 0; i < N; i++)
    {
        x[i] = i;
        y[i] = N-i-1;
    }
}
int main(void)
{
    int N = 1000000;
    int M = 1;
    Timer timer;

    cudaDeviceSynchronize();
    timer.reset();

    init_kernel<<<(M+255)/256, 256>>>(N);
    cudaDeviceSynchronize();
    printf("Kernel_init_Time: %g[ms]\n", (1000*timer.get()));

    //Runtime 32.188[ms]
return EXIT_SUCCESS;
}
```

b) Init by cudaMemcopy

Listing 3: code for b)

```
1  # include <stdio.h>
2  # include "timer.hpp"
3
4  int main (void)
5  {
6      int N = 1000000;
7      double *x, *y, *d_x, *d_y;
8      Timer timer;
9
10     x = new double[N];
11     y = new double[N];
12
13     for (int i = 0; i < N; i++)
14     {
15         x[i] = i;
16         y[i] = N-i-1;
17     }
18
19     cudaDeviceSynchronize();
20     timer.reset();
21
22     cudaMalloc(&d_x, N*sizeof(double));
23     cudaMalloc(&d_y, N*sizeof(double));
24
25     cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
26     cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
27
28     cudaDeviceSynchronize();
29     printf("Kernel_init_Time: %g[ms]\n", (1000*timer.get()));
30
31     cudaFree(d_x);
32     cudaFree(d_y);
33     free(x);
34     free(y);
35 return EXIT_SUCCESS;
36 //11.189[ms]
37 }
```

b) Init by individual element cudaMemcopy

Listing 4: code for b)

```
1  # include <stdio.h>
2  # include "timer.hpp"
3
4  int main (void)
5  {
6      int N = 1000;
7      double *x, *y, *d_x, *d_y;
8      Timer timer;
9
10     x = new double[N];
11     y = new double[N];
12
13     for (int i = 0; i < N; i++)
14     {
15         x[i] = i;
16         y[i] = N-i-1;
17     }
18
19     cudaDeviceSynchronize();
20     timer.reset();
21
22     cudaMalloc(&d_x, N*sizeof(double));
23     cudaMalloc(&d_y, N*sizeof(double));
24     for (int i = 0; i < N;i++)
25     {
26         cudaMemcpy(d_x+i, x+i, 1*sizeof(double), cudaMemcpyHostToDevice);
27         cudaMemcpy(d_y+i, y+i, 1*sizeof(double), cudaMemcpyHostToDevice);
28     }
29
30
31     cudaDeviceSynchronize();
32     printf("Kernel_init_Time: %g[ms]\n", (1000*timer.get()));
33
34     cudaFree(d_x);
35     cudaFree(d_y);
36     free(x);
37     free(y);
38
39 return EXIT_SUCCESS;
40
41 //41729.4[ms] N = 1000
42 }
```

For that point I only test the program with $N = 1000$ to reduce the time to run the program. It is clear that this method is way slower than the other two.

c) Kernel to sum up two vectors

Listing 5: code for c)

```
1  # include <stdio.h>
2  # include "timer.hpp"
3
4  __global__ void SumOfVectors(double *x, double *y, double *z, int N)
5  {
6      int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
7
8      for (size_t i = thread_id; i < N; i += blockDim.x * gridDim.x)
9      {
10         z[i] = x[i] + y[i];
11     }
12 }
13
14 int main (void)
15 {
16     int N = 100;
17     int s = 16
18     double *x, *y, *z, *d_x, *d_y, *d_z;
19     Timer timer;
20
21     x = new double[N];
22     y = new double[N];
23     z = new double[N];
24
25     for (int i = 0; i < N; i++)
26     {
27         x[i] = i;
28         y[i] = N-i-1;
29         z[i] = 0;
30     }
31
32     cudaMalloc(&d_x, N*sizeof(double));
33     cudaMalloc(&d_y, N*sizeof(double));
34     cudaMalloc(&d_z, N*sizeof(double));
35
36     cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
37     cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
38     cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);
39
40     cudaDeviceSynchronize();
41     timer.reset();
42
43     SumOfVectors<<<s, s>>>(d_x, d_y, d_z, N);
44     cudaDeviceSynchronize();
45
46     cudaMemcpy(z, d_z, N*sizeof(double), cudaMemcpyDeviceToHost);
47
```

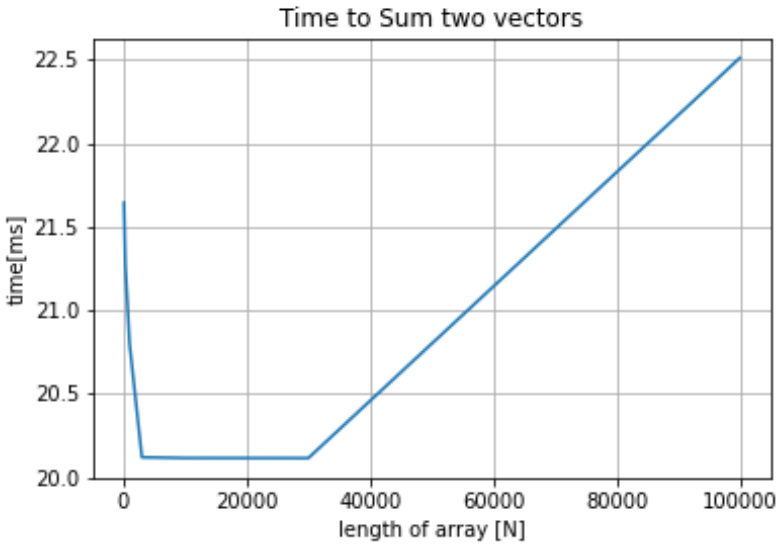
```
48      printf("SumTime: %g[ms]\n", (1000*timer.get()));
49      printf("FirstEntrieOfSumVec: %f\n",z[N]);
50
51
52      cudaFree(d_x);
53      cudaFree(d_y);
54      cudaFree(d_z);
55      free(x);
56      free(y);
57      free(z);
58  return EXIT_SUCCESS;
59  }
```

d) Kernel to sum up two vectors with different N

Listing 6: code for d)

```
1   # include <stdio.h>
2   # include "timer.hpp"
3
4   __global__ void SumOfVectors(double *x, double *y, double *z, int N)
5   {
6       int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
7
8       for (size_t i = thread_id; i < N; i += blockDim.x * gridDim.x)
9       {
10      z[i] = x[i] + y[i];
11      }
12  }
13
14  int main (void)
15  {
16      int N = 100;
17      int anz = 100;
18      double *x, *y, *z, *d_x, *d_y, *d_z;
19      Timer timer;
20
21      x = new double[N];
22      y = new double[N];
23      z = new double[N];
24
25      for (int i = 0; i < N; i++)
26      {
27          x[i] = i;
28          y[i] = N-i-1;
29          z[i] = 0;
30      }
31
32      cudaMalloc(&d_x, N*sizeof(double));
33      cudaMalloc(&d_y, N*sizeof(double));
```

```
34      cudaMalloc(&d_z, N*sizeof(double));
35
36      cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
37      cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
38      cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);
39
40      cudaDeviceSynchronize();
41      timer.reset();
42
43      for (int i = 0; i < anz; i++)
44      {
45          SumOfVectors<<<(N + 255)/256, 256>>>(d_x, d_y, d_z, N);
46          cudaDeviceSynchronize();
47      }
48      cudaDeviceSynchronize();
49      printf("MidSumTime: %g[ms]\n", (1000*timer.get())/anz);
50      cudaMemcpy(z, d_z, N*sizeof(double), cudaMemcpyDeviceToHost);
51
52      cudaFree(d_x);
53      cudaFree(d_y);
54      cudaFree(d_z);
55      free(x);
56      free(y);
57      free(z);
58
59  return EXIT_SUCCESS;
60  }
```



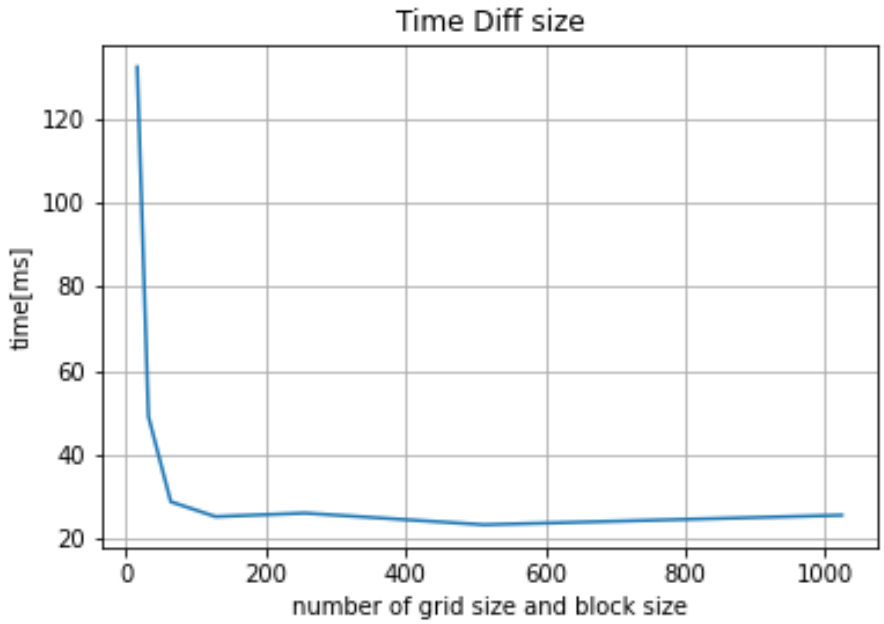
The time difference between small number of N and a large number of N is very small 2.5[ms]. I would expect a larger time difference.

e) Different size of blocks and grid

Listing 7: code for e)

```
1  # include <stdio.h>
2  # include "timer.hpp"
3
4  __global__ void SumOfVectors(double *x, double *y, double *z, int N)
5  {
6      int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
7
8      for (size_t i = thread_id; i < N; i += blockDim.x * gridDim.x)
9      {
10         z[i] = x[i] + y[i];
11     }
12 }
13
14 int main (void)
15 {
16     int N = 10000000;
17     int s = 16;
18     int anz = 10;
19     double *x, *y, *z, *d_x, *d_y, *d_z;
20     Timer timer;
21
22     x = new double[N];
23     y = new double[N];
24     z = new double[N];
25
26     for (int i = 0; i < N; i++)
27     {
28         x[i] = i;
29         y[i] = N-i-1;
30         z[i] = 0;
31     }
32
33     cudaMalloc(&d_x, N*sizeof(double));
34     cudaMalloc(&d_y, N*sizeof(double));
35     cudaMalloc(&d_z, N*sizeof(double));
36     cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
37     cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
38     cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);
39     cudaDeviceSynchronize();
40     timer.reset();
41     for (int i = 0; i < anz; i++)
42     {
43         SumOfVectors<<<s, s>>>(d_x, d_y, d_z, N);
44         cudaDeviceSynchronize();
45     }
46     cudaMemcpy(z, d_z, N*sizeof(double), cudaMemcpyDeviceToHost);
47
```

```
48      printf("SumTime: %g[ms]\n", (1000*timer.get())/anz);
49      printf("FirstEntrieOfSumVec: %f\n",z[N]);
50
51      cudaFree(d_x);
52      cudaFree(d_y);
53      cudaFree(d_z);
54      free(x);
55      free(y);
56      free(z);
57  return EXIT_SUCCESS;
58  }
```



There is nearly no time saving after a number of grid and block size of 64. So the jump from 16 to 32 is ruffly 90ms time save.

Example 2 Dot Product

To start with this code I orientated me at the following source[1].

a) use two GPUs

Listing 8: code for a)

```
1  # include <stdio.h>
2  # include "timer.hpp"
3
4  __global__ void dot_pro_first(double *x, double *y, double *tmp, unsigned int N)
5  {
6      unsigned int ind = threadIdx.x + blockDim.x*blockIdx.x;
7      unsigned int str = blockDim.x*gridDim.x;
8
9      __shared__ double cache[256];
10     double tmpsum = 0.0;
11     while(ind < N)
12     {
13         tmpsum += x[ind]*y[ind];
14         ind += str;
15     }
16     cache[threadIdx.x] = tmpsum;
17     __syncthreads();
18     for(int i = blockDim.x/2; i>0; i/=2)
19     {
20             __syncthreads();
21         if(threadIdx.x < i)
22         {
23             cache[threadIdx.x] += cache[threadIdx.x + i];
24         }
25     }
26
27     if(threadIdx.x == 0)
28     {
29         tmp[blockIdx.x] = cache[0];
30     }
31 }
32
33 __global__ void dot_pro_second(double *tmp, double *dot_prd)
34 {
35     for (int i = blockDim.x/2; i > 0; i/=2)
36     {
37         if(threadIdx.x < i)
38         {
39             tmp[threadIdx.x] += tmp[threadIdx.x + i];
40         }
41     }
42     __syncthreads();
```

[1] https://bitbucket.org/jsandham/algorithms_in_cuda/src/master/dot_product/

```
43
44      if(threadIdx.x == 0)
45      {
46          *dot_prd = tmp[0];
47      }
48  }
49
50  int main (void)
51  {
52      int N = 10000;
53      int s = 256;
54      int anz = 10;
55      double *px, *py, *d_px, *d_py;
56      double *prod, *d_prod, *d_tmp;
57      Timer timer;
58
59      prod = new double[N];
60      px = new double[N];
61      py = new double[N];
62
63      for (int i = 0; i < N; i++)
64      {
65          px[i] = 1;
66          py[i] = 3;
67      }
68      cudaMalloc(&d_px, N*sizeof(double));
69      cudaMalloc(&d_py, N*sizeof(double));
70      cudaMalloc(&d_prod, sizeof(double));
71      cudaMalloc(&d_tmp, s*sizeof(double));
72      cudaMemcpy(d_px, px, N*sizeof(double), cudaMemcpyHostToDevice);
73      cudaMemcpy(d_py, py, N*sizeof(double), cudaMemcpyHostToDevice);
74      cudaDeviceSynchronize();
75      timer.reset();
76      for (int i = 0; i < anz; i++)
77      {
78          dot_pro_first<<<s, s>>>(d_px, d_py, d_tmp, N);
79          dot_pro_second<<<1, s>>>(d_tmp, d_prod);
80          cudaDeviceSynchronize();
81      }
82      cudaMemcpy(prod, d_prod, sizeof(double), cudaMemcpyDeviceToHost);
83
84      printf("Time: %g[ms] result: %f\n", (1000*timer.get())/anz,*prod);
85      cudaFree(d_px);
86      cudaFree(d_py);
87      cudaFree(d_prod);
88      free(px);
89      free(py);
90      free(prod);
91  return EXIT_SUCCESS;
92  }
```

b) use a GPU and the CPU

Listing 9: code for a)

```
1  # include <stdio.h>
2  # include "timer.hpp"
3  //# include <"random">
4
5  __global__ void dot_pro(double *x, double *y, double *tmp, unsigned int N)
6  {
7      unsigned int ind = threadIdx.x + blockDim.x*blockIdx.x;
8      unsigned int str = blockDim.x*gridDim.x;
9
10     __shared__ double cache[256];
11
12     double tmpsum = 0.0;
13     while(ind < N)
14     {
15         tmpsum += x[ind]*y[ind];
16         ind += str;
17     }
18
19     cache[threadIdx.x] = tmpsum;
20
21     __syncthreads();
22
23     for(int i = blockDim.x/2; i>0; i/=2)
24     {
25         __syncthreads();
26         if(threadIdx.x < i)
27         {
28             cache[threadIdx.x] += cache[threadIdx.x + i];
29         }
30     }
31
32     if(threadIdx.x == 0)
33     {
34         tmp[blockIdx.x] = cache[0];
35     }
36 }
37
38 int main (void)
39 {
40     int N = 10000;
41     int s = 256;
42     int anz = 10;
43     double *px, *py, *d_px, *d_py;
44     double *prod, *d_prod;
45     double *tmp, *d_tmp;
46     double sumdot = 0;
47     Timer timer;
```

```
48
49      prod = new double[N];
50      px = new double[N];
51      py = new double[N];
52      tmp = new double[s];
53
54
55      for (int i = 0; i < N; i++)
56      {
57          px[i] = 1;
58          py[i] = 3;
59      }
60      cudaMalloc(&d_px, N*sizeof(double));
61      cudaMalloc(&d_py, N*sizeof(double));
62      cudaMalloc(&d_prod, sizeof(double));
63      cudaMalloc(&d_tmp, s*sizeof(double));
64      cudaMemset(d_prod, 0.0, sizeof(double));
65
66      cudaMemcpy(d_px, px, N*sizeof(double), cudaMemcpyHostToDevice);
67      cudaMemcpy(d_py, py, N*sizeof(double), cudaMemcpyHostToDevice);
68
69
70      cudaDeviceSynchronize();
71      timer.reset();
72      for (int i = 0; i < anz; i++)
73      {
74          dot_pro<<<s, s>>>(d_px, d_py, d_tmp, N);
75          cudaDeviceSynchronize();
76
77          cudaMemcpy(tmp, d_tmp, s*sizeof(double), cudaMemcpyDeviceToHost);
78
79          for(int j = 0; j < s; j++)
80          {
81              sumdot += tmp[j];
82          }
83      }
84      printf("Time: %g[ms] result: %f\n", (1000*timer.get())/anz,sumdot/anz);
85
86      cudaFree(d_px);
87      cudaFree(d_py);
88      cudaFree(d_prod);
89      free(px);
90      free(py);
91      free(prod);
92
93  return EXIT_SUCCESS;
94  }
```

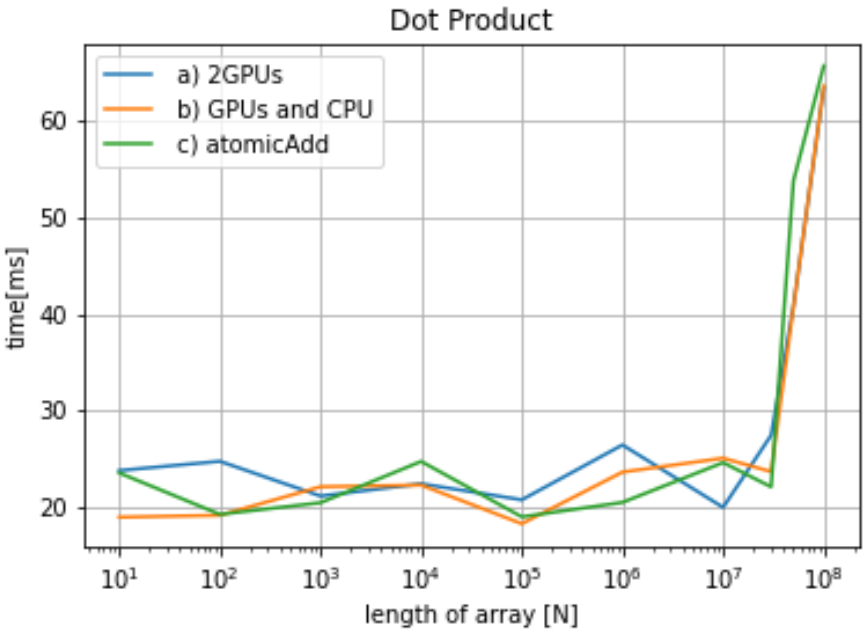
c) use atomicAdd

Listing 10: code for a)

```
1  # include <stdio.h>
2  # include "timer.hpp"
3  //# include <"random">
4
5  __global__ void dot_pro(double *x, double *y, double *dot, unsigned int N)
6  {
7      unsigned int ind = threadIdx.x + blockDim.x*blockIdx.x;
8      unsigned int str = blockDim.x*gridDim.x;
9
10     __shared__ double cache[256];
11
12     double tmpsum = 0.0;
13     while(ind < N)
14     {
15         tmpsum += x[ind]*y[ind];
16         ind += str;
17     }
18
19     cache[threadIdx.x] = tmpsum;
20
21     __syncthreads();
22
23     for(int i = blockDim.x/2; i>0; i/=2)
24     {
25         __syncthreads();
26         if(threadIdx.x < i)
27         {
28             cache[threadIdx.x] += cache[threadIdx.x + i];
29         }
30     }
31
32     if(threadIdx.x == 0)
33     {
34         atomicAdd(dot,cache[0]);
35     }
36 }
37
38 int main (void)
39 {
40     int N = 10;
41     int s = 256;
42     int anz = 10;
43     double *px, *py, *d_px, *d_py;
44     double *prod, *d_prod;
45
46     Timer timer;
47
```

```
48      prod = new double[N];
49      px = new double[N];
50      py = new double[N];
51
52
53
54      for (int i = 0; i < N; i++)
55      {
56          px[i] = 1;
57          py[i] = 3;
58      }
59      cudaMalloc(&d_px, N*sizeof(double));
60      cudaMalloc(&d_py, N*sizeof(double));
61      cudaMalloc(&d_prod, sizeof(double));
62      cudaMemset(d_prod, 0.0, sizeof(double));
63
64      cudaMemcpy(d_px, px, N*sizeof(double), cudaMemcpyHostToDevice);
65      cudaMemcpy(d_py, py, N*sizeof(double), cudaMemcpyHostToDevice);
66
67
68      cudaDeviceSynchronize();
69      timer.reset();
70      for (int i = 0; i < anz; i++)
71      {
72          dot_pro<<<s, s>>>(d_px, d_py, d_prod, N);
73          cudaDeviceSynchronize();
74          cudaMemcpy(prod, d_prod, sizeof(double), cudaMemcpyDeviceToHost);
75      }
76      printf("Time: %g[ms] result: %f\n", (1000*timer.get())/anz,*prod/anz);
77
78      cudaFree(d_px);
79      cudaFree(d_py);
80      cudaFree(d_prod);
81      free(px);
82      free(py);
83      free(prod);
84
85  return EXIT_SUCCESS;
86  }
```

explenarision



In the end all three different methods needs approximately the same amount of time. I think the third method should be the fastest method.