VIENNA UNIVERSITY OF TECHNOLOGY

360.252 COMPUTATIONAL SCIENCE ON MANY CORE ARCHITECTURES

INSTITUTE FOR MICROELECTRONICS

Exercise 2

Authors:
Camilo TELLO FACHIN
12127084

Supervisor:
Dipl.-Ing. Dr.techn. Edoardo
BONETTI

November 1, 2022

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Abstract

Here documented the results of exercise 2.

Contents

1   Basic Cuda Tasks

1.1   Task a
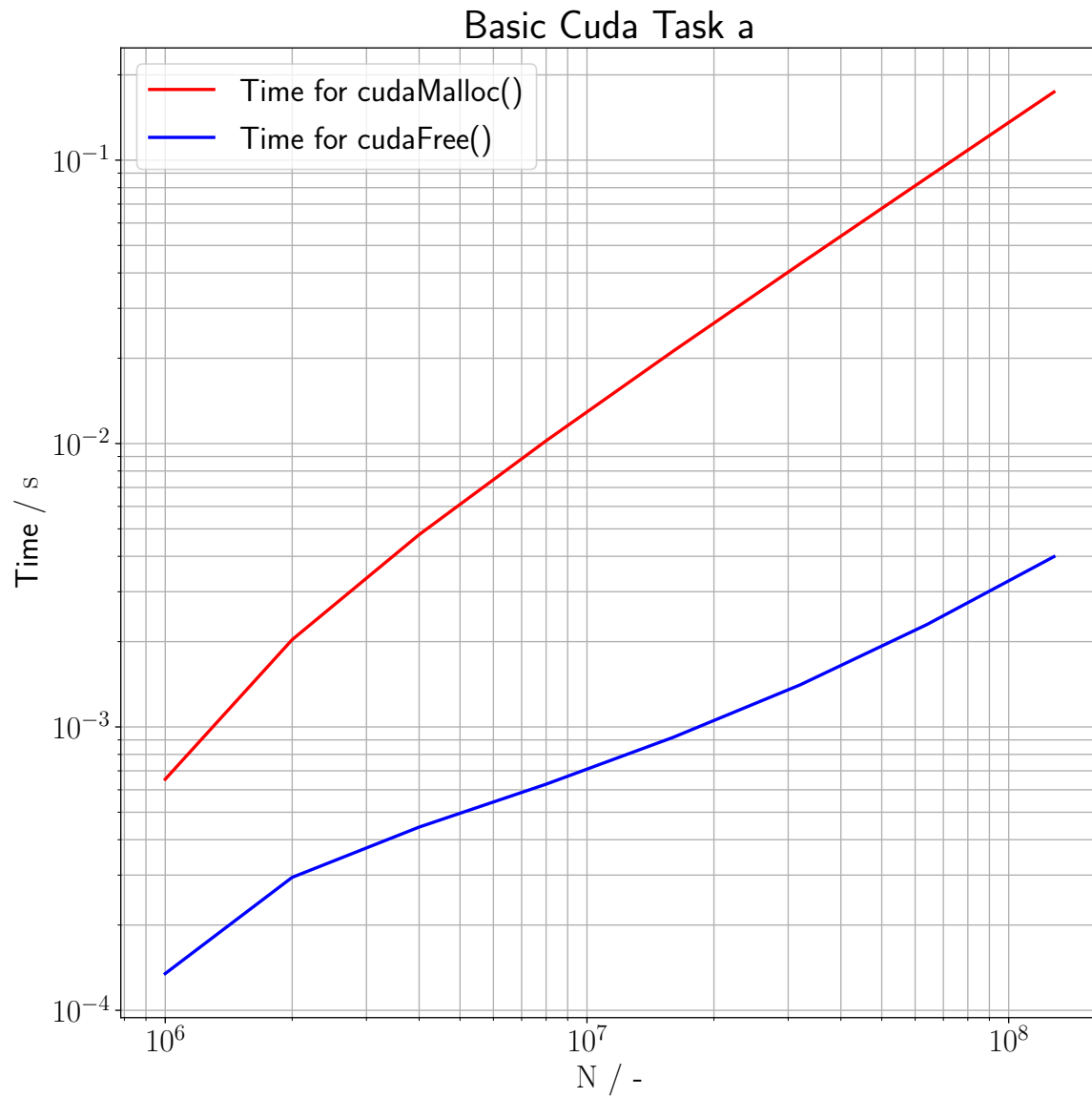

Figure 1: Plot for task 1a - see code in appendix A

1.2   Task b
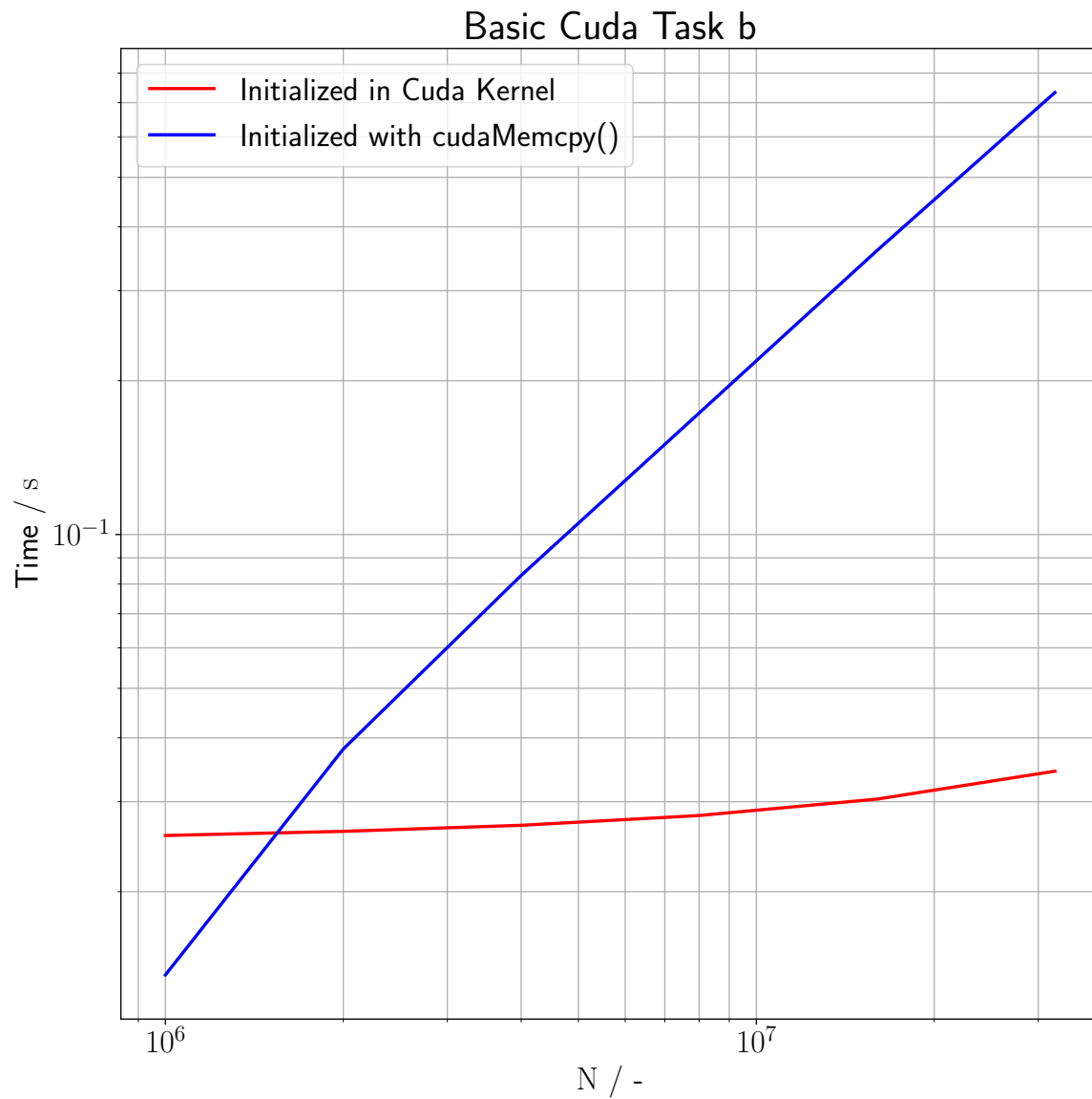
Figure 2: Plot for task 1b - see code in appendix B

1.3   Task c

See Implementation in appendix C.

1.4   Task d
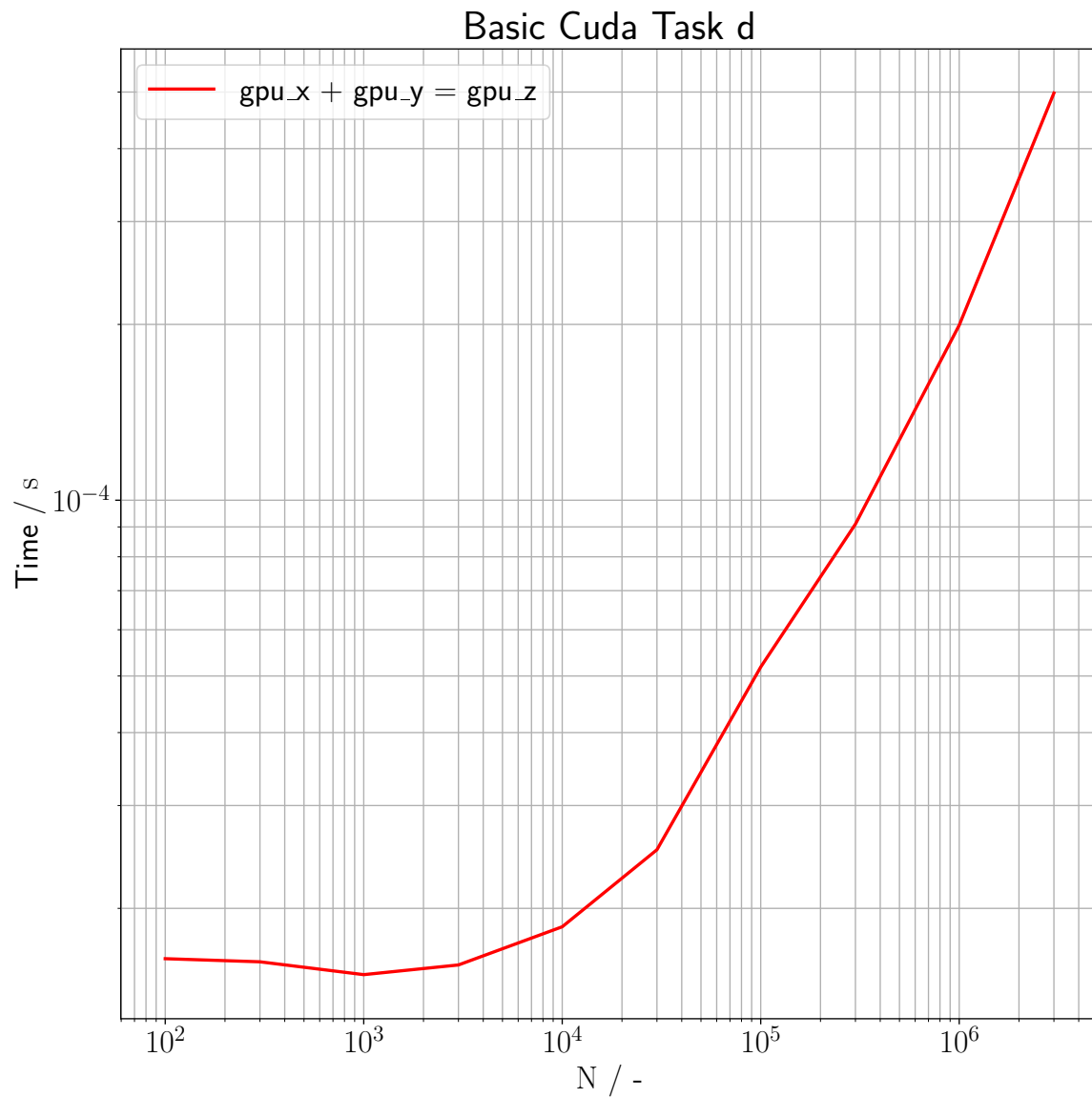
Figure 3: Plot for task 1d - See code in appendix C

Until $N = 10^4$, the runtime stays approximately the same, after that, a significant increase in runtime can be observed.

1.5   Task e
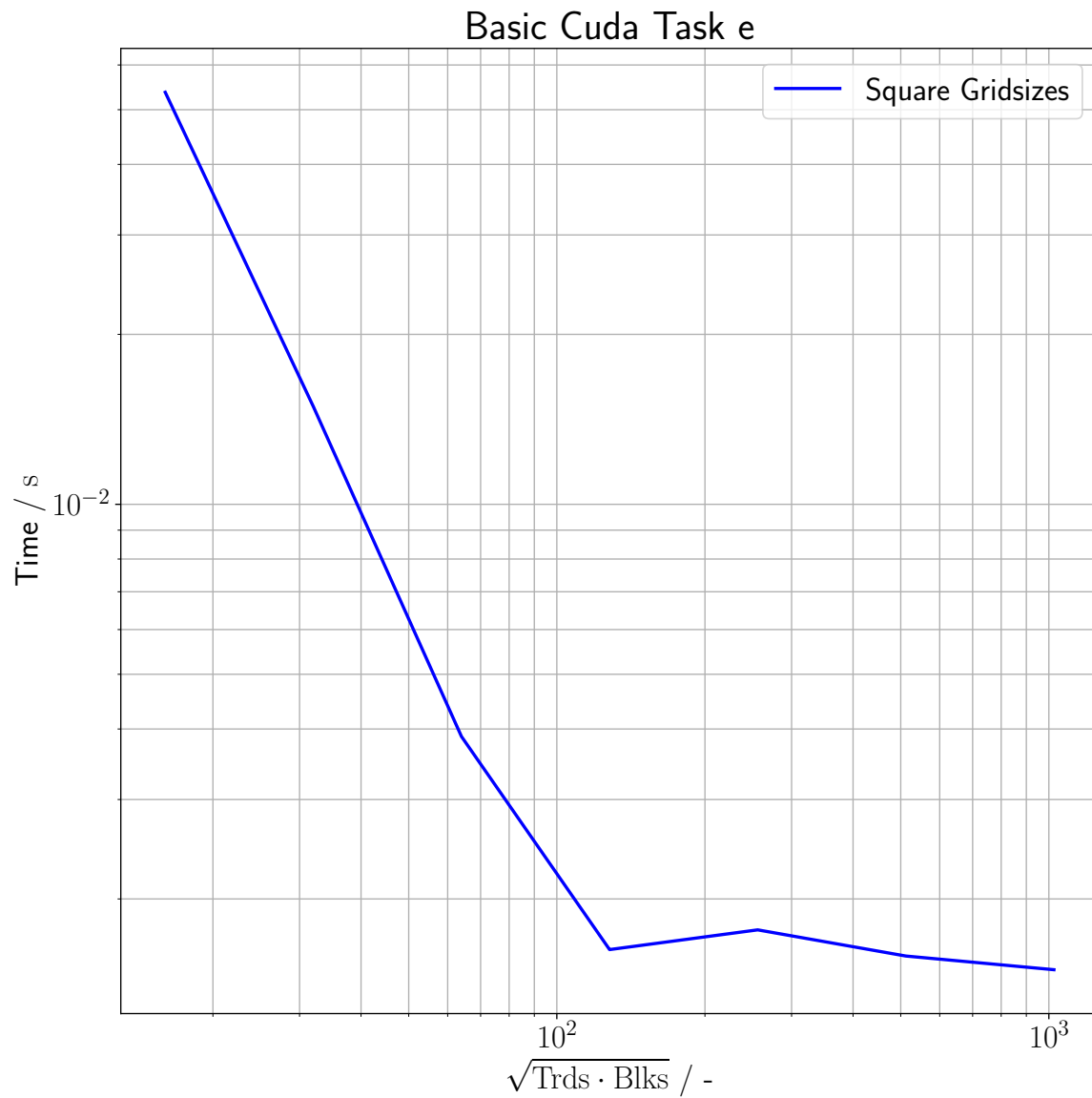
Figure 4: Plot for Task 1e - See code in appendix C

For $\sqrt{\text{Trds} \cdot \text{Blks}} < 128$, there is a signifcant performance decline.

2 Dot Product with Cuda

2.1 Task a and b
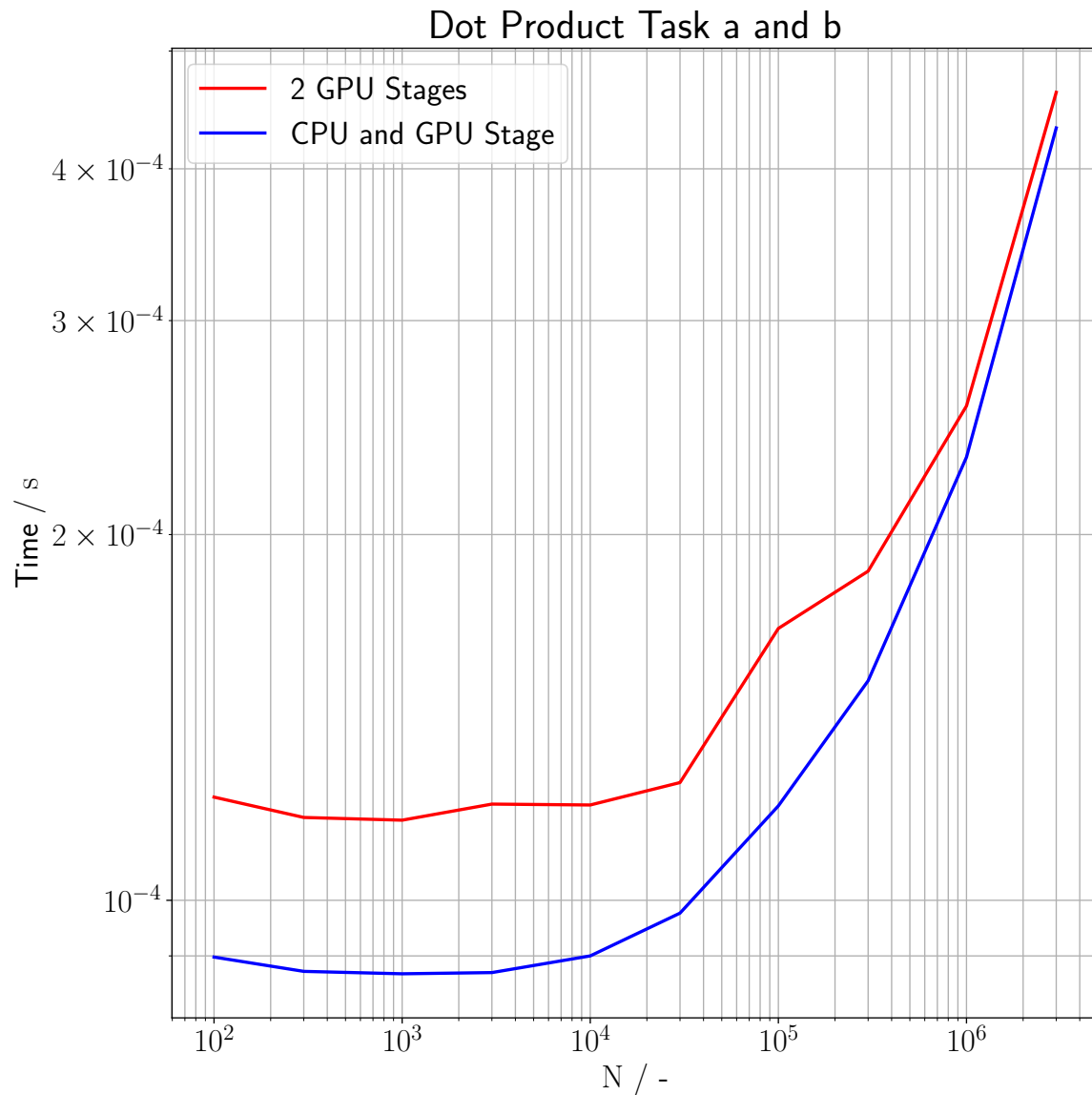
Figure 5: Plots for the Dot Product Task - See code in appendix D and E

Computational tasks which benefit from parallel reduction operations are faster on the given CPU.

A   CPP CUDA Code - Basic Cuda - Task 1a

C++ Listing

```cpp
#include <stdio.h>
#include "timer.hpp"
int main(void)
{
  int k=0, i=8;
  int N_values[i] = {1000000, 2000000, 4000000, 8000000, 16000000, 32000000, 64000000, 128000000};
  double *gpu_x;
  float  t_malloc=0, t_free=0;
  Timer timer;
  printf ("\nsize, malloc, free\n");
  while(k < i) {
    int N = N_values[k];
    for (int n=0; n<5; n++) {
      timer. reset ();
      cudaMalloc(&gpu_x, N*sizeof(double));
      cudaDeviceSynchronize();
      t_malloc += timer.get();
      timer. reset ();
      cudaFree(gpu_x);
      cudaDeviceSynchronize();
      t_free += timer.get();
    }
    printf ("%d,%g,%g\n", N, 0.2*t_malloc, 0.2*t_free);
    k++;
  }
  return EXIT_SUCCESS;
}
```

B   CPP CUDA Code - Basic Cuda - Task 1b

<div align="center">C++ Listing</div>

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

__global__ void initVec (double *vec1, double *vec2, int N) {
  unsigned int total_threads  = blockDim.x * gridDim.x;
  unsigned int global_tid  = blockIdx.x * blockDim.x + threadIdx.x;

  for (unsigned int i  = global_tid ; i<N; i += total_threads) {
    vec1[i] = (double)(i);
    vec2[i] = (double)(N-i-1);
  }
}

int main(void)
{
  int k=0, i=6;
  int N_values[i] = { 1000000, 2000000, 4000000, 8000000, 16000000, 32000000 };
  double *x, *y, *gpu_x, *gpu_y;
  Timer timer;
  float option_1=0;
  k = 0;
  printf ("\nsize,option1\n");
  while(k < i) {
    int N = N_values[k];
    for (int n=0; n<5; n++) {
      timer.reset ();
      x = (double*)malloc(N*sizeof(double));
      y = (double*)malloc(N*sizeof(double));
      cudaMalloc(&gpu_x, N*sizeof(double));
      cudaMalloc(&gpu_y, N*sizeof(double));
      initVec<<<256, 256>>>(gpu_x, gpu_y, N);
      cudaDeviceSynchronize();
      option_1 += timer.get();
    }
    printf ("%d,%g\n", N, 0.2*option_1);
    cudaFree(gpu_x);
    cudaFree(gpu_y);
    free (x);
    free (y);
    k++;
  }
  float option_2=0;
  k = 0;
  printf ("\nsize,option2\n");
  while(k < i) {
    int N = N_values[k];
    for (int n=0; n<5; n++) {
```

```
48        timer. reset ();
49        cudaDeviceSynchronize();
50        x = (double*)malloc(N*sizeof(double));
51        y = (double*)malloc(N*sizeof(double));
52        for (int i = 0; i < N; i++) {
53          x[i] = (double)(i);
54          y[i] = (double)(N−i−1);
55        }
56        cudaMalloc(&gpu_x, N*sizeof(double));
57        cudaMalloc(&gpu_y, N*sizeof(double));
58        cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
59        cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
60        cudaDeviceSynchronize();
61        option_2 += timer.get();
62      }
63      printf ("%d,%g\n", N, 0.2*option_2);
64      cudaMemcpy(x, gpu_x, N*sizeof(double), cudaMemcpyDeviceToHost);
65      cudaMemcpy(y, gpu_y, N*sizeof(double), cudaMemcpyDeviceToHost);
66      cudaFree(gpu_x);
67      cudaFree(gpu_y);
68      free (x);
69      free (y);
70      k++;
71    }
72    return EXIT_SUCCESS;
73 }
```

C   CPP CUDA Code - Basic Cuda - Task 1c, 1d, 1e

C++ Listing

```cpp
1   #include <stdio.h>
2   #include "timer.hpp"
3   __global__ void addVec(double *x, double *y, double *z, int N) {
4     unsigned int total_threads = blockDim.x * gridDim.x;
5     unsigned int global_tid = blockIdx.x * blockDim.x + threadIdx.x;
6     for (unsigned int i = global_tid ; i<N; i += total_threads) {
7       z[i] = x[i] + y[i];
8     }
9   }
10  int main(void)
11  {
12    // Task c //
13    double *x, *y, *z, *gpu_x, *gpu_y, *gpu_z;
14    Timer timer;
15    int N = 100;
16    x = (double*)malloc(N*sizeof(double));
17    y = (double*)malloc(N*sizeof(double));
18    z = (double*)malloc(N*sizeof(double));
19    for (int i = 0; i < N; i++) {
20      x[i] = (double)(i);
21      y[i] = (double)(N-i-1);
22    }
23    cudaMalloc(&gpu_x, N*sizeof(double));
24    cudaMalloc(&gpu_y, N*sizeof(double));
25    cudaMalloc(&gpu_z, N*sizeof(double));
26    cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
27    cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
28    addVec<<<256, 256>>>(gpu_x, gpu_y, gpu_z, N);
29    cudaMemcpy(z, gpu_z, N*sizeof(double), cudaMemcpyDeviceToHost);
30    cudaFree(gpu_x);
31    cudaFree(gpu_y);
32    cudaFree(gpu_z);
33    free(x);
34    free(y);
35    free(z);
36
37    // Task d //
38    int k = 0;
39    int N_values[10] = { 100, 300, 1000, 3000, 10000, 30000, 100000, 300000, 1000000, 3000000 };
40    printf("\nsize,time\n");
41    while(k < 10) {
42      float t_kernel =0;
43      int N = N_values[k];
44      x = (double*)malloc(N*sizeof(double));
45      y = (double*)malloc(N*sizeof(double));
46      z = (double*)malloc(N*sizeof(double));
47      for (int i = 0; i < N; i++) {
```

```
48         x[i] = (double)(i);
49         y[i] = (double)(N−i−1);
50       }
51       cudaMalloc(&gpu_x, N*sizeof(double));
52       cudaMalloc(&gpu_y, N*sizeof(double));
53       cudaMalloc(&gpu_z, N*sizeof(double));
54       cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
55       cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
56       timer.reset();
57       for (int n=0; n<5; n++) {
58         addVec<<<256, 256>>>(gpu_x, gpu_y, gpu_z, N);
59         cudaDeviceSynchronize();
60       }
61       t_kernel += timer.get();
62       printf("%d,%g\n", N, 0.2*t_kernel);
63       cudaMemcpy(z, gpu_z, N*sizeof(double), cudaMemcpyDeviceToHost);
64       cudaFree(gpu_x);
65       cudaFree(gpu_y);
66       cudaFree(gpu_z);
67       free(x);
68       free(y);
69       free(z);
70       k++;
71     }
72     // Task e //
73     N = 10000000;
74     k = 0;
75     int params[7] = { 16, 32, 64, 128, 256, 512, 1024};
76     printf("\nsqrt(threads),time\n");
77     while(k < 7) {
78       float t_kernel =0;
79       int param = params[k];
80       x = (double*)malloc(N*sizeof(double));
81       y = (double*)malloc(N*sizeof(double));
82       z = (double*)malloc(N*sizeof(double));
83       for (int i = 0; i < N; i++) {
84         x[i] = (double)(i);
85         y[i] = (double)(N−i−1);
86       }
87       cudaMalloc(&gpu_x, N*sizeof(double));
88       cudaMalloc(&gpu_y, N*sizeof(double));
89       cudaMalloc(&gpu_z, N*sizeof(double));
90       cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
91       cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
92       timer.reset();
93       for (int n=0; n<5; n++) {
94         addVec<<<param, param>>>(gpu_x, gpu_y, gpu_z, N);
95         cudaDeviceSynchronize();
96       }
97       t_kernel += timer.get();
98       printf("%d,%g\n", param, 0.2*t_kernel);
```

```
 99        cudaMemcpy(z, gpu_z, N*sizeof(double), cudaMemcpyDeviceToHost);
100        cudaFree(gpu_x);
101        cudaFree(gpu_y);
102        cudaFree(gpu_z);
103         free(x);
104         free(y);
105         free(z);
106        k++;
107     }
108     return EXIT_SUCCESS;
109  }
```

D   CPP CUDA Code - Dot Product - Task 2a

C++ Listing

```cpp
1   #include <stdio.h>
2   #include "timer.hpp"
3
4   const int threads_per_block = 256;
5   double dot_cpu(double *a, double *b, int N) {
6       double product = 0;
7       for (int i = 0; i < N; i++)
8       product = product + a[i] * b[i];
9       return product;
10  }
11  __global__ void dotVec_one(double *x, double *y, double * partial_z , int N) {
12      __shared__ double temp_arr[threads_per_block ];
13      double thread_product = 0;
14      unsigned int global_tid = threadIdx.x + blockIdx.x * blockDim.x;
15      unsigned int local_tid = threadIdx.x;
16      unsigned int total_threads = blockDim.x * gridDim.x;
17      for (unsigned int i=global_tid ; i<N; i+=total_threads) {
18          thread_product += x[i] * y[i ];
19      }
20      temp_arr[ local_tid ] = thread_product;
21      for (unsigned int stride = blockDim.x/2; stride>0; stride /=2) {
22          __syncthreads ();
23          if (threadIdx.x < stride) {
24              temp_arr[threadIdx.x] += temp_arr[threadIdx.x + stride ];
25          }
26      }
27      if (threadIdx.x == 0) {
28          partial_z [blockIdx.x] = temp_arr[0];
29      }
30  }
31  __global__ void dotVec_two(double * partial_z ) {
32      for (int stride = blockDim.x/2; stride>0; stride /=2) {
33          __syncthreads ();
34          if (threadIdx.x < stride)
35              partial_z [threadIdx.x] += partial_z[threadIdx.x+stride ];
36      }
37  }
38  int main(void)
39  {
40      // Task a //
41      double *x, *y, *z;
42      double *gpu_x, *gpu_y, * gpu_partial_z ;
43      Timer timer;
44      int k = 0;
45      int N_values_d [10] = { 100, 300, 1000, 3000, 10000, 30000, 100000, 300000, 1000000, 3000000 };
46      printf ("\nsize ,time\n");
47      while(k < 10) {
```

```
48        int N = N_values_d[k];
49        x = (double*)malloc(N*sizeof(double));
50        y = (double*)malloc(N*sizeof(double));
51        z = (double*)malloc(threads_per_block*sizeof(double));
52        for (int i = 0; i < N; i++) {
53          x[i] = 1.0;
54          y[i] = 1.0;
55        }
56        cudaMalloc(&gpu_x, N*sizeof(double));
57        cudaMalloc(&gpu_y, N*sizeof(double));
58        cudaMalloc(&gpu_partial_z, threads_per_block *sizeof(double));
59        cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
60        cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
61        timer.reset();
62        for (int n=0; n<5; n++) {
63          dotVec_one<<<256, threads_per_block>>>(gpu_x, gpu_y, gpu_partial_z, N);
64          cudaDeviceSynchronize();
65          dotVec_two<<<1, threads_per_block>>>(gpu_partial_z);
66          cudaMemcpy(z, gpu_partial_z, threads_per_block *sizeof(double), cudaMemcpyDeviceToHost);
67        }
68        printf("%g,%g\n", z[0], 0.2*timer.get());
69        cudaFree(gpu_x);
70        cudaFree(gpu_y);
71        cudaFree(gpu_partial_z);
72        free(x);
73        free(y);
74        free(z);
75        k++;
76      }
77      return EXIT_SUCCESS;
78    }
```

E   CPP CUDA Code - Dot Product - Task 2b

C++ Listing

```
1   #include <stdio.h>
2   #include "timer.hpp"
3
4   const int threads_per_block = 256;
5   double dotVec_cpu(double *a, double *b, int N) {
6       double product = 0;
7       for (int i = 0; i < N; i++)
8       product = product + a[i] * b[i];
9       return product;
10  }
11   __global__ void dotVec_gpu(double *x, double *y, double * partial_z , int N) {
12      __shared__ double temp_arr[ threads_per_block ];
13      double thread_product = 0;
14      unsigned int global_tid = threadIdx.x + blockIdx.x * blockDim.x;
15      unsigned int local_tid  = threadIdx.x;
16      unsigned int total_threads = blockDim.x * gridDim.x;
17      for (unsigned int i=global_tid; i<N; i+=total_threads) {
18          thread_product += x[i] * y[i];
19      }
20      temp_arr[ local_tid ] = thread_product;
21      for (unsigned int stride = blockDim.x/2; stride>0; stride /=2) {
22          __syncthreads ();
23          if (threadIdx.x < stride) {
24              temp_arr[threadIdx.x] += temp_arr[threadIdx.x + stride ];
25          }
26      }
27      if (threadIdx.x == 0) {
28          partial_z [blockIdx.x] = temp_arr[0];
29      }
30  }
31
32  int main(void)
33  {
34      // Task b //
35      double *x, *y, z, * partial_z ;
36      double *gpu_x, *gpu_y, * gpu_partial_z ;
37      Timer timer;
38      int k = 0;
39      int N_values_d [10] = {100, 300, 1000, 3000, 10000, 30000, 100000, 300000, 1000000, 3000000};
40       printf ("\nsize,time\n");
41      while(k < 10) {
42          int N = N_values_d[k];
43          x = (double*)malloc(N*sizeof(double));
44          y = (double*)malloc(N*sizeof(double));
45           partial_z = (double*)malloc(256*sizeof(double));
46          for (int i = 0; i < N; i++) {
47              x[i] = 1.0;
```

```
48        y[i] = 1.0;
49      }
50      cudaMalloc(&gpu_x, N*sizeof(double));
51      cudaMalloc(&gpu_y, N*sizeof(double));
52      cudaMalloc(&gpu_partial_z, 256*sizeof(double));
53      cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
54      cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
55      timer.reset();
56      for (int n=0; n<5; n++) {
57        dotVec_gpu<<<256, 256>>>(gpu_x, gpu_y, gpu_partial_z, N);
58        cudaMemcpy(partial_z, gpu_partial_z, 256*sizeof(double), cudaMemcpyDeviceToHost);
59        z = 0;
60        for(int i=0; i<256; i++) {
61          z += partial_z[i];
62        }
63      }
64      printf("%d,%g\n", N, 0.2*timer.get());
65      cudaFree(gpu_x);
66      cudaFree(gpu_y);
67      cudaFree( gpu_partial_z );
68      free(x);
69      free(y);
70      free( partial_z );
71      k++;
72    }
73    return EXIT_SUCCESS;
74  }
```