Vienna University of Technology

360.252 Computational Science on Many Core Architectures

Institute for Microelectronics

# Exercise 3

*Authors:*
Camilo Tello Fachin
12127084

*Supervisor:*
Dipl.-Ing. Dr.techn. Karl Rupp

October 28, 2022

TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

# Abstract

Here documented the results of exercise 3.

# Contents

# 1 Strided and Offset Memory Access (3 Points)

## 1.1 Task a

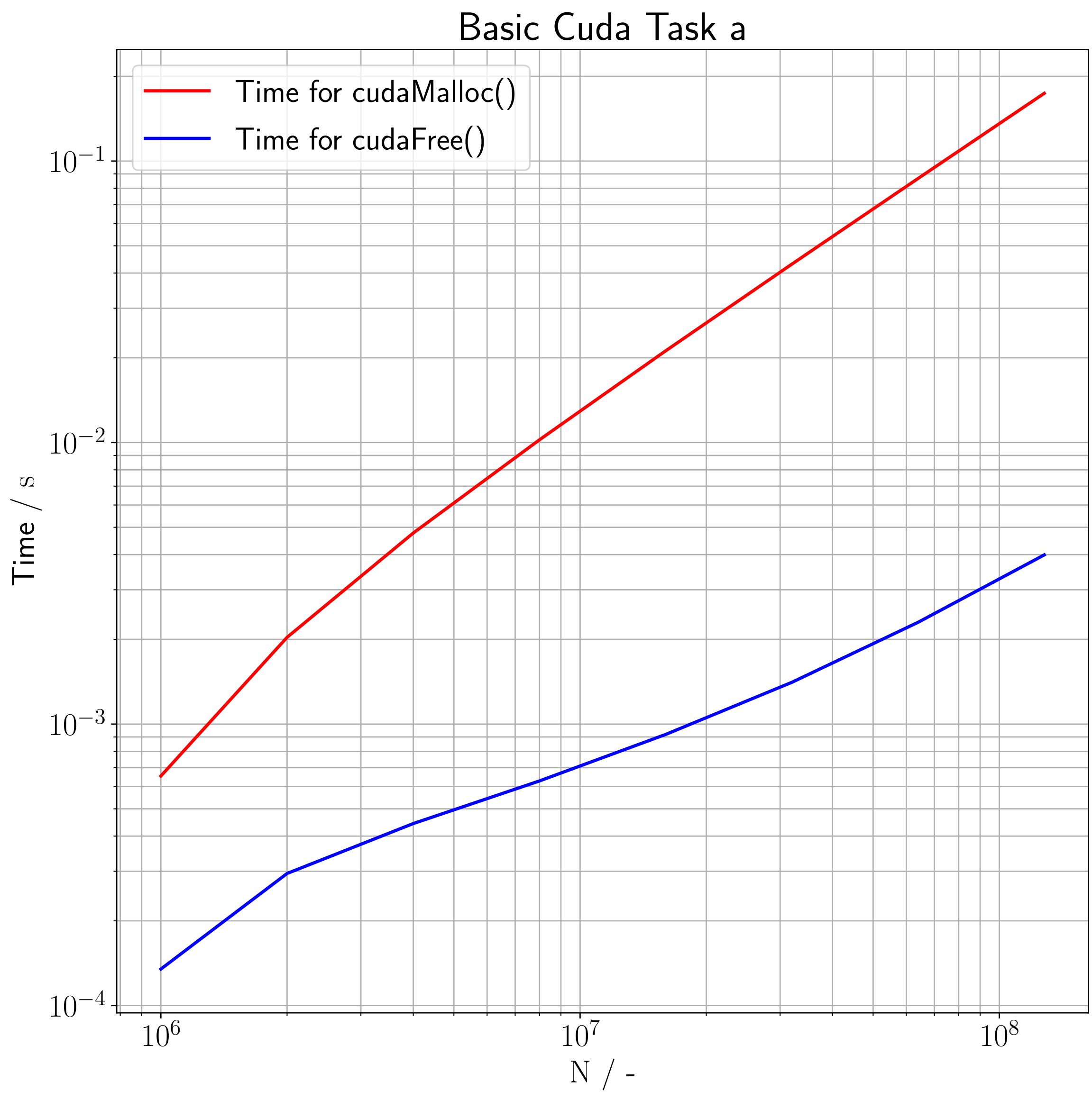


Figure 1: Plot for task 1a - see code in appendix A

## 1.2 Task b

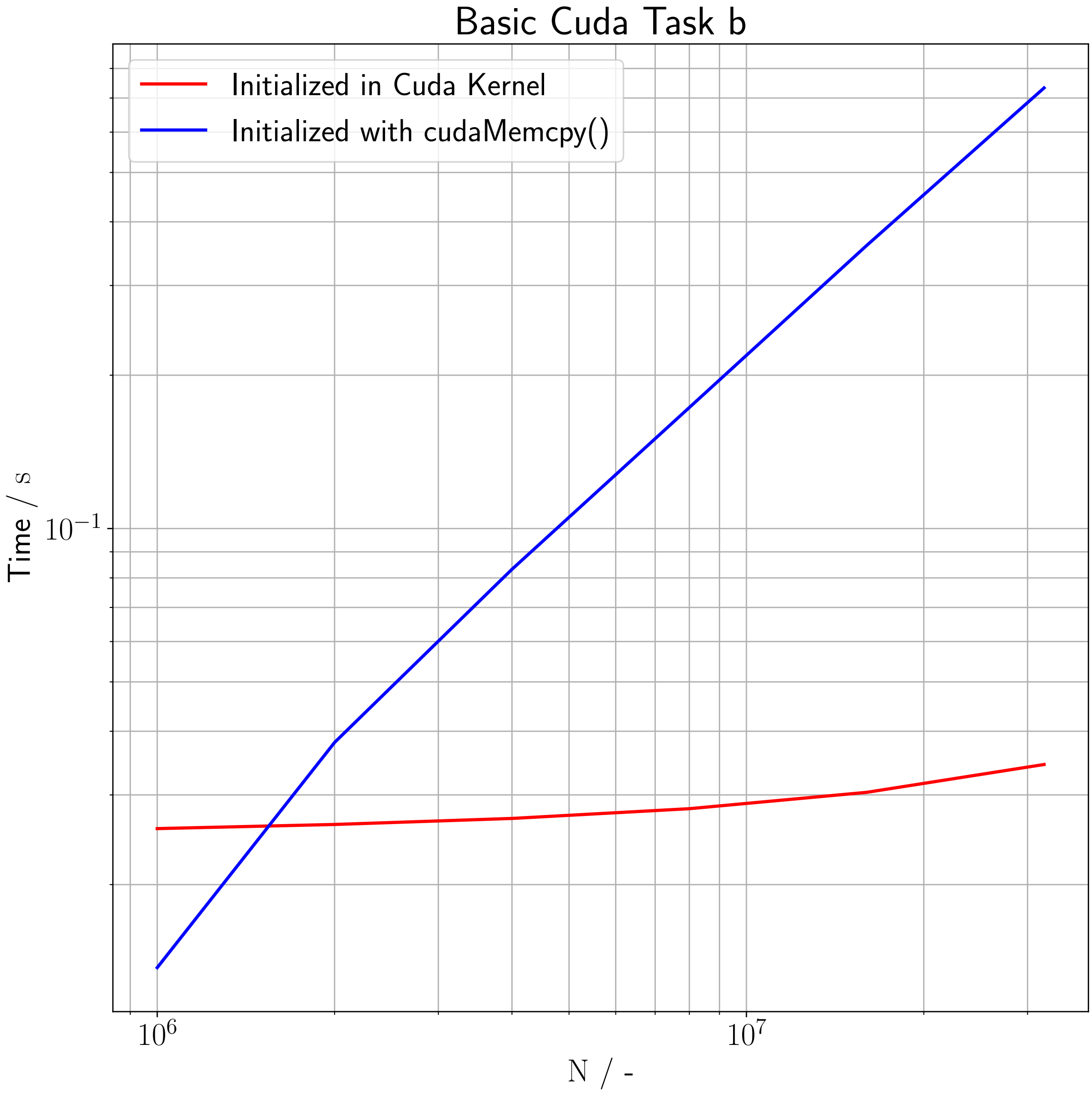


Figure 2: Plot for task 1b - see code in appendix B

### 1.3 Task c

See Implementation in appendix C.

### 1.4 Task d

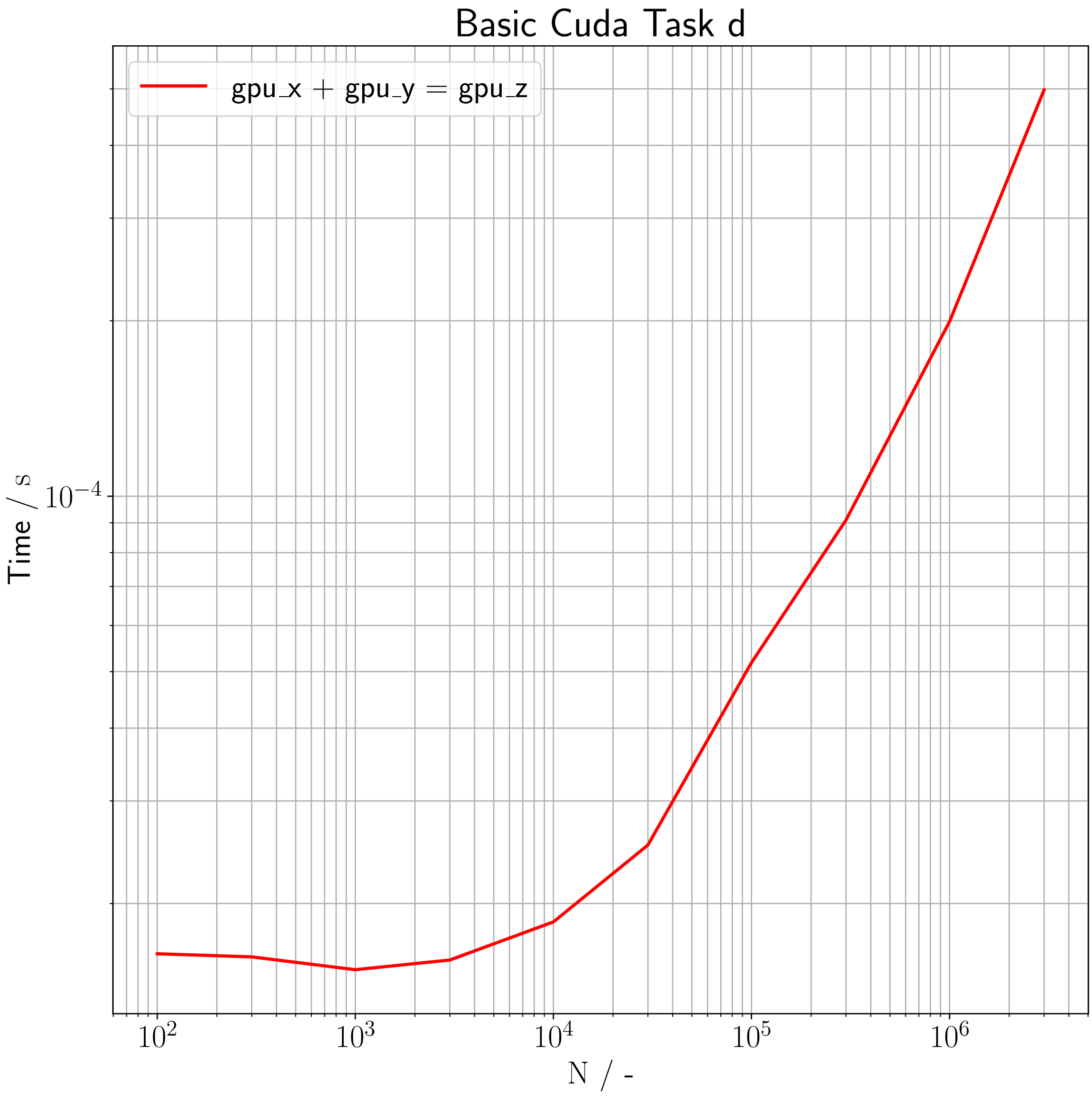


Figure 3: Plot for task 1d - See code in appendix C

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

## 1.5 Task e

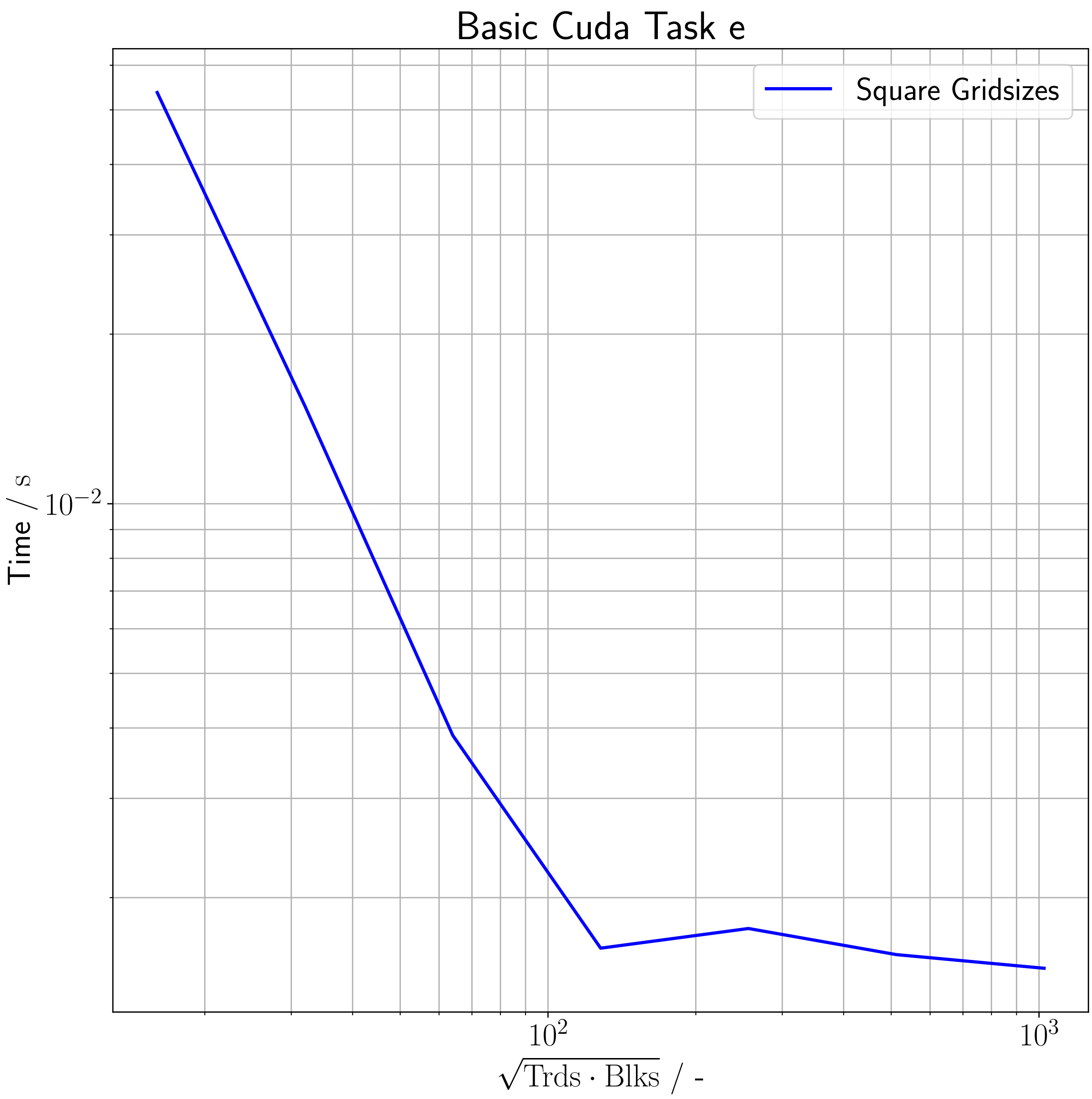


Figure 4: Plot for Task 1e - See code in appendix C

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

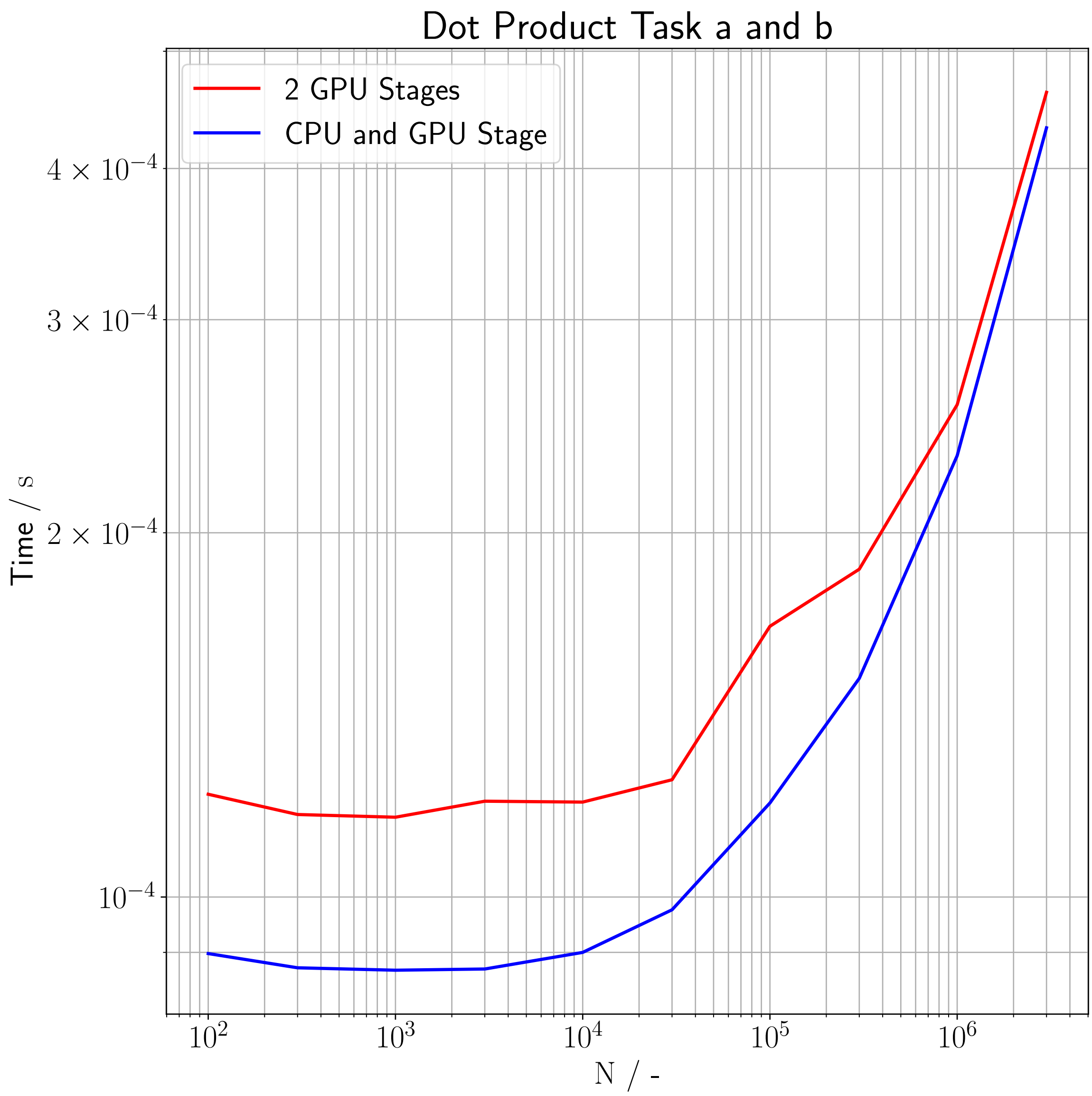


Figure 5: Plots for the Dot Product Task - See code in appendix D and E

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

# A CPP CUDA Code - Basic Cuda - Task 1a

C++ Listing

```
#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

C++ Listing

```
#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++) {
```

```
      timer.reset();
      cudaDeviceSynchronize();
      x = (double*)malloc(N*sizeof(double));
      y = (double*)malloc(N*sizeof(double));
      for (int i = 0; i < N; i++) {
        x[i] = (double)(i);
        y[i] = (double)(N-i-1);
      }
      cudaMalloc(&gpu_x, N*sizeof(double));
      cudaMalloc(&gpu_y, N*sizeof(double));
      cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
      cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
      cudaDeviceSynchronize();
      option_2 += timer.get();
    }
    printf("%d,%g\n", N, 0.2*option_2);
    cudaMemcpy(x, gpu_x, N*sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(y, gpu_y, N*sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(gpu_x);
    cudaFree(gpu_y);
    free(x);
    free(y);
    k++;
  }
  return EXIT_SUCCESS;
}
```

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

C++ Listing

```cpp
#include <stdio.h>
#include "timer.hpp"
__global__ void addVec(double *x, double *y, double *z, 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) {
    z[i] = x[i] + y[i];
  }
}
int main(void)
{
  // Task c //
  double *x, *y, *z, *gpu_x, *gpu_y, *gpu_z;
  Timer timer;
  int N = 100;
  x = (double*)malloc(N*sizeof(double));
  y = (double*)malloc(N*sizeof(double));
  z = (double*)malloc(N*sizeof(double));
  for (int i = 0; i < N; i++) {
    x[i] = (double)(i);
    y[i] = (double)(N-i-1);
  }
  cudaMalloc(&gpu_x, N*sizeof(double));
  cudaMalloc(&gpu_y, N*sizeof(double));
  cudaMalloc(&gpu_z, N*sizeof(double));
  cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
  addVec<<<256, 256>>>(gpu_x, gpu_y, gpu_z, N);
  cudaMemcpy(z, gpu_z, N*sizeof(double), cudaMemcpyDeviceToHost);
  cudaFree(gpu_x);
  cudaFree(gpu_y);
  cudaFree(gpu_z);
  free(x);
  free(y);
  free(z);

  // Task d //
  int k = 0;
  int N_values[10] = { 100, 300, 1000, 3000, 10000, 30000, 100000, 300000, 1000000, 3000000 };
  printf("\nsize,time\n");
  while(k < 10) {
    float t_kernel =0;
    int N = N_values[k];
    x = (double*)malloc(N*sizeof(double));
    y = (double*)malloc(N*sizeof(double));
    z = (double*)malloc(N*sizeof(double));
    for (int i = 0; i < N; i++) {
```

```
        x[i] = (double)(i);
        y[i] = (double)(N-i-1);
      }
      cudaMalloc(&gpu_x, N*sizeof(double));
      cudaMalloc(&gpu_y, N*sizeof(double));
      cudaMalloc(&gpu_z, N*sizeof(double));
      cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
      cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
      timer.reset();
      for (int n=0; n<5; n++) {
        addVec<<<256, 256>>>(gpu_x, gpu_y, gpu_z, N);
        cudaDeviceSynchronize();
      }
      t_kernel += timer.get();
      printf("%d,%g\n", N, 0.2*t_kernel);
      cudaMemcpy(z, gpu_z, N*sizeof(double), cudaMemcpyDeviceToHost);
      cudaFree(gpu_x);
      cudaFree(gpu_y);
      cudaFree(gpu_z);
      free(x);
      free(y);
      free(z);
      k++;
    }
    // Task e //
    N = 10000000;
    k = 0;
    int params[7] = { 16, 32, 64, 128, 256, 512, 1024};
    printf("\nsqrt(threads),time\n");
    while(k < 7) {
      float t_kernel =0;
      int param = params[k];
      x = (double*)malloc(N*sizeof(double));
      y = (double*)malloc(N*sizeof(double));
      z = (double*)malloc(N*sizeof(double));
      for (int i = 0; i < N; i++) {
        x[i] = (double)(i);
        y[i] = (double)(N-i-1);
      }
      cudaMalloc(&gpu_x, N*sizeof(double));
      cudaMalloc(&gpu_y, N*sizeof(double));
      cudaMalloc(&gpu_z, N*sizeof(double));
      cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
      cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
      timer.reset();
      for (int n=0; n<5; n++) {
        addVec<<<param, param>>>(gpu_x, gpu_y, gpu_z, N);
        cudaDeviceSynchronize();
      }
      t_kernel += timer.get();
      printf("%d,%g\n", param, 0.2*t_kernel);
```

```
    cudaMemcpy(z, gpu_z, N*sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(gpu_x);
    cudaFree(gpu_y);
    cudaFree(gpu_z);
    free(x);
    free(y);
    free(z);
    k++;
  }
  return EXIT_SUCCESS;
}
```

# D CPP CUDA Code - Dot Product - Task 2a

C++ Listing

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

const int threads_per_block = 256;
double dot_cpu(double *a, double *b, int N) {
    double product = 0;
    for (int i = 0; i < N; i++)
    product = product + a[i] * b[i];
    return product;
}
__global__ void dotVec_one(double *x, double *y, double * partial_z , int N) {
  __shared__ double temp_arr[threads_per_block];
  double thread_product = 0;
  unsigned int global_tid = threadIdx.x + blockIdx.x * blockDim.x;
  unsigned int local_tid  = threadIdx.x;
  unsigned int total_threads = blockDim.x * gridDim.x;
  for (unsigned int i=global_tid; i<N; i+=total_threads) {
    thread_product += x[i] * y[i];
  }
  temp_arr[ local_tid ] = thread_product;
  for (unsigned int stride = blockDim.x/2; stride>0; stride/=2) {
    __syncthreads();
    if (threadIdx.x < stride) {
      temp_arr[threadIdx.x] += temp_arr[threadIdx.x + stride];
    }
  }
  if (threadIdx.x == 0) {
     partial_z [blockIdx.x] = temp_arr[0];
  }
}
__global__ void dotVec_two(double *partial_z) {
  for (int stride = blockDim.x/2; stride>0; stride/=2) {
    __syncthreads();
    if (threadIdx.x < stride)
      partial_z [threadIdx.x] += partial_z[threadIdx.x+stride];
  }
}
int main(void)
{
  // Task a //
  double *x, *y, *z;
  double *gpu_x, *gpu_y, * gpu_partial_z ;
  Timer timer;
  int k = 0;
  int N_values_d[10] = { 100, 300, 1000, 3000, 10000, 30000, 100000, 300000, 1000000, 3000000 };
  printf("\nsize,time\n");
  while(k < 10) {
```

```
    int N = N_values_d[k];
    x = (double*)malloc(N*sizeof(double));
    y = (double*)malloc(N*sizeof(double));
    z = (double*)malloc(threads_per_block*sizeof(double));
    for (int i = 0; i < N; i++) {
      x[i] = 1.0;
      y[i] = 1.0;
    }
    cudaMalloc(&gpu_x, N*sizeof(double));
    cudaMalloc(&gpu_y, N*sizeof(double));
    cudaMalloc(&gpu_partial_z, threads_per_block*sizeof(double));
    cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
    timer.reset();
    for (int n=0; n<5; n++) {
      dotVec_one<<<256, threads_per_block>>>(gpu_x, gpu_y, gpu_partial_z, N);
      cudaDeviceSynchronize();
      dotVec_two<<<1, threads_per_block>>>(gpu_partial_z);
      cudaMemcpy(z, gpu_partial_z, threads_per_block*sizeof(double), cudaMemcpyDeviceToHost);
    }
    printf("%g,%g\n", z[0], 0.2*timer.get());
    cudaFree(gpu_x);
    cudaFree(gpu_y);
    cudaFree(gpu_partial_z);
    free(x);
    free(y);
    free(z);
    k++;
  }
  return EXIT_SUCCESS;
}
```

# E CPP CUDA Code - Dot Product - Task 2b

C++ Listing

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

const int threads_per_block = 256;
double dotVec_cpu(double *a, double *b, int N) {
   double product = 0;
   for (int i = 0; i < N; i++)
   product = product + a[i] * b[i];
   return product;
}
 __global__ void dotVec_gpu(double *x, double *y, double * partial_z , int N) {
  __shared__ double temp_arr[threads_per_block];
  double thread_product = 0;
  unsigned int global_tid = threadIdx.x + blockIdx.x * blockDim.x;
  unsigned int local_tid = threadIdx.x;
  unsigned int total_threads = blockDim.x * gridDim.x;
  for (unsigned int i=global_tid; i<N; i+=total_threads) {
    thread_product += x[i] * y[i];
  }
  temp_arr[ local_tid ] = thread_product;
  for (unsigned int stride = blockDim.x/2; stride>0; stride/=2) {
    __syncthreads();
    if (threadIdx.x < stride) {
      temp_arr[threadIdx.x] += temp_arr[threadIdx.x + stride];
    }
  }
  if (threadIdx.x == 0) {
     partial_z [blockIdx.x] = temp_arr[0];
  }
}

int main(void)
{
  // Task b //
  double *x, *y, z, * partial_z ;
  double *gpu_x, *gpu_y, * gpu_partial_z ;
  Timer timer;
  int k = 0;
  int N_values_d[10] = {100, 300, 1000, 3000, 10000, 30000, 100000, 300000, 1000000, 3000000};
  printf("\nsize,time\n");
  while(k < 10) {
    int N = N_values_d[k];
    x = (double*)malloc(N*sizeof(double));
    y = (double*)malloc(N*sizeof(double));
     partial_z = (double*)malloc(256*sizeof(double));
    for (int i = 0; i < N; i++) {
      x[i] = 1.0;
```

```
      y[i] = 1.0;
    }
    cudaMalloc(&gpu_x, N*sizeof(double));
    cudaMalloc(&gpu_y, N*sizeof(double));
    cudaMalloc(&gpu_partial_z, 256*sizeof(double));
    cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
    timer.reset();
    for (int n=0; n<5; n++) {
      dotVec_gpu<<<256, 256>>>(gpu_x, gpu_y, gpu_partial_z, N);
      cudaMemcpy(partial_z, gpu_partial_z, 256*sizeof(double), cudaMemcpyDeviceToHost);
      z = 0;
      for(int i=0; i<256; i++) {
        z += partial_z[i];
      }
    }
    printf("%d,%g\n", N, 0.2*timer.get());
    cudaFree(gpu_x);
    cudaFree(gpu_y);
    cudaFree(gpu_partial_z);
    free(x);
    free(y);
    free(partial_z);
    k++;
  }
  return EXIT_SUCCESS;
}
```