# Computational Science on Many-Core Architectures: Exercise 4:

Viktor Beck, 11713110

November 2022

## Task 1: Dot Product with Warp Shuffles

### a)

Here, for each of the results a shared memory is created. The results are computed from the vectors and stored / summed into the shared memory. In the end `atomicAdd()` is used to write the results into the result vector which is then passed to the host with a single `cudaMemcpy`.

### b)

Instead of shared memory warp shuffles are used. At first, each thread gets its own and the corresponding entries of x in the other blocks. After that block 0 holds all the important information. Next, the sums within the warps are summed with the warp shuffle method (like in the lecture). Finally, the relevant threads use `atomicAdd()` to sum the sums of the warps.

### c)

Like b) but here the kernel is initialized with $(N + 255)/256$ blocks instead of a fixed number of 256 blocks.

### d)

In the figures below we can see that for lower numbers of N all the approaches are constant and almost equally good (speaking of time). c) gets (surprisingly) worse for higher numbers of N, maybe this is because too many thread blocks are created which is not efficient anymore for higher N. In Figure 2 one can also see that a) and the dot product are quite similar because they are based on the same approach. Also, they are a little bit better than the CUBLAS implementation of my (untalented) friend.

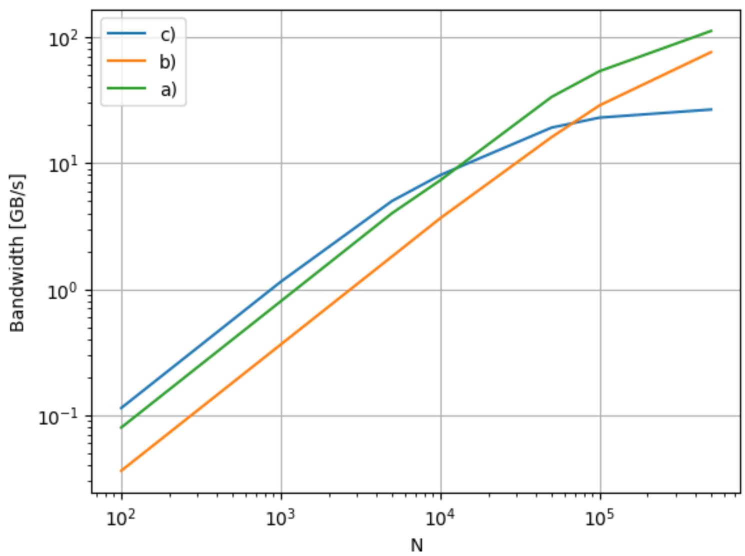


Figure 1: Bandwidth for different N for a), b), c)

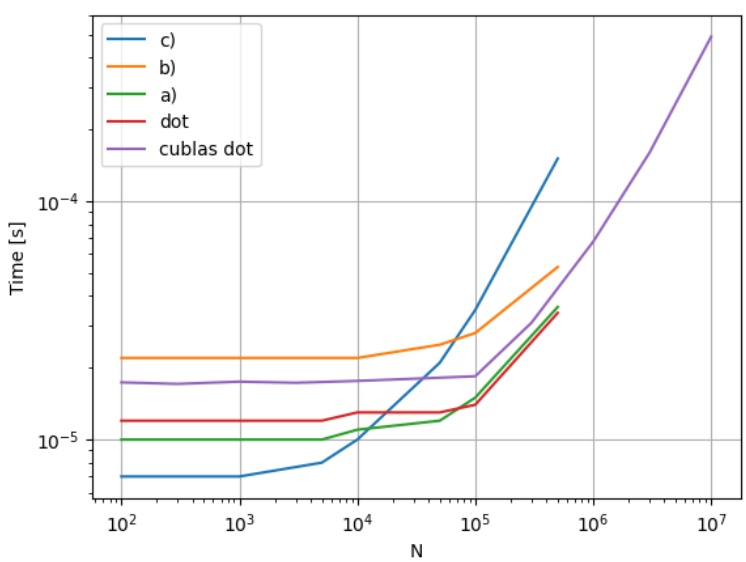


Figure 2: Times for different N for a), b), c)

# Task 2: Multiple Dot Products

## a)

To calculate 8 dot products at the same time the code from the provided "dot-product.cu" was modified in the simplest way possible by introducing 7 more shared memories. See code "Task

2.abc)" in appendix. On the other side, it was quite tricky to get it running since the allocations with double pointers were really tricky.

## b)

Everything here was quite straightforward. See code.

## c)

Here we can see that for different K's the execution times are linearly rising. It seems that having multiple batches is not that efficient which can also be seen in the plot for the CUBLAS implementation you provided. Own implementations were plotted on the GTX GPU and not on the K40 since `atomicAdd()` does not work. I did not had the time to adapt the kernel such that it runs on the K40 GPU and it was therefore not possible for me to compare my implementations to CUBLAS on the same machine. But at least my own implementation on the GTX is faster than CUBLAS on the K40 :-).
If we compare the relative differences then CUBLAS is way better. In CUBLAS it is about $10^{-15}$ while for the own implementation it is about $10^{-3}$.

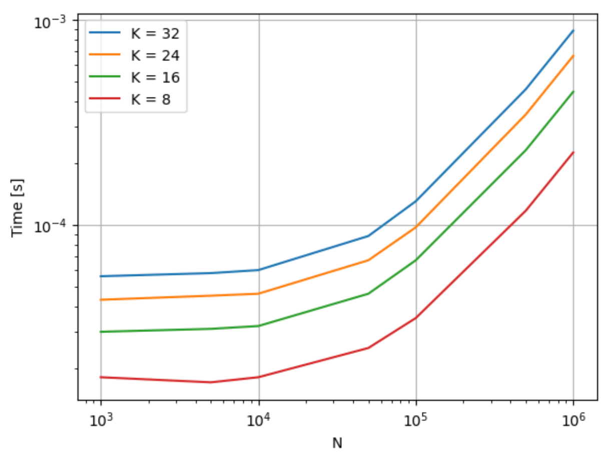


Figure 3: Times for different N logarithmically scaled on GTX GPU

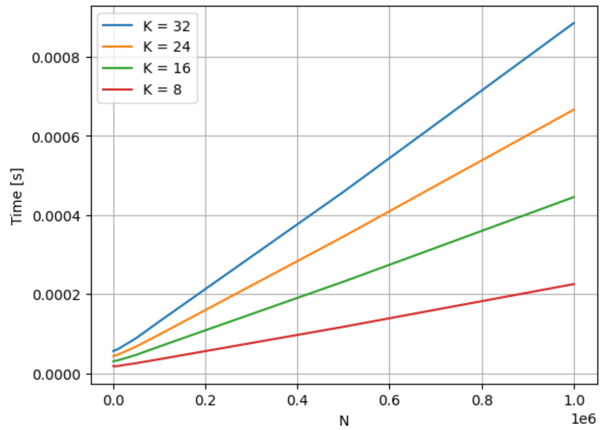


Figure 4: Times for different N linearly scaled on GTX GPU

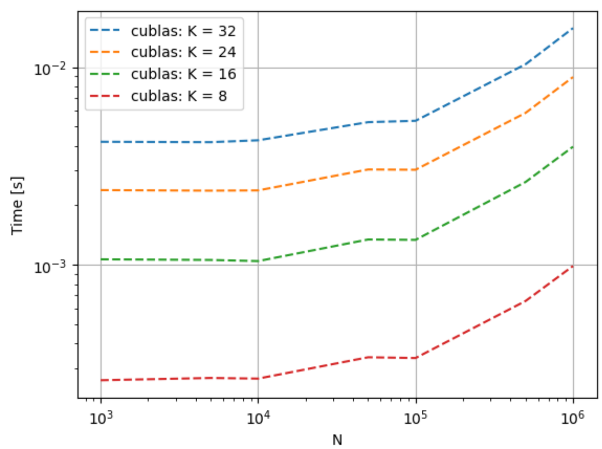


Figure 5: Times for different N with the CUBLAS implementation on K40 GPU

## d)

The easiest approach would be to write a new kernel such that every dot product is computed individually. Also, one could then write an if statement such that every case with a K which is dividable by 8 gets calculated with the implementation of the kernel from a).

# Appendix

## Task 1.a)

```
#include "timer.hpp"
#include "cuda_errchk.hpp"
#include <algorithm>
#include <iostream>
#include <stdio.h>
#include <vector>

// result = (x, y)
__global__ void cuda_sums(int N, double *x, double *res)
{
    // clean res to wipe results from previous repetition
    if (blockIdx.x * blockDim.x + threadIdx.x == 0){
        res[0] = 0;
        res[1] = 0;
        res[2] = 0;
        res[3] = 0;
    }
    // shared mem for each
    __shared__ double shared_sum[256];
    __shared__ double shared_abs_sum[256];
    __shared__ double shared_sq_sum[256];
    __shared__ double shared_zero;

    double entry = 0;
    double entry_abs = 0;
    double entry_sq = 0;
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
    {
        entry += x[i];
        entry_abs += abs(x[i]);
        entry_sq += x[i]*x[i];
        if (x[i] == 0){
            shared_zero++;
            //printf("zero_entries=%g\n", shared_zero);
            atomicAdd(&res[3], shared_zero);
        }
    }
    // write x to every shared memory
    shared_sum[threadIdx.x] = entry;
    shared_abs_sum[threadIdx.x] = entry_abs;
    shared_sq_sum[threadIdx.x] = entry_sq;

    for (int k = blockDim.x / 2; k > 0; k /= 2)
    {
        __syncthreads();
        if (threadIdx.x < k)
        {
            // sum
            shared_sum[threadIdx.x] += shared_sum[threadIdx.x + k];
            // abs sum
            shared_abs_sum[threadIdx.x] += shared_abs_sum[threadIdx.x + k];
            // square sum
            shared_sq_sum[threadIdx.x] += shared_sq_sum[threadIdx.x + k];
        }
    }

    if (threadIdx.x == 0)
```

```
    {
        atomicAdd(&res[0], shared_sum[0]);
        atomicAdd(&res[1], shared_abs_sum[0]);
        atomicAdd(&res[2], shared_sq_sum[0]);
    }
}


int main()
{
    std::vector<int> N_vec = {100,500,1000,5000,10000,50000,100000,500000};
    std::vector<double> bandwidth;
    std::vector<double> times;
    Timer timer;
    int reps = 6;

    for (const auto &N : N_vec){
        std::vector<double> time_vec;

        // Allocate and initialize arrays on CPU

        double *x = (double *)malloc(sizeof(double) * N);
        // double *y = (double *)malloc(sizeof(double) * N);
        int res_size = 4; // size of result array
        double *result = (double *)malloc(sizeof(double) * res_size);

        std::fill(x, x + N, -2);
        std::fill(result, result + res_size, 0);
        // add a zero to x
        x[1] = 0;
        // std::fill(y, y + N, 2);

        // Allocate and initialize arrays on GPU

        double *cuda_x;
        //double *cuda_y;
        double *cuda_result;

        CUDA_ERRCHK(cudaMalloc(&cuda_x, sizeof(double) * N));
        //CUDA_ERRCHK(cudaMalloc(&cuda_y, sizeof(double) * N));
        CUDA_ERRCHK(cudaMalloc(&cuda_result, sizeof(double) * res_size));

        CUDA_ERRCHK(cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice));
        //CUDA_ERRCHK(cudaMemcpy(cuda_y, y, sizeof(double) * N, cudaMemcpyHostToDevice));
        CUDA_ERRCHK(cudaMemcpy(cuda_result, &result, sizeof(double) * res_size, cudaMemcpyHostToDevic

        // repetitions
        for (int j = 0; j < reps; j++){
            // wait for previous operations to finish, then start timings
            CUDA_ERRCHK(cudaDeviceSynchronize());
            timer.reset();

            cuda_sums<<<256, 256>>>(N, cuda_x, cuda_result);

            CUDA_ERRCHK(cudaDeviceSynchronize());
            time_vec.push_back(timer.get());
        }
        std::sort(time_vec.begin(), time_vec.end());
        times.push_back(time_vec[reps/2]);
        bandwidth.push_back(N*sizeof(double)/time_vec[reps/2]*1e-9);

        CUDA_ERRCHK(cudaMemcpy(result, cuda_result, sizeof(double)*res_size, cudaMemcpyDeviceToHost))
```

```

        std::cout << "Result sum: " << result[0] << std::endl;
        std::cout << "Result abs sum: " << result[1] << std::endl;
        std::cout << "Result square sum: " << result[2] << std::endl;
        std::cout << "Result zero entries: " << result[3] << "\n" << std::endl;

        // Clean up

        CUDA_ERRCHK(cudaFree(cuda_x));
        //CUDA_ERRCHK(cudaFree(cuda_y));
        CUDA_ERRCHK(cudaFree(cuda_result));
        free(x);
        free(result);
        //free(y);
    }

    std::cout << "Gb/s:\n" << std::endl;
    for (const auto& value : bandwidth){
        std::cout << value << "," << std::endl;
    }

    std::cout << "\ns:\n" << std::endl;
    for (const auto& value : times){
        std::cout << value << "," << std::endl;
    }

    CUDA_ERRCHK(cudaDeviceReset()); // for CUDA leak checker to work

    return EXIT_SUCCESS;
}
```

## Task 1.b)

```
#include "timer.hpp"
#include "cuda_errchk.hpp"
#include <algorithm>
#include <iostream>
#include <stdio.h>
#include <vector>

__global__ void my_warp_reduction(int N, double *x, double *res) {

    if (blockIdx.x * blockDim.x + threadIdx.x == 0){
        res[0] = 0;
        res[1] = 0;
        res[2] = 0;
        res[3] = 0;
    }

    double sum = 0;
    double abs_sum = 0;
    double sq_sum = 0;
    double zeros = 0;

    int id = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = id; i < N; i += blockDim.x * gridDim.x) {
        sum += x[i];
        abs_sum += abs(x[i]);
        sq_sum += x[i]*x[i];
        if (x[i]==0) zeros++;
```

```
    }
    for (int i=16; i>0; i=i/2){
        sum += __shfl_down_sync(-1, sum, i);
        abs_sum += __shfl_down_sync(-1, abs_sum, i);
        sq_sum += __shfl_down_sync(-1, sq_sum, i);
        zeros += __shfl_down_sync(-1, zeros, i);
    } // thread 0 contains sum of all values

    if ((threadIdx.x & 31) == 0){
        atomicAdd(&res[0], sum);
        atomicAdd(&res[1], abs_sum);
        atomicAdd(&res[2], sq_sum);
        atomicAdd(&res[3], zeros);
    }
}


int main()
{
    std::vector<int> N_vec = {100,500,1000,5000,10000,50000,100000,500000};
    std::vector<double> bandwidth;
    std::vector<double> times;
    Timer timer;
    int reps = 6;

    for (const auto &N : N_vec){
        std::vector<double> time_vec;

        // Allocate and initialize arrays on CPU

        double *x = (double *)malloc(sizeof(double) * N);
        // double *y = (double *)malloc(sizeof(double) * N);
        int res_size = 4; // size of result array
        double *result = (double *)malloc(sizeof(double) * res_size);

        std::fill(x, x + N, -2);
        std::fill(result, result + res_size, 0);
        // add a zero to x
        x[1] = 0;
        // std::fill(y, y + N, 2);

        // Allocate and initialize arrays on GPU

        double *cuda_x;
        //double *cuda_y;
        double *cuda_result;

        CUDA_ERRCHK(cudaMalloc(&cuda_x, sizeof(double) * N));
        //CUDA_ERRCHK(cudaMalloc(&cuda_y, sizeof(double) * N));
        CUDA_ERRCHK(cudaMalloc(&cuda_result, sizeof(double) * res_size));

        CUDA_ERRCHK(cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice));
        //CUDA_ERRCHK(cudaMemcpy(cuda_y, y, sizeof(double) * N, cudaMemcpyHostToDevice));
        CUDA_ERRCHK(cudaMemcpy(cuda_result, &result, sizeof(double) * res_size, cudaMemcpyHostToDevic

        // repetitions
        for (int j = 0; j < reps; j++){
            // wait for previous operations to finish, then start timings
            CUDA_ERRCHK(cudaDeviceSynchronize());
            timer.reset();

            my_warp_reduction<<<256, 256>>>(N, cuda_x, cuda_result);
```

```
            //cuda_sums<<<256, 256>>>(N, cuda_x, cuda_result);

            CUDA_ERRCHK(cudaDeviceSynchronize());
            time_vec.push_back(timer.get());
        }
        std::sort(time_vec.begin(), time_vec.end());
        times.push_back(time_vec[reps/2]);
        bandwidth.push_back(N*sizeof(double)/time_vec[reps/2]*1e-9);

        CUDA_ERRCHK(cudaMemcpy(result, cuda_result, sizeof(double)*res_size, cudaMemcpyDeviceToHost))

        std::cout << "Result sum: " << result[0] << std::endl;
        std::cout << "Result abs sum: " << result[1] << std::endl;
        std::cout << "Result square sum: " << result[2] << std::endl;
        std::cout << "Result zero entries: " << result[3] << "\n" << std::endl;

        // Clean up

        CUDA_ERRCHK(cudaFree(cuda_x));
        //CUDA_ERRCHK(cudaFree(cuda_y));
        CUDA_ERRCHK(cudaFree(cuda_result));
        free(x);
        free(result);
        //free(y);
    }

    std::cout << "Gb/s:\n" << std::endl;
    for (const auto& value : bandwidth){
        std::cout << value << "," << std::endl;
    }

    std::cout << "\ns:\n" << std::endl;
    for (const auto& value : times){
        std::cout << value << "," << std::endl;
    }

    CUDA_ERRCHK(cudaDeviceReset()); // for CUDA leak checker to work

    return EXIT_SUCCESS;
}
```

### Task 1.c)

```
#include "timer.hpp"
#include "cuda_errchk.hpp"
#include <algorithm>
#include <iostream>
#include <stdio.h>
#include <vector>

__global__ void my_warp_reduction(int N, double *x, double *res) {

    if (blockIdx.x * blockDim.x + threadIdx.x == 0){
        res[0] = 0;
        res[1] = 0;
        res[2] = 0;
        res[3] = 0;
    }

    double sum = 0;
```

```
    double abs_sum = 0;
    double sq_sum = 0;
    double zeros = 0;


    int id = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = id; i < N; i += blockDim.x * gridDim.x) {
        sum += x[i];
        abs_sum += abs(x[i]);
        sq_sum += x[i]*x[i];
        if (x[i]==0) zeros++;
    }
    for (int i=16; i>0; i=i/2){
        sum += __shfl_down_sync(-1, sum, i);
        abs_sum += __shfl_down_sync(-1, abs_sum, i);
        sq_sum += __shfl_down_sync(-1, sq_sum, i);
        zeros += __shfl_down_sync(-1, zeros, i);
    } // thread 0 contains sum of all values

    if ((threadIdx.x & 31) == 0){
        atomicAdd(&res[0], sum);
        atomicAdd(&res[1], abs_sum);
        atomicAdd(&res[2], sq_sum);
        atomicAdd(&res[3], zeros);
    }
}


int main()
{
    std::vector<int> N_vec = {100,500,1000,5000,10000,50000,100000,500000};
    std::vector<double> bandwidth;
    std::vector<double> times;
    Timer timer;
    int reps = 6;

    for (const auto &N : N_vec){
        std::vector<double> time_vec;

        // Allocate and initialize arrays on CPU

        double *x = (double *)malloc(sizeof(double) * N);
        // double *y = (double *)malloc(sizeof(double) * N);
        int res_size = 4; // size of result array
        double *result = (double *)malloc(sizeof(double) * res_size);

        std::fill(x, x + N, -2);
        std::fill(result, result + res_size, 0);
        // add a zero to x
        x[1] = 0;
        // std::fill(y, y + N, 2);

        // Allocate and initialize arrays on GPU

        double *cuda_x;
        //double *cuda_y;
        double *cuda_result;

        CUDA_ERRCHK(cudaMalloc(&cuda_x, sizeof(double) * N));
        //CUDA_ERRCHK(cudaMalloc(&cuda_y, sizeof(double) * N));
        CUDA_ERRCHK(cudaMalloc(&cuda_result, sizeof(double) * res_size));

```

```
        CUDA_ERRCHK(cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice));
        //CUDA_ERRCHK(cudaMemcpy(cuda_y, y, sizeof(double) * N, cudaMemcpyHostToDevice));
        CUDA_ERRCHK(cudaMemcpy(cuda_result, &result, sizeof(double) * res_size, cudaMemcpyHostToDevic

        // repetitions
        for (int j = 0; j < reps; j++){
            // wait for previous operations to finish, then start timings
            CUDA_ERRCHK(cudaDeviceSynchronize());
            timer.reset();

            my_warp_reduction<<<(N+255)/256, 256>>>(N, cuda_x, cuda_result);

            CUDA_ERRCHK(cudaDeviceSynchronize());
            time_vec.push_back(timer.get());
        }
        std::sort(time_vec.begin(), time_vec.end());
        times.push_back(time_vec[reps/2]);
        bandwidth.push_back(N*sizeof(double)/time_vec[reps/2]*1e-9);

        CUDA_ERRCHK(cudaMemcpy(result, cuda_result, sizeof(double)*res_size, cudaMemcpyDeviceToHost))

        std::cout << "Result sum: " << result[0] << std::endl;
        std::cout << "Result abs sum: " << result[1] << std::endl;
        std::cout << "Result square sum: " << result[2] << std::endl;
        std::cout << "Result zero entries: " << result[3] << "\n" << std::endl;

        // Clean up

        CUDA_ERRCHK(cudaFree(cuda_x));
        //CUDA_ERRCHK(cudaFree(cuda_y));
        CUDA_ERRCHK(cudaFree(cuda_result));
        free(x);
        free(result);
        //free(y);
    }

    std::cout << "Gb/s:\n" << std::endl;
    for (const auto& value : bandwidth){
        std::cout << value << "," << std::endl;
    }

    std::cout << "\ns:\n" << std::endl;
    for (const auto& value : times){
        std::cout << value << "," << std::endl;
    }



    CUDA_ERRCHK(cudaDeviceReset()); // for CUDA leak checker to work

    return EXIT_SUCCESS;
}
```

### Task 2.abc)

```
#include "timer.hpp"
#include "cuda_errchk.hpp"
#include <algorithm>
#include <cuda_runtime.h>
#include <cublas_v2.h>
```

```
#include <stdio.h>
#include <cmath>
#include <iostream>
#include <vector>

__global__ void cuda_mdot_product(int N, double *x, double *y, double *res)
{
  // clean res to wipe results from previous repetition
  if (blockIdx.x * blockDim.x + threadIdx.x == 0){
    res[0] = 0;
    res[1] = 0;
    res[2] = 0;
    res[3] = 0;
    res[4] = 0;
    res[5] = 0;
    res[6] = 0;
    res[7] = 0;
  }

  __shared__ double shared_mem1[256]; // remember to only use 256 threads per block then!
  __shared__ double shared_mem2[256];
  __shared__ double shared_mem3[256];
  __shared__ double shared_mem4[256];
  __shared__ double shared_mem5[256];
  __shared__ double shared_mem6[256];
  __shared__ double shared_mem7[256];
  __shared__ double shared_mem8[256];

  double dot1 = 0, dot2 = 0, dot3 = 0, dot4 = 0, dot5 = 0, dot6 = 0, dot7 = 0, dot8 = 0;
  double val_w = 0;

  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
  {
    // printf("y = %g\n", y[i]);
    val_w = x[i];
    dot1 += val_w * y[i];
    dot2 += val_w * y[i + 1 * N];
    dot3 += val_w * y[i + 2 * N];
    dot4 += val_w * y[i + 3 * N];
    dot5 += val_w * y[i + 4 * N];
    dot6 += val_w * y[i + 5 * N];
    dot7 += val_w * y[i + 6 * N];
    dot8 += val_w * y[i + 7 * N];
  }

  shared_mem1[threadIdx.x] = dot1;
  shared_mem2[threadIdx.x] = dot2;
  shared_mem3[threadIdx.x] = dot3;
  shared_mem4[threadIdx.x] = dot4;
  shared_mem5[threadIdx.x] = dot5;
  shared_mem6[threadIdx.x] = dot6;
  shared_mem7[threadIdx.x] = dot7;
  shared_mem8[threadIdx.x] = dot8;

  for (int k = blockDim.x / 2; k > 0; k /= 2)
  {
    __syncthreads();
    if (threadIdx.x < k)
    {
      shared_mem1[threadIdx.x] += shared_mem1[threadIdx.x + k];
      shared_mem2[threadIdx.x] += shared_mem2[threadIdx.x + k];
      shared_mem3[threadIdx.x] += shared_mem3[threadIdx.x + k];
```

```
      shared_mem4[threadIdx.x] += shared_mem4[threadIdx.x + k];
      shared_mem5[threadIdx.x] += shared_mem5[threadIdx.x + k];
      shared_mem6[threadIdx.x] += shared_mem6[threadIdx.x + k];
      shared_mem7[threadIdx.x] += shared_mem7[threadIdx.x + k];
      shared_mem8[threadIdx.x] += shared_mem8[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0)
  {
    atomicAdd(&res[0], shared_mem1[0]);
    atomicAdd(&res[1], shared_mem2[0]);
    atomicAdd(&res[2], shared_mem3[0]);
    atomicAdd(&res[3], shared_mem4[0]);
    atomicAdd(&res[4], shared_mem5[0]);
    atomicAdd(&res[5], shared_mem6[0]);
    atomicAdd(&res[6], shared_mem7[0]);
    atomicAdd(&res[7], shared_mem8[0]);
  }
}

int main(void)
{
  std::vector<size_t> N_vec = {1000,5000,10000,50000,100000,500000,1000000};
  std::vector<double> times;
  Timer timer;
  int reps = 6;

  for (const auto &N : N_vec){
    std::vector<double> time_vec;

    //const size_t N = 4000;
    const size_t K = 32;

    //
    // allocate host memory:
    //
    std::cout << "Allocating host arrays..." << std::endl;
    // for x and y
    double *x = (double *)malloc(sizeof(double) * N);
    // use double pointers for vectors (like matrix)
    double **y = (double **)malloc(sizeof(double *) * K / 8);

    // for results on cpu and gpu
    double *results = (double *)malloc(sizeof(double) * K);
    for (size_t i = 0; i < K / 8; ++i)
    {
      y[i] = (double *)malloc(sizeof(double) * N * 8);
    }
    double **results2 = (double **)malloc(sizeof(double) * K / 8);
    for (size_t i = 0; i < K / 8; ++i)
    {
      results2[i] = (double *)malloc(sizeof(double) * 8);
    }
    // for the calculation on the cpu to compare with gpu results
    double **y_cpu = (double **)malloc(sizeof(double *) * K);
    for (size_t i = 0; i < K; ++i)
    {
      y_cpu[i] = (double *)malloc(sizeof(double) * N);
    }

    //
```

```
    // allocate device memory
    //
    std::cout << "Allocating␣CUDA␣arrays..." << std::endl;
    double *cuda_x;
    cudaMalloc((&cuda_x), sizeof(double) * N);
    // pointers to vectors on cpu
    double **cuda_y = (double **)malloc(sizeof(double *) * K / 8);
    for (size_t i = 0; i < K / 8; ++i)
    {
      cudaMalloc((void **)(&cuda_y[i]), sizeof(double) * N * 8);
    }
    double **cuda_results2 = (double **)malloc(sizeof(double *) * K / 8);
    for (size_t i = 0; i < K / 8; ++i)
    {
      cudaMalloc((void **)(&cuda_results2[i]), sizeof(double) * 8);
    }

    //
    // fill host arrays with values
    //
    std::fill(x, x + N, 1.0);
    for (size_t i = 0; i < K / 8; ++i)
    {
      for (size_t j = 0; j < N * 8; ++j)
      {
        y[i][j] = 1 + rand() / (1.1 * RAND_MAX);
      }
    }
    // fill y_cpu
    for (size_t i = 0; i < K; ++i)
    {
      for (size_t j = 0; j < N; ++j)
      {
        y_cpu[i][j] = 1 + rand() / (1.1 * RAND_MAX);
      }
    }

    //
    // Reference calculation on CPU:
    //
    for (size_t i = 0; i < K; ++i)
    {
      results[i] = 0;
      for (size_t j = 0; j < N; ++j)
      {
        results[i] += x[j] * y_cpu[i][j];
      }
    }

    //
    // Copy data to GPU
    //
    std::cout << "Copying␣data␣to␣GPU..." << std::endl;
    cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice);
    for (size_t i = 0; i < K / 8; ++i)
    {
      cudaMemcpy(cuda_y[i], y[i], sizeof(double) * N * 8, cudaMemcpyHostToDevice);
    }

    //
    // CUDA implementation
    //
```

```
    // repetitions
    for (int j = 0; j < reps; j++){
      // wait for previous operations to finish, then start timings
      CUDA_ERRCHK(cudaDeviceSynchronize());
      timer.reset();

      for (int i = K / 8; i > 0; i--)
      {
        cuda_mdot_product<<<256, 256>>>(N, cuda_x, cuda_y[i - 1], cuda_results2[i - 1]);
      }

      CUDA_ERRCHK(cudaDeviceSynchronize());
      time_vec.push_back(timer.get());
    }
    std::sort(time_vec.begin(), time_vec.end());
    times.push_back(time_vec[reps/2]);

    std::cout << "Copying data to CPU..." << std::endl;
    for (size_t i = 0; i < K / 8; ++i)
    {
      cudaMemcpy(results2[i], cuda_results2[i], sizeof(double) * 8, cudaMemcpyDeviceToHost);
    }

    //
    // Compare results
    //

    std::cout << "Copying results back to host..." << std::endl;
    /*
    for (size_t i = 0; i < K / 8; ++i)
    {
      for (size_t j = 0; j < 8; ++j)
      {
        std::cout << results[i * 8 + j] << " on CPU "
                  << results2[i][j] << " on GPU. Relative difference: "
                  << fabs(results[i * 8 + j] - results2[i][j]) / results[i * 8 + j] << std::endl;
      }
    }*/

    //
    // Clean up:
    // important: clean up inside of loop!
    std::cout << "Cleaning up..." << std::endl;
    for (int i = 0; i < K; ++i)
    {
      free(y_cpu[i]);
    }
    free(y_cpu);

    free(x);
    cudaFree(cuda_x);

    for (size_t i = 0; i < K / 8; ++i)
    {
      free(y[i]);
      cudaFree(cuda_y[i]);
      free(results2[i]);
      cudaFree(cuda_results2[i]);
    }
    free(y);
    free(cuda_y);

```

```
    free(results);
    free(results2);
    free(cuda_results2);
  }

    std::cout << "\ntime␣[s]:\n" << std::endl;
    for (const auto& value : times){
        std::cout << value << "," << std::endl;
    }

  return 0;

}
```