

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.

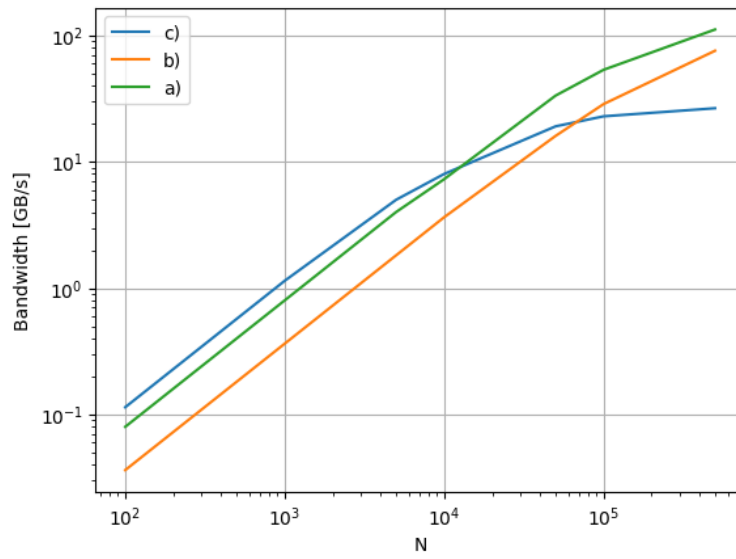


Figure 1: Bandwidth for different N for a), b), c)

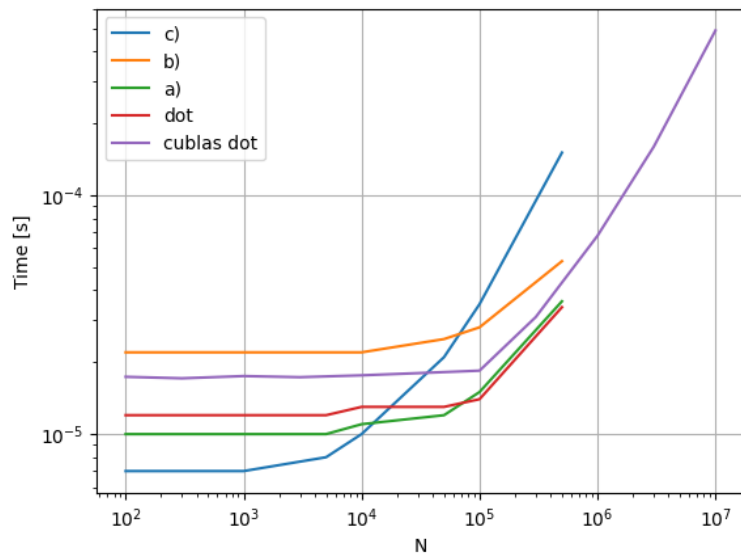


Figure 2: Times for different N for a), b), c)

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} .

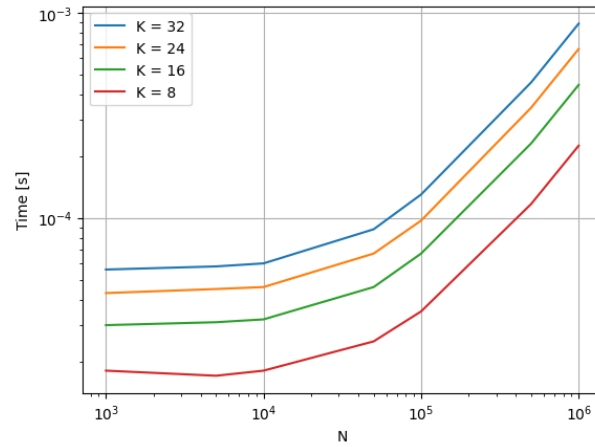


Figure 3: Times for different N logarithmically scaled on GTX GPU

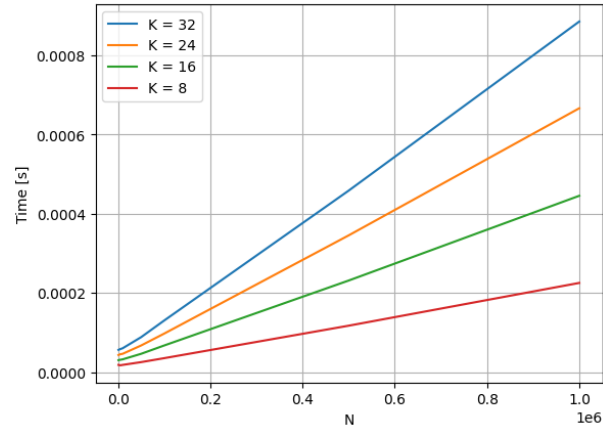


Figure 4: Times for different N linearly scaled on GTX GPU

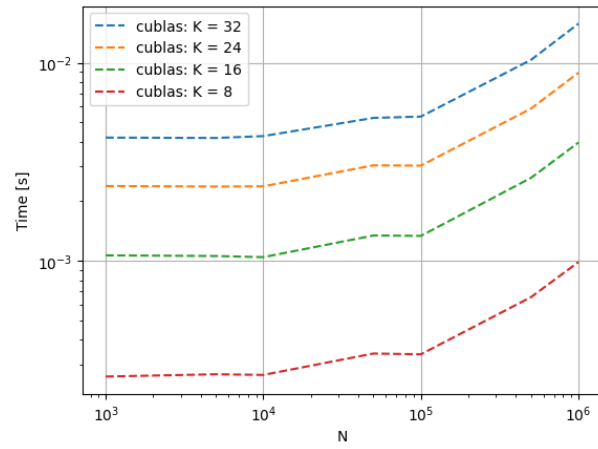


Figure 5: Times for different N with the CUBLAS implementation on K40 GPU

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)

```
1  #include "timer.hpp"
2  #include "cuda_errchk.hpp"
3  #include <algorithm>
4  #include <iostream>
5  #include <stdio.h>
6  #include <vector>
7
8  // result = (x, y)
9  __global__ void cuda_sums(int N, double *x, double *res)
10 {
11     // clean res to wipe results from previous repetition
12     if (blockIdx.x * blockDim.x + threadIdx.x == 0){
13         res[0] = 0;
14         res[1] = 0;
15         res[2] = 0;
16         res[3] = 0;
17     }
18     // shared mem for each
19     __shared__ double shared_sum[256];
20     __shared__ double shared_abs_sum[256];
21     __shared__ double shared_sq_sum[256];
22     __shared__ double shared_zero;
23
24     double entry = 0;
25     double entry_abs = 0;
26     double entry_sq = 0;
27     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
28     {
29         entry += x[i];
30         entry_abs += abs(x[i]);
31         entry_sq += x[i]*x[i];
32         if (x[i] == 0){
33             shared_zero++;
34             //printf("zero_entries=%g\n", shared_zero);
35             atomicAdd(&res[3], shared_zero);
36         }
37     }
38     // write x to every shared memory
39     shared_sum[threadIdx.x] = entry;
40     shared_abs_sum[threadIdx.x] = entry_abs;
41     shared_sq_sum[threadIdx.x] = entry_sq;
42
43     for (int k = blockDim.x / 2; k > 0; k /= 2)
44     {
45         __syncthreads();
46         if (threadIdx.x < k)
47         {
48             // sum
49             shared_sum[threadIdx.x] += shared_sum[threadIdx.x + k];
50             // abs sum
51             shared_abs_sum[threadIdx.x] += shared_abs_sum[threadIdx.x + k];
52             // square sum
53             shared_sq_sum[threadIdx.x] += shared_sq_sum[threadIdx.x + k];
54         }
55     }
56
57     if (threadIdx.x == 0)
```

```

58     {
59         atomicAdd(&res[0], shared_sum[0]);
60         atomicAdd(&res[1], shared_abs_sum[0]);
61         atomicAdd(&res[2], shared_sq_sum[0]);
62     }
63 }
64
65
66 int main()
67 {
68     std::vector<int> N_vec = {100,500,1000,5000,10000,50000,100000,500000};
69     std::vector<double> bandwidth;
70     std::vector<double> times;
71     Timer timer;
72     int reps = 6;
73
74     for (const auto &N : N_vec){
75         std::vector<double> time_vec;
76
77         // Allocate and initialize arrays on CPU
78
79         double *x = (double *)malloc(sizeof(double) * N);
80         // double *y = (double *)malloc(sizeof(double) * N);
81         int res_size = 4; // size of result array
82         double *result = (double *)malloc(sizeof(double) * res_size);
83
84         std::fill(x, x + N, -2);
85         std::fill(result, result + res_size, 0);
86         // add a zero to x
87         x[1] = 0;
88         // std::fill(y, y + N, 2);
89
90         // Allocate and initialize arrays on GPU
91
92         double *cuda_x;
93         //double *cuda_y;
94         double *cuda_result;
95
96         CUDA_ERRCHK(cudaMalloc(&cuda_x, sizeof(double) * N));
97         //CUDA_ERRCHK(cudaMalloc(&cuda_y, sizeof(double) * N));
98         CUDA_ERRCHK(cudaMalloc(&cuda_result, sizeof(double) * res_size));
99
100        CUDA_ERRCHK(cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice));
101        //CUDA_ERRCHK(cudaMemcpy(cuda_y, y, sizeof(double) * N, cudaMemcpyHostToDevice));
102        CUDA_ERRCHK(cudaMemcpy(cuda_result, &result, sizeof(double) * res_size, cudaMemcpyHostToDevice));
103
104        // repetitions
105        for (int j = 0; j < reps; j++){
106            // wait for previous operations to finish, then start timings
107            CUDA_ERRCHK(cudaDeviceSynchronize());
108            timer.reset();
109
110            cuda_sums<<<256, 256>>>(N, cuda_x, cuda_result);
111
112            CUDA_ERRCHK(cudaDeviceSynchronize());
113            time_vec.push_back(timer.get());
114        }
115        std::sort(time_vec.begin(), time_vec.end());
116        times.push_back(time_vec[reps/2]);
117        bandwidth.push_back(N*sizeof(double)/time_vec[reps/2]*1e-9);
118
119        CUDA_ERRCHK(cudaMemcpy(result, cuda_result, sizeof(double)*res_size, cudaMemcpyDeviceToHost))

```

```

120         std::cout << "Result_sum:" << result[0] << std::endl;
121         std::cout << "Result_abs_sum:" << result[1] << std::endl;
122         std::cout << "Result_square_sum:" << result[2] << std::endl;
123         std::cout << "Result_zero_entries:" << result[3] << "\n" << std::endl;
124
125         // Clean up
126
127         CUDA_ERRCHK(cudaFree(cuda_x));
128         //CUDA_ERRCHK(cudaFree(cuda_y));
129         CUDA_ERRCHK(cudaFree(cuda_result));
130         free(x);
131         free(result);
132         //free(y);
133     }
134
135     std::cout << "Gb/s:\n" << std::endl;
136     for (const auto& value : bandwidth){
137         std::cout << value << "," << std::endl;
138     }
139
140     std::cout << "\ns:\n" << std::endl;
141     for (const auto& value : times){
142         std::cout << value << "," << std::endl;
143     }
144
145     CUDA_ERRCHK(cudaDeviceReset()); // for CUDA leak checker to work
146
147     return EXIT_SUCCESS;
148 }
149

```

Task 1.b)

```

1  #include "timer.hpp"
2  #include "cuda_errchk.hpp"
3  #include <algorithm>
4  #include <iostream>
5  #include <stdio.h>
6  #include <vector>
7
8  __global__ void my_warp_reduction(int N, double *x, double *res) {
9
10     if (blockIdx.x * blockDim.x + threadIdx.x == 0){
11         res[0] = 0;
12         res[1] = 0;
13         res[2] = 0;
14         res[3] = 0;
15     }
16
17     double sum = 0;
18     double abs_sum = 0;
19     double sq_sum = 0;
20     double zeros = 0;
21
22     int id = blockIdx.x * blockDim.x + threadIdx.x;
23     for (int i = id; i < N; i += blockDim.x * gridDim.x) {
24         sum += x[i];
25         abs_sum += abs(x[i]);
26         sq_sum += x[i]*x[i];
27         if (x[i]==0) zeros++;

```

```

28     }
29     for (int i=16; i>0; i=i/2){
30         sum += __shfl_down_sync(-1, sum, i);
31         abs_sum += __shfl_down_sync(-1, abs_sum, i);
32         sq_sum += __shfl_down_sync(-1, sq_sum, i);
33         zeros += __shfl_down_sync(-1, zeros, i);
34     } // thread 0 contains sum of all values
35
36     if ((threadIdx.x & 31) == 0){
37         atomicAdd(&res[0], sum);
38         atomicAdd(&res[1], abs_sum);
39         atomicAdd(&res[2], sq_sum);
40         atomicAdd(&res[3], zeros);
41     }
42 }
43
44
45 int main()
46 {
47     std::vector<int> N_vec = {100,500,1000,5000,10000,50000,100000,500000};
48     std::vector<double> bandwidth;
49     std::vector<double> times;
50     Timer timer;
51     int reps = 6;
52
53     for (const auto &N : N_vec){
54         std::vector<double> time_vec;
55
56         // Allocate and initialize arrays on CPU
57
58         double *x = (double *)malloc(sizeof(double) * N);
59         // double *y = (double *)malloc(sizeof(double) * N);
60         int res_size = 4; // size of result array
61         double *result = (double *)malloc(sizeof(double) * res_size);
62
63         std::fill(x, x + N, -2);
64         std::fill(result, result + res_size, 0);
65         // add a zero to x
66         x[1] = 0;
67         // std::fill(y, y + N, 2);
68
69         // Allocate and initialize arrays on GPU
70
71         double *cuda_x;
72         //double *cuda_y;
73         double *cuda_result;
74
75         CUDA_ERRCHK(cudaMalloc(&cuda_x, sizeof(double) * N));
76         //CUDA_ERRCHK(cudaMalloc(&cuda_y, sizeof(double) * N));
77         CUDA_ERRCHK(cudaMalloc(&cuda_result, sizeof(double) * res_size));
78
79         CUDA_ERRCHK(cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice));
80         //CUDA_ERRCHK(cudaMemcpy(cuda_y, y, sizeof(double) * N, cudaMemcpyHostToDevice));
81         CUDA_ERRCHK(cudaMemcpy(cuda_result, &result, sizeof(double) * res_size, cudaMemcpyHostToDevice));
82
83         // repetitions
84         for (int j = 0; j < reps; j++){
85             // wait for previous operations to finish, then start timings
86             CUDA_ERRCHK(cudaDeviceSynchronize());
87             timer.reset();
88
89             my_warp_reduction<<<256, 256>>>(N, cuda_x, cuda_result);

```



```

90         //cuda_sums<<<256, 256>>>(N, cuda_x, cuda_result);
91
92         CUDA_ERRCHK(cudaDeviceSynchronize());
93         time_vec.push_back(timer.get());
94     }
95     std::sort(time_vec.begin(), time_vec.end());
96     times.push_back(time_vec[reps/2]);
97     bandwidth.push_back(N*sizeof(double)/time_vec[reps/2]*1e-9);
98
99     CUDA_ERRCHK(cudaMemcpy(result, cuda_result, sizeof(double)*res_size, cudaMemcpyDeviceToHost));
100
101     std::cout << "Result_sum:" << result[0] << std::endl;
102     std::cout << "Result_abs_sum:" << result[1] << std::endl;
103     std::cout << "Result_square_sum:" << result[2] << std::endl;
104     std::cout << "Result_zero_entries:" << result[3] << "\n" << std::endl;
105
106     // Clean up
107
108     CUDA_ERRCHK(cudaFree(cuda_x));
109     //CUDA_ERRCHK(cudaFree(cuda_y));
110     CUDA_ERRCHK(cudaFree(cuda_result));
111     free(x);
112     free(result);
113     //free(y);
114 }
115
116 std::cout << "Gb/s:\n" << std::endl;
117 for (const auto& value : bandwidth){
118     std::cout << value << "," << std::endl;
119 }
120
121 std::cout << "\ns:\n" << std::endl;
122 for (const auto& value : times){
123     std::cout << value << "," << std::endl;
124 }
125
126 CUDA_ERRCHK(cudaDeviceReset()); // for CUDA leak checker to work
127
128 return EXIT_SUCCESS;
129 }

```

Task 1.c)

```

1  #include "timer.hpp"
2  #include "cuda_errchk.hpp"
3  #include <algorithm>
4  #include <iostream>
5  #include <stdio.h>
6  #include <vector>
7
8  __global__ void my_warp_reduction(int N, double *x, double *res) {
9
10     if (blockIdx.x * blockDim.x + threadIdx.x == 0){
11         res[0] = 0;
12         res[1] = 0;
13         res[2] = 0;
14         res[3] = 0;
15     }
16
17     double sum = 0;

```

```

18     double abs_sum = 0;
19     double sq_sum = 0;
20     double zeros = 0;
21
22
23     int id = blockIdx.x * blockDim.x + threadIdx.x;
24     for (int i = id; i < N; i += blockDim.x * gridDim.x) {
25         sum += x[i];
26         abs_sum += abs(x[i]);
27         sq_sum += x[i]*x[i];
28         if (x[i]==0) zeros++;
29     }
30     for (int i=16; i>0; i=i/2){
31         sum += __shfl_down_sync(-1, sum, i);
32         abs_sum += __shfl_down_sync(-1, abs_sum, i);
33         sq_sum += __shfl_down_sync(-1, sq_sum, i);
34         zeros += __shfl_down_sync(-1, zeros, i);
35     } // thread 0 contains sum of all values
36
37     if ((threadIdx.x & 31) == 0){
38         atomicAdd(&res[0], sum);
39         atomicAdd(&res[1], abs_sum);
40         atomicAdd(&res[2], sq_sum);
41         atomicAdd(&res[3], zeros);
42     }
43 }
44
45
46 int main()
47 {
48     std::vector<int> N_vec = {100,500,1000,5000,10000,50000,100000,500000};
49     std::vector<double> bandwidth;
50     std::vector<double> times;
51     Timer timer;
52     int reps = 6;
53
54     for (const auto &N : N_vec){
55         std::vector<double> time_vec;
56
57         // Allocate and initialize arrays on CPU
58
59         double *x = (double *)malloc(sizeof(double) * N);
60         // double *y = (double *)malloc(sizeof(double) * N);
61         int res_size = 4; // size of result array
62         double *result = (double *)malloc(sizeof(double) * res_size);
63
64         std::fill(x, x + N, -2);
65         std::fill(result, result + res_size, 0);
66         // add a zero to x
67         x[1] = 0;
68         // std::fill(y, y + N, 2);
69
70         // Allocate and initialize arrays on GPU
71
72         double *cuda_x;
73         //double *cuda_y;
74         double *cuda_result;
75
76         CUDA_ERRCHK(cudaMalloc(&cuda_x, sizeof(double) * N));
77         //CUDA_ERRCHK(cudaMalloc(&cuda_y, sizeof(double) * N));
78         CUDA_ERRCHK(cudaMalloc(&cuda_result, sizeof(double) * res_size));
79

```

```

80     CUDA_ERRCHK(cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice));
81     //CUDA_ERRCHK(cudaMemcpy(cuda_y, y, sizeof(double) * N, cudaMemcpyHostToDevice));
82     CUDA_ERRCHK(cudaMemcpy(cuda_result, &result, sizeof(double) * res_size, cudaMemcpyHostToDevice));
83
84     // repetitions
85     for (int j = 0; j < reps; j++){
86         // wait for previous operations to finish, then start timings
87         CUDA_ERRCHK(cudaDeviceSynchronize());
88         timer.reset();
89
90         my_warp_reduction<<<(N+255)/256, 256>>>(N, cuda_x, cuda_result);
91
92         CUDA_ERRCHK(cudaDeviceSynchronize());
93         time_vec.push_back(timer.get());
94     }
95     std::sort(time_vec.begin(), time_vec.end());
96     times.push_back(time_vec[reps/2]);
97     bandwidth.push_back(N*sizeof(double)/time_vec[reps/2]*1e-9);
98
99     CUDA_ERRCHK(cudaMemcpy(result, cuda_result, sizeof(double)*res_size, cudaMemcpyDeviceToHost));
100
101     std::cout << "Result_sum:" << result[0] << std::endl;
102     std::cout << "Result_abs_sum:" << result[1] << std::endl;
103     std::cout << "Result_square_sum:" << result[2] << std::endl;
104     std::cout << "Result_zero_entries:" << result[3] << "\n" << std::endl;
105
106     // Clean up
107
108     CUDA_ERRCHK(cudaFree(cuda_x));
109     //CUDA_ERRCHK(cudaFree(cuda_y));
110     CUDA_ERRCHK(cudaFree(cuda_result));
111     free(x);
112     free(result);
113     //free(y);
114 }
115
116     std::cout << "Gb/s:\n" << std::endl;
117     for (const auto& value : bandwidth){
118         std::cout << value << "," << std::endl;
119     }
120
121     std::cout << "\ns:\n" << std::endl;
122     for (const auto& value : times){
123         std::cout << value << "," << std::endl;
124     }
125
126
127
128     CUDA_ERRCHK(cudaDeviceReset()); // for CUDA leak checker to work
129
130     return EXIT_SUCCESS;
131 }

```

Task 2.abc)

```

1  #include "timer.hpp"
2  #include "cuda_errchk.hpp"
3  #include <algorithm>
4  #include <cuda_runtime.h>
5  #include <cublas_v2.h>

```

```

6  #include <stdio.h>
7  #include <cmath>
8  #include <iostream>
9  #include <vector>
10
11 __global__ void cuda_mdots_product(int N, double *x, double *y, double *res)
12 {
13     // clean res to wipe results from previous repetition
14     if (blockIdx.x * blockDim.x + threadIdx.x == 0){
15         res[0] = 0;
16         res[1] = 0;
17         res[2] = 0;
18         res[3] = 0;
19         res[4] = 0;
20         res[5] = 0;
21         res[6] = 0;
22         res[7] = 0;
23     }
24
25     __shared__ double shared_mem1[256]; // remember to only use 256 threads per block then!
26     __shared__ double shared_mem2[256];
27     __shared__ double shared_mem3[256];
28     __shared__ double shared_mem4[256];
29     __shared__ double shared_mem5[256];
30     __shared__ double shared_mem6[256];
31     __shared__ double shared_mem7[256];
32     __shared__ double shared_mem8[256];
33
34     double dot1 = 0, dot2 = 0, dot3 = 0, dot4 = 0, dot5 = 0, dot6 = 0, dot7 = 0, dot8 = 0;
35     double val_w = 0;
36
37     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
38     {
39         // printf("y = %g\n", y[i]);
40         val_w = x[i];
41         dot1 += val_w * y[i];
42         dot2 += val_w * y[i + 1 * N];
43         dot3 += val_w * y[i + 2 * N];
44         dot4 += val_w * y[i + 3 * N];
45         dot5 += val_w * y[i + 4 * N];
46         dot6 += val_w * y[i + 5 * N];
47         dot7 += val_w * y[i + 6 * N];
48         dot8 += val_w * y[i + 7 * N];
49     }
50
51     shared_mem1[threadIdx.x] = dot1;
52     shared_mem2[threadIdx.x] = dot2;
53     shared_mem3[threadIdx.x] = dot3;
54     shared_mem4[threadIdx.x] = dot4;
55     shared_mem5[threadIdx.x] = dot5;
56     shared_mem6[threadIdx.x] = dot6;
57     shared_mem7[threadIdx.x] = dot7;
58     shared_mem8[threadIdx.x] = dot8;
59
60     for (int k = blockDim.x / 2; k > 0; k /= 2)
61     {
62         __syncthreads();
63         if (threadIdx.x < k)
64         {
65             shared_mem1[threadIdx.x] += shared_mem1[threadIdx.x + k];
66             shared_mem2[threadIdx.x] += shared_mem2[threadIdx.x + k];
67             shared_mem3[threadIdx.x] += shared_mem3[threadIdx.x + k];

```

```

68     shared_mem4[threadIdx.x] += shared_mem4[threadIdx.x + k];
69     shared_mem5[threadIdx.x] += shared_mem5[threadIdx.x + k];
70     shared_mem6[threadIdx.x] += shared_mem6[threadIdx.x + k];
71     shared_mem7[threadIdx.x] += shared_mem7[threadIdx.x + k];
72     shared_mem8[threadIdx.x] += shared_mem8[threadIdx.x + k];
73 }
74 }
75
76 if (threadIdx.x == 0)
77 {
78     atomicAdd(&res[0], shared_mem1[0]);
79     atomicAdd(&res[1], shared_mem2[0]);
80     atomicAdd(&res[2], shared_mem3[0]);
81     atomicAdd(&res[3], shared_mem4[0]);
82     atomicAdd(&res[4], shared_mem5[0]);
83     atomicAdd(&res[5], shared_mem6[0]);
84     atomicAdd(&res[6], shared_mem7[0]);
85     atomicAdd(&res[7], shared_mem8[0]);
86 }
87 }
88
89 int main(void)
90 {
91     std::vector<size_t> N_vec = {1000,5000,10000,50000,100000,500000,1000000};
92     std::vector<double> times;
93     Timer timer;
94     int reps = 6;
95
96     for (const auto &N : N_vec){
97         std::vector<double> time_vec;
98
99         //const size_t N = 4000;
100         const size_t K = 32;
101
102         //
103         // allocate host memory:
104         //
105         std::cout << "Allocating host arrays..." << std::endl;
106         // for x and y
107         double *x = (double *)malloc(sizeof(double) * N);
108         // use double pointers for vectors (like matrix)
109         double **y = (double **)malloc(sizeof(double *) * K / 8);
110
111         // for results on cpu and gpu
112         double *results = (double *)malloc(sizeof(double) * K);
113         for (size_t i = 0; i < K / 8; ++i)
114         {
115             y[i] = (double *)malloc(sizeof(double) * N * 8);
116         }
117         double **results2 = (double **)malloc(sizeof(double) * K / 8);
118         for (size_t i = 0; i < K / 8; ++i)
119         {
120             results2[i] = (double *)malloc(sizeof(double) * 8);
121         }
122         // for the calculation on the cpu to compare with gpu results
123         double **y_cpu = (double **)malloc(sizeof(double *) * K);
124         for (size_t i = 0; i < K; ++i)
125         {
126             y_cpu[i] = (double *)malloc(sizeof(double) * N);
127         }
128
129         //

```

```

130 // allocate device memory
131 //
132 std::cout << "Allocating CUDA arrays..." << std::endl;
133 double *cuda_x;
134 cudaMalloc(&cuda_x, sizeof(double) * N);
135 // pointers to vectors on cpu
136 double **cuda_y = (double **)malloc(sizeof(double *) * K / 8);
137 for (size_t i = 0; i < K / 8; ++i)
138 {
139     cudaMalloc((void **)&cuda_y[i], sizeof(double) * N * 8);
140 }
141 double **cuda_results2 = (double **)malloc(sizeof(double *) * K / 8);
142 for (size_t i = 0; i < K / 8; ++i)
143 {
144     cudaMalloc((void **)&cuda_results2[i], sizeof(double) * 8);
145 }
146
147 //
148 // fill host arrays with values
149 //
150 std::fill(x, x + N, 1.0);
151 for (size_t i = 0; i < K / 8; ++i)
152 {
153     for (size_t j = 0; j < N * 8; ++j)
154     {
155         y[i][j] = 1 + rand() / (1.1 * RAND_MAX);
156     }
157 }
158 // fill y_cpu
159 for (size_t i = 0; i < K; ++i)
160 {
161     for (size_t j = 0; j < N; ++j)
162     {
163         y_cpu[i][j] = 1 + rand() / (1.1 * RAND_MAX);
164     }
165 }
166
167 //
168 // Reference calculation on CPU:
169 //
170 for (size_t i = 0; i < K; ++i)
171 {
172     results[i] = 0;
173     for (size_t j = 0; j < N; ++j)
174     {
175         results[i] += x[j] * y_cpu[i][j];
176     }
177 }
178
179 //
180 // Copy data to GPU
181 //
182 std::cout << "Copying data to GPU..." << std::endl;
183 cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice);
184 for (size_t i = 0; i < K / 8; ++i)
185 {
186     cudaMemcpy(cuda_y[i], y[i], sizeof(double) * N * 8, cudaMemcpyHostToDevice);
187 }
188
189 //
190 // CUDA implementation
191 //

```

```

192 // repetitions
193 for (int j = 0; j < reps; j++){
194     // wait for previous operations to finish, then start timings
195     CUDA_ERRCHK(cudaDeviceSynchronize());
196     timer.reset();
197
198     for (int i = K / 8; i > 0; i--)
199     {
200         cuda_mdot_product<<<256, 256>>>(N, cuda_x, cuda_y[i - 1], cuda_results2[i - 1]);
201     }
202
203     CUDA_ERRCHK(cudaDeviceSynchronize());
204     time_vec.push_back(timer.get());
205 }
206 std::sort(time_vec.begin(), time_vec.end());
207 times.push_back(time_vec[reps/2]);
208
209 std::cout << "Copying data to CPU..." << std::endl;
210 for (size_t i = 0; i < K / 8; ++i)
211 {
212     cudaMemcpy(results2[i], cuda_results2[i], sizeof(double) * 8, cudaMemcpyDeviceToHost);
213 }
214
215 //
216 // Compare results
217 //
218
219 std::cout << "Copying results back to host..." << std::endl;
220 /*
221 for (size_t i = 0; i < K / 8; ++i)
222 {
223     for (size_t j = 0; j < 8; ++j)
224     {
225         std::cout << results[i * 8 + j] << " on CPU "
226             << results2[i][j] << " on GPU. Relative difference: "
227             << fabs(results[i * 8 + j] - results2[i][j]) / results[i * 8 + j] << std::endl;
228     }
229 }*/
230
231 //
232 // Clean up:
233 // important: clean up inside of loop!
234 std::cout << "Cleaning up..." << std::endl;
235 for (int i = 0; i < K; ++i)
236 {
237     free(y_cpu[i]);
238 }
239 free(y_cpu);
240
241 free(x);
242 cudaFree(cuda_x);
243
244 for (size_t i = 0; i < K / 8; ++i)
245 {
246     free(y[i]);
247     cudaFree(cuda_y[i]);
248     free(results2[i]);
249     cudaFree(cuda_results2[i]);
250 }
251 free(y);
252 free(cuda_y);
253

```

```
254     free(results);
255     free(results2);
256     free(cuda_results2);
257 }
258
259     std::cout << "\ntime[s]:\n" << std::endl;
260     for (const auto& value : times){
261         std::cout << value << ", " << std::endl;
262     }
263
264     return 0;
265
266 }
```