



# 360.252 - COMPUTATIONAL SCIENCE ON MANY-CORE ARCHITECTURES

WS 2020 - EXERCISE 6

Christian GOLLMANN, 01435044

Last update: November 30, 2020

## Contents

1	Dot Product with warp shuffles - shared memory	1
2	Dot Product with warp shuffles - warp shuffles	4
3	Dot Product with warp shuffles - performance comparison	6
4	Sparse Matrix Times Dense Matrix - The kernel	8
5	Sparse Matrix Times Dense Matrix - benchmark	12

# 1 Dot Product with warp shuffles - shared memory

I tested my code setting  $N = 5$  which gave me the result shown in figure 1. I decided to implement everything with integers since the `atomicMin()` and `atomicMax()` would need some additional code snippets to work for double which, of course, I could have copied from the cuda documentation. But I wanted to avoid more overhead since I wanted to only focus on the principals here.

## Compile output

[Compilation successful]

## Run output

Input  
-2|-1|0|1|2|  
Sum of all entries: 0  
Sum of maximum values: 6  
Sum of squares: 10  
Max-norm: 2  
minimum value: -2  
maximum value: 2  
number of zeros: 1

Figure 1: Output of shared memory version

Listing 1: Calculations done with shared memory

---

```
1 #include <iostream>
2
3 --global-- void sharedMemoryKernel(const int* x, int* y, const int N) {
4     --shared-- int sharedMemory[7][256];
5     int sum = 0;
6     int maxSum = 0;
7     int sqrSum = 0;
8     int maxMod = 0;
9     int min = x[0];
10    int max = 0;
11    int zeros = 0;
12
13    for (int tid = blockDim.x * blockIdx.x + threadIdx.x;
14         tid < N; tid += gridDim.x * blockDim.x) {
15        int val = x[tid];
16
17        sum += val;
18        maxSum += std::abs(val);
19        sqrSum += val*val;
20        maxMod = std::abs(val) > maxMod ? val : maxMod;
21        min = val < min ? val : min;
22        max = val > max ? val : max;
23        zeros += val == 0 ? 1 : 0;
24    }
25
26    int tid = threadIdx.x;
27    if (tid < N) {
28        sharedMemory[0][threadIdx.x] = sum;
```

```

29     sharedMemory[1][threadIdx.x] = maxSum;
30     sharedMemory[2][threadIdx.x] = sqrSum;
31     sharedMemory[3][threadIdx.x] = maxMod;
32     sharedMemory[4][threadIdx.x] = min;
33     sharedMemory[5][threadIdx.x] = max;
34     sharedMemory[6][threadIdx.x] = zeros;
35
36     __syncthreads();
37     // blockDim.x needs to be a power of 2 in order for this to work
38     for (int i = blockDim.x/2; i != 0; i /= 2) {
39         __syncthreads();
40         if (tid < i) {
41             sharedMemory[0][tid] += sharedMemory[0][tid + i];
42             sharedMemory[1][tid] += sharedMemory[1][tid + i];
43             sharedMemory[2][tid] += sharedMemory[2][tid + i];
44             sharedMemory[3][tid] = sharedMemory[3][tid] > sharedMemory[3][tid + i]
45             ? sharedMemory[3][tid] : sharedMemory[3][tid + i];
46             sharedMemory[4][tid] = sharedMemory[4][tid] < sharedMemory[4][tid + i]
47             ? sharedMemory[4][tid] : sharedMemory[4][tid + i];
48             sharedMemory[5][tid] = sharedMemory[5][tid] > sharedMemory[5][tid + i]
49             ? sharedMemory[5][tid] : sharedMemory[5][tid + i];
50             sharedMemory[6][tid] += sharedMemory[6][tid + i];
51         }
52     }
53 }
54
55 if (tid == 0) {
56     atomicAdd(y, sharedMemory[0][0]);
57     atomicAdd(y+1, sharedMemory[1][0]);
58     atomicAdd(y+2, sharedMemory[2][0]);
59     atomicMax(y+3, sharedMemory[3][0]);
60     atomicMin(y+4, sharedMemory[4][0]);
61     atomicMax(y+5, sharedMemory[5][0]);
62     atomicAdd(y+6, sharedMemory[6][0]);
63 }
64 }
65
66 template <typename T>
67 void printContainer(T container, int N) {
68     for (int i = 0; i < N; i++) {
69         std::cout << container[i] << " | ";
70     }
71 }
72
73
74 int main() {
75
76     int N = 5;
77
78     int *x = (int *)malloc(sizeof(int) * N);
79     int *y = (int *)malloc(sizeof(int) * 7);
80
81     for (int i = 0; i < N; i++) {
82         x[i] = i - N/2;
83     }
84
85     int *cuda_x;

```

```

86     int *cuda_y;
87     cudaMalloc(&cuda_x, sizeof( int) * N);
88     cudaMalloc(&cuda_y, sizeof( int) * 7);
89
90     cudaMemcpy(cuda_x, x, sizeof( int) * N, cudaMemcpyHostToDevice);
91
92     sharedMemoryKernel<<<256, 256>>>(cuda_x, cuda_y, N);
93
94     cudaMemcpy(y, cuda_y, sizeof( int) * 7, cudaMemcpyDeviceToHost);
95
96     std::cout << "Input" << std::endl;
97     printContainer(x, N);
98     std::cout << std::endl;
99
100    std::cout << "Sum of all entries: " << y[0] << std::endl;
101    std::cout << "Sum of maximum values: " << y[1] << std::endl;
102    std::cout << "Sum of squares: " << y[2] << std::endl;
103    std::cout << "Max-norm: " << y[3] << std::endl;
104    std::cout << "minimum value: " << y[4] << std::endl;
105    std::cout << "maximum value: " << y[5] << std::endl;
106    std::cout << "number of zeros: " << y[6] << std::endl;
107
108    free(x);
109    free(y);
110    cudaFree(cuda_x);
111    cudaFree(cuda_y);
112
113    return EXIT_SUCCESS;
114 }

```

---

## 2 Dot Product with warp shuffles - warp shuffles

I tested my code by comparing to the version in point 1.

Listing 2: Calculations done with warp shuffles

---

```
1  #include <iostream>
2
3  __global__ void shuffleKernel(const int* x, int* y, const int N) {
4
5      int sum                = 0;
6      int maxSum             = 0;
7      int sqrSum             = 0;
8      int maxMod             = 0;
9      int min                = x[0];
10     int max                 = 0;
11     int zeros               = 0;
12
13     for (int tid = blockDim.x * blockIdx.x + threadIdx.x;
14          tid < N; tid += gridDim.x * blockDim.x) {
15         int val = x[tid];
16
17         sum      += val;
18         maxSum   += std::abs(val);
19         sqrSum   += val*val;
20         maxMod   = std::abs(val) > maxMod ? val : maxMod;
21         min      = val < min ? val : min;
22         max      = val > max ? val : max;
23         zeros    += val == 0 ? 1 : 0;
24     }
25
26     int tid = threadIdx.x;
27     for (int i = warpSize / 2; i != 0; i /= 2) {
28         sum      += __shfl_down_sync(0xffffffff, sum, i);
29         maxSum   += __shfl_down_sync(0xffffffff, maxSum, i);
30         sqrSum   += __shfl_down_sync(0xffffffff, sqrSum, i);
31         int temporary = __shfl_down_sync(0xffffffff, maxMod, i);
32         maxMod   = temporary > maxMod ? temporary : maxMod;
33         temporary = __shfl_down_sync(0xffffffff, min, i);
34         min      = temporary < min ? temporary : min;
35         temporary = __shfl_down_sync(0xffffffff, max, i);
36         max      = temporary > max ? temporary : max;
37         zeros    += __shfl_down_sync(0xffffffff, zeros, i);
38     }
39     __syncthreads();
40     if (tid % warpSize == 0) {
41         atomicAdd(y, sum);
42         atomicAdd(y+1, maxSum);
43         atomicAdd(y+2, sqrSum);
44         atomicMax(y+3, maxMod);
45         atomicMin(y+4, min);
46         atomicMax(y+5, max);
47         atomicAdd(y+6, zeros);
48     }
49 }
50
51 template <typename T>
```

```

52 void printContainer(T container, int N) {
53     for (int i = 0; i < N; i++) {
54         std::cout << container[i] << " | ";
55     }
56 }
57
58
59 int main() {
60
61     int N = 100000;
62
63     int *x = (int *)malloc(sizeof(int) * N);
64     int *y = (int *)malloc(sizeof(int) * 7);
65
66     for (int i = 0; i < N; i++) {
67         x[i] = i - N/2;
68     }
69
70     int *cuda_x;
71     int *cuda_y;
72     cudaMalloc(&cuda_x, sizeof(int) * N);
73     cudaMalloc(&cuda_y, sizeof(int) * 7);
74
75     cudaMemcpy(cuda_x, x, sizeof(int) * N, cudaMemcpyHostToDevice);
76
77     shuffleKernel<<<N/256, 128>>>(cuda_x, cuda_y, N);
78
79     cudaMemcpy(y, cuda_y, sizeof(int) * 7, cudaMemcpyDeviceToHost);
80
81     //std::cout << "Input" << std::endl;
82     //printContainer(x, N);
83     //std::cout << std::endl;
84
85     std::cout << "Sum of all entries: " << y[0] << std::endl;
86     std::cout << "Sum of maximum values: " << y[1] << std::endl;
87     std::cout << "Sum of squares: " << y[2] << std::endl;
88     std::cout << "Max-norm: " << y[3] << std::endl;
89     std::cout << "minimum value: " << y[4] << std::endl;
90     std::cout << "maximum value: " << y[5] << std::endl;
91     std::cout << "number of zeros: " << y[6] << std::endl;
92
93     free(x);
94     free(y);
95     cudaFree(cuda_x);
96     cudaFree(cuda_y);
97
98     return EXIT_SUCCESS;
99 }

```

---

### 3 Dot Product with warp shuffles - performance comparison

In order to compare the performances, I launched every kernel with  $[N/256, 128]$ . I additionally implemented the dot product in shared and shuffled version. The results can be found in figure 5. It can be seen that the shuffled versions perform a little better than the shared memory versions unless for one data point at the last N. I cannot really explain this behaviour. I doublechecked my tests and didn't find any errors in there.

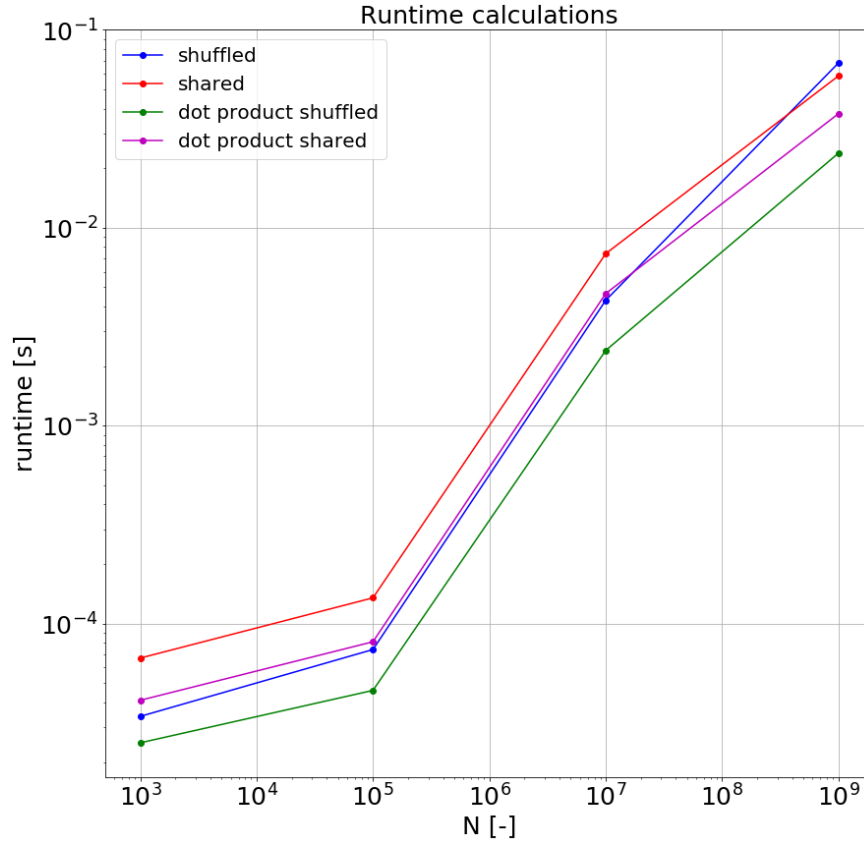


Figure 2: Runtimes for different versions

Listing 3: Dot product using shared memory

```

1  __global__ void dot_product(int* x, int* y, int* dot, int N) {
2
3      int index = threadIdx.x + blockDim.x * blockIdx.x;
4      int stride = blockDim.x * gridDim.x;
5
6      __shared__ int cache[128];
7
8      int temp = 0;
9      while (index < N) {
10         temp += x[index] * y[index];
11         index += stride;
12     }
13
14     cache[threadIdx.x] = temp;
15
16     __syncthreads();

```



```

17
18     for (int i = blockDim.x/2; i > 0; i/= 2) {
19         __syncthreads();
20         if (threadIdx.x < i)
21             cache[threadIdx.x] += cache[threadIdx.x + i];
22     }
23
24     if (threadIdx.x == 0)
25         atomicAdd(dot, cache[0]);
26
27 }

```

---

Listing 4: Dot product using warp shuffles

```

1  __global__ void dot_product_shuffle(int* x, int* y, int* dot, int N) {
2
3     int index = threadIdx.x + blockDim.x * blockIdx.x;
4     int stride = blockDim.x * gridDim.x;
5
6     int temp = 0;
7     while (index < N) {
8         temp += x[index] * y[index];
9         index += stride;
10    }
11
12    for (int i = warpSize / 2; i != 0; i /= 2) {
13        temp += __shfl_down_sync(0xffffffff, temp, i);
14    }
15
16    __syncthreads();
17
18    int tid = threadIdx.x;
19
20    if (tid % warpSize == 0) {
21        atomicAdd(dot, temp);
22    }
23
24 }

```

---

## 4 Sparse Matrix Times Dense Matrix - The kernel

I tested my implementation of the sparse matrix times dense matrix with the matrix from the lecture, see figure 3, and by comparing it to a sparse matrix times a vector. Details and results please take from figure 4 and the code listing. I implemented it in column major style.

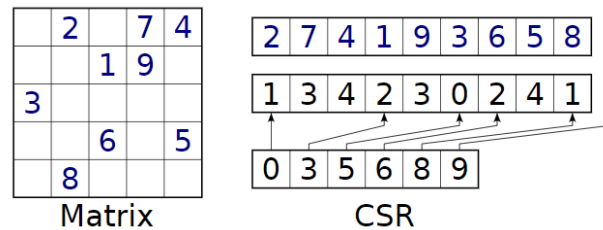


Figure 3: testmatrix from the lecture

### Run output

```
x:
1|2|3|4|5|
Result:
52|39|3|43|16|

X:
1|2|3|4|5|1|2|3|4|5|1|2|3|4|1|
Result:
52|39|3|43|16|52|39|3|43|16|36|39|3|23|16|
```

Figure 4: results of matrix vector and matrix matrix product

Listing 5: matrix\*vector and matrix\*matrix

```
1 #include <iostream>
2
3 // y = A * x
4 __global__ void sparseVector(int N, int *csr_rowoffsets,
5                               int *csr_colindices, double *csr_values,
6                               double *x, double *y)
7 {
8     for (int i = blockIdx.x * blockDim.x + threadIdx.x;
9          i < N; i += blockDim.x * gridDim.x) {
10         double sum = 0;
11
12         for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
13             sum += csr_values[k] * x[csr_colindices[k]];
14         }
15
16         y[i] = sum;
17     }
18 }
19
20
21 // Y = A * X
22 __global__ void sparseDense(int N, int K, int *csr_rowoffsets,
```

```

23     int *csr_colindices , double *csr_values ,
24     double *x, double *y)
25 {
26     for (int i = blockIdx.x * blockDim.x + threadIdx.x;
27     i < N; i += blockDim.x * gridDim.x) {
28
29         for (int k = 0; k < K; k++) {
30             double sum = 0;
31
32             for (int jj = csr_rowoffsets[i]; jj < csr_rowoffsets[i + 1]; jj++) {
33                 sum += csr_values[jj] * x[csr_colindices[jj] + N*k];
34             }
35
36             y[i + N*k] = sum;
37         }
38     }
39 }
40 }
41
42
43 template <typename T>
44 void printContainer(T container , int N) {
45     for (int i = 0; i < N; i++) {
46         std::cout << container[i] << " | ";
47     }
48     std::cout << std::endl;
49 }
50
51
52 int main() {
53
54     int N = 5;
55     int K = 3;
56
57     double *values = ( double *)malloc(sizeof( double ) * 9);
58     int *colindices = (int*)malloc(sizeof(int) * 9);
59     int *offsets = (int*)malloc(sizeof(int) * 6);
60
61     double *x = ( double *)malloc(sizeof( double ) * N);
62     double *X = (double*)malloc(sizeof(double) * N*K);
63
64     double *y = ( double *)malloc(sizeof( double ) * N);
65     double *Y = (double*)malloc(sizeof(double) * N*K);
66
67     values[0] = 2;
68     values[1] = 7;
69     values[2] = 4;
70     values[3] = 1;
71     values[4] = 9;
72     values[5] = 3;
73     values[6] = 6;
74     values[7] = 5;
75     values[8] = 8;
76
77     colindices[0] = 1;
78     colindices[1] = 3;
79     colindices[2] = 4;

```

```

80     colindices[3] = 2;
81     colindices[4] = 3;
82     colindices[5] = 0;
83     colindices[6] = 2;
84     colindices[7] = 4;
85     colindices[8] = 1;
86
87     offsets[0] = 0;
88     offsets[1] = 3;
89     offsets[2] = 5;
90     offsets[3] = 6;
91     offsets[4] = 8;
92     offsets[5] = 9;
93
94     x[0] = 1;
95     x[1] = 2;
96     x[2] = 3;
97     x[3] = 4;
98     x[4] = 5;
99
100    X[0] = 1;
101    X[1] = 2;
102    X[2] = 3;
103    X[3] = 4;
104    X[4] = 5;
105    X[5] = 1;
106    X[6] = 2;
107    X[7] = 3;
108    X[8] = 4;
109    X[9] = 5;
110    X[10] = 1;
111    X[11] = 2;
112    X[12] = 3;
113    X[13] = 4;
114    X[14] = 1;
115
116    double *cuda_values;
117    int *cuda_colindices;
118    int *cuda_offsets;
119    double *cuda_x;
120    double *cuda_X;
121    double *cuda_y;
122    double *cuda_Y;
123
124    cudaMalloc(&cuda_values, sizeof( double) * 9);
125    cudaMalloc(&cuda_colindices, sizeof( int) * 9);
126    cudaMalloc(&cuda_offsets, sizeof( int) * 6);
127    cudaMalloc(&cuda_x, sizeof( double) * N);
128    cudaMalloc(&cuda_X, sizeof( double) * N*K);
129    cudaMalloc(&cuda_y, sizeof( double) * N);
130    cudaMalloc(&cuda_Y, sizeof( double) * N*K);
131
132    cudaMemcpy(cuda_values, values, sizeof( double) * 9, cudaMemcpyHostToDevice);
133    cudaMemcpy(cuda_colindices, colindices, sizeof( int) * 9, cudaMemcpyHostToDevice);
134    cudaMemcpy(cuda_offsets, offsets, sizeof( int) * 6, cudaMemcpyHostToDevice);
135    cudaMemcpy(cuda_x, x, sizeof( double) * N, cudaMemcpyHostToDevice);
136    cudaMemcpy(cuda_X, X, sizeof( double) * N*K, cudaMemcpyHostToDevice);

```

```

137
138     sparseVector<<<256, 256>>>(N, cuda_offsets ,
139     cuda_colindices , cuda_values , cuda_x , cuda_y );
140     cudaMemcpy(y, cuda_y , sizeof(double) * N, cudaMemcpyDeviceToHost);
141
142     sparseDense<<<256, 256>>>(N, K, cuda_offsets ,
143     cuda_colindices , cuda_values , cuda_X , cuda_Y );
144     cudaMemcpy(Y, cuda_Y , sizeof(double) * N*K, cudaMemcpyDeviceToHost);
145
146
147     std::cout << "x : " << std::endl;
148     printContainer(x, N);
149     std::cout << "Result : " << std::endl;
150     printContainer(y, N);
151
152     std::cout << std::endl;
153     std::cout << "X : " << std::endl;
154     printContainer(X, N*K);
155     std::cout << "Result : " << std::endl;
156     printContainer(Y, N*K);
157
158     free(x);
159     free(X);
160     free(y);
161     free(Y);
162     free(values);
163     free(colindices);
164     free(offsets);
165
166     cudaFree(cuda_values);
167     cudaFree(cuda_colindices);
168     cudaFree(cuda_offsets);
169     cudaFree(cuda_x);
170     cudaFree(cuda_X);
171     cudaFree(cuda_y);
172     cudaFree(cuda_Y);
173
174     return EXIT_SUCCESS;
175 }

```

---

## 5 Sparse Matrix Times Dense Matrix - benchmark

I added a row major implementation to my code, both kernels can be seen in the listing. For the benchmark I constructed dummy matrices without specific values, except for a few. Maybe this is a reason why my row and column major implementations perform almost identical. But on the other hand, both my implementations are very similar, even for the strided access in line 13 and 35 of the first code listing. I also timed the Memcopy(), since the matrix vector has to do that K times which makes my timings more realistic. As expected, the matrix \* vector K times is always slower than the matrix \* matrix multiplication.

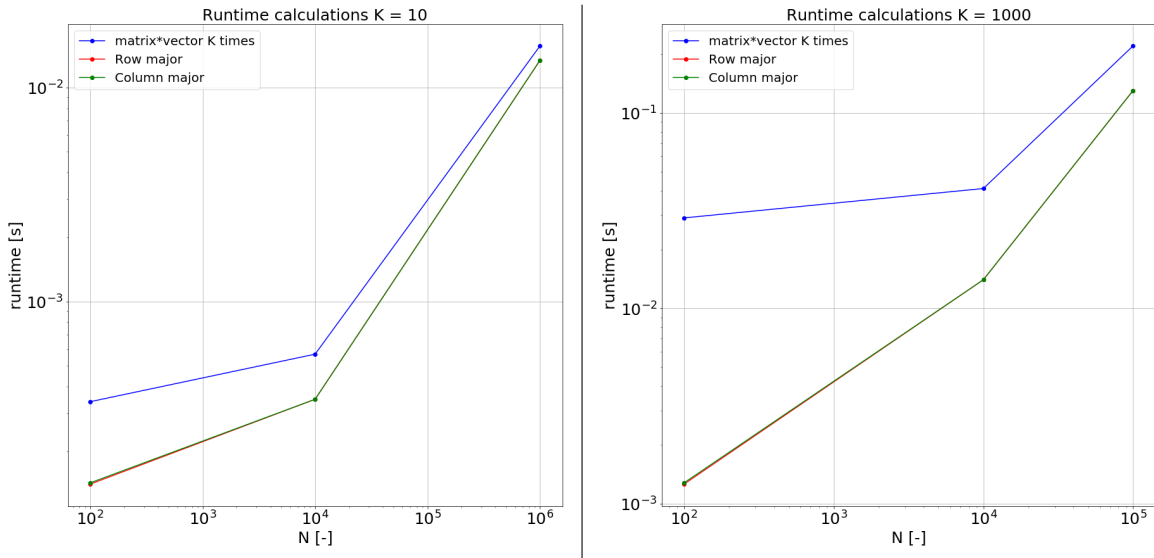


Figure 5: Runtimes of calculations for different N and K

Listing 6: Row- and Column major matrix matrix multiplication

```

1  // Y = A * X
2  __global__ void sparseDenseRowMajor(int N, int K, int *csr_rowoffsets,
3  int *csr_colindices, double *csr_values,
4  double *x, double *y)
5  {
6  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
7  i < N; i += blockDim.x * gridDim.x) {
8
9      for (int k = 0; k < K; k++) {
10         double sum = 0;
11
12         for (int jj = csr_rowoffsets[i]; jj < csr_rowoffsets[i + 1]; jj++) {
13             sum += csr_values[jj] * x[csr_colindices[jj]*K + k];
14         }
15
16         y[i + N*k] = sum;
17     }
18 }
19 }
20 }
21
22
23 // Y = A * X

```

```

24 --global-- void sparseDenseColumnMajor(int N, int K, int *csr_rowoffsets ,
25     int *csr_colindices , double *csr_values ,
26     double *x, double *y)
27 {
28     for (int i = blockIdx.x * blockDim.x + threadIdx.x;
29         i < N; i += blockDim.x * gridDim.x) {
30
31         for (int k = 0; k < K; k++) {
32             double sum = 0;
33
34             for (int jj = csr_rowoffsets[i]; jj < csr_rowoffsets[i + 1]; jj++) {
35                 sum += csr_values[jj] * x[csr_colindices[jj] + N*k];
36             }
37
38             y[i + N*k] = sum;
39         }
40     }
41 }
42 }

```

---

Listing 7: Code for the timing benchmark

---

```

1  #include <iostream>
2  #include "timer.hpp"
3  #include <algorithm>
4  #include <iostream>
5  #include <stdio.h>
6  #include <vector>
7
8  // y = A * x
9  --global-- void sparseVector(int N, int *csr_rowoffsets ,
10                             int *csr_colindices , double *csr_values ,
11                             double *x, double *y)
12 {
13     for (int i = blockIdx.x * blockDim.x + threadIdx.x;
14         i < N; i += blockDim.x * gridDim.x) {
15         double sum = 0;
16
17         for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
18             sum += csr_values[k] * x[csr_colindices[k]];
19         }
20
21         y[i] = sum;
22     }
23 }
24
25
26 // Y = A * X
27 --global-- void sparseDenseRowMajor(int N, int K, int *csr_rowoffsets ,
28     int *csr_colindices , double *csr_values ,
29     double *x, double *y)
30 {
31     for (int i = blockIdx.x * blockDim.x + threadIdx.x;
32         i < N; i += blockDim.x * gridDim.x) {
33
34         for (int k = 0; k < K; k++) {
35             double sum = 0;

```

```

36
37     for (int jj = csr_rowoffsets[i]; jj < csr_rowoffsets[i + 1]; jj++) {
38         sum += csr_values[jj] * x[csr_colindices[jj]*K + k];
39     }
40
41     y[i + N*k] = sum;
42 }
43
44 }
45 }
46
47
48 // Y = A * X
49 __global__ void sparseDenseColumnMajor(int N, int K, int *csr_rowoffsets,
50     int *csr_colindices, double *csr_values,
51     double *x, double *y)
52 {
53     for (int i = blockIdx.x * blockDim.x + threadIdx.x;
54         i < N; i += blockDim.x * gridDim.x) {
55
56         for (int k = 0; k < K; k++) {
57             double sum = 0;
58
59             for (int jj = csr_rowoffsets[i]; jj < csr_rowoffsets[i + 1]; jj++) {
60                 sum += csr_values[jj] * x[csr_colindices[jj] + N*k];
61             }
62
63             y[i + N*k] = sum;
64         }
65     }
66 }
67 }
68
69
70 template <typename T>
71 void printContainer(T container, int N) {
72     for (int i = 0; i < N; i++) {
73         std::cout << container[i] << " | ";
74     }
75     std::cout << std::endl;
76 }
77
78
79 int main() {
80
81     Timer tim1, tim2, tim3;
82     int N = 100000;
83     int K = 1000;
84
85     double *values = (double *)malloc(sizeof(double) * N*10);
86     int *colindices = (int*)malloc(sizeof(int) * N*10);
87     int *offsets = (int*)malloc(sizeof(int) * (N+1));
88
89     double *x = (double *)malloc(sizeof(double) * N);
90     double *X = (double*)malloc(sizeof(double) * N*K);
91
92     double *y = (double *)malloc(sizeof(double) * N);

```



```

93     double *Y = (double*)malloc(sizeof(double) * N*K);
94
95     values[0] = 2;
96     values[1] = 7;
97     values[2] = 4;
98     values[3] = 1;
99     values[4] = 9;
100    values[5] = 3;
101    values[6] = 6;
102    values[7] = 5;
103    values[8] = 8;
104
105    colindices[0] = 1;
106    colindices[1] = 3;
107    colindices[2] = 4;
108    colindices[3] = 2;
109    colindices[4] = 3;
110    colindices[5] = 0;
111    colindices[6] = 2;
112    colindices[7] = 4;
113    colindices[8] = 1;
114
115    offsets[0] = 0;
116    offsets[1] = 3;
117    offsets[2] = 5;
118    offsets[3] = 6;
119    offsets[4] = 8;
120    offsets[5] = 9;
121
122    x[0] = 1;
123    x[1] = 2;
124    x[2] = 3;
125    x[3] = 4;
126    x[4] = 5;
127
128    X[0] = 1;
129    X[1] = 1;
130    X[2] = 1;
131    X[3] = 2;
132    X[4] = 2;
133    X[5] = 2;
134    X[6] = 3;
135    X[7] = 3;
136    X[8] = 3;
137    X[9] = 4;
138    X[10] = 4;
139    X[11] = 4;
140    X[12] = 5;
141    X[13] = 5;
142    X[14] = 5;
143
144    double *cuda_values;
145    int *cuda_colindices;
146    int *cuda_offsets;
147    double *cuda_x;
148    double *cuda_X;
149    double *cuda_y;

```

```

150     double      *cuda_Y;
151
152     cudaMalloc(&cuda_values , sizeof( double) * N*10);
153     cudaMalloc(&cuda_colindices , sizeof( int) * N*10);
154     cudaMalloc(&cuda_offsets , sizeof( int) * (N+1));
155     cudaMalloc(&cuda_x , sizeof( double) * N);
156     cudaMalloc(&cuda_X , sizeof( double) * N*K);
157     cudaMalloc(&cuda_y , sizeof( double) * N);
158     cudaMalloc(&cuda_Y , sizeof( double) * N*K);
159
160     cudaMemcpy(cuda_values , values , sizeof( double) * N*10, cudaMemcpyHostToDevice);
161     cudaMemcpy(cuda_colindices , colindices , sizeof( int) * N*10, cudaMemcpyHostToDevice);
162     cudaMemcpy(cuda_offsets , offsets , sizeof( int) * (N+1), cudaMemcpyHostToDevice);
163     cudaMemcpy(cuda_x , x , sizeof( double) * N, cudaMemcpyHostToDevice);
164     cudaMemcpy(cuda_X , X , sizeof( double) * N*K, cudaMemcpyHostToDevice);
165
166     std::vector<double> timings1;
167     for(int reps=0; reps < 10; ++reps) {
168         tim1.reset();
169         for (int j = 0; j < K; j++) {
170             sparseVector<<<256, 256>>>(N, cuda_offsets , cuda_colindices , cuda_values ,
171             cuda_x , cuda_y);
172             cudaMemcpy(y, cuda_y , sizeof(double) * N, cudaMemcpyDeviceToHost);
173         }
174         timings1.push_back(tim1.get());
175     }
176     std::sort(timings1.begin(), timings1.end());
177     double time_elapsed1 = timings1[10/2];
178     std::cout << "Time elapsed vector: " << time_elapsed1 << std::endl << std::endl;
179
180
181
182
183
184
185     std::vector<double> timings2;
186     for(int reps=0; reps < 10; ++reps) {
187         tim2.reset();
188         sparseDenseRowMajor<<<256, 256>>>(N, K, cuda_offsets , cuda_colindices ,
189         cuda_values , cuda_X , cuda_Y);
190         cudaMemcpy(Y, cuda_Y , sizeof(double) * N*K, cudaMemcpyDeviceToHost);
191         timings2.push_back(tim2.get());
192     }
193     std::sort(timings2.begin(), timings2.end());
194     double time_elapsed2 = timings2[10/2];
195     std::cout << "Time elapsed row: " << time_elapsed2 << std::endl << std::endl;
196
197
198
199
200
201     std::vector<double> timings3;
202     for(int reps=0; reps < 10; ++reps) {
203         tim3.reset();
204         sparseDenseRowMajor<<<256, 256>>>(N, K, cuda_offsets , cuda_colindices ,
205         cuda_values , cuda_X , cuda_Y);
206         cudaMemcpy(Y, cuda_Y , sizeof(double) * N*K, cudaMemcpyDeviceToHost);

```

```

207         timings3.push_back(tim3.get());
208     }
209     std::sort(timings3.begin(), timings3.end());
210     double time_elapsed3 = timings3[10/2];
211     std::cout << "Time elapsed column: " << time_elapsed3 << std::endl << std::endl;
212
213
214
215
216     free(x);
217     free(X);
218     free(y);
219     free(Y);
220     free(values);
221     free(colindices);
222     free(offsets);
223
224     cudaFree(cuda_values);
225     cudaFree(cuda_colindices);
226     cudaFree(cuda_offsets);
227     cudaFree(cuda_x);
228     cudaFree(cuda_X);
229     cudaFree(cuda_y);
230     cudaFree(cuda_Y);
231
232
233     return EXIT_SUCCESS;
234 }

```

---