



INSTITUTE OF MICROELECTRONICS

360.252 COMPUTATIONAL SCIENCE ON MANY-CORE ARCHITECTURES

Exercise 6

Author:

Peter HOLZNER, 01426733

Submission: December 1, 2020

Contents

1 Task 1: Dot Product with Warp Shuffles	1
1.1 Implementation	1
1.2 Runtime comparisons	2
2 Task 2: Sparse Matrix Times Dense Matrix	4
2.1 K is even	4
2.2 K is uneven	6
3 Code and Kernels	8

1 Task 1: Dot Product with Warp Shuffles

Code listings for this task:

- Kernels: Listing 2
- Main: Listing 3

1.1 Implementation

The first hurdle I encountered was the fact that there exists no provided implementation of `atomicMin(...)` and `atomicMax(...)` in the CUDA (standard-) library. So, I had to write my own versions using the following (additional) resources:

- NVIDIA CUDA programming guide
- [stackoverflow](#)

Listing 1: `atomicMax(..)` implementation using `atomicCAS`.

```
1 __device__ void atomicMax(double* address, double val){
2   unsigned long long int* address_as_u11 = (unsigned long long int*) address;
3   unsigned long long int old = *address_as_u11, assumed;
4   do {
5       assumed = old;
6       old = atomicCAS(address_as_u11, assumed, __double_as_longlong(fmax(val,
7   } while (assumed != old);
```

With that out of the way, the next hurdle was adapting the kernel for synchronization using warp shuffles, which was not too difficult. One needs to remember though, that synchronization now occurs between threads of one warp and NOT within a block, which can consist of multiple warps (i.e if `BLOCK_SIZE > warpSize`). That means, that using warp shuffles, more atomic function calls are needed, if `BLOCK_SIZE > warpSize`. We're going to assume from here on that this is the case - usually `BLOCK_SIZE` ranges from `[128, 512]` and `warpSize` is always 32, as far as I could find out ¹. Compared to the version using shared memory, one trades shared memory accesses + fewer atomic function calls for warp shuffles + more atomic function calls. We'll see how and where this pays off.

¹I saw, that AMD uses a similar concept called wavefronts, though I heard they use a size of 64 threads here.

1.2 Runtime comparisons

In Fig. 1, we can see an overview of the different tested versions. We can see that the GPU versions are faster than the CPU reference version for $N > 10^4$. Our implementations of the dot product, both the shared `dot` and the warp shuffle version `dot_warp` are faster or equally as fast as the cuBLAS version.

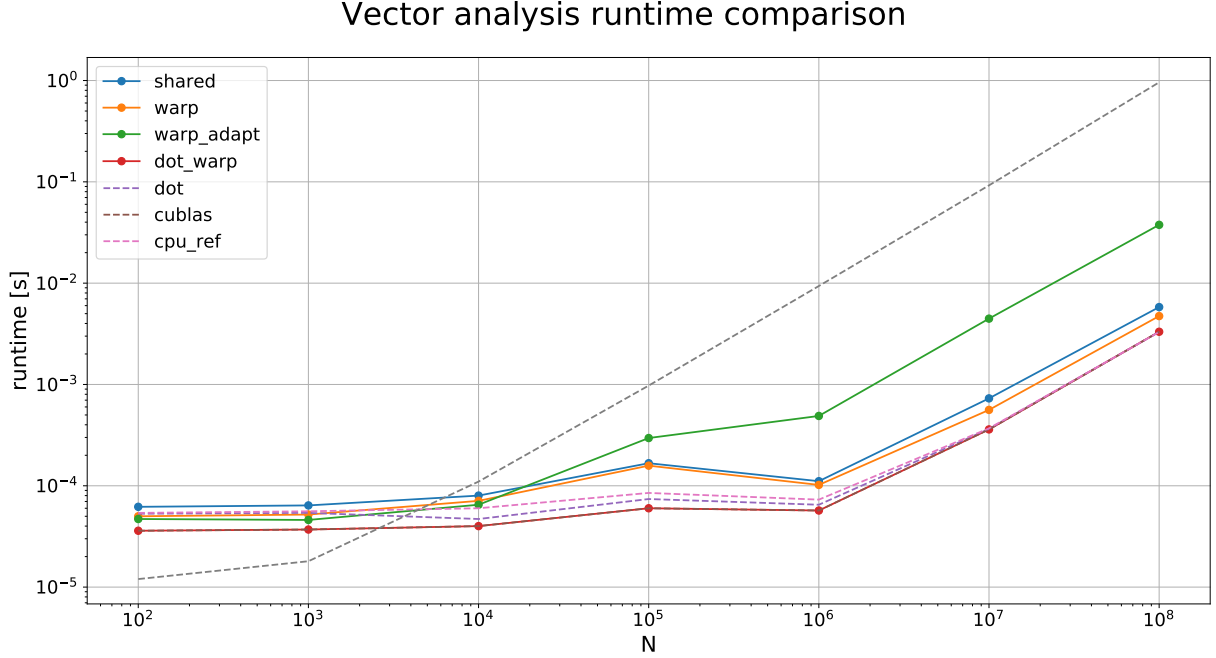


Fig. 1. Runtimes of the different vector dot product based kernels.

In Fig. 2, I split the above plot into two different ranges (small N and big N , or CPU faster and GPU faster, respectively) and also dropped some references that were not of interest (namely the runtimes of the CPU and the cuBLAS implementations). Firstly, the adaptive version of the warp kernel performs the worst. Adaptive means, that the grid size was set at runtime to be $grid_size = (N + BLOCK_SIZE - 1) / BLOCK_SIZE$. I assume that the reason for this it that a less optimized, more general version of the kernel is selected because of the runtime based computation of the grid size. You have noted this behaviour to me when I experimented with this during one of the first exercises.

The warp version of the kernel using a grid size, that is set at compile time, is the fastest but not by a lot compared to the shared memory version. The difference between this warp and the shared version seem to be rather constant across all N . The difference in runtime between the dot product kernel versions using shared memory (`dot`) and warp shuffles (`dot_warp`) is slightly in favor of the warp version again, though the difference decreases as N grows. Here, the difference to the cuBLAS implementation is also negligible. I assume, that the reason is that the vast majority of the time is spent in the summation-for-loop compared to the synchronization part of the respective implementation. One would need to carefully profile the algorithm to confirm, though, so I will leave this part as simple speculation on my side.

As a closing remark, I want to add that warp shuffles provide a more intuitive, less cumbersome and more readable method of synchronization between threads. Since these benefits are present WITHOUT sacrificing performance, I'm very happy to know this technique now.

Vector analysis runtime comparison

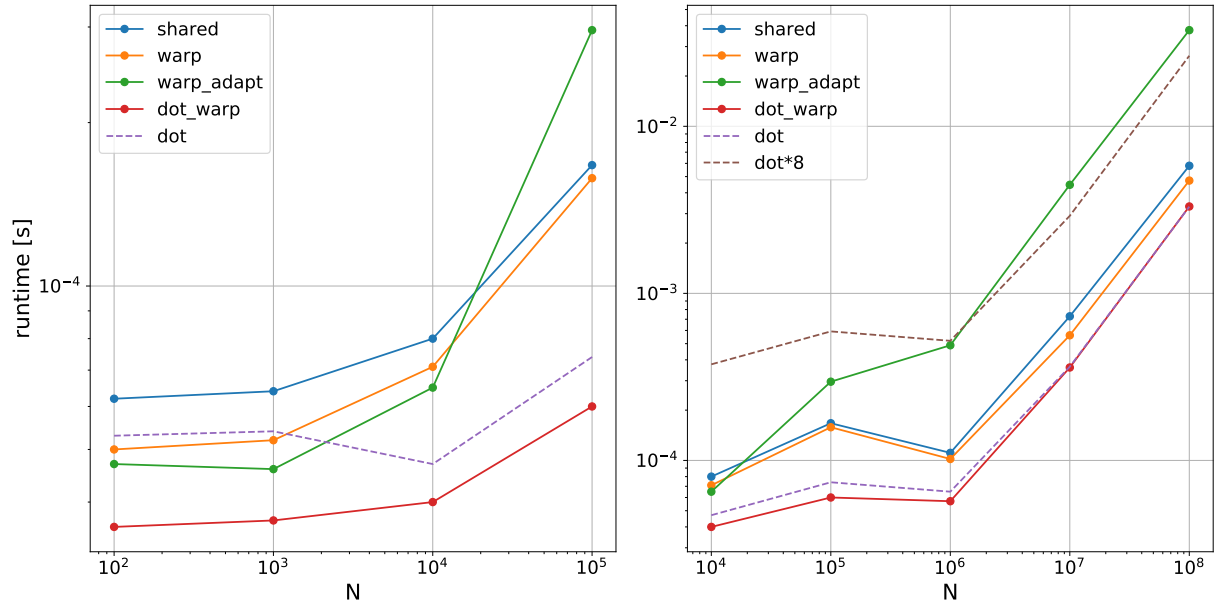


Fig. 2. Runtimes of the different vector dot product based kernels.

2 Task 2: Sparse Matrix Times Dense Matrix

Code listings for this task:

- Main + Kernels: Listing 4

I considered a range of $K = [2, 16]$ split into two different cases: (1) K is even and (2) K is uneven. My findings apply similarly to both cases and the division is merely there to structure the visualizations better. I will discuss them in detail during the (1) K is even section and will only list my plots for the second case (2) K is uneven for validation.

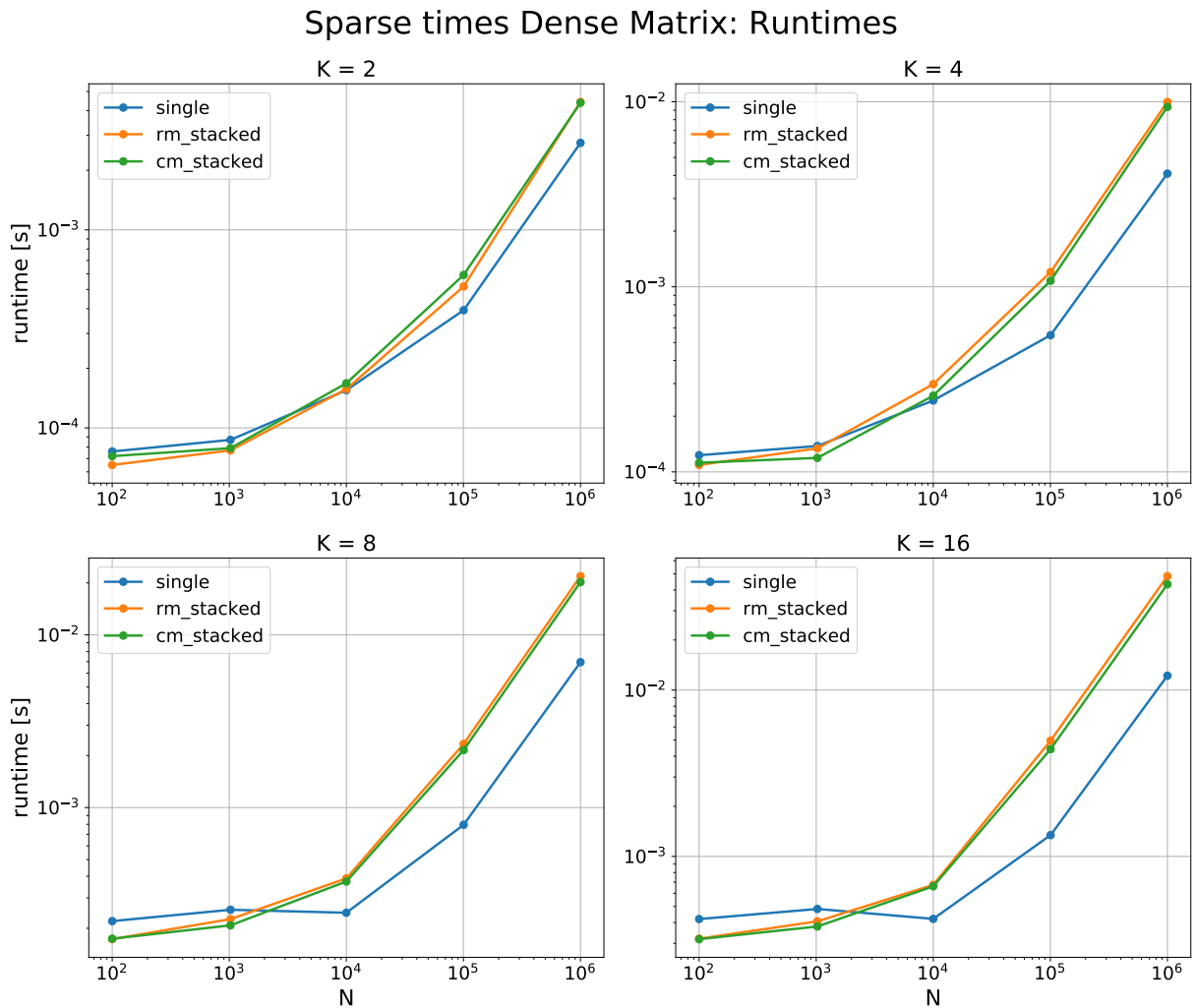


Fig. 3. Runtimes of the different vector dot product based kernels.

2.1 K is even

Surprisingly, the stacked versions (where the vectors x are given as a matrix) of the kernel actually performed worse. The actual runtimes are graphed in Fig. 3, but I think the speedup plots illustrate the behaviour better. For later reference, I used the `generate_fdm_laplace(...)` function, that you provided for the CG exercises, to generate my sparse matrix.

In Fig. 4, the speedup of the two matrix-times-matrix kernels are plotted in reference to the runtime of K calls of the reference matrix-times-vector kernel.

A speedup of :

- $S < 1$ means slower than the reference
- $S = 1$ means equally as fast as the reference
- $S > 1$ means faster than the reference.

As one can see, the performance for small N of the stacked variants start out similar for all K - with a slight speed up. Here, the main factor is the reduced number of kernel calls compared to the K calls of the matrix-vector product. As N increases, the speedup reduces and crosses the $S = 1$ line around $N = 10^4$. The speedup decreases further afterwards based on the number of vectors K - bigger K equals worse performance (lower speedup). The column major stored version performs slightly better than the row-major version.

I was honestly expecting better results - an actual speedup compared to the K calls of the simple matrix-vector product. I believe though, that the structure (and storage method) of the sparse matrix plays a significant influence. The sparse matrix I used was, in essence, a tridiagonal one (LaPlace problem). The results might be different for unstructured matrices or skyline matrices, etc. The CSR format orders non zero entries by row, so it makes sense that the column major stored version of the stacked kernel performs slightly better than the row-major version ("column" of vector times row of matrix). There might also be a better way to nest the for-loops here (which also depends on the structure of the sparse matrix), but I did not figure it out.

Sparse times Dense Matrix: Speedup compared to single

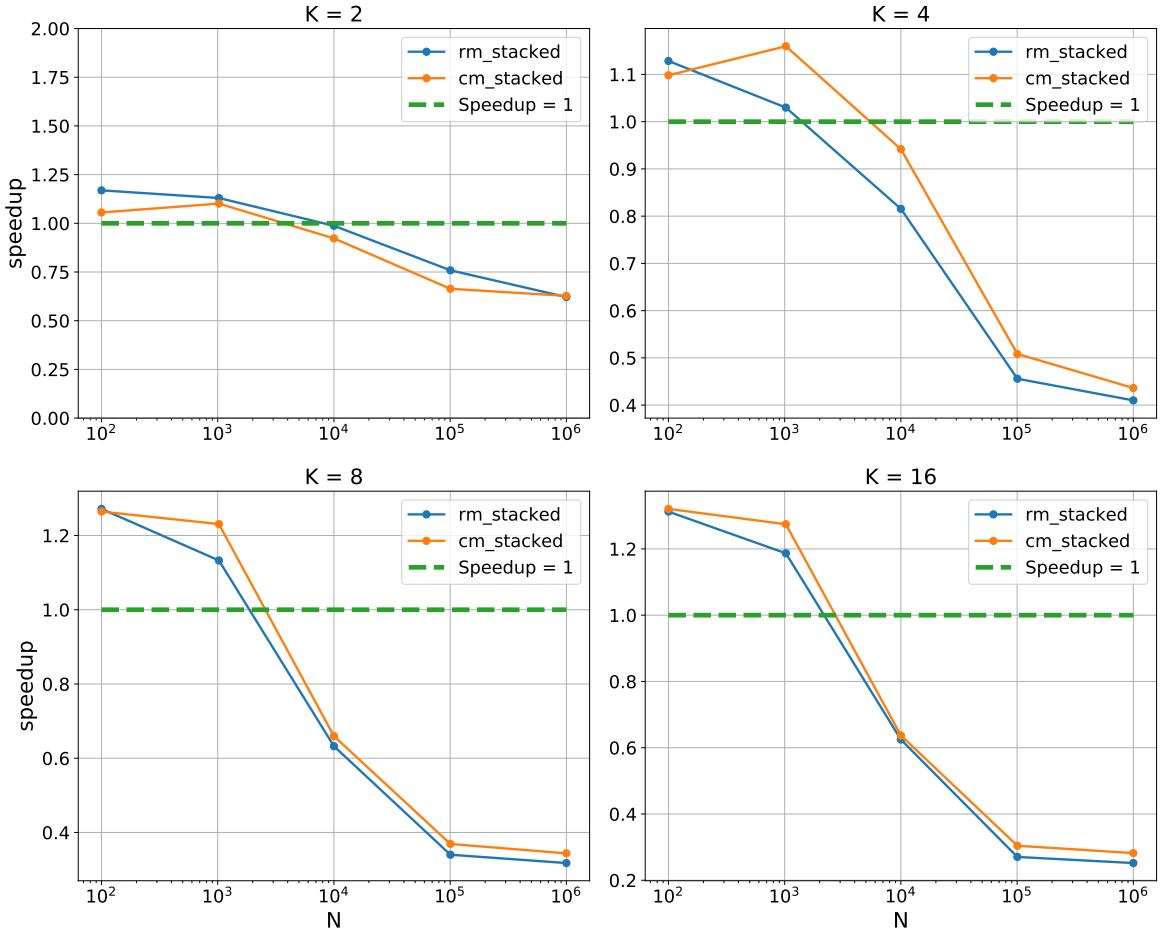


Fig. 4. Runtimes of the different vector dot product based kernels.

2.2 K is uneven

My findings for this case are the same as for the even K case. I'm listing similar plots in the following for reference and so you can validate that I formed my conclusions based on both cases.

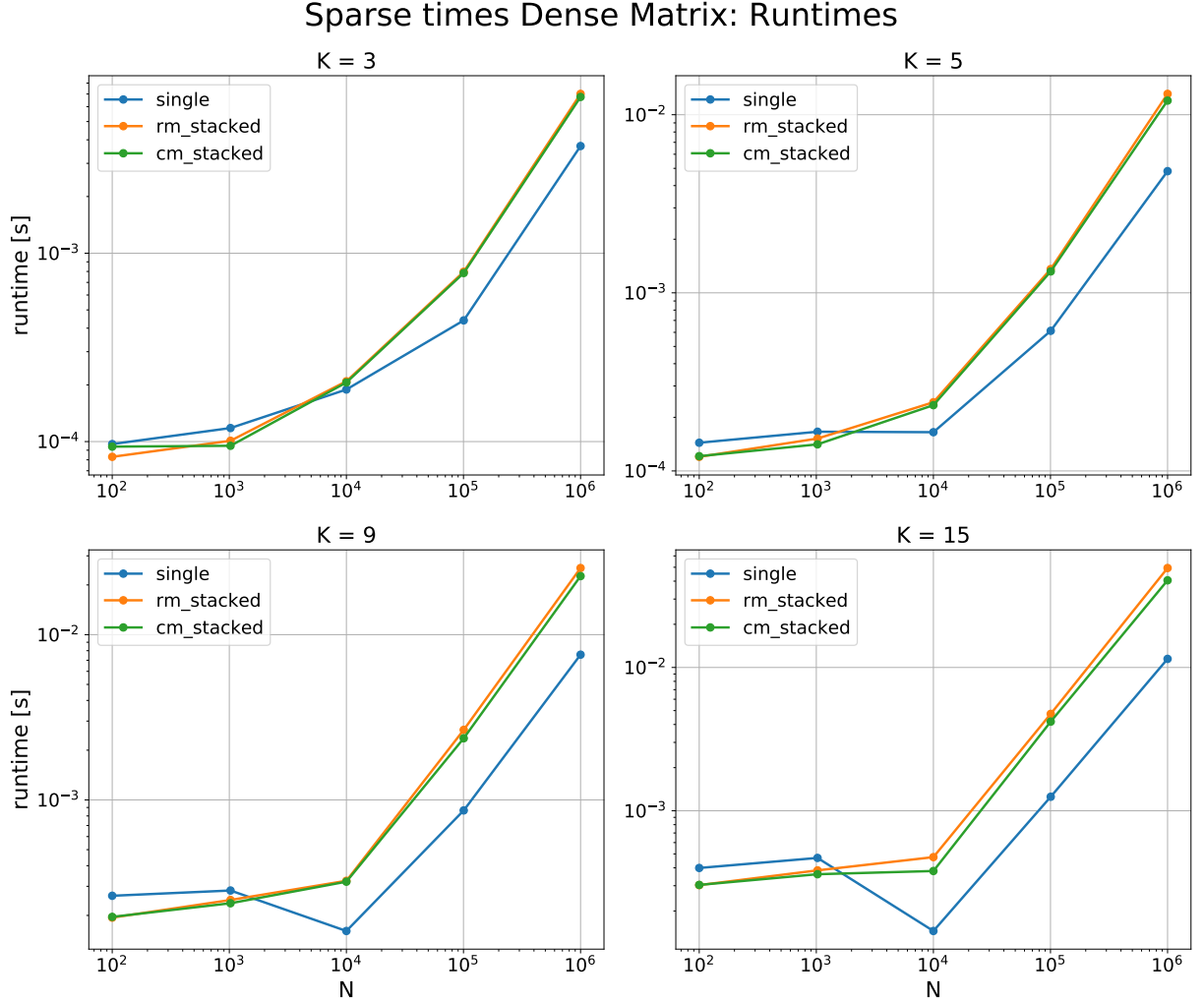


Fig. 5. Runtimes of the different vector dot product based kernels.

Sparse times Dense Matrix: Speedup compared to single

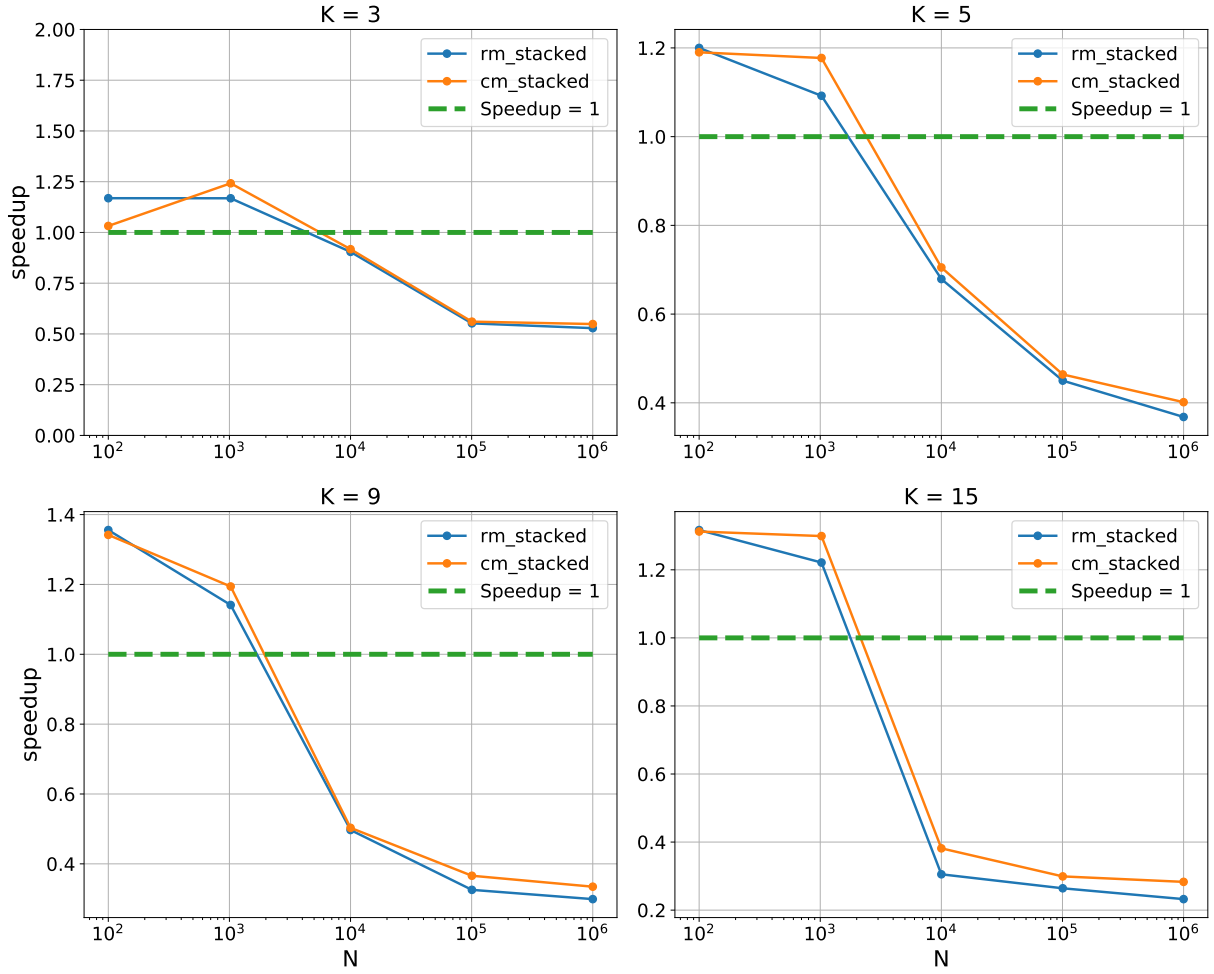


Fig. 6. Runtimes of the different vector dot product based kernels.

Task	kernel name (Code Line)
Counts and stores the number of nonzero entries for each row	nz_for_this_row (Line 239)
Form the row offset array of the CSR format	in main (Line 474 - 476)
Write the column indices + the nonzero matrix values to the CSR arrays	assembleA (Line 255)

Table 1: Kernels for this task and where to find them in the code.

3 Code and Kernels

Listings

1	atomicMax(..) implementation using atomicCAS.	1
2	Ex6.1: Dot products with Warp shuffles - Kernels	9
3	Ex6.1: Dot products with Warp shuffles	12
4	Ex6.2: Sparse Matrix times Dense Matrix	20

Listing 2: Ex6.1: Dot products with Warp shuffles - Kernels

```

1  // ----- KERNELS -----
2
3  /** atomicMax for double
4   *
5   * References:
6   * (1) https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicmax
7   * (2) https://stackoverflow.com/questions/17399119/cant-we-use-atomic-operations-for-floating-point-variables-in-cuda
8   */
9  __device__ void atomicMax(double* address, double val){
10     unsigned long long int* address_as_ull = (unsigned long long int*) address;
11     unsigned long long int old = *address_as_ull, assumed;
12     do {
13         assumed = old;
14         old = atomicCAS(address_as_ull, assumed, __double_as_longlong(fmax(val,
15             __longlong_as_double(assumed))));
16     } while (assumed != old);
17 }
18
19 /** atomicMin for double
20  */
21 __device__ void atomicMin(double* address, double val){
22     unsigned long long int* address_as_ull = (unsigned long long int*) address;
23     unsigned long long int old = *address_as_ull, assumed;
24     do {
25         assumed = old;
26         old = atomicCAS(address_as_ull, assumed, __double_as_longlong(fmin(val,
27             __longlong_as_double(assumed))));
28     } while (assumed != old);
29 }
30
31 /** scalar = x DOT y
32  */
33 __global__ void xDOTy(const int N, double *x, double *y, double *scalar) {
34     int tid = threadIdx.x + blockDim.x * blockIdx.x;
35     const int stride = blockDim.x * gridDim.x;
36
37     __shared__ double cache[BLOCK_SIZE];
38
39     double tid_sum = 0.0;
40     for (; tid < N; tid += stride) {
41         double tmp_x = x[tid];
42         tid_sum += tmp_x * y[tid];
43     }
44     tid = threadIdx.x;
45     cache[tid] = tid_sum;
46
47     __syncthreads();
48     for (int i = blockDim.x / 2; i != 0; i /= 2) {
49         __syncthreads();
50         if (tid < i) // lower half does smth, rest idles
51             cache[tid] += cache[tid + i]; // lower looks up by stride and sums up
52     }
53
54     if (tid == 0) // cache[0] now contains block_sum
55     {
56         atomicAdd(scalar, cache[0]);
57     }
58 }
59
60 /** scalar = x DOT y
61  */
62 __global__ void xDOTy_warp(const int N, double *x, double *y, double *scalar) {
63     int tid = threadIdx.x + blockDim.x * blockIdx.x;
64     const int stride = blockDim.x * gridDim.x;
65
66     double sum = 0.0;
67     for (; tid < N; tid += stride) {
68         sum += x[tid] * y[tid];
69     }
70     atomicAdd(scalar, sum);
71 }

```

```

68     }
69     tid = threadIdx.x;
70
71     __syncthreads();
72     for (int i = warpSize / 2; i != 0; i /= 2) {
73         sum += __shfl_down_sync(0xffffffff, sum, i);
74     }
75
76     if (tid % warpSize == 0) // a block can consist of multiple warps
77     {
78         atomicAdd(scalar, sum);
79     }
80 }
81
82 /** analyze_x_shared
83  *
84  * result[0] = sum;
85  * result[1] = abs_sum;
86  * result[2] = sqr_sum;
87  * result[3] = mod_max;
88  * result[4] = min;
89  * result[5] = max;
90  * result[6] = z_entries;
91  */
92 // template <int block_size=BLOCK_SIZE>
93 __global__ void analyze_x_shared(const int N, double *x, double *results) {
94     if (blockDim.x * blockIdx.x < N) {
95         int tid = threadIdx.x + blockDim.x * blockIdx.x; // global tid
96         const int stride = blockDim.x * gridDim.x;
97
98         __shared__ double cache[7][BLOCK_SIZE];
99
100        double sum = 0.0, abs_sum = 0.0, sqr_sum = 0.0;
101        // double mod_max = 0.0;
102        double max = x[0];
103        double min = max;
104        double z_entries = 0;
105        for (; tid < N; tid += stride) {
106            double value = x[tid];
107            sum += value;
108            abs_sum += std::abs(value);
109            sqr_sum += value*value;
110
111            // mod_max = (std::abs(value) > mod_max)? value : mod_max;
112            min = fmin(value, min);
113            max = fmax(value, max);
114            z_entries += (value)? 0 : 1;
115        }
116        tid = threadIdx.x; // block tid
117        cache[0][tid] = sum;
118        cache[1][tid] = abs_sum;
119        cache[2][tid] = sqr_sum;
120        cache[3][tid] = fmax(std::abs(min), max);
121        cache[4][tid] = min;
122        cache[5][tid] = max;
123        cache[6][tid] = z_entries;
124
125        __syncthreads();
126        for (int i = blockDim.x / 2; i != 0; i /= 2) {
127            __syncthreads();
128            if (tid < i) { // lower half does smth, rest idles
129                // sums
130                cache[0][tid] += cache[0][tid + i];
131                cache[1][tid] += cache[1][tid + i];
132                cache[2][tid] += cache[2][tid + i];
133                // min/max values
134                cache[3][tid] = fmax(cache[3][tid + i], cache[3][tid]); // already all values are
135                                std::abs(...)
136                cache[4][tid] = fmin(cache[4][tid + i], cache[4][tid]);
137                cache[5][tid] = fmax(cache[5][tid + i], cache[5][tid]);
138
139                // "sum"

```

```

139         cache[6][tid] += cache[6][tid + i];
140     }
141 }
142
143 if (tid == 0) // cache[0] now contains block_sum
144 {
145     atomicAdd(results, cache[0][0]);
146     atomicAdd(results+1, cache[1][0]);
147     atomicAdd(results+2, cache[2][0]);
148
149     // Ideally...
150     atomicMax(results+3, cache[3][0]);
151     atomicMin(results+4, cache[4][0]);
152     atomicMax(results+5, cache[5][0]);
153
154     atomicAdd(results+6, cache[6][0]);
155 }
156 }
157 }
158
159 /** analyze_x_shared
160  *
161  * result[0] = sum;
162  * result[1] = abs_sum;
163  * result[2] = sqr_sum;
164  * result[3] = mod_max;
165  * result[4] = min;
166  * result[5] = max;
167  * result[6] = z_entries;
168  */
169 __global__ void analyze_x_warp(const int N, double *x, double *results) {
170     if (blockDim.x * blockIdx.x < N) {
171         int tid = threadIdx.x + blockDim.x * blockIdx.x; // global tid
172         const int stride = blockDim.x * gridDim.x;
173
174         double sum = 0.0, abs_sum = 0.0, sqr_sum = 0.0;
175         // double mod_max = 0.0;
176         double max = x[0];
177         double min = max;
178         int z_entries = 0;
179         for (; tid < N; tid += stride) {
180             double value = x[tid];
181             sum += value;
182             abs_sum += std::abs(value);
183             sqr_sum += value*value;
184
185             min = fmin(value, min);
186             max = fmax(value, max);
187             z_entries += (value)? 0 : 1;
188         }
189         tid = threadIdx.x; // block tid
190         double mod_max = fmax(std::abs(min), max);
191
192         __syncthreads();
193         for (int i = warpSize / 2; i != 0; i /= 2) {
194             //__syncthreads();
195             sum += __shfl_down_sync(0xffffffff, sum, i);
196             abs_sum += __shfl_down_sync(0xffffffff, abs_sum, i);
197             sqr_sum += __shfl_down_sync(0xffffffff, sqr_sum, i);
198
199             double tmp = __shfl_down_sync(0xffffffff, mod_max, i);
200             mod_max = fmax(tmp, mod_max);
201             tmp = __shfl_down_sync(0xffffffff, min, i);
202             min = fmin(tmp, min);
203             tmp = __shfl_down_sync(0xffffffff, max, i);
204             max = fmax(tmp, max);
205
206             z_entries += __shfl_down_sync(0xffffffff, z_entries, i);
207         }
208         // for (int i = blockDim.x / 2; i != 0; i /= 2) {
209         // for (int i = warpSize / 2; i != 0; i /= 2) {
210         //     __syncthreads();

```

```

211     // sum += __shfl_xor_sync(-1, sum, i);
212     // abs_sum += __shfl_xor_sync(-1, abs_sum, i);
213     // sqr_sum += __shfl_xor_sync(-1, sqr_sum, i);
214
215     // double tmp = __shfl_xor_sync(-1, mod_max, i);
216     // mod_max = (tmp > mod_max) ? tmp : mod_max;
217     // tmp = __shfl_xor_sync(-1, min, i);
218     // min = (tmp < min) ? tmp : min;
219     // tmp = __shfl_xor_sync(-1, max, i);
220     // max = (tmp > max) ? tmp : max;
221
222     // z_entries += __shfl_xor_sync(-1, z_entries, i);
223     // }
224
225     if (tid % warpSize == 0) // a block can consist of multiple warps
226     {
227         atomicAdd(results, sum);
228         atomicAdd(results+1, abs_sum);
229         atomicAdd(results+2, sqr_sum);
230
231         atomicMax(results+3, mod_max);
232         atomicMin(results+4, min);
233         atomicMax(results+5, max);
234
235         atomicAdd(results+6, z_entries);
236     }
237 }
238 }

```

Listing 3: Ex6.1: Dot products with Warp shuffles

```

1  #include "timer.hpp"
2  #include <algorithm>
3  #include <numeric>
4  #include <cmath>
5  #include <cuda_blas_v2.h>
6  #include <cuda_runtime.h>
7  #include <fstream>
8  #include <iostream>
9  #include <stdio.h>
10 #include <string>
11 #include <vector>
12
13 #define BLOCK_SIZE 256
14 #define GRID_SIZE 128
15 #define TESTS 5
16 // #define SEP " ; "
17
18 // #define DEBUG
19 #ifndef DEBUG
20     #define CSV
21 #endif
22
23 template <typename T>
24 void printContainer(T container, const int size) {
25     std::cout << container[0];
26     for (int i = 1; i < size; ++i)
27         std::cout << " | " << container[i] ;
28     std::cout << std::endl;
29 }
30
31 template <typename T>
32 void printContainer(T container, const int size, const int only) {
33     std::cout << container[0];
34     for (int i = 1; i < only; ++i)
35         std::cout << " | " << container[i];
36     std::cout << " | ...";
37     for (int i = size - only; i < size; ++i)
38         std::cout << " | " << container[i];
39     std::cout << std::endl;
40 }
41

```

```

42 void printResults(double* results, std::vector<std::string> names, int size){
43     std::cout << "Results:" << std::endl;
44     for (int i = 0; i < size; ++i) {
45         std::cout << names[i] << " : " << results[i] << std::endl;
46     }
47 }
48
49 void printResults(double* results, double* ref, std::vector<std::string> names, int size){
50     std::cout << "Results (with difference to reference):" << std::endl;
51     for (int i = 0; i < size; ++i) {
52         std::cout << names[i] << " = " << results[i] << " || " << ref[i] - results[i] << std::
            endl;
53     }
54 }
55
56 double median(std::vector<double>& vec)
57 {
58     // modified taken from here: https://stackoverflow.com/questions/2114797/compute-median-of-values-stored-in-vector-c
59
60     size_t size = vec.size();
61
62     if (size == 0)
63         return 0.;
64
65     sort(vec.begin(), vec.end());
66
67     size_t mid = size/2;
68
69     return size % 2 == 0 ? (vec[mid] + vec[mid-1]) / 2 : vec[mid];
70 }
71
72 // ----- KERNELS -----
73
74 /** atomicMax for double
75  *
76  * References:
77  * (1) https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicmax
78  * (2) https://stackoverflow.com/questions/17399119/cant-we-use-atomic-operations-for-floating-point-variables-in-cuda
79  */
80 __device__ void atomicMax(double* address, double val){
81     unsigned long long int* address_as_ull = (unsigned long long int*) address;
82     unsigned long long int old = *address_as_ull, assumed;
83     do {
84         assumed = old;
85         old = atomicCAS(address_as_ull, assumed, __double_as_longlong(fmax(val,
            __longlong_as_double(assumed))));
86     } while (assumed != old);
87 }
88
89 /** atomicMin for double
90  */
91 __device__ void atomicMin(double* address, double val){
92     unsigned long long int* address_as_ull = (unsigned long long int*) address;
93     unsigned long long int old = *address_as_ull, assumed;
94     do {
95         assumed = old;
96         old = atomicCAS(address_as_ull, assumed, __double_as_longlong(fmin(val,
            __longlong_as_double(assumed))));
97     } while (assumed != old);
98 }
99
100
101 /** scalar = x DOT y
102  */
103 __global__ void xDOTy(const int N, double *x, double *y, double *scalar) {
104     int tid = threadIdx.x + blockDim.x * blockIdx.x;
105     const int stride = blockDim.x * gridDim.x;
106
107     __shared__ double cache[BLOCK_SIZE];
108

```

```

109     double tid_sum = 0.0;
110     for (; tid < N; tid += stride) {
111         double tmp_x = x[tid];
112         tid_sum += tmp_x * y[tid];
113     }
114     tid = threadIdx.x;
115     cache[tid] = tid_sum;
116
117     __syncthreads();
118     for (int i = blockDim.x / 2; i != 0; i /= 2) {
119         __syncthreads();
120         if (tid < i) // lower half does smth, rest idles
121             cache[tid] += cache[tid + i]; // lower looks up by stride and sums up
122     }
123
124     if (tid == 0) // cache[0] now contains block_sum
125     {
126         atomicAdd(scalar, cache[0]);
127     }
128 }
129
130 /** scalar = x DOT y
131 */
132 __global__ void xDOTy_warp(const int N, double *x, double *y, double *scalar) {
133     int tid = threadIdx.x + blockDim.x * blockIdx.x;
134     const int stride = blockDim.x * gridDim.x;
135
136     double sum = 0.0;
137     for (; tid < N; tid += stride) {
138         sum += x[tid] * y[tid];
139     }
140     tid = threadIdx.x;
141
142     __syncthreads();
143     for (int i = warpSize / 2; i != 0; i /= 2) {
144         sum += __shfl_down_sync(0xffffffff, sum, i);
145     }
146
147     if (tid % warpSize == 0) // a block can consist of multiple warps
148     {
149         atomicAdd(scalar, sum);
150     }
151 }
152
153 /** analyze_x_shared
154 *
155 * result[0] = sum;
156 * result[1] = abs_sum;
157 * result[2] = sqr_sum;
158 * result[3] = mod_max;
159 * result[4] = min;
160 * result[5] = max;
161 * result[6] = z_entries;
162 */
163 // template <int block_size=BLOCK_SIZE>
164 __global__ void analyze_x_shared(const int N, double *x, double *results) {
165     if (blockDim.x * blockIdx.x < N) {
166         int tid = threadIdx.x + blockDim.x * blockIdx.x; // global tid
167         const int stride = blockDim.x * gridDim.x;
168
169         __shared__ double cache[7][BLOCK_SIZE];
170
171         double sum = 0.0, abs_sum = 0.0, sqr_sum = 0.0;
172         // double mod_max = 0.0;
173         double max = x[0];
174         double min = max;
175         double z_entries = 0;
176         for (; tid < N; tid += stride) {
177             double value = x[tid];
178             sum += value;
179             abs_sum += std::abs(value);
180             sqr_sum += value*value;

```

```

181
182     // mod_max = (std::abs(value) > mod_max)? value : mod_max;
183     min = fmin(value, min);
184     max = fmax(value, max);
185     z_entries += (value)? 0 : 1;
186 }
187 tid = threadIdx.x; // block tid
188 cache[0][tid] = sum;
189 cache[1][tid] = abs_sum;
190 cache[2][tid] = sqr_sum;
191 cache[3][tid] = fmax(std::abs(min), max);
192 cache[4][tid] = min;
193 cache[5][tid] = max;
194 cache[6][tid] = z_entries;
195
196 __syncthreads();
197 for (int i = blockDim.x / 2; i != 0; i /= 2) {
198     __syncthreads();
199     if (tid < i) { // lower half does smth, rest idles
200         // sums
201         cache[0][tid] += cache[0][tid + i];
202         cache[1][tid] += cache[1][tid + i];
203         cache[2][tid] += cache[2][tid + i];
204         // min/max values
205         cache[3][tid] = fmax(cache[3][tid + i], cache[3][tid]); // already all values are
                std::abs(...)
206         cache[4][tid] = fmin(cache[4][tid + i], cache[4][tid]);
207         cache[5][tid] = fmax(cache[5][tid + i], cache[5][tid]);
208
209         // "sum"
210         cache[6][tid] += cache[6][tid + i];
211     }
212 }
213
214 if (tid == 0) // cache[0] now contains block_sum
215 {
216     atomicAdd(results, cache[0][0]);
217     atomicAdd(results+1, cache[1][0]);
218     atomicAdd(results+2, cache[2][0]);
219
220     // Ideally...
221     atomicMax(results+3, cache[3][0]);
222     atomicMin(results+4, cache[4][0]);
223     atomicMax(results+5, cache[5][0]);
224
225     atomicAdd(results+6, cache[6][0]);
226 }
227 }
228 }
229
230 /** analyze_x_shared
231  *
232  * result[0] = sum;
233  * result[1] = abs_sum;
234  * result[2] = sqr_sum;
235  * result[3] = mod_max;
236  * result[4] = min;
237  * result[5] = max;
238  * result[6] = z_entries;
239  */
240 __global__ void analyze_x_warp(const int N, double *x, double *results) {
241     if (blockDim.x * blockIdx.x < N) {
242         int tid = threadIdx.x + blockDim.x * blockIdx.x; // global tid
243         const int stride = blockDim.x * gridDim.x;
244
245         double sum = 0.0, abs_sum = 0.0, sqr_sum = 0.0;
246         // double mod_max = 0.0;
247         double max = x[0];
248         double min = max;
249         int z_entries = 0;
250         for (; tid < N; tid += stride) {
251             double value = x[tid];

```



```

252     sum += value;
253     abs_sum += std::abs(value);
254     sqr_sum += value*value;
255
256     min = fmin(value, min);
257     max = fmax(value, max);
258     z_entries += (value)? 0 : 1;
259 }
260 tid = threadIdx.x; // block tid
261 double mod_max = fmax(std::abs(min), max);
262
263 __syncthreads();
264 for (int i = warpSize / 2; i != 0; i /= 2) {
265     //__syncthreads();
266     sum += __shfl_down_sync(0xffffffff, sum, i);
267     abs_sum += __shfl_down_sync(0xffffffff, abs_sum, i);
268     sqr_sum += __shfl_down_sync(0xffffffff, sqr_sum, i);
269
270     double tmp = __shfl_down_sync(0xffffffff, mod_max, i);
271     mod_max = fmax(tmp, mod_max);
272     tmp = __shfl_down_sync(0xffffffff, min, i);
273     min = fmin(tmp, min);
274     tmp = __shfl_down_sync(0xffffffff, max, i);
275     max = fmax(tmp, max);
276
277     z_entries += __shfl_down_sync(0xffffffff, z_entries, i);
278 }
279 // for (int i = blockDim.x / 2; i != 0; i /= 2) {
280 // for (int i = warpSize / 2; i != 0; i /= 2) {
281 //     __syncthreads();
282 //     sum += __shfl_xor_sync(-1, sum, i);
283 //     abs_sum += __shfl_xor_sync(-1, abs_sum, i);
284 //     sqr_sum += __shfl_xor_sync(-1, sqr_sum, i);
285
286 //     double tmp = __shfl_xor_sync(-1, mod_max, i);
287 //     mod_max = (tmp > mod_max) ? tmp : mod_max;
288 //     tmp = __shfl_xor_sync(-1, min, i);
289 //     min = (tmp < min) ? tmp : min;
290 //     tmp = __shfl_xor_sync(-1, max, i);
291 //     max = (tmp > max) ? tmp : max;
292
293 //     z_entries += __shfl_xor_sync(-1, z_entries, i);
294 // }
295
296 if (tid % warpSize == 0) // a block can consist of multiple warps
297 {
298     atomicAdd(results, sum);
299     atomicAdd(results+1, abs_sum);
300     atomicAdd(results+2, sqr_sum);
301
302     atomicMax(results+3, mod_max);
303     atomicMin(results+4, min);
304     atomicMax(results+5, max);
305
306     atomicAdd(results+6, z_entries);
307 }
308 }
309 }
310
311 template <typename T>
312 void toCSV(std::fstream& csv, T* array, int size) {
313     csv << size;
314     for (int i = 0; i < size; ++i) {
315         csv << " "; << array[i];
316     }
317     csv << std::endl;
318 }
319
320 int main(void) {
321     Timer timer;
322     std::vector<int> vec_Ns{100, 1000, 10000, 100000, 1000000, 10000000, 100000000};
323     // std::vector<int> vec_Ns{100, 1000};

```

```

324     std::vector<double> times(TESTS, 0.);
325
326 #ifdef CSV
327     std::fstream csv_times, csv_results, csv_results2, csv_results3, csv_results_ref;
328     std::string csv_times_name = "ph_data.csv";
329     std::string csv_results_name = "ph_results.csv";
330     std::string csv_results2_name = "ph_results2.csv";
331     std::string csv_results3_name = "ph_results3.csv";
332     std::string csv_results_ref_name = "ph_results_ref.csv";
333     csv_times.open(csv_times_name, std::fstream::out | std::fstream::trunc);
334     csv_results.open(csv_results_name, std::fstream::out | std::fstream::trunc);
335     csv_results2.open(csv_results2_name, std::fstream::out | std::fstream::trunc);
336     csv_results3.open(csv_results3_name, std::fstream::out | std::fstream::trunc);
337     csv_results_ref.open(csv_results_ref_name, std::fstream::out | std::fstream::trunc);
338
339     std::string header = "N;time_shared;time_warp;time_warp_adapt;time_dot;time_dot_warp;
        time_cpuref;time_cublas";
340     // to csv file
341     csv_times << header << std::endl;
342
343     std::string header_results = "N;sum;abs_sum;sqr_sum;mod_max;min;max;z_entries";
344     csv_results << header_results << std::endl;
345     csv_results2 << header_results << std::endl;
346     csv_results3 << header_results << std::endl;
347     csv_results_ref << header_results << std::endl;
348 #endif
349
350     for (int& N : vec_Ns) {
351         //
352         // Initialize CUBLAS:
353         //
354 #ifdef DEBUG
355         std::cout << "N = " << N << std::endl;
356         std::cout << "Init CUBLAS..." << std::endl;
357 #endif
358         cublasHandle_t h;
359         cublasCreate(&h);
360
361         //
362         // allocate + init host memory:
363         //
364 #ifdef DEBUG
365         std::cout << "Allocating host arrays..." << std::endl;
366 #endif
367         double *x = (double *)malloc(sizeof(double) * N);
368         double *results = (double *)malloc(sizeof(double) * 7);
369         double *results2 = (double *)malloc(sizeof(double) * 7);
370         double *results3 = (double *)malloc(sizeof(double) * 7);
371         double *results_ref = (double *)malloc(sizeof(double) * 7);
372         std::vector<std::string> names {"sum", "abs_sum", "sqr_sum", "mod_max", "min", "max", "
            zero_entries"};
373
374         std::generate_n(x, N, [n = -N/2] () mutable { return n++; });
375         std::random_shuffle(x, x+N);
376         // I'm placing some values here by hand, so that certain results can be forced
377         // --> to test: mod_max, min, max...
378         x[0] = -1.1;
379         x[N/5] = 0.;
380         x[N/3] = -(N-1);
381         x[2*N/3] = N;
382
383         std::fill(results, results+7, 0.0);
384         results[3] = x[0];
385         results[4] = x[0];
386         results[5] = x[0];
387         std::copy(results, results+7, results2);
388         std::copy(results, results+7, results3);
389         std::copy(results, results+7, results_ref);
390         timer.reset();
391         // results_ref[0] = std::accumulate(x, x+N, 0.0);
392         for (int i = 0; i < N; ++i){
393             double tmp = x[i];

```

```

394     results_ref[0] += tmp;
395     results_ref[1] += std::abs(tmp);
396     results_ref[2] += tmp*tmp;
397     results_ref[4] = fmin(tmp, results_ref[4]);
398     results_ref[5] = fmax(tmp, results_ref[5]);
399     results_ref[6] += tmp ? 0 : 1;
400 }
401 results_ref[3] = fmax(std::abs(results_ref[4]), results_ref[5]);
402 double time_cpuref = timer.get();
403 /*result[0] = sum;
404 * result[1] = abs_sum;
405 * result[2] = sqr_sum;
406 * result[3] = mod_max;
407 * result[4] = min;
408 * result[5] = max;
409 * result[6] = z_entries;*/
410
411 //
412 // allocate device memory
413 //
414 #ifdef DEBUG
415     std::cout << "Initialized results containers: " << std::endl;
416     printContainer(results, 7);
417     printContainer(results2, 7);
418     std::cout << "Allocating CUDA arrays..." << std::endl;
419 #endif
420     double *cuda_x;
421     double *cuda_results;
422     double *cuda_scalar;
423     cudaMalloc(&cuda_x, sizeof(double) * N);
424     cudaMalloc(&cuda_results, sizeof(double) * 7);
425     cudaMalloc(&cuda_scalar, sizeof(double));
426     //
427     // Copy data to GPU
428     //
429 #ifdef DEBUG
430     std::cout << "Copying data to GPU..." << std::endl;
431 #endif
432     cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice);
433
434     //
435     // Let CUBLAS do the work:
436     //
437 #ifdef DEBUG
438     std::cout << "Running dot products with CUBLAS..." << std::endl;
439 #endif
440     double *cublas = (double *)malloc(sizeof(double));
441     for (int iter = 0; iter < TESTS; ++iter) {
442         *cublas = 0.0;
443         cudaMemcpy(cuda_scalar, &cublas, sizeof(double), cudaMemcpyHostToDevice);
444         timer.reset();
445         cublasDdot(h, N, cuda_x, 1, cuda_x, 1, cublas);
446         // cublasDnrm2(h, N-1, cuda_x, 1, cuda_scalar);
447         // cudaMemcpy(&cublas, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
448         times[iter] = timer.get();
449     }
450     double time_cublas = median(times);
451 #ifdef DEBUG
452     std::cout << "cublas: " << *cublas << std::endl;
453 #endif
454 #ifdef DEBUG
455     std::cout << "Running with analyze_x_shared..." << std::endl;
456 #endif
457     for (int iter = 0; iter < TESTS; ++iter) {
458         cudaMemcpy(cuda_results, results, sizeof(double) * 7, cudaMemcpyHostToDevice);
459         timer.reset();
460         analyze_x_shared<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_x, cuda_results);
461         cudaMemcpy(results, cuda_results, sizeof(double) * 7, cudaMemcpyDeviceToHost);
462         times[iter] = timer.get();
463     }
464     double time_shared = median(times);
465

```

```

466 #ifdef DEBUG
467     std::cout << "Running analyze_x_warp<GS, BS>..." << std::endl;
468 #endif
469     for (int iter = 0; iter < TESTS; ++iter) {
470         cudaMemcpy(cuda_results, results2, sizeof(double) * 7, cudaMemcpyHostToDevice);
471         timer.reset();
472         analyze_x_warp<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_x, cuda_results);
473         cudaMemcpy(results2, cuda_results, sizeof(double) * 7, cudaMemcpyDeviceToHost);
474         times[iter] = timer.get();
475     }
476     double time_warp = median(times);
477
478 #ifdef DEBUG
479     std::cout << "Running analyze_x_warp<N/BS, BS>..." << std::endl;
480 #endif
481     for (int iter = 0; iter < TESTS; ++iter) {
482         cudaMemcpy(cuda_results, results3, sizeof(double) * 7, cudaMemcpyHostToDevice);
483         int adapt_gridsize = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
484         // N/BLOCK_SIZE could results in a gridsize smaller than 1.
485         // also,
486         timer.reset();
487         analyze_x_warp<<<adapt_gridsize, BLOCK_SIZE>>>(N, cuda_x, cuda_results);
488         cudaMemcpy(results3, cuda_results, sizeof(double) * 7, cudaMemcpyDeviceToHost);
489         times[iter] = timer.get();
490     }
491     double time_warp_adapt = median(times);
492
493 #ifdef DEBUG
494     std::cout << "Running dot product xDOTy..." << std::endl;
495 #endif
496     for (int iter = 0; iter < TESTS; ++iter) {
497         double dot = 0.0;
498         cudaMemcpy(cuda_scalar, &dot, sizeof(double), cudaMemcpyHostToDevice);
499         timer.reset();
500         xDOTy<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_x, cuda_x, cuda_scalar);
501         cudaMemcpy(&dot, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
502         times[iter] = timer.get();
503     }
504     double time_dot = median(times);
505
506 #ifdef DEBUG
507     std::cout << "Running dot product xDOTy_warp..." << std::endl;
508 #endif
509     for (int iter = 0; iter < TESTS; ++iter) {
510         double dot = 0.0;
511         cudaMemcpy(cuda_scalar, &dot, sizeof(double), cudaMemcpyHostToDevice);
512         timer.reset();
513         xDOTy_warp<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_x, cuda_x, cuda_scalar);
514         cudaMemcpy(&dot, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
515         times[iter] = timer.get();
516     }
517     double time_dot_warp = median(times);
518     //
519     // Compare results
520     //
521 #ifdef DEBUG
522     std::cout << "DEBUG output:" << std::endl;
523     std::cout << "x:" << std::endl;
524     int only = 4;
525     printContainer(x, N, only);
526
527     std::cout << ">SHARED<" << std::endl;
528     printResults(results, results_ref, names, names.size());
529
530     std::cout << ">WARP<" << std::endl;
531     printResults(results2, results_ref, names, names.size());
532
533     std::cout << "GPU shared runtime: " << time_shared << std::endl;
534     std::cout << "GPU warp runtime: " << time_warp << std::endl;
535     std::cout << "GPU warp adaptive runtime: " << time_warp_adapt << std::endl;
536     std::cout << "GPU dot runtime: " << time_dot << std::endl;
537     std::cout << "GPU dot_warp runtime: " << time_dot_warp << std::endl;

```

```

538     std::cout << "CPU ref runtime: " << time_cpuref << std::endl;
539
540     //
541     // Clean up:
542     //
543     std::cout << "Cleaning up..." << std::endl;
544     std::cout << "-----" << std::endl;
545 #endif
546
547 #ifndef CSV
548     std::string sep = ",";
549     csv_times << N << sep << time_shared << sep << time_warp << sep << time_warp_adapt <<
        sep << time_dot << sep << time_dot_warp << sep << time_cpuref << sep << time_cublas
        << std::endl;
550
551     toCSV(csv_results, results, 7);
552     toCSV(csv_results2, results2, 7);
553     toCSV(csv_results3, results3, 7);
554     toCSV(csv_results_ref, results_ref, 7);
555 #endif
556     free(x);
557     free(results);
558     free(results2);
559     free(results3);
560     free(results_ref);
561     free(cublas);
562
563     cudaFree(cuda_x);
564     cudaFree(cuda_results);
565     cudaFree(cuda_scalar);
566
567     cublasDestroy(h);
568 }
569
570 #ifndef CSV
571     csv_times.close();
572     csv_results.close();
573     csv_results2.close();
574     csv_results3.close();
575     csv_results_ref.close();
576
577     std::cout << "\nRuntimes in csv form can be found here\nhttps://gtx1080.360252.org/2020/
        ex6/" + csv_times_name << std::endl;
578     std::cout << "\nResults in csv form can be found here\nhttps://gtx1080.360252.org/2020/ex6
        /" + csv_results_name << std::endl;
579     std::cout << "\nResults in csv form can be found here\nhttps://gtx1080.360252.org/2020/ex6
        /" + csv_results2_name << std::endl;
580     std::cout << "\nResults in csv form can be found here\nhttps://gtx1080.360252.org/2020/ex6
        /" + csv_results3_name << std::endl;
581     std::cout << "\nResults in csv form can be found here\nhttps://gtx1080.360252.org/2020/ex6
        /" + csv_results_ref_name << std::endl;
582 #endif
583     return EXIT_SUCCESS;
584 }

```

Listing 4: Ex6.2: Sparse Matrix times Dense Matrix

```

1  #include "timer.hpp"
2  #include "poisson2d.hpp"
3  #include <algorithm>
4  #include <numeric>
5  #include <cmath>
6  // #include <cublas_v2.h>
7  // #include <cuda_runtime.h>
8  #include <fstream>
9  #include <iostream>
10 #include <stdio.h>
11 #include <string>
12 #include <vector>
13
14 #define BLOCK_SIZE 256
15 #define GRID_SIZE 128

```

```

16 // #define SEP ";"
17
18 // #define DEBUG
19 #ifndef DEBUG
20     #define CSV
21 #endif
22
23 // START----- CONVENIENCE FUNTIONS -----START //
24 // template <typename T>
25 // void printContainer(T* container, const int size) {
26 //     std::cout << *container;
27 //     for (int i = 1; i < size; ++i)
28 //         std::cout << " | " << *(container+i) ;
29 //     std::cout << std::endl;
30 // }
31
32 template <typename T>
33 void printContainer(T container, const int size) {
34     std::cout << container[0];
35     for (int i = 1; i < size; ++i)
36         std::cout << " | " << container[i] ;
37     std::cout << std::endl;
38 }
39
40 template <typename T>
41 void printContainer(T container, const int size, const int only) {
42     std::cout << container[0];
43     for (int i = 1; i < only; ++i)
44         std::cout << " | " << container[i];
45     std::cout << " | ...";
46     for (int i = size - only; i < size; ++i)
47         std::cout << " | " << container[i];
48     std::cout << std::endl;
49 }
50
51 template <typename T>
52 void printContainerStrided(T container, const int size, const int stride) {
53     std::cout << container[0];
54     for (int i = stride; i < size; i+=stride)
55         std::cout << " | " << container[i];
56     std::cout << std::endl;
57 }
58
59 void printResults(double* results, std::vector<std::string> names, int size){
60     std::cout << "Results:" << std::endl;
61     for (int i = 0; i < size; ++i) {
62         std::cout << names[i] << " : " << results[i] << std::endl;
63     }
64 }
65
66 void printResults(double* results, double* ref, std::vector<std::string> names, int size){
67     std::cout << "Results (with difference to reference):" << std::endl;
68     for (int i = 0; i < size; ++i) {
69         std::cout << names[i] << " = " << results[i] << " || " << ref[i] - results[i] << std::
            endl;
70     }
71 }
72
73 template <typename T>
74 void toCSV(std::fstream& csv, T* array, int size) {
75     csv << size;
76     for (int i = 0; i < size; ++i) {
77         csv << ";" << array[i];
78     }
79     csv << std::endl;
80 }
81 // END----- CONVENIENCE FUNCTIONS -----END //
82
83 //
84 // START----- KERNELS -----START //
85 //
86 // y = A * x

```

```

87 __global__ void cuda_csr_matvec_product(int N, int *csr_rowoffsets,
88 int *csr_colindices, double *csr_values,
89 double *x, double *y)
90 {
91     for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
92         double sum = 0;
93         for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
94             sum += csr_values[k] * x[csr_colindices[k]];
95         }
96         y[i] = sum;
97     }
98 }
99
100 // Y= A * X
101 __global__ void A_MatMul_Xcm(int N, int K,
102 int *csr_rowoffsets, int *csr_colindices, double *csr_values,
103 double *X, double *Y)
104 {
105     int tid = blockIdx.x * blockDim.x + threadIdx.x;
106
107     if (tid < N){
108         int row_start = csr_rowoffsets[tid];
109         int row_end = csr_rowoffsets[tid + 1];
110
111         // for (int k = 0; k < K; ++k){
112         //     double sum = 0.0;
113         //     for (int i = row_start; i < row_end; i++) {
114         //         sum += csr_values[i]* X[csr_colindices[i]*K + k];
115         //     }
116         //     Y[k + tid*K] = sum;
117         // }
118
119         for (int i = row_start; i < row_end; i++) {
120             double aij = csr_values[i];
121             int row_of_X = csr_colindices[i]*K;
122             for (int k = 0; k < K; ++k){
123                 Y[k + tid*K] += aij * X[row_of_X + k];
124             }
125         }
126     }
127 }
128
129 // Y= A * X
130 __global__ void A_MatMul_Xrm(int N, int K,
131 int *csr_rowoffsets, int *csr_colindices, double *csr_values,
132 double *X, double *Y)
133 {
134     int tid = blockIdx.x * blockDim.x + threadIdx.x;
135
136     if (tid < N){
137         int row_start = csr_rowoffsets[tid];
138         int row_end = csr_rowoffsets[tid + 1];
139
140         for (int k = 0; k < K; ++k){
141             double sum = 0.0;
142             for (int i = row_start; i < row_end; i++) {
143                 sum += csr_values[i]* X[csr_colindices[i] + k*N];
144             }
145             Y[k + tid*K] = sum;
146         }
147     }
148 }
149
150 // Y= A * X
151 __global__ void A_MatMul_Xrm_boost(int N, int K,
152 int *csr_rowoffsets, int *csr_colindices, double *csr_values,
153 double *X, double *Y)
154 {
155     int tid = blockIdx.x * blockDim.x + threadIdx.x;
156
157     if (tid < N){
158         int row_start = csr_rowoffsets[tid];

```

```

159     int row_end = csr_rowoffsets[tid + 1];
160
161     for (int k = 0; k < K; ++k){
162         double sum = 0.0;
163         for (int i = row_start; i < row_end; i++) {
164             sum += csr_values[i]* X[csr_colindices[i] + k*N];
165         }
166         Y[k + tid*K] = sum;
167     }
168 }
169 }
170 //
171 // END----- KERNELS -----END //
172 //
173
174 /**TO-DO:
175  * Adapt signature
176  *
177  */
178 // void assemble_csr_on_gpu(){
179 //     // Perform the calculations
180 //     int numberOfValues;
181 //     timer.reset();
182 //     count_nnz<<<GRID_SIZE, BLOCK_SIZE>>>(cuda_row_offsets_2, N, M);
183 //     exclusive_scan(cuda_row_offsets_2, cuda_row_offsets, N*M+1);
184 //     cudaMemcpy(row_offsets, cuda_row_offsets, sizeof(int) * (N*M+1),
185 //         cudaMemcpyDeviceToHost);
186 //     numberOfValues = row_offsets[N*M];
187
188 //     #ifdef DEBUG
189 //         printContainer(row_offsets, N*M+1, 4);
190 //         std::cout << std::endl;
191 //     #endif
192
193 //     double *values = (double *)malloc(sizeof(double) * numberOfValues);
194 //     int *columns = (int *)malloc(sizeof(int) * numberOfValues);
195 //     cudaMalloc(&cuda_columns, sizeof(int) * numberOfValues);
196 //     cudaMalloc(&cuda_values, sizeof(double) * numberOfValues);
197 //     assembleA<<<GRID_SIZE, BLOCK_SIZE>>>(cuda_values, cuda_columns, cuda_row_offsets, N,
198 //         M);
199 //     double time_assemble_gpu = timer.get();
200 //     cudaMemcpy(values, cuda_values, sizeof(double) * numberOfValues,
201 //         cudaMemcpyDeviceToHost);
202 //     cudaMemcpy(columns, cuda_columns, sizeof(int) * numberOfValues,
203 //         cudaMemcpyDeviceToHost);
204 // }
205
206 int main(void) {
207     Timer timer;
208     std::vector<int> vec_Ns{100, 10000, 1000000};
209     // std::vector<int> vec_Ks{2, 4, 8, 16};
210     // std::vector<int> vec_Ns{1000, 100000};
211     std::vector<int> vec_Ks{3, 5, 9, 15};
212     // std::vector<int> vec_Ns{1000000};
213
214 #ifdef CSV
215     std::fstream csv_times;
216     std::string csv_times_name = "ph_data.csv";
217     csv_times.open(csv_times_name, std::fstream::out | std::fstream::trunc);
218
219     std::string header = "N;K;time_single;time_rm_stacked;time_cm_stacked";
220     // to csv file
221     csv_times << header << std::endl;
222 #endif
223
224     for (int& N : vec_Ns) {
225         for (int& K : vec_Ks) {
226             // cublasHandle_t h;
227             // cublasCreate(&h);

```



```

227     //
228     // allocate + init host memory:
229     //
230 #ifdef DEBUG
231     std::cout << "Allocating host + device arrays..." << std::endl;
232 #endif
233     // "Vectors"
234     double* X = (double *)malloc(sizeof(double) * N * K);
235     double* Y = (double *)malloc(sizeof(double) * N * K);
236     double* Y2 = (double *)malloc(sizeof(double) * N * K);
237     // double* x = (double *)malloc(sizeof(double) * N);
238     double* y = (double *)malloc(sizeof(double) * N);
239     std::fill(X, X + (N*K), 1.);
240     std::fill(Y, Y + (N*K), 0.);
241     std::fill(Y2, Y2 + (N*K), 0.);
242     // std::fill(x, x + N, 1.);
243
244     double *cuda_X;
245     double *cuda_Y;
246     // double *cuda_Y2;
247     // double *cuda_x;
248     double *cuda_y;
249     cudaMalloc(&cuda_X, sizeof(double) * N*K);
250     cudaMalloc(&cuda_Y, sizeof(double) * N*K);
251     // cudaMalloc(&cuda_Y2, sizeof(double) * N*K);
252     // cudaMalloc(&cuda_x, sizeof(double) * N);
253     cudaMalloc(&cuda_y, sizeof(double) * N);
254
255     // Matrix
256     int* csr_rowoffsets = (int*)malloc(sizeof(int) * (N+1));
257     int* csr_colindices = (int*)malloc(sizeof(int) * 5*N);
258     double* csr_values = (double*)malloc(sizeof(double) * 5*N);
259
260     int* cuda_csr_rowoffsets;
261     int* cuda_csr_colindices;
262     double* cuda_csr_values;
263     cudaMalloc(&cuda_csr_rowoffsets, sizeof(int) * (N+1));
264     cudaMalloc(&cuda_csr_colindices, sizeof(int) * 5*N);
265     cudaMalloc(&cuda_csr_values, sizeof(double) * 5*N);
266     //
267     // Copy data to GPU
268     //
269 #ifdef DEBUG
270     std::cout << "Copying data to GPU..." << std::endl;
271 #endif
272     cudaMemcpy(cuda_X, X, sizeof(double) * N*K, cudaMemcpyHostToDevice);
273     cudaMemcpy(cuda_Y, Y, sizeof(double) * N*K, cudaMemcpyHostToDevice);
274     // cudaMemcpy(cuda_Y2, Y2, sizeof(double) * N*K, cudaMemcpyHostToDevice);
275     // cudaMemcpy(cuda_x, X, sizeof(double) * N, cudaMemcpyHostToDevice);
276     // cudaMemcpy(cuda_y, y, sizeof(double) * N*K, cudaMemcpyHostToDevice);
277
278     // Assemble A
279 #ifdef DEBUG
280     std::cout << "Generating A..." << std::endl;
281 #endif
282     generate_fdm_laplace(sqrt(N), csr_rowoffsets, csr_colindices, csr_values);
283 #ifdef DEBUG
284     std::cout << "Generating A done!" << std::endl;
285 #endif
286     cudaMemcpy(cuda_csr_rowoffsets, csr_rowoffsets, sizeof(int) * (N+1),
                cudaMemcpyHostToDevice);
287     cudaMemcpy(cuda_csr_colindices, csr_colindices, sizeof(int) * 5*N,
                cudaMemcpyHostToDevice);
288     cudaMemcpy(cuda_csr_values, csr_values, sizeof(double) * 5*N, cudaMemcpyHostToDevice);
289
290     // ----- TEST ----- //
291
292 #ifdef DEBUG
293     std::cout << "N = " << N << std::endl;
294     std::cout << "K = " << K << std::endl;
295
296     std::cout << "Running per vector product kernel K times..." << std::endl;

```

```

297 #endif
298     timer.reset();
299     for (int k = 0; k < K; ++k)
300         cuda_csr_matvec_product<<<GRID_SIZE, BLOCK_SIZE>>>(
301             N,
302             cuda_csr_rowoffsets, cuda_csr_colindices, cuda_csr_values,
303             cuda_X, cuda_y);
304     cudaMemcpy(y, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);
305     double time_single = timer.get();
306
307 #ifdef DEBUG
308     std::cout << "Running RowMajor stacked kernel..." << std::endl;
309 #endif
310     timer.reset();
311     A_MatMul_Xrm<<<GRID_SIZE, BLOCK_SIZE>>>(
312         N, K,
313         cuda_csr_rowoffsets, cuda_csr_colindices, cuda_csr_values,
314         cuda_X, cuda_Y);
315     cudaMemcpy(Y, cuda_Y, sizeof(double) * N*K, cudaMemcpyDeviceToHost);
316     double time_rm_stacked = timer.get();
317
318 #ifdef DEBUG
319     std::cout << "Running ColumnMajor stacked kernel..." << std::endl;
320 #endif
321     cudaMemcpy(cuda_Y, Y2, sizeof(double) * N*K, cudaMemcpyHostToDevice);
322     timer.reset();
323     A_MatMul_Xcm<<<GRID_SIZE, BLOCK_SIZE>>>(
324         N, K,
325         cuda_csr_rowoffsets, cuda_csr_colindices, cuda_csr_values,
326         cuda_X, cuda_Y);
327     cudaMemcpy(Y2, cuda_Y, sizeof(double) * N*K, cudaMemcpyDeviceToHost);
328     double time_cm_stacked = timer.get();
329
330
331
332
333     //
334     // Compare results
335     //
336 #ifdef DEBUG
337     std::cout << "DEBUG output:" << std::endl;
338     // int only = 4;
339     std::cout << "A (non zero entries by row)" << std::endl;
340     // int csr_values_size = csr_rowoffsets[N+1];
341     // printContainer(y, N);
342     std::cout << "Row" << std::endl;
343     int max_output = 10;
344     for (int row = 0; row < min(N, max_output); row++){
345         std::cout << row << ": ";
346         printContainer(csr_values+ csr_rowoffsets[row], min(csr_rowoffsets[row+1]-
            csr_rowoffsets[row], max_output));
347     }
348
349     std::cout << "y:" << std::endl;
350     printContainer(y, min(N, max_output));
351     std::cout << "Y_rm:" << std::endl;
352     printContainerStrided(Y, min(N, max_output)*K, K);
353     std::cout << "Y_cm:" << std::endl;
354     printContainerStrided(Y2, min(N, max_output)*K, K);
355
356
357     std::cout << "Single runtime: " << time_single << std::endl;
358     std::cout << "RM Stacked runtime: " << time_rm_stacked << std::endl;
359     std::cout << "CM Stacked runtime: " << time_cm_stacked << std::endl;
360
361     //
362     // Clean up:
363     //
364     std::cout << "-----" << std::endl;
365     std::cout << "Cleaning up..." << std::endl;
366 #endif
367

```

```

368     #ifdef CSV
369         std::string sep = ";";
370         csv_times << N << sep
371             << K << sep
372             << time_single << sep
373             << time_rm_stacked << sep
374             << time_cm_stacked
375             << std::endl;
376     #endif
377     free(X);
378     free(Y);
379     free(Y2);
380     // free(x);
381     free(y);
382     free(csr_rowoffsets);
383     free(csr_colindices);
384     free(csr_values);
385
386     cudaFree(cuda_X);
387     cudaFree(cuda_Y);
388     // cudaFree(cuda_Y2);
389     // cudaFree(cuda_x);
390     cudaFree(cuda_y);
391     cudaFree(cuda_csr_rowoffsets);
392     cudaFree(cuda_csr_colindices);
393     cudaFree(cuda_csr_values);
394 #ifdef DEBUG
395     std::cout << "Clean up done!" << std::endl;
396 #endif
397 }
398 }
399
400 #ifdef CSV
401     csv_times.close();
402
403     std::cout << "\nRuntimes in csv form can be found here\nhttps://gtx1080.360252.org/2020/
404         ex6/" + csv_times_name << std::endl;
405 #endif
406     return EXIT_SUCCESS;
407 }

```