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

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.

```
__device__ void atomicMax(double* address, double val){
  unsigned long long int* address_as_ull = (unsigned long long int*) address;
  unsigned long long int old = *address_as_ull, assumed;
  do {
    assumed = old;
    old = atomicCAS(address_as_ull, assumed, __double_as_longlong(fmax(val,
        __longlong_as_double(assumed))));
  } 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.

[1] I 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 verions are faster than the 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.

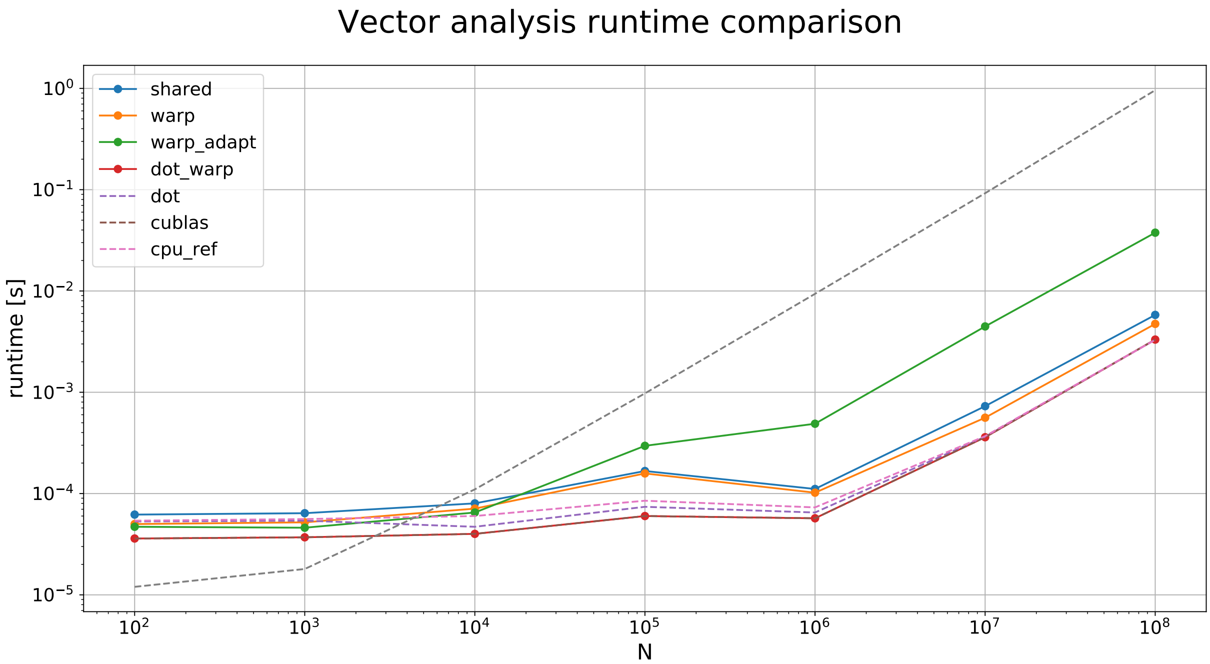

Fig. 1. Runtimes of the different vector dot product based kernels.

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 referencs 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 negligeable. 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.

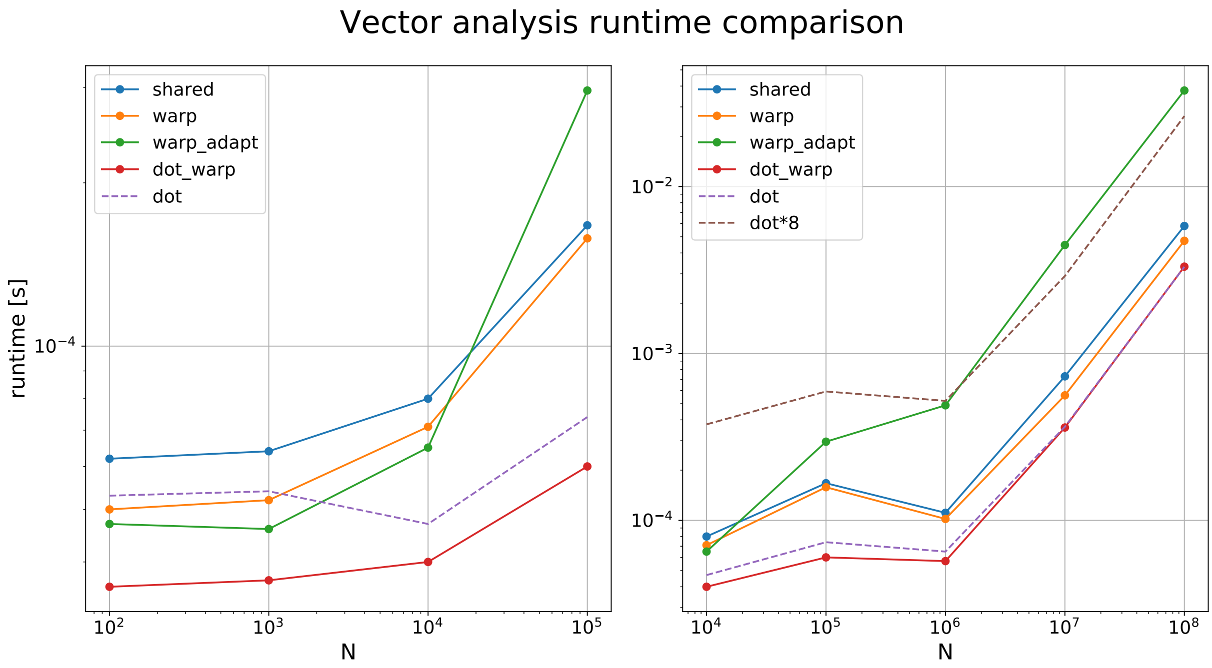

Fig. 2. Runtimes of the different vector dot product based kernels.

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.

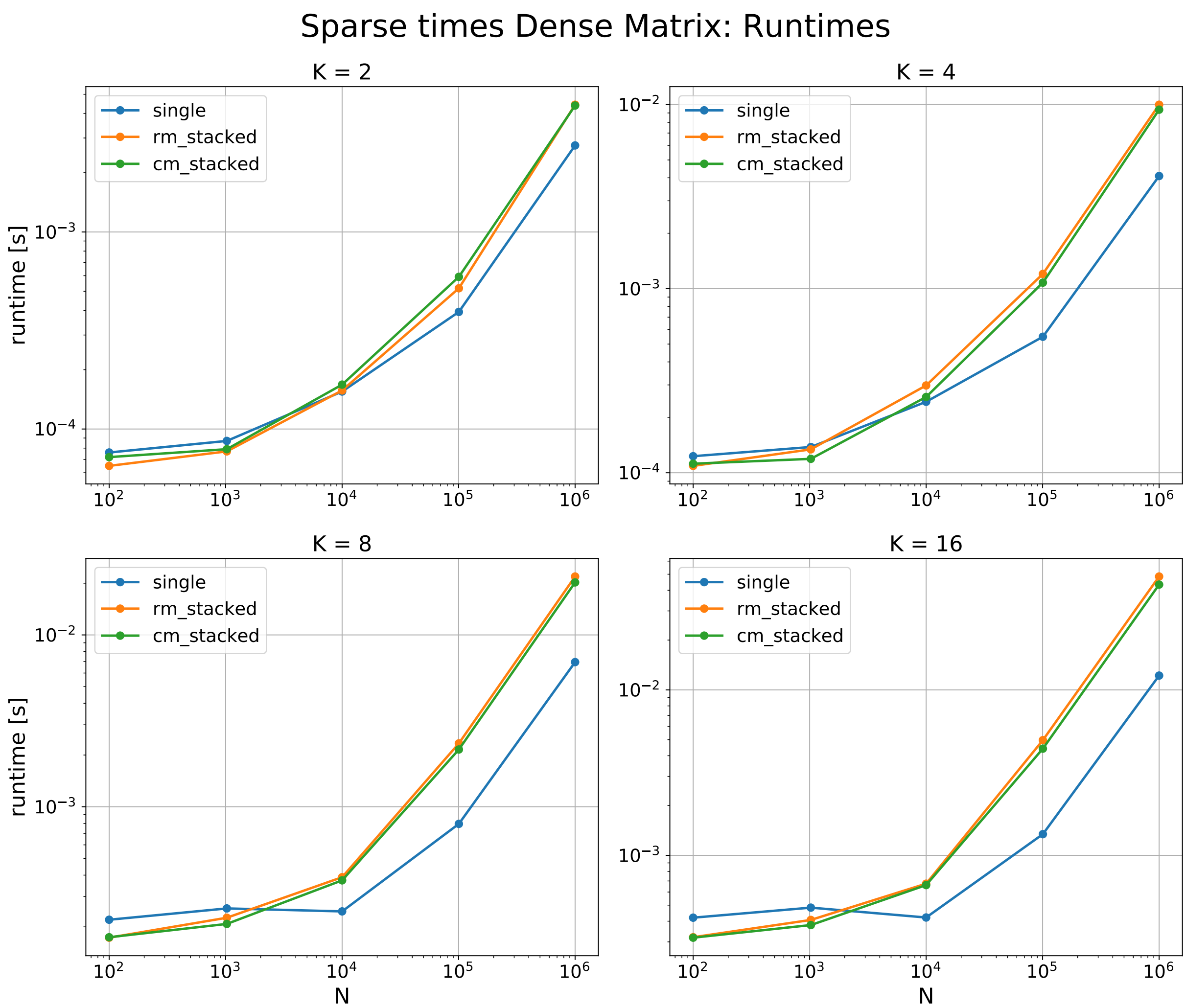

Fig. 3. Runtimes of the different vector dot product based kernels.

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.

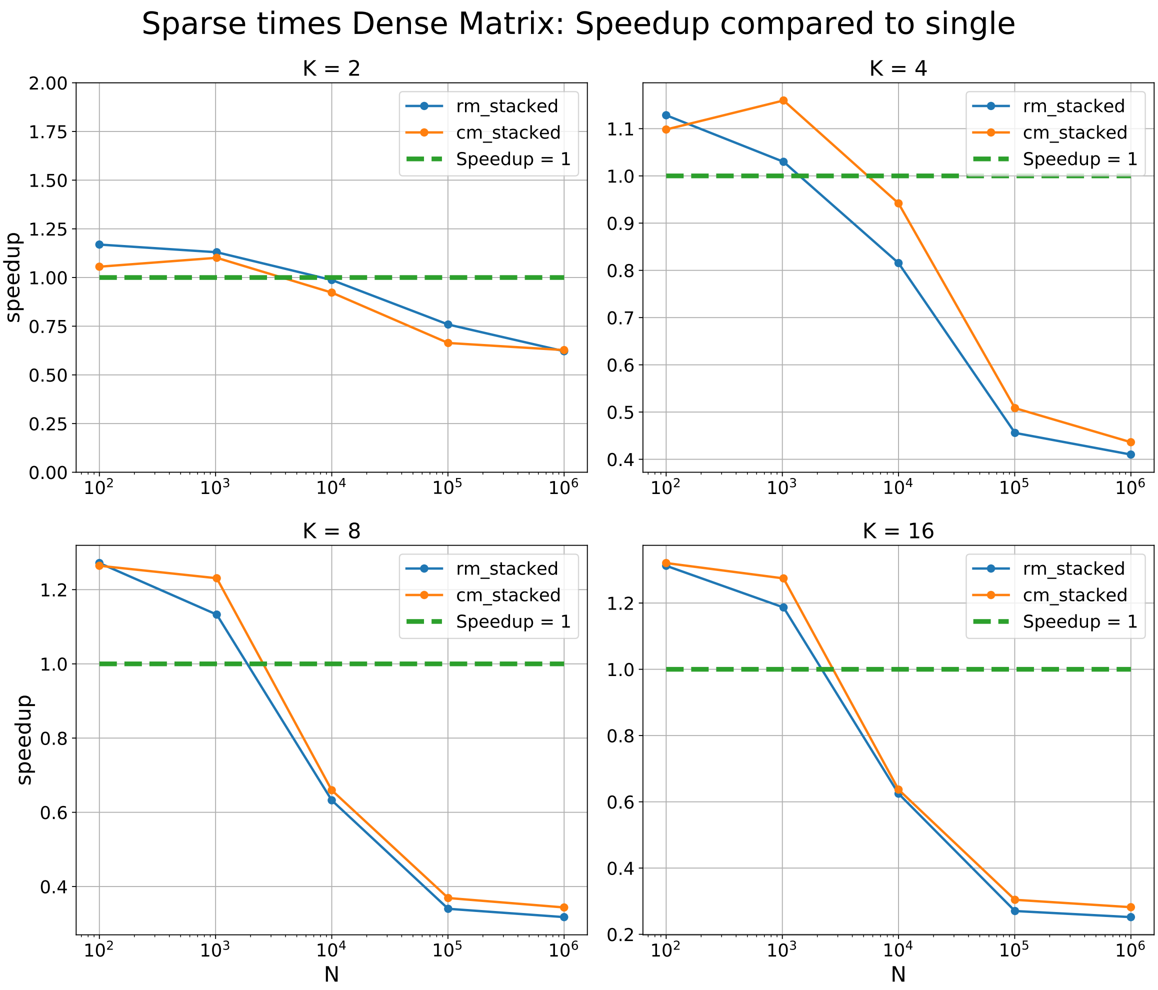

Fig. 4. Runtimes of the different vector dot product based kernels.

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.

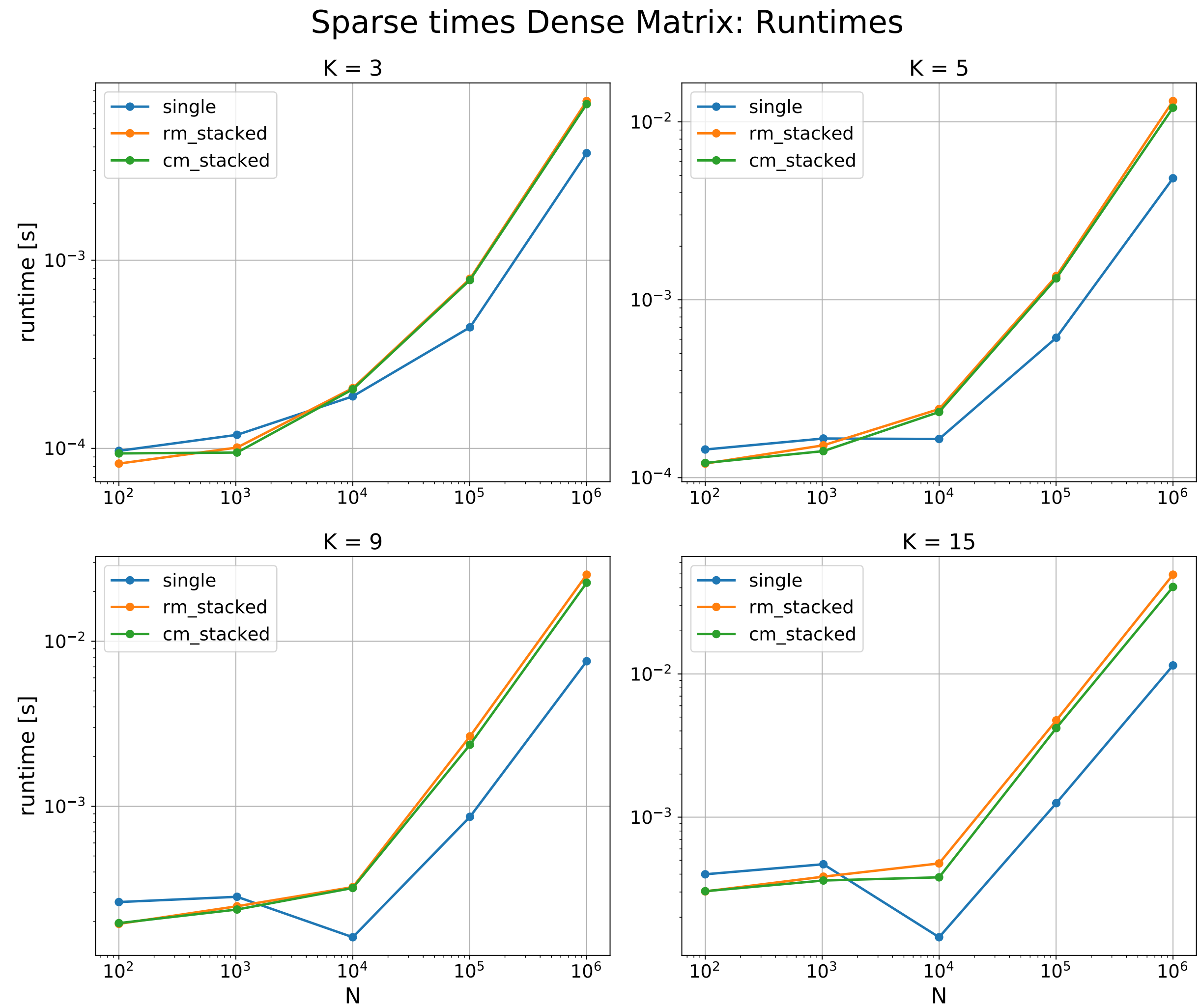

Fig. 5. Runtimes of the different vector dot product based kernels.

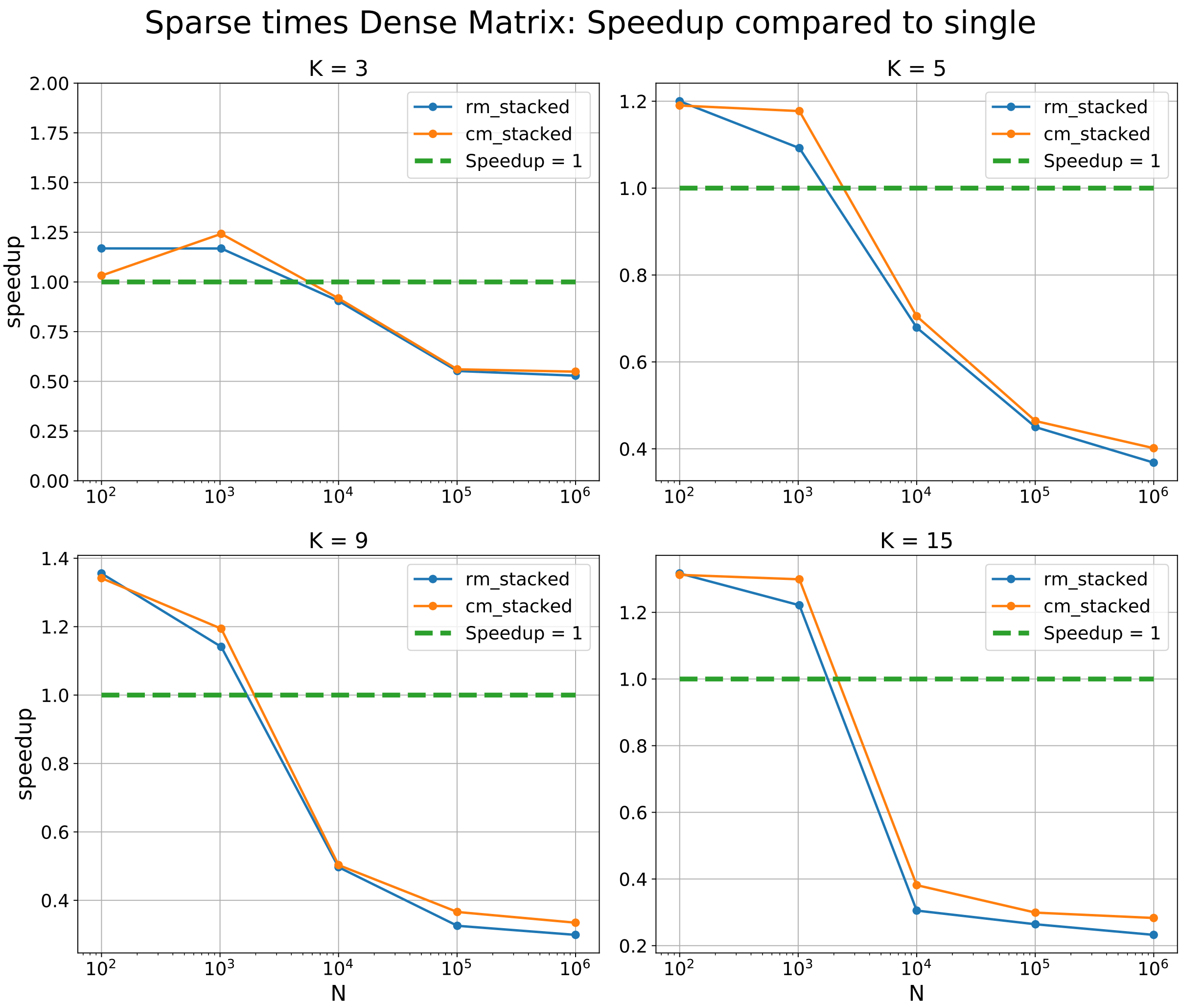

Fig. 6. Runtimes of the different vector dot product based kernels.

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

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

```
// ------------------ KERNELS ---------------

/** atomicMax for double
 *
 * References:
 * (1) https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicmax
 * (2) https://stackoverflow.com/questions/17399119/cant-we-use-atomic-operations-for-floating-point-variables-in-cuda
 */
 __device__ void atomicMax(double* address, double val){
  unsigned long long int* address_as_ull = (unsigned long long int*) address;
  unsigned long long int old = *address_as_ull, assumed;
  do  {
    assumed = old;
    old = atomicCAS(address_as_ull, assumed, __double_as_longlong(fmax(val, __longlong_as_double(assumed))));
  } while (assumed != old);
}

/** atomicMin for double
 */
__device__ void atomicMin(double* address, double val){
  unsigned long long int* address_as_ull = (unsigned long long int*) address;
  unsigned long long int old = *address_as_ull, assumed;
  do  {
    assumed = old;
    old = atomicCAS(address_as_ull, assumed, __double_as_longlong(fmin(val, __longlong_as_double(assumed))));
  } while (assumed != old);
}


/** scalar = x DOT y
 */
__global__ void xDOTy(const int N, double *x, double *y, double *scalar) {
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  const int stride = blockDim.x * gridDim.x;

  __shared__ double cache[BLOCK_SIZE];

  double tid_sum = 0.0;
  for (; tid < N; tid += stride) {
    double tmp_x = x[tid];
    tid_sum += tmp_x * y[tid];
  }
  tid = threadIdx.x;
  cache[tid] = tid_sum;

  __syncthreads();
  for (int i = blockDim.x / 2; i != 0; i /= 2) {
    __syncthreads();
    if (tid < i)                      // lower half does smth, rest idles
      cache[tid] += cache[tid + i]; // lower looks up by stride and sums up
  }

  if (tid == 0) // cache[0] now contains block_sum
  {
    atomicAdd(scalar, cache[0]);
  }
}

/** scalar = x DOT y
 */
 __global__ void xDOTy_warp(const int N, double *x, double *y, double *scalar) {
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  const int stride = blockDim.x * gridDim.x;

  double sum = 0.0;
  for (; tid < N; tid += stride) {
    sum += x[tid] * y[tid];
```

```
  }
  tid = threadIdx.x;

  __syncthreads();
  for (int i = warpSize / 2; i != 0; i /= 2) {
    sum += __shfl_down_sync(0xffffffff, sum, i);
  }

  if (tid % warpSize == 0) // a block can consist of multiple warps
  {
    atomicAdd(scalar, sum);
  }
}

/** analyze_x_shared
 *
 * result[0] = sum;
 * result[1] = abs_sum;
 * result[2] = sqr_sum;
 * result[3] = mod_max;
 * result[4] = min;
 * result[5] = max;
 * result[6] = z_entries;
 */
// template <int block_size=BLOCK_SIZE>
__global__ void analyze_x_shared(const int N, double *x, double *results) {
  if (blockDim.x * blockIdx.x < N) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x; // global tid
    const int stride = blockDim.x * gridDim.x;

    __shared__ double cache[7][BLOCK_SIZE];

    double sum = 0.0, abs_sum = 0.0, sqr_sum = 0.0;
    // double mod_max = 0.0;
    double max = x[0];
    double min = max;
    double z_entries = 0;
    for (; tid < N; tid += stride) {
      double value = x[tid];
      sum += value;
      abs_sum += std::abs(value);
      sqr_sum += value*value;

      // mod_max = (std::abs(value) > mod_max)? value : mod_max;
      min = fmin(value, min);
      max = fmax(value, max);
      z_entries += (value)? 0 : 1;
    }
    tid = threadIdx.x; // block tid
    cache[0][tid] = sum;
    cache[1][tid] = abs_sum;
    cache[2][tid] = sqr_sum;
    cache[3][tid] = fmax(std::abs(min), max);
    cache[4][tid] = min;
    cache[5][tid] = max;
    cache[6][tid] = z_entries;

    __syncthreads();
    for (int i = blockDim.x / 2; i != 0; i /= 2) {
      __syncthreads();
      if (tid < i) { // lower half does smth, rest idles
        // sums
        cache[0][tid] += cache[0][tid + i];
        cache[1][tid] += cache[1][tid + i];
        cache[2][tid] += cache[2][tid + i];
        // min/max values
        cache[3][tid] = fmax(cache[3][tid + i], cache[3][tid]); // already all values are
            std::abs(...)
        cache[4][tid] = fmin(cache[4][tid + i], cache[4][tid]);
        cache[5][tid] = fmax(cache[5][tid + i], cache[5][tid]);

        // "sum"
```

```
        cache[6][tid] += cache[6][tid + i];
      }
    }

    if (tid == 0) // cache[0] now contains block_sum
    {
      atomicAdd(results, cache[0][0]);
      atomicAdd(results+1, cache[1][0]);
      atomicAdd(results+2, cache[2][0]);

      // Ideally...
      atomicMax(results+3, cache[3][0]);
      atomicMin(results+4, cache[4][0]);
      atomicMax(results+5, cache[5][0]);

      atomicAdd(results+6, cache[6][0]);
    }
  }
}

/** analyze_x_shared
 *
 * result[0] = sum;
 * result[1] = abs_sum;
 * result[2] = sqr_sum;
 * result[3] = mod_max;
 * result[4] = min;
 * result[5] = max;
 * result[6] = z_entries;
 */
__global__ void analyze_x_warp(const int N, double *x, double *results) {
  if (blockDim.x * blockIdx.x < N) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x; // global tid
    const int stride = blockDim.x * gridDim.x;

    double sum = 0.0, abs_sum = 0.0, sqr_sum = 0.0;
    // double mod_max = 0.0;
    double max = x[0];
    double min = max;
    int z_entries = 0;
    for (; tid < N; tid += stride) {
      double value = x[tid];
      sum += value;
      abs_sum += std::abs(value);
      sqr_sum += value*value;

      min = fmin(value, min);
      max = fmax(value, max);
      z_entries += (value)? 0 : 1;
    }
    tid = threadIdx.x; // block tid
    double mod_max = fmax(std::abs(min), max);

    __syncthreads();
    for (int i = warpSize / 2; i != 0; i /= 2) {
      //__syncthreads();
      sum += __shfl_down_sync(0xffffffff, sum, i);
      abs_sum += __shfl_down_sync(0xffffffff, abs_sum, i);
      sqr_sum += __shfl_down_sync(0xffffffff, sqr_sum, i);

      double tmp = __shfl_down_sync(0xffffffff, mod_max, i);
      mod_max = fmax(tmp, mod_max);
      tmp = __shfl_down_sync(0xffffffff, min, i);
      min = fmin(tmp, min);
      tmp = __shfl_down_sync(0xffffffff, max, i);
      max = fmax(tmp, max) ;

      z_entries += __shfl_down_sync(0xffffffff, z_entries, i);
    }
    // for (int i = blockDim.x / 2; i != 0; i /= 2) {
    // for (int i = warpSize / 2; i != 0; i /= 2) {
    //   //__syncthreads();
```

```
    //   sum += __shfl_xor_sync(-1, sum, i);
    //   abs_sum += __shfl_xor_sync(-1, abs_sum, i);
    //   sqr_sum += __shfl_xor_sync(-1, sqr_sum, i);

    //   double tmp = __shfl_xor_sync(-1, mod_max, i);
    //   mod_max = (tmp > mod_max) ? tmp : mod_max;
    //   tmp = __shfl_xor_sync(-1, min, i);
    //   min = (tmp < min) ? tmp : min;
    //   tmp = __shfl_xor_sync(-1, max, i);
    //   max = (tmp > max) ? tmp : max;

    //   z_entries += __shfl_xor_sync(-1, z_entries, i);
    // }

    if (tid % warpSize == 0) // a block can consist of multiple warps
    {
      atomicAdd(results, sum);
      atomicAdd(results+1, abs_sum);
      atomicAdd(results+2, sqr_sum);

      atomicMax(results+3, mod_max);
      atomicMin(results+4, min);
      atomicMax(results+5, max);

      atomicAdd(results+6, z_entries);
    }
  }
}
```

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

```
#include "timer.hpp"
#include <algorithm>
#include <numeric>
#include <cmath>
#include <cublas_v2.h>
#include <cuda_runtime.h>
#include <fstream>
#include <iostream>
#include <stdio.h>
#include <string>
#include <vector>

#define BLOCK_SIZE 256
#define GRID_SIZE 128
#define TESTS 5
// #define SEP ";"

// #define DEBUG
#ifndef DEBUG
  #define CSV
#endif

template <typename T>
void printContainer(T container, const int size) {
  std::cout << container[0];
  for (int i = 1; i < size; ++i)
    std::cout << " | " << container[i] ;
  std::cout << std::endl;
}

template <typename T>
void printContainer(T container, const int size, const int only) {
  std::cout << container[0];
  for (int i = 1; i < only; ++i)
      std::cout  << " | " << container[i];
  std::cout << " | ...";
  for (int i = size - only; i < size; ++i)
    std::cout  << " | " << container[i];
  std::cout << std::endl;
}

```

```
void printResults(double* results, std::vector<std::string> names, int size){
  std::cout << "Results:" << std::endl;
  for (int i = 0; i < size; ++i) {
    std::cout << names[i] << " : " << results[i] << std::endl;
  }
}

void printResults(double* results, double* ref, std::vector<std::string> names, int size){
  std::cout << "Results (with difference to reference):" << std::endl;
  for (int i = 0; i < size; ++i) {
    std::cout << names[i] << " = " << results[i] << " ||  " << ref[i] - results[i] << std::endl;
  }
}

double median(std::vector<double>& vec)
{
  // modified taken from here: https://stackoverflow.com/questions/2114797/compute-median-of-values-stored-in-vector-c

  size_t size = vec.size();

  if (size == 0)
          return 0.;

  sort(vec.begin(), vec.end());

  size_t mid = size/2;

  return size % 2 == 0 ? (vec[mid] + vec[mid-1]) / 2 : vec[mid];
}

// ------------------ KERNELS ---------------

/** atomicMax for double
 *
 * References:
 * (1) https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicmax
 * (2) https://stackoverflow.com/questions/17399119/cant-we-use-atomic-operations-for-floating-point-variables-in-cuda
 */
__device__ void atomicMax(double* address, double val){
  unsigned long long int* address_as_ull = (unsigned long long int*) address;
  unsigned long long int old = *address_as_ull, assumed;
  do  {
    assumed = old;
    old = atomicCAS(address_as_ull, assumed, __double_as_longlong(fmax(val, __longlong_as_double(assumed))));
  } while (assumed != old);
}

/** atomicMin for double
 */
__device__ void atomicMin(double* address, double val){
  unsigned long long int* address_as_ull = (unsigned long long int*) address;
  unsigned long long int old = *address_as_ull, assumed;
  do  {
    assumed = old;
    old = atomicCAS(address_as_ull, assumed, __double_as_longlong(fmin(val, __longlong_as_double(assumed))));
  } while (assumed != old);
}


/** scalar = x DOT y
 */
__global__ void xDOTy(const int N, double *x, double *y, double *scalar) {
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  const int stride = blockDim.x * gridDim.x;

  __shared__ double cache[BLOCK_SIZE];

```

```
  double tid_sum = 0.0;
  for (; tid < N; tid += stride) {
    double tmp_x = x[tid];
    tid_sum += tmp_x * y[tid];
  }
  tid = threadIdx.x;
  cache[tid] = tid_sum;

  __syncthreads();
  for (int i = blockDim.x / 2; i != 0; i /= 2) {
    __syncthreads();
    if (tid < i)                      // lower half does smth, rest idles
      cache[tid] += cache[tid + i]; // lower looks up by stride and sums up
  }

  if (tid == 0) // cache[0] now contains block_sum
  {
    atomicAdd(scalar, cache[0]);
  }
}

/** scalar = x DOT y
 */
 __global__ void xDOTy_warp(const int N, double *x, double *y, double *scalar) {
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  const int stride = blockDim.x * gridDim.x;

  double sum = 0.0;
  for (; tid < N; tid += stride) {
    sum += x[tid] * y[tid];
  }
  tid = threadIdx.x;

  __syncthreads();
  for (int i = warpSize / 2; i != 0; i /= 2) {
    sum += __shfl_down_sync(0xffffffff, sum, i);
  }

  if (tid % warpSize == 0) // a block can consist of multiple warps
  {
    atomicAdd(scalar, sum);
  }
}

/** analyze_x_shared
 *
 * result[0] = sum;
 * result[1] = abs_sum;
 * result[2] = sqr_sum;
 * result[3] = mod_max;
 * result[4] = min;
 * result[5] = max;
 * result[6] = z_entries;
 */
// template <int block_size=BLOCK_SIZE>
__global__ void analyze_x_shared(const int N, double *x, double *results) {
  if (blockDim.x * blockIdx.x < N) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x; // global tid
    const int stride = blockDim.x * gridDim.x;

    __shared__ double cache[7][BLOCK_SIZE];

    double sum = 0.0, abs_sum = 0.0, sqr_sum = 0.0;
    // double mod_max = 0.0;
    double max = x[0];
    double min = max;
    double z_entries = 0;
    for (; tid < N; tid += stride) {
      double value = x[tid];
      sum += value;
      abs_sum += std::abs(value);
      sqr_sum += value*value;
```

```
      // mod_max = (std::abs(value) > mod_max)? value : mod_max;
      min = fmin(value, min);
      max = fmax(value, max);
      z_entries += (value)? 0 : 1;
    }
    tid = threadIdx.x; // block tid
    cache[0][tid] = sum;
    cache[1][tid] = abs_sum;
    cache[2][tid] = sqr_sum;
    cache[3][tid] = fmax(std::abs(min), max);
    cache[4][tid] = min;
    cache[5][tid] = max;
    cache[6][tid] = z_entries;

    __syncthreads();
    for (int i = blockDim.x / 2; i != 0; i /= 2) {
      __syncthreads();
      if (tid < i) { // lower half does smth, rest idles
        // sums
        cache[0][tid] += cache[0][tid + i];
        cache[1][tid] += cache[1][tid + i];
        cache[2][tid] += cache[2][tid + i];
        // min/max values
        cache[3][tid] = fmax(cache[3][tid + i], cache[3][tid]); // already all values are
            std::abs(...)
        cache[4][tid] = fmin(cache[4][tid + i], cache[4][tid]);
        cache[5][tid] = fmax(cache[5][tid + i], cache[5][tid]);

        // "sum"
        cache[6][tid] += cache[6][tid + i];
      }
    }

    if (tid == 0) // cache[0] now contains block_sum
    {
      atomicAdd(results, cache[0][0]);
      atomicAdd(results+1, cache[1][0]);
      atomicAdd(results+2, cache[2][0]);

      // Ideally...
      atomicMax(results+3, cache[3][0]);
      atomicMin(results+4, cache[4][0]);
      atomicMax(results+5, cache[5][0]);

      atomicAdd(results+6, cache[6][0]);
    }
  }
}

/** analyze_x_shared
 *
 * result[0] = sum;
 * result[1] = abs_sum;
 * result[2] = sqr_sum;
 * result[3] = mod_max;
 * result[4] = min;
 * result[5] = max;
 * result[6] = z_entries;
 */
__global__ void analyze_x_warp(const int N, double *x, double *results) {
  if (blockDim.x * blockIdx.x < N) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x; // global tid
    const int stride = blockDim.x * gridDim.x;

    double sum = 0.0, abs_sum = 0.0, sqr_sum = 0.0;
    // double mod_max = 0.0;
    double max = x[0];
    double min = max;
    int z_entries = 0;
    for (; tid < N; tid += stride) {
      double value = x[tid];
```

```
      sum += value;
      abs_sum += std::abs(value);
      sqr_sum += value*value;

      min = fmin(value, min);
      max = fmax(value, max);
      z_entries += (value)? 0 : 1;
    }
    tid = threadIdx.x; // block tid
    double mod_max = fmax(std::abs(min), max);

    __syncthreads();
    for (int i = warpSize / 2; i != 0; i /= 2) {
      //__syncthreads();
      sum += __shfl_down_sync(0xffffffff, sum, i);
      abs_sum += __shfl_down_sync(0xffffffff, abs_sum, i);
      sqr_sum += __shfl_down_sync(0xffffffff, sqr_sum, i);

      double tmp = __shfl_down_sync(0xffffffff, mod_max, i);
      mod_max = fmax(tmp, mod_max);
      tmp = __shfl_down_sync(0xffffffff, min, i);
      min = fmin(tmp, min);
      tmp = __shfl_down_sync(0xffffffff, max, i);
      max = fmax(tmp, max) ;

      z_entries += __shfl_down_sync(0xffffffff, z_entries, i);
    }
    // for (int i = blockDim.x / 2; i != 0; i /= 2) {
    // for (int i = warpSize / 2; i != 0; i /= 2) {
    //   //__syncthreads();
    //   sum += __shfl_xor_sync(-1, sum, i);
    //   abs_sum += __shfl_xor_sync(-1, abs_sum, i);
    //   sqr_sum += __shfl_xor_sync(-1, sqr_sum, i);

    //   double tmp = __shfl_xor_sync(-1, mod_max, i);
    //   mod_max = (tmp > mod_max) ? tmp : mod_max;
    //   tmp = __shfl_xor_sync(-1, min, i);
    //   min = (tmp < min) ? tmp : min;
    //   tmp = __shfl_xor_sync(-1, max, i);
    //   max = (tmp > max) ? tmp : max;

    //   z_entries += __shfl_xor_sync(-1, z_entries, i);
    // }

    if (tid % warpSize == 0) // a block can consist of multiple warps
    {
      atomicAdd(results, sum);
      atomicAdd(results+1, abs_sum);
      atomicAdd(results+2, sqr_sum);

      atomicMax(results+3, mod_max);
      atomicMin(results+4, min);
      atomicMax(results+5, max);

      atomicAdd(results+6, z_entries);
    }
  }
}

template <typename T>
void toCSV(std::fstream& csv, T* array, int size) {
  csv << size;
  for (int i = 0; i < size; ++i) {
    csv << ";" << array[i];
  }
  csv << std::endl;
}

int main(void) {
  Timer timer;
  std::vector<int> vec_Ns{100, 1000, 10000,  100000, 1000000, 10000000, 100000000};
  // std::vector<int> vec_Ns{100, 1000};
```

```
  std::vector<double> times(TESTS, 0.);

#ifdef CSV
  std::fstream csv_times, csv_results, csv_results2, csv_results3, csv_results_ref;
  std::string csv_times_name = "ph_data.csv";
  std::string csv_results_name = "ph_results.csv";
  std::string csv_results2_name = "ph_results2.csv";
  std::string csv_results3_name = "ph_results3.csv";
  std::string csv_results_ref_name = "ph_results_ref.csv";
  csv_times.open(csv_times_name, std::fstream::out | std::fstream::trunc);
  csv_results.open(csv_results_name, std::fstream::out | std::fstream::trunc);
  csv_results2.open(csv_results2_name, std::fstream::out | std::fstream::trunc);
  csv_results3.open(csv_results3_name, std::fstream::out | std::fstream::trunc);
  csv_results_ref.open(csv_results_ref_name, std::fstream::out | std::fstream::trunc);

  std::string header = "N;time_shared;time_warp;time_warp_adapt;time_dot;time_dot_warp;time_cpuref;time_cublas";
    // to csv file
  csv_times << header << std::endl;

  std::string header_results = "N;sum;abs_sum;sqr_sum;mod_max;min;max;z_entries";
  csv_results << header_results << std::endl;
  csv_results2 << header_results << std::endl;
  csv_results3 << header_results << std::endl;
  csv_results_ref << header_results << std::endl;
#endif

  for (int& N : vec_Ns) {
    //
    // Initialize CUBLAS:
    //
#ifdef DEBUG
    std::cout << "N = " << N << std::endl;
    std::cout << "Init CUBLAS..." << std::endl;
#endif
    cublasHandle_t h;
    cublasCreate(&h);

    //
    // allocate + init host memory:
    //
#ifdef DEBUG
    std::cout << "Allocating host arrays..." << std::endl;
#endif
    double *x = (double *)malloc(sizeof(double) * N);
    double *results = (double *)malloc(sizeof(double) * 7);
    double *results2 = (double *)malloc(sizeof(double) * 7);
    double *results3 = (double *)malloc(sizeof(double) * 7);
    double *results_ref = (double *)malloc(sizeof(double) * 7);
    std::vector<std::string> names {"sum", "abs_sum", "sqr_sum", "mod_max", "min", "max", "zero_entries"};

    std::generate_n(x, N, [n = -N/2] () mutable { return n++; });
    std::random_shuffle(x, x+N);
    // I'm placing some values here by hand, so that certain results can be forced
    // --> to test: mod_max, min, max...
    x[0] = -1.1;
    x[N/5] = 0.;
    x[N/3] = -(N-1);
    x[2*N/3] = N;

    std::fill(results, results+7, 0.0);
    results[3] = x[0];
    results[4] = x[0];
    results[5] = x[0];
    std::copy(results, results+7, results2);
    std::copy(results, results+7, results3);
    std::copy(results, results+7, results_ref);
    timer.reset();
    // results_ref[0] = std::accumulate(x, x+N, 0.0);
    for (int i = 0; i < N; ++i){
      double tmp = x[i];
```

```
      results_ref[0] += tmp;
      results_ref[1] += std::abs(tmp);
      results_ref[2] += tmp*tmp;
      results_ref[4] = fmin(tmp, results_ref[4]);
      results_ref[5] = fmax(tmp, results_ref[5]);
      results_ref[6] += tmp ? 0 : 1;
    }
    results_ref[3] = fmax(std::abs(results_ref[4]), results_ref[5]);
    double time_cpuref = timer.get();
    /*result[0] = sum;
    * result[1] = abs_sum;
    * result[2] = sqr_sum;
    * result[3] = mod_max;
    * result[4] = min;
    * result[5] = max;
    * result[6] = z_entries;*/

    //
    // allocate device memory
    //
#ifdef DEBUG
    std::cout << "Initialized results containers: " << std::endl;
    printContainer(results, 7);
    printContainer(results2, 7);
    std::cout << "Allocating CUDA arrays..." << std::endl;
#endif
    double *cuda_x;
    double *cuda_results;
    double *cuda_scalar;
    cudaMalloc(&cuda_x, sizeof(double) * N);
    cudaMalloc(&cuda_results, sizeof(double) * 7);
    cudaMalloc(&cuda_scalar, sizeof(double));
    //
    // Copy data to GPU
    //
#ifdef DEBUG
    std::cout << "Copying data to GPU..." << std::endl;
#endif
    cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice);

    //
    // Let CUBLAS do the work:
    //
#ifdef DEBUG
    std::cout << "Running dot products with CUBLAS..." << std::endl;
#endif
    double *cublas = (double *)malloc(sizeof(double));
    for (int iter = 0; iter < TESTS; ++iter) {
      *cublas= 0.0;
      cudaMemcpy(cuda_scalar, &cublas, sizeof(double), cudaMemcpyHostToDevice);
      timer.reset();
      cublasDdot(h, N, cuda_x, 1, cuda_x, 1, cublas);
      // cublasDnrm2(h, N-1, cuda_x, 1, cuda_scalar);
      // cudaMemcpy(&cublas, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
      times[iter] = timer.get();
    }
    double time_cublas = median(times);
#ifdef DEBUG
    std::cout << "cublas: " << *cublas << std::endl;
#endif
#ifdef DEBUG
    std::cout << "Running with analyze_x_shared..." << std::endl;
#endif
  for (int iter = 0; iter < TESTS; ++iter) {
      cudaMemcpy(cuda_results, results, sizeof(double) * 7, cudaMemcpyHostToDevice);
      timer.reset();
      analyze_x_shared<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_x, cuda_results);
      cudaMemcpy(results, cuda_results, sizeof(double) * 7, cudaMemcpyDeviceToHost);
      times[iter] = timer.get();
    }
    double time_shared = median(times);

```

```
#ifdef DEBUG
  std::cout << "Running analyze_x_warp<GS, BS>..." << std::endl;
#endif
  for (int iter = 0; iter < TESTS; ++iter) {
   cudaMemcpy(cuda_results, results2, sizeof(double) * 7, cudaMemcpyHostToDevice);
   timer.reset();
   analyze_x_warp<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_x, cuda_results);
   cudaMemcpy(results2, cuda_results, sizeof(double) * 7, cudaMemcpyDeviceToHost);
   times[iter] = timer.get();
  }
  double time_warp = median(times);

#ifdef DEBUG
  std::cout << "Running analyze_x_warp<N/BS, BS>..." << std::endl;
#endif
  for (int iter = 0; iter < TESTS; ++iter) {
   cudaMemcpy(cuda_results, results3, sizeof(double) * 7, cudaMemcpyHostToDevice);
   int adapt_gridsize = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
   // N/BLOCK_SIZE could results in a gridsize smaller than 1.
   // also,
   timer.reset();
   analyze_x_warp<<<adapt_gridsize, BLOCK_SIZE>>>(N, cuda_x, cuda_results);
   cudaMemcpy(results3, cuda_results, sizeof(double) * 7, cudaMemcpyDeviceToHost);
   times[iter] = timer.get();
  }
  double time_warp_adapt = median(times);

#ifdef DEBUG
  std::cout << "Running dot product xDOTy..." << std::endl;
#endif
  for (int iter = 0; iter < TESTS; ++iter) {
   double dot = 0.0;
   cudaMemcpy(cuda_scalar, &dot, sizeof(double), cudaMemcpyHostToDevice);
   timer.reset();
   xDOTy<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_x, cuda_x, cuda_scalar);
   cudaMemcpy(&dot, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
   times[iter] = timer.get();
  }
  double time_dot = median(times);

#ifdef DEBUG
  std::cout << "Running dot product xDOTy_warp..." << std::endl;
#endif
  for (int iter = 0; iter < TESTS; ++iter) {
   double dot = 0.0;
   cudaMemcpy(cuda_scalar, &dot, sizeof(double), cudaMemcpyHostToDevice);
   timer.reset();
   xDOTy_warp<<<GRID_SIZE, BLOCK_SIZE>>>(N, cuda_x, cuda_x, cuda_scalar);
   cudaMemcpy(&dot, cuda_scalar, sizeof(double), cudaMemcpyDeviceToHost);
   times[iter] = timer.get();
  }
  double time_dot_warp = median(times);
  //
  // Compare results
  //
#ifdef DEBUG
  std::cout << "DEBUG output:" << std::endl;
  std::cout << "x:" << std::endl;
  int only = 4;
  printContainer(x, N, only);

  std::cout << ">SHARED<" << std::endl;
  printResults(results, results_ref, names, names.size());

  std::cout << ">WARP<" << std::endl;
  printResults(results2, results_ref, names, names.size());

  std::cout << "GPU shared runtime: " << time_shared << std::endl;
  std::cout << "GPU warp runtime: " << time_warp << std::endl;
  std::cout << "GPU warp adaptive runtime: " << time_warp_adapt << std::endl;
  std::cout << "GPU dot runtime: " << time_dot << std::endl;
  std::cout << "GPU dot_warp runtime: " << time_dot_warp << std::endl;
```

```
    std::cout << "CPU ref runtime: " << time_cpuref << std::endl;

    //
    // Clean up:
    //
    std::cout << "Cleaning up..." << std::endl;
    std::cout << "----------------------------------------------------" << std::endl;
#endif

#ifdef CSV
    std::string sep = ";";
    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;

    toCSV(csv_results, results, 7);
    toCSV(csv_results2, results2, 7);
    toCSV(csv_results3, results3, 7);
    toCSV(csv_results_ref, results_ref, 7);
#endif
    free(x);
    free(results);
    free(results2);
    free(results3);
    free(results_ref);
    free(cublas);

    cudaFree(cuda_x);
    cudaFree(cuda_results);
    cudaFree(cuda_scalar);

    cublasDestroy(h);
  }

#ifdef CSV
  csv_times.close();
  csv_results.close();
  csv_results2.close();
  csv_results3.close();
  csv_results_ref.close();

  std::cout << "\nRuntimes in csv form can be found here\nhttps://gtx1080.360252.org/2020/
      ex6/" + csv_times_name << std::endl;
  std::cout << "\nResults in csv form can be found here\nhttps://gtx1080.360252.org/2020/ex6
      /" + csv_results_name << std::endl;
  std::cout << "\nResults in csv form can be found here\nhttps://gtx1080.360252.org/2020/ex6
      /" + csv_results2_name << std::endl;
  std::cout << "\nResults in csv form can be found here\nhttps://gtx1080.360252.org/2020/ex6
      /" + csv_results3_name << std::endl;
  std::cout << "\nResults in csv form can be found here\nhttps://gtx1080.360252.org/2020/ex6
      /" + csv_results_ref_name << std::endl;
#endif
  return EXIT_SUCCESS;
}
```

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

```
#include "timer.hpp"
#include "poisson2d.hpp"
#include <algorithm>
#include <numeric>
#include <cmath>
// #include <cublas_v2.h>
// #include <cuda_runtime.h>
#include <fstream>
#include <iostream>
#include <stdio.h>
#include <string>
#include <vector>

#define BLOCK_SIZE 256
#define GRID_SIZE 128
```

```
// #define SEP ";"

// #define DEBUG
#ifndef DEBUG
  #define CSV
#endif

// START--------------- CONVENIENCE FUNTIONS ------------------START //
// template <typename T>
// void printContainer(T* container, const int size) {
//   std::cout << *container;
//   for (int i = 1; i < size; ++i)
//     std::cout << " | " << *(container+i) ;
//   std::cout << std::endl;
// }

template <typename T>
void printContainer(T container, const int size) {
  std::cout << container[0];
  for (int i = 1; i < size; ++i)
    std::cout << " | " << container[i] ;
  std::cout << std::endl;
}

template <typename T>
void printContainer(T container, const int size, const int only) {
  std::cout << container[0];
  for (int i = 1; i < only; ++i)
      std::cout  << " | " << container[i];
  std::cout << " | ...";
  for (int i = size - only; i < size; ++i)
    std::cout  << " | " << container[i];
  std::cout << std::endl;
}

template <typename T>
void printContainerStrided(T container, const int size, const int stride) {
  std::cout << container[0];
  for (int i = stride; i < size; i+=stride)
      std::cout  << " | " << container[i];
  std::cout << std::endl;
}

void printResults(double* results, std::vector<std::string> names, int size){
  std::cout << "Results:" << std::endl;
  for (int i = 0; i < size; ++i) {
    std::cout << names[i] << " : " << results[i] << std::endl;
  }
}

void printResults(double* results, double* ref, std::vector<std::string> names, int size){
  std::cout << "Results (with difference to reference):" << std::endl;
  for (int i = 0; i < size; ++i) {
    std::cout << names[i] << " = " << results[i] << " ||  " << ref[i] - results[i] << std::endl;
  }
}

template <typename T>
void toCSV(std::fstream& csv, T* array, int size) {
  csv << size;
  for (int i = 0; i < size; ++i) {
    csv << ";" << array[i];
  }
  csv << std::endl;
}
// END--------------- CONVENIENCE FUNCTIONS ------------------END //

//
// START--------------- KERNELS ------------------START //
//
// y = A * x
```

```
__global__ void cuda_csr_matvec_product(int N, int *csr_rowoffsets,
  int *csr_colindices, double *csr_values,
  double *x, double *y)
{
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
    double sum = 0;
    for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++) {
      sum += csr_values[k] * x[csr_colindices[k]];
    }
    y[i] = sum;
  }
}

// Y= A * X
__global__ void A_MatMul_Xcm(int N, int K,
  int *csr_rowoffsets, int *csr_colindices, double *csr_values,
  double *X, double *Y)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  if (tid < N){
    int row_start = csr_rowoffsets[tid];
    int row_end = csr_rowoffsets[tid + 1];

    // for (int k = 0; k < K; ++k){
    //   double sum = 0.0;
    //   for (int i = row_start; i < row_end; i++) {
    //     sum += csr_values[i]* X[csr_colindices[i]*K + k];
    //   }
    //   Y[k + tid*K] = sum;
    // }

    for (int i = row_start; i < row_end; i++) {
      double aij = csr_values[i];
      int row_of_X = csr_colindices[i]*K;
      for (int k = 0; k < K; ++k){
        Y[k + tid*K] += aij * X[row_of_X + k];
      }
    }
  }
}

// Y= A * X
__global__ void A_MatMul_Xrm(int N, int K,
  int *csr_rowoffsets, int *csr_colindices, double *csr_values,
  double *X, double *Y)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  if (tid < N){
    int row_start = csr_rowoffsets[tid];
    int row_end = csr_rowoffsets[tid + 1];

    for (int k = 0; k < K; ++k){
      double sum = 0.0;
      for (int i = row_start; i < row_end; i++) {
        sum += csr_values[i]* X[csr_colindices[i] + k*N];
      }
      Y[k + tid*K] = sum;
    }
  }
}

// Y= A * X
__global__ void A_MatMul_Xrm_boost(int N, int K,
  int *csr_rowoffsets, int *csr_colindices, double *csr_values,
  double *X, double *Y)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  if (tid < N){
    int row_start = csr_rowoffsets[tid];
```

```
    int row_end = csr_rowoffsets[tid + 1];

    for (int k = 0; k < K; ++k){
      double sum = 0.0;
      for (int i = row_start; i < row_end; i++) {
        sum += csr_values[i]* X[csr_colindices[i] + k*N];
      }
      Y[k + tid*K] = sum;
    }
  }
}
//
// END--------------- KERNELS ------------------END //
//

/**TO-DO:
 * Adapt signature
 *
 */
// void assemble_csr_on_gpu(){
//     // Perform the calculations
//     int numberOfValues;
//     timer.reset();
//     count_nnz<<<GRID_SIZE, BLOCK_SIZE>>>(cuda_row_offsets_2, N, M);
//     exclusive_scan(cuda_row_offsets_2, cuda_row_offsets, N*M+1);
//     cudaMemcpy(row_offsets, cuda_row_offsets, sizeof(int) * (N*M+1),
    cudaMemcpyDeviceToHost);
//     numberOfValues = row_offsets[N*M];

//   #ifdef DEBUG
//     printContainer(row_offsets, N*M+1, 4);
//     std::cout << std::endl;
//   #endif

//     double *values = (double *)malloc(sizeof(double) * numberOfValues);
//     int *columns = (int *)malloc(sizeof(int) * numberOfValues);
//     cudaMalloc(&cuda_columns, sizeof(int) * numberOfValues);
//     cudaMalloc(&cuda_values, sizeof(double) * numberOfValues);

//     assembleA<<<GRID_SIZE, BLOCK_SIZE>>>(cuda_values, cuda_columns, cuda_row_offsets, N,
    M);
//     double time_assemble_gpu = timer.get();

//     cudaMemcpy(values, cuda_values, sizeof(double) * numberOfValues,
    cudaMemcpyDeviceToHost);
//     cudaMemcpy(columns, cuda_columns, sizeof(int) * numberOfValues,
    cudaMemcpyDeviceToHost);
//   }

int main(void) {
  Timer timer;
  std::vector<int> vec_Ns{100, 10000, 1000000};
  // std::vector<int> vec_Ks{2, 4, 8, 16};
  // std::vector<int> vec_Ns{1000, 100000};
  std::vector<int> vec_Ks{3, 5, 9, 15};
  // std::vector<int> vec_Ns{1000000};

#ifdef CSV
  std::fstream csv_times;
  std::string csv_times_name = "ph_data.csv";
  csv_times.open(csv_times_name, std::fstream::out | std::fstream::trunc);

  std::string header = "N;K;time_single;time_rm_stacked;time_cm_stacked";
    // to csv file
  csv_times << header << std::endl;
#endif

  for (int& N : vec_Ns) {
    for (int& K : vec_Ks) {
      // cublasHandle_t h;
      // cublasCreate(&h);

```

```
        //
        // allocate + init host memory:
        //
    #ifdef DEBUG
        std::cout << "Allocating host + device arrays..." << std::endl;
    #endif
        // "Vectors"
        double* X = (double *)malloc(sizeof(double) * N * K);
        double* Y = (double *)malloc(sizeof(double) * N * K);
        double* Y2 = (double *)malloc(sizeof(double) * N * K);
        // double* x = (double *)malloc(sizeof(double) * N);
        double* y = (double *)malloc(sizeof(double) * N);
        std::fill(X, X + (N*K), 1.);
        std::fill(Y, Y + (N*K), 0.);
        std::fill(Y2, Y2 + (N*K), 0.);
        // std::fill(x, x + N, 1.);

        double *cuda_X;
        double *cuda_Y;
        // double *cuda_Y2;
        // double *cuda_x;
        double *cuda_y;
        cudaMalloc(&cuda_X, sizeof(double) * N*K);
        cudaMalloc(&cuda_Y, sizeof(double) * N*K);
        // cudaMalloc(&cuda_Y2, sizeof(double) * N*K);
        // cudaMalloc(&cuda_x, sizeof(double) * N);
        cudaMalloc(&cuda_y, sizeof(double) * N);

        // Matrix
        int* csr_rowoffsets = (int* )malloc(sizeof(int) * (N+1));
        int* csr_colindices = (int* )malloc(sizeof(int) * 5*N);
        double* csr_values = (double* )malloc(sizeof(double) * 5*N);

        int* cuda_csr_rowoffsets;
        int* cuda_csr_colindices;
        double* cuda_csr_values;
        cudaMalloc(&cuda_csr_rowoffsets, sizeof(int) * (N+1));
        cudaMalloc(&cuda_csr_colindices, sizeof(int) * 5*N);
        cudaMalloc(&cuda_csr_values, sizeof(double) * 5*N);
        //
        // Copy data to GPU
        //
    #ifdef DEBUG
        std::cout << "Copying data to GPU..." << std::endl;
    #endif
        cudaMemcpy(cuda_X, X, sizeof(double) * N*K, cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_Y, Y, sizeof(double) * N*K, cudaMemcpyHostToDevice);
        // cudaMemcpy(cuda_Y2, Y2, sizeof(double) * N*K, cudaMemcpyHostToDevice);
        // cudaMemcpy(cuda_x, X, sizeof(double) * N, cudaMemcpyHostToDevice);
    //    cudaMemcpy(cuda_y, y, sizeof(double) * N*K, cudaMemcpyHostToDevice);

    // Assemble A
    #ifdef DEBUG
        std::cout << "Generating A..." << std::endl;
    #endif
        generate_fdm_laplace(sqrt(N), csr_rowoffsets, csr_colindices, csr_values);
    #ifdef DEBUG
        std::cout << "Generating A done!" << std::endl;
    #endif
        cudaMemcpy(cuda_csr_rowoffsets, csr_rowoffsets, sizeof(int) * (N+1),
            cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_csr_colindices, csr_colindices, sizeof(int) * 5*N,
            cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_csr_values, csr_values, sizeof(double) * 5*N, cudaMemcpyHostToDevice);

        // ------------------ TEST ---------------- //

  #ifdef DEBUG
        std::cout << "N = " << N << std::endl;
        std::cout << "K = " << K << std::endl;

        std::cout << "Running per vector product kernel K times..." << std::endl;
```

```
#endif
      timer.reset();
      for (int k = 0; k < K; ++k)
        cuda_csr_matvec_product<<<GRID_SIZE, BLOCK_SIZE>>>(
          N,
          cuda_csr_rowoffsets, cuda_csr_colindices, cuda_csr_values,
          cuda_X, cuda_y);
      cudaMemcpy(y, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);
      double time_single = timer.get();

  #ifdef DEBUG
      std::cout << "Running RowMajor stacked kernel..." << std::endl;
  #endif
      timer.reset();
      A_MatMul_Xrm<<<GRID_SIZE, BLOCK_SIZE>>>(
          N, K,
          cuda_csr_rowoffsets, cuda_csr_colindices, cuda_csr_values,
          cuda_X, cuda_Y);
      cudaMemcpy(Y, cuda_Y, sizeof(double) * N*K, cudaMemcpyDeviceToHost);
      double time_rm_stacked = timer.get();

  #ifdef DEBUG
      std::cout << "Running ColumnMajor stacked kernel..." << std::endl;
  #endif
      cudaMemcpy(cuda_Y, Y2, sizeof(double) * N*K, cudaMemcpyHostToDevice);
      timer.reset();
      A_MatMul_Xcm<<<GRID_SIZE, BLOCK_SIZE>>>(
          N, K,
          cuda_csr_rowoffsets, cuda_csr_colindices, cuda_csr_values,
          cuda_X, cuda_Y);
      cudaMemcpy(Y2, cuda_Y, sizeof(double) * N*K, cudaMemcpyDeviceToHost);
      double time_cm_stacked = timer.get();




      //
      // Compare results
      //
  #ifdef DEBUG
      std::cout << "DEBUG output:" << std::endl;
      // int only = 4;
      std::cout << "A (non zero entries by row)" << std::endl;
      // int csr_values_size = csr_rowoffsets[N+1];
      // printContainer(y, N);
      std::cout << "Row" << std::endl;
      int max_output = 10;
      for (int row = 0; row < min(N, max_output); row++){
        std::cout << row << ": ";
        printContainer(csr_values + csr_rowoffsets[row], min(csr_rowoffsets[row+1]-csr_rowoffsets[row], max_output));
      }

      std::cout << "y:" << std::endl;
      printContainer(y, min(N, max_output));
      std::cout << "Y_rm:" << std::endl;
      printContainerStrided(Y, min(N, max_output)*K, K);
      std::cout << "Y_cm:" << std::endl;
      printContainerStrided(Y2, min(N, max_output)*K, K);


      std::cout << "Single runtime: " << time_single << std::endl;
      std::cout << "RM Stacked runtime: " << time_rm_stacked << std::endl;
      std::cout << "CM Stacked runtime: " << time_cm_stacked << std::endl;

      //
      // Clean up:
      //
      std::cout << "---------------------------------------------------" << std::endl;
      std::cout << "Cleaning up..." << std::endl;
  #endif

```

```
  #ifdef CSV
      std::string sep = ";";
      csv_times << N << sep
                << K << sep
                << time_single << sep
                << time_rm_stacked << sep
                << time_cm_stacked
                << std::endl;
  #endif
    free(X);
    free(Y);
    free(Y2);
    // free(x);
    free(y);
    free(csr_rowoffsets);
    free(csr_colindices);
    free(csr_values);

    cudaFree(cuda_X);
    cudaFree(cuda_Y);
    // cudaFree(cuda_Y2);
    // cudaFree(cuda_x);
    cudaFree(cuda_y);
    cudaFree(cuda_csr_rowoffsets);
    cudaFree(cuda_csr_colindices);
    cudaFree(cuda_csr_values);
#ifdef DEBUG
    std::cout << "Clean up done!" << std::endl;
#endif
    }
  }

#ifdef CSV
  csv_times.close();

  std::cout << "\nRuntimes in csv form can be found here\nhttps://gtx1080.360252.org/2020/
      ex6/" + csv_times_name << std::endl;
#endif
  return EXIT_SUCCESS;
}
```