360.252 - Computational Science on Many-Core Architectures

WS 2020 - Exercise 6

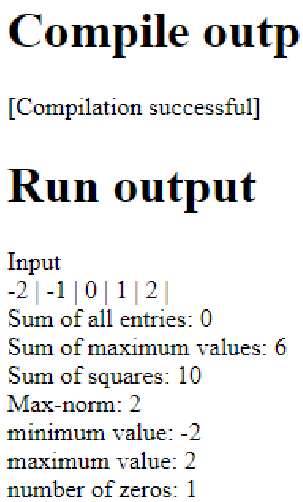
Christian Gollmann, 01435044

Last update: November 30, 2020

Contents

1 Dot Product with warp shuffles - shared memory

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



Compile output

[Compilation successful]

Run output

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

Figure 1: Output of shared memory version

Listing 1: Calculations done with shared memory

```
#include <iostream>

__global__ void sharedMemoryKernel(const int* x, int* y, const int N) {
    __shared__ int sharedMemory[7][256];
     int sum = 0;
     int maxSum = 0;
     int sqrSum = 0;
     int maxMod = 0;
     int min = x[0];
     int max = 0;
     int zeros = 0;

    for (int tid = blockDim.x * blockIdx.x + threadIdx.x;
    tid < N; tid += gridDim.x * blockDim.x) {
        int val = x[tid];

        sum += val;
        maxSum += std::abs(val);
        sqrSum += val*val;
        maxMod = std::abs(val) > maxMod ? val : maxMod;
        min = val < min ? val : min;
        max = val > max ? val :max;
        zeros += val == 0 ? 1 : 0;
    }

    int tid = threadIdx.x;
    if (tid < N) {
        sharedMemory[0][threadIdx.x] = sum;
```

```
        sharedMemory[1][threadIdx.x] = maxSum;
        sharedMemory[2][threadIdx.x] = sqrSum;
        sharedMemory[3][threadIdx.x] = maxMod;
        sharedMemory[4][threadIdx.x] = min;
        sharedMemory[5][threadIdx.x] = max;
        sharedMemory[6][threadIdx.x] = zeros;

        __syncthreads();
        // blockDim.x needs to be a power of 2 in order for this to work
        for (int i = blockDim.x/2; i != 0; i /= 2) {
            __syncthreads();
            if (tid < i) {
                sharedMemory[0][tid] += sharedMemory[0][tid + i];
                sharedMemory[1][tid] += sharedMemory[1][tid + i];
                sharedMemory[2][tid] += sharedMemory[2][tid + i];
                sharedMemory[3][tid] = sharedMemory[3][tid] > sharedMemory[3][tid + i]
                ? sharedMemory[3][tid] : sharedMemory[3][tid + i];
                sharedMemory[4][tid] = sharedMemory[4][tid] < sharedMemory[4][tid + i]
                ? sharedMemory[4][tid] : sharedMemory[4][tid + i];
                sharedMemory[5][tid] = sharedMemory[5][tid] > sharedMemory[5][tid + i]
                ? sharedMemory[5][tid] : sharedMemory[5][tid + i];
                sharedMemory[6][tid] += sharedMemory[6][tid + i];
            }
        }
    }

    if (tid == 0) {
        atomicAdd(y, sharedMemory[0][0]);
        atomicAdd(y+1, sharedMemory[1][0]);
        atomicAdd(y+2, sharedMemory[2][0]);
        atomicMax(y+3, sharedMemory[3][0]);
        atomicMin(y+4, sharedMemory[4][0]);
        atomicMax(y+5, sharedMemory[5][0]);
        atomicAdd(y+6, sharedMemory[6][0]);
    }
}

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


int main() {

    int N = 5;

     int *x = (int *)malloc(sizeof(int) * N);
     int *y = (int *)malloc(sizeof(int) * 7);

    for (int i = 0; i < N; i++) {
        x[i] = i - N/2;
    }

     int *cuda_x;
```

```
    int *cuda_y;
    cudaMalloc(&cuda_x, sizeof(int) * N);
    cudaMalloc(&cuda_y, sizeof(int) * 7);

    cudaMemcpy(cuda_x, x, sizeof(int) * N, cudaMemcpyHostToDevice);

    sharedMemoryKernel<<<256, 256>>>(cuda_x, cuda_y, N);

    cudaMemcpy(y, cuda_y, sizeof(int) * 7, cudaMemcpyDeviceToHost);

    std::cout << "Input" << std::endl;
    printContainer(x, N);
    std::cout << std::endl;

    std::cout << "Sum of all entries: " << y[0] << std::endl;
    std::cout << "Sum of maximum values: " << y[1] << std::endl;
    std::cout << "Sum of squares: " << y[2] << std::endl;
    std::cout << "Max-norm: " << y[3] << std::endl;
    std::cout << "minimum value: " << y[4] << std::endl;
    std::cout << "maximum value: " << y[5] << std::endl;
    std::cout << "number of zeros: " << y[6] << std::endl;

    free(x);
    free(y);
    cudaFree(cuda_x);
    cudaFree(cuda_y);

    return EXIT_SUCCESS;
}
```

2 Dot Product with warp shuffles - warp shuffles

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

Listing 2: Calculations done with warp shuffles

```
#include <iostream>

__global__ void shuffleKernel(const int* x, int* y, const int N) {

    int sum          = 0;
    int maxSum       = 0;
    int sqrSum       = 0;
    int maxMod       = 0;
    int min          = x[0];
    int max          = 0;
    int zeros        = 0;

    for (int tid = blockDim.x * blockIdx.x + threadIdx.x;
    tid < N; tid += gridDim.x * blockDim.x) {
        int val = x[tid];

        sum        += val;
        maxSum     += std::abs(val);
        sqrSum     += val*val;
        maxMod     = std::abs(val) > maxMod ? val : maxMod;
        min        = val < min ? val : min;
        max        = val > max ? val :max;
        zeros      += val == 0 ? 1 : 0;
    }

    int tid = threadIdx.x;
    for (int i = warpSize / 2; i != 0; i /= 2) {
        sum        += __shfl_down_sync(0xffffffff, sum, i);
        maxSum     += __shfl_down_sync(0xffffffff, maxSum, i);
        sqrSum     += __shfl_down_sync(0xffffffff, sqrSum, i);
        int temporary = __shfl_down_sync(0xffffffff, maxMod, i);
        maxMod     = temporary > maxMod ? temporary : maxMod;
        temporary  = __shfl_down_sync(0xffffffff, min, i);
        min        = temporary < min ? temporary : min;
        temporary  = __shfl_down_sync(0xffffffff, max, i);
        max        = temporary > max ? temporary : max;
        zeros      += __shfl_down_sync(0xffffffff, zeros, i);
    }
    __syncthreads();
    if (tid % warpSize == 0) {
        atomicAdd(y,   sum);
        atomicAdd(y+1, maxSum);
        atomicAdd(y+2, sqrSum);
        atomicMax(y+3, maxMod);
        atomicMin(y+4, min);
        atomicMax(y+5, max);
        atomicAdd(y+6, zeros);
    }
}

template <typename T>
```

```
void printContainer(T container, int N) {
    for (int i = 0; i < N; i++) {
        std::cout << container[i] << " | ";
    }
}


int main() {

    int N = 100000;

    int *x = (int *)malloc(sizeof(int) * N);
    int *y = (int *)malloc(sizeof(int) * 7);

    for (int i = 0; i < N; i++) {
        x[i] = i - N/2;
    }

    int *cuda_x;
    int *cuda_y;
    cudaMalloc(&cuda_x, sizeof(int) * N);
    cudaMalloc(&cuda_y, sizeof(int) * 7);

    cudaMemcpy(cuda_x, x, sizeof(int) * N, cudaMemcpyHostToDevice);

    shuffleKernel<<<N/256, 128>>>(cuda_x, cuda_y, N);

    cudaMemcpy(y, cuda_y, sizeof(int) * 7, cudaMemcpyDeviceToHost);

    //std::cout << "Input" << std::endl;
    //printContainer(x, N);
    //std::cout << std::endl;

    std::cout << "Sum of all entries: " << y[0] << std::endl;
    std::cout << "Sum of maximum values: " << y[1] << std::endl;
    std::cout << "Sum of squares: " << y[2] << std::endl;
    std::cout << "Max-norm: " << y[3] << std::endl;
    std::cout << "minimum value: " << y[4] << std::endl;
    std::cout << "maximum value: " << y[5] << std::endl;
    std::cout << "number of zeros: " << y[6] << std::endl;

    free(x);
    free(y);
    cudaFree(cuda_x);
    cudaFree(cuda_y);

    return EXIT_SUCCESS;
}
```

3 Dot Product with warp shuffles - performance comparison

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

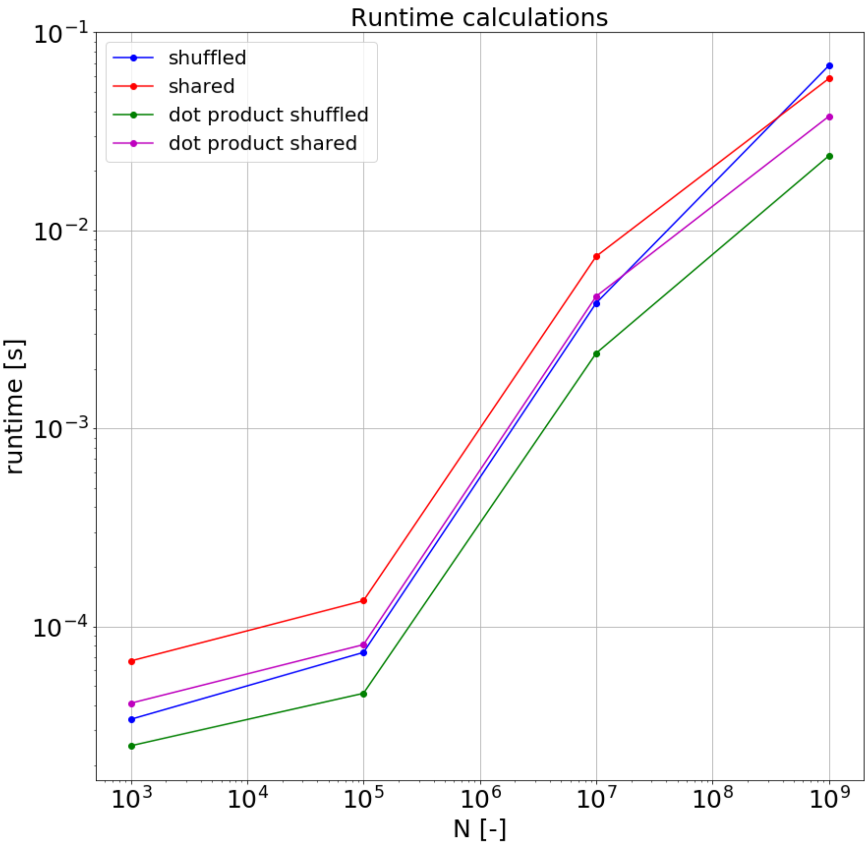


Figure 2: Runtimes for different versions

Listing 3: Dot product using shared memory

```
__global__ void dot_product(int* x, int* y, int* dot, int N) {

    int index = threadIdx.x + blockDim.x * blockIdx.x;
    int stride = blockDim.x * gridDim.x;

    __shared__ int cache[128];

    int temp = 0;
    while (index < N) {
        temp += x[index] * y[index];
        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();
```

```

    for (int i = blockDim.x/2; i > 0; i/= 2) {
        __syncthreads();
        if (threadIdx.x < i)
            cache[threadIdx.x] += cache[threadIdx.x + i];
    }

    if (threadIdx.x == 0)
        atomicAdd(dot, cache[0]);

}
```

Listing 4: Dot product using warp shuffles

```
__global__ void dot_product_shuffle(int* x, int* y, int* dot, int N) {

    int index = threadIdx.x + blockDim.x * blockIdx.x;
    int stride = blockDim.x * gridDim.x;

    int temp = 0;
    while (index < N) {
        temp += x[index] * y[index];
        index += stride;
    }

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

    __syncthreads();

    int tid = threadIdx.x;

    if (tid % warpSize == 0) {
        atomicAdd(dot,   temp);
    }

}
```

4 Sparse Matrix Times Dense Matrix - The kernel

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

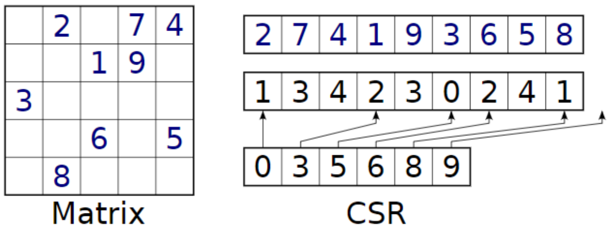


Figure 3: testmatrix from the lecture

Run output

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

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

Figure 4: results of matrix vector and matrix matrix product

Listing 5: matrix*vector and matrix*matrix

```
#include <iostream>

// y = A * x
__global__ void sparseVector(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 sparseDense(int N, int K, 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) {

    for (int k = 0; k < K; k++) {
      double sum = 0;

      for (int jj = csr_rowoffsets[i]; jj < csr_rowoffsets[i + 1]; jj++) {
        sum += csr_values[jj] * x[csr_colindices[jj] + N*k];
      }

      y[i + N*k] = sum;
    }

  }
}


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


int main() {

  int N = 5;
  int K = 3;

  double *values = ( double *)malloc(sizeof( double) * 9);
  int *colindices = (int*)malloc(sizeof(int) * 9);
  int *offsets = (int*)malloc(sizeof(int) * 6);

  double *x = ( double *)malloc(sizeof( double) * N);
  double *X = (double*)malloc(sizeof(double) * N*K);

  double *y = ( double *)malloc(sizeof( double) * N);
  double *Y = (double*)malloc(sizeof(double) * N*K);

  values[0] = 2;
  values[1] = 7;
  values[2] = 4;
  values[3] = 1;
  values[4] = 9;
  values[5] = 3;
  values[6] = 6;
  values[7] = 5;
  values[8] = 8;

  colindices[0] = 1;
  colindices[1] = 3;
  colindices[2] = 4;
```

```
    colindices[3] = 2;
    colindices[4] = 3;
    colindices[5] = 0;
    colindices[6] = 2;
    colindices[7] = 4;
    colindices[8] = 1;

    offsets[0] = 0;
    offsets[1] = 3;
    offsets[2] = 5;
    offsets[3] = 6;
    offsets[4] = 8;
    offsets[5] = 9;

    x[0] = 1;
    x[1] = 2;
    x[2] = 3;
    x[3] = 4;
    x[4] = 5;

    X[0] = 1;
    X[1] = 2;
    X[2] = 3;
    X[3] = 4;
    X[4] = 5;
    X[5] = 1;
    X[6] = 2;
    X[7] = 3;
    X[8] = 4;
    X[9] = 5;
    X[10] = 1;
    X[11] = 2;
    X[12] = 3;
    X[13] = 4;
    X[14] = 1;

    double *cuda_values;
    int *cuda_colindices;
    int *cuda_offsets;
    double *cuda_x;
    double *cuda_X;
    double *cuda_y;
    double *cuda_Y;

    cudaMalloc(&cuda_values, sizeof(double) * 9);
    cudaMalloc(&cuda_colindices, sizeof(int) * 9);
    cudaMalloc(&cuda_offsets, sizeof(int) * 6);
    cudaMalloc(&cuda_x, sizeof(double) * N);
    cudaMalloc(&cuda_X, sizeof(double) * N*K);
    cudaMalloc(&cuda_y, sizeof(double) * N);
    cudaMalloc(&cuda_Y, sizeof(double) * N*K);

    cudaMemcpy(cuda_values, values, sizeof(double) * 9, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_colindices, colindices, sizeof(int) * 9, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_offsets, offsets, sizeof(int) * 6, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_X, X, sizeof(double) * N*K, cudaMemcpyHostToDevice);
```

```
    sparseVector<<<256, 256>>>(N, cuda_offsets,
    cuda_colindices, cuda_values, cuda_x, cuda_y);
    cudaMemcpy(y, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);

    sparseDense<<<256, 256>>>(N, K, cuda_offsets,
    cuda_colindices, cuda_values, cuda_X, cuda_Y);
    cudaMemcpy(Y, cuda_Y, sizeof(double) * N*K, cudaMemcpyDeviceToHost);


    std::cout << "x : " << std::endl;
    printContainer(x, N);
    std::cout << "Result : " << std::endl;
    printContainer(y, N);

    std::cout << std::endl;
    std::cout << "X : " << std::endl;
    printContainer(X, N*K);
    std::cout << "Result : " << std::endl;
    printContainer(Y, N*K);

    free(x);
    free(X);
    free(y);
    free(Y);
    free(values);
    free(colindices);
    free(offsets);

    cudaFree(cuda_values);
    cudaFree(cuda_colindices);
    cudaFree(cuda_offsets);
    cudaFree(cuda_x);
    cudaFree(cuda_X);
    cudaFree(cuda_y);
    cudaFree(cuda_Y);

    return EXIT_SUCCESS;
}
```

5 Sparse Matrix Times Dense Matrix - benchmark

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

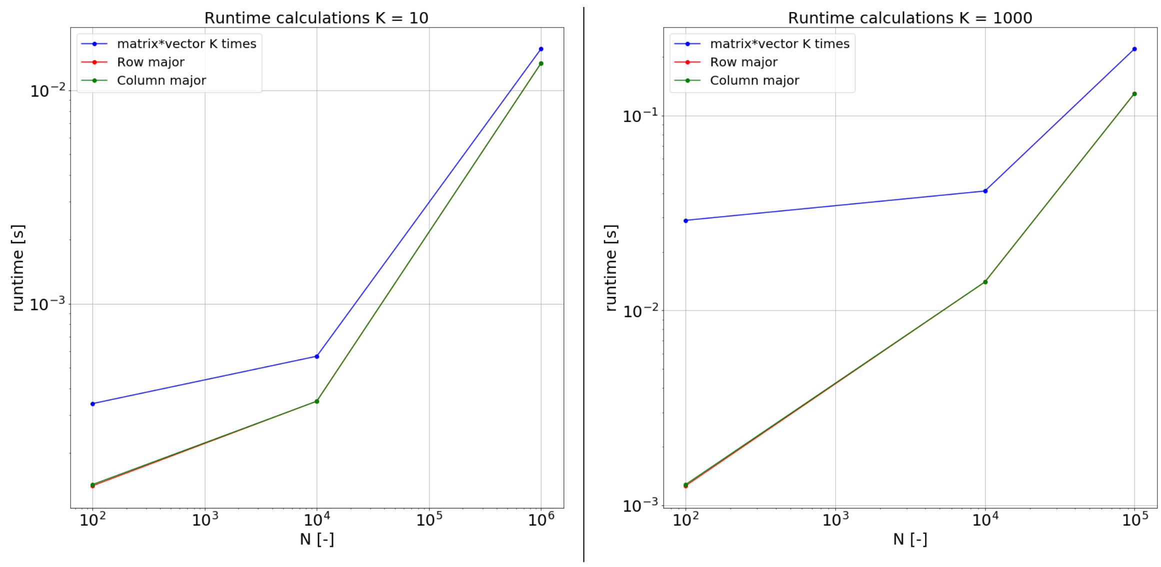


Figure 5: Runtimes of calculations for different N and K

Listing 6: Row- and Column major matrix matrix multiplication

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

```
__global__ void sparseDenseColumnMajor(int N, int K, 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) {

    for (int k = 0; k < K; k++) {
      double sum = 0;

      for (int jj = csr_rowoffsets[i]; jj < csr_rowoffsets[i + 1]; jj++) {
        sum += csr_values[jj] * x[csr_colindices[jj] + N*k];
      }

      y[i + N*k] = sum;
    }

  }
}
```

Listing 7: Code for the timing benchmark

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

// y = A * x
__global__ void sparseVector(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 sparseDenseRowMajor(int N, int K, 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) {

    for (int k = 0; k < K; k++) {
      double sum = 0;
```

```

        for (int jj = csr_rowoffsets[i]; jj < csr_rowoffsets[i + 1]; jj++) {
          sum += csr_values[jj] * x[csr_colindices[jj]*K + k];
        }

        y[i + N*k] = sum;
      }

  }
}


// Y = A * X
__global__ void sparseDenseColumnMajor(int N, int K, 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) {

    for (int k = 0; k < K; k++) {
      double sum = 0;

      for (int jj = csr_rowoffsets[i]; jj < csr_rowoffsets[i + 1]; jj++) {
        sum += csr_values[jj] * x[csr_colindices[jj] + N*k];
      }

      y[i + N*k] = sum;
    }

  }
}


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


int main() {

  Timer tim1, tim2, tim3;
  int N = 100000;
  int K = 1000;

  double *values = ( double *)malloc(sizeof( double) * N*10);
  int *colindices = (int*)malloc(sizeof(int) * N*10);
  int *offsets = (int*)malloc(sizeof(int) * (N+1));

  double *x = ( double *)malloc(sizeof( double) * N);
  double *X = (double*)malloc(sizeof(double) * N*K);

  double *y = ( double *)malloc(sizeof( double) * N);
```

```
double *Y = (double*)malloc(sizeof(double) * N*K);

values[0] = 2;
values[1] = 7;
values[2] = 4;
values[3] = 1;
values[4] = 9;
values[5] = 3;
values[6] = 6;
values[7] = 5;
values[8] = 8;

colindices[0] = 1;
colindices[1] = 3;
colindices[2] = 4;
colindices[3] = 2;
colindices[4] = 3;
colindices[5] = 0;
colindices[6] = 2;
colindices[7] = 4;
colindices[8] = 1;

offsets[0] = 0;
offsets[1] = 3;
offsets[2] = 5;
offsets[3] = 6;
offsets[4] = 8;
offsets[5] = 9;

x[0] = 1;
x[1] = 2;
x[2] = 3;
x[3] = 4;
x[4] = 5;

X[0] = 1;
X[1] = 1;
X[2] = 1;
X[3] = 2;
X[4] = 2;
X[5] = 2;
X[6] = 3;
X[7] = 3;
X[8] = 3;
X[9] = 4;
X[10] = 4;
X[11] = 4;
X[12] = 5;
X[13] = 5;
X[14] = 5;

double    *cuda_values;
int       *cuda_colindices;
int       *cuda_offsets;
double    *cuda_x;
double    *cuda_X;
double    *cuda_y;
```

```
double      *cuda_Y;

cudaMalloc(&cuda_values, sizeof(double) * N*10);
cudaMalloc(&cuda_colindices, sizeof(int) * N*10);
cudaMalloc(&cuda_offsets, sizeof(int) * (N+1));
cudaMalloc(&cuda_x, sizeof(double) * N);
cudaMalloc(&cuda_X, sizeof(double) * N*K);
cudaMalloc(&cuda_y, sizeof(double) * N);
cudaMalloc(&cuda_Y, sizeof(double) * N*K);

cudaMemcpy(cuda_values, values, sizeof(double) * N*10, cudaMemcpyHostToDevice);
cudaMemcpy(cuda_colindices, colindices, sizeof(int) * N*10, cudaMemcpyHostToDevice);
cudaMemcpy(cuda_offsets, offsets, sizeof(int) * (N+1), cudaMemcpyHostToDevice);
cudaMemcpy(cuda_x, x, sizeof(double) * N, cudaMemcpyHostToDevice);
cudaMemcpy(cuda_X, X, sizeof(double) * N*K, cudaMemcpyHostToDevice);

std::vector<double> timings1;
  for(int reps=0; reps < 10; ++reps) {
      tim1.reset();
      for (int j = 0; j < K; j++) {
        sparseVector<<<256, 256>>>(N, cuda_offsets, cuda_colindices, cuda_values,
        cuda_x, cuda_y);
        cudaMemcpy(y, cuda_y, sizeof(double) * N, cudaMemcpyDeviceToHost);
      }
      timings1.push_back(tim1.get());
  }
std::sort(timings1.begin(), timings1.end());
double time_elapsed1 = timings1[10/2];
std::cout << "Time elapsed vector: " << time_elapsed1 << std::endl << std::endl;






std::vector<double> timings2;
  for(int reps=0; reps < 10; ++reps) {
      tim2.reset();
      sparseDenseRowMajor<<<256, 256>>>(N, K, cuda_offsets, cuda_colindices,
      cuda_values, cuda_X, cuda_Y);
      cudaMemcpy(Y, cuda_Y, sizeof(double) * N*K, cudaMemcpyDeviceToHost);
      timings2.push_back(tim2.get());
  }
std::sort(timings2.begin(), timings2.end());
double time_elapsed2 = timings2[10/2];
std::cout << "Time elapsed row: " << time_elapsed2 << std::endl << std::endl;





std::vector<double> timings3;
for(int reps=0; reps < 10; ++reps) {
    tim3.reset();
    sparseDenseRowMajor<<<256, 256>>>(N, K, cuda_offsets, cuda_colindices,
    cuda_values, cuda_X, cuda_Y);
    cudaMemcpy(Y, cuda_Y, sizeof(double) * N*K, cudaMemcpyDeviceToHost);
```

```
        timings3.push_back(tim3.get());
    }
    std::sort(timings3.begin(), timings3.end());
    double time_elapsed3 = timings3[10/2];
    std::cout << "Time elapsed column: " << time_elapsed3 << std::endl << std::endl;




    free(x);
    free(X);
    free(y);
    free(Y);
    free(values);
    free(colindices);
    free(offsets);

    cudaFree(cuda_values);
    cudaFree(cuda_colindices);
    cudaFree(cuda_offsets);
    cudaFree(cuda_x);
    cudaFree(cuda_X);
    cudaFree(cuda_y);
    cudaFree(cuda_Y);


    return EXIT_SUCCESS;
}
```