Computational Science on Many-Core Architectures Exercise 6

Example 1 Dot Product with Warp Shuffels (4 Points)

First I want to test my implementation with a small test for a small vector.

Compile output

[Compilation successful]

Run output

N: 6
1 | 1 | 0 | 2 | 3 | -4 |
Sum of all elements: 3 Ref of Sum all elements: 3
1Norm: 11 Ref of 1Norm: 11
2Norm: 5.56776 Ref of 2Norm: 5.56776
MaxNorm: 4 Ref of MaxNorm: 4
minvalue: -4 Ref of minxalue: -4
maxvalue: 3 Ref of maxvalue: 3
nomOfzeros: 1 Ref of nomOfzeros: 1

a)

Listing 1: additional atomicMax

```
__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))));
        // atomicCAS returns the value that is stored in address AFTER the CAS
        // atomicCAS(a, b, c) --> return c
        //
    } while (assumed != old);
}
```

Listing 2: additional atomicMin

```
 __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))));
        // atomicCAS returns the value that is stored in address AFTER the CAS
```

```
11          // atomicCAS(a, b, c) --> return c
12          //
13      } while (assumed != old);
14  }
```

Listing 3: kernel for the seven operations

```
1   __global__ void mmsszz(double *x, double *dot, int N)
2   {
3       __shared__ double sumofallelements[BLOCK_SIZE];
4       __shared__ double einsnorm[BLOCK_SIZE];
5       __shared__ double zweisnorm[BLOCK_SIZE];
6       __shared__ double maxnorm[BLOCK_SIZE];
7       __shared__ double maxval[BLOCK_SIZE];
8       __shared__ double minval[BLOCK_SIZE];
9       __shared__ double zeros[BLOCK_SIZE];
10
11      if (blockDim.x * blockIdx.x < N)
12      {
13      unsigned int ind = threadIdx.x + blockDim.x*blockIdx.x;
14      unsigned int str = blockDim.x*gridDim.x;
15      double sum = 0;
16      double einssum = 0;
17      double zweissum = 0;
18      double max = x[0];
19      double min = max;
20      double count = 0;
21      while(ind < N)
22      {
23          for (int i = 0; i < N; i += str)
24          {
25              sum += x[i]; // sum of all entries
26              einssum += std::abs(x[i]);  // 1-norm
27              zweissum += x[i] * x[i];  // 2-norm
28              max = fmax(x[i], max);  // max_value
29              min = fmin(x[i], min);  // min_value
30              if (x[i] == 0)  // counts the zero entries
31              {
32                  count = count + 1;
33              }
34          }
35      ind += str;
36      }
37      sumofallelements[threadIdx.x] = sum;
38      einsnorm[threadIdx.x] = einssum;
39      zweisnorm[threadIdx.x] = zweissum;
40      maxnorm[threadIdx.x] = fmax(std::abs(min), max);
41      maxval[threadIdx.x] = max;
42      minval[threadIdx.x] = min;
43      zeros[threadIdx.x] = count;
44      __syncthreads();
```

```
45      for(int i = blockDim.x/2; i>0; i/=2)
46      {
47      __syncthreads();
48       if(threadIdx.x < i)
49       {
50          sumofallelements[threadIdx.x] += sumofallelements[threadIdx.x + i];
51          einsnorm[threadIdx.x] += einsnorm[threadIdx.x + i];
52          zweisnorm[threadIdx.x] += zweisnorm[threadIdx.x + i];
53          maxnorm[threadIdx.x] = fmax(maxnorm[threadIdx.x + i],maxnorm[threadIdx.x]
54          minval[threadIdx.x] = fmin(minval[threadIdx.x + i],minval[threadIdx.x]);
55          maxval[threadIdx.x] = fmax(maxval[threadIdx.x + i],maxval[threadIdx.x]);
56          zeros[threadIdx.x] += zeros[threadIdx.x + i];
57       }
58      }
59           if(threadIdx.x == 0)
60           {
61                atomicAdd(dot + 0,sumofallelements[0]);
62                atomicAdd(dot + 1,einsnorm[0]);
63                atomicAdd(dot + 2,std::sqrt(zweisnorm[0]));
64                atomicMax(dot + 3, maxnorm[0]);
65                atomicMin(dot + 4, minval[0]);
66                atomicMax(dot + 5, maxval[0]);
67                atomicAdd(dot + 6, zeros[0]);
68           }
69      }
70  }
```

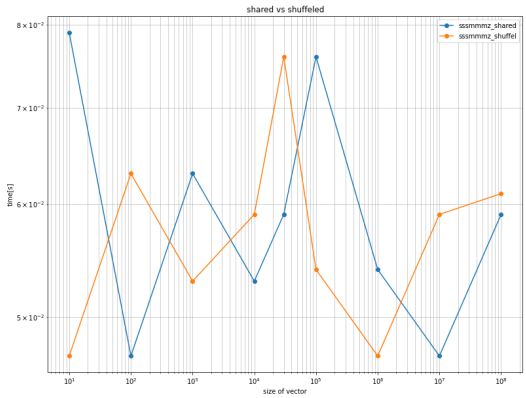
b)

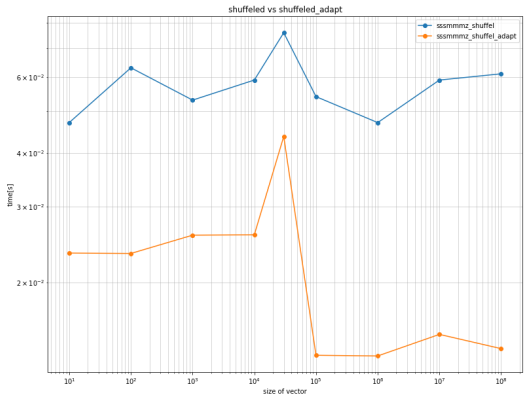
Listing 4: kernel for warp shuffeld

```
1   __global__ void analyze_x_warp(double *x, double *results, int N)
2   {
3       if (blockDim.x * blockIdx.x < N)
4       {
5           int tid = threadIdx.x + blockDim.x * blockIdx.x; // global tid
6           const int stride = blockDim.x * gridDim.x;
7
8           double sum = 0.0, abs_sum = 0.0, sqr_sum = 0.0;
9           // double mod_max = 0.0;
10          double max = x[0];
11          double min = max;
12          int z_entries = 0;
13          for (; tid < N; tid += stride)
14          {
15              double value = x[tid];
16              sum += value;
17              abs_sum += std::abs(value);
18              sqr_sum += value*value;
19
```

```
20              min = fmin(value, min);
21              max = fmax(value, max);
22              z_entries += (value)? 0 : 1;
23          }
24          tid = threadIdx.x; // block tid
25          double mod_max = fmax(std::abs(min), max);
26
27          __syncthreads();
28          for (int i = warpSize / 2; i != 0; i /= 2)
29          {
30              //__syncthreads();
31              sum += __shfl_down_sync(0xffffffff, sum, i);
32              abs_sum += __shfl_down_sync(0xffffffff, abs_sum, i);
33              sqr_sum += __shfl_down_sync(0xffffffff, sqr_sum, i);
34
35              double tmp = __shfl_down_sync(0xffffffff, mod_max, i);
36              mod_max = fmax(tmp, mod_max);
37              tmp = __shfl_down_sync(0xffffffff, min, i);
38              min = fmin(tmp, min);
39              tmp = __shfl_down_sync(0xffffffff, max, i);
40              max = fmax(tmp, max) ;
41
42              z_entries += __shfl_down_sync(0xffffffff, z_entries, i);
43          }
44          if (tid % warpSize == 0) // a block can consist of multiple warps
45          {
46              atomicAdd(results, sum);
47              atomicAdd(results+1, abs_sum);
48              atomicAdd(results+2, sqr_sum);
49
50              atomicMax(results+3, mod_max);
51              atomicMin(results+4, min);
52              atomicMax(results+5, max);
53
54              atomicAdd(results+6, z_entries);
55          }
56      }
57  }
```

c)



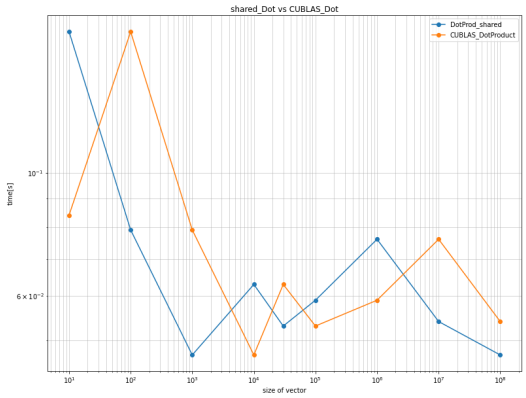


The "sssmmmz" means the 7 different functions that we have to program into only one kernel. There is a strange oscillation between the two. I tested it also with a multiple number of iterations and always take the mean vaule from that.

On the left side is the same function plotted but with the worp shuffle instead of the shared memory method. The blue line was tested with a fixed GRID_SIZE of 128 and a BLOCK_SIZE of 256. The orange line is the adapt one. It means that the GRID_SIZE is every step different:

$$GS = \frac{N}{256}$$

GS means the GRID_SIZE. Interesting is that this method is always faster in runtime than the other method.



For a small vector size the CUBLAS dot function is slightly slower than the shared one. For higher size vectors is it not clear where the winner is.

Example 2 Sparse Matrix Times Dense Matrix (3 Points + 1 Bonus)

a)

Use the same function from "poisson.hpp" to populate the matrix A.

Listing 5: Kernel for Column-Major

```
__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 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];
            }
        }
    }
}
```

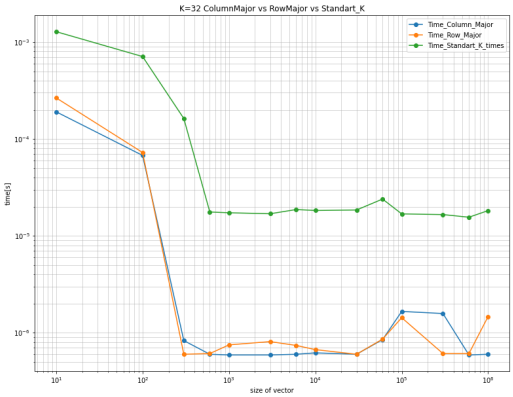
Listing 6: Kernel for Row-Major

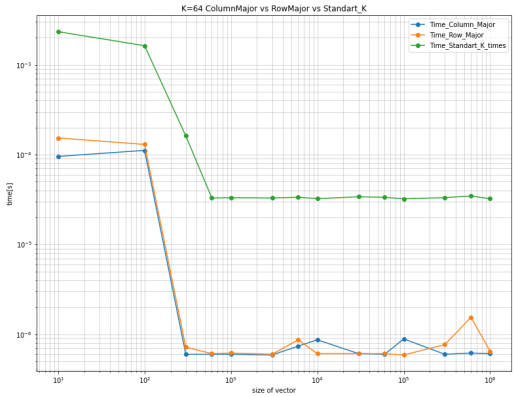
```
__global__ void A_MatMul_Xrm(int N, int K,
int *csr_rowoffsets, int *csr_colindices, double *csr_values,
double *X, double *Y)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < N)
    {
        int row_start = csr_rowoffsets[tid];
        int row_end = csr_rowoffsets[tid + 1];

        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;
        }
    }
}
```

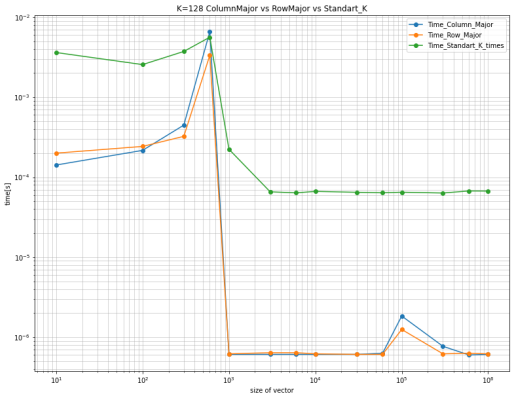
Listing 7: Kernel for Standart Matrix Vector

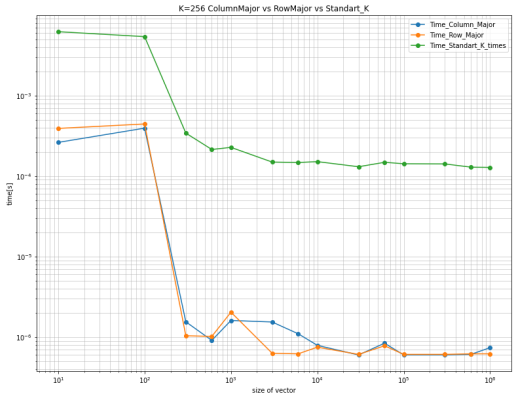
```
1  __global__ void cuda_csr_matvec_product(int N, int *csr_rowoffsets,
2  int *csr_colindices, double *csr_values,
3  double *x, double *y)
4  {
5      for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N;
6      i += blockDim.x * gridDim.x)
7      {
8          double sum = 0;
9          for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++)
10         {
11             sum += csr_values[k] * x[csr_colindices[k]];
12         }
13         y[i] = sum;
14     }
15 }
```

b) results with different N and K



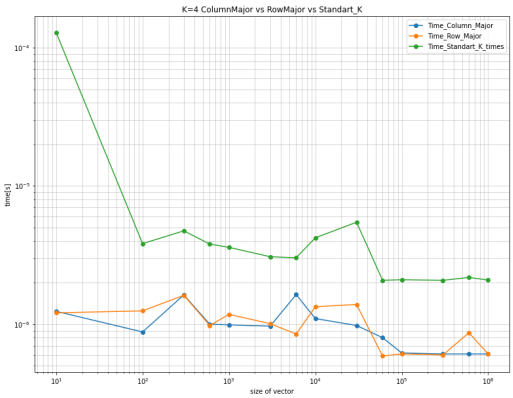


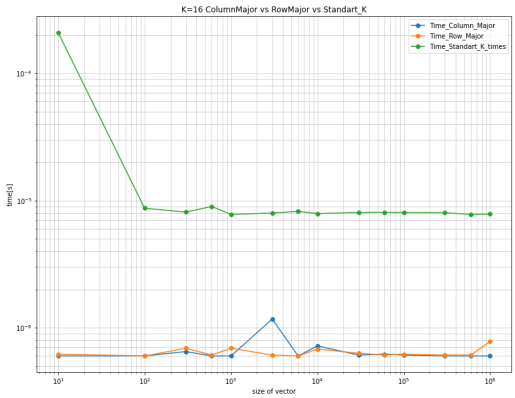


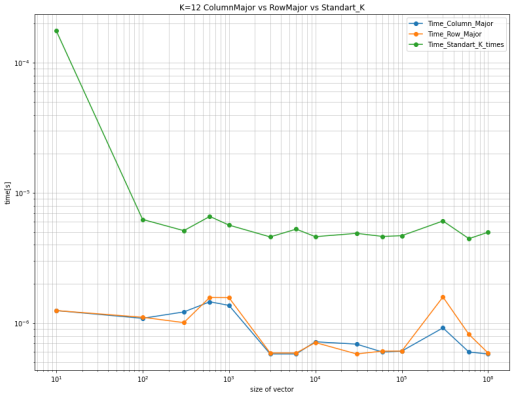


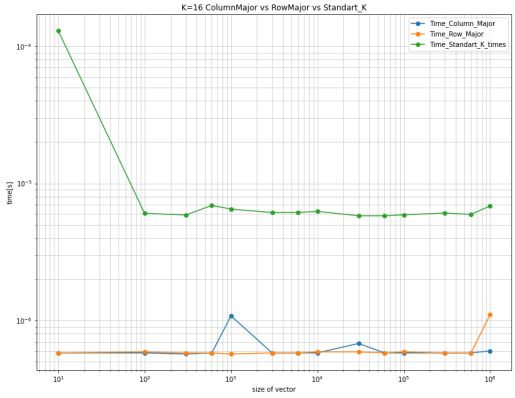
For increasing number of K the runtimes at a certain point $N = 600$ does not change much. For relatively small numbers of N the runtime is always larger than for large numbers of N. I expected the opposite case. For changing number of K the offset between the runtimes at about $N = 1000$

does not change much. There is no difference between the column-major and row-major for different number of N (rows) and number of K. The green line is always slower in time as the other two lines.









First I tested it with low number of K, $K = \{4, 8, 12, 16\}$. But there are basically no differences between the 4 graphs. I expect the runtime grows with the size N of the vector. Or that the row-major method is faster than the column-major method. Because C os more a row-major language instead of for example Fortran.