

Computational Science on Many-Core Architectures Exercise 3

Example 1 Strided and Offset Memory Access (2 Points)

a) stride

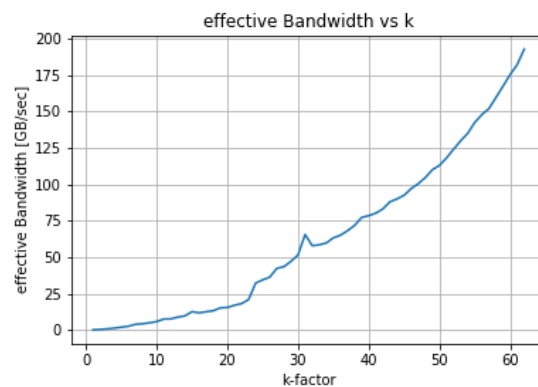
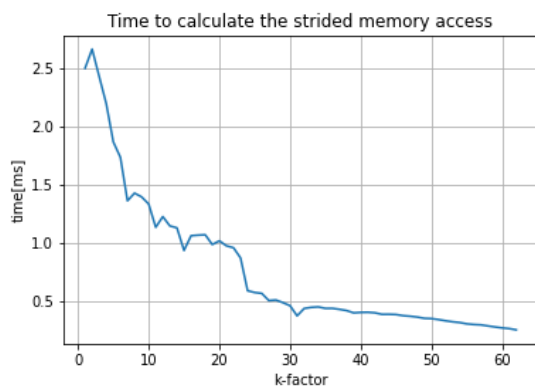
Listing 1: code for a)

```
1  #include <stdio.h>
2  #include "timer.hpp"
3  #include <vector>
4
5  __global__ void sumofVectors(double* x, double* y, double* z, int N, int k)
6  {
7      int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
8
9      for(int i = thread_id; i < N/k; i += blockDim.x * gridDim.x)
10     {
11         z[i*k] = x[i*k] + y[i*k];
12     }
13 }
14
15 int main(void)
16 {
17     int N = 10000000;
18     double *x, *y, *z, *d_x, *d_y, *d_z;
19     int anz = 10;
20     std::vector<double> results;
21
22     Timer timer;
23
24     x = new double [N];
25     y = new double [N];
26     z = new double [N];
27
28     for (int i = 0; i < N; i++)
29     {
30         x[i] = 1;
31         y[i] = 3;
32         z[i] = 0;
33     }
34
35     cudaMalloc(&d_x, N*sizeof(double));
36     cudaMalloc(&d_y, N*sizeof(double));
37     cudaMalloc(&d_z, N*sizeof(double));
38     cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
39     cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
40     cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);
41
```

```

42     for (int k = 1; k < 64; k++)
43     {
44         cudaDeviceSynchronize();
45         timer.reset();
46
47         for (int i = 0; i < anz; i++)
48         {
49             sumofVectors<<<265, 256>>>(d_x, d_y, d_z, N, k);
50         }
51
52         cudaDeviceSynchronize();
53         results.push_back(1000*timer.get()/anz);
54     }
55     for (int i = 0; i < 64; i++)
56     {
57         printf("%f, ", results[i]);
58     }
59
60     cudaFree(d_x);
61     cudaFree(d_y);
62     cudaFree(d_z);
63
64     delete x;
65     delete y;
66     delete z;
67
68     return EXIT_SUCCESS;
69 }

```



The tendency of the left graph is clear because if the k-factor rises the number of elements for the summation decreases and also the computing time as well. For the right graph I really do not know why the bandwidth increases rapidly when the number of k is also rising.

$$Bw = \frac{8 * N}{10^{-3} * t * k[s]}$$

were N is the number of array elements $N = 10^8$, 8 stands for 8 Bytes and for the right unit the given 10^x powers to convert the bandwidth to $\frac{GB}{sec}$.

b) offset

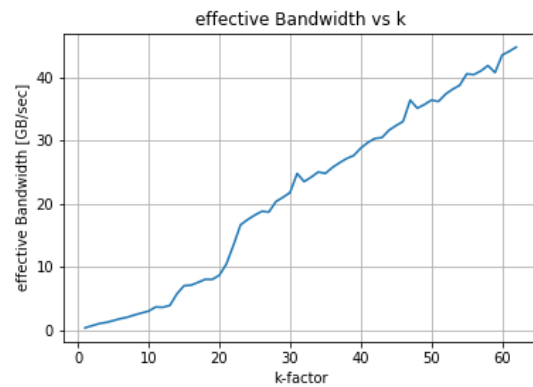
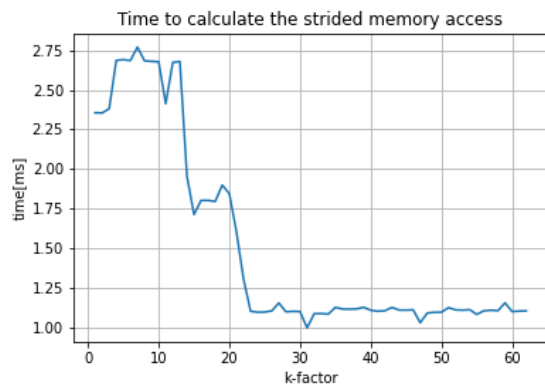
For the second part with the offset memory I only changed the kernel with the different setting.

Listing 2: code for a)

```

1  __global__ void sumofVectors(double* x, double* y, double* z, int N, int k)
2  {
3      int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
4
5      for(int i = thread_id; i < N-k; i += blockDim.x * gridDim.x)
6      {
7          z[i+k] = x[i+k] + y[i+k];
8      }
9  }

```



For the second part with the offset memory the runtime is not that different than the plot from the previous example but the effective bandwidth is slightly smaller.

Example 2 Conjugate Gradients (5 Points)

For the kernel for the matrix vector computation I used a code from the internet and slithly changed it to my porpus.

a) matrix-vector kernel

Listing 3: code for a)

```

1  __global__ void csr_matvec_product(size_t N ,
2  int *csr_rowoffsets , int *csr_colindices , double *csr_values ,
3  double *x, double *y)
4  {
5      int row = blockDim.x * blockIdx.x + threadIdx.x;
6      if(row < N )
7      {
8          float dot_Ax = 0;
9          int row_start = csr_rowoffsets[row];
10         int row_end     = csr_rowoffsets[row +1];
11
12         for (int jj = row_start; jj < row_end; jj++)
13         {
14             dot_Ax += csr_values[jj] * x[csr_colindices[jj]];
15         }
16         y[row] += dot_Ax;
17     }
18 }
```

b) dot-product kernel

For that part I used the code sniped from the previous exercise with the summation with the atomicstructure.

Listing 4: code for a)

```

1  __global__ void dot_pro(double *x, double *y, double *dot, unsigned int N)
2  {
3      unsigned int ind = threadIdx.x + blockDim.x*blockIdx.x;
4      unsigned int str = blockDim.x*gridDim.x;
5
6      __shared__ double cache[256];
7
8      double tmpsum = 0.0;
9      while(ind < N)
10     {
11         tmpsum += x[ind]*y[ind];
12         ind += str;
13     }
14
15     cache[threadIdx.x] = tmpsum;
16
17     __syncthreads();
```

```
18
19     for (int i = blockDim.x/2; i>0; i/=2)
20     {
21         __syncthreads();
22         if (threadIdx.x < i)
23         {
24             cache[threadIdx.x] += cache[threadIdx.x + i];
25         }
26     }
27
28     if (threadIdx.x == 0)
29     {
30         atomicAdd(&dot, cache[0]);
31     }
32 }
```

I simply add the coefficient alpha to the vector addition from the previous examples.

Listing 5: code for a)

```
1  __global__ void vec_plusmin_alpha_vector (double* x, double*y, double*z,
2  double alpha, int N)
3  {
4      int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
5
6      for (size_t i = thread_id; i < N; i += blockDim.x * gridDim.x)
7      {
8          z[i] = x[i] + alpha * y[i];
9      }
10 }
```
