Computational Science on Many-Core Architectures Exercise 3

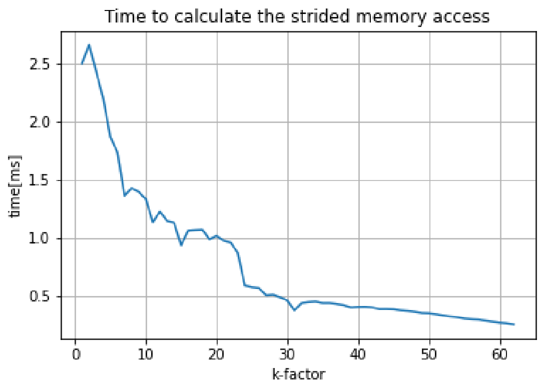
Example 1 Strided and Offset Memory Access (2 Points)

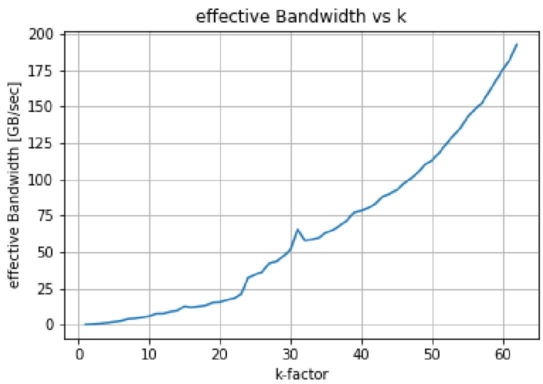
a) stride

Listing 1: code for 1a)

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

__global__ void sumofVectors(double* x, double* y, double* z, int N, int k)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    for(int i = thread_id; i < N/k; i += blockDim.x * gridDim.x)
    {
        z[i*k] = x[i*k] + y[i*k];
    }
}

int main(void)
{
    int N = 10000000;
    double *x, *y, *z, *d_x, *d_y, *d_z;
    int anz = 10;
    std::vector<double> results;

    Timer timer;

    x = new double [N];
    y = new double [N];
    z = new double [N];

    for (int i = 0; i < N; i++)
    {
        x[i] = 1;
        y[i] = 3;
        z[i] = 0;
    }

    cudaMalloc(&d_x, N*sizeof(double));
    cudaMalloc(&d_y, N*sizeof(double));
    cudaMalloc(&d_z, N*sizeof(double));
    cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);

```

```
    for (int k = 1; k < 64; k++)
    {
        cudaDeviceSynchronize();
        timer.reset();

        for (int i = 0; i < anz; i++)
        {
            sumofVectors<<<265, 256>>>(d_x, d_y, d_z, N, k);
        }

        cudaDeviceSynchronize();
        results.push_back(1000*timer.get()/anz);
    }
    for (int i = 0; i < 64; i++)
    {
        printf("%f, ",results[i]);
    }

    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_z);

    delete x;
    delete y;
    delete z;

    return EXIT_SUCCESS;
}
```





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.

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

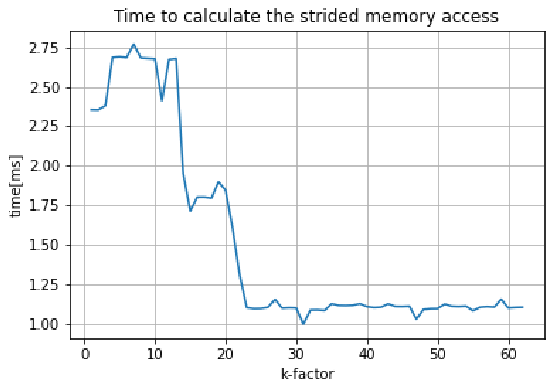
where N is the number of array elements $N = 10^8$, 8 stands for 8 Bytes and for the right unit the given 10^x powers to convert the bandwidth to $\frac{GB}{sec}$.

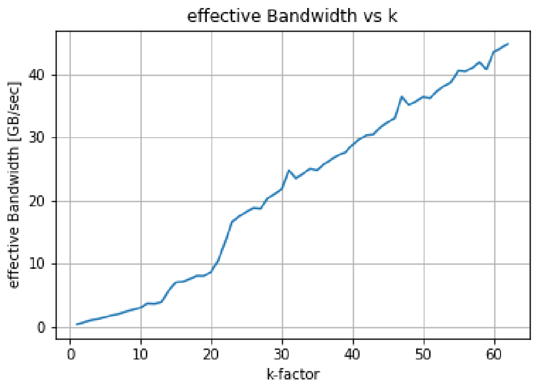
b) offset

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

Listing 2: code for 1b)

```
__global__ void sumofVectors(double* x, double* y, double* z, int N, int k)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    for(int i = thread_id; i < N-k; i += blockDim.x * gridDim.x)
    {
        z[i+k] = x[i+k] + y[i+k];
    }
}
```





For the second part with the offset memory the runtime is not that different from 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 slightly changed it to my purpose.

a) matrix-vector kernel

Listing 3: code for row 4 in the code

```
__global__ void csr_matvec_product(size_t N ,
int *csr_rowoffsets , int *csr_colindices , double *csr_values,
double *x, double *y)
{
    int  row = blockDim.x * blockIdx.x + threadIdx.x;
    if(row < N )
    {
        float  dot_Ax = 0;
        int  row_start = csr_rowoffsets[row];
        int  row_end   = csr_rowoffsets[row +1];

        for (int jj = row_start; jj < row_end; jj++)
        {
            dot_Ax += csr_values[jj] * x[csr_colindices[jj]];
        }
        y[row] += dot_Ax;
    }
}
```

b) dot-product kernel

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

Listing 4: code for row 5 and 9

```
__global__ void dot_pro(double *x, double *y, double *dot, unsigned int N)
{
    unsigned int ind = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int str = blockDim.x*gridDim.x;

    __shared__ double cache[256];

    double tmpsum = 0.0;
    while(ind < N)
    {
        tmpsum += x[ind]*y[ind];
        ind += str;
    }

    cache[threadIdx.x] = tmpsum;

    __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]);
    }
}
```

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

Listing 5: code for row 7, 8 and 12

```
__global__ void vec_plusmin_alpha_vector(double* x, double*y, double*z,
double alpha, int N)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    for (size_t i = thread_id; i < N; i += blockDim.x * gridDim.x)
    {
        z[i] = x[i] + alpha * y[i];
    }
}
```