360.252 - Computational Science on Many-Core Architectures

WS 2020 - Exercise 3

Christian Gollmann, 01435044

Last update: November 9, 2020

Contents

1 Strided Memory Access

I first wrote an implementation working on a small dataset to check if my kernel was operating correctly. Then I moved to the actual timing benchmark.

In both cases there can be seen a clear trend. In general it definitely makes sense that the runtime goes down when operating on less vector entries.

That the bandwidth goes down so drastically was something I didn't expect, but can also be explained. I therefore refer to an analogy when working on the CPU. There, the smallest amount of loadable memory is the cacheline. So even if only one 8 byte element is needed, the whole cacheline hast to be loaded. That is why from a performance point of view, it is so important to keep an eye on data locality.

In the example here, we destroy the data locality concept by leaving empty spaces in between our vector usage. That's why we are no longer using what we have very efficiently.

As a recommendation for more complex cases, I would say max out the use of data locality.

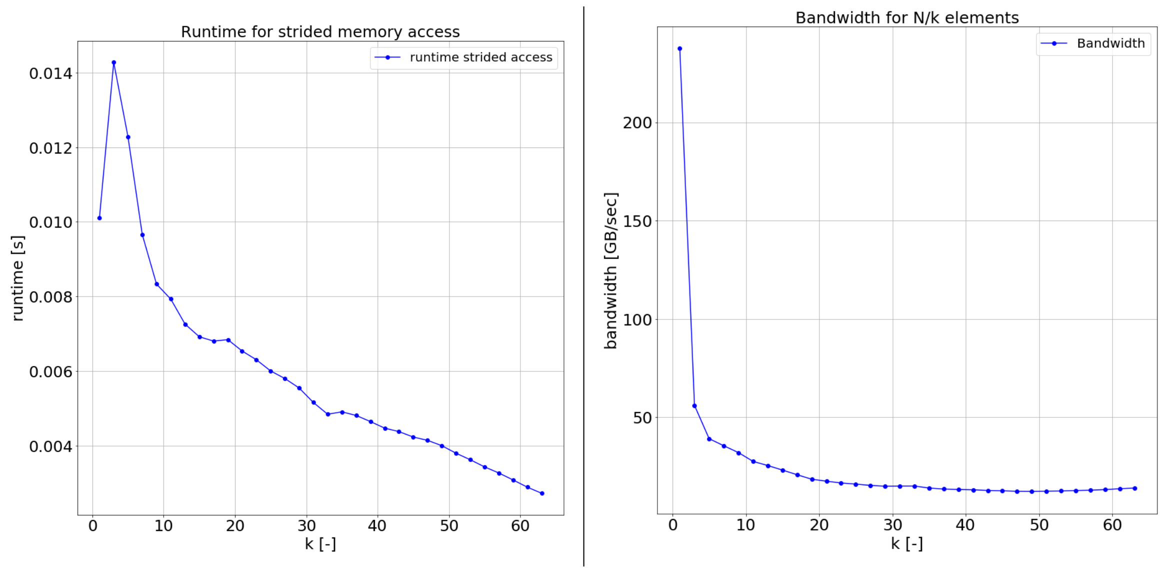

Figure 1: Runtime and bandwidth for strided memory access

Listing 1: Validation of Code for strided memory access

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


#define MAGIC_NUMBER 10

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

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

}

int main(void)
{
    int N = 100;
    int k = 20;

    double*x, *y, *z, *d_x, *d_y, *d_z;
    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] = 2;
        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);

    cudaDeviceSynchronize();
    timer.reset();
    std::vector<double> timings;

    for(int reps=0; reps < MAGIC_NUMBER; ++reps)
    {
        sumVectors<<<256, 256>>>(d_x, d_y, d_z, N, k);
        cudaDeviceSynchronize();
        timings.push_back(timer.get());
    }
    std::sort(timings.begin(), timings.end());
    double time_elapsed = timings[MAGIC_NUMBER/2];

    cudaMemcpy(z, d_z, N*sizeof(double), cudaMemcpyDeviceToHost);

    printf("Addition took %g seconds", time_elapsed);

    std::cout << std::endl << "z[0] = " << z[0] << std::endl;
    std::cout << "z[1] = " << z[1] << std::endl;
    std::cout << "z[k] = " << z[k] << std::endl;
    std::cout << "z[2*k] = " << z[2*k-1] << std::endl;
    std::cout << "z[2*k+1] = " << z[2*k-1+1] << std::endl;

    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_z);
    delete x;
    delete y;
    delete z;

    return EXIT_SUCCESS;
}
```

Listing 2: Actual Code used in benchmark

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


#define MAGIC_NUMBER 10

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

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

}

int main(void)
{
    int N = 100000000;

    double*x, *y, *z, *d_x, *d_y, *d_z;
    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] = 2;
        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 += 2)
    {
        cudaDeviceSynchronize();
        std::vector<double> timings;

        for(int reps=0; reps < MAGIC_NUMBER; ++reps)
        {
            timer.reset();
            sumVectors<<<256, 256>>>(d_x, d_y, d_z, N, k);
            cudaDeviceSynchronize();
            timings.push_back(timer.get());
        }

        std::sort(timings.begin(), timings.end());
        double time_elapsed = timings[MAGIC_NUMBER/2];

        std::cout << time_elapsed << std::endl;
    }

    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_z);
    delete x;
    delete y;
    delete z;

    return EXIT_SUCCESS;
}
```

2 Offset Memory Access

Two things can be recognized from the following plot:

1. The runtime, and so the bandwidth, are ”good” whenever k equals a multiple of sixteen (0, 16, 32, ...).
2. A slow runtime does not really depend on k, eg. it does not matter whether we skip 3 or 15 entries.

I strongly suspect that the GPU’s cacheline size is 8*16 = 128 bytes. If, let’s say, we skip the first 5 entries, we don’t use the cacheline at its most efficiency and therefore have some kind of cache misses. Those cache misses then influence the achievable bandwidth. I’m not completely sure how all of this ties together, but I’m positive that it origins from the cacheline concept.

For future implementations, I suggest always using a ”full” cacheline if possible, thereby referring to the principle of data locality from before.

As an update, after completing the exercises, I found out more about Global Memory Coalescing on https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/. There I read that during execution, threads are grouped into warps whose warp size is 32 threads. On the webpage there is also more or less the same example we are supposed to do here. I was happy to see, that I was able to reproduce the results and that my assumptions more or less aligned with what was explained there. As I don’t intend to just reproduce the content there, I decided to stick with my initial assumptions and give this finding now as a reference.

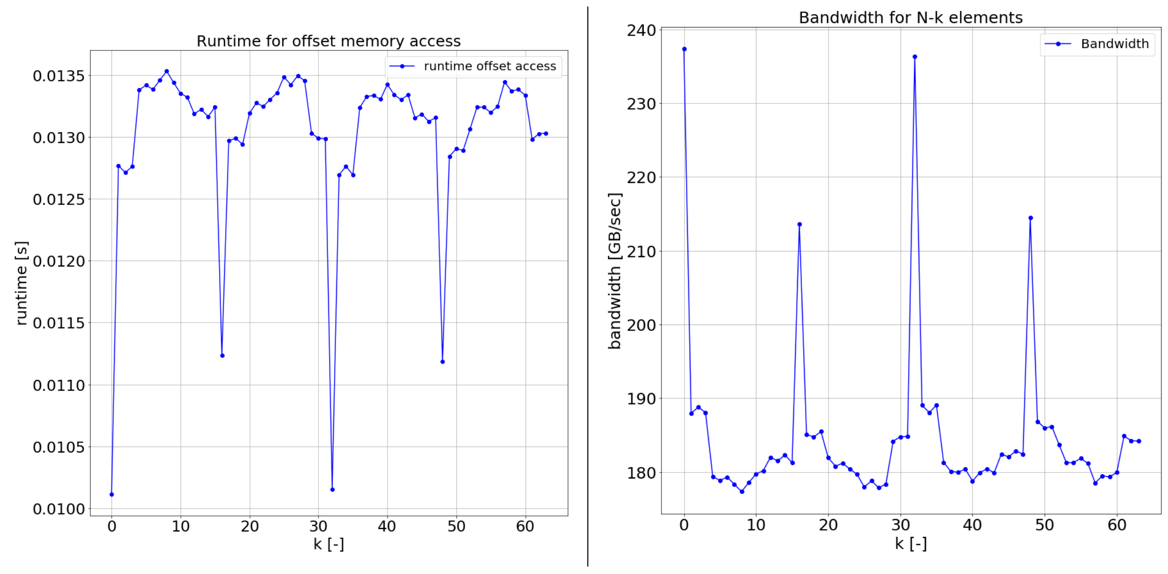

Figure 2: Runtime and bandwidth for offset memory access

Listing 3: Code for offset access

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

```

```

#define MAGIC_NUMBER 10

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

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

}

int main(void)
{
    int N = 100000000;

    double*x, *y, *z, *d_x, *d_y, *d_z;
    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] = 2;
        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 = 0; k < 64; k += 1)
    {
        cudaDeviceSynchronize();
        std::vector<double> timings;

        for(int reps=0; reps < MAGIC_NUMBER; ++reps)
        {
            timer.reset();
            sumVectors<<<256, 256>>>(d_x, d_y, d_z, N, k);
            cudaDeviceSynchronize();
            timings.push_back(timer.get());
        }

        std::sort(timings.begin(), timings.end());
        double time_elapsed = timings[MAGIC_NUMBER/2];

        std::cout << time_elapsed << std::endl;
    }

    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_z);
    delete x;
    delete y;
    delete z;

    return EXIT_SUCCESS;
}
```

3 Conjugate Gradient - matrix vector product

Here I went with the example from the lecture and corrected it slightly because I think there was a math typo in line 9. But I also took much inspiration from https://medium.com/analytics-vidhya/sparse-matrix-vector-multiplication-with-cuda-42d191878e8f which turned out to be a good reference also for later and more sophisticated implementations.

Listing 4: CUDA kernel for calculating sparse matrix vector multiplication

```
 1  __global__ void csr_matvec(int N, int *rowoffsets, int *colindices, double *values,
 2  double const *x, double *y)
 3  {
 4      for (int row = blockDim.x * blockIdx.x + threadIdx.x;
 5          row < N;
 6          row += gridDim.x * blockDim.x)
 7      {
 8        double val = 0;
 9        for (int jj = rowoffsets[row]; jj < rowoffsets[row+1]; ++jj)
10        {
11            val += values[jj] * x[colindices[jj]];
12        }
13        y[row] = val;
14      }
15  }
```

4 Conjugate Gradient - further kernels

For the following two kernels, I could build upon code I had already written for previous exercises and tweaked it slightly.

Listing 5: kernel for the vector operations in lines 5 and 9

```
__global__ void dot_product(double *x, double *y, double *dot, unsigned int n)
{
    unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int stride = blockDim.x*gridDim.x;

    __shared__ double cache[256];

    double temp = 0.0;
    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 6: kernel for the vector operations in lines 7, 8 and 12

```
__global__ void vector_plus_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];

}
```