Computational Science on Many-Core Architectures Exercise 7

Example 1 Dot Product with OpenCL (4 Points)

a)

Listing 1: opencl kernel

```
const char *my_opencl_program = ""
"__kernel void vec_mult(__global double *x,\n"
"                       __global double *y,\n"
"                       __global double *result,\n"
"                       unsigned int N\n)"
"{\n"
"  for (unsigned int i  = get_global_id(0);\n"
"                    i  < N;\n"
"                    i += get_global_size(0))\n"
"    result[i] = x[i] * y[i];\n"
"}";
```

This opencl kernel only multiplies the single entries and stored the results in the result vector.

Listing 2: Benchmark for the opencl

```
for (int i = 0;i < anz; i++)
{
    //
    // Set kernel arguments:
    //
    timer.reset();
    err = clSetKernelArg(my_kernel, 0, sizeof(cl_mem),
    (void*)&ocl_x); OPENCL_ERR_CHECK(err);

    err = clSetKernelArg(my_kernel, 1, sizeof(cl_mem),
    (void*)&ocl_y); OPENCL_ERR_CHECK(err);

    err = clSetKernelArg(my_kernel, 2, sizeof(cl_mem),
    (void*)&ocl_result); OPENCL_ERR_CHECK(err);

    err = clSetKernelArg(my_kernel, 3, sizeof(cl_uint),
    (void*)&vector_size); OPENCL_ERR_CHECK(err);

    //
    // Enqueue kernel in command queue:
    //
    err = clEnqueueNDRangeKernel(my_queue, my_kernel, 1,
   NULL, &global_size, &local_size, 0, NULL, NULL); OPENCL_ERR_CHECK(err);

    // wait for all operations in queue to finish:
    err = clFinish(my_queue); OPENCL_ERR_CHECK(err);
```

```
27 }
28 opencl_time = timer.get();
29 std::cout << "Time for opencl kernel: " << opencl_time/anz << std::endl;
```

The for loop is only for the benchmark and $anz = 100$. I only add the line 13 and 14 to set the buffer for the result vector.

Listing 3: CUDA kernel

```
__global__ void GPU_dot (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;

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

Listing 4: CUDA benchmark

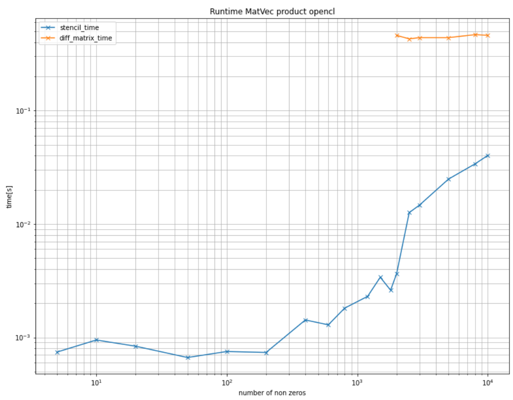
```
timer.reset();
for (int i = 0;i < anz; i++)
{
    GPU_dot<<<128, 128>>>(cuda_X, cuda_Y, cuda_dot_GPU, N);
    cudaDeviceSynchronize();
    cudaMemcpy(dot_GPU, cuda_dot_GPU, sizeof(double), cudaMemcpyDeviceToHost);
}
//cudaMemcpy(dot_GPU, cuda_dot_GPU, sizeof(double), cudaMemcpyDeviceToHost);
GPU_time = timer.get();
std::cout << "Time for GPU kernel: " << GPU_time/anz << std::endl;
```

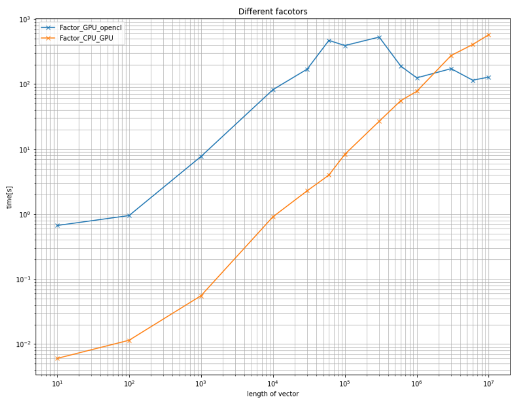
Listing 5: CPU kernel

```
void CPU_dot(ScalarType *x, ScalarType *y, ScalarType *result,
      unsigned int N)
{
    for (int i = 0; i < N; i++)
    {
        result[i] = x[i] * y[i];
    }
}
```

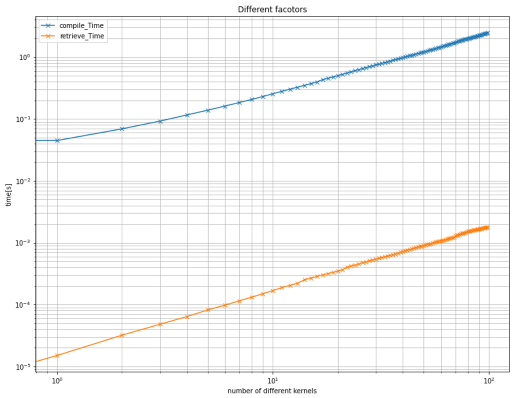
Listing 6: CPU benchmark

```
timer.reset();
for (int i = 0;i < anz; i++)
{
    CPU_dot(X, Y, dot_CPU, N);
}
CPU_time = timer.get();
std::cout << "Time for CPU kernel: " << CPU_time/anz << std::endl;
```





On the left we could see that the different runtimes for the same number of entries of a vector and apply the dot-product on it. For the GPU-times it is not clear why the times are like this. I also switch the mode of the machine to the GTX 1080. On the right side I plotted the different factors which one is faster opencl or CPU and GPU or CPU. At a certain point opencl is over 200 times faster than the CPU one.



The time to create the kernels is linear with the numbers of M different kernels and also the time to retrieve it. The second case is much faster than the fist.
For 1.4 I tried first to write a routine which adds a line with respect to a given number of M to create every time a different kernel and to measure the time. I tried this with a string implementation. That does not work probably because the declaration at the beginning of the provided code is a *const char* and I got errors that I could not change a *const char* for that. Then I tried to implement it with *char* datatyps and that also does not work. So I wrote 100 different kernels with if conditions and add a line "d = 0;" which does not affect the implementation from the dot product, and so I measure the time. My code is 4500 lines long so don't afraid I could not find a better solution.

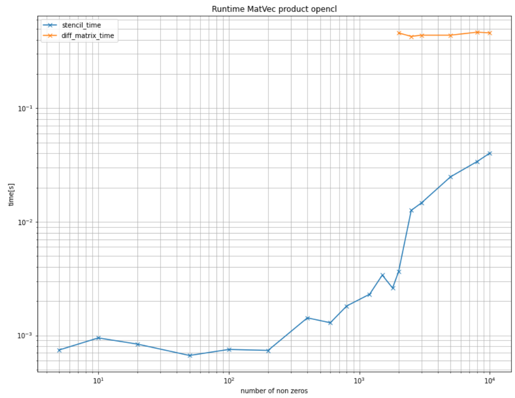
Example 2 Opencl Dot-product (3 Points)

opencl kernel

Listing 7: kernel for sparce matrix prod

```
const char *my_opencl_program = ""
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n"
"__kernel void csr_matvec_product_opencl(unsigned int N,\n"
"                                        __global int *csr_rowoffsets,\n"
"                                        __global int  *csr_colindices,\n"
"                                        __global double *csr_values,\n"
"                                        __global double *x,\n"
"                                        __global double *y)\n"
"{\n"
"  for (unsigned int i  = get_global_id(0);\n"
"                    i  < N;\n"
"                    i += get_global_size(0))\n"
"   {\n"
"          double value = 0;\n"
"          for (int j=csr_rowoffsets[i]; j<csr_rowoffsets[i+1]; ++j)\n"
"          value += csr_values[j] * x[csr_colindices[j]];\n"
"                                                       \n"
"          y[i] = value;\n"
"    }\n"
"}";
```

Benchmark



I do not know how to implement the second kernel so I benchmarked only the form point 1 with different max numbers of zero and a fixed number of 100×100 unknowns. I also had an SEGMENTATION FAULT for the second strategy to initialize the matrix that's the reason why I started the benchmark at 2000 (orange line). I also include a function to control if the solution is correct.