Introduction to High-Performance Computing, Winter 2021

Exercise 6: GPUs

Problem 1 (GPU Performance Modeling)**.** Given are the following hardware architectures:

2-socket Intel Skylake SP architecture with the following characteristics (cf. ex 2):

- 48 cores in total
- 2.1 GHz clock frequency
- 2 fused multiply add (FMA) units per core (each unit: 1 ADD & 1 MULT per cycle)
- FMA units operate on AVX-512 registers (512 bits)
- Cache sizes: L1 $\widehat{=}$ 32 KB, L2 $\widehat{=}$ 1024 KB, L3 $\widehat{=}$ 33 MB
- Sustainable main memory bandwidth gained by Stream benchmark: 170 GB/s

GPU-based system with the following characteristics:

- 80 streaming multiprocessors (SM)
- 32 (DP) cores per SM
- 1.4 GHz clock frequency
- 1 fused multiply add (FMA) unit per core (1 ADD & 1 MULT per cycle)
- Cache sizes: L1 $\widehat{=}$ 128 KB/SM, L2 $\widehat{=}$ 6 MB/SM
- Memory sizes: shared memory $\widehat{=}$ 128 KB/SM, global memory $\widehat{=}$ 16 GB
- Global memory bandwidth (peak): 840 GB/s

The GPU is attached to the CPU as follows:

- PCIe with a peak bandwidth of 16 GB/s
- Latencies: host-to-device $\widehat{=}$ 2 μs, device-to-host $\widehat{=}$ 3 μs

Furthermore, the following code snippet is given

Listing 6.1: Code snippet

```
#define N 100000
double a[N];
double b[N];
double c[N];
double s;

// init a, b, c, and s...

// compute a
for (int i=0; i<N; ++i) {
  a[i] = b[i] + s * c[i];
}
```

Problem 1.1. Kernel Execution Time
The code given in Listing 6.1 shall be offloaded to the GPU. Determine the potential kernel execution time t_{kernel}. In addition, state whether this kernel is compute bound or memory bound on the GPU.

Problem 1.2. Data Transfer Time
Determine the potential data transfer time t_{data} that is needed to move data between CPU and GPU (and vice versa).

Problem 1.3. GPU Runtime
Compute the complete GPU runtime t_{GPU}. Is it beneficial to run this code on the GPU (in comparison to running it on the CPU)?

Problem 1.4. GPU Tuning Impact
If you consider overlapping kernel computations and data transfers, what would be the potential GPU runtime $t_{GPU\_Overlap}$ using 4 streams? What would be the speedup over the synchronous execution t_{GPU}?

Problem 2 (GPU Occupancy)**.** Given is an NVIDIA GPU of compute capability 7.0. Its technical specification can be found under `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications`.
Furthermore, a GPU kernel is given with the following characteristics:

- Launch configuration: 128 threads per block
- Each thread block uses 8 KB of shared memory
- 8192 32-bit registers for each thread block (evenly distributed across threads within block)

What is the limiting factor here and why? To answer this question, use the technical specification given above and compute the occupancies (per SM) for the different limiters. Compare the limiters for warps (block size), shared memory and registers.
Hint: Here 1 K = 1024

Problem 3 (OpenMP Target Code and GPU Performance Tuning)**.** Consider the following small problem described in figure 6.1 and listing 6.2: Assume we have a rack in the machine hall that consists of two parts (A and B) that both have their own width (see lines 4 and 5). In a new setup, all racks in the hall shall have the doubled width (i.e. doubling both rack parts). For the examination whether the new setup will still fit into the machine hall, we double the width for all racks and offload these computations to the GPU (see line 12).

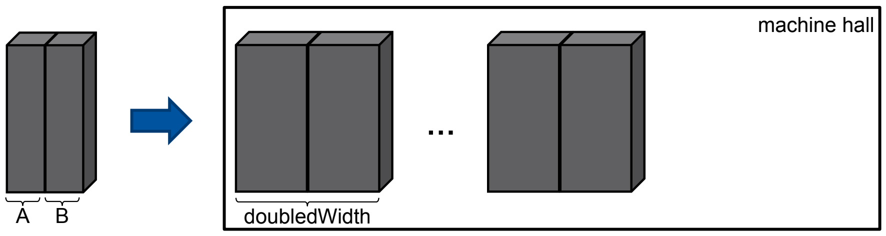

Figure 6.1: Racks placed in a machine hall

Problem 3.1. Complete the OpenMP Target code
Complete the OpenMP Target code in listing 6.2 by filling the gaps on this exercise sheet or on the code template in *exercise06.tar.gz*.

Problem 3.2. Performance Tuning
The program in listing 6.2 has a performance issue when looking at the GPU kernel *doubleTheWidth(...)*. What is this performance problem? How can you solve it?
Hint: Express the problem in terms of memory bus utilization of the GPU (assume a GPU NVIDIA Volta architecture and a 128 B cache line). The data resides on the GPU during the kernel execution.
Re-define the `struct` correspondingly.

Listing 6.2: OpenMP Target code for rack problem

```
#define N 67107840

struct rack_t {
    float widthA;
    float widthB;
    float doubledWidth;
};

static void initRacks(rack_t *racks, int n); // given init function

// GPU kernel
void doubleTheWidth(struct rack_t *racks, int n)
{


    for (int i = 0; i < n; i++) {


        racks[i].doubledWidth = 2 * (racks[i].widthA + racks[i].widthB);


    }


}

int main(int argc, char** argv)
{
    // define variables, pointers
    const int n = N;

    struct rack_t *racks = 0;
    racks = (struct rack_t *)malloc(n * sizeof(struct rack_t));

    initRacks(racks, n); // init racks struct w/ value

    // TODO: Offload to GPU


    doubleTheWidth(racks, n);


    // free memory
    free(racks);
    return 0;
}
```

Problem 4 (Jacobi - Parallelize the Code with OpenMP)**.** This exercise may be done with pen and paper (see listing 6.3) or by using the C code template in *exercise06.tar.gz*. If you choose pen and paper, you can use the given binaries to evaluate the performance.

During the following exercises, you will port a Jacobi solver to the GPU with OpenMP Target constructs. This Jacobi example solves a finite difference discretization using a 5-point-stencil (similar to exercise05) of the 2D Laplace equation

$$\Delta u(x, y) = \nabla^2 u(x, y) = 0$$

using the Jacobi iterative method. To this end, the Jacobi method starts with an approximation of the objective function $u(x, y)$ and reuses formerly-computed matrix elements to solve the current one (see Figure 6.2). It iterates the inner elements of the 2D-grid (see Figure 6.3) such that the boundary elements are only used within the stencil given by:

$$u_{k+1}(i, j) = \frac{u_k(i-1, j) + u_k(i+1, j) + u_k(i, j-1) + u_k(i, j+1)}{4}.$$

Here, i and j denote the indices within the matrix and k is the time step of the iterative solution. The solving process is aborted if the computed approximation is close to the solution or a certain maximal number (here 20) of iterations is achieved. The first one is true if the biggest change on any matrix element is smaller than a particular tolerance value.

Problem 4.1. Prepare your GPU cluster environment
Login to one of the frontend nodes of the RWTH Compute Cluster (e.g. login18-1.hpc.itc.rwth-aachen.de) and then jump per

```
ssh -Y login18-g-1
```

to the frontend node of the GPU Cluster (see `https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/a1beccd9e9dc4044a740ed248f478839` for further information). Next, download the Jacobi source files from RWTHmoodle (*exercise06.tar.gz*) and put them into your home directory. We provide a Makefile that allows easy execution of the Jacobi program. Available targets are:

- make help: Get help information
- make [release]: Compile the code
- run [dev=<deviceID> | notify=1 | time=1]: Run the code (with options)
- make clean: Clean the directory

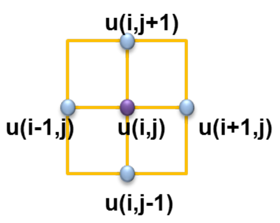

Figure 6.2: 5-point-stencil

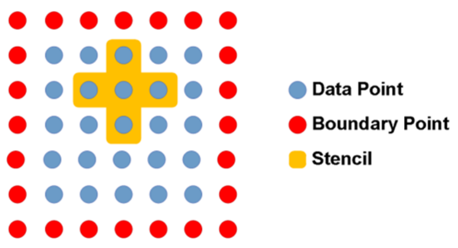

Figure 6.3: Boundaries on matrix

Finally, switch the compiler to Clang, as this is the compiler that supports OpenMP directives and load the CUDA library:

```
module switch intel clang/12
module load cuda/10.1
```

Before you start programming GPGPUs, it is always a good idea to examine your actual GPU hardware. To this end, execute the command: `nvidia-smi -q`
If everything works properly, you will get a list of the most important features of your GPU. Complete Table 6.1 with the Cluster GPU details.

Table 6.1: Details of RWTH GPUs

Feature	Value
Number and name of devices	
Number of SMs and cores	
CUDA compute capability (cc)	

Problem 4.2. First Jacobi Parallelization with OpenMP on GPUs

a) Use the OpenMP target directives to parallelize the first Jacobi loop (line 31). Make sure that you share the work of the loops among GPU threads.

b) Ensure that your solution is correct. Hint: Especially check the error variable for correctness issues when parallelizing your code.

c) Run your code or the given binary `jacobi_task4.2.exe`. How fast does this version execute? Write down the runtime in Table 6.2. Compare this runtime to the given ones.

Table 6.2: Runtimes of different Jacobi versions

Version	Total runtime[s]
Serial (Clang compiler)[1]	4.88
OpenMP (Clang compiler)[2]	0.61
Basic (Task 4.2)[3]	
Data (Task 4.4)[3]	
Loop (Task 4.5)[3]	

Problem 4.3. Code Profiling
Use profiling tools to investigate the performance of your code/ the binary `jacobi_task4.2.exe`.

a) The nvprof profiler enables a simple way to get timing information by prepending the profiler in front of the application: `nvprof ./jacobitask4.2.exe`. How much time was spent executing the kernel and data transfers?

[1]2x Intel Xeon Platinum 8160 CPU @ 2.1 GHz, 1 core, OMP_PROC_BIND=true
[2]2x Intel Xeon Platinum 8160 CPU @ 2.1 GHz, 48 cores in total, OMP_PROC_BIND=true
[3]1x NVIDIA Volta V100

b) Another way to analyze the performance of your code is NVIDIA's Visual Profiler that ships with the CUDA toolkit. It provides a graphical user interface and more detailed information on kernel executions.

   a) First, start the Visual Profiler with the command `nvvp`. Then, create a new session, choose your executable file, and start the profile. Note, you need to forward your X11-session with `ssh -Y login18-g-1` for graphical applications to work.

   b) In the left profiler pane, click on "MemCpy(HtoD)" and "MemCpy(DtoH)". Now, you can see the duration of the Memcpy command on the right hand side in the tab "Properties". If you click on the kernel that is listed under "Compute" (left pane), the kernel duration is displayed in the properties tab as well.

   c) In the timeline, can you see where data is moved between host and device? It might be necessary to zoom into the timeline (CTRL + mouse). When do we want to have the data copied between host and device?

Problem 4.4. Data Transfers
As you have seen in the previous task, the data transfers consume a lot of time (more than the compute part). The following tuning activity is the reduction of unneeded data movements.

a) To keep the data on the GPU device, you should offload all loops to the GPU that uses this data. Thus, parallelize the copy loop (line 44) in the code first.

b) Use a data region to remove the excess of data transfers.

c) How fast is your program now (use binary `jacobi_task4.4.exe` if you used pen and paper)? Write down the runtime in Table 6.2.

d) Check the changes by profiling the code again.

Problem 4.5. Loop Scheduling
The two main loops both contain nested loops not exposed so far.

a) Optimize their loop schedule to leverage additional parallelism.

b) How fast is your program now (use binary `jacobi_task4.5.exe` if you used pen and paper)? Write down the runtime in Table 6.2.

Discussion of this exercise on January 28, 2022.

Listing 6.3: Serial version of the Jacobi Solver

```
int main(int argc, char **argv)
{
    const int n = 4096, m = 4096;
    const int iter_max = 20;
    const double tol = 1.0e-6;
    double err  = 1.0;

    // Initialize arrays
    memset(U, 0, n * m * sizeof(double));
    memset(Unew, 0, n * m * sizeof(double));
    for (int i = 0; i < n; i++)
    {
        U[0][i]    = -1.0;
        Unew[0][i] = -1.0;
    }

    double runtime = GetRealTime();
    int iter = 0;


    // while solution is not accurate enough (or max iteration number is reached)

    while (err > tol \&\& iter < iter_max)
    {
        err = 0.0;

        // compute stencil and the error value
        // that denotes whether approximation is
        // close to solution

        for( int i = 1; i < n-1; i++ )
        {
            for( int j = 1; j < m-1; j++ )
            {
                Unew[i][j] = 0.25 * ( U[i][j+1] + U[i][j-1]
                                    + U[i-1][j] + U[i+1][j]);
                err = fmax(err, fabs(Unew[i][j] - U[i][j]));
            }
        }


        // Copy new solution into old one

        for( int i = 1; i < n-1; i++ )
        {
            for( int j = 1; j < m-1; j++ )
            {
                U[i][j] = Unew[i][j];
            }
        }

        iter++;

    } // end while

    runtime = GetRealTime() - runtime;

    printf("Time Elapsed: \%f s\n", runtime);

    return 0;
}
```