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 (CUDA 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 5 and 6). 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 13).

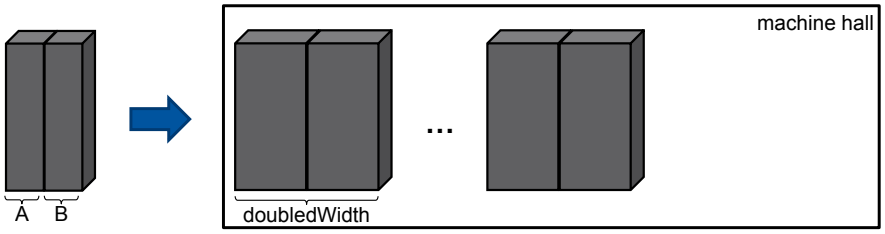

Figure 6.1: Racks placed in a machine hall

Problem 3.1. Complete the CUDA code
As you can see, the source code listing 6.2 contains five todos. Complete the CUDA code 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: CUDA code for rack problem

```
1  #define N 67107840
2  #define THREADSPERBLOCK 1024
3
4  struct rack_t {
5      float widthA;
6      float widthB;
7      float doubledWidth;
8  };
9
10 static void initRacks(rack_t *racks, int n); // given init function
11
12 // GPU kernel
13 __global__ void doubleTheWidth(rack_t *racks, int n)
14 {
15     int tid = blockDim.x * blockIdx.x + threadIdx.x;
16     if (tid < n)
17     {
18       // Compute the doubled width for each rack element
19       racks[tid].doubledWidth = 2 * (racks[tid].widthA + racks[tid].widthB);
20     }
21 }
22
23 int main(int argc, char** argv)
24 {
25     // define variables, pointers
26     const int n = N;
27     rack_t *h_racks = 0;
28     rack_t *d_racks = 0;
29     h_racks = (rack_t*) malloc(n*sizeof(rack_t));
30     initRacks(h_racks,n); // init racks struct w/ values
31
32     // TODO 1: allocate memory on GPU
33
34
35
36
37     // TODO 2: copy initialized data from CPU to GPU
38
39
40
41
42     dim3 threads_per_block(THREADSPERBLOCK);
43     dim3 blocks_per_grid;
44
45     // TODO 3: Compute the number of blocks_per_grid
46     // so that each thread works on one rack element
47
48
49
50
51     // TODO 4: Call the CUDA kernel
52
53
54
55
56     // TODO 5: Copy results data from GPU to CPU
57
58
59
60
61     // free memory
62     free(h_racks);
63     cudaFree(d_racks);
64     return 0;
65 }
```

Problem 4 (Jacobi - Parallelize the Code with OpenACC)**.** 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 OpenACC. 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://doc.itc.rwth-aachen.de/display/CC/GPU+cluster` 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

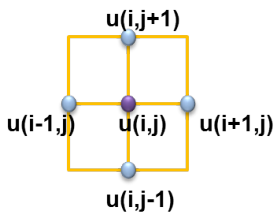

Figure 6.2: 5-point-stencil

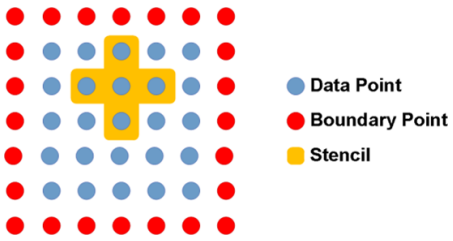

Figure 6.3: Boundaries on matrix

Finally, switch the compiler to PGI, as this is the compiler that supports OpenACC directives:

```
module switch intel pgi/19.9
```

Before you start programming GPGPUs, it is always a good idea to examine your actual GPU hardware. To this end, execute the command: `pgaccelinfo`
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 OpenACC on GPUs

a) Use the `acc parallel` directive to parallelize the first Jacobi loop (line 31). Make sure that you share the work of the loops among GPU threads.

b) If you chose to write the code without pen and paper, compile your code and have a look at the compiler feedback.

   a) Make sure that for all GPU kernels there is a line `Accelerator kernel generated`.

   b) Is a reduction generated for the `err` value? Add explicitly the `reduction` clause where it is needed.

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 (PGI compiler)[1]	4.88
OpenMP (PGI 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`.

[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

a) The PGI compiler enables a simple way to get timing information of your code by setting the environment variable `PGI_ACC_TIME` to a positive integer. Using the Makefiles provided, you can enable this option by running your code with: `make run time=1`. If you use the given binary, rename the executable in the Makefile to `jacobi_task4.2.exe` and then run the binary with `make run time=1`. Examine the output at the end of the program run. How much time was spent for the kernel execution and how much time was spent for the data transfers?

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, load the CUDA toolkit by `module load cuda` and 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 the `acc data` directive to remove the excess of data transfers. You should also apply the `present` clause to the loops.

c) If you wrote your own code, examine the new compiler feedback. Can you see any changes to the previous version (with respect to data movements)?

d) 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.

e) Check the changes by profiling the code again.

Problem 4.5. Loop Scheduling
Compare the compiler feedback for two different OpenACC versions of the Jacobi solver in Listing 6.4 and 6.5. Focus on the loop schedules.

a) Which program version would you expect to run faster? Why?

b) The compiler feedback in Listing 6.4 belongs to the previous task. The compiler feedback in Listing 6.5 belongs to the binary `jacobi_task4.5.exe`. Verify your assumption be executing `jacobi_task4.5.exe` and writing down the runtime in Table 6.2.

Discussion of this exercise on January 24, 2020.

Listing 6.3: Serial version of the Jacobi Solver

```
1  int main(int argc, char **argv)
2  {
3      const int n = 4096, m = 4096;
4      const int iter_max = 20;
5      const double tol = 1.0e-6;
6      double err  = 1.0;
7
8      // Initialize arrays
9      memset(U, 0, n * m * sizeof(double));
10     memset(Unew, 0, n * m * sizeof(double));
11     for (int i = 0; i < n; i++)
12     {
13         U[0][i]    = -1.0;
14         Unew[0][i] = -1.0;
15     }
16
17     double runtime = GetRealTime();
18     int iter = 0;
19
20
21     // while solution is not accurate enough (or max iteration number is reached)
22
23     while (err > tol \&\& iter < iter_max)
24     {
25         err = 0.0;
26
27         // compute stencil and the error value
28         // that denotes whether approximation is
29         // close to solution
30
31         for( int i = 1; i < n-1; i++ )
32         {
33             for( int j = 1; j < m-1; j++ )
34             {
35                 Unew[i][j] = 0.25 * ( U[i][j+1] + U[i][j-1]
36                                     + U[i-1][j] + U[i+1][j]);
37                 err = fmax(err, fabs(Unew[i][j] - U[i][j]));
38             }
39         }
40
41
42         // Copy new solution into old one
43
44         for( int i = 1; i < n-1; i++ )
45         {
46             for( int j = 1; j < m-1; j++ )
47             {
48                 U[i][j] = Unew[i][j];
49             }
50         }
51
52         iter++;
53
54     } // end while
55
56     runtime = GetRealTime() - runtime;
57
58     printf("Time Elapsed: \%f s\n", runtime);
59
60     return 0;
61 }
```

Listing 6.4: Compiler feedback from the Jacobi version in Task 4.4

```
main:
     52, Generating copy(U[:n][:m])
         Generating create(Unew[:n][:m])
     58, Generating present(Unew[:][:],U[:][:])
         Accelerator kernel generated
         Generating Tesla code
         58, Generating reduction(max:err)
         60, #pragma acc loop gang /* blockIdx.x */
         63, #pragma acc loop vector(128) /* threadIdx.x */
     63, Loop is parallelizable
     71, Generating present(Unew[:][:],U[:][:])
         Accelerator kernel generated
         Generating Tesla code
         73, #pragma acc loop gang /* blockIdx.x */
         76, #pragma acc loop vector(128) /* threadIdx.x */
     76, Loop is parallelizable
```

Listing 6.5: Compiler feedback from another (unknown) Jacobi version

```
main:
     52, Generating copy(U[:n][:m])
         Generating create(Unew[:n][:m])
     58, Generating present(Unew[:][:],U[:][:])
         Accelerator kernel generated
         Generating Tesla code
         58, Generating reduction(max:err)
         60, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         63, #pragma acc loop seq
     63, Loop is parallelizable
     71, Generating present(Unew[:][:],U[:][:])
         Accelerator kernel generated
         Generating Tesla code
         73, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         76, #pragma acc loop seq
     76, Loop is parallelizable
```