OpenACC Programming Lab

https://sharepoint.campus.rwth-aachen.de/units/rz/HPC/public/Shared%20Documents/PPCES2016_OpenACC-Lab.tar.gz

Sandra Wienke, RWTH Aachen University

PPCES, March 2016

If you need help or have any question please do not hesitate to ask!

1 RWTH GPU Cluster Environment

1.1 Login & Setup

User your own laptop or one of the provided laptops for logging in.

1. Login to one frontend node (e.g. cluster-x) of the RWTH Compute Cluster:
   Use the **`hpclab<XY>`** account and the provided password.
2. Jump to the GPU cluster.
   Use the GPU node name provided on the small sheet of paper.
   ```
   ssh -Y linuxgpus<AB>
   ```

For setting up the correct environment, do the following:

1. Make PGI's OpenACC compiler available by switching from the default Intel compiler to the PGI compiler:
   ```
   module switch intel pgi
   ```
2. Load the CUDA toolkit to make its tools available
   ```
   module load cuda
   ```
3. Set the GPU to run on. The number is provided on the small sheet of paper.
   ```
   export CUDA_VISIBLE_DEVICES=<no>
   ```

1.2 Getting GPU Information

Before you start programming GPGPUs, check your used GPU hardware by:

```
pgaccelinfo
```

If everything works properly, you will get a list of the most important features of your GPU. Complete Table 1 with the Cluster GPU details.

Table 1: Output of *pgaccelinfo*

Feature	Value
Number & name of devices	
Number of cores	
CUDA compute capability (cc)[1]	

1.3 Compiling & Executing the Examples

In the `GPU` directory, you can find all sources for the programming lab. The directory structure looks as follows:

- exercises
- solutions
- openmp

In the `exercises` folder, you will find C-skeletons for all tasks that will be covered during this lab. Use the Makefiles provided for compiling and executing your implemented programs:

`make help`	*Get information*
`make [jacobi]`	*Compile*
`      dbg=1`	*- with debug information*
`make run`	*Run*
`      threads=<numOmpThreads> |`	*- with given number of OpenMP threads*
`      time=1                  |`	*- enable timing information*
`      notify=<verbosityLevel> |`	*- enable runtime notifications*
`      rows=<rowsOnCPU>        ]`	*- modify the no of matrix rows on the CPU (only hetero versions)*
`make gprof`	*Profile the (CPU) code using gprof*
`make clean`	*Clean*

2 Jacobi Iteration

During the following exercises, you will port a Jacobi solver to OpenACC. This **Jacobi** example solves a finite difference discretization (5-point-stencil) of the Laplace equation (2D):

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

using the Jacobi iterative method. To this end, the Jacobi method starts with an approximation of the objective function *f(x,y)* and reuses formerly-computed matrix elements to solve the current one (see Figure 2). It iterates only about the inner elements of the 2D-grid (see Figure 1) so that the boundary elements are only used within the stencil. The solving process is aborted if either a certain number of iterations is achieved (see `iter_max`) or the computed approximation is probably close to the solution. In this code, the latter is evaluated by checking whether the biggest change on any matrix element (see array `err` and variable `err`) is smaller than a given tolerance value, in the current iteration.

[1] The compute capability (cc) corresponds to the core architecture of the GPU and describes the features supported by the CUDA-capable GPU. For instance, you need a device of cc 1.3 or higher to enable double precision floating point operations. The PGI compiler calls this `device revision number`.

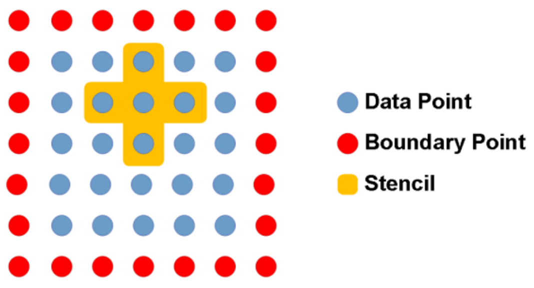
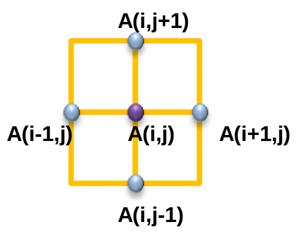


Figure 1: 5-point stencil



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Figure 2: Computation of matrix element A(i,j)

2.1 Reference Version

First, execute the OpenMP reference version:

a) Move to the folder `openmp`.
b) Run `make`. Use 12 OpenMP threads for the execution (`make run threads=12` or `export OMP_NUM_THREADS=12 && make run`).
c) Write down the runtime in Table 2.
d) Profile the code using gprof by executing `make gprof`. The output contains a “Flat profile” that lists the percentages of runtime for compute-intensive code lines. Find out where the most three time-consuming code lines are in the code. A couple of them will map to a certain for loop.

Table 2: Runtimes of different Jacobi implementations

Task	Software	Hardware	Runtime [sec]
0	OpenMP	2x Intel Westmere (=12 cores)	
1	OpenACC-Offload	1x NVIDIA Quadro 6000	
2	OpenACC-Data	1x NVIDIA Quadro 6000	
3	OpenACC-Hetero	1x NVIDIA Quadro 6000 + 2x Intel Westmere	
4	OpenACC-MultiGPU	2x NVIDIA Quadro 6000 + 2x Intel Wetmere	

2.2 Offloading Work

Now, you start writing your first OpenACC program. Move to the folder `task1` and modify the source code file `jacobi.c`. Follow the `TODO`s in the code:

a) Use the `acc parallel` and `loop` directives to parallelize the one most compute-intensive loop from section 2.1. Denote all needed clauses.
b) Compile your code and have a look at the compiler feedback.
   a. Make sure that for GPU kernels the line “`Accelerator kernel generated`” is printed.
   b. Check which data and how many elements are moved forth and back to the GPU.
c) Run your code. How fast does this version execute? Write down the runtime in Table 2.

2.3 Tools

As you might have recognized, your first OpenACC version is slow. In this task, you will figure out why. To this end, using profiling tools are a good approach.

PGI Timing Environment

The PGI compiler enables a simple way to get some basic timing information of your code. You just have to set 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
```

a) Compile your code and run it using the timing flag mentioned above. There might be introduced a small runtime overhead for collecting corresponding data.
b) 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?

NVIDIA Visual Profiler

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. If you have any problems with the Java Runtime, set `export JAVA_TOOL_OPTIONS=”-Xmx4096m -Dorg.eclipse.swt.internal.gtk.cairoGraphics=false”`.

a) If not already done, load the CUDA toolkit (check with `module list`): `module load cuda`
b) Start the Visual Profiler: `nvvp &`
c) Then create a new session.
d) In the `Executable Properties`, choose your executable file.
e) Click `Next` and `Finish`.
f) In the left 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 different kernels that are listed under `Compute` (left pane), the `kernel` duration is displayed in the properties tab as the sum of all kernel executions.
g) Now, try the `Guided Analysis`: Activate the `Analysis` tab and `Examine GPU Usage`. Have a look at the first entry. What does it say? If you have lots of time left for the lab session, also have a look at the other entries. Read their explanations. Can we do anything about these issues?
h) Can you see where data is moved between host and device in the timeline? It might be necessary to zoom into the timeline. When do we want to have the data copied between host and device?
If you need help in understanding the plots/tables, ask one of our team members.

PGI OpenACC Debugging

Debugging with PGI's OpenACC is supported by Allinea's DDT debugger and Roguewave's TotalView debugger. However, for the purpose of this task, we rely on PGI's command-line feedback.

a) Some offload information is available during runtime using the environment variable `ACC_NOTIFY`. Using the provided Makefile, you can enable this by
`make run notify=3`

b) Run your program. Which information do you get? Which grid and block dimensions are used for the first kernel? Does it fit to the compiler feedback?

2.4 Data Transfers

As starting point for the second OpenACC programming task, you can either use your source code that you have just created or you can move to the folder `task2` and work on the source files located there (and follow the `TODO`s in the code).

a) Use the `acc data` directive to remove the excess of data transfers. You may offload more loops to the GPU and use the `present` clause for defining the data status.
b) Examine the compiler feedback. Can you see any changes?
c) How fast is your program now? Write down the runtime in Table 2.
d) Profile you code again using PGI's timing information or the Visual Profiler. Can you see any changes?

2.5 Heterogeneous Computing

So far, the compute-intensive code ran only on the GPU. However, it is often a good idea to utilize all available compute resources. Therefore, you should enable heterogeneous computing in this task by letting the CPU compute simultaneously to the GPU.
In this case, you should give the GPU more work to do than the CPU by distributing more matrix rows to the GPU (see Figure 3). Be aware that you have to exchange some halo data between host and device (Figure 4) in each iteration.

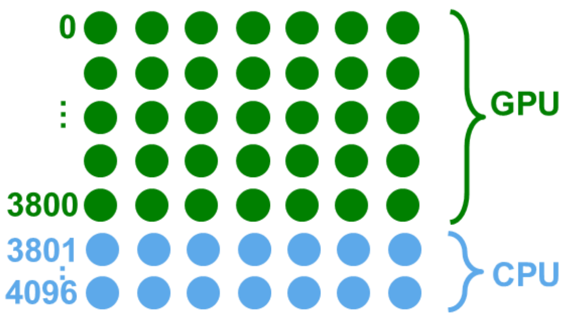


Figure 3: Work distribution between CPU and GPU

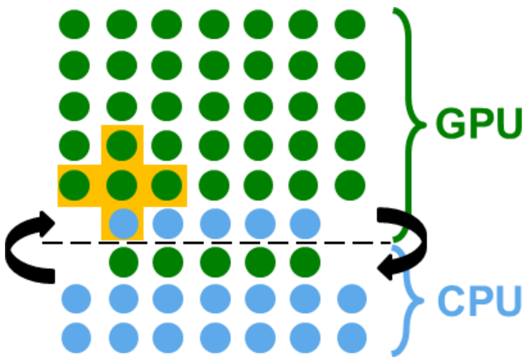


Figure 4: Halo exchange between CPU and GPU

You can either use the skeleton in folder `task3` and follow these instructions and the `TODO`s in the code (recommended because of limited lab time) [or see "or" below].

a) Decompose the matrix rows to GPU and CPU. See the variable `n_cpu` that denotes the number of rows that shall be computed on the CPU. Use it and the corresponding index `j_cpu_start` to distribute the loops. Later you will have to adapt this value.
b) Use the `async` clause to enable overlap of CPU and GPU work.
c) Use OpenMP to fully utilize the CPU. You need `#pragma omp parallel for` on the inner-while loops. However, you should also specify OpenMP's data and `reduction` clauses if appropriate. Some loops were already parallelized with OpenMP for you to

ensure good data locality. If you run the application at the end, don't forget to increase the number of threads by `make run threads=<noThreads>`.

d) Use the `update` directive to manage the halo data exchange.

e) The OpenACC specification says that reduction variables are directly copied back to the host after the loop. However, this would prevent simultaneous execution of the first loop on CPU and GPU. Therefore, you have to decouple the reduction copy of `acc_err` from the computation and update its value manually: Put `acc_err` into the data region and `update` it before and (when needed) after the computation loop.

f) Think about synchronizing data again. Do you have to insert a `wait` directive to avoid inconsistent data?

g) Fix the calculation of `err`.

h) What is the runtime of this heterogeneous version? Play around with the decomposition size of the matrix. Use `make run rows=<rowsOnCPU> threads=<noThreads>` to do so. Write down the shortest runtime in Table 2.

or start from your source code that you have already implemented and follow these steps

a) Define a variable that denotes the matrix split between host and device. For this domain decomposition, which data is needed on which architecture? Don't forget that we need a stencil for updating one matrix element.

b) Split all compute-intensive loops along this variable (matrix element computation and swap loop). Make sure to reduce the error variable of both loop parts after splitting the reduction loop.

c) Afterwards, execute one part of the loop on the GPU and simultaneously the other part of the loop on the CPU. You might need the `async` clause.

d) Before the swap loop, the halo data must be exchanged (see Figure 1) which can be done using the `update` directive.

e) The OpenACC specification says that reduction variables are directly copied back to the host after the loop. However, this would prevent simultaneous execution of the first loop on CPU and GPU. Therefore, you have to decouple the reduction copy of `acc_err` from the computation and update its value manually: Put `acc_err` into the data region and `update` it before and (when needed) after the computation loop.

f) Think about synchronizing data again. Where must a `wait` directive be used to avoid inconsistent data?

g) Parallelize the CPU code using OpenMP. You will usually only need `#pragma omp parallel for`. You should also use OpenMP's `reduction` clause where appropriate. If you run the application at the end, don't forget to increase the number of threads by `make run threads=<noThreads>`.

h) What is the runtime of this heterogeneous version? Play around with the decomposition size of the matrix. Write down the shortest runtime in Table 2.

2.6 Multiple GPUs *(optional)*

If you have a cluster of GPU nodes, you can utilize their compute power by having an MPI program that runs on different nodes with GPUs. If you have several GPUs within <u>one</u> node, you can use both accelerators simultaneously even easier by specifying in your program which one to use by OpenACC API calls. Here, you will do the latter.
To distribute the work, we follow the strategy described in section *2.5 Heterogeneous Computing*. The only difference is that we now have three

partitions: one on GPU 0, one on GPU 1 and one on the CPU (as shown in Figure 5).

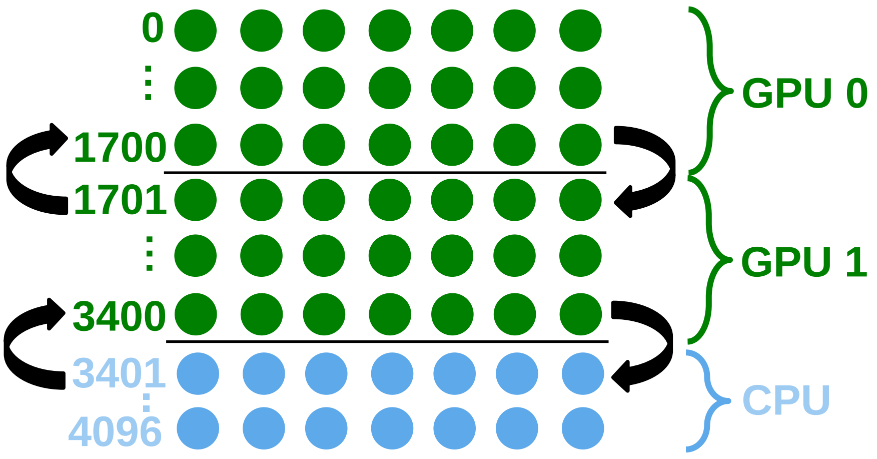


Figure 5: Work distribution between two GPUs and a 2-socket CPUs

You can either start from your source code that you have already implemented or use the skeleton in the folder `task4` (and follow the `TODO`s in the code).

a) On your exercise nodes, you have generally 2 GPUs available. To re-activate computations on both GPUs, set the environment variable:
`export CUDA_VISIBLE_DEVICES=0,1`

b) Leave the CPU matrix size as it is and divide evenly the number of rows, which were located on one GPU so far, to both GPUs.

c) For specifying that the following OpenACC pragmas shall be executed on a certain GPU use the OpenACC API call `acc_set_device_num(<id>,acc_device_nvidia)`. You have to include `openacc.h` for using API routines.

d) Start by moving only the necessary data to GPU 0 and GPU 1. Therefore, exchange the structured `data` region to unstructured `enter data` directives. Don't forget to `create` the reduction variables for both GPUs explicitly on the devices. After the equation system was solved, delete temporary data and copy back the result matrix. Use `exit data` for that.

e) Note that you have to manually `update` the reduction variables on both GPUs now.

f) Before the swap loop, the halo data must be exchanged (see Figure 5) which can be done using the `update` directive. The first GPU needs to update its last matrix row, the second GPU needs to update its first and last row and the CPU needs to update its first row. Make sure that the data is synchronized.

g) Combine the reduction variables of both GPUs and the CPU.

h) Determine a good value for the number of rows on the CPU by playing around with `make run rows=<rowsOnCPU> threads=<noThreads>`. Use the results from task 3 to obtain reasonable starting values.

i) What is the runtime of this heterogeneous multi-device version? Write down the shortest runtime in Table 2.