

GHCG 2013: OpenACC Programming Lab

May 2013

<https://sharepoint.campus.rwth-aachen.de/units/rz/HPC/public/Shared%20Documents/ghcg-openacc-2013.tar.gz>

Sandra Wienke, RWTH Aachen University
Timo Stich, NVIDIA

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

1. RWTH GPU Cluster Environment

The RWTH GPU Cluster comprises a total of 57 NVIDIA GPUs. These GPUs have compute capability 2.0 (Fermi architecture) and enable features like double precision floating-point operations. The GPU Cluster runs Linux as operating system and works with a module system such as the normal RWTH Compute Cluster. More information can be found on the slides *GPU-Cluster@RZ*.

1.1. Login & Setup

Login to the RWTH Linux Compute Cluster first. You can either use your own laptop or one of the provided laptops. In the handout *How to login*, see the sections *Access to Laptop* and *Access to Cluster* for more information.

Then, jump from the frontend node (e.g. `cluster-x`) of the Linux Cluster to one node of the GPU Cluster using the *hpclab* account, the password and the GPU node name that are provided on the small sheet of paper:

```
ssh -Y hpclab<XY>@linuxgpus<AB>
```

During the following exercises, you will use the OpenACC compiler from PGI. Make it available by switching the default Intel compiler to the PGI compiler:

```
module switch intel pgi
```

1.2. Compiling & Executing the Examples

In the *OpenACC* directory, you can find all sources for the programming lab, as well as the presented slides. The directory structure looks as follows:

- ▶ slides
- ▶ lab
 - exercises
 - solutions
 - openmp_reference
 - cuda_c_reference

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

```
make help
make [c]
make fc
make run [dev=<deviceID>]
make clean
```

Get information
Compile for C
Compile for Fortran
Run (on certain device)
Clean

Since several users may use the same machine as you do and our GPUs can only be exclusively used by one user, it is possible that the GPU is already occupied and you get a corresponding error message. If so, try to use the second GPU on the node (`make run dev=1`) or try again after a few seconds.

If you want to run the OpenMP reference version some time, you have to switch back to the Intel compiler first. If you want to run the CUDA reference version, you have to load the CUDA toolkit first. In summary:

- ▶ OpenACC: `module switch intel pgi`
- ▶ OpenMP: `module switch pgi intel`
- ▶ CUDA: `module load cuda`

2. Getting GPU Information

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 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	

3. 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):

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

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 the residual becomes very small or a certain number of iterations is achieved.

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.

If you encounter strange behavior during your implementation activities, it might make sense to check the results for errors. For this purpose, we have prepared some functions for you. See Appendix **Fehler! Verweisquelle konnte nicht gefunden werden.** for more details.

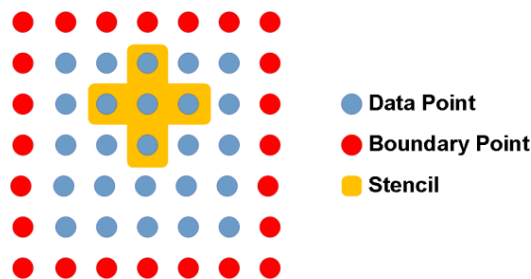
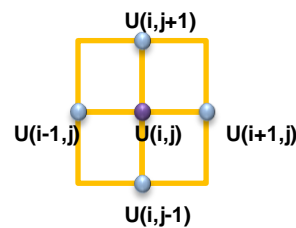


Figure 1: 5-point stencil



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

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

3.1. Reference Versions

First, execute the OpenMP and the CUDA reference versions (cf. Section 1.2 *Compiling & Executing the Examples*):

- Move to `openmp_reference` or `cuda_c_reference`, respectively.
- Switch to the appropriate compiler or load the CUDA toolkit.
- Run `make` and `make run`.
- Write down the runtimes of these two reference versions in the corresponding lines of Table 2.

Table 2: Runtimes of different Jacobi implementations

Software	Hardware	Runtime [sec]
OpenMP	2x Intel Westmere (=12 cores)	
CUDA C (simple)	NVIDIA Quadro 6000	
OpenACC-Kernels	NVIDIA Quadro 6000	
OpenACC-Data	NVIDIA Quadro 6000	
OpenACC-Loop	NVIDIA Quadro 6000	

4. OpenACC Basics

4.1. Kernels Construct

Now, you start writing your first OpenACC program. Move to the folder `1-kernels` and modify the source code files.

- Use the `acc kernels` directives to parallelize the Jacobi loops.
- Compile your code and have a look at the compiler feedback.
 - Make sure that for all GPU kernels there is a line "Accelerator kernel generated".
 - Is a reduction generated for the `err` value?
 - Check which data and how many elements are moved forth and back to the GPU.
- Run your code. How fast does this version execute? Write down the runtime in Table 2.

4.2. Tools

As you might have recognized, your first OpenACC version is kind of 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
```

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

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`.

- If not already done, load the CUDA toolkit (check with `module list`): `module load cuda`
- Start the Visual Profiler: `nvvp &`
- Then create a new session.
- In the Executable Properties, choose your executable file.
- Click Next and Finish.
- 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.
- In the timeline, can you see where data is moved between host and device? 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.

It is also possible to get simple profile information from the command line stored in a log file.

- Set the environment variable: `export COMPUTE_PROFILE=1`
- Run your program. You will get the log file `cuda_profile_<DEVID>.log`.
- Open this log file and `grep` for `"_gpu"`.
- Which information do you see?
- Unset the environment variable again (either by setting its value to "nothing" or to "0").

PGI OpenACC Debugging

At the moment, one big problem of PGI's OpenACC is the lack of debugging support. As this might change in future, today, you have only limited capabilities to see what is going on in your PGI OpenACC program.

- Set the following environment variable: `export ACC_NOTIFY=3`
- 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?
- Afterwards, unset the environment variable again (either by setting its value to "nothing" or to "0").

Remark: Of course, you can use any common debugger to investigate problems in the non-kernel source code.

4.3. 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 `2-data` and work on the source files located there.

- Use the `acc data` directive to remove the excess of data transfers. You may also use the `present` clause.
- Examine the compiler feedback. Can you see any changes?
- How fast is your program now? Write down the runtime in Table 2.

5. OpenACC Tuning

Now that you have a basic OpenACC version of the Jacobi solver, you should think about some further tuning steps...

Sometimes, it is a good idea to take back the responsibility of finding the parallelism from the compiler to the programmer (especially if the compiler ignores some of your hints). One approach may be to replace the `kernels` construct by the `parallel` construct. Modify/ copy your self-created source code or use the files in `3-loop`.

You should also try to experiment with different loop schedules since it can strongly affect performance (depending on the application).

Note: Due to a bug in the recent PGI Compiler (13.3), the specification of the number of gangs will be ignored. But keep in mind that this might also be a tuning opportunity. Especially, computing multiple elements per CUDA thread can reduce overheads such as address calculation (depending on your code). Additionally, the nesting of identical worksharing types is not allowed with the `parallel` construct right now so that you cannot use more-dimensional grids and blocks. The latter might change with the next release of the OpenACC specification (compare `tile` clause).

- a) Apply the `parallel` constructs. Note that you now have to specify all `loop` work-sharing constructs as well.
- b) Specify the reduction manually.
- c) Think about a different parallelization strategies (loop schedule): Try to distribute the work of the outer loops to the GPU multiprocessors (CUDA blocks, `gangs`) in 1D grids and the work of the inner loops to the cores of the multiprocessors (CUDA threads, `vector`) in 1D blocks.
- d) Try also out different vector lengths.
- e) You can see the grid (`gangs`) and block (`vector`) sizes with `ACC_NOTIFY` or using the Visual Profiler (see section 4.2). Try it out.
- f) What is the best performance you can achieve? Write down the runtime in Table 2.

6. OpenACC & CUDA Libs

Sometimes, you already have a single optimized CUDA/ OpenCL kernel programmed or you need to use a tuned CUDA library (such as CUBLAS, CUSPARSE or CUFFT). If you want to use the OpenACC API for the rest of your code acceleration, the OpenACC kernels have to interact with the data produced by the CUDA kernel/ libraries. You will implement this during this exercise.

The scenario for this exercise is the filtering of a 1D signal. The implementation works by first transforming the signal and the filter in the fourier domain and doing a point-wise multiplication followed by an inverse fourier transform of the filtered signal. The fourier transforms are implemented using the Nvidia CUFFT library. The point-wise multiplication should be an OpenACC kernel. You can find the skeleton for this task in folder `4-lib`.

- a) Implement a complex multiply and scale of two complex arrays with OpenACC (function `complexPointwiseMulAndScale()` in `cufft_acc.c`). Save the filtered result in `signal`.
 - ▶ Hint: We have the two arrays `signal` and `filter_kernel` which contain complex numbers:
 - real parts: `array[2*i]` $i=0,\dots,n-1$
 - imaginary parts: `array[2*i+1]`
 - ▶ Reminder
 - Multiplication of two complex numbers: $(x + yi)(u + vi) = (xu - yv) + (xv + yu)i$.
 - Multiplication of a real and a complex number: $(x + yi) u = xu + yu i$
 - ▶ Hint: Use the `deviceptr` clause that enables access on externally allocated and managed memory inside OpenACC code.
- b) Make sure that the correctness test at the end is passed.