Leibniz Supercomputing Centre
of the Bavarian Academy of Sciences and Humanities

# OpenACC-lab: Simple tests with the PGI compiler

In this lab you will get first experiences with the PGI compiler and learn how to add OpenACC pragmas to your code to offload computations to an accelerator like a GPU.

## Appropriate Environment

| | LRZ GPU Cluster | TUM MAC Cluster |
|---|---|---|
| Login to the GPU-Cluster | *ssh lxlogin_gpu.lrz.de* | ssh mac-login-intel.tum-mac.cos.lrz.de |
| Reserve a GPU for you | *salloc --gres=gpu:1 --reservation=gpu_course* | salloc --partition=nvd |
| Facilitate the execution on GPU by defining: | *export RUN="srun --gres=gpu:1* | export RUN="srun --partition=nvd" |

- Load the PGI environment modules:
  . /etc/profile.d/modules.sh
  *module unload ccomp fortran*
  *module load fortran/pgi*
  *module load ccomp/pgi*
  *module load cuda*

## Lab 0

- Run the commands *pgcpuid* and *pgaccelinfo* using
  *$RUN command vs. command*
  and interpret the information given.

## Lab 1

- Add an accelerator kernels loop region

  *#pragma acc kernels loop*
  *{*
  *...*
  *}*

  block around the computation of r[i] in the C code c1.c and compile the code using
  *pgcc –acc -Minfo=accel -fast -o c1 c1.c*

- Run the code using
  *$RUN ./c1*

- *export ACC_NOTIFY=1* and run the code again.

- Run the code directly using *./c1* . Where does the code run now? Compile using
  *pgcc -acc -Minfo=accel -fast -ta=nvidia c1.c*
  and run the code again directly and via $RUN. Compare with previous case.

- Recompile the code using the option *–ta=nvidia,keepgpu* and have a look at the generated intermediate CUDA source file.

- Remove the *restrict* keyword in the declaration of the pointers to the arrays *a* and *r* and compile and run again. Look at compiler output.

- Compile with the option *-Msafeptr* and run again.

- Add a

  *!$acc kernels loop*
  *…*
  *!$acc end kernels loop*

  block around the computation of r(i) in the Fortran code f1.f90 and compile the code using

  *pgfortran –acc -Minfo=accel -fast -o f1 f1.f90*

## Lab 2

- Add a ***kernels loop*** region around the for-loop which computes r[i] in c2.c, compile and run the code
- Add a call to *acc_init(acc_device_nvidia)* prior to the timing of the compute region (in a copy of the program) and compare the timings
- Add a clause to the region to ensure that the (single precision) code runs on the GPU only for values of n > min for which the execution time on the host is larger than on the GPU (try to find out the value min experimentally)
- Convert (a copy of) the program to double precision and compare the timings.
- Look at performance output using *PGI_ACC_TIME=1*

## Lab 3

- Add a single kernels loop region around all for-loops in the routine smooth in c3.c and compile and run the code.
- Look on the default data clauses generated.
- Modify the code and declare in the kernels directive that the full array b should be copied and that *a* does not need to be copied to the GPU.
- Test other data clauses and the compiler output / runtime behaviour.
- Specify the loop scheduling of the first 2 for-loops explicitly, by adding some of the following pragmas in front of the for loops

  *#pragma acc loop seq*
  *#pragma acc loop gang(n), worker(w), vector(k)*
  *...*

- Run with ACC_NOTIFY=1 and look at the corresponding grid and block dimensions.

## Lab 4

- Surround the smooth call on the GPU with a *data construct*:

```
#pragma acc data copy(…)
{
      smooth( a, b, w0, w1, w2, n, m, iters );
}
```

This tells the compiler to copy both arrays to the GPU before the call, and bring the results back to the host memory after the call. Inside the function, replace the *copyin* or *copy* data clauses by a *present* clause

## Useful Directives and Functions:

```
#pragma acc kernels loop
#pragma acc kernels loop if (n>…)
#pragma acc kernels loop copy(data[base:nelem][base:nelem])
#include <openacc.h>
acc_init( acc_device_nvidia );
#pragma acc loop gang(g) worker(w) vector(v)
#pragma acc data …
```