CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Porting Applications to GPUs

CSCS Summer School 2025

Andreas Jocksch, Prashanth Kanduri, Radim Janalik & Ben Cumming

Using GPUs in your Application?

Rule #1: **DO NOT** develop your own GPU code

Libraries

There are numerous libraries available for GPUs. Some from Nvidia and many open source.

- cuBLAS: Dense linear algebra primitives
- Thrust: C++ STL-like algorithms and containers
- cuRAND and Random123: Random numbers
- cuFFT: Fast-Fourier Transforms
- Kokkos: Generic performance portable parallel motifs

... and many more!

Take some time to investigate what is available before starting.

Are you going to write your own code?

Directives
OpenACC and OpenMP define *directives* that can be used to instruct the compiler how to generate GPU code.
In theory the easiest path for porting.

GPU-Specific Languages
Languages designed for GPU programming.
Maximum flexibility and performance.
For example: CUDA, OpenCL and SYCL.

Things to Consider

Before starting on a GPU implementation, it pays to ask some questions and do some preliminary exploration:

1. Is my program computationally or bandwidth intensive?
2. Does it have enough parallel work to utilize the GPU?
3. Must I change algorithms to expose enough parallelelism?
4. Are there serial bottlenecks that will limit scaling? (see Amdahl's and Gustafson's laws)
5. Is the pain worth the gain?

- Questions 1, 2 and 3 will be discussed in this course.
- Questions 5 requires answers for 1-4.

CUDA

CUDA (Compute Unified Device Architecture) is a **parallel computing platform** and **API** for CUDA-enabled Nvidia GPUs

We use CUDA as short hand for CUDA C/C++ and API

- CUDA C++ is a **superset** of C++
- Adds keywords for writing kernels to run on the GPU
- Adds syntax for launching kernels on the GPU

The CUDA toolkit is more than a programming language. It also provides:

- Runtime API for managing GPU resources and execution
- Tools including profilers and debuggers

Compiling CUDA

CUDA code is compiled with the `nvcc` compiler driver

- source files have `.cu` extension
- headers have `.h`, `.hpp`, `.hcu` extension

CUDA compilation involves multiple splitting, compilation, preprocessing and merging steps

- `nvcc` hides this complexity from the user
- It closely mimics the interface of the GNU compiler
- Behind the scenes it:
    - uses GCC to compile the code that runs on CPU;
    - and compiles the GPU code separately

Compiling CUDA

Example CUDA compilation

```
> nvcc -arch=sm_90 -lineinfo -O2 -std=c++11 -g -o foo foo.cu
```

Some flags are for **device** code generation:

- `-arch=sm_90` target GPU architecture (Hopper streaming multiprocessor)
- `-lineinfo` debug information for device code

Some are for **host**:

- `-g` debug information for host code

And some are for both **host** and **device**:

- `-O2` optimization level
- `-std=c++11` target language
- `-o foo` name of executable

Exercise: Getting Started on Daint (Alps)

In this exercise we will get introduced to Daint and make sure that everybody is set up.

```
# log on to daint (from ela.cscs.ch) with your course username & password
> ssh -X <your account name>@daint

# go to scratch and get the course material
> cd $SCRATCH/SummerUniversity
> git pull

# compile and test the demo
> cd cuda/practicals/demos
> cat hello.cu
> uenv image pull prgenv-gnu/24.11:v2
> uenv start --view=default prgenv-gnu/24.11:v2
> nvcc -arch=sm_90 hello.cu -o hello
# run the job on the course reservation for maximum 60 minutes
# find your group id with the "groups" command
> srun -N 1 -A class05 --reservation=summer_uni -t60 ./hello
```