CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Introduction to GPUs in HPC

Ben Cumming, CSCS
July 9, 2020

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Using GPUs in Your Application

Rule #1: **don't** develop your own GPU code!

Libraries

There are many open libraries for GPUs.

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

. . . And many more!

Take some time to investigate what is available before starting.

You are 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?
5. Is the pain worth the gain?

- Questions 1, 2 and 3 will be discussed in this course.
- Question 4 will be considered briefly here.
- Questions 5 requires answers for 1–4.

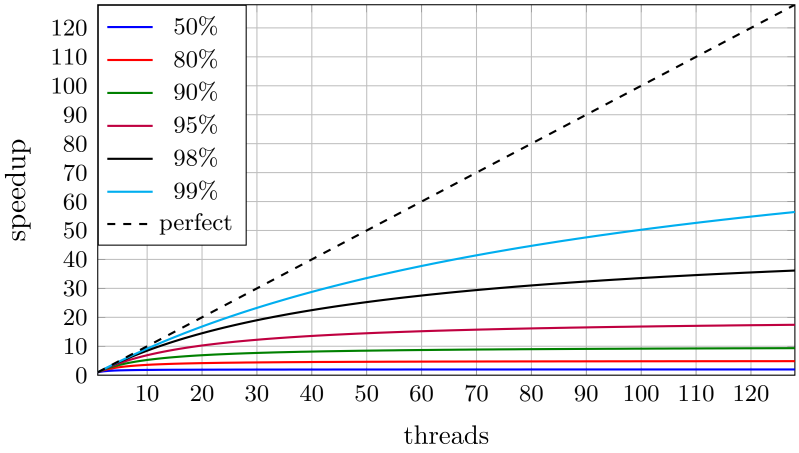
Limitations to parallel speedup

- Parallel speedup is limited by **the proportion of serial work** in your code.
- **Amdahl's law** defines the **maximum possible speedup** when only parts of the code can be parallelised

  $$t_n = t_1 \left(p + \frac{(1-p)}{n} \right),$$

  where t_n is time to solution for n threads and $p \in [0, 1]$ is the proportion of sequential code.
- The limit on time to solution is $\lim_{n\to\infty} = pt_1$
  - e.g. 1% of serial code gives a maximum 100$\times$ speedup.

Amdahl illustrated



CUDA

CUDA 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:

- 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 with Clang

Clang now supports compilation of CUDA code, targetting NVIDIA GPUs.

- Performance of the generated code is on par with nvcc.
- It is a good idea to test your CUDA code with both Clang and nvcc.
- The most recent version of the Cray C++ compiler on Daint is Clang based.

Compiling CUDA

Example CUDA compilation

```
> nvcc -arch=sm_60 -lineinfo -O2 -std=c++11 -g -o foo foo.cu
```

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

- `-arch=sm_60` target GPU architecture (Pascal)
- `-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.

Compiling CUDA with Clang

Compilation with Clang uses different gpu-specific options:

```
> CC -xcuda --cuda-gpu-arch=sm_60 --cuda-path=$CUDA_ROOT
         -O2 -std=c++11 -g -o foo foo.cu
```

Where `CC` is the Cray compiler wrapper on Daint.

Exercise: Getting Started on Piz Daint

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

```
# log on to daint with your course username & password
> ssh -X <your account name>@daint

# get one node on the course reservation for 60 minutes
> salloc -Cgpu --reservation=summer_school -t60

# go to scratch and get the course material
> cd $SCRATCH/SummerSchool2020
> git pull

# compile and test the demo
> cd topics/cuda/practicals/demos
> cat hello.cu
> module load gcc cudatoolkit
> nvcc -arch=sm_60 hello.cu -o hello
> srun ./hello
```