GPU Programming: When, Why and How?

Graphical processing units (GPUs) are the workhorse of many high performance computing (HPC) systems around the world. The number of GPU-enabled supercomputers on the Top500 has been steadily increasing in recent years and this development is expected to continue. In the near future, the majority of HPC computing power available to researchers and engineers is likely to be provided by GPUs or other types of accelerators. Programming GPUs and other accelerators is thus crucial to developers of software run on HPC systems.

However, the landscape of GPU hardware, software and programming environments is complicated. Multiple vendors compete in the high-end GPU market, with each vendor providing its own software stack and development toolkits, and even beyond that, there is a proliferation of tools, languages and frameworks that can be used to write code for GPUs. It can thus be difficult for individual developers and project owners to know how to navigate across this landscape and select the most appropriate GPU programming framework for their projects based on the requirements of a given project and technical requirements of any existing code.

This material is meant to help both software developers and decision makers navigate the GPU programming landscape and make more informed decisions on which languages or frameworks to learn and use for their projects. Specifically, you will:

- Understand why and when to use GPUs.
- Become comfortable with key concepts in GPU programming.
- Acquire a comprehensive overview of different software frameworks, what levels they operate at, and which to use when.
- Learn the fundamentals in at least one framework to a level which will enable you to quickly become a productive GPU programmer.

⚙ Prerequisites

Familiarity with one or more programming languages like C/C++, Fortran, Python or Julia is recommended.

Setup

Local installation

Since this lesson is taught using an HPC cluster, no software installation on your own computer is needed.

Running on LUMI

Interactive job, 1 node, 1 GPU, 1 hour:

```
$ salloc -A project_465002387 -N 1 -t 1:00:00 -p standard-g --gpus-per-node=1
$ srun <some-command>
```

Exit interactive allocation with `exit`.

Interacive terminal session on compute node:

```
$ srun --account=project_465002387 --partition=standard-g --nodes=1 --cpus-per-task=1 --ntasks-per-node=1 --gpus-per-node=1 --time=1:00:00 --pty bash
$ <some-command>
```

Corresponding batch script `submit.sh`:

```
#!/bin/bash -l
#SBATCH --account=project_465002387
#SBATCH --job-name=example-job
#SBATCH --output=examplejob.o%j
#SBATCH --error=examplejob.e%j
#SBATCH --partition=standard-g
#SBATCH --nodes=1
#SBATCH --gpus-per-node=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=1:00:00

srun <some_command>
```

- Submit the job: `sbatch submit.sh`
- Monitor your job: `squeue --me`
- Kill job: `scancel <JOB_ID>`

Running Julia on LUMI

In order to run Julia with `AMDGPU.jl` on LUMI, we use the following directory structure and assume it is our working directory.

```
.
├── Project.toml  # Julia environment
├── script.jl     # Julia script
└── submit.sh     # Slurm batch script
```

An example of a `Project.toml` project file.

```
[deps]
AMDGPU = "21141c5a-9bdb-4563-92ae-f87d6854732e"
```

For the `submit.sh` batch script, include additional content to the batch script mentioned above.

```
#SBATCH --cpus-per-task=2
#SBATCH --mem-per-cpu=1750

module use /appl/local/csc/modulefiles

module load julia
module load julia-amdgpu

julia --project=. -e 'using Pkg; Pkg.instantiate()'
julia --project=. script.jl
```

An example of the `script.jl` code is provided below.

```
using AMDGPU

A = rand(2^9, 2^9)
A_d = ROCArray(A)
B_d = A_d * A_d

println("----EOF----")
```

Running Python

On LUMI

A singularity container containing all the necessary dependencies has been created. To launch the container and the `IPython` interpreter within it, do as follows:

```
$ salloc -p small-g -A project_465002387 -t 1:00:00 -N 1 --gpus=1
$ srun --pty \
     singularity exec --no-home \
     -B $PWD:/work \
     /scratch/project_465002387/containers/gpu-programming/python-from-
docker/container.sif \
     bash

Singularity> cd /work
Singularity> . /.venv/bin/activate
Singularity> python  # or ipython
```

Recipe for creating the container

For reference, the following files were used to create the above singularity container. First a singularity def file,

```
Bootstrap: docker
From: rocm/dev-ubuntu-24.04:6.4.4-complete

%environment
    CUPY_INSTALL_USE_HIP=1
    ROCM_HOME=/opt/rocm
    HCC_AMDGPU_TARGET=gfx90a
    LLVM_PATH=/opt/rocm/llvm

%post
    export CUPY_INSTALL_USE_HIP=1
    export ROCM_HOME=/opt/rocm
    export HCC_AMDGPU_TARGET=gfx90a
    export LLVM_PATH=/opt/rocm/llvm
    export PATH="$HOME/.local/bin/:$PATH"

    apt-get update && apt-get install -y --no-install-recommends curl ca-
certificates git
    curl -L https://astral.sh/uv/install.sh -o /uv-installer.sh
    sh /uv-installer.sh && rm /uv-installer.sh

    . $HOME/.local/bin/env

    uv python install 3.12
    uv venv -p 3.12 --seed
    uv pip install --index-strategy unsafe-best-match -r /tmp/envs/requirements.txt
    uv pip freeze >> /tmp/envs/requirements_pinned.txt

    touch /usr/lib64/libjansson.so.4 /usr/lib64/libcxi.so.1
/usr/lib64/libjansson.so.4
    mkdir /var/spool/slurmd /opt/cray
    mkdir /scratch /projappl /project /flash /appl
```

and a bash script to build the container,

```
#!/bin/sh
ml purge
ml LUMI/24.03 partition/G
ml load systools/24.03  # For proot

export SINGULARITY_CACHEDIR="$PWD/singularity/cache"
export SINGULARITY_TMPDIR="$FLASH/$USER/singularity/tmp"
singularity build -B "$PWD":/tmp/envs --fix-perms container.sif
build_singularity.def
```

Tip

You can also interactively build using:

```
singularity build --sandbox <other flags> container-sandbox
build_singularity.def
singularity shell --writable container-sandbox
```

and finally a `requirements.txt` file:

```
jupyterlab
jupyterlab-git
nbclassic
matplotlib
numpy
cupy
# jax[rocm]
jax-rocm7-pjrt @ https://github.com/ROCm/rocm-jax/releases/download/rocm-jax-
v0.6.0/jax_rocm7_pjrt-0.6.0-py3-none-manylinux_2_28_x86_64.whl
jax-rocm7-plugin @ https://github.com/ROCm/rocm-jax/releases/download/rocm-jax-
v0.6.0/jax_rocm7_plugin-0.6.0-cp312-cp312-manylinux_2_28_x86_64.whl
jaxlib @ https://github.com/ROCm/rocm-jax/releases/download/rocm-jax-v0.6.0/jaxlib-
0.6.2-cp312-cp312-manylinux2014_x86_64.whl
jax==0.6.2
--extra-index-url https://test.pypi.org/simple
numba-hip[rocm-6-4-4] @ git+https://github.com/ROCm/numba-hip.git
```

LUMI also has official singularity images for Jax. These can be found under the path:

```
/appl/local/containers/sif-images/
```

```
$ srun --pty \
     singularity exec -B $PWD:/work \
     /appl/local/containers/sif-images/lumi-jax-rocm-6.2.4-python-3.12-jax-0.4.35.sif \
     bash
Singularity> cd /work
Singularity> $WITH_CONDA
Singularity> python
Python 3.12.9 | packaged by Anaconda, Inc. | (main, Feb  6 2025, 18:56:27) [GCC 11.2.0]
on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import jax
>>> jax.devices()
[RocmDevice(id=0)]
```

On LUMI (only CPU)

On LUMI, you can set up Python distribution as following:

```
$ module load cray-python/3.9.13.1
$ # install needed dependencies locally
$ pip3 install --user numpy numba matplotlib
```

On Google Colab

Google Colaboratory, commonly referred to as "Colab", is a cloud-based Jupyter notebook environment which runs in your web browser. Using it requires login with a Google account.

This is how you can get access to NVIDIA GPUs on Colab:

- Visit https://colab.research.google.com/ and sign in to your Google account
- In the menu in front of you, click "New notebook" in the bottom right corner
- After the notebook loads, go to the "Runtime" menu at the top and select "Change runtime type"
- Select "GPU" under "Hardware accelerator" and choose an available type of NVIDIA GPU (e.g. T4)
- Click "Save". The runtime takes a few seconds to load - you can see the status in the top right corner
- After the runtime has loaded, you can type `!nvidia-smi` to see information about the GPU.
- You can now write Python code that runs on GPUs through e.g. the numba library.

Access to code examples

Some exercises in this lesson rely on source code that you should download and modify in your own home directory on the cluster. All code examples are available in the same GitHub repository as this lesson itself. To download it you should use Git:

```
$ git clone https://github.com/ENCCS/gpu-programming.git
$ cd gpu-programming/content/examples/
$ ls
```

Why GPUs?

❓ Questions

- What is Moore's law?
- What problems do GPUs solve?

❗ Objectives

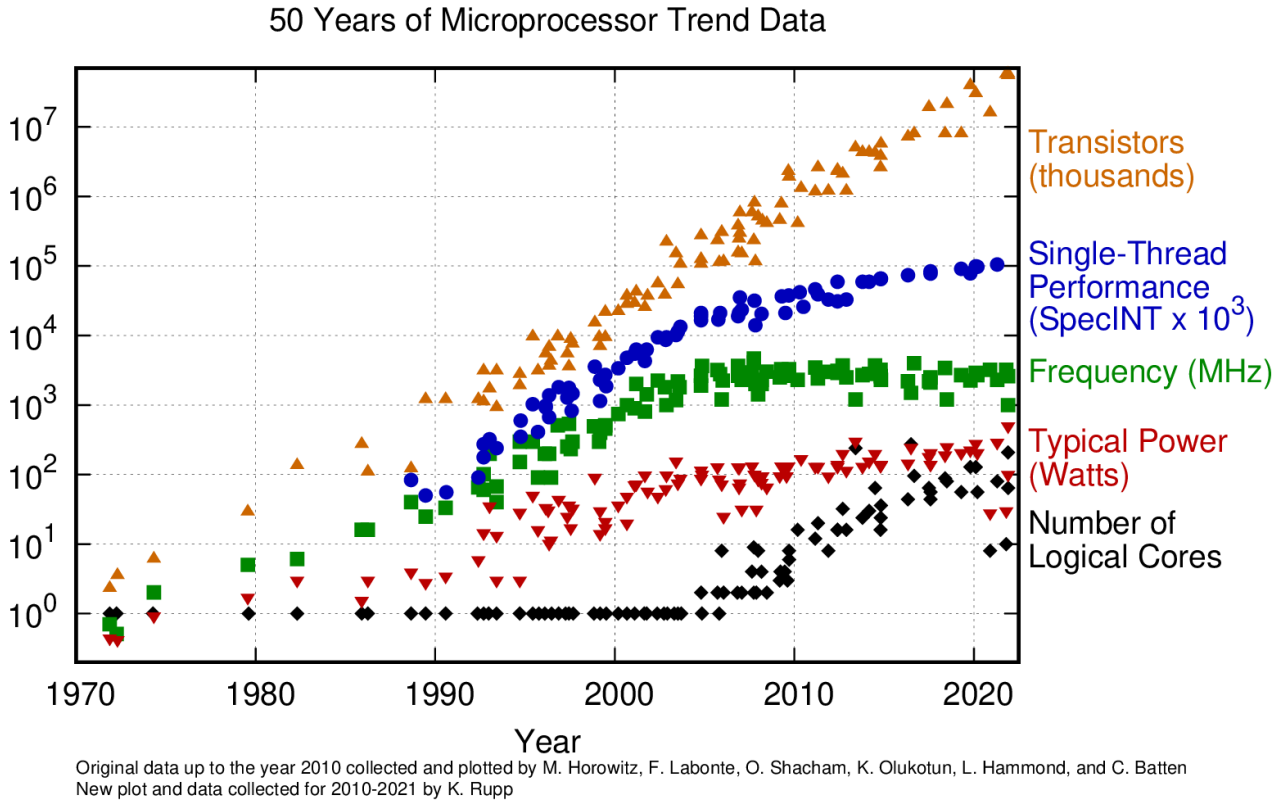
- Explain the historical development of microprocessors and how GPUs enable continued scaling in computational power

Instructor note

- 15 min teaching
- 0 min exercises

Moore's law

It states that the number of transistors in a dense integrated circuit doubles about every two years. More transistors means smaller size of a single element, so higher core frequency can be achieved. However, power consumption scales with frequency to the third power, therefore the growth in the core frequency has slowed down significantly. Higher performance of a single node has to rely on its more complicated structure and can still be achieved with SIMD (single instruction multiple data), branch prediction, etc.



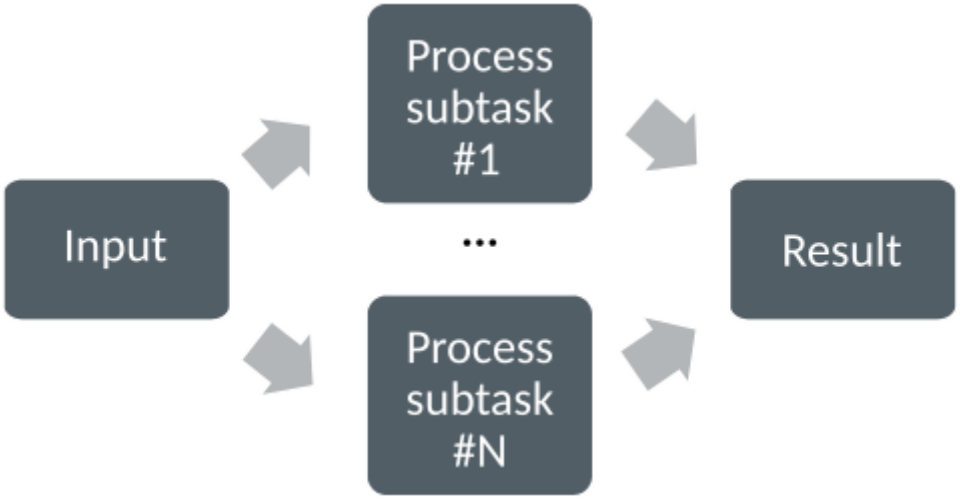
The evolution of microprocessors. The number of transistors per chip doubles roughly every 2 years. However, it can no longer be explored by the core frequency due to the power consumption limits. Before 2000, the increase in the single core clock frequency was the major source of the increase in the performance. Mid 2000 mark a transition towards multi-core processors.

Increasing performance has been sustained with two main strategies over the years:

- Increase the single processor performance:
- More recently, increase the number of physical cores.

Computing in parallel

The underlying idea of parallel computing is to split a computational problem into smaller subtasks. Many subtasks can then be solved *simultaneously* by multiple processing units.



Computing in parallel.

How a problem is split into smaller subtasks strongly depends on the problem. There are various paradigms and programming approaches to do this.

Graphics processing units

Graphics processing units (GPU) have been the most common accelerators during the last few years, the term GPU sometimes is used interchangeably with the term *accelerator*. GPUs were initially developed for highly-parallel tasks of graphic processing. But over the years, they were used more and more in high-performance computing (HPC).

GPUs are a specialized parallel hardware for floating point operations. They are basically co-processors (helpers) for traditional CPUs: a CPU still controls the work flow but it delegates highly parallel tasks to the GPU. GPUs are based on highly parallel architectures, which allows taking advantage of the increasing number of transistors.

Using GPUs allows one to achieve extreme performance per node. As a result, a single GPU-equipped workstation can outperform small CPU-based clusters for some types of computational tasks. The drawback is: usually major rewrites of programs are required with an accompanying change in the programming paradigm.

Host vs device

GPU-enabled systems require a heterogeneous programming model that involves both CPU and GPU, where the CPU and its memory are referred to as the host, and the GPU and its memory as the device.

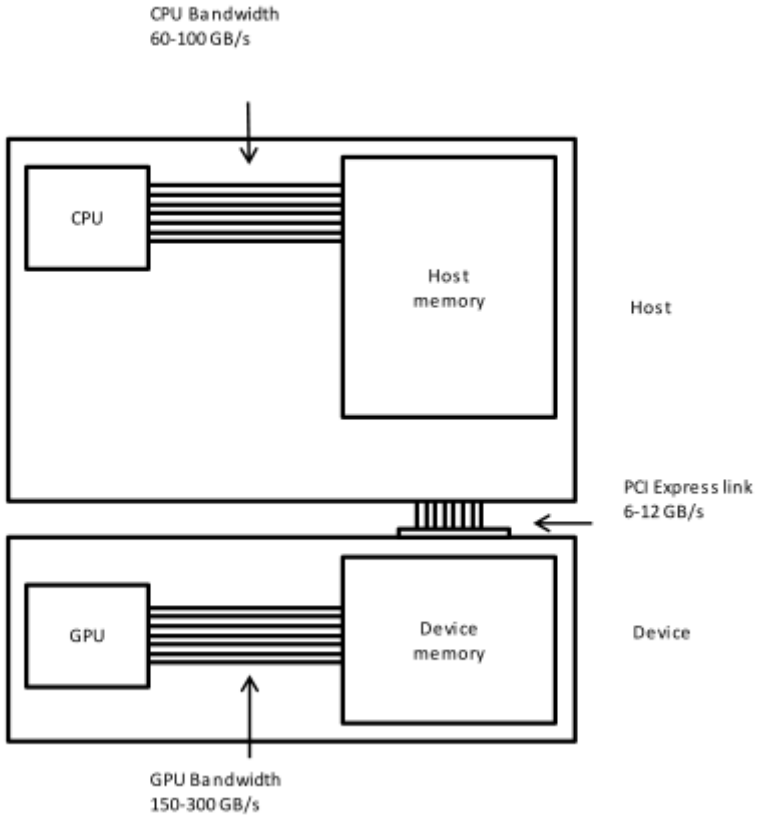


Figure adapted from the Carpentry GPU Programming lesson.

A look at the TOP500 list

The TOP500 project ranks and details the 500 most powerful non-distributed computer systems in the world. The project was started in 1993 and publishes an updated list of the supercomputers twice a year. The snapshot below shows the top-5 HPC systems as of June 2025, where the columns show:

- **Cores** - Number of processors
- **Rmax** - Maximal LINPACK performance achieved
- **Rpeak** - Theoretical peak performance
- **Power** - Power consumption

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	**JUPITER Booster** - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, RedHat Enterprise Linux, EVIDEN EuroHPC/FZJ Germany	4,801,344	793.40	930.00	13,088
5	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	

Snapshot from the TOP500 list from June, 2025.

All systems in the top-5 positions contain GPUs from AMD, Intel, or NVIDIA.

Why GPUs?

- **Speed**: GPU computing can significantly accelerate many types of scientific workloads.
- **Improved energy efficiency**: Compared to CPUs, GPUs can perform more calculations per watt of power consumed, which can result in significant energy savings. This is indeed evident from the GREEN500 list.
- **Cost-effectiveness**: GPUs can be more cost-effective than traditional CPU-based systems for certain workloads.

Limitations and drawbacks

- **Only for certain workloads**: Not all workloads can be efficiently parallelized and accelerated on GPUs. Certain types of workloads, such as those with irregular data access patterns or high branching behavior, may not see significant performance improvements on GPUs.
- **Steeper learning curve**: Depending on the GPU programming API that you choose, GPU computing could require specialized skills in GPU programming and knowledge of GPU architecture, leading to a steeper learning curve compared to CPU programming.

Fortunately, if you study this training material closely you will become productive with GPU programming quickly!

Keypoints

- GPUs are accelerators for some types of tasks
- Highly parallelizable compute-intensive tasks are suitable for GPUs
- GPU-based systems dominate the top spots of the TOP500 list
- New programming skills are needed to use GPUs efficiently

The GPU hardware and software ecosystem

❓ Questions

- What are the differences between GPUs and CPUs?
- What GPU software stacks are available? What do they provide?

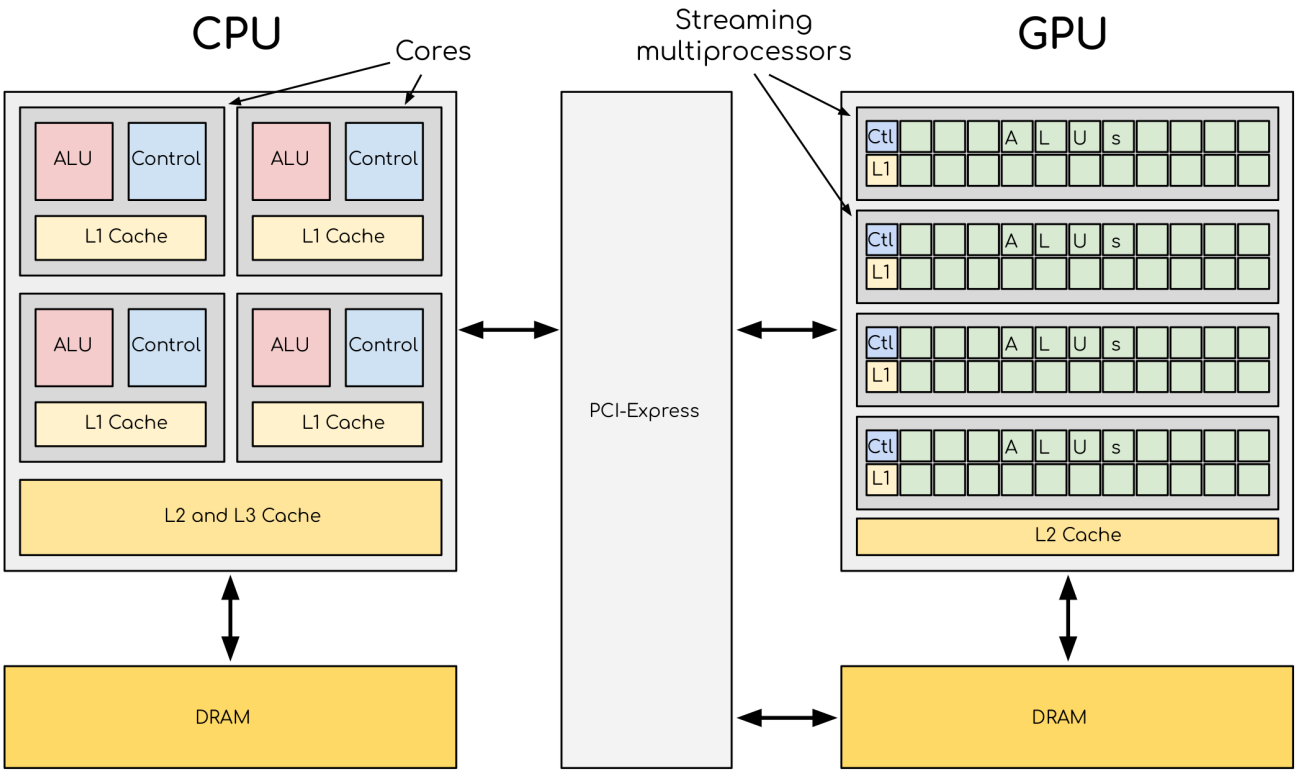
Objectives

- Understand the fundamental differences between GPUs and CPUs
- Explore the major GPU software suites available, such as CUDA, ROCm, and oneAPI, and gain a basic understanding of them

Instructor note

- 20 min teaching
- 0 min exercises

Overview of GPU hardware



A comparison of the CPU and GPU architecture. CPU (left) has complex core structure and pack several cores on a single chip. GPU cores are very simple in comparison, they also share data and control between each other. This allows to pack more cores on a single chip, thus achieving very high compute density.

In short

- Accelerators offer high performance due to their scalability and high density of compute elements.
- They have separate circuit boards connected to CPUs via PCIe bus, with their own memory.
- CPUs copy data from their own memory to the GPU memory, execute the program, and copy the results back.
- GPUs run thousands of threads simultaneously, quickly switching between them to hide memory operations.
- Effective data management and access pattern is critical on the GPU to avoid running out of memory.

Accelerators are a separate main circuit board with the processor, memory, power management, etc. It is connected to the motherboard with CPUs via PCIe bus. Having its own memory means that the data has to be copied to and from it (not neceseraly true anymore). CPU acts as a main processor, controlling the execution workflow. It copies the data from its own memory to the GPU memory, executes the program and copies the results back. GPUs runs tens of thousands of threads simultaneously on thousands of cores and does not do much of the data management. With many cores trying to access the memory simultaneously and with little cache available, the accelerator can run out of memory very

quickly. This makes the data management and its access pattern is essential on the GPU. Accelerators like to be overloaded with the number of threads, because they can switch between threads very quickly. This allows to hide the memory operations: while some threads wait, others can compute.

A very important feature of the accelerators is their scalability. Computational cores on accelerators are usually grouped into multiprocessors. The multiprocessors share the data and logical elements. This allows to achieve a very high density of compute elements on a GPU. This also allows the scaling: more multiprocessors means more raw performance and this is very easy to achieve with more transistors available.

How do GPUs differ from CPUs?

CPUs and GPUs were designed with different goals in mind. While the CPU is designed to excel at executing a sequence of operations, called a thread, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing many thousands of them in parallel. GPUs were initially developed for highly-parallel task of graphic processing and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. More transistors dedicated to data processing is beneficial for highly parallel computations; the GPU can hide memory access latencies with computation, instead of relying on large data caches and complex flow control to avoid long memory access latencies, both of which are expensive in terms of transistors.

CPU	GPU
General purpose	Highly specialized for parallelism
Good for serial processing	Good for parallel processing
Great for task parallelism	Great for data parallelism
Low latency per thread	High-throughput
Large area dedicated cache and control	Hundreds of floating-point execution units

GPU platforms

GPUs come together with software stacks or APIs that work in conjunction with the hardware and give a standard way for the software to interact with the GPU hardware. They are used by software developers to write code that can take advantage of the parallel processing power of the GPU, and they provide a standard way for software to interact with the GPU hardware. Typically, they provide access to low-level functionality, such as memory management, data transfer between the CPU and the GPU, and the scheduling and execution of parallel processing tasks on the GPU. They may also provide higher level functions and libraries optimized for specific HPC workloads, like linear algebra or fast Fourier transforms. Finally, in order to facilitate the developers to optimize and write correct codes, debugging and profiling tools are also included.

NVIDIA, *AMD*, and *Intel* are the major companies which design and produces GPUs for HPC providing each its own suite **CUDA**, **ROCm**, and respectively **oneAPI**. This way they can offer optimization, differentiation (offering unique features tailored to their devices), vendor lock-in, licensing, and royalty fees, which can result in better performance, profitability, and customer loyalty. There are also cross-platform APIs such **DirectCompute** (only for Windows operating system), **OpenCL**, and **SYCL**.

CUDA - In short

- **CUDA: NVIDIA's parallel computing platform**
  - Components: CUDA Toolkit & CUDA driver
  - Supports C, C++, and Fortran languages
- **CUDA API Libraries: cuBLAS, cuFFT, cuRAND, cuSPARSE**
  - Accelerate complex computations on GPUs
- **Compilers: nvcc, nvc, nvc++, nvfortran**
  - Support GPU and multicore CPU programming
  - Compatible with OpenACC and OpenMP
- **Debugging tools: cuda-gdb, compute-sanitizer**
  - Debug GPU and CPU code simultaneously
  - Identify memory access issues
- **Performance analysis tools: NVIDIA Nsight Systems, NVIDIA Nsight Compute**
  - Analyze system-wide and kernel-level performance
  - Optimize CPU and GPU usage, memory bandwidth, instruction throughput
- Comprehensive CUDA ecosystem with extensive tools and features

ROCm - In short

- **ROCm: Open software platform for AMD accelerators**
  - Built for open portability across multiple vendors and architectures
  - Offers libraries, compilers, and development tools for AMD GPUs
  - Supports C, C++, and Fortran languages
  - Support GPU and multicore CPU programming
- **Debugging:** `roc-gdb` **command line tool**
  - Facilitates debugging of GPU programs
- **Performance analysis:** `rocprof` **and** `roctracer` **tools**
  - Analyze and optimize program performance

- Supports various heterogeneous programming models such as **HIP**, **OpenMP**, and **OpenCL**
- **Heterogeneous-Computing Interface for Portability (HIP)**
    - Enables source portability for NVIDIA and AMD platforms, Intel in plan
    - Provides `hipcc` compiler driver and runtime libraries
- **Libraries: Prefixed with `roc` for AMD platforms**
    - Can be called directly from HIP
    - `hip`-prefixed wrappers ensure portability with no performance cost

oneAPI - In short

- **Intel oneAPI: Unified software toolkit for optimizing and deploying applications across various architectures**
    - Supports CPUs, GPUs, and FPGAs
    - Enables code reusability and performance portability
- **Intel oneAPI Base Toolkit: Core set of tools and libraries for high-performance, data-centric applications**
    - Includes C++ compiler with SYCL support
    - Features Collective Communications Library, Data Analytics Library, Deep Neural Networks Library, and more
- **Additional toolkits: Intel oneAPI HPC Toolkit**
    - Contains compilers, debugging tools, MPI library, and performance analysis tool
- **Multiple programming models and languages supported:**
    - OpenMP, Classic Fortran, C++, SYCL
    - Unless custom Intel libraries are used, the code is portable to other OpenMP and SYCL frameworks
- **DPC++ Compiler: Supports Intel, NVIDIA, and AMD GPUs**
    - Targets Intel GPUs using oneAPI Level Zero interface
    - Added support for NVIDIA GPUs with CUDA and AMD GPUs with ROCm
- Debugging and performance analysis tools: Intel Adviser, Intel Vtune Profiler, Cluster Checker, Inspector, Intel Trace Analyzer and Collector, Intel Distribution for GDB
- **Comprehensive and unified approach to heterogeneous computing**
    - Abstracts complexities and provides consistent programming interface
    - Promotes code reusability, productivity, and performance portability

CUDA

Compute Unified Device Architecture is the parallel computing platform from NVIDIA. The CUDA API provides a comprehensive set of functions and tools for developing high-performance applications that run on NVIDIA GPUs. It consists of two main components: the CUDA Toolkit and the CUDA driver. The toolkit provides a set of libraries, compilers, and development tools for programming and optimizing CUDA applications, while the driver is responsible for communication between the host CPU and the device GPU. CUDA is designed to work with programming languages such as C, C++, and Fortran.

CUDA API provides many highly optimize libraries such as: **cuBLAS** (for linear algebra operations, such a dense matrix multiplication), **cuFFT** (for performing fast Fourier transforms), **cuRAND** (for generating pseudo-random numbers), **cuSPARSE** (for sparse matrices operations). Using these libraries, developers can quickly and easily accelerate complex computations on NVIDIA GPUs without having to write low-level GPU code themselves.

There are several compilers that can be used for developing and executing code on NVIDIA GPUs: **nvcc**. The latest versions are based on the widely used LLVM (low level virtual machine) open source compiler infrastructure. nvcc produces optimized code for NVIDIA GPUs and drives a supported host compiler for AMD, Intel, OpenPOWER, and Arm CPUs.

In addition to this are provided **nvc** (C11 compiler), **nvc++** (C++17 compiler), and **nvfortran** (ISO Fortran 2003 compiler). These compilers can as well create code for execution on the NVIDIA GPUs, and also support GPU and multicore CPU programming with parallel language features, OpeanACC and OpenMP.

When programming mistakes are inevitable they have to be fixed as soon as possible. The CUDA toolkit includes the command line tool **cuda-gdb** which can be used to find errors in the code. It is an extension to GDB, the GNU Project debugger. The existing GDB debugging features are inherently present for debugging the host code, and additional features have been provided to support debugging CUDA device code, allowing simultaneous debugging of both GPU and CPU code within the same application. The tool provides developers with a mechanism for debugging CUDA applications running on actual hardware. This enables developers to debug applications without the potential variations introduced by simulation and emulation environments.

In addition to this the command line tool **compute-sanitizer** can be used to look exclusively for memory access problems: unallocated buffers, out of bounds accesses, race conditions, and uninitialized variables.

Finally, in order to utilize the GPUs at maximum some performance analysis tools. NVIDIA provides NVIDIA Nsight Systems and NVIDIA Nsight Compute tools for helping the developers to optimize their applications. The former, NVIDIA Nsight Systems, is a system-wide performance analysis tool that provides detailed metrics on both CPU and GPU usage, memory bandwidth, and other system-level metrics. The latter, NVIDIA Nsight Compute, is a kernel-level performance analysis tool that allows developers to analyze the performance of

individual CUDA kernels. It provides detailed metrics on kernel execution, including memory usage, instruction throughput, and occupancy. These tools have graphical which can be used for all steps of the performance analysis, however on supercomputers it is recommended to use the command line interface for collecting the information needed and then visualize and analyse the results using the graphical interface on personal computers.

Apart from what was presented above there are many others tools and features provided by NVIDIA. The CUDA ecosystem is very well developed.

ROCm

ROCm is an open software platform allowing researchers to tap the power of AMD accelerators. The ROCm platform is built on the foundation of open portability, supporting environments across multiple accelerator vendors and architectures. In some way it is very similar to CUDA API. It contains libraries, compilers, and development tools for programming and optimizing programs for AMD GPUs. For debugging, it provides the command line tool `rocgdb`, while for performance analysis `rocprof` and `roctracer`. In order to produce code for the AMD GPUs, one can use the Heterogeneous-Computing Interface for Portability (HIP). HIP is a C++ runtime API and a set of tools that allows developers to write portable GPU-accelerated code for both NVIDIA and AMD platforms. It provides the `hipcc` compiler driver, which will call the appropriate toolchain depending on the desired platform. On the AMD ROCm platform, HIP provides a header and runtime library built on top of the HIP-Clang (ROCm compiler). On an NVIDIA platform, HIP provides a header file which translates from the HIP runtime APIs to CUDA runtime APIs. The header file contains mostly inlined functions and thus has very low overhead. The code is then compiled with `nvcc`, the standard C++ compiler provided with CUDA. On AMD platforms, libraries are prefixed by `roc`, which can be called directly from HIP. In order to make portable calls, one can call the libraries using `hip`-prefixed wrappers. These wrappers can be used at no performance cost and ensure that HIP code can be used on other platforms with no changes. Libraries included in the ROCm, are almost one-to-one equivalent to the ones supplied with CUDA.

ROCm also integrates with popular machine learning frameworks such as TensorFlow and PyTorch and provides optimized libraries and drivers to accelerate machine learning workloads on AMD GPUs enabling the researchers to leverage the power of ROCm and AMD accelerators to train and deploy machine learning models efficiently.

oneAPI

Intel oneAPI is a unified software toolkit developed by Intel that allows developers to optimize and deploy applications across a variety of architectures, including CPUs, GPUs, and FPGAs. It provides a comprehensive set of tools, libraries, and frameworks, enabling developers to leverage the full potential of heterogeneous computing environments. With oneAPI, the developers can write code once and deploy it across different hardware targets without the need for significant modifications or rewriting. This approach promotes code

reusability, productivity, and performance portability, as it abstracts the complexities of heterogeneous computing and provides a consistent programming interface based on open standards.

The core of suite is **Intel oneAPI Base Toolkit**, a set of tools and libraries for developing high-performance, data-centric applications across diverse architectures. It features an industry-leading C++ compiler that implements SYCL, an evolution of C++ for heterogeneous computing. It includes the **Collective Communications Library**, the **Data Analytics Library**, the **Deep Neural Networks Library**, the **DPC++/C++ Compiler**, the **DPC++ Library**, the **Math Kernel Library**, the **Threading Building Blocks**, debugging tool **Intel Distribution for GDB**, performance analysis tools **Intel Adviser** and **Intel Vtune Profiler**, the **Video Processing Library**, **Intel Distribution for Python**, the **DPC++ Compatibility Tool**, the **FPGA Add-on for oneAPI Base Toolkit**, the **Integrated Performance Primitives**. This can be complemented with additional toolkits. The **Intel oneAPI HPC Toolkit** contains **DPC++/C++ Compiler**, **Fortran** and **C++** Compiler Classic, debugging tools **Cluster Checker** and **Inspector**, **Intel MPI Library**, and performance analysis tool **Intel Trace Analyzer and Collector**.

oneAPI supports multiple programming models and programming languages. It enables developers to write **OpenMP** codes targeting multi-core CPUs and Intel GPUs using the Classic Fortran and C++ compilers and as well **SYCL** programs for GPUs and FPGAs using the **DPC++** compiler. Initially, the **DPC++** compiler only targeted Intel GPUs using the **oneAPI Level Zero** low-level programming interface, but now support for NVIDIA GPUs (using CUDA) and AMD GPUs (using ROCm) has been added. Overall, Intel oneAPI offers a comprehensive and unified approach to heterogeneous computing, empowering developers to optimize and deploy applications across different architectures with ease. By abstracting the complexities and providing a consistent programming interface, oneAPI promotes code reusability, productivity, and performance portability, making it an invaluable toolkit for developers in the era of diverse computing platforms.

Differences and similarities

GPUs in general support different features, even among the same producer. In general newer cards come with extra features and sometimes old features are not supported anymore. It is important when compiling to create binaries targeting the specific architecture when compiling. A binary built for a newer card will not run on older devices, while a binary build for older devices might not run efficiently on newer architectures. In CUDA the compute capability which is targeted is specified by the `-arch=sm_XY`, where `X` specifies the major architecture and it is between 1 and 9, and `Y` the minor. When using HIP on NVIDIA platforms one needs to use compiling option `--gpu-architecture=sm_XY`, while on AMD platforms `--offload-arch=gfxabc` (where `abc` is the architecture code such as `90a` for the MI200 series or `908` for MI100 series). Note that in the case of portable (single source) programs one would specify `openmp` as well as target for compilation, enabling to run the same code on multicore CPU.

Terminology

NVIDIA	AMD	Intel
Streaming processor/streaming core	SIMD lane	Processing element
SIMT unit	SIMD unit	Vector engine (XVE)
Streaming Multiprocessor (SM)	Computing Unit (CU)	Xe-core / Execution unit (EU)
GPU processing clusters (GPC)	Compute Engine	Xe-slice

Please keep in mind, that this table is only a rough approximation. Each GPU architecture is different, and it's impossible to make a 1-to-1 mapping between terms used by different vendors.

Summary

- GPUs are designed to execute thousands of threads simultaneously, making them highly parallel processors. In contrast, CPUs excel at executing a smaller number of threads in parallel.
- GPUs allocate a larger portion of transistors to data processing rather than data caching and flow control. This prioritization of data processing enables GPUs to effectively handle parallel computations and hide memory access latencies through computation.
- GPU producers provide comprehensive toolkits, libraries, and compilers for developing high-performance applications that leverage the parallel processing power of GPUs. Examples include CUDA (NVIDIA), ROCm (AMD), and oneAPI (Intel).
- These platforms offer debugging tools (e.g., `cuda-gdb`, `rocgdb`) and performance analysis tools (e.g., NVIDIA Nsight Systems, NVIDIA Nsight Compute, `rocprof`, `roctracer`) to facilitate code optimization and ensure efficient utilization of GPU resources.

Exercises

✍️ GPUs and memory

Which statement about the relationship between GPUs and memory is true?

- A. GPUs are not affected by memory access latencies.
- B. GPUs can run out of memory quickly with many cores trying to access the memory simultaneously.
- C. GPUs have an unlimited cache size.
- D. GPUs prefer to run with a minimal number of threads to manage memory effectively.

✓ Solution

The correct answer is B). This is true because GPUs run many threads simultaneously on thousands of cores, and with limited cache available, this can lead to the GPU running out of memory quickly if many cores are trying to access the memory simultaneously. This is why data management and access patterns are essential in GPU computing.

Keypoints

- GPUs vs. CPUs, key differences between them
- GPU software suites, support specific GPU features, programming models, compatibility
- Applications of GPUs

What problems fit to GPU?

❓ Questions

- What are the strengths and weaknesses of GPUs?
- What makes a particular problem suitable for GPU-porting?
- Why are GPUs so ubiquitous in machine learning applications?

Objectives

- Get a feeling for the type of use cases that GPUs excel at.

Instructor note

- 10 min teaching
- 10 min exercises

What are GPUs good for?

Answer from Stack Exchange:

> *From a metaphorical point of view, the GPU can be seen as a person lying on a bed of nails. The person lying on top is the data and in the base of each nail there is a processor, so the nail is actually an arrow pointing from processor to memory. All nails are in a regular pattern, like a grid. If the body is well spread, it feels good (performance is good), if the body only touches some spots of the nail bed, then the pain is bad (bad performance).*

GPU computing is well-suited to problems that involve large amounts of data parallelism. Specifically, you can expect good performance on GPUs for:

- **Large-scale matrix and vector operations**: Common in machine learning, scientific computing, and image processing.
- **Fourier transforms**: Also common in machine learning, scientific computing, and image processing.
- **Monte Carlo simulations**: Used across finance, physics, and other fields to simulate complex systems.
- **Molecular dynamics simulations**: Used in chemistry, biochemistry and physics.
- **Computational fluid dynamics**: Used in engineering, physics, and other fields.
- **Convolutional neural networks** and **computer vision algorithms**.
- **Big data analytics**: Clustering, classification, regression, etc.
- **Graphics rendering**: Original use-case for GPUs.

What are GPUs not good for?

Not all programming problems can efficiently leverage the parallelism offered by GPUs. Some types of problems that do not fit well on a GPU include:

- **Sequential tasks**: Problems that require a series of dependent steps, where each step relies on the outcome of the previous step, are not well-suited for parallel processing. Examples include recursive algorithms, certain dynamic programming problems, and some graph traversal algorithms.
- **Fine-grained branching**: GPUs perform best when the code being executed across different threads follows a similar control flow. When there is extensive branching (i.e., many `if` statements) within a kernel or algorithm, performance may suffer due to the divergence in execution paths among the GPU threads.
- **Low arithmetic intensity**: GPUs excel at performing a large number of mathematical operations quickly. If a problem has low arithmetic intensity (i.e., a low ratio of arithmetic operations to memory accesses), the GPU may not be able to efficiently utilize its computational power, leading to underperformance.
- **Small data sets**: If the problem involves a small data set that does not require significant parallelism, using a GPU may not result in noticeable performance gains. In such cases, the overhead of transferring data between the CPU and GPU, and the time spent initializing the GPU, may outweigh any potential benefits.
- **Limited parallelism**: Some algorithms have inherent limitations on the degree of parallelism that can be achieved. In these cases, using a GPU may not lead to significant performance improvements.
- **Memory-bound problems**: GPUs generally have less memory available compared to CPUs, and their memory bandwidth can be a limiting factor. If a problem requires a large amount of memory or involves memory-intensive operations, it may not be well-suited for a GPU.

Examples of GPU acceleration

To give a flavor of what type of performance gains we can achieve by porting a calculations to a GPU (if we're lucky!), let's look at a few case examples.

💬 Effect of array size

Consider the case of matrix multiplication in the Julia language:

```
using AMDGPU
using BenchmarkTools

N = [9, 10, 11, 12]

for n in N
   A = rand(2^n, 2^n); A_d = ROCArray(A);

   @btime $A * $A;

   @btime begin
      $A_d * $A_d;
      AMDGPU.synchronize()
   end
end
```

- How much faster do you think the GPU version is compared to running on a single CPU core?
- Julia automatically parallelises matrix multiplication over available CPU cores. Will the GPU version be faster than running on 64 cores?
- Does the size of the array affect how much the performance improves?

✔ Solution

Example results from running on LUMI (MI250X AMD GPU, 64-core AMD Trento CPUs):

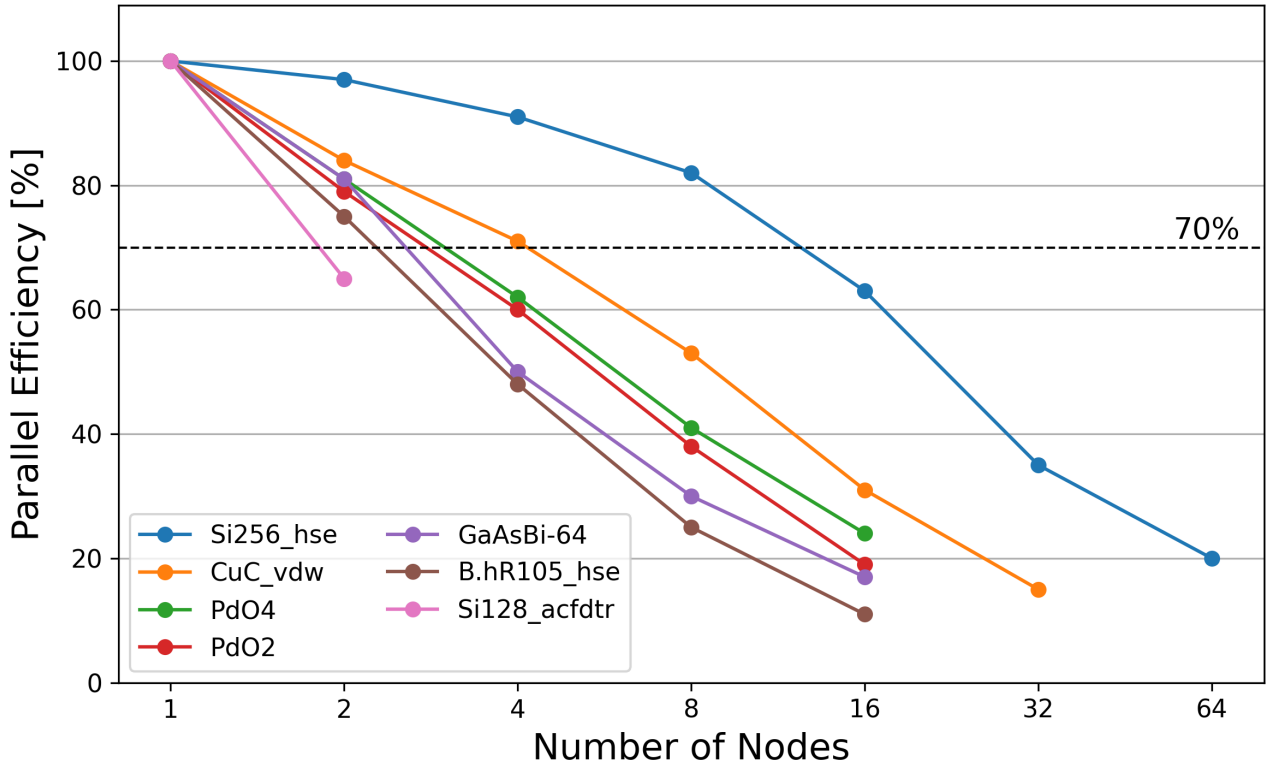
GPU acceleration for matrix multiply in Julia

Matrix size	1 CPU core	64 CPU cores	1 GPU	GPU speedup
(512, 512)	5.472 ms	517.722 µs	115.805 µs	~47x / ~5x
(1024, 1024)	43.364 ms	2.929 ms	173.316 µs	~250x / ~17x
(2048, 2048)	344.364 ms	30.081 ms	866.348 µs	~400x / ~35x
(4096, 4096)	3.221 s	159.563 ms	5.910 ms	~550x / ~27x

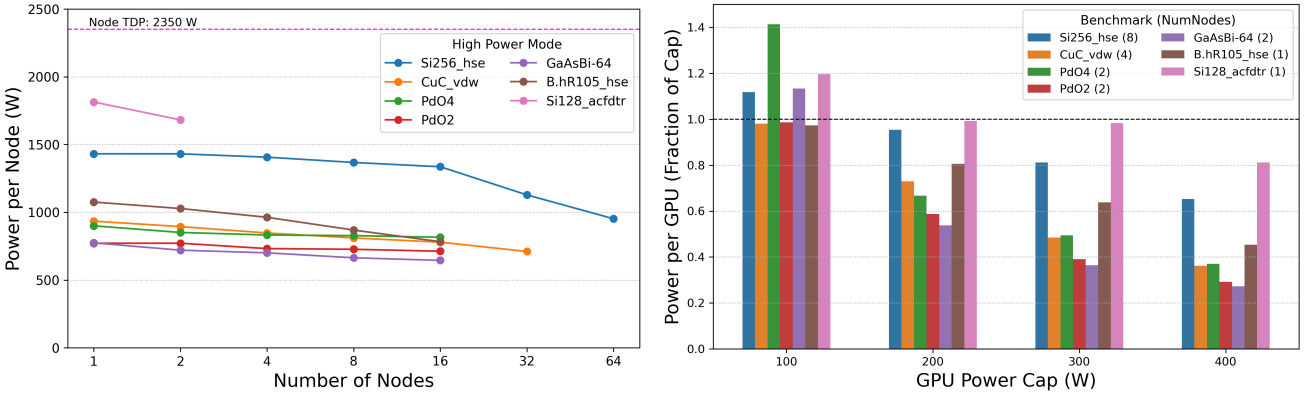
Electronic structure calculations

VASP is a popular software package used for electronic structure calculations. The figures below show the speedup observed in a recent benchmark study on the VASP Power Profiles on NVIDIA A100 GPUs, which was conducted on the Perlmutter system at NERSC. An analysis of total energy usage demonstrated that VASP's power usage varies significantly

with different workloads, more so than with parallel concurrency. Additionally, power capping GPUs to 50% of their Thermal Design Power can be applied to most VASP workloads with less than a 10% performance loss.



Parallel efficiency of VASP on seven test cases representing diverse VASP production workloads and ensuring a comprehensive coverage of various code paths, elements, and problem sizes.



(Left) Power usage of seven representative VASP workloads. The horizontal axis shows number of nodes used, and vertical axis shows high power mode per node. (Right) Power consumed per GPU when running VASP under four different power caps: 400 W (default), 300 W, 200 W, and 100 W. Horizontal axis shows power caps applied to GPUs, and vertical axis shows high power mode per GPU as a fraction of applied cap. The dashed horizontal line represents applied power cap. Each benchmark was run with a node count optimizing runtime while remaining above 70% parallel efficiency.

Computational Chemistry

A great deal of computational resources are spent in Quantum Chemical calculations which involve the solution of the Hartree-Fock eigenvalue problem, which requires the diagonalization of the Fock matrix whose elements are given by:

$$F_{\alpha \beta} = H^{\textrm{core}}_{\alpha \beta} + \sum_{\gamma \delta}D_{\gamma \delta} \left [(\alpha \beta|\gamma \delta) - \frac{1}{2} (\alpha \delta|\gamma \beta) \right],$$

The first term is related to the one electron contributions and the second term is related to the electron repulsion integrals (ERIs), in parenthesis, weighted by the by the density matrix $D_{\gamma \delta}$. One of the most expensive parts in the solution of the Hartree-Fock equations is the processing (digestion) of the ERIs, one algorithm to do this task is as follows:

```
for α = 1,...,N do
  for β = α,...,N do
    p = α(α+1)/2+β;
    for γ = 1,...,N do
      for δ = γ,...,N do
        q = γ(γ+1)/2+δ;
        if p ≤ q then
          compute (αβ|γδ);
          F_αβ ← D_γδ(αβ|γδ);
          F_γδ ← D_αβ(αβ|γδ);
          F_αγ ← D_αγ(αβ|γδ);
          F_αδ ← D_αδ(αβ|γδ);
          F_βγ ← D_βγ(αβ|γδ);
          F_βδ ← D_βδ(αβ|γδ);
        end
      end
    end
  end
end
```

Algorithm for processing ERIs [see JCTC, 17, 7486, (2021) for details]

This algorithm is suitable for GPUs as it involves many arithmetic operations. In addition to this, there are symmetries and properties of the integrals that could be used to rearrange the loops in an efficient manner that fit GPU architectures.

Humanities

A brief introduction into some of the work that is being done in the humanities that can benefit from utilizing GPUs.

Language models and NLP (natural language processing)

With the recent popularity of ChatGPT, the use of language models has come into the mainstream, however such models have been used in the humanities many years already. One of the biggest goals of humanities researchers is working with textual data which has increased exponentially over recent years due to the rise in social media. Analyzing such textual data to gain insights into questions of sociology, linguistics and various other fields have become increasingly reliant on using language models. Along with language models, the need for GPU access has become essential.

Archeology

The field of archeology also makes use of GPUs in their 3D modelling and rendering work. The biggest problem with archeological sites is that once they are excavated, they are destroyed, so any researchers who aren't present at the site, would lose valuable insights into

how it looked when it was found. However, with recent developments in technology and accessibility to high-performance computing, they are able to generate extremely detailed renderings of the excavation sites which act as a way to preserve the site for future researchers to gain critical insights and contribute to the research.

Cognitive Science

Techniques such as Markov Chain Monte Carlo (MCMC) sampling have proven to be invaluable in studies that delve into human behavior or population dynamics. MCMC sampling allows researchers to simulate and analyze complex systems by iteratively sampling from a Markov chain, enabling the exploration of high-dimensional parameter spaces. This method is particularly useful when studying human behavior, as it can capture the inherent randomness and interdependencies that characterize social systems. By leveraging MCMC sampling, researchers can gain insights into various aspects of human behavior, such as decision-making, social interactions, and the spread of information or diseases within populations.

By offloading the computational workload to GPUs, researchers can experience substantial speedup in the execution of MCMC algorithms. This speedup allows for more extensive exploration of parameter spaces and facilitates the analysis of larger datasets, leading to more accurate and detailed insights into human behavior or population dynamics. Examples of studies done using these methods can be found at the Center for Humanities Computing Aarhus (CHCAA) and Interacting Minds Centre (IMC) at Aarhus University.

Exercises

✍ Discussion

- What type of problems have you used GPUs for?
- How large was the performance boost?

✍ Good and bad use cases for GPU porting

Which of the following computational tasks is likely to gain the least performance benefit from being ported to a GPU?

1. Training a large, deep neural network.
2. Performing a Monte Carlo simulation with a large number of independent trials.
3. Executing an algorithm with heavy use of recursion and frequent branching.
4. Processing a large image with a convolutional filter.

✔ Solution

The right answer is option 3. GPUs do not handle recursion and branching as effectively as more data-heavy algorithms.

Keypoints

- GPUs excel in processing tasks with high data parallelism, such as large-scale matrix operations, Fourier transforms, and big data analytics.
- GPUs struggle with sequential tasks, problems with extensive control flow divergence, low arithmetic intensity tasks, small data sets, and memory-bound problems.

GPU programming concepts

❓ Questions

- What types of parallel computing is possible?
- How does data parallelism differ from task parallelism, and how are they utilized in parallel computing?
- How is the work parallelized and executed on GPUs?
- What are general considerations for an efficient code running on GPUs?

Objectives

- Understand parallel computing principles and architectures.
- Differentiate data parallelism from task parallelism.
- Learn the GPU execution model.
- Parallelize and execute work on GPUs.
- Develop efficient GPU code for high performance.

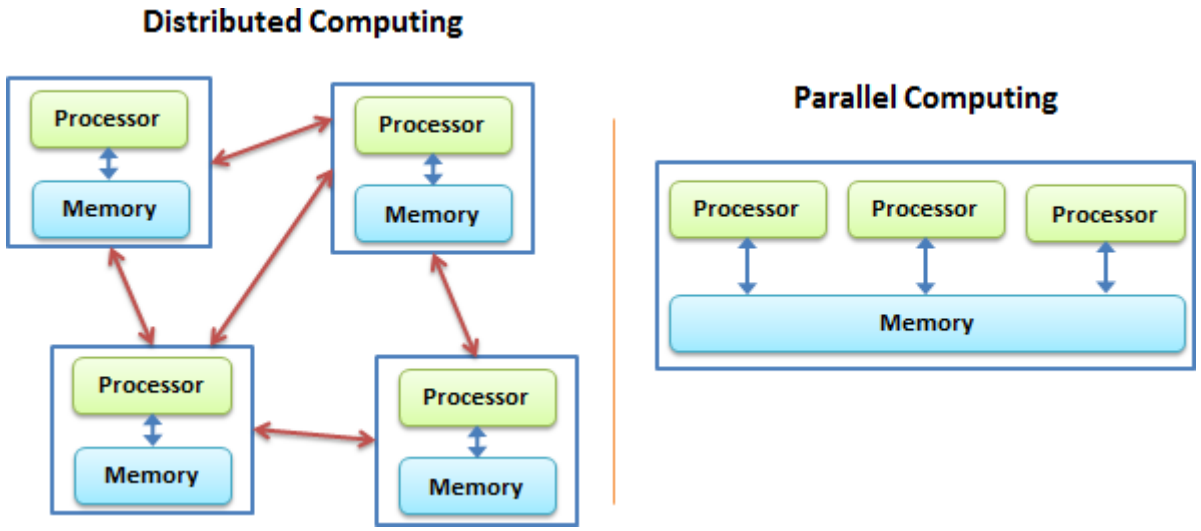
Instructor note

- 25 min teaching
- 0 min exercises

Different types of parallelism

Distributed- vs. Shared-Memory Architecture

Most of computing problems are not trivially parallelizable, which means that the subtasks need to have access from time to time to some of the results computed by other subtasks. The way subtasks exchange needed information depends on the available hardware.

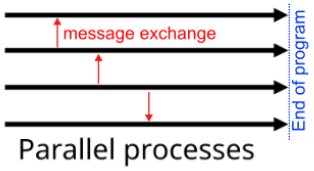


Distributed- vs shared-memory parallel computing.

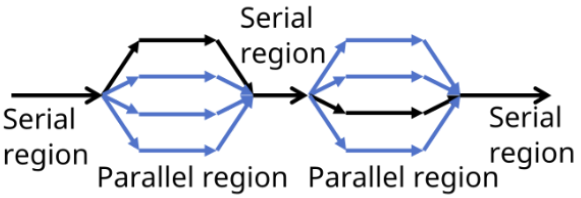
In a distributed memory environment each processing unit operates independently from the others. It has its own memory and it **cannot** access the memory in other nodes. The communication is done via network and each computing unit runs a separate copy of the operating system. In a shared memory machine all processing units have access to the memory and can read or modify the variables within.

Processes and Threads

The type of environment (distributed- or shared-memory) determines the programming model. There are two types of parallelism possible, process based and thread based.



Distributed memory parallelism



Shared memory parallelism

For distributed memory machines, a process-based parallel programming model is employed. The processes are independent execution units which have their *own memory* address spaces. They are created when the parallel program is started and they are only terminated at the end. The communication between them is done explicitly via message passing like MPI.

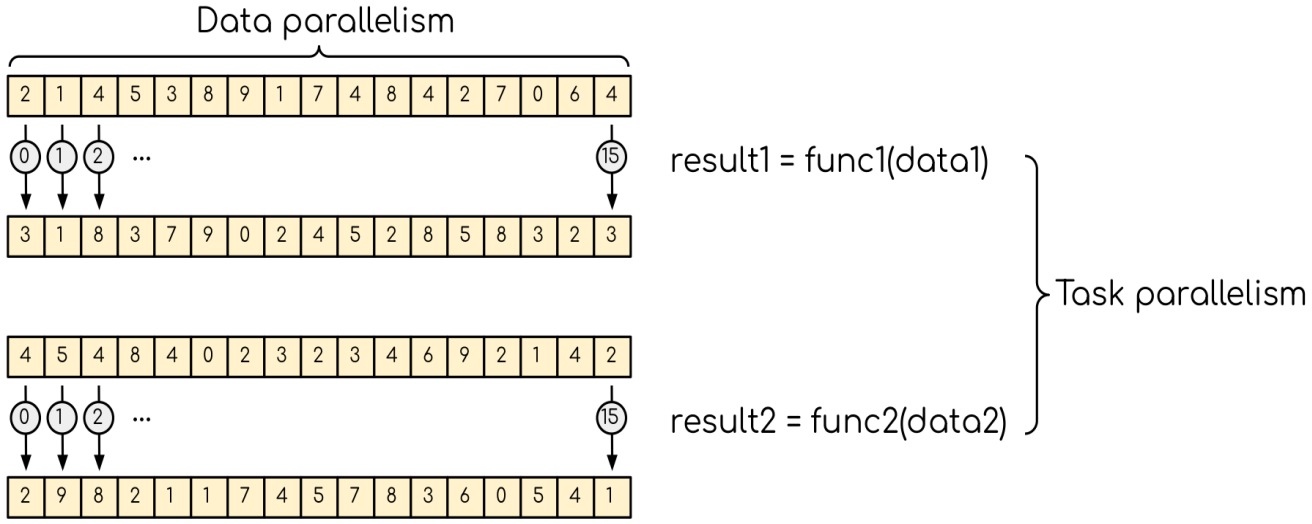
On the shared memory architectures it is possible to use a thread based parallelism. The threads are light execution units and can be created and destroyed at a relatively small cost. The threads have their own state information, but they *share* the *same memory* address space. When needed the communication is done though the shared memory.

Both approaches have their advantages and disadvantages. Distributed machines are relatively cheap to build and they have an “infinite “ capacity. In principle one could add more and more computing units. In practice the more computing units are used the more time consuming is the communication. The shared memory systems can achieve good

performance and the programming model is quite simple. However they are limited by the memory capacity and by the access speed. In addition in the shared parallel model it is much easier to create race conditions.

Exposing parallelism

There are two types of parallelism that can be explored. The data parallelism is when the data can be distributed across computational units that can run in parallel. The units process the data by applying the same or very similar operation to different data elements. A common example is applying a blur filter to an image — the same function is applied to all the pixels on an image. This parallelism is natural for the GPU, where the same instruction set is executed in multiple threads.



Data parallelism and task parallelism. The data parallelism is when the same operation applies to multiple data (e.g. multiple elements of an array are transformed). The task parallelism implies that there are more than one independent task that, in principle, can be executed in parallel.

Data parallelism can usually be explored by the GPUs quite easily. The most basic approach would be finding a loop over many data elements and converting it into a GPU kernel. If the number of elements in the data set is fairly large (tens or hundred of thousands elements), the GPU should perform quite well. Although it would be odd to expect absolute maximum performance from such a naive approach, it is often the one to take. Getting absolute maximum out of the data parallelism requires good understanding of how GPU works.

Another type of parallelism is a task parallelism. This is when an application consists of more than one task that requiring to perform different operations with (the same or) different data. An example of task parallelism is cooking: slicing vegetables and grilling are very different tasks and can be done at the same time. Note that the tasks can consume totally different resources, which also can be explored.

In short

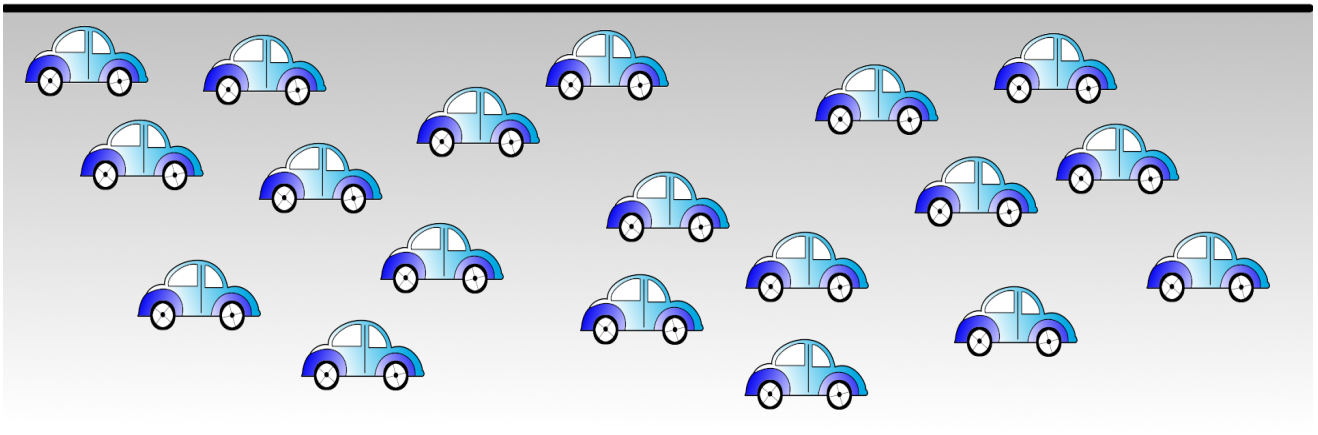
- Computing problems can be parallelized in distributed memory or shared memory architectures.

- In distributed memory, each unit operates independently, with no direct memory access between nodes.
- In shared memory, units have access to the same memory and can communicate through shared variables.
- Parallel programming can be process-based (distributed memory) or thread-based (shared memory).
- Process-based parallelism uses independent processes with separate memory spaces and explicit message passing.
- Thread-based parallelism uses lightweight threads that share the same memory space and communicate through shared memory.
- Data parallelism distributes data across computational units, processing them with the same or similar operations.
- Task parallelism involves multiple independent tasks that perform different operations on the same or different data.
- Task parallelism involves executing different tasks concurrently, leveraging different resources.

GPU Execution Model

In order to obtain maximum performance it is important to understand how GPUs execute the programs. As mentioned before a CPU is a flexible device oriented towards general purpose usage. It's fast and versatile, designed to run operating systems and various, very different types of applications. It has lots of features, such as better control logic, caches and cache coherence, that are not related to pure computing. CPUs optimize the execution by trying to achieve low latency via heavy caching and branch prediction.

CPU



GPU

Cars and roads analogy for the CPU and GPU behavior. The compact road is analogous to the CPU (low latency, low throughput) and the broader road is analogous to the GPU (high latency, high throughput).

In contrast the GPUs contain a relatively small amount of transistors dedicated to control and caching, and a much larger fraction of transistors dedicated to the mathematical operations. Since the cores in a GPU are designed just for 3D graphics, they can be made much simpler and there can be a very larger number of cores. The current GPUs contain thousands of CUDA cores. Performance in GPUs is obtain by having a very high degree of parallelism. Lots of threads are launched in parallel. For good performance there should be at least several times more than the number of CUDA cores. GPU threads are much lighter than the usual CPU threads and they have very little penalty for context switching. This way when some threads are performing some memory operations (reading or writing) others execute instructions.

CUDA Threads, Warps, Blocks

In order to understand the GPU execution model let's look at the so called *axpy* operation. On a single CPU core this operation would be executed in a serial manner in a *for/do* loop going over each element on the array, *id*, and computing *y[id]=y[id]+a*x[id]*.

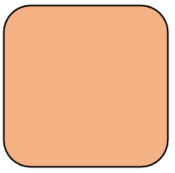
```
void axpy_(int n, double a, double *x, double *y)
{
    for(int id=0;id<n; id++) {
       y[id] += a * x[id];
    }
}
```

In order to perform the some operation on a GPU the program launches a function called *kernel*, which is executed simultaneously by tens of thousands of threads that can be run on GPU cores parallelly.

```
GPU_K void ker_axpy_(int n, double a, double *x, double *y, int id)
{
    y[id] += a * x[id]; // id<n
}
```

The programmers control how many instances of *ker_axpy_* are created and they have to make sure that all the elements are processed and also that no out of bounds accessed are happening.

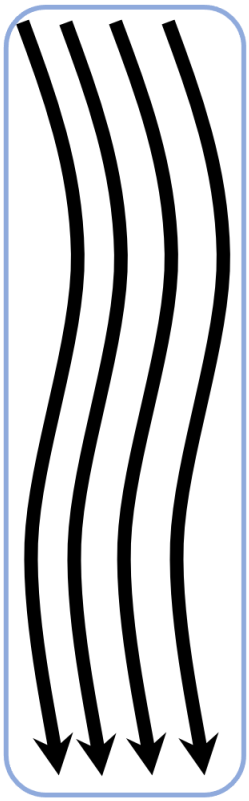
GPU threads are much lighter than the usual CPU threads and they have very little penalty for context switching. By "over-subscribing" the GPU there are threads that are performing some memory operations (reading or writing), while others execute instructions.



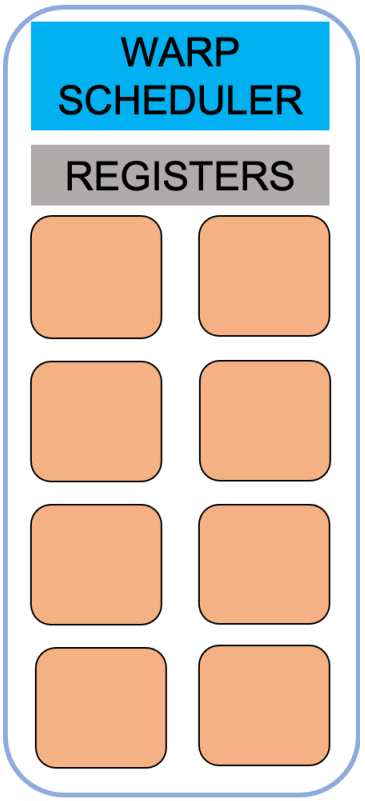
CUDA THREAD CUDA CORE

Every thread is associated with a particular intrinsic index which can be used to calculate and access memory locations in an array. Each thread has its context and set of private variables. All threads have access to the global GPU memory, but there is no general way to synchronize when executing a kernel. If some threads need data from the global memory which was modified by other threads the code would have to be split in several kernels because only at the completion of a kernel it is ensured that the writing to the global memory was completed.

Apart from being much light weighted there are more differences between GPU threads and CPU threads. GPU threads are grouped together in groups called warps. This done at hardware level.



WARP

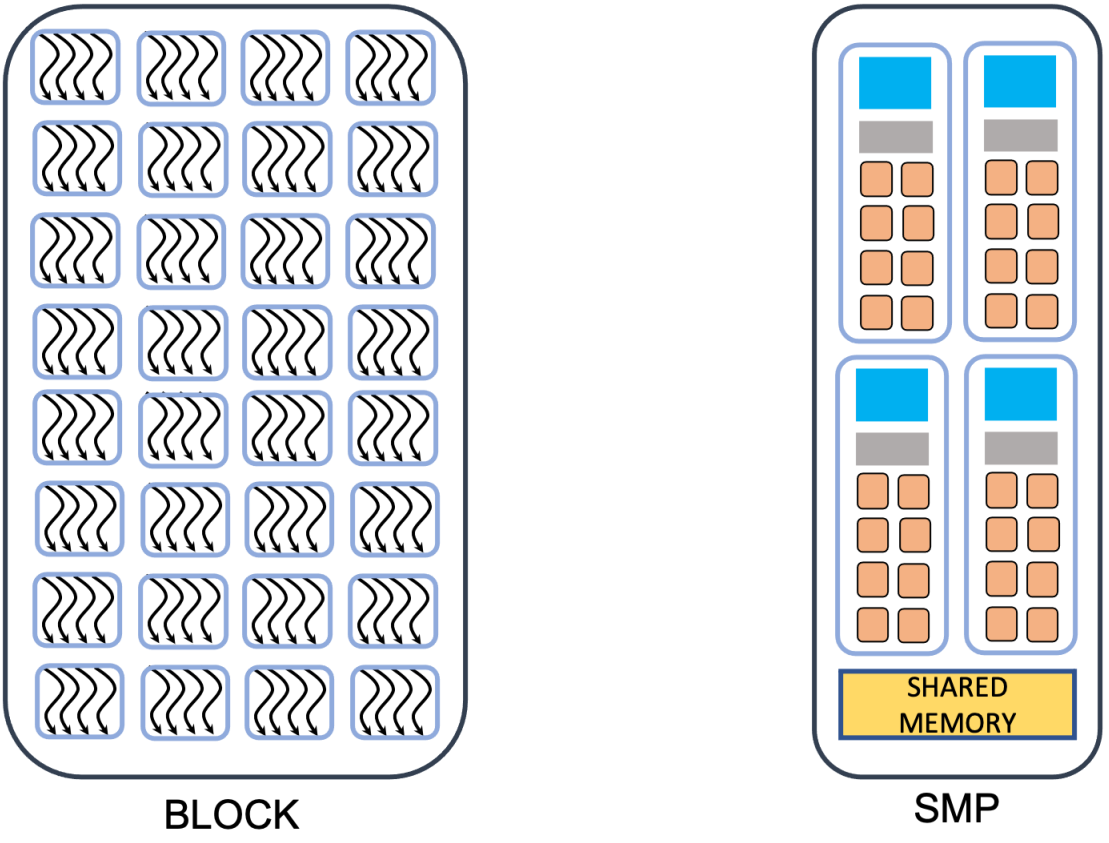


SIMT UNIT

All memory accesses to the GPU memory are as a group in blocks of specific sizes (32B, 64B, 128B etc.). To obtain good performance the CUDA threads in the same warp need to access elements of the data which are adjacent in the memory. This is called *coalesced* memory access.

On some architectures, all members of a warp have to execute the same instruction, the so-called “lock-step” execution. This is done to achieve higher performance, but there are some drawbacks. If an **if** statement is present inside a warp will cause the warp to be executed more than once, one time for each branch. When different threads within a single warp take different execution paths based on a conditional statement (if), both branches are executed sequentially, with some threads being active while others are inactive. On architectures without lock-step execution, such as NVIDIA Volta / Turing (e.g., GeForce 16xx-series) or newer, warp divergence is less costly.

There is another level in the GPU threads hierarchy. The threads are grouped together in so called blocks. Each block is assigned to one Streaming Multiprocessor (SMP) unit. A SMP contains one or more SIMT (single instruction multiple threads) units, schedulers, and very fast on-chip memory. Some of this on-chip memory can be used in the programs, this is called shared memory. The shared memory can be used to “cache” data that is used by more than one thread, thus avoiding multiple reads from the global memory. It can also be used to avoid memory accesses which are not efficient. For example in a matrix transpose operation, we have two memory operations per element and only can be coalesced. In the first step a tile of the matrix is saved read a coalesced manner in the shared memory. After all the reads of the block are done the tile can be locally transposed (which is very fast) and then written to the destination matrix in a coalesced manner as well. Shared memory can also be used to perform block-level reductions and similar collective operations. All threads can be synchronized at block level. Furthermore when the shared memory is written in order to ensure that all threads have completed the operation the synchronization is compulsory to ensure correctness of the program.



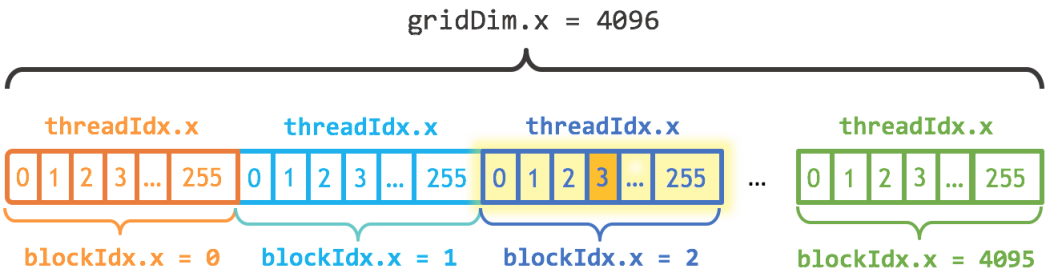
Finally, a block of threads can not be split among SMPs. For performance blocks should have more than one warp. The more warps are active on an SMP the better is hidden the latency associated with the memory operations. If the resources are sufficient, due to fast context switching, an SMP can have more than one block active in the same time. However these blocks can not share data with each other via the on-chip memory.

To summarize this section. In order to take advantage of GPUs the algorithms must allow the division of work in many small subtasks which can be executed in the same time. The computations are offloaded to GPUs, by launching tens of thousands of threads all executing the same function, *kernel*, each thread working on different part of the problem. The threads are executed in groups called *blocks*, each block being assigned to a SMP. Furthermore the threads of a block are divided in *warps*, each executed by SIMT unit. All threads in a warp execute the same instructions and all memory accesses are done collectively at warp level. The threads can synchronize and share data only at block level. Depending on the architecture, some data sharing can be done as well at warp level.

In order to hide latencies it is recommended to "over-subscribe" the GPU. There should be many more blocks than SMPs present on the device. Also in order to ensure a good occupancy of the CUDA cores there should be more warps active on a given SMP than SIMT units. This way while some warps of threads are idle waiting for some memory operations to complete, others use the CUDA cores, thus ensuring a high occupancy of the GPU.

In addition to this there are some architecture-specific features of which the developers can take advantage. Warp-level operations are primitives provided by the GPU architecture to allow for efficient communication and synchronization within a warp. They allow threads within a warp to exchange data efficiently, without the need for explicit synchronization. These warp-level operations, combined with the organization of threads into blocks and clusters, make it possible to implement complex algorithms and achieve high performance on the GPU. The cooperative groups feature introduced in recent versions of CUDA provides even finer-grained control over thread execution, allowing for even more efficient processing by giving more flexibility to the thread hierarchy. Cooperative groups allow threads within a block to organize themselves into smaller groups, called cooperative groups, and to synchronize their execution and share data within the group.

Below there is an example of how the threads in a grid can be associated with specific elements of an array




```
index = blockIdx.x  * blockDim.x + threadIdx.x
index =    (2)      *   (256)    +     (3)      = 515
```

The thread marked by orange color is part of a grid of threads size 4096. The threads are grouped in blocks of size 256. The “orange” thread has index 3 in the block 2 and the global calculated index 515.

For a vector addition example this would be used as follow `c[index]=a[index]+b[index]`.

In short

- GPUs have a different execution model compared to CPUs, with a focus on parallelism and mathematical operations.
- GPUs consist of thousands of lightweight threads that can be executed simultaneously on GPU cores.
- Threads are organized into warps, and warps are grouped into blocks assigned to streaming multiprocessors (SMPs).
- GPUs achieve performance through high degrees of parallelism and efficient memory access.
- Shared memory can be used to cache data and improve memory access efficiency within a block.
- Synchronization and data sharing are limited to the block level, with some possible sharing at the warp level depending on the architecture.
- Over-subscribing the GPU and maximizing warp and block occupancy help hide latencies and improve performance.
- Warp-level operations and cooperative groups provide efficient communication and synchronization within a warp or block.
- Thread indexing allows associating threads with specific elements in an array for parallel processing.

Terminology

At the moment there are three major GPU producers: NVIDIA, Intel, and AMD. While the basic concept behind GPUs is pretty similar they use different names for the various parts. Furthermore there are software environments for GPU programming, some from the producers and some from external groups all having different naming as well. Below there is a short compilation of the some terms used across different platforms and software environments.

Software mapping naming

CUDA	HIP	OpenCL	SYCL
grid of threads		NDRange	
block		work-group	
warp	wavefront	sub-group	
thread		work-item	
registers		private memory	
shared memory	local data share	local memory	
threadIdx.{x,y,z}		get_local_id({0,1,2})	nd_item::get_local({2,1,0}) [1]
blockIdx.{x,y,z}		get_group_id({0,1,2})	nd_item::get_group({2,1,0}) [1]
blockDim.{x,y,z}		get_local_size({0,1,2})	nd_item::get_local_range({2,1,0}) [1]

[1] (1,2,3) In SYCL, the thread indexing is inverted. In a 3D grid, physically adjacent threads have consecutive X (0) index in CUDA, HIP, and OpenCL, but consecutive Z (2) index in SYCL. In a 2D grid, CUDA, HIP, and OpenCL still has contiguous indexing along X (0) dimension, while in SYCL it is Y (1). Same applies to block dimensions and indexing.

Exercises

✍ What are threads in the context of shared memory architectures?

a. Independent execution units with their own memory address spaces
b. Light execution units with shared memory address spaces
c. Communication devices between separate memory units
d. Programming models for distributed memory machines

✔ Solution

Correct answer: *b) Light execution units with shared memory address spaces*

✍ What is data parallelism?

a. Distributing data across computational units that run in parallel, applying the same or similar operations to different data elements.
b. Distributing tasks across computational units that run in parallel, applying different operations to the same data elements.
c. Distributing data across computational units that run sequentially, applying the same operation to all data elements.
d. Distributing tasks across computational units that run sequentially, applying different operations to different data elements.

✔ Solution

Correct answer: *a) Distributing data across computational units that run in parallel, applying the same or similar operations to different data elements.*

✍ What type of parallelism is natural for GPU?

a. Task Parallelism
b. Data Parallelism
c. Both data and task parallelism
d. Neither data nor task parallelism

✔ Solution

Correct answer: *b) Data Parallelism*

✍ What is a kernel in the context of GPU execution?

a. A specific section of the CPU used for memory operations.
b. A specific section of the GPU used for memory operations.
c. A type of thread that operates on the GPU.
d. A function that is executed simultaneously by tens of thousands of threads on GPU cores.

✔ Solution

Correct answer: *d) A function that is executed simultaneously by tens of thousands of threads on GPU cores.*

✍ What is coalesced memory access?

a. It's when CUDA threads in the same warp access elements of the data which are adjacent in the memory.
b. It's when CUDA threads in different warps access elements of the data which are far in the memory.
c. It's when all threads have access to the global GPU memory.
d. It's when threads in a warp perform different operations.

✔ Solution

Correct answer: *a) It's when CUDA threads in the same warp access elements of the data which are adjacent in the memory.*

✍ What is the function of shared memory in the context of GPU execution?

a. It's used to store global memory.
b. It's used to store all the threads in a block.
c. It can be used to "cache" data that is used by more than one thread, avoiding multiple reads from the global memory.
d. It's used to store all the CUDA cores.

✔ Solution

Correct answer: *c) It can be used to "cache" data that is used by more than one thread, avoiding multiple reads from the global memory.*

✍ What is the significance of over-subscribing the GPU?

a. It reduces the overall performance of the GPU.
b. It ensures that there are more blocks than SMPs present on the device, helping to hide latencies and ensure high occupancy of the GPU.
c. It leads to a memory overflow in the GPU.
d. It ensures that there are more SMPs than blocks present on the device.

✔ Solution

Correct answer: *b) It ensures that there are more blocks than SMPs present on the device, helping to hide latencies and ensure high occupancy of the GPU.*

❗ Keypoints

- Parallel computing can be classified into distributed-memory and shared-memory architectures
- Two types of parallelism that can be explored are data parallelism and task parallelism.
- GPUs are a type of shared memory architecture suitable for data parallelism.
- GPUs have high parallelism, with threads organized into warps and blocks and.
- GPU optimization involves coalesced memory access, shared memory usage, and high thread and warp occupancy. Additionally, architecture-specific features such as warp-level operations and cooperative groups can be leveraged for more efficient processing.

Introduction to GPU programming models

❓ Questions

- What are the key differences between different GPU programming approaches?
- How should I choose which framework to use for my project?

Objectives

- Understand the basic ideas in different GPU programming frameworks
- Perform a quick cost-benefit analysis in the context of own code projects

Instructor note

- 20 min teaching
- 10 min discussion

There are different ways to use GPUs for computations. In the best case, when the code has already been written, one only needs to set the parameters and initial configuration in order to get started. In some other cases the problem is posed in such a way that a third-party library can be used to solve the most intensive part of the code (for example, this is increasingly the case with machine-learning workflows in Python). However, these cases are stil quite limited; in general, some additional programming might be needed. There are many GPU programming software environments and APIs available, which can be broadly grouped into **directive-based models**, **non-portable kernel-based models**, and **portable kernel-based models**, as well as high-level frameworks and libraries (including attempts at language-level support).

Standard C++/Fortran

Programs written in standard C++ and Fortran languages can now take advantage of NVIDIA GPUs without depending on any external library. This is possible thanks to the NVIDIA SDK suite of compilers that translates and optimizes the code for running on GPUs.

- Here is the series of articles on acceleration with standard language parallelism.
- Guidelines for writing C++ code can be found here,
- while those for Fortran code can be found here.

The performance of these two approaches is promising, as can be seen in the examples provided in those guidelines.

Directive-based programming

A fast and cheap way is to use **directive based** approaches. In this case the existing *serial* code is annotated with *hints* which indicate to the compiler which loops and regions should be executed on the GPU. In the absence of the API the directives are treated as comments and the code will just be executed as a usual serial code. This approach is focused on productivity and easy usage (but to the possible detriment of performance), and allows employing accelerators with minimal programming effort by adding parallelism to existing code without the need to write accelerator-specific code. There are two common ways to program using directives, namely **OpenACC** and **OpenMP**.

OpenACC

OpenACC is developed by a consortium formed in 2010 with the goal of developing a standard, portable, and scalable programming model for accelerators, including GPUs. Members of the OpenACC consortium include GPU vendors, such as NVIDIA and AMD, as well as leading supercomputing centers, universities, and software companies. Until recently it was supporting only NVIDIA GPUs, but now there is effort to support more devices and architectures.

OpenMP

OpenMP started as a multi-platform, shared-memory parallel programming API for multi-core CPUs and relatively recently has added support for GPU offloading. OpenMP aims to support various types of GPUs, which is done through the parent compiler.

The directive based approaches work with C/C++ and FORTRAN codes, while some third party extensions are available for other languages.

Non-portable kernel-based models (native programming models)

When doing direct GPU programming the developer has a large level of control by writing low-level code that directly communicates with the GPU and its hardware. Theoretically direct GPU programming methods provide the ability to write low-level, GPU-accelerated code that can provide significant performance improvements over CPU-only code. However, they also require a deeper understanding of the GPU architecture and its capabilities, as well as the specific programming method being used.

CUDA

CUDA is a parallel computing platform and API developed by NVIDIA. It is historically the first mainstream GPU programming framework. It allows developers to write C-like code that is executed on the GPU. CUDA provides a set of libraries and tools for low-level GPU programming and provides a performance boost for demanding computationally-intensive applications. While there is an extensive ecosystem, CUDA is restricted to NVIDIA hardware.

HIP

HIP (Heterogeneous Interface for Portability) is an API developed by AMD that provides a low-level interface for GPU programming. HIP is designed to provide a single source code that can be used on both NVIDIA and AMD GPUs. It is based on the CUDA programming model and provides an almost identical programming interface to CUDA.

Multiple examples of CUDA/HIP code are available in the content/examples/cuda-hip directory of this repository.

Portable kernel-based models (cross-platform portability ecosystems)

Cross-platform portability ecosystems typically provide a higher-level abstraction layer which enables a convenient and portable programming model for GPU programming. They can help reduce the time and effort required to maintain and deploy GPU-accelerated applications. The goal of these ecosystems is to achieve performance portability with a single-source application. In C++, the most notable cross-platform portability ecosystems are SYCL, OpenCL (C and C++ APIs), and Kokkos; others include alpaka and RAJA.

OpenCL

OpenCL (Open Computing Language) is a cross-platform, open-standard API for general-purpose parallel computing on CPUs, GPUs and FPGAs. It supports a wide range of hardware from multiple vendors. OpenCL provides a low-level programming interface for GPU programming and enables developers to write programs that can be executed on a variety of platforms. Unlike programming models such as CUDA, HIP, Kokkos, and SYCL, OpenCL uses a separate-source model. Recent versions of the OpenCL standard added C++ support for both API and the kernel code, but the C-based interface is still more widely used. The OpenCL Working Group doesn't provide any frameworks of its own. Instead, vendors who produce OpenCL-compliant devices release frameworks as part of their software development kits (SDKs). The two most popular OpenCL SDKs are released by NVIDIA and AMD. In both cases, the development kits are free and contain the libraries and tools that make it possible to build OpenCL applications.

Kokkos

Kokkos is an open-source performance portable programming model for heterogeneous parallel computing that has been mainly developed at Sandia National Laboratories. It is a C++-based ecosystem that provides a programming model for developing efficient and scalable parallel applications that run on many-core architectures such as CPUs, GPUs, and FPGAs. The Kokkos ecosystem consists of several components, such as the Kokkos core library, which provides parallel execution and memory abstraction, the Kokkos kernel library, which provides math kernels for linear algebra and graph algorithms, and the Kokkos tools library, which provides profiling and debugging tools. Kokkos components integrate well with other software libraries and technologies, such as MPI and OpenMP. Furthermore, the project collaborates with other projects, in order to provide interoperability and standardization for portable C++ programming.

alpaka

alpaka (Abstraction Library for Parallel Kernel Acceleration) is an open-source C++ header-only library that aims to provide performance portability across heterogeneous accelerator architectures by abstracting the underlying levels of parallelism. The library is platform-

independent and supports the concurrent and cooperative use of multiple devices, including host CPUs (x86, ARM, RISC-V) and GPUs from different vendors (NVIDIA, AMD, and Intel).

A key advantage of alpaka is that it requires only a single implementation of a user kernel, expressed as a function object with a standardized interface. This eliminates the need to write specialized code for different backends. The library provides a variety of accelerator backends, including CUDA, HIP, SYCL, OpenMP, and serial execution, that can be selected based on the target device. Moreover, multiple accelerator backends can even be combined to target different vendor hardware within a single application.

SYCL

SYCL is a royalty-free, open-standard C++ programming model for multi-device programming. It provides a high-level, single-source programming model for heterogeneous systems, including GPUs. Originally SYCL was developed on top of OpenCL; however, it is no more limited to just that. It can be implemented on top of other low-level heterogeneous computing APIs, such as CUDA or HIP, enabling developers to write programs that can be executed on a variety of platforms. Note that while SYCL is relatively high-level model, the developers are still required to write GPU kernels explicitly.

While alpaka, Kokkos, and RAJA refer to specific projects, SYCL itself is only a standard, for which several implementations exist. For GPU programming, Intel oneAPI DPC++ (supporting Intel GPUs natively, and NVIDIA and AMD GPUs with Codeplay oneAPI plugins) and AdaptiveCpp (previously also known as hipSYCL or Open SYCL, supporting NVIDIA and AMD GPUs, with experimental Intel GPU support available in combination with Intel oneAPI DPC++) are the most widely used. Other implementations of note are triSYCL and ComputeCPP.

High-level language support

Python

Python offers support for GPU programming through multiple abstraction levels.

CUDA Python, HIP Python and PyCUDA

These projects are, respectively, NVIDIA-, AMD- and community-supported wrappers providing Python bindings to the low-level CUDA and HIP APIs. To use these approaches directly, in most cases knowledge of CUDA or HIP programming is needed.

CUDA Python also aims to support higher-level toolkits and libraries, such as **CuPy** and **Numba**.

CuPy

CuPy is a GPU-based data array library compatible with NumPy/SciPy. It offers a highly similar interface to NumPy and SciPy, making it easy for developers to transition to GPU computing. Code written with NumPy can often be adapted to use CuPy with minimal modifications; in most straightforward cases, one might simply replace 'numpy' and 'scipy' with 'cupy' and 'cupyx.scipy' in their Python code.

Numba

Numba is an open-source JIT compiler that translates a subset of Python and NumPy code into optimized machine code. Numba supports CUDA-capable GPUs and is able to generate code for them using several different syntax variants. In 2021, upstream support for AMD (ROCm) support was discontinued. However, as of 2025, AMD has added downstream support for the Numba API through the Numba HIP package.

Julia

Julia has first-class support for GPU programming through the following packages that target GPUs from all three major vendors:

- CUDA.jl for NVIDIA GPUs
- AMDGPU.jl for AMD GPUs
- oneAPI.jl for Intel GPUs
- Metal.jl for Apple M-series GPUs

`CUDA.jl` is the most mature, `AMDGPU.jl` is somewhat behind but still ready for general use, while `oneAPI.jl` and `Metal.jl` are functional but might contain bugs, miss some features and provide suboptimal performance. Their respective APIs are however completely analogous and translation between libraries is straightforward.

All packages offer both high-level abstractions that require very little programming effort and a lower level approach for writing kernels for fine-grained control.

In short

- **Directive-based programming:**
  - Existing serial code is annotated with directives to indicate which parts should be executed on the GPU.
  - OpenACC and OpenMP are common directive-based programming models.
  - Productivity and easy usage are prioritized over performance.
  - Minimum programming effort is required to add parallelism to existing code.
- **Non-portable kernel-based models:**
  - Low-level code is written to directly communicate with the GPU.
  - CUDA is NVIDIA's parallel computing platform and API for GPU programming.
  - HIP is an API developed by AMD that provides a similar programming interface to CUDA for both NVIDIA and AMD GPUs.
  - Deeper understanding of GPU architecture and programming methods is needed.

- **Portable kernel-based models:**
  - Higher-level abstractions for GPU programming that provide portability.
  - Examples include OpenCL, Kokkos, alpaka, RAJA, and SYCL.
  - Aim to achieve performance portability with a single-source application.
  - Can run on various GPUs and platforms, reducing the effort required to maintain and deploy GPU-accelerated applications.
- **High-level language support:**
  - C++ and Fortran feature initiatives to support GPUs through language-standard parallelism.
  - Python libraries like PyCUDA, CuPy, and Numba offer GPU programming capabilities.
  - Julia has packages such as CUDA.jl, AMDGPU.jl, oneAPI.jl, and Metal.jl for GPU programming.
  - These approaches provide high-level abstraction and interfaces for GPU programming in the respective languages.

Summary

Each of these GPU programming environments has its own strengths and weaknesses, and the best choice for a given project will depend on a range of factors, including:

- the hardware platforms being targeted,
- the type of computation being performed, and
- the developer's experience and preferences.

High-level and productivity-focused APIs provide a simplified programming model and maximize code portability, while **low-level and performance-focused APIs** provide a high level of control over the GPU's hardware but also require more coding effort and expertise.

Exercises

✍ Discussion

- Which GPU programming frameworks have you already used previously, if any?
- What did you find most challenging? What was most useful?

Let us know in the main room or via comments in HackMD document.

Keypoints

- GPU programming approaches can be split into 1) directive-based, 2) non-portable kernel-based, 3) portable kernel-based, and 4) high-level language support.
- There are multiple frameworks/languages available for each approach, each with pros and cons.

Directive-based models

❓ Questions

- What is OpenACC and OpenMP offloading
- How to write GPU code using directives

❗ Objectives

- Understand the process of offloading
- Understand the differences between OpenACC and OpenMP offloading
- Understand the various levels of parallelism on a GPU
- Understand what is data movement

Instructor note

- 40 min teaching
- 40 min exercises

The most common directive-based models for GPU parallel programming are OpenMP offloading and OpenACC. The parallelization is done by introducing directives in places which are targeted for parallelization.

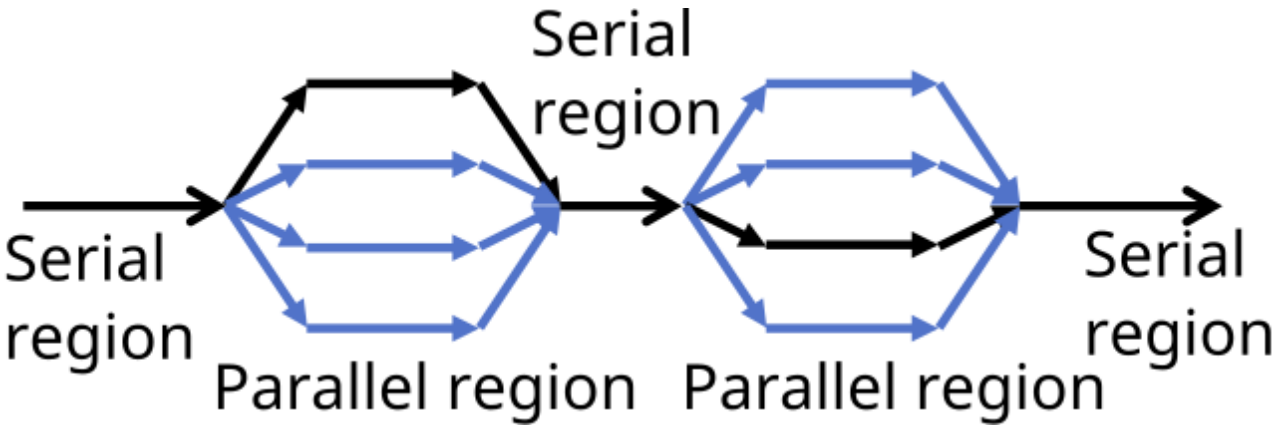
- **OpenACC** is known to be more **descriptive**, which means the programmer uses directives to tell the compiler how/where to parallelize the code and to move the data.
- **OpenMP offloading**, on the other hand, is known to be more **prescriptive**, where the programmer uses directives to tell the compiler more explicitly how/where to parallelize the code, instead of letting the compiler decide.

In OpenMP/OpenACC the compiler directives are specified by using **#pragma** in C/C++ or as special comments identified by unique sentinels in Fortran. Compilers can ignore the directives if the support for OpenMP/OpenACC is not enabled.

The compiler directives are used for various purposes: for thread creation, workload distribution (work sharing), data-environment management, serializing sections of code or for synchronization of work among the threads.

Execution model

OpenMP and OpenACC use the fork-join model of parallel execution. The program begins as a single thread of execution, the **master** thread. Everything is executed sequentially until the first parallel region construct is encountered.



When a parallel region is encountered, the master thread creates a group of threads, becomes the master of this group of threads, and is assigned the thread index 0 within the group. There is an implicit barrier at the end of the parallel regions.

Offloading Directives

OpenACC

In OpenACC, one of the most commonly used directives is `kernels`, which defines a region to be transferred into a series of kernels to be executed in sequence on a GPU. Work sharing is defined automatically for the separate kernels, but tuning prospects are limited.

✍ Example: `kernels`

C/C++ | Fortran

```c
#include <stdio.h>
#include <openacc.h>

#define NX 102400

int main(void)
{
    double vecA[NX], vecB[NX], vecC[NX];
    int i;

    /* Initialization of the vectors */
    for (i = 0; i < NX; i++) {
        vecA[i] = 1.0;
        vecB[i] = 2.0;
    }

    #pragma acc kernels
    for (i = 0; i < NX; i++) {
        vecC[i] = vecA[i] + vecB[i];
    }

    return 0;
}
```

The other approach of OpenACC to define parallel regions is to use the `parallel` directive. Contrary to the `kernels` directive, the `parallel` directive is more explicit and requires more analysis by the programmer. Work sharing has to be defined manually using the `loop` directive, and refined tuning is possible to achieve. The above example can be re-written as the following:

✍ Example: `parallel loop`

C/C++ | Fortran

```c
#include <stdio.h>
#include <openacc.h>

#define NX 102400

int main(void)
{
    double vecA[NX], vecB[NX], vecC[NX];
    int i;

    /* Initialization of the vectors */
    for (i = 0; i < NX; i++) {
        vecA[i] = 1.0;
        vecB[i] = 2.0;
    }

    #pragma acc parallel loop
    for (i = 0; i < NX; i++) {
        vecC[i] = vecA[i] + vecB[i];
    }

    return 0;
}
```

Sometimes we can obtain a little more performance by guiding the compiler to make specific choices. OpenACC has four levels of parallelism for offloading execution:

- **gang** coarse grain: the iterations are distributed among the gangs
- **worker** fine grain: worker's threads are activated within gangs and iterations are shared among the threads
- **vector** each worker activates its threads working in SIMT fashion and the work is shared among the threads
- **seq** the iterations are executed sequentially

Note

By default, `gang`, `worker` and `vector` parallelism are automatically decided and applied by the compiler.

The programmer could add clauses like `num_gangs`, `num_workers` and `vector_length` within the parallel region to specify the number of gangs, workers and vector length.

The optimal numbers are highly dependent on the GPU architecture and the compiler implementation though.

There is no thread synchronization at `gang` level, which means there is a risk of race condition.

OpenMP Offloading

With OpenMP, the `target` directive is used for device offloading.

✍ Example: `target` construct

C/C++ | Fortran

```c
#include <stdio.h>

#define NX 102400

int main(void)
{
    double vecA[NX], vecB[NX], vecC[NX];
    int i;

    /* Initialization of the vectors */
    for (i = 0; i < NX; i++) {
        vecA[i] = 1.0;
        vecB[i] = 2.0;
    }

    #pragma omp target
    for (i = 0; i < NX; i++) {
        vecC[i] = vecA[i] + vecB[i];
    }

    return 0;
}
```

Compared to the OpenACC's `kernels` directive, the `target` directive will not parallelise the underlying loop at all. To achieve proper parallelisation, one needs to be more prescriptive and specify what one wants. OpenMP offloading offers multiple levels of parallelism as well:

- **teams** coarse grain: creates a league of teams and one master thread in each team, but no worksharing among the teams
- **distribute** distributes the iterations across the master threads in the teams, but no worksharing among the threads within one team
- **parallel do/for** fine grain: threads are activated within one team and worksharing among them
- **SIMD** like the `vector` directive in OpenACC

Note

The programmer can add clauses like `num_teams` and `thread_limit` to specify the number of teams and threads within a team.

Threads in a team can synchronize but no synchronization among the teams.

Since OpenMP 5.0, there is a new `loop` directive available, which has a functionality similar to the corresponding one in OpenACC.

Keypoints

*Mapping between OpenACC/OpenMP directives and GPU (**HPE implementation**)*

Nvidia	AMD	Fortran OpenACC/OpenMP	C/C++ OpenMP
Threadblock	Work group	gang/teams	teams
Warp	Wavefront	worker/simd	parallel for simd
Thread	Work item	vector/simd	parallel for simd

- Each compiler supports different levels of parallelism.
- The size of gang/team/worker/vector_length can be chosen arbitrarily by the user but there are limits defined by the implementation.
- The maximum thread/grid/block size can be found via `rocminfo` / `nvaccelinfo`.

✍ Exercise: Change the levels of parallelism

In this exercise we would like to change the levels of parallelism using clauses. First compile and run one of the examples to find out the default number of blocks and threads set by the compiler at runtime. To make a change, try adding clauses like `num_gangs`, `num_workers`, `vector_length` for OpenACC and `num_teams`, `thread_limit` for OpenMP offloading.

Remember to set the environment by executing `export CRAY_ACC_DEBUG=2` at runtime.

How to compile and run the code interactively:

C/C++ | Fortran

```
salloc -A project_465002387 -N 1 -t 1:00:00 -p standard-g --gpus-per-node=1

module load LUMI/24.03
module load partition/G
module load rocm/6.0.3

# OpenMP
cc -O2 -fopenmp -o ex1 ex1.c
# Only OpenACC Fortran is supported by HPE compiler.

export CRAY_ACC_DEBUG=2
srun ./ex1
```

Example of a trivially parallelizable vector addition problem:

OpenMP | OpenACC

C/C++ | Fortran

```
#include <stdio.h>
#include <math.h>
#define NX 102400

int main(void){
    double vecA[NX],vecB[NX],vecC[NX];

    /* Initialize vectors */
    for (int i = 0; i < NX; i++) {
        vecA[i] = 1.0;
        vecB[i] = 1.0;
    }

    #pragma omp target teams distribute parallel for simd
    {
    for (int i = 0; i < NX; i++) {
       vecC[i] = vecA[i] + vecB[i];
    }
    }
 }
```

Data Movement

Due to distinct memory spaces on host and device, transferring data becomes inevitable. New directives are needed to specify how variables are transferred from the host to the device data environment. Commonly transferred items consist of arrays (array sections), scalars, pointers, and structure elements. Various data clauses used for data movement are summarised in the following table:

`OpenMP`	`OpenACC`	
`map(to:list)`	`copyin(list)`	On entering the region, variables in the list are initialized on the device using the original values from the host
`map(from:list)`	`copyout(list)`	At the end of the target region, the values from variables in the list are copied into the original variables on the host. On entering the region, the initial value of the variables on the device is not initialized
`map(tofrom:list)`	`copy(list)`	The effect of both a map-to and a map-from
`map(alloc:list)`	`create(list)`	On entering the region, data is allocated and uninitialized on the device
`map(delete:list)`	`delete(list)`	Delete data on the device

Note

When mapping data arrays or pointers, be careful about the array section notation:

- In C/C++: array[lower-bound:length]. The notation :N is equivalent to 0:N.
- In Fortran:array[lower-bound:upper-bound]. The notation :N is equivalent to 1:N.

Data region

The specific data clause combined with the data directive constitutes the start of a data region. How the directives create storage, transfer data, and remove storage on the device are classified as two categories: structured data region and unstructured data region.

Structured Data Region

A structured data region is convenient for providing persistent data on the device which could be used for subsequent GPU directives.

✍ Syntax for structured data region

OpenMP | OpenACC

C/C++ | Fortran

```
#pragma omp target data [clauses]
{structured-block}
```

Unstructured Data Region

However it is inconvenient in real applications to use a structured data region. An unstructured data region gives more freedom in creating and deleting data on the device at any appropriate point.

✍ Syntax for unstructured data region

OpenMP | OpenACC

C/C++ | Fortran

```
#pragma omp target enter data [clauses]
```

```
#pragma omp target exit data
```

❗ Keypoints

Structured Data Region

- Start and end points within a single subroutine
- Memory exists within the data region

Unstructured Data Region

- Multiple start and end points across different subroutines

- Memory exists until explicitly deallocated

Update

Sometimes, variables need to be synchronized between the host and the device memory, e.g. in order to write out variables on the host for debugging or visualization, and it is often used in conjunction with unstructured data regions. To control the data transfer direction, a motion-clause must be present.

✍ Syntax for update directive

OpenMP | OpenACC

C/C++ | Fortran

```
#pragma omp target update [clauses]
```

```
motion-clause:
          to (list)
          from (list)
```

❗ Note

- `update` directive can only be used in host code since data movement must be initiated from the host, i.e. it may not appear inside of a compute region.
- in OpenACC, motion-clause “host” has been deprecated and renamed “self”

✍ Exercise: `update`

Try to figure out the variable values on host and device at each check point.

C/C++ | Fortran

```c
#include <stdio.h>
int main(void)
{
int x = 0;

#pragma omp target data map(tofrom:x)
{
   /* check point 1 */
  x = 10;
   /* check point 2 */
#pragma omp target update to(x)
   /* check point 3 */
}

return 0;
}
```

✔ Solution

check point	x on host	x on device
check point1	0	0
check point2	10	0
check point3	10	10

✍ Exercise: Adding data mapping clauses

Add proper data mapping clauses explicitly to the directives

OpenMP | OpenACC

C/C++ | Fortran

```c
#include <stdio.h>
#include <math.h>
#define NX 102400

int main(void){
    double vecA[NX],vecB[NX],vecC[NX];

    /* Initialize vectors */
    for (int i = 0; i < NX; i++) {
    vecA[i] = 1.0;
    vecB[i] = 1.0;
    }
    /* Adding mapping clauses here */
    #pragma omp target teams distribute parallel for simd
    {
    for (int i = 0; i < NX; i++) {
       vecC[i] = vecA[i] + vecB[i];
    }
    }

    double sum = 0.0;
    for (int i = 0; i < NX; i++) {
       sum += vecC[i];
    }
    printf("The sum is: %8.6f \n", sum);
}
```

✔ Solution

OpenMP | OpenACC

C/C++ | Fortran

```c
#include <stdio.h>
#include <math.h>
#define NX 102400

int main(void){
    double vecA[NX],vecB[NX],vecC[NX];

    /* Initialize vectors */
    for (int i = 0; i < NX; i++) {
        vecA[i] = 1.0;
        vecB[i] = 1.0;
    }

    #pragma omp target teams distribute parallel for simd
map(to:vecA[0:NX],vecB[0:NX]) map(from:vecC[0:NX])
    {
    for (int i = 0; i < NX; i++) {
         vecC[i] = vecA[i] + vecB[i];
    }
    }

    double sum = 0.0;
    for (int i = 0; i < NX; i++) {
       sum += vecC[i];
    }
    printf("The sum is: %8.6f \n", sum);
    }
```

Optimize Data Transfers

- Explicitly transfer the data as much as possible.
- Reduce the amount of data mapping between host and device, get rid of unnecessary data transfers.
- Try to keep the data environment residing on the device as long as possible.

Pros of directive-based frameworks

- Incremental programming
- Porting of existing software requires less work
- Same code can be compiled to CPU and GPU versions easily using compiler flag
- Low learning curve, do not need to know low-level hardware details
- Good portability

See also

- ENCCS lesson on OpenACC
- ENCCS lesson on OpenMP for GPU offloading

Keypoints

- OpenACC and OpenMP-offloading enables you to annotate your code with special directives to identify areas to be executed in parallel on a GPU.
- Both allow to fine-tune the distribution of the work to match architecture characteristics.
- Both allow to control the flow of data to/from the GPU.
- The directive-based approaches save time compared to lower-level approaches, but you need to be mindful of data movement in particular to obtain good performance.

Non-portable kernel-based models

Questions

- How to program GPUs with CUDA and HIP?
- What optimizations are possible when programming with CUDA and HIP?

Objectives

- Be able to use CUDA and HIP to write basic codes
- Understand how the execution is done and how to do optimizations

Instructor note

- 45 min teaching
- 30 min exercises

Fundamentals of GPU programming with CUDA and HIP

Unlike some cross-platform portability ecosystems, such as alpaka, Kokkos, OpenCL, RAJA, and SYCL, which cater to multiple architectures, CUDA and HIP are solely focused on GPUs. They provide extensive libraries, APIs, and compiler toolchains that optimize code execution on NVIDIA GPUs (in the case of CUDA) and both NVIDIA and AMD GPUs (in the case of HIP). Because they are developed by the device producers, these programming models provide high-performance computing capabilities and offer advanced features like shared memory, thread synchronization, and memory management specific to GPU architectures.

CUDA, developed by NVIDIA, has gained significant popularity and is widely used for GPU programming. It offers a comprehensive ecosystem that includes not only the CUDA programming model but also a vast collection of GPU-accelerated libraries. Developers can write CUDA kernels using C++ and seamlessly integrate them into their applications to harness the massive parallelism of GPUs.

HIP, on the other hand, is an open-source project that aims to provide a more “portable” GPU programming interface. It allows developers to write GPU code in a syntax similar to CUDA and provides a translation layer that enables the same code to run on both NVIDIA and AMD GPUs. This approach minimizes the effort required to port CUDA code to different GPU architectures and provides flexibility for developers to target multiple platforms.

By being closely tied to the GPU hardware, CUDA and HIP provide a level of performance optimization that may not be achievable with cross-platform portability ecosystems. The libraries and toolchains offered by these programming models are specifically designed to exploit the capabilities of the underlying GPU architectures, enabling developers to achieve high performance.

Developers utilizing CUDA or HIP can tap into an extensive ecosystem of GPU-accelerated libraries, covering various domains, including linear algebra, signal processing, image processing, machine learning, and more. These libraries are highly optimized to take advantage of the parallelism and computational power offered by GPUs, allowing developers to accelerate their applications without having to implement complex algorithms from scratch.

As mentioned before, CUDA and HIP are very similar so it makes sense to cover both at the same time.

Comparison to portable kernel-based models

In code examples below, we will also show examples in the portable kernel-based frameworks Kokkos, SYCL and OpenCL, which will be covered in the next episode.

Hello World

Below we have the most basic example of CUDA and HIP, the “Hello World” program:

CUDA | HIP | Kokkos | OpenCL | SYCL

```cuda
#include <cuda.h>
#include <cuda_runtime.h>
#include <stdio.h>

int main(void) {
  int count, device;

  cudaGetDeviceCount(&count);
  cudaGetDevice(&device);

  printf("Hello! I'm GPU %d out of %d GPUs in total.\n", device, count);
  return 0;
}
```

In both versions, we include the necessary headers: **cuda_runtime.h** and **cuda.h** for CUDA, and **hip_runtime.h** for HIP. These headers provide the required functionality for GPU programming.

To retrieve information about the available devices, we use the functions **<cuda/hip>GetDeviceCount** and **<cuda/hip>GetDevice**. These functions allow us to determine the total number of GPUs and the index of the currently used device. In the code examples, we default to using device 0.

As an exercise, modify the "Hello World" code to explicitly use a specific GPU. Do this by using the **<cuda/hip>SetDevice** function, which allows to set the desired GPU device. Note that the device number provided has to be within the range of available devices, otherwise, the program may fail to run or produce unexpected results. To experiment with different GPUs, modify the code to include the following line before retrieving device information:

```
cudaSetDevice(deviceNumber); // For CUDA
hipSetDevice(deviceNumber); // For HIP
```

Replace **deviceNumber** with the desired GPU device index. Run the code with different device numbers to observe the output (more examples for the "Hello World" program are available in the content/examples/cuda-hip subdirectory of this lesson repository).

Vector Addition

To demonstrate the fundamental features of CUDA/HIP programming, let's begin with a straightforward task of element-wise vector addition. The code snippet below demonstrates how to utilize CUDA and HIP for efficiently executing this operation.

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <math.h>
#include <stdio.h>

__global__ void vector_add(float *A, float *B, float *C, int n) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}

int main(void) {
  const int N = 10000;
  float *Ah, *Bh, *Ch, *Cref;
  float *Ad, *Bd, *Cd;
  int i;

  // Allocate the arrays on CPU
  Ah = (float *)malloc(N * sizeof(float));
  Bh = (float *)malloc(N * sizeof(float));
  Ch = (float *)malloc(N * sizeof(float));
  Cref = (float *)malloc(N * sizeof(float));

  // initialise data and calculate reference values on CPU
  for (i = 0; i < N; i++) {
    Ah[i] = sin(i) * 2.3;
    Bh[i] = cos(i) * 1.1;
    Cref[i] = Ah[i] + Bh[i];
  }

  // Allocate the arrays on GPU
  cudaMalloc((void **)&Ad, N * sizeof(float));
  cudaMalloc((void **)&Bd, N * sizeof(float));
  cudaMalloc((void **)&Cd, N * sizeof(float));

  // Transfer the data from CPU to GPU
  cudaMemcpy(Ad, Ah, sizeof(float) * N, cudaMemcpyHostToDevice);
  cudaMemcpy(Bd, Bh, sizeof(float) * N, cudaMemcpyHostToDevice);

  // define grid dimensions + launch the device kernel
  dim3 blocks, threads;
  threads = dim3(256, 1, 1);
  blocks = dim3((N + 256 - 1) / 256, 1, 1);

  // Launch Kernel
  vector_add<<<blocks, threads>>>(Ad, Bd, Cd, N);

  // copy results back to CPU
  cudaMemcpy(Ch, Cd, sizeof(float) * N, cudaMemcpyDeviceToHost);

  printf("reference: %f %f %f %f ... %f %f\n", Cref[0], Cref[1], Cref[2],
         Cref[3], Cref[N - 2], Cref[N - 1]);
  printf("   result: %f %f %f %f ... %f %f\n", Ch[0], Ch[1], Ch[2], Ch[3],
         Ch[N - 2], Ch[N - 1]);

  // confirm that results are correct
  float error = 0.0;
  float tolerance = 1e-6;
  float diff;
  for (i = 0; i < N; i++) {
    diff = fabs(Cref[i] - Ch[i]);
```

```
    if (diff > tolerance) {
      error += diff;
    }
  }
  printf("total error: %f\n", error);
  printf("  reference: %f at (42)\n", Cref[42]);
  printf("     result: %f at (42)\n", Ch[42]);

  // Free the GPU arrays
  cudaFree(Ad);
  cudaFree(Bd);
  cudaFree(Cd);

  // Free the CPU arrays
  free(Ah);
  free(Bh);
  free(Ch);
  free(Cref);

  return 0;
}
```

In this case, the CUDA and HIP codes are equivalent one to one so we will only refer to the CUDA version. The CUDA and HIP programming model are host centric programming models. The main program is executed on CPU and controls all the operations, memory allocations, data transfers between CPU and GPU, and launches the kernels to be executed on the GPU. The code starts with defining the GPU kernel function called **vector_add** with attribute **__global__**. It takes three input arrays *A*, *B*, and *C* along with the array size *n*. The kernel function contains the actually code which is executed on the GPU by multiple threads in parallel.

Accelerators in general and GPUs in particular usually have their own dedicated memory separate from the system memory (AMD MI300A is one exception, using the same memory for both CPU and GPU). When programming for GPUs, there are two sets of pointers involved and it's necessary to manage data movement between the host memory and the accelerator memory. Data needs to be explicitly copied from the host memory to the accelerator memory before it can be processed by the accelerator. Similarly, results or modified data may need to be copied back from the accelerator memory to the host memory to make them accessible to the CPU.

The main function of the code initializes the input arrays *Ah*, *Bh* on the CPU and computes the reference array *Cref*. It then allocates memory on the GPU for the input and output arrays *Ad*, *Bd*, and *Cd* using **cudaMalloc**. Herein, *h* is for the 'host' (CPU) and *d* for the 'device' (GPU). The data is transferred from the CPU to the GPU using hipMemcpy, and then the GPU kernel is launched using the <<<.>>> syntax. All kernels launch are asynchronous. After launch the control returns to the *main()* and the code proceeds to the next instructions.

After the kernel execution, the result array *Cd* is copied back to the CPU using **cudaMemcpy**. The code then prints the reference and result arrays, calculates the error by comparing the reference and result arrays. Finally, the GPU and CPU memory are deallocated using **cudaFree** and **free** functions, respectively.

The host functions **cudaSetDevice**, **cudaMalloc**, **cudaMemcpy**, and **cudaFree** are blocking, i.e. the code does not continues to next instructions until the operations are completed. However this is not the default behaviour, for many operations there are asynchronous equivalents and there are as well many library calls return the control to the *main()* after calling. This allows the developers to launch independent operations and overlap them.

In short, this code demonstrates how to utilize the CUDA and HIP to perform vector addition on a GPU, showcasing the steps involved in allocating memory, transferring data between the CPU and GPU, launching a kernel function, and handling the results. It serves as a starting point for GPU-accelerated computations using CUDA and HIP. More examples for the vector (array) addition program are available at content/examples.

In order to practice the concepts shown above, edit the skeleton code in the repository and the code corresponding to setting the device, memory allocations and transfers, and the kernel execution.

Vector Addition with Unified Memory

For a while already GPUs support unified memory, which allows to use the same pointer for both CPU and GPU data. This simplifies developing codes by removing the explicit data transfers. The data resides on CPU until it is needed on GPU or vice-versa. However the data transfers still happens “under the hood” and the developer needs to construct the code to avoid unnecessary transfers. Below one can see the modified vector addition codes:

CUDA | HIP | SYCL

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <math.h>
#include <stdio.h>

__global__ void vector_add(float *A, float *B, float *C, int n) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}

int main(void) {
  const int N = 10000;
  float *Ah, *Bh, *Ch, *Cref;
  int i;

  // Allocate the arrays using Unified Memory
  cudaMallocManaged(&Ah, N * sizeof(float));
  cudaMallocManaged(&Bh, N * sizeof(float));
  cudaMallocManaged(&Ch, N * sizeof(float));
  cudaMallocManaged(&Cref, N * sizeof(float));

  // initialise data and calculate reference values on CPU
  for (i = 0; i < N; i++) {
    Ah[i] = sin(i) * 2.3;
    Bh[i] = cos(i) * 1.1;
    Cref[i] = Ah[i] + Bh[i];
  }

  // define grid dimensions
  dim3 blocks, threads;
  threads = dim3(256, 1, 1);
  blocks = dim3((N + 256 - 1) / 256, 1, 1);

  // Launch Kernel
  vector_add<<<blocks, threads>>>(Ah, Bh, Ch, N);
  cudaDeviceSynchronize(); // Wait for the kernel to complete

  // At this point we want to access the data on CPU
  printf("reference: %f %f %f %f ... %f %f\n", Cref[0], Cref[1], Cref[2],
         Cref[3], Cref[N - 2], Cref[N - 1]);
  printf("   result: %f %f %f %f ... %f %f\n", Ch[0], Ch[1], Ch[2], Ch[3],
         Ch[N - 2], Ch[N - 1]);

  // confirm that results are correct
  float error = 0.0;
  float tolerance = 1e-6;
  float diff;
  for (i = 0; i < N; i++) {
    diff = fabs(Cref[i] - Ch[i]);
    if (diff > tolerance) {
      error += diff;
    }
  }
  printf("total error: %f\n", error);
  printf("  reference: %f at (42)\n", Cref[42]);
  printf("     result: %f at (42)\n", Ch[42]);

  // Free the GPU arrays
  cudaFree(Ah);
  cudaFree(Bh);
```

```
    cudaFree(Ch);
    cudaFree(Cref);

    return 0;
}
```

Now the arrays *Ah*, *Bh*, *Ch*, and *Cref* are using *cudaMallocManaged* to allocate Unified Memory. The **vector_add kernel** is launched by passing these Unified Memory pointers directly. After the kernel launch, **cudaDeviceSynchronize** is used to wait for the kernel to complete execution. Finally, **cudaFree** is used to free the Unified Memory arrays. The Unified Memory allows for transparent data migration between CPU and GPU, eliminating the need for explicit data transfers.

As an exercise modify the skeleton code for vector addition to use Unified Memory.

Basics - In short

- CUDA is developed by NVIDIA, while HIP is an open-source project (from AMD) for multi-platform GPU programming.
- CUDA and HIP are GPU-focused programming models for optimized code execution on NVIDIA and AMD GPUs.
- CUDA and HIP are similar, allowing developers to write GPU code in a syntax similar to CUDA and target multiple platforms.
- CUDA and HIP are programming models focused solely on GPUs
- CUDA and HIP offer high-performance computing capabilities and advanced features specific to GPU architectures, such as shared memory and memory management.
- They provide highly GPU-accelerated libraries in various domains like linear algebra, signal processing, image processing, and machine learning.
- Programming for GPUs involves managing data movement between host and accelerator memory.
- Unified Memory simplifies data transfers by using the same pointer for CPU and GPU data, but code optimization is still necessary.

Memory Optimizations

Vector addition is a relatively simple, straight forward case. Each thread reads data from memory, does an addition and then saves the result. Two adjacent threads access memory location in memory close to each other. Also the data is used only once. In practice this not the case. Also sometimes the same data is used several times resulting in additional memory accesses.

Memory optimization is one of the most important type of optimization done to efficiently use the GPUs. Before looking how it is done in practice let's revisit some basic concepts about GPUs and execution model.

GPUs are comprised many light cores, the so-called Streaming Processors (SP) in CUDA, which are physically group together in units, i.e. Streaming Multi-Processors (SMP) in CUDA architecture (note that in AMD the equivalent is called Computing Units, while in Intel GPUs they are Execution Units). The work is done on GPUs by launching many threads each executing an instance of the same kernel. The order of execution is not defined, and the threads can only exchange information in specific conditions. Because of the way the SPs are grouped the threads are also grouped in **blocks**. Each **block** is assigned to an SMP, and can not be split. An SMP can have more than block residing at a moment, however there is no communications between the threads in different blocks. In addition to the SPs, each SMP contains very fast memory which in CUDA is referred to as *shared memory*. The threads in a block can read and write to the shared memory and use it as a user controlled cache. One thread can for example write to a location in the shared memory while another thread in the same block can read and use that data. In order to be sure that all threads in the block completed writing **__syncthreads()** function has to be used to make the threads in the block wait until all of them reached the specific place in the kernel. Another important aspect in the GPU programming model is that the threads in the block are not executed independently. The threads in a block are physically grouped in warps of size 32 in NVIDIA devices or wavefronts of size 32 or 64 in AMD devices (depending on device architecture). Intel devices are notable in that the warp size, called SIMD width, is highly configurable, with typical possible values of 8, 16, or 32 (depends on the hardware). All memory accesses of the global GPU memory are done per warp. When data is needed for some calculations a warp loads from the GPU memory blocks of specific size (64 or 128 Bytes). These operation is very expensive, it has a latency of hundreds of cycles. This means that the threads in a warp should work with elements of the data located close in the memory. In the vector addition two threads near each other, of index tid and tid+1, access elements adjacent in the GPU memory.

The shared memory can be used to improve performance in two ways. It is possible to avoid extra reads from the memory when several threads in the same block need the same data (see stencil code) or it can be used to improve the memory access patterns like in the case of matrix transpose.

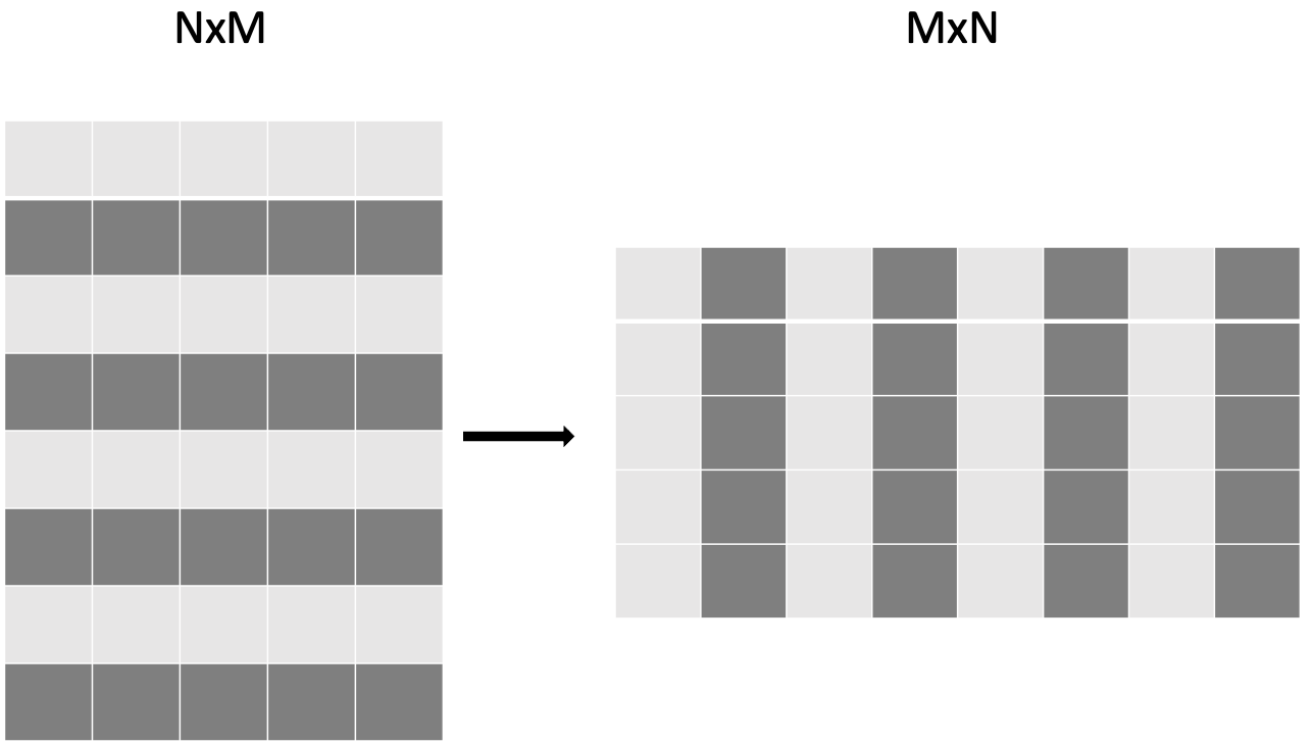
Memory, Execution - In short

- GPUs consist of streaming processors (SPs) grouped together in units, such as Streaming Multi-Processors (SMPs) in CUDA architecture.
- Work on GPUs is done by launching threads, with each thread executing an instance of the same kernel, and the execution order is not defined.
- Threads are organized into blocks, assigned to an SMP, and cannot be split, and there is no communication between threads in different blocks.
- Each SMP contains shared memory, which acts as a user-controlled cache for threads within a block, allowing efficient data sharing and synchronization.
- The shared memory can be used to avoid extra memory reads when multiple threads in the same block need the same data or to improve memory access patterns, such as in matrix transpose operations.

- Memory accesses from global GPU memory are performed per warp (groups of threads), and loading data from GPU memory has high latency.
- To optimize memory access, threads within a warp should work with adjacent elements in memory to reduce latency.
- Proper utilization of shared memory can improve performance by reducing memory reads and enhancing memory access patterns.

Matrix Transpose

Matrix transpose is a classic example where shared memory can significantly improve the performance. The use of shared memory reduces global memory accesses and exploits the high bandwidth and low latency of shared memory.



First as a reference we use a simple kernel which copy the data from one array to the other.

CUDA | HIP | SYCL

```
#include <cstdlib>
#include <cuda.h>
#include <cuda_runtime.h>
#include <math.h>
#include <stdio.h>
#include <vector>

const static int width = 4096;
const static int height = 4096;
const static int tile_dim = 16;

__global__ void copy_kernel(float *in, float *out, int width, int height) {
  int x_index = blockIdx.x * tile_dim + threadIdx.x;
  int y_index = blockIdx.y * tile_dim + threadIdx.y;

  int index = y_index * width + x_index;

  out[index] = in[index];
}

int main() {
  std::vector<float> matrix_in;
  std::vector<float> matrix_out;

  matrix_in.resize(width * height);
  matrix_out.resize(width * height);

  for (int i = 0; i < width * height; i++) {
    matrix_in[i] = (float)rand() / (float)RAND_MAX;
  }

  float *d_in, *d_out;

  cudaMalloc((void **)&d_in, width * height * sizeof(float));
  cudaMalloc((void **)&d_out, width * height * sizeof(float));

  cudaMemcpy(d_in, matrix_in.data(), width * height * sizeof(float),
             cudaMemcpyHostToDevice);

  printf("Setup complete. Launching kernel \n");
  int block_x = width / tile_dim;
  int block_y = height / tile_dim;

  // Create events
  cudaEvent_t start_kernel_event;
  cudaEventCreate(&start_kernel_event);
  cudaEvent_t end_kernel_event;
  cudaEventCreate(&end_kernel_event);

  printf("Warm up the gpu!\n");
  for (int i = 1; i <= 10; i++) {
    copy_kernel<<<dim3(block_x, block_y), dim3(tile_dim, tile_dim)>>>(
        d_in, d_out, width, height);
  }

  cudaEventRecord(start_kernel_event, 0);

  for (int i = 1; i <= 10; i++) {
    copy_kernel<<<dim3(block_x, block_y), dim3(tile_dim, tile_dim)>>>(
        d_in, d_out, width, height);
  }
```

```
  cudaEventRecord(end_kernel_event, 0);
  cudaEventSynchronize(end_kernel_event);

  cudaDeviceSynchronize();
  float time_kernel;
  cudaEventElapsedTime(&time_kernel, start_kernel_event, end_kernel_event);

  printf("Kernel execution complete \n");
  printf("Event timings:\n");
  printf("  %.6f ms - copy \n  Bandwidth %.6f GB/s\n", time_kernel / 10,
         2.0 * 10000 * (((double)(width) * (double)height) * sizeof(float)) /
             (time_kernel * 1024 * 1024 * 1024));

  cudaMemcpy(matrix_out.data(), d_out, width * height * sizeof(float),
             cudaMemcpyDeviceToHost);

  return 0;
}
```

We note that this code does not do any calculations. Each thread reads one element and then writes it to another locations. By measuring the execution time of the kernel we can compute the effective bandwidth achieve by this kernel. We can measure the time using **rocprof** or **cuda/hip events**. On a NVIDIA V100 GPU this code achieves *717 GB/s* out of the theoretical peak *900 GB/s*.

Now we do the first iteration of the code, a naive transpose. The reads have a nice *coalesced* access pattern, but the writing is now very inefficient.

CUDA/HIP | SYCL

```
__global__ void transpose_naive_kernel(float *in, float *out, int width,
                                       int height) {
  int x_index = blockIdx.x * tile_dim + threadIdx.x;
  int y_index = blockIdx.y * tile_dim + threadIdx.y;

  int in_index = y_index * width + x_index;
  int out_index = x_index * height + y_index;

  out[out_index] = in[in_index];
}
```

Checking the index *in_index* we see that two adjacent threads (*threadIx.x*, *threadIdx.x+1*) access location in memory near each other. However the writes are not. Threads access data which in a strided way. Two adjacent threads access data separated by *height* elements. This practically results in 32 memory operations, however due to under the hood optimizations the achieved bandwidth is *311 GB/s*.

We can improve the code by reading the data in a *coalesced* way, save it in the shared memory row by row and then write in the global memory column by column.

CUDA/HIP | SYCL

```
__global__ void transpose_SM_kernel(float *in, float *out, int width,
                                    int height) {
  __shared__ float tile[tile_dim][tile_dim];

  int x_tile_index = blockIdx.x * tile_dim;
  int y_tile_index = blockIdx.y * tile_dim;

  int in_index =
      (y_tile_index + threadIdx.y) * width + (x_tile_index + threadIdx.x);
  int out_index =
      (x_tile_index + threadIdx.y) * height + (y_tile_index + threadIdx.x);

  tile[threadIdx.y][threadIdx.x] = in[in_index];

  __syncthreads();

  out[out_index] = tile[threadIdx.x][threadIdx.y];
}
```

We define a **tile_dim** constant to determine the size of the shared memory tile. The matrix transpose kernel uses a 2D grid of thread blocks, where each thread block operates on a *tile_dim x tile_dim* tile of the input matrix.

The kernel first loads data from the global memory into the shared memory tile. Each thread loads a single element from the input matrix into the shared memory tile. Then, a **__syncthreads()** barrier ensures that all threads have finished loading data into shared memory before proceeding.

Next, the kernel writes the transposed data from the shared memory tile back to the output matrix in global memory. Each thread writes a single element from the shared memory tile to the output matrix. By using shared memory, this optimized implementation reduces global memory accesses and exploits memory coalescence, resulting in improved performance compared to a naive transpose implementation.

This kernel achieved on NVIDIA V100 *674 GB/s*.

This is pretty close to the bandwidth achieved by the simple copy kernel, but there is one more thing to improve.

Shared memory is composed of *banks*. Each banks can service only one request at the time. Bank conflicts happen when more than 1 thread in a specific warp try to access data in bank. The bank conflicts are resolved by serializing the accesses resulting in less performance. In the above example when data is saved to the shared memory, each thread in the warp will save an element of the data in a different one. Assuming that shared memory has 16 banks after writing each bank will contain one column. At the last step when we write from the shared memory to the global memory each warp load data from the same bank. A simple way to avoid this is by just padding the temporary array.

CUDA/HIP | SYCL

```
__global__ void transpose_SM_nobc_kernel(float *in, float *out, int width,
                                         int height) {
  __shared__ float tile[tile_dim][tile_dim + 1];

  int x_tile_index = blockIdx.x * tile_dim;
  int y_tile_index = blockIdx.y * tile_dim;

  int in_index =
      (y_tile_index + threadIdx.y) * width + (x_tile_index + threadIdx.x);
  int out_index =
      (x_tile_index + threadIdx.y) * height + (y_tile_index + threadIdx.x);

  tile[threadIdx.y][threadIdx.x] = in[in_index];

  __syncthreads();

  out[out_index] = tile[threadIdx.x][threadIdx.y];
}
```

By padding the array the data is slightly shifting it resulting in no bank conflicts. The effective bandwidth for this kernel is *697 GB/s*.

Using sharing memory as a cache - In short

- Shared memory can significantly improve performance in operations like matrix transpose.
- Shared memory reduces global memory accesses and exploits the high bandwidth and low latency of shared memory.
- An optimized implementation utilizes shared memory, loads data coalescedly, and performs transpose operations.
- The optimized implementation uses a 2D grid of thread blocks and a shared memory tile size determined by a constant.
- The kernel loads data from global memory into the shared memory tile and uses a synchronization barrier.

- To avoid bank conflicts in shared memory, padding the temporary array is a simple solution.

Reductions

Reductions refer to operations in which the elements of an array are aggregated in a single value through operations such as summing, finding the maximum or minimum, or performing logical operations.

In the serial approach, the reduction is performed sequentially by iterating through the collection of values and accumulating the result step by step. This will be enough for small sizes, but for big problems this results in significant time spent in this part of an application. On a GPU, this approach is not feasible. Using just one thread to do this operation means the rest of the GPU is wasted. Doing reduction in parallel is a little tricky. In order for a thread to do work, it needs to have some partial result to use. If we launch, for example, a kernel performing a simple vector summation, `sum[0]+=a[tid]`, with *N* threads we notice that this would result in undefined behaviour. GPUs have mechanisms to access the memory and lock the access for other threads while 1 thread is doing some operations to a given data via **atomics**, however this means that the memory access gets again to be serialized. There is not much gain. We note that when doing reductions the order of the iterations is not important (barring the typical non-associative behavior of floating-point operations). Also we can we might have to divide our problem in several subsets and do the reduction operation for each subset separately. On the GPUs, since the GPU threads are grouped in blocks, the size of the subset based on that. Inside the block, threads can cooperate with each other, they can share data via the shared memory and can be synchronized as well. All threads read the data to be reduced, but now we have significantly less partial results to deal with. In general, the size of the block ranges from 256 to 1024 threads. In case of very large problems, after this procedure if we are left too many partial results this step can be repeated.

At the block level we still have to perform a reduction in an efficient way. Doing it serially means that we are not using all GPU cores (roughly 97% of the computing capacity is wasted). Doing it naively parallel using **atomics**, but on the shared memory is also not a good option. Going back back to the fact the reduction operations are commutative and associative we can set each thread to “reduce” two elements of the local part of the array. Shared memory can be used to store the partial “reductions” as shown below in the code:

CUDA/HIP | SYCL

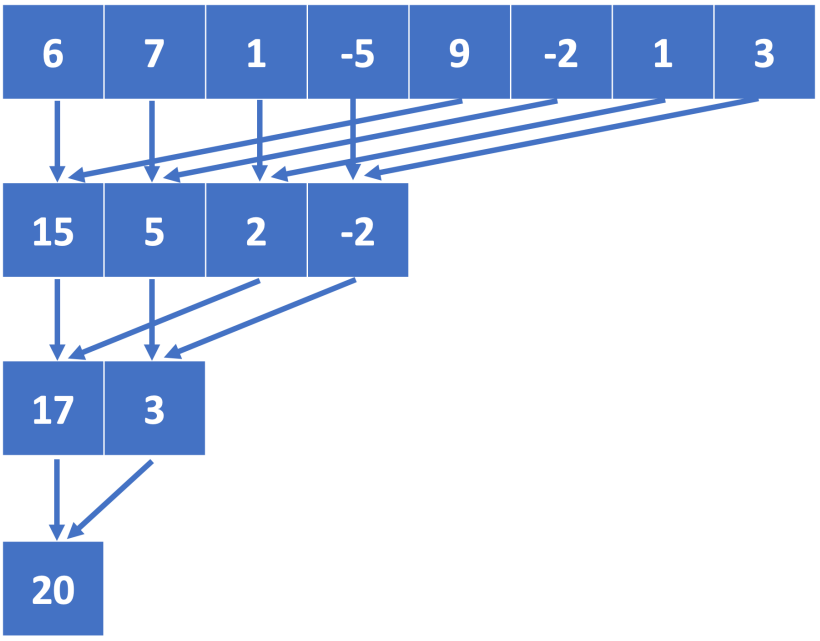
```
#define tpb 512 // size in this case has to be known at compile time
// this kernel has to be launched with at least N/2 threads
__global__ void reduction_one(double x, double *sum, int N){
  int ibl=blockIdx.y+blockIdx.x*gridDim.y;
  int ind=threadIdx.x+blockDim.x*ibl;

  __shared__ double shtmp[2*tpb];
  shtmp[threadIdx.x]=0; // for sums we initiate with 0, for other operations
should be different
  if(ind<N/2)
  {
     shtmp[threadIdx.x]=x[ind];
  }
  if(ind+N/2<N)
  {
     shtmp[threadIdx.x+tpb]=x[ind+N/2];
  }
  __syncthreads();
  for(int s=tpb;s>0;s>>=1){
    if(threadIdx.x<s){
       shtmp[threadIdx.x]+=shtmp[threadIdx.x+s];}
    __syncthreads();
  }
  if(threadIdx.x==0)
  {
    sum[ibl]=shtmp[0]; // each block saves its partial result to an array
    // atomicAdd(&sum[0], shene[0]); // alternatively could aggregate everything
together at index 0. Only use when there not many partial sums left
  }
}
```

In the kernel we have each GPU performing thread a reduction of two elements from the local portion of the array. If we have *tpb* GPU threads per block, we utilize them to store *2xtpb elements* in the local shared memory. To ensure synchronization until all data is available in the shared memory, we employ the *syncthreads()* function.

Next, we instruct each thread to “reduce” the element in the array at *threadIdx.x* with the element at *threadIdx.x+tpb*. As this operation saves the result back into the shared memory, we once again employ *syncthreads()*. By doing this, we effectively halve the number of elements to be reduced.

This procedure can be repeated, but now we only utilize *tpb/2 threads*. Each thread is responsible for “reducing” the element in the array at *threadIdx.x* with the element at *threadIdx.x+tpb/2*. After this step, we are left with *tpb/4* numbers to be reduced. We continue applying this procedure until only one number remains.

At this point, we can either “reduce” the final number with a global partial result using atomic read and write operations, or we can save it into an array for further processing.



Schematic representation on the reduction algorithm with 8 GPU threads.

For a detail analysis of how to optimize reduction operations in CUDA/HIP check this presentation Optimizing Parallel Reduction in CUDA

Reductions - In short

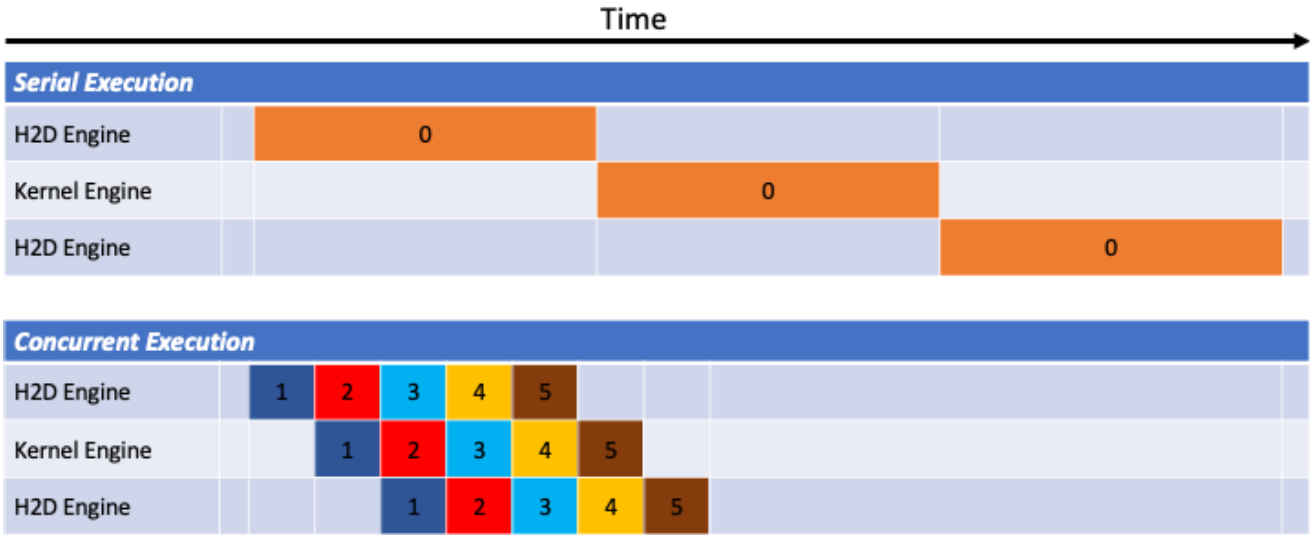
- Reductions refer to aggregating elements of an array into a single value through operations like summing, finding maximum or minimum, or performing logical operations.
- Performing reductions sequentially in a serial approach is inefficient for large problems, while parallel reduction on GPUs offers better performance.
- Parallel reduction on GPUs involves dividing the problem into subsets, performing reductions within blocks of threads using shared memory, and repeatedly reducing the number of elements (two per GPU thread) until only one remains.

Overlapping Computations and Memory transfer. CUDA/HIP Streams

Modern GPUs can overlap independent operations. They can do transfers between CPU and GPU and execute kernels in the same time, or they can execute kernels concurrently. CUDA/HIP streams are independent execution units, a sequence of operations that execute in issue-order on the GPU. The operations issue in different streams can be executed concurrently.

Consider the previous case of vector addition, which involves copying data from CPU to GPU, computations and then copying back the result to GPU. In this way nothing can be overlap.

We can improve the performance by dividing the problem in smaller independent parts. Let's consider 5 streams and consider the case where copy in one direction and computation take the same amount of time.



After the first and second stream copy data to the GPU, the GPU is practically occupied all time. We can see that significant performance improvements can be obtained by eliminating the time in which the GPU is idle, waiting for data to arrive from the CPU. This very useful for problems where there is often communication to the CPU because the GPU memory can not fit all the problem or the application runs in a multi-GPU set up and communication is needed often.

We can apply this to the vector addition problem above.

CUDA | HIP

```
// Distribute kernel for 'n_streams' streams, and record each stream's timing
for (int i = 0; i < n_streams; ++i) {
  int offset = i * stream_size;
  cudaEventRecord(start_event[i], stream[i]); // stamp the moment when the kernel
is submitted on stream i

  cudaMemcpyAsync( &Ad[offset],  &Ah[offset], N/n_streams*sizeof(float),
cudaMemcpyHostToDevice, stream[i]);
  cudaMemcpyAsync( &Bd[offset],  &Bh[offset], N/n_streams*sizeof(float),
cudaMemcpyHostToDevice, stream[i]);
  vector_add<<<gridsize / n_streams, blocksize, 0, stream[i]>>>(&Ad[offset],
&Bd[offset], &Cd[offset], N/n_streams); //each call processes N/n_streams elements
  cudaMemcpyAsync( &Ch[offset],  &Cd[offset], N/n_streams*sizeof(float),
cudaMemcpyDeviceToHost, stream[i]);

  cudaEventRecord(stop_event[i], stream[i]);  // stamp the moment when the kernel
on stream i finished
}
```

Instead of having one copy to gpu, one execution of the kernel and one copy back, we now have several of these calls independent of each other.

Note that even when streams are not explicitly used it is possible to launch all the GPU operations asynchronous and overlap CPU operations (such I/O) and GPU operations. In order to learn more about how to improve performance using streams check the NVIDIA blog How to Overlap Data Transfers in CUDA C/C++.

Streams - In short

- CUDA/HIP streams are independent execution contexts on the GPU that allow for concurrent execution of operations issued in different streams.
- Using streams can improve GPU performance by overlapping operations such as data transfers between CPU and GPU and kernel executions.
- By dividing a problem into smaller independent parts and utilizing multiple streams, the GPU can avoid idle time, resulting in significant performance improvements, especially for problems with frequent CPU communication or multi-GPU setups.

Pros and cons of native programming models

There are advantages and limitations to CUDA and HIP:

CUDA Pros:

1. Performance Boost: CUDA is designed for NVIDIA GPUs and delivers excellent performance.
2. Wide Adoption: CUDA is popular, with many resources and tools available.
3. Mature Ecosystem: NVIDIA provides comprehensive libraries and tools for CUDA programming.

HIP Pros:

1. Portability: HIP is portable across different GPU architectures.
2. Open Standards: HIP is based on open standards, making it more accessible.
3. Growing Community: The HIP community is growing, providing more resources and support.

Cons:

0. Exclusive for GPUs
1. Vendor Lock-in: CUDA is exclusive to NVIDIA GPUs, limiting compatibility.
2. Learning Curve: Both CUDA and HIP require learning GPU programming concepts.
3. Limited Hardware Support: HIP may face limitations on older or less common GPUs.

Keypoints

- CUDA and HIP are two GPU programming models
- Memory optimizations are very important

- Asynchronous launching can be used to overlap operations and avoid idle GPU

Portable kernel-based models

❓ Questions

- How to program GPUs with alpaka, C++ StdPar, Kokkos, OpenCL, and SYCL?
- What are the differences between these programming models.

Objectives

- Be able to use portable kernel-based models to write simple codes
- Understand how different approaches to memory and synchronization in Kokkos and SYCL work

Instructor note

- 60 min teaching
- 30 min exercises

The goal of the cross-platform portability ecosystems is to allow the same code to run on multiple architectures, therefore reducing code duplication. They are usually based on C++, and use function objects/lambda functions to define the loop body (i.e., the kernel), which can run on multiple architectures like CPU, GPU, and FPGA from different vendors. An exception to this is OpenCL, which originally offered only a C API (although currently also C++ API is available), and uses a separate-source model for the kernel code. However, unlike in many conventional CUDA or HIP implementations, the portability ecosystems require kernels to be written only once if one prefers to run it on CPU and GPU for example. Some notable cross-platform portability ecosystems are alpaka, Kokkos, OpenCL, RAJA, and SYCL. Kokkos, alpaka, and RAJA are individual projects whereas OpenCL and SYCL are standards followed by several projects implementing (and extending) them. For example, some notable SYCL implementations include Intel oneAPI DPC++, AdaptiveCpp (previously known as hipSYCL or Open SYCL), triSYCL, and ComputeCPP.

C++ StdPar

In C++17, the initial support for parallel execution of standard algorithms has been introduced. Most algorithms available via the standard `<algorithms>` header were given an overload accepting with an *execution policy* argument which allows the programmer to request parallel execution of the standard library function. While the main goal was to allow low-effort, high-level interface to run existing algorithms like `std::sort` on many CPU cores, implementations are allowed to use other hardware, and functions like `std::for_each` or `std::transform` offer great flexibility in writing the algorithm.

C++ StdPar, also called Parallel STL or PSTL, could be considered similar to directive-based models, as it is very high-level and does not give the programmer fine-grained control over data movement or any access to hardware-specific features like shared (local) memory. Even the GPU to run on is selected automatically, since standard C++ does not have the concept of a *device* (but there are vendor extensions allowing the programmer more control) However, for applications that already relies on algorithms from C++ standard library, StdPar can be a good way to reap the performance benefits of both CPUs and GPUs with minimal code modifications.

For GPU programming, all three vendors offer their implementations of StdPar with the ability to offload code to the GPU: NVIDIA has `nvc++`, AMD has experimental roc-stdpar, and Intel offers StdPar offload with their oneAPI compiler. AdaptiveCpp offers an independent StdPar implementation, able to target devices from all three vendors. While being a part of the C++ standard, the level of support and the maturity of StdPar implementations varies a lot between different compilers: not all compilers support all algorithms, and different heuristics for mapping the algorithm to hardware and for managing data movement can have effect on performance.

StdPar compilation

The build process depends a lot on the used compiler:

- AdaptiveCpp: Add `--acpp-stdpar` flag when calling `acpp`.
- Intel oneAPI: Add `-fsycl -fsycl-pstl-offload=gpu` flags when calling `icpx`.
- NVIDIA NVC++: Add `-stdpar` flag when calling `nvc++` (not supported with plain `nvcc`).

StdPar programming

In its simplest form, using C++ standard parallelism requires including an additional `<execution>` header and adding one argument to a supported standard library function.

For example, let's look at the following sequential code sorting a vector:

```
#include <algorithm>
#include <vector>

void f(std::vector<int>& a) {
  std::sort(a.begin(), a.end());
}
```

To make it run sorting on the GPU, only a minor modification is needed:

```cpp
#include <algorithm>
#include <vector>
#include <execution> // To get std::execution

void f(std::vector<int>& a) {
  std::sort(
      std::execution::par_unseq, // This algorithm can be run in parallel
      a.begin(), a.end()
    );
}
```

Now, when compiled with one of the supported compilers, the code will run the sorting on a GPU.

While the can initially seem very limiting, many standard algorithms, such as `std::transform`, `std::accumulate`, `std::transform_reduce`, and `std::for_each` can run custom functions over an array, thus allowing one to offload an arbitrary algorithm, as long as it does not violate typical limitations of GPU kernels, such as not throwing any exceptions and not doing system calls.

StdPar execution policies

In C++, there are four different execution policies to choose from:

- `std::execution::seq`: run algorithm serially, don't parallelize it.
- `std::execution::par`: allow parallelizing the algorithm (as if using multiple threads),
- `std::execution::unseq`: allow vectorizing the algorithm (as if using SIMD),
- `std::execution::par_unseq`: allow both vectorizing and parallelizing the algorithm.

The main difference between `par` and `unseq` is related to thread progress and locks: using `unseq` or `par_unseq` requires that the algorithms does not contain mutexes and other locks between the processes, while `par` does not have this limitation.

For GPU, the optimal choice is `par_unseq`, since this places the least requirement on the compiler in terms of operation ordering. While `par` is also supported in some cases, it is best avoided, both due to limited compiler support and as an indication that the algorithm is likely a poor fit for the hardware.

Kokkos

Kokkos is an open-source performance portability ecosystem for parallelization on large heterogeneous hardware architectures of which development has mostly taken place on Sandia National Laboratories. The project started in 2011 as a parallel C++ programming model, but have since expanded into a more broad ecosystem including Kokkos Core (the programming model), Kokkos Kernels (math library), and Kokkos Tools (debugging, profiling

and tuning tools). By preparing proposals for the C++ standard committee, the project also aims to influence the ISO/C++ language standard such that, eventually, Kokkos capabilities will become native to the language standard. A more detailed introduction is found HERE.

The Kokkos library provides an abstraction layer for a variety of different parallel programming models, currently CUDA, HIP, SYCL, HPX, OpenMP, and C++ threads. Therefore, it allows better portability across different hardware manufactured by different vendors, but introduces an additional dependency to the software stack. For example, when using CUDA, only CUDA installation is required, but when using Kokkos with NVIDIA GPUs, Kokkos and CUDA installation are both required. Kokkos is not a very popular choice for parallel programming, and therefore, learning and using Kokkos can be more difficult compared to more established programming models such as CUDA, for which a much larger amount of search results and Stack Overflow discussions can be found.

Kokkos compilation

Furthermore, one challenge with some cross-platform portability libraries is that even on the same system, different projects may require different combinations of compilation settings for the portability library. For example, in Kokkos, one project may wish the default execution space to be a CUDA device, whereas another requires a CPU. Even if the projects prefer the same execution space, one project may desire the Unified Memory to be the default memory space and the other may wish to use pinned GPU memory. It may be burdensome to maintain a large number of library instances on a single system.

However, Kokkos offers a simple way to compile Kokkos library simultaneously with the user project. This is achieved by specifying Kokkos compilation settings (see HERE) and including the Kokkos Makefile in the user Makefile. CMake is also supported. This way, the user application and Kokkos library are compiled together. The following is an example Makefile for a single-file Kokkos project (hello.cpp) that uses CUDA (Volta architecture) as the backend (default execution space) and Unified Memory as the default memory space:

Makefile for hello.cpp

```
default: build

# Set compiler
KOKKOS_PATH = $(shell pwd)/kokkos
CXX = hipcc
# CXX = ${KOKKOS_PATH}/bin/nvcc_wrapper

# Variables for the Makefile.kokkos
KOKKOS_DEVICES = "HIP"
# KOKKOS_DEVICES = "Cuda"
KOKKOS_ARCH = "VEGA90A"
# KOKKOS_ARCH = "Volta70"
KOKKOS_CUDA_OPTIONS = "enable_lambda,force_uvm"

# Include Makefile.kokkos
include $(KOKKOS_PATH)/Makefile.kokkos

build: $(KOKKOS_LINK_DEPENDS) $(KOKKOS_CPP_DEPENDS) hello.cpp
 $(CXX) $(KOKKOS_CPPFLAGS) $(KOKKOS_CXXFLAGS) $(KOKKOS_LDFLAGS) hello.cpp
$(KOKKOS_LIBS) -o hello
```

To build a **hello.cpp** project with the above Makefile, no steps other than cloning the Kokkos project into the current directory is required.

Kokkos programming

When starting to write a project using Kokkos, the first step is understand Kokkos initialization and finalization. Kokkos must be initialized by calling `Kokkos::initialize(int& argc, char* argv[])` and finalized by calling `Kokkos::finalize()`. More details are given in HERE.

Kokkos uses an execution space model to abstract the details of parallel hardware. The execution space instances map to the available backend options such as CUDA, OpenMP, HIP, or SYCL. If the execution space is not explicitly chosen by the programmer in the source code, the default execution space `Kokkos::DefaultExecutionSpace` is used. This is chosen when the Kokkos library is compiled. The Kokkos execution space model is described in more detail in HERE.

Similarly, Kokkos uses a memory space model for different types of memory, such as host memory or device memory. If not defined explicitly, Kokkos uses the default memory space specified during Kokkos compilation as described HERE.

The following is an example of a Kokkos program that initializes Kokkos and prints the execution space and memory space instances:

hello.cpp

```
#include <Kokkos_Core.hpp>
#include <iostream>

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);
  std::cout << "Execution Space: " <<
    typeid(Kokkos::DefaultExecutionSpace).name() << std::endl;
  std::cout << "Memory Space: " <<
    typeid(Kokkos::DefaultExecutionSpace::memory_space).name() << std::endl;
  Kokkos::finalize();
  return 0;
}
```

With Kokkos, the data can be accessed either through raw pointers or through Kokkos Views. With raw pointers, the memory allocation into the default memory space can be done using `Kokkos::kokkos_malloc(n * sizeof(int))`. Kokkos Views are a data type that provides a way to access data more efficiently in memory corresponding to a certain Kokkos memory space, such as host memory or device memory. A 1-dimensional view of type int* can be created by `Kokkos::View<int*> a("a", n)`, where `"a"` is a label, and `n` is the size of the allocation in the number of integers. Kokkos determines the optimal layout for the data at compile time for best overall performance as a function of the computer architecture. Furthermore, Kokkos handles the deallocation of such memory automatically. More details about Kokkos Views are found HERE.

Finally, Kokkos provides three different parallel operations: `parallel_for`, `parallel_reduce`, and `parallel_scan`. The `parallel_for` operation is used to execute a loop in parallel. The `parallel_reduce` operation is used to execute a loop in parallel and reduce the results to a single value. The `parallel_scan` operation implements a prefix scan. The usage of `parallel_for` and `parallel_reduce` are demonstrated in the examples later in this chapter. More detail about the parallel operations are found HERE.

Run Kokkos hello.cpp example in simple steps

The following should work on AMD VEGA90A devices straight out of the box (needs ROCm installation). On NVIDIA Volta V100 devices (needs CUDA installation), use the variables commented out on the Makefile.

1. `git clone https://github.com/kokkos/kokkos.git`
2. Copy the above Makefile into the current folder (make sure the indentation of the last line is tab, and not space)
3. Copy the above hello.cpp file into the current folder
4. `make`
5. `./hello`

OpenCL

OpenCL is a cross-platform, open-standard API for writing parallel programs that execute across heterogeneous platforms consisting of CPUs, GPUs, FPGAs and other devices. The first version of OpenCL (1.0) was released in December 2008, and the latest version of OpenCL (3.0) was released in September 2020. OpenCL is supported by a number of vendors, including AMD, ARM, Intel, NVIDIA, and Qualcomm. It is a royalty-free standard, and the OpenCL specification is maintained by the Khronos Group. OpenCL provides a low-level programming interface initially based on C, but more recently also a C++ interface has become available.

OpenCL compilation

OpenCL supports two modes for compiling the programs: online and offline. Online compilation occurs at runtime, when the host program calls a function to compile the source code. Online mode allows dynamic generation and loading of kernels, but may incur some overhead due to compilation time and possible errors. Offline compilation occurs before runtime, when the source code of a kernel is compiled into a binary format that can be loaded by the host program. This mode allows faster execution and better optimization of kernels, but may limit the portability of the program, because the binary can only run on the architectures it was compiled for.

OpenCL comes bundled with several parallel programming ecosystems, such as NVIDIA CUDA and Intel oneAPI. For example, after successfully installing such packages and setting up the environment, one may simply compile an OpenCL program by the commands such as `icx cl_devices.c -lOpenCL` (Intel oneAPI) or `nvcc cl_devices.c -lOpenCL` (NVIDIA CUDA), where `cl_devices.c` is the compiled file. Unlike most other programming models, OpenCL stores kernels as text and compiles them for the device in runtime (JIT-compilation), and thus does not require any special compiler support: one can compile the code using simply `gcc cl_devices.c -lOpenCL` (or `g++` when using C++ API), as long as the required libraries and headers are installed in a standard locations.

The AMD compiler installed on LUMI supports both OpenCL C and C++ API, the latter with some limitations. To compile a program, you can use the AMD compilers on a GPU partition:

```
$ module load LUMI/24.03 partition/G
$ module load rocm/6.0.3
$ module load PrgEnv-cray-amd
$ CC program.cpp -lOpenCL -o program # C++ program
$ cc program.c -lOpenCL -o program # C program
```

OpenCL programming

OpenCL programs consist of two parts: a host program that runs on the host device (usually a CPU) and one or more kernels that run on compute devices (such as GPUs). The host program is responsible for the tasks such as managing the devices for the selected platform, allocating memory objects, building and enqueueing kernels, and managing memory objects.

The first steps when writing an OpenCL program are to initialize the OpenCL environment by selecting the platform and devices, creating a context or contexts associated with the selected device(s), and creating a command queue for each device. A simple example of selecting the default device, creating a context and a queue associated with the device is show below.

OpenCL initialization (C++ API) | OpenCL initialization (C API)

```
// Initialize OpenCL
cl::Device device = cl::Device::getDefault();
cl::Context context(device);
cl::CommandQueue queue(context, device);
```

OpenCL provides two main programming models to manage the memory hierarchy of host and accelerator devices: buffers and shared virtual memory (SVM). Buffers are the traditional memory model of OpenCL, where the host and the devices have separate address spaces and the programmer has to explicitly specify the memory allocations and how and where the memory is accessed. This can be done with class `cl::Buffer` and functions such as `cl::CommandQueue::enqueueReadBuffer()`. Buffers are supported since early versions of OpenCL, and work well across different architectures. Buffers can also take advantage of device-specific memory features, such as constant or local memory.

SVM is a newer memory model of OpenCL, introduced in version 2.0, where the host and the devices share a single virtual address space. Thus, the programmer can use the same pointers to access the data from host and devices simplifying the programming effort. In OpenCL, SVM comes in different levels such as coarse-grained buffer SVM, fine-grained buffer SVM, and fine-grained system SVM. All levels allow using the same pointers across a host and devices, but they differ in their granularity and synchronization requirements for the memory regions. Furthermore, the support for SVM is not universal across all OpenCL platforms and devices, and for example, GPUs such as NVIDIA V100 and A100 only support the coarse-grained SVM buffer. This level requires explicit synchronization for memory accesses from a host and devices (using functions such as `cl::CommandQueue::enqueueMapSVM()` and `cl::CommandQueue::enqueueUnmapSVM()`), making the usage of SVM less convenient. It is further noted that this is unlike the regular Unified Memory offered by CUDA, which is closer to the fine-grained system SVM level in OpenCL.

OpenCL uses a separate-source kernel model where the kernel code is often kept in separate files that may be compiled during runtime. The model allows the kernel source code to be passed as a string to the OpenCL driver after which the program object can be executed on a specific device. Although referred to as the separate-source kernel model, the kernels can still be defined as a string in the host program compilation units as well, which may be a more convenient approach in some cases.

The online compilation with the separate-source kernel model has several advantages over the binary model, which requires offline compilation of kernels into device-specific binaries that can are loaded by the application at runtime. Online compilation preserves the portability and flexibility of OpenCL, as the same kernel source code can run on any supported device. Furthermore, dynamic optimization of kernels based on runtime information, such as input size, work-group size, or device capabilities, is possible. An example of an OpenCL kernel, defined by a string in the host compilation unit, and assigning the global thread index into a global device memory is shown below.

OpenCL kernel example

```
static const std::string kernel_source = R"(
  __kernel void dot(__global int *a) {
    int i = get_global_id(0);
    a[i] = i;
  }
)";
```

The above kernel named `dot` and stored in the string `kernel_source` can be set to build in the host code as follows:

OpenCL kernel build example (C++ API) | OpenCL kernel build example (C API)

```
cl::Program program(context, kernel_source);
program.build({device});
cl::Kernel kernel_dot(program, "dot");
```

SYCL

SYCL is a royalty-free, open-standard C++ programming model for multi-device programming. It provides a high-level, single-source programming model for heterogeneous systems, including GPUs. There are several implementations of the standard. For GPU

programming, Intel oneAPI DPC++ and AdaptiveCpp (also known as hipSYCL) are the most popular for desktop and HPC GPUs; ComputeCPP is a good choice for embedded devices. The same standard-compliant SYCL code should work with any implementation, but they are not binary-compatible.

The most recent version of the SYCL standard is SYCL 2020, and it is the version we will be using in this course.

SYCL compilation

Intel oneAPI DPC++

For targeting Intel GPUs, it is enough to install Intel oneAPI Base Toolkit. Then, the compilation is as simple as `icpx -fsycl file.cpp`.

It is also possible to use oneAPI for NVIDIA and AMD GPUs. In addition to oneAPI Base Toolkit, the vendor-provided runtime (CUDA or HIP) and the corresponding Codeplay oneAPI plugin must be installed. Then, the code can be compiled using Intel LLVM compiler bundled with oneAPI:

- `clang++ -fsycl -fsycl-targets=nvidia_gpu_sm_86 file.cpp` for targeting CUDA 8.6 NVIDIA GPU,
- `clang++ -fsycl -fsycl-targets=amd_gpu_gfx90a` for targeting GFX90a AMD GPU.

AdaptiveCpp

Using AdaptiveCpp for NVIDIA or AMD GPUs also requires having CUDA or HIP installed first. Then `acpp` can be used for compiling the code, specifying the target devices. For example, here is how to compile the program supporting an AMD and an NVIDIA device:

- `acpp --acpp-targets='hip:gfx90a;cuda:sm_70' file.cpp`

Using SYCL on LUMI

LUMI does not have a system-wide installation of any SYCL framework, but a recent AdaptiveCpp installation is available in CSC modules:

```
$ module load LUMI/24.03 partition/G
$ module load rocm/6.0.3
$ module use /appl/local/csc/modulefiles
$ module load acpp/24.06.0
```

The default compilation target is preset to MI250 GPUs, so to compile a single C++ file it is enough to call `acpp -O2 file.cpp`.

When running applications built with AdaptiveCpp, one can often see the warning "dag_direct_scheduler: Detected a requirement that is neither of discard access mode", reflecting the lack of an optimization hint when using buffer-accessor model. The warning is harmless and can be ignored.

SYCL programming

SYCL is, in many aspects, similar to OpenCL, but uses, like Kokkos, a single-source model with kernel lambdas.

To submit a task to device, first a *sycl::queue* must be created, which is used as a way to manage the task scheduling and execution. In the simplest case, that's all the initialization one needs:

```
int main() {
  // Create an out-of-order queue on the default device:
  sycl::queue q;
  // Now we can submit tasks to q!
}
```

If one wants more control, the device can be explicitly specified, or additional properties can be passed to a queue:

```
// Iterate over all available devices
for (const auto &device : sycl::device::get_devices()) {
  // Print the device name
  std::cout << "Creating a queue on " << device.get_info<sycl::info::device::name>() <<
"\n";
  // Create an in-order queue for the current device
  sycl::queue q(device, {sycl::property::queue::in_order()});
  // Now we can submit tasks to q!
}
```

Memory management can be done in two different ways: *buffer-accessor* model and *unified shared memory* (USM). The choice of the memory management models also influences how the GPU tasks are synchronized.

In the *buffer-accessor* model, a `sycl::buffer` objects are used to represent arrays of data. A buffer is not mapped to any single one memory space, and can be migrated between the GPU and the CPU memory transparently. The data in `sycl::buffer` cannot be read or written directly, an accessor must be created. `sycl::accessor` objects specify the location of data access (host or a certain GPU kernel) and the access mode (read-only, write-only, read-write). Such approach allows optimizing task scheduling by building a directed acyclic graph (DAG) of data dependencies: if kernel *A* creates a write-only accessor to a buffer, and then kernel *B* is submitted with a read-only accessor to the same buffer, and then a host-side read-only accessor is requested, then it can be deduced that *A* must complete before *B* is launched and

also that the results must be copied to the host before the host task can proceed, but the host task can run in parallel with kernel *B*. Since the dependencies between tasks can be built automatically, by default SYCL uses *out-of-order queues*: when two tasks are submitted to the same `sycl::queue`, it is not guaranteed that the second one will launch only after the first one completes. When launching a kernel, accessors must be created:

```
// Create a buffer of n integers
auto buf = sycl::buffer<int>(sycl::range<1>(n));
// Submit a kernel into a queue; cgh is a helper object
q.submit([&](sycl::handler &cgh) {
  // Create write-only accessor for buf
  auto acc = buf.get_access<sycl::access_mode::write>(cgh);
  // Define a kernel: n threads execute the following lambda
  cgh.parallel_for<class KernelName>(sycl::range<1>{n}, [=](sycl::id<1> i) {
      // The data is written to the buffer via acc
      acc[i] = /*...*/
  });
});
/* If we now submit another kernel with accessor to buf, it will not
 * start running until the kernel above is done */
```

Buffer-accessor model simplifies many aspects of heterogeneous programming and prevents many synchronization-related bugs, but it only allows very coarse control of data movement and kernel execution.

The *USM* model is similar to how NVIDIA CUDA or AMD HIP manage memory. The programmer has to explicitly allocate the memory on the device (`sycl::malloc_device`), on the host (`sycl::malloc_host`), or in the shared memory space (`sycl::malloc_shared`). Despite its name, unified shared memory, and the similarity to OpenCL's SVM, not all USM allocations are shared: for example, a memory allocated by `sycl::malloc_device` cannot be accessed from the host. The allocation functions return memory pointers that can be used directly, without accessors. This means that the programmer have to ensure the correct synchronization between host and device tasks to avoid data races. With USM, it is often convenient to use *in-order queues* with USM, instead of the default *out-of-order* queues. More information on USM can be found in the Section 4.8 of SYCL 2020 specification.

```
// Create a shared (migratable) allocation of n integers
// Unlike with buffers, we need to specify a queue (or, explicitly, a device and a
context)
int* v = sycl::malloc_shared<int>(n, q);
// Submit a kernel into a queue; cgh is a helper object
q.submit([&](sycl::handler &cgh) {
  // Define a kernel: n threads execute the following lambda
  cgh.parallel_for<class KernelName>(sycl::range<1>{n}, [=](sycl::id<1> i) {
      // The data is directly written to v
      v[i] = /*...*/
  });
});
// If we want to access v, we have to ensure that the kernel has finished
q.wait();
// After we're done, the memory must be deallocated
sycl::free(v, q);
```

Exercise

✍️ Exercise: Implement SAXPY in SYCL

In this exercise we would like to write (fill-in-the-blanks) a simple code doing SAXPY (vector addition).

To compile and run the code interactively, first make an allocation and load the AdaptiveCpp module:

```
$ salloc -A project_465002387 -N 1 -t 1:00:00 -p standard-g --gpus-per-node=1
....
salloc: Granted job allocation 123456

$ module load LUMI/24.03 partition/G
$ module use /appl/local/csc/modulefiles
$ module load rocm/6.0.3 acpp/24.06.0
```

Now you can run a simple device-detection utility to check that a GPU is available (note `srun`):

```
$ srun acpp-info -l
=================Backend information==================
Loaded backend 0: HIP
  Found device: AMD Instinct MI250X
Loaded backend 1: OpenMP
  Found device: hipSYCL OpenMP host device
```

If you have not done it already, clone the repository using `git clone https://github.com/ENCCS/gpu-programming.git` or **update it** using `git pull origin main`.

Now, let's look at the example code in `content/examples/portable-kernel-models/exercise-syc1-saxpy.cpp`:

```cpp
#include <iostream>
#include <sycl/sycl.hpp>
#include <vector>

int main() {
  // Create an in-order queue
  sycl::queue q{sycl::property::queue::in_order()};
  // Print the device name, just for fun
  std::cout << "Running on "
            << q.get_device().get_info<sycl::info::device::name>() << std::endl;
  const int n = 1024; // Vector size

  // Allocate device and host memory for the first input vector
  float *d_x = sycl::malloc_device<float>(n, q);
  float *h_x = sycl::malloc_host<float>(n, q);
  // Bonus question: Can we use `std::vector` here instead of `malloc_host`?
  // TODO: Allocate second input vector on device and host, d_y and h_y
  // Allocate device and host memory for the output vector
  float *d_z = sycl::malloc_device<float>(n, q);
  float *h_z = sycl::malloc_host<float>(n, q);

  // Initialize values on host
  for (int i = 0; i < n; i++) {
    h_x[i] = i;
    // TODO: Initialize h_y somehow
  }
  const float alpha = 0.42f;

  q.copy<float>(h_x, d_x, n);
  // TODO: Copy h_y to d_y
  // Bonus question: Why don't we need to wait before using the data?

  // Run the kernel
  q.parallel_for(sycl::range<1>{n}, [=](sycl::id<1> i) {
    // TODO: Modify the code to compute z[i] = alpha * x[i] + y[i]
    d_z[i] = alpha * d_x[i];
  });

  // TODO: Copy d_z to h_z
  // TODO: Wait for the copy to complete

  // Check the results
  bool ok = true;
  for (int i = 0; i < n; i++) {
    float ref = alpha * h_x[i] + h_y[i]; // Reference value
    float tol = 1e-5;                    // Relative tolerance
    if (std::abs((h_z[i] - ref)) > tol * std::abs(ref)) {
      std::cout << i << " " << h_z[i] << " " << h_x[i] << " " << h_y[i]
                << std::endl;
      ok = false;
      break;
    }
  }
  if (ok)
    std::cout << "Results are correct!" << std::endl;
  else
    std::cout << "Results are NOT correct!" << std::endl;

  // Free allocated memory
  sycl::free(d_x, q);
  sycl::free(h_x, q);
  // TODO: Free d_y, h_y.
  sycl::free(d_y, q);
  sycl::free(h_y, q);
```

```
  return 0;
}
```

To compile and run the code, use the following command:

```
$ acpp -O3 exercise-sycl-saxpy.cpp -o exercise-sycl-saxpy
$ srun ./exercise-sycl-saxpy
Running on AMD Instinct MI250X
Results are correct!
```

The code will not compile as-is! Your task is to fill in missing bits indicated by `TODO` comments. You can also test your understanding using the "Bonus questions" in the code.

If you feel stuck, take a look at the `exercise-sycl-saxpy-solution.cpp` file.

alpaka

The alpaka library is an open-source header-only C++20 abstraction library for accelerator development.

Its aim is to provide performance portability across accelerators by abstracting the underlying levels of parallelism. The project provides a single-source C++ API that enables developers to write parallel code once and run it on different hardware architectures without modification. The name "alpaka" comes from **A**bstractions for **L**evels of **P**arallelism, **A**lgorithms, and **K**ernels for **A**ccelerators. The library is platform-independent and supports the concurrent and cooperative use of multiple devices, including host CPUs (x86, ARM, and RISC-V) and GPUs from different vendors (NVIDIA, AMD, and Intel). A variety of accelerator backends, CUDA, HIP, SYCL, OpenMP, and serial execution, are available and can be selected based on the target device. Only a single implementation of a user kernel is required, expressed as a function object with a standardized interface. This eliminates the need to write specialized CUDA, HIP, SYCL, OpenMP, Intel TBB or threading code. Moreover, multiple accelerator backends can be combined to target different vendor hardware within a single system and even within a single application.

The abstraction is based on a virtual index domain decomposed into equally sized chunks called frames. **alpaka** provides a uniform abstraction to traverse these frames, independent of the underlying hardware. Algorithms to be parallelized map the chunked index domain and native worker threads onto the data, expressing the computation as kernels that are executed in parallel threads (SIMT), thereby also leveraging SIMD units. Unlike native parallelism models such as CUDA, HIP, and SYCL, **alpaka** kernels are not restricted to three dimensions. Explicit caching of data within a frame via shared memory allows developers to fully unleash the performance of the compute device. Additionally, **alpaka** offers primitive

functions such as iota, transform, transform-reduce, reduce, and concurrent, simplifying the development of portable high-performance applications. Host, device, mapped, and managed multi-dimensional views provide a natural way to operate on data.

Here we demonstrate the usage of **alpaka3**, which is a complete rewrite of alpaka. It is planned to merge this separate codebase back into the mainline alpaka repository before the first release in Q2/Q3 of 2026. Nevertheless, the code is well-tested and can be used for development today.

Installing alpaka on your system

For ease of use, we recommend installing alpaka using CMake as described below. For other ways to use alpaka in your projects, see the alpaka3 documentation.

1. **Clone the repository**

   Clone the alpaka source code from GitHub to a directory of your choice:

   ```
   git clone https://github.com/alpaka-group/alpaka3.git
   cd alpaka
   ```

2. **Set installation directory**

   Set the `ALPAKA_DIR` environment variable to the directory where you want to install alpaka. This can be any directory you choose where you have write access.

   ```
   export ALPAKA_DIR=/path/to/your/alpaka/install/dir
   ```

3. **Build and install**

   Create a build directory and use CMake to build and install alpaka. We use `CMAKE_INSTALL_PREFIX` to tell CMake where to install the library.

   ```
   mkdir build
   cmake -B build -S . -DCMAKE_INSTALL_PREFIX=$ALPAKA_DIR
   cmake --build build --parallel
   ```

4. **Update environment**

   To make sure that other projects can find your alpaka installation, you should add the installation directory to your `CMAKE_PREFIX_PATH`. You can do this by adding the following line to your shell configuration file (e.g. `~/.bashrc`):

   ```
   export CMAKE_PREFIX_PATH=$ALPAKA_DIR:$CMAKE_PREFIX_PATH
   ```

You will need to source your shell configuration file or open a new terminal for the changes to take effect.

alpaka Compilation

We recommend building your projects which use alpaka using CMake. A variety of strategies can be used to deal with building your application for a specific device or set of devices. Here we show a minimal way to get started, but this is by no means the only way to set up your projects. Please refer to the alpaka3 documentation for alternative ways to use alpaka in your project, including a way to make your source code agnostic to the accelerator being targeted by defining a device specification in CMake.

The following example demonstrates a `CMakeLists.txt` for a single-file project using alpaka3 (`main.cpp` which is presented in the section below):

```
cmake_minimum_required(VERSION 3.25)
project(myAlpakaApp VERSION 1.0)

# Find installed alpaka
find_package(alpaka REQUIRED)

# Build the executable
add_executable(myAlpakaApp main.cpp)
target_link_libraries(myAlpakaApp PRIVATE alpaka::alpaka)
alpaka_finalize(myAlpakaApp)
```

Using alpaka on LUMI

To load the environment for using the AMD GPUs on LUMI with HIP, one can use the following modules -

```
$ module load LUMI/24.03 partition/G
$ module load rocm/6.0.3
$ module load buildtools/24.03
$ module load PrgEnv-amd
$ module load craype-accel-amd-gfx90a
$ export CXX=hipcc
```

alpaka Programming

When starting with alpaka3, the first step is understanding the **device selection model**. Unlike frameworks that require explicit initialization calls, alpaka3 uses a device specification to determine which backend and hardware to use. The device specification consists of two components:

- **API**: The parallel programming interface (host, cuda, hip, oneApi)
- **Device Kind**: The type of hardware (cpu, nvidiaGpu, amdGpu, intelGpu)

Here we specify and use these at runtime to select and initialize devices. The device selection process is described in detail in the alpaka3 documentation.

alpaka3 uses an **execution space model** to abstract parallel hardware details. A device selector is created using `alpaka::onHost::makeDeviceSelector(devSpec)`, which returns an object that can query available devices and create device instances for the selected backend.

The following example demonstrates a basic alpaka program that initializes a device and prints information about it:

```
#include <alpaka/alpaka.hpp>
#include <cstdlib>
#include <iostream>

namespace ap = alpaka;

auto getDeviceSpec()
{
    /* Select a device, possible combinations of api+deviceKind:
     * host+cpu, cuda+nvidiaGpu, hip+amdGpu, oneApi+intelGpu, oneApi+cpu,
     * oneApi+amdGpu, oneApi+nvidiaGpu
     */
    return ap::onHost::DeviceSpec{ap::api::hip, ap::deviceKind::amdGpu};
}

int main(int argc, char** argv)
{
    // Initialize device specification and selector
    ap::onHost::DeviceSpec devSpec = getDeviceSpec();
    auto deviceSelector = ap::onHost::makeDeviceSelector(devSpec);

    // Query available devices
    auto num_devices = deviceSelector.getDeviceCount();
    std::cout << "Number of available devices: " << num_devices << "\n";

    if (num_devices == 0) {
        std::cerr << "No devices found for the selected backend\n";
        return EXIT_FAILURE;
    }

    // Select and initialize the first device
    auto device = deviceSelector.makeDevice(0);
    std::cout << "Using device: " << device.getName() << "\n";

    return EXIT_SUCCESS;
}
```

alpaka3 provides memory management abstractions through buffers and views. Memory can be allocated on host or device using `alpaka::allocBuf<T, Idx>(device, extent)`. Data transfers between host and device are handled through `alpaka::memcpy(queue, dst, src)`. The library automatically manages memory layouts for optimal performance on different architectures.

For parallel execution, alpaka3 provides kernel abstractions. Kernels are defined as functors or lambda functions and executed using work division specifications that define the parallelization strategy. The framework supports various parallel patterns including element-wise operations, reductions, and scans.

Tour of alpaka Features

Now we will quickly explore the most commonly used features of alpaka and go over some basic usage. A quick reference of commonly used alpaka features is available here.

General setup: Include the consolidated header once and you are ready to start using alpaka.

```
#include <alpaka/alpaka.hpp>

namespace myProject
{
    namespace ap = alpaka;
    // Your code here
}
```

Accelerator, platform, and device management: Select devices by combining the desired API with the appropriate hardware kind using the device selector.

```
auto devSelector = ap::onHost::makeDeviceSelector(ap::api::hip,
ap::deviceKind::amdGpu);
if (devSelector.getDeviceCount() == 0)
{
    throw std::runtime_error("No device found!");
}
auto device = devSelector.makeDevice(0);
```

Queues and events: Create blocking or non-blocking queues per device, record events, and synchronize work as needed.

```
auto queue = device.makeQueue();
auto nonBlockingQueue = device.makeQueue(ap::queueKind::nonBlocking);
auto blockingQueue = device.makeQueue(ap::queueKind::blocking);

auto event = device.makeEvent();
queue.enqueue(event);
ap::onHost::wait(event);
ap::onHost::wait(queue);
```

Memory management: Allocate host, device, mapped, unified, or deferred buffers, create non-owning views, and move data portably with *memcpy*, *memset*, and *fill*.

```
auto hostBuffer = ap::onHost::allocHost<DataType>(extent3D);
auto devBuffer = ap::onHost::alloc<DataType>(device, extentMd);
auto devMappedBuffer = ap::onHost::allocMapped<DataType>(device, extentMd);

auto hostView = ap::makeView(api::host, externPtr, ap::Vec{numElements});
auto devNonOwningView = devBuffer.getView();

ap::onHost::memset(queue, devBuffer, uint8_t{0});
ap::onHost::memcpy(queue, devBuffer, hostBuffer);
ap::onHost::fill(queue, devBuffer, DataType{42});
```

Kernel execution: Build a *FrameSpec* manually or request one tuned for your data type, then enqueue kernels with automatic or explicit executors.

```
constexpr uint32_t dim = 2u;
using IdxType = size_t;
using DataType = int;

IdxType valueX, valueY;
auto extentMD = ap::Vec{valueY, valueX};

auto frameSpec = ap::onHost::FrameSpec{numFramesMd, frameExtentMd};
auto tunedSpec = ap::onHost::getFrameSpec<DataType>(device, extentMd);

queue.enqueue(tunedSpec, ap::KernelBundle{kernel, kernelArgs...});

auto executor = ap::exec::cpuSerial;
queue.enqueue(executor, tunedSpec, ap::KernelBundle{kernel, kernelArgs...});
```

Kernel implementation: Write kernels as functors annotated with *ALPAKA_FN_ACC*, use shared memory, synchronization, atomics, and math helpers directly inside the kernel body.

```
struct MyKernel
{
    ALPAKA_FN_ACC void operator()(ap::onAcc::concepts::Acc auto const& acc, auto...
args) const
    {
        auto idxMd = acc.getIdxWithin(ap::onAcc::origin::grid,
ap::onAcc::unit::blocks);

        auto sharedMdArray =
            ap::onAcc::declareSharedMdArray<float, ap::uniqueId()>(acc,
ap::CVec<uint32_t, 3, 4>{});

        ap::onAcc::syncBlockThreads(acc);
        auto old = onAcc::atomicAdd(acc, args...);
        ap::onAcc::memFence(acc, ap::onAcc::scope::block);
        auto sinValue = ap::math::sin(args[0]);
    }
};
```

Run alpaka3 Example in Simple Steps

The following example works on systems with CMake 3.25+ and an appropriate C++ compiler. For GPU execution, ensure the corresponding runtime (CUDA, ROCm, or oneAPI) is installed.

1. Create a directory for your project:

```
mkdir my_alpaka_project && cd my_alpaka_project
```

2. Copy the CMakeLists.txt from above into the current folder
3. Copy the main.cpp file into the current folder
4. Configure and build:

```
cmake -B build -S . -Dalpaka_DEP_HIP=ON
cmake --build build --parallel
```

5. Run the executable:

```
./build/myAlpakaApp
```

Note

The device specification system allows you to select the target device at CMake configuration time. The format is `"api:deviceKind"`, where:

- **api**: The parallel programming interface (`host`, `cuda`, `hip`, `oneApi`)
- **deviceKind**: The type of device (`cpu`, `nvidiaGpu`, `amdGpu`, `intelGpu`)

Available combinations are: `host:cpu`, `cuda:nvidiaGpu`, `hip:amdGpu`, `oneApi:cpu`, `oneApi:intelGpu`, `oneApi:nvidiaGpu`, `oneApi:amdGpu`

Warning

The CUDA, HIP, or Intel backends only work if the CUDA SDK, HIP SDK, or OneAPI SDK are available respectively

Expected output

```
Number of available devices: 1
Using device: [Device Name]
```

The device name will vary depending on your hardware (e.g., “NVIDIA A100”, “AMD MI250X”, or your CPU model).

Compile and Execute Examples

You can test the **alpaka** provided examples from the example section. The examples have hard coded the usage of the AMD ROCm platform required on LUMI. To switch to CPU usage only you can simply replace `ap::onHost::makeDeviceSelector(ap::api::hip, ap::deviceKind::amdGpu);` with `ap::onHost::makeDeviceSelector(ap::api::host, ap::deviceKind::cpu);`

The following steps assume you have downloaded alpaka already and the path to the **alapka** source code is stored in the environment variable `ALPAKA_DIR`. To test the example copy the code into a file `main.cpp`

Alternatively, click here to try the first example using in the godbolt compiler explorer.

HIP for AMD GPUs | Host compiler for CPU | CUDA for NVIDIA GPUs

oneAPI SYCL for CPU | oneAPI SYCL for Intel GPUs

oneAPI SYCL for AMD GPUs | oneAPI SYCL for NVIDIA GPUs

```
# use the following in C++ code
# auto devSelector = ap::onHost::makeDeviceSelector(ap::api::hip, ap::deviceKind::amdGpu);
# We use CC to refer to the compiler to work smoothly with the LUMI environment
CC -I $ALPAKA_DIR/include/ -std=c++20 -x hip --offload-arch=gfx90a main.cpp
./a.out
```

Exercise

✍ Exercise: Write a vector add kernel in alpaka

In this exercise we would like to write (fill-in-the-blanks) a simple kernel to add two vectors.

To compile and run the code interactively, first we first need to get an allocation on a GPU node and load the modules for alpaka:

```
$ srun -p dev-g --gpus 1 -N 1 -n 1 --time=00:20:00 --account=project_465002387 --
pty bash
....
srun: job 1234 queued and waiting for resources
srun: job 1234 has been allocated resources

$ module load LUMI/24.03 partition/G
$ module load rocm/6.0.3
$ module load buildtools/24.03
$ module load PrgEnv-amd
$ module load craype-accel-amd-gfx90a
$ export CXX=hipcc
```

Now you can run a simple device-detection utility to check that a GPU is available (note `srun`):

```
$ rocm-smi

======================================== ROCm System Management Interface
========================================
================================================ Concise Info
================================================
Device  [Model : Revision]    Temp    Power  Partitions      SCLK    MCLK     Fan
Perf    PwrCap  VRAM%  GPU%
        Name (20 chars)       (Edge)  (Avg)  (Mem, Compute)
================================================================================
0       [0x0b0c : 0x00]       45.0°C  N/A    N/A, N/A        800Mhz  1600Mhz  0%
manual  0.0W      0%   0%
        AMD INSTINCT MI200 (
================================================================================
============================================ End of ROCm SMI Log
============================================
```

Now, let's look at the code to set up the exercise:

Below we use fetch content with our CMake to get started with alpaka quickly.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.25)
project(vectorAdd LANGUAGES CXX VERSION 1.0)
#Use CMake's FetchContent to download and integrate alpaka3 directly from GitHub
include(FetchContent)
#Declare where to fetch alpaka3 from
#This will download the library at configure time
FetchContent_Declare(alpaka3 GIT_REPOSITORY https://github.com/alpaka-
group/alpaka3.git GIT_TAG dev)
#Make alpaka3 available for use in this project
#This downloads, configures, and makes the library targets available
FetchContent_MakeAvailable(alpaka3)
#Finalize the alpaka FetchContent setup
alpaka_FetchContent_Finalize()
#Create the executable target from the source file
add_executable(vectorAdd main.cpp)
#Link the alpaka library to the executable
target_link_libraries(vectorAdd PRIVATE alpaka::alpaka)
#Finalize the alpaka configuration for this target
#This sets up backend - specific compiler flags and dependencies
alpaka_finalize(vectorAdd)
```

Below we have the main alpaka code doing a vector addition on device using a high level transform function

main.cpp

```cpp
#include <alpaka/alpaka.hpp>

namespace ap = alpaka;

auto main() -> int {
  unsigned n = 5;

  /* Select a device, possible combinations:
   * host+cpu, cuda+nvidiaGpu, hip+amdGpu, oneApi+intelGpu, oneApi+cpu,
   * oneApi+amdGpu, oneApi+nvidiaGpu
   */

  // auto devSelector = ap::onHost::makeDeviceSelector(ap::api::host,
  // ap::deviceKind::cpu);
  auto devSelector =
      ap::onHost::makeDeviceSelector(ap::api::hip, ap::deviceKind::amdGpu);
  ap::onHost::Device devAcc = devSelector.makeDevice(0);
  printf("Using alpaka device: %s\n", devAcc.getName().c_str());

  // Blocking device queue (requires synchronization)
  ap::onHost::Queue queue = devAcc.makeQueue(ap::queueKind::blocking);

  // Allocate unified memory that is accessible on host and device
  auto a = ap::onHost::allocUnified<int>(devAcc, n);
  auto b = ap::onHost::allocUnified<int>(devAcc, n);
  auto c = ap::onHost::allocUnified<int>(devAcc, n);

  // Initialize values on host
  for (unsigned i = 0; i < n; i++) {
    a[i] = i;
    b[i] = 1;
  }

  // Run element-wise vector addition on device
  ap::onHost::transform(queue, c, std::plus{}, a, b);

  for (unsigned i = 0; i < n; i++) {
    printf("c[%d] = %d\n", i, c[i]);
  }

  return 0;
}
```

To set up our project, we create a folder and place our CMakeLists.txt and main.cpp in there.

```
$ mkdir alpakaExercise && cd alpakaExercise
$ vim CMakeLists.txt
and now paste the CMakeLsits here (Press i, followed by Ctrl+Shift+V)
Press esc and then :wq to exit vim
$ vim main.cpp
Similarly, paste the C++ code here
```

To compile and run the code, use the following commands:

```
configure step, we additionaly specify that HIP is available
$ cmake -B build -S . -Dalpaka_DEP_HIP=ON
build
$ cmake --build build --parallel
run
$ ./build/vectorAdd
Using alpaka device: AMD Instinct MI250X id=0
c[0] = 1
c[1] = 2
c[2] = 3
c[3] = 4
c[4] = 5
```

Now your task will be to write and launch your first alpaka kernel. This kernel will do the vector addition and we will use this instead of the transform helper.

✔ Writing the vector add kernel

```cpp
#include <alpaka/alpaka.hpp>

namespace ap = alpaka;

struct AddKernel {
  constexpr void operator()(ap::onAcc::concepts::Acc auto const &acc,
                            ap::concepts::IMdSpan auto c,
                            ap::concepts::IMdSpan auto const a,
                            ap::concepts::IMdSpan auto const b) const {
    for (auto idx : ap::onAcc::makeIdxMap(acc, ap::onAcc::worker::threadsInGrid,
                                          ap::IdxRange{c.getExtents()})) {
      c[idx] = a[idx] + b[idx];
    }
  }
};

auto main() -> int {
  unsigned n = 5;

  /* Select a device, possible combinations:
   * host+cpu, cuda+nvidiaGpu, hip+amdGpu, oneApi+intelGpu, oneApi+cpu,
   * oneApi+amdGpu, oneApi+nvidiaGpu
   */

  // auto devSelector = ap::onHost::makeDeviceSelector(ap::api::host,
  // ap::deviceKind::cpu);
  auto devSelector =
      ap::onHost::makeDeviceSelector(ap::api::hip, ap::deviceKind::amdGpu);
  ap::onHost::Device devAcc = devSelector.makeDevice(0);
  printf("Using alpaka device: %s\n", devAcc.getName().c_str());

  // Blocking device queue (requires synchronization)
  ap::onHost::Queue queue = devAcc.makeQueue(ap::queueKind::blocking);

  // Allocate unified memory that is accessible on host and device
  auto a = ap::onHost::allocUnified<int>(devAcc, n);
  auto b = ap::onHost::allocUnified<int>(devAcc, n);
  auto c = ap::onHost::allocUnified<int>(devAcc, n);

  // Initialize values on host
  for (unsigned i = 0; i < n; i++) {
    a[i] = i;
    b[i] = 1;
  }

  auto frameSpec = ap::onHost::getFrameSpec<int>(devAcc, c.getExtents());

  // Call the element-wise addition kernel on device
  queue.enqueue(frameSpec, ap::KernelBundle{AddKernel{}, c, a, b});

  for (unsigned i = 0; i < n; i++) {
    printf("c[%d] = %d\n", i, c[i]);
  }

  return 0;
}
```

Examples

Parallel for with Unified Memory

StdPar | Kokkos | OpenCL | SYCL | alpaka-algorithms | alpaka

```cpp
#include <algorithm>
#include <cstdio>
#include <execution>
#include <vector>

int main() {
  unsigned n = 5;

  // Allocate arrays
  std::vector<int> a(n), b(n), c(n);

  // Initialize values
  for (unsigned i = 0; i < n; i++) {
    a[i] = i;
    b[i] = 1;
  }

  // Run element-wise multiplication on device
  std::transform(std::execution::par_unseq, a.begin(), a.end(), b.begin(),
                 c.begin(), [](int i, int j) { return i * j; });

  for (unsigned i = 0; i < n; i++) {
    printf("c[%d] = %d\n", i, c[i]);
  }

  return 0;
}
```

Parallel for with GPU buffers

Kokkos | OpenCL | SYCL | alpaka-algorithms | alpaka

```cpp
#include <Kokkos_Core.hpp>

int main(int argc, char *argv[]) {

  // Initialize Kokkos
  Kokkos::initialize(argc, argv);

  {
    unsigned n = 5;

    // Allocate space for 5 ints on Kokkos host memory space
    Kokkos::View<int *, Kokkos::HostSpace> h_a("h_a", n);
    Kokkos::View<int *, Kokkos::HostSpace> h_b("h_b", n);
    Kokkos::View<int *, Kokkos::HostSpace> h_c("h_c", n);

    // Allocate space for 5 ints on Kokkos default memory space (eg, GPU memory)
    Kokkos::View<int *> a("a", n);
    Kokkos::View<int *> b("b", n);
    Kokkos::View<int *> c("c", n);

    // Initialize values on host
    for (unsigned i = 0; i < n; i++) {
      h_a[i] = i;
      h_b[i] = 1;
    }

    // Copy from host to device
    Kokkos::deep_copy(a, h_a);
    Kokkos::deep_copy(b, h_b);

    // Run element-wise multiplication on device
    Kokkos::parallel_for(n, KOKKOS_LAMBDA(const int i) { c[i] = a[i] * b[i]; });

    // Copy from device to host
    Kokkos::deep_copy(h_c, c);

    // Print results
    for (unsigned i = 0; i < n; i++)
      printf("c[%d] = %d\n", i, h_c[i]);
  }

  // Finalize Kokkos
  Kokkos::finalize();
  return 0;
}
```

Asynchronous parallel for kernels

Kokkos | OpenCL | SYCL | alpaka-algorithms | alpaka

```cpp
#include <Kokkos_Core.hpp>

int main(int argc, char *argv[]) {

  // Initialize Kokkos
  Kokkos::initialize(argc, argv);

  {
    unsigned n = 5;
    unsigned nx = 20;

    // Allocate on Kokkos default memory space (Unified Memory)
    int *a = (int *)Kokkos::kokkos_malloc(nx * sizeof(int));

    // Create 'n' execution space instances (maps to streams in CUDA/HIP)
    auto ex = Kokkos::Experimental::partition_space(
        Kokkos::DefaultExecutionSpace(), 1, 1, 1, 1, 1);

    // Launch 'n' potentially asynchronous kernels
    // Each kernel has their own execution space instances
    for (unsigned region = 0; region < n; region++) {
      Kokkos::parallel_for(
          Kokkos::RangePolicy<Kokkos::DefaultExecutionSpace>(
              ex[region], nx / n * region, nx / n * (region + 1)),
          KOKKOS_LAMBDA(const int i) { a[i] = region + i; });
    }

    // Sync execution space instances (maps to streams in CUDA/HIP)
    for (unsigned region = 0; region < n; region++)
      ex[region].fence();

    // Print results
    for (unsigned i = 0; i < nx; i++)
      printf("a[%d] = %d\n", i, a[i]);

    // Free Kokkos allocation (Unified Memory)
    Kokkos::kokkos_free(a);
  }

  // Finalize Kokkos
  Kokkos::finalize();
  return 0;
}
```

Reduction

StdPar | Kokkos | OpenCL | SYCL | alpaka-algorithms

```cpp
#include <cstdio>
#include <execution>
#include <numeric>
#include <vector>

int main() {
  unsigned n = 10;

  std::vector<int> a(n);

  std::iota(a.begin(), a.end(), 0); // Fill the array

  // Run reduction on the device
  int sum = std::reduce(std::execution::par_unseq, a.cbegin(), a.cend(), 0,
                        std::plus<int>{});

  // Print results
  printf("sum = %d\n", sum);

  return 0;
}
```

Pros and cons of cross-platform portability ecosystems

General observations

- The amount of code duplication is minimized.
- The same code can be compiled to multiple architectures from different vendors.
- Limited learning resources compared to CUDA (Stack Overflow, course material, documentation).

Lambda-based kernel models (Kokkos, SYCL)

- Higher level of abstraction.
- Less knowledge of the underlying architecture is needed for initial porting.
- Very nice and readable source code (C++ API).
- The models are relatively new and not very popular yet.

Functor-based kernel model (alpaka)

- Very good portability.
- Higher level of abstraction.
- Low-level API always available which gives more control and allows fine tuning.
- User friendly C++ API for both the host and kernel code.
- Small community and ecosystem.

Separate-source kernel models (OpenCL)

- Very good portability.
- Mature ecosystem.
- Limited number of vendor-provided libraries.
- Low-level API gives more control and allows fine tuning.
- Both C and C++ APIs available (C++ API is less well supported).
- The low-level API and separate-source kernel model are less user friendly.

C++ Standard Parallelism (StdPar, PSTL)

- Very high level of abstraction.
- Easy to speed up code which already relying on STL algorithms.
- Very little control over hardware.
- Support by compilers is improving, but is far from mature.

Keypoints

- General code organization is similar to non-portable kernel-based models.
- As long as no vendor-specific functionality is used, the same code can run on any GPU.

High-level language support

❓ Questions

- Can I port code in high-level languages to run on GPUs?

Objectives

- Get an overview of libraries for GPU programming in Python and Julia

Instructor note

- 40 min teaching
- 20 min exercises

Julia

Julia has first-class support for GPU programming through the following packages that target GPUs from all three major vendors:

- CUDA.jl for NVIDIA GPUs
- AMDGPU.jl for AMD GPUs
- oneAPI.jl for Intel GPUs
- Metal.jl for Apple M-series GPUs

`CUDA.jl` is the most mature, `AMDGPU.jl` is somewhat behind but still ready for general use, while `oneAPI.jl` and `Metal.jl` are functional but might contain bugs, miss some features and provide suboptimal performance.

The APIs of these libraries are completely analogous and translation between them is normally straightforward. The libraries offer both user-friendly **high-level abstractions** (the array interface and higher-level abstractions) that require little programming effort, and a **lower level** approach for writing kernels for fine-grained control.

Installing these packages is done with the Julia package manager:

NVIDIA | AMD | Intel | Apple

Installing `CUDA.jl`:

```
using Pkg
Pkg.add("CUDA")
```

To use the Julia GPU stack, one needs to have the relevant GPU drivers and programming toolkits installed. GPU drivers are already installed on HPC systems while on your own machine you will need to install them yourself (see e.g. these instructions from NVIDIA). Programming toolkits for CUDA can be installed automatically through Julia's artifact system upon the first usage:

```
using CUDA
CUDA.versioninfo()
```

The array interface

GPU programming with Julia can be as simple as using a different array type instead of regular `Base.Array` arrays:

- `CuArray` from CUDA.jl for NVIDIA GPUs
- `ROCArray` from AMDGPU.jl for AMD GPUs
- `oneArray` from oneAPI.jl for Intel GPUs
- `MtlArray` from Metal.jl for Apple GPUs

These array types closely resemble `Base.Array` which enables us to write generic code which works on both types.

The following code copies an array to the GPU and executes a simple operation on the GPU:

NVIDIA | AMD | Intel | Apple

```
using CUDA

A_d = CuArray([1,2,3,4])
A_d .+= 1
```

Moving an array back from the GPU to the CPU is simple:

```
A = Array(A_d)
```

Let's have a look at a more realistic example: matrix multiplication. We create two random arrays, one on the CPU and one on the GPU, and compare the performance using the BenchmarkTools package:

NVIDIA | AMD | Intel | Apple

```
using BenchmarkTools
using CUDA

A = rand(2^9, 2^9);
A_d = CuArray(A);

@btime $A * $A;
@btime CUDA.@sync $A_d * $A_d;
```

Vendor libraries

Support for using GPU vendor libraries from Julia is currently most mature on NVIDIA GPUs. NVIDIA libraries contain precompiled kernels for common operations like matrix multiplication (*cuBLAS*), fast Fourier transforms (*cuFFT*), linear solvers (*cuSOLVER*), etc. These kernels are wrapped in `CUDA.jl` and can be used directly with `CuArrays`:

```
# create a 100x100 Float32 random array and an uninitialized array
A = CUDA.rand(2^9, 2^9);
B = CuArray{Float32, 2}(undef, 2^9, 2^9);

# regular matrix multiplication uses cuBLAS under the hood
A * A

# use LinearAlgebra for matrix multiplication
using LinearAlgebra
mul!(B, A, A)

# use cuSOLVER for QR factorization
qr(A)

# solve equation A*X == B
A \ B

# use cuFFT for FFT
using CUDA.CUFFT
fft(A)
```

`AMDGPU.jl` currently supports some of the ROCm libraries:

- *rocBLAS* for BLAS support
- *rocFFT* for FFT support
- *rocRAND* for RNG support
- *MIOpen* for DNN support

Higher-order abstractions

A powerful way to program GPUs with arrays is through Julia's higher-order array abstractions. The simple element-wise addition we saw above, `a .+= 1`, is an example of this, but more general constructs can be created with `broadcast`, `map`, `reduce`, `accumulate` etc:

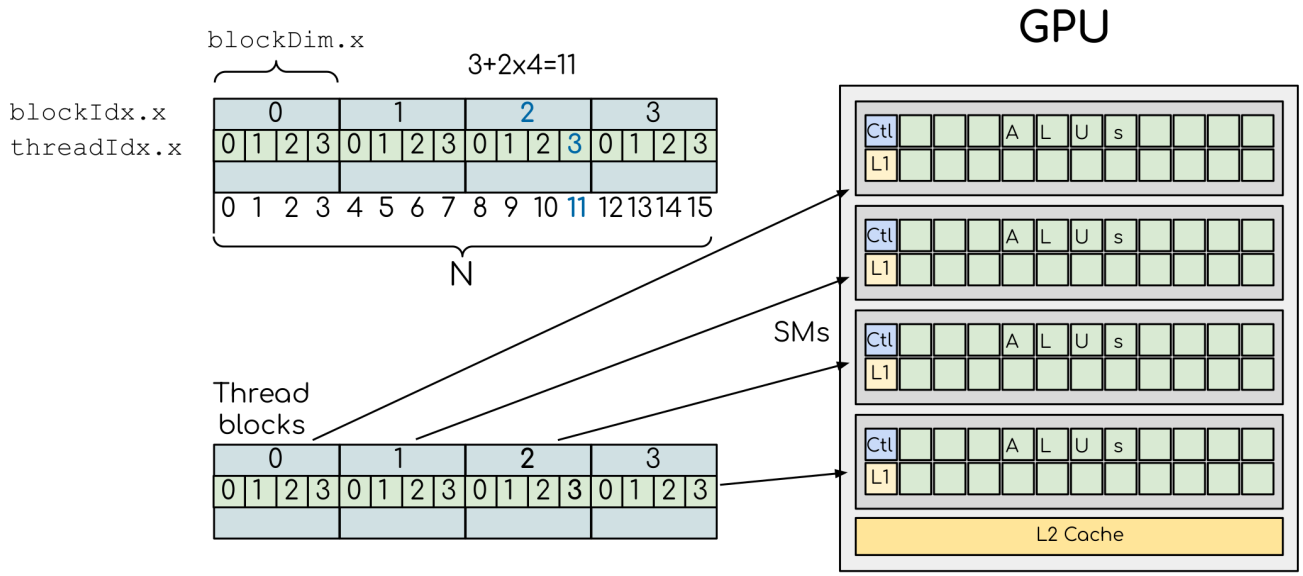
broadcast | map | reduce | accumulate

```
broadcast(A) do x
    x += 1
end
```

Writing your own kernels

Not all algorithms can be made to work with the higher-level abstractions in `CUDA.jl`. In such cases it's necessary to explicitly write our own GPU kernel.

Similarly to writing kernels in CUDA or HIP, we use a special function to return the index of the GPU thread which executes it (e.g., `threadIdx().x` for NVIDIA and `workitemIdx().x` for AMD), and two additional functions to parallelise over multiple blocks (e.g., **`blockDim().x()`** and **`blockIdx().x()`** for NVIDIA, and **`workgroupDim().x()`** and **`workgroupIdx().x()`** for AMD).



Here's an example of vector addition kernels for NVIDIA, AMD, Intel and Apple GPUs:

NVIDIA | AMD | Intel | Apple

```
using CUDA

function vadd!(C, A, B)
    i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
    if i <= length(A)
        @inbounds C[i] = A[i] + B[i]
    end
    return
end

A, B = CUDA.ones(2^9)*2, CUDA.ones(2^9)*3;
C = similar(A);

nthreads = 256
# smallest integer larger than or equal to length(A)/threads
numblocks = cld(length(A), nthreads)

# run using 256 threads
@cuda threads=nthreads blocks=numblocks vadd!(C, A, B)

@assert all(Array(C) .== 5.0)
```

Restrictions in kernel programming

Within kernels, most of the Julia language is supported with the exception of functionality that requires the Julia runtime library. This means one cannot allocate memory or perform dynamic function calls, both of which are easy to do accidentally!

1D, 2D and 3D

CUDA.jl and AMDGPU.jl support indexing in up to 3 dimensions (x, y and z, e.g. `threadIdx().x` and `workitemIdx().x`). This is convenient for multidimensional data where thread blocks can be organised into 1D, 2D or 3D arrays of threads.

Writing protable kernels

KernelAbstractions.jl allows you to write generic GPU code and run it on GPUs from Nvidia, AMD, Intel or Apple, similar to alpaka and Kokkos for C++. The backend is the object that decides where the code will be executed. A specific backend such as `ROCBackend()` becomes available when the corresponding package is loaded.

NVIDIA | AMD | Intel | Apple

```
using KernelAbstractions

using CUDA
backend = CUDABackend()

@kernel function vadd!(C, @Const(A), @Const(B))
   i = @index(Global)
   if i <= length(A)
      @inbounds C[i] = A[i] + B[i]
   end
end

A = KernelAbstractions.ones(backend, Float64, 2^9)*2;
B = KernelAbstractions.ones(backend, Float64, 2^9)*3;
C = similar(A)

kernel! = vadd!(backend)
kernel!(C, A, B, ndrange=size(C))
KernelAbstractions.synchronize(backend)

@assert all(Array(C) .== 5.0)
```

Python

There has been a lot of progress in GPU programming using Python and the ecosystem is still evolving. There are a couple of options available to work with GPU.

CuPy

CuPy is a NumPy/SciPy-compatible data array library used on GPU. It has been developed for NVIDIA GPUs but as experimental support for AMD GPUs. CuPy has a highly compatible interface with NumPy and SciPy. As stated on its official website, “All you need to do is just replace `numpy` and `scipy` with `cupy` and `cupyx.scipy` in your Python code.” If you know NumPy, CuPy is a very easy way to get started on the GPU.

cuDF

RAPIDS is a high level packages collections which implement CUDA functionalities and API with Python bindings. It only supports NVIDIA GPUs. cuDF belongs to RAPIDS and is the library for manipulating data frames on GPU. cuDF provides a pandas-like API, so if you are familiar with Pandas, you can accelerate your work without knowing too much CUDA programming.

PyCUDA

PyCUDA is a Python programming environment for CUDA. It allows users to access to NVIDIA’s CUDA API from Python. PyCUDA is powerful library but only runs on NVIDIA GPUs. Knowledge of CUDA programming is needed.

Numba

Numba allows users to just-in-time (JIT) compile Python code to run fast on CPUs, but can also be used for JIT compiling for GPUs. In the following we will focus on using Numba, which supports GPUs from both NVIDIA and AMD.

Using Numba in AMD GPUs

To use Numba with AMD GPUs `numba-hip` extension can be used. By adding the following lines in the beginning of the code, a single-source code can be made to work in both Nvidia and AMD GPUs:

```
try:
    from numba import hip
except ImportError:
    pass
else:
    hip.pose_as_cuda()
```

Read more how to install and use it here.

Numba supports GPU programming by directly compiling a restricted subset of Python code into kernels and device functions following the execution model. Kernels written in Numba appear to have direct access to NumPy arrays. NumPy arrays are transferred between the CPU and the GPU automatically.

ufunc (gufunc) decorator

Using ufuncs (and generalized ufuncs) is the easiest way to run on a GPU with Numba, and it requires minimal understanding of GPU programming. Numba `@vectorize` will produce a ufunc-like object. This object is a close analog but not fully compatible with a regular NumPy ufunc. Generating a ufunc for GPU requires the explicit type signature and target attribute.

Examples

👀 Demo: Numba ufunc

Let's look at a simple mathematical problem:

Python | Numba ufunc cpu | Numba ufunc gpu

```
import math

def f(x,y):
    return math.pow(x,3.0) + 4*math.sin(y)
```

To benchmark, first initialize:

```
import numpy as np
x = np.random.rand(10_000_000)
res = np.random.rand(10_000_000)
```

Python | Numba ufunc cpu | Numba ufunc gpu

```
%%timeit -r 1
for i in range(10000000):
    res[i]=f(x[i], x[i])
    # 6.75 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Numba `@vectorize` is limited to scalar arguments in the core function, for multi-dimensional arrays arguments, `@guvectorize` is used. Consider the following example which does matrix multiplication.

Warning

One should never implement things like matrix multiplication by oneself since there are plenty of highly optimized libraries available!

👀 Numba gufunc

Python | Numba gufunc cpu | Numba gufunc gpu

```
import numpy as np

def matmul_cpu(A,B,C):
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            tmp=0.0
            for k in range(B.shape[0]):
                tmp += A[i, k] * B[k, j]
            C[i,j] += tmp
```

To benchmark, first, initialize some arrays:

```
import numpy as np
N = 50
A = np.random.rand(N,N)
B = np.random.rand(N,N)
C = np.random.rand(N,N)
```

Python | Numba gufunc cpu | Numba gufunc gpu

```
%timeit matmul_cpu(A,B,C)
```

Note

Numba automatically did a lot of things for us:

- Memory was allocated on GPU
- Data was copied from CPU and GPU
- The kernel was configured and launched

- Data was copied back from GPU to CPU

Using ufuncs (or gfuncs) for GPU processing can be straightforward, but this approach may not always yield optimal performance due to automatic handling of data transfer to and from the GPU, as well as kernel launching. Additionally, in practice, not every function can be constructed as a ufunc.

To gain greater control and flexibility, one may need to craft their own kernels and manually manage data transfer. Refer to the *Python for HPDA* resource linked below for guidance on implementing such techniques using Numba.

Exercises

✍ Play around yourself

Are you a Julian or a Pythonista? Maybe neither, but take a pick between Python and Julia and play around with the code examples provided above.

You can find instructions for running Julia on LUMI and Python on LUMI / Google Colab in the Setup episode.

See also

- Introduction to programming in Julia (ENCCS)
- Julia for High-Performance Scientific Computing (ENCCS)
- Julia for high-performance data analytics (ENCCS)
- Introduction to running R, Python, Julia, and Matlab in HPC (NAISS-LUNARC-HPC2N-UPPMAX)
- High Performance Data Analytics in Python (ENCCS)
- Practical Intro to GPU Programming using Python (ENCCS)
- Using Python in an HPC environment (UPPMAX-HPC2N)
- Python for Scientific Computing (Aalto Scientific Computing)

Multiple GPU programming with MPI

❓ Questions

- What approach should be adopted to extend the synchronous OpenACC and OpenMP offloading models to utilise multiple GPUs across multiple nodes?

❗ Objectives

- To learn about combining MPI with either OpenACC or OpenMP offloading models.
- To learn about implementing GPU-awareness MPI approach.

Instructor note

- 30 min teaching
- 30 min exercises

Introduction

Exploring multiple GPUs (Graphics Processing Units) across distributed nodes offers the potential to fully leveraging the capacity of modern HPC (High-Performance Computing) systems at a large scale. Here one of the approaches to accelerate computing on distributed systems is to combine MPI (Message Passing Interface) with a GPU programming model such as OpenACC and OpenMP application programming interfaces (APIs). This combination is motivated by both the simplicity of these APIs, and the widespread use of MPI.

In this guide we provide readers, who are familiar with MPI, with insights on implementing a hybrid model in which the MPI communication framework is combined with either OpenACC or OpenMP APIs. A special focus will be on performing point-to-point (e.g. *MPI_Send* and *MPI_Recv*) and collective operations (e.g. *MPI_Allreduce*) from OpenACC and OpenMP APIs. Here we address two scenarios: (i) a scenario in which MPI operations are performed in the CPU-host followed by an offload to the GPU-device; and (ii) a scenario in which MPI operations are performed between a pair of GPUs without involving the CPU-host memory. The latter scenario is referred to as GPU-awareness MPI, and has the advantage of reducing the computing time caused by transferring data via the host-memory during heterogeneous communications, thus rendering HPC applications efficient.

This guide is organized as follows: we first introduce how to assign each MPI rank to a GPU device within the same node. We consider a situation in which the host and the device have a distinct memory. This is followed by a presentation on the hybrid MPI-OpenACC/OpenMP offloading with and without the GPU-awareness MPI. Exercises to help understanding these concepts are provided at the end.

Assigning MPI-ranks to GPU-devices

Accelerating MPI applications to utilise multiple GPUs on distributed nodes requires as a first step assigning each MPI rank to a GPU device, such that two MPI ranks do not use the same GPU device. This is necessarily in order to prevent the application from a potential crash. This is because GPUs are designed to handle multiple threading tasks, but not multiple MPI ranks.

One of the way to ensure that two MPI ranks do not use the same GPU, is to determine which MPI processes run on the same node, such that each process can be assigned to a GPU device within the same node. This can be done, for instance, by splitting the world communicator into sub-groups of communicators (or sub-communicators) using the routine *MPI_COMM_SPLIT_TYPE()*.

Splitting communicator in MPI (Fortran) | Splitting communicator in MPI (C++)

```
! Split the world communicator into subgroups of commu, each of which
! contains processes that run on the same node, and which can create a
! shared memory region (via the type MPI_COMM_TYPE_SHARED).
! The call returns a new communicator "host_comm", which is created by
! each subgroup.
call MPI_COMM_SPLIT_TYPE(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0,&
                         MPI_INFO_NULL, host_comm,ierr)
call MPI_COMM_RANK(host_comm, host_rank,ierr)
```

Here, the size of each sub-communicator corresponds to the number of GPUs per node (which is also the number of tasks per node), and each sub-communicator contains a list of processes indicated by a rank. These processes have a shared-memory region defined by the argument *MPI_COMM_TYPE_SHARED* (see the MPI report) for more details). Calling the routine *MPI_COMM_SPLIT_TYPE()* returns a sub-communicator labelled in the code above *"host_comm"*, and in which MPI-ranks are ranked from 0 to number of processes per node -1. These MPI ranks are in turn assigned to different GPU devices within the same node. This procedure is done according to which directive-based model is implemented. The retrieved MPI ranks are then stored in the variable **myDevice**. The variable is passed to an OpenACC or OpenMP routine as indicated in the code below.

Example: `Assign device`

Fortran OpenACC | Fortran OpenMP | C++ OpenMP

```
myDevice = host_rank

! Sets the device number and the device type to be used
call acc_set_device_num(myDevice, acc_get_device_type())

! Returns the number of devices available on the host
numDevice = acc_get_num_devices(acc_get_device_type())
```

Another useful function for retrieving the device number of a specific device, which is useful, e.g., to map data to a specific device is

OpenACC | OpenMP

```
acc_get_device_num()
```

The syntax of assigning MPI ranks to GPU devices is summarised below

Example: `Set device`

Fortran OpenACC | Fortran OpenMP | C++ OpenMP

```
! Initialise MPI communication
call MPI_Init(ierr)
! Identify the ID rank (process)
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
! Get number of active processes (from 0 to nproc-1)
call MPI_COMM_SIZE( MPI_COMM_WORLD, nproc, ierr )

! Split the world communicator into subgroups of commu, each of which
! contains processes that run on the same node, and which can create a
! shared memory region (via the type MPI_COMM_TYPE_SHARED).
! The call returns a new communicator "host_comm", which is created by
! each subgroup.
call MPI_COMM_SPLIT_TYPE(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0,&
                         MPI_INFO_NULL, host_comm,ierr)
call MPI_COMM_RANK(host_comm, host_rank,ierr)

! Gets the node name
call MPI_GET_PROCESSOR_NAME(name, resulten, ierror)

myDevice = host_rank

! Sets the device number and the device type to be used
call acc_set_device_num(myDevice, acc_get_device_type())

! Returns the number of devices available on the host
numDevice = acc_get_num_devices(acc_get_device_type())
```

Hybrid MPI-OpenACC/OpenMP without GPU-awareness approach

After covering how to assign each MPI-rank to a GPU device, we now address the concept of combining MPI with either OpenACC or OpenMP offloading. In this approach, calling an MPI routine from an OpenACC or OpenMP API requires updating the data in the CPU host before and after an MPI call. In this scenario, the data is copied back and forth between the host and the device before and after each MPI call. In the hybrid MPI-OpenACC model, the procedure is defined by specifying the directive *update host()* for copying the data from the device to the host before an MPI call; and by the directive *update device()* specified after an MPI call for copying the data back to the device. Similarly in the hybrid MPI-OpenMP. Here,

updating the data in the host can be done by specifying the OpenMP directives *update device() from()* and *update device() to()*, respectively, for copying the data from the device to the host and back to the device.

To illustrate the concept of the hybrid MPI-OpenACC/OpenMP, we show below an example of an implementation that involves the MPI functions *MPI_Send()* and *MPI_Recv()*.

Example: `Update host/device directives`

Fortran OpenACC | Fortran OpenMP | C++ OpenMP

```
    !offload f to GPUs
    !$acc enter data copyin(f)

    !update f: copy f from GPU to CPU
    !$acc update host(f)

    if(myid.lt.nproc-1) then
        call
MPI_Send(f(np:np),1,MPI_DOUBLE_PRECISION,myid+1,tag,MPI_COMM_WORLD, ierr)
    endif

    if(myid.gt.0) then
        call MPI_Recv(f(1),1,MPI_DOUBLE_PRECISION,myid-1,tag,MPI_COMM_WORLD,
status,ierr)
    endif

    !update f: copy f from CPU to GPU
    !$acc update device(f)
```

Here we present a code example that combines MPI with OpenACC/OpenMP API.

Example: `Update host/device directives`

Fortan OpenACC | Fortran OpenMP | C++ OpenMP

```
    call MPI_Scatter(f_send,np,MPI_DOUBLE_PRECISION,f,
np,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD, ierr)

    !offload f to GPUs
    !$acc enter data copyin(f)

    !update f: copy f from GPU to CPU
    !$acc update host(f)

    if(myid.lt.nproc-1) then
        call
MPI_Send(f(np:np),1,MPI_DOUBLE_PRECISION,myid+1,tag,MPI_COMM_WORLD, ierr)
    endif

    if(myid.gt.0) then
        call MPI_Recv(f(1),1,MPI_DOUBLE_PRECISION,myid-1,tag,MPI_COMM_WORLD,
status,ierr)
    endif

    !update f: copy f from CPU to GPU
    !$acc update device(f)

    !do something .e.g.
    !$acc kernels
    f = f/2.
    !$acc end kernels

    SumToT=0d0
    !$acc parallel loop reduction(+:SumToT)
    do k=1,np
        SumToT = SumToT + f(k)
    enddo
    !$acc end parallel loop

    !SumToT is by default copied back to the CPU
    call
MPI_ALLREDUCE(MPI_IN_PLACE,SumToT,1,MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD,i
)

    !$acc exit data delete(f)
```

Despite the simplicity of implementing the hybrid MPI-OpenACC/OpenMP offloading, it suffers from a low performance caused by an explicit transfer of data between the host and the device before and after calling an MPI routine. This constitutes a bottleneck in GPU-programming. To improve the performance affected by the host staging during the data transfer, one can implement the GPU-awareness MPI approach as described in the following section.

Hybrid MPI-OpenACC/OpenMP with GPU-awareness approach

The concept of the GPU-aware MPI enables an MPI library to directly access the GPU-device memory without necessarily using the CPU-host memory as an intermediate buffer (see e.g. OpenMPI documentation). This offers the benefit of transferring data from one GPU

to another GPU without the involvement of the CPU-host memory.

To be specific, in the GPU-awareness approach, the device pointers point to the data allocated in the GPU memory space (data should be present in the GPU device). Here, the pointers are passed as arguments to an MPI routine that is supported by the GPU memory. As MPI routines can directly access GPU memory, it offers the possibility of communicating between pairs of GPUs without transferring data back to the host.

In the hybrid MPI-OpenACC model, the concept is defined by combining the directive *host_data* together with the clause *use_device(list_array)*. This combination enables the access to the arrays listed in the clause *use_device(list_array)* from the host (see here). The list of arrays, which are already present in the GPU-device memory, are directly passed to an MPI routine without a need of a staging host-memory for copying the data. Note that for initially copying data to GPU, we use unstructured data blocks characterized by the directives *enter data* and *exit data*. The unstructured data has the advantage of allowing to allocate and deallocate arrays within a data region.

To illustrate the concept of the GPU-awareness MPI, we show below two examples that make use of point-to-point and collective operations from OpenACC and OpenMP APIs. In the first code example, the device pointer **f** is passed to the MPI functions *MPI_Send()* and *MPI_Recv()*; and in the second one, the pointer **SumToT** is passed to the MPI function *MPI_Allreduce*. Here, the MPI operations *MPI_Send* and *MPI_Recv* as well as *MPI_Allreduce* are performed between a pair of GPUs without passing through the CPU-host memory.

Example: `GPU-awareness: MPI_Send & MPI_Recv`

GPU-aware MPI with OpenACC (Fortran)

GPU-aware MPI with OpenMP (Fortran)

GPU-aware MPI with OpenMP (C++)

```
    !Device pointer f will be passed to MPI_send & MPI_recv
    !$acc host_data use_device(f)
    if(myid.lt.nproc-1) then
        call
MPI_Send(f(np:np),1,MPI_DOUBLE_PRECISION,myid+1,tag,MPI_COMM_WORLD, ierr)
    endif

    if(myid.gt.0) then
        call MPI_Recv(f(1),1,MPI_DOUBLE_PRECISION,myid-1,tag,MPI_COMM_WORLD,
status,ierr)
    endif
    !$acc end host_data
```

Example: `GPU-awareness: MPI_Allreduce`

GPU-aware MPI with OpenACC (Fortran)

GPU-aware MPI with OpenMP (Fortran)

GPU-aware MPI with OpenMP (C++)

```
    !$acc data copy(SumToT)
    !$acc host_data use_device(SumToT)
    call
MPI_ALLREDUCE(MPI_IN_PLACE,SumToT,1,MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD,i
)
    !$acc end host_data
    !$acc end data
```

We provide below a code example that illustrates the implementation of the MPI functions *MPI_Send()*, *MPI_Recv()* and *MPI_Allreduce()* within an OpenACC/OpenMP API. This implementation is specifically designed to support GPU-aware MPI operations.

Example: `GPU-awareness approach`

GPU-aware MPI with OpenACC (Fortran)

GPU-aware MPI with OpenMP (Fortran)

GPU-aware MPI with OpenMP (C++)

```
    call MPI_Scatter(f_send,np,MPI_DOUBLE_PRECISION,f,
np,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD, ierr)

    !offload f to GPUs
    !$acc enter data copyin(f)

    !Device pointer f will be passed to MPI_send & MPI_recv
    !$acc host_data use_device(f)
    if(myid.lt.nproc-1) then
        call
MPI_Send(f(np:np),1,MPI_DOUBLE_PRECISION,myid+1,tag,MPI_COMM_WORLD, ierr)
    endif

    if(myid.gt.0) then
        call MPI_Recv(f(1),1,MPI_DOUBLE_PRECISION,myid-1,tag,MPI_COMM_WORLD,
status,ierr)
    endif
    !$acc end host_data

    !do something .e.g.
    !$acc kernels
    f = f/2.
    !$acc end kernels

    SumToT=0d0
    !$acc parallel loop reduction(+:SumToT)
    do k=1,np
        SumToT = SumToT + f(k)
    enddo
    !$acc end parallel loop

    !SumToT is by default copied back to the CPU

    !$acc data copy(SumToT)
    !$acc host_data use_device(SumToT)
    call
MPI_ALLREDUCE(MPI_IN_PLACE,SumToT,1,MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD,i
)
    !$acc end host_data
    !$acc end data

    !$acc exit data delete(f)
```

The GPU-aware MPI with OpenACC/OpenMP APIs has the capability of directly communicating between a pair of GPUs within a single node. However, performing the GPU-to-GPU communication across multiple nodes requires the the GPUDirect RDMA (Remote Direct Memory Access) technology. This technology can further improve performance by reducing latency.

Compilation process

The compilation process of the hybrid MPI-OpenACC and MPI-OpenMP offloading is described below. This description is given for a Cray compiler of the wrapper *ftn*. On LUMI-G, the following modules may be necessary before compiling (see the LUMI documentation for further details about the available programming environments):

```
$ ml LUMI/24.03
$ ml PrgEnv-cray
$ ml cray-mpich
$ ml rocm
$ ml craype-accel-amd-gfx90a
```

Example: `Compilation process`

Compiling MPI-OpenACC (Fortran) | Compiling MPI-OpenMP (Fortran) | Compiling MPI-OpenMP (C++)

```
$ ftn -hacc -o mycode.mpiacc.exe mycode_mpiacc.f90
```

Here, the flags *hacc* and *homp* enable the OpenACC and OpenMP directives in the hybrid MPI-OpenACC and MPI-OpenMP applications, respectively.

Enabling GPU-aware support

To enable the GPU-aware support in MPICH library, one needs to set the following environment variable before running the application.

```
$ export MPICH_GPU_SUPPORT_ENABLED=1
```

Conclusion

In conclusion, we have presented an overview of a GPU-hybrid programming by integrating GPU-directive models, specifically OpenACC and OpenMP APIs, with the MPI library. The approach adopted here allows us to utilise multiple GPU-devices not only within a single node but it extends to distributed nodes. In particular, we have addressed GPU-aware MPI

approach, which has the advantage of enabling a direct interaction between an MPI library and a GPU-device memory. In other words, it permits performing MPI operations between a pair of GPUs, thus reducing the computing time caused by the data locality.

Exercises

We consider an MPI fortran code that solves a 2D-Laplace equation, and which is partially accelerated. The focus of the exercises is to complete the acceleration using either OpenACC or OpenMP API by following these steps.

❗ Access exercise material

Code examples for the exercises below can be accessed in the *content/examples/exercise_multipleGPU* subdirectory of this repository. To access them, you need to clone the repository:

```
$ git clone https://github.com/ENCCS/gpu-programming.git
$ cd gpu-programming/content/examples/exercise_multipleGPU
$ ls
```

✍ Exercise I: Set a GPU device

1. Implement OpenACC/OpenMP functions that enable assigning each MPI rank to a GPU device.

1.1 Compile and run the code on multiple GPUs.

✍ Exercise II: Apply traditional MPI-OpenACC/OpenMP

2.1 Incorporate the OpenACC directives **update host()** and **update device()** before and after calling an MPI function, respectively.

❗ Note

The OpenACC directive **update host()** is used to transfer data from GPU to CPU within a data region; while the directive **update device()** is used to transfer the data from CPU to GPU.

2.2 Incorporate the OpenMP directives **update device() from()** and **update device() to()** before and after calling an MPI function, respectively.

❗ Note

The OpenMP directive *update device() from()* is used to transfer data from GPU to CPU within a data region; while the directive *update device() to()* is used to transfer the data from CPU to GPU.

2.3 Compile and run the code on multiple GPUs.

✍ Exercise III: Implement GPU-aware support

3.1 Incorporate the OpenACC directive *host_data use_device()* to pass a device pointer to an MPI function.

3.2 Incorporate the OpenMP directive *data use_device_ptr()* to pass a device pointer to an MPI function.

3.3 Compile and run the code on multiple GPUs.

✍ Exercise IV: Evaluate the performance

1. Evaluate the execution time of the accelerated codes in the exercises **II** and **III**, and compare it with that of a pure MPI implementation.

See also

- GPU-aware MPI.
- MPI documentation.
- OpenACC specification.
- OpenMP specification.
- LUMI documentation.
- OpenACC vs OpenMP offloading.
- OpenACC course.

Preparing code for GPU porting

❓ Questions

- What are the key steps involved in porting code to take advantage of GPU parallel processing capability?
- How can I identify the computationally intensive parts of my code that can benefit from GPU acceleration?
- What are the considerations for refactoring loops to suit the GPU architecture and improve memory access patterns?
- Are there any tools that can translate automatically between different frameworks?

❗ Objectives

- Getting familiarized the steps involved in porting code to GPUs to take advantage of parallel processing capabilities.
- Giving some idea about refactoring loops and modifying operations to suit the GPU architecture and improve memory access patterns.
- Learn to use automatic translation tools to port from CUDA to HIP and from OpenACC to OpenMP

Instructor note

- 30 min teaching
- 20 min exercises

Porting from CPU to GPU

When porting code to take advantage of the parallel processing capability of GPUs, several steps need to be followed and some additional work is required before writing actual parallel code to be executed on the GPUs:

- **Identify Targeted Parts**: Begin by identifying the parts of the code that contribute significantly to the execution time. These are often computationally intensive sections such as loops or matrix operations. The Pareto principle suggests that roughly 10-20% of the code accounts for 80-90% of the execution time.
- **Equivalent GPU Libraries**: If the original code uses CPU libraries like BLAS, FFT, etc, it's crucial to identify the equivalent GPU libraries. For example, *cuBLAS* or *hipBLAS* can replace CPU-based BLAS libraries. Utilizing GPU-specific libraries ensures efficient GPU utilization.
- **Refactor Loops**: When porting loops directly to GPUs, some refactoring is necessary to suit the GPU architecture. This typically involves splitting the loop into multiple steps or modifying operations to exploit the independence between iterations and improve memory access patterns. Each step of the original loop can be mapped to a kernel, executed by multiple GPU threads, with each thread corresponding to an iteration.
- **Memory Access Optimization**: Consider the memory access patterns in the code. GPUs perform best when memory access is coalesced and aligned. Minimizing global memory accesses and maximizing utilization of shared memory or registers can significantly enhance performance. Review the code to ensure optimal memory access for GPU execution.

Discussion

✍ How would this be ported? (n_soap ≈ 100, n_sites ⩾ 10000, k_max ≈ 20*n_sites)

Inspect the following Fortran code (if you don’t read Fortran: do-loops == for-loops)

```
k2 = 0
do i = 1, n_sites
  do j = 1, n_neigh(i)
    k2 = k2 + 1
    counter = 0
    counter2 = 0
    do n = 1, n_max
      do np = n, n_max
        do l = 0, l_max
          if( skip_soap_component(l, np, n) )cycle

          counter = counter+1
          do m = 0, l
            k = 1 + l*(l+1)/2 + m
            counter2 = counter2 + 1
            multiplicity = multiplicity_array(counter2)
            soap_rad_der(counter, k2) = soap_rad_der(counter, k2) +
multiplicity * real( cnk_rad_der(k, n, k2) * conjg(cnk(k, np, i)) + cnk(k, n,
i) * conjg(cnk_rad_der(k, np, k2)) )
            soap_azi_der(counter, k2) = soap_azi_der(counter, k2) +
multiplicity * real( cnk_azi_der(k, n, k2) * conjg(cnk(k, np, i)) + cnk(k, n,
i) * conjg(cnk_azi_der(k, np, k2)) )
            soap_pol_der(counter, k2) = soap_pol_der(counter, k2) +
multiplicity * real( cnk_pol_der(k, n, k2) * conjg(cnk(k, np, i)) + cnk(k, n,
i) * conjg(cnk_pol_der(k, np, k2)) )
          end do
        end do
      end do
    end do

    soap_rad_der(1:n_soap, k2) = soap_rad_der(1:n_soap, k2) / sqrt_dot_p(i) -
soap(1:n_soap, i) / sqrt_dot_p(i)**3 * dot_product( soap(1:n_soap, i),
soap_rad_der(1:n_soap, k2) )
    soap_azi_der(1:n_soap, k2) = soap_azi_der(1:n_soap, k2) / sqrt_dot_p(i) -
soap(1:n_soap, i) / sqrt_dot_p(i)**3 * dot_product( soap(1:n_soap, i),
soap_azi_der(1:n_soap, k2) )
    soap_pol_der(1:n_soap, k2) = soap_pol_der(1:n_soap, k2) / sqrt_dot_p(i) -
soap(1:n_soap, i) / sqrt_dot_p(i)**3 * dot_product( soap(1:n_soap, i),
soap_pol_der(1:n_soap, k2) )

    if( j == 1 )then
      k3 = k2
    else
      soap_cart_der(1, 1:n_soap, k2) = dsin(thetas(k2)) * dcos(phis(k2)) *
soap_rad_der(1:n_soap, k2) - dcos(thetas(k2)) * dcos(phis(k2)) / rjs(k2) *
soap_pol_der(1:n_soap, k2) - dsin(phis(k2)) / rjs(k2) *
soap_azi_der(1:n_soap, k2)
      soap_cart_der(2, 1:n_soap, k2) = dsin(thetas(k2)) * dsin(phis(k2)) *
soap_rad_der(1:n_soap, k2) - dcos(thetas(k2)) * dsin(phis(k2)) / rjs(k2) *
soap_pol_der(1:n_soap, k2) + dcos(phis(k2)) / rjs(k2) *
soap_azi_der(1:n_soap, k2)
      soap_cart_der(3, 1:n_soap, k2) = dcos(thetas(k2)) *
soap_rad_der(1:n_soap, k2) + dsin(thetas(k2)) / rjs(k2) *
soap_pol_der(1:n_soap, k2)
      soap_cart_der(1, 1:n_soap, k3) = soap_cart_der(1, 1:n_soap, k3) -
soap_cart_der(1, 1:n_soap, k2)
      soap_cart_der(2, 1:n_soap, k3) = soap_cart_der(2, 1:n_soap, k3) -
soap_cart_der(2, 1:n_soap, k2)
      soap_cart_der(3, 1:n_soap, k3) = soap_cart_der(3, 1:n_soap, k3) -
soap_cart_der(3, 1:n_soap, k2)
    end if
  end do
end do
```

Some steps at first glance:

- the code could (has to) be splitted in 3-4 kernels. Why?
- check if there are any variables that could lead to false dependencies between iterations, like the index *k2*
- is it efficient for GPUs to split the work over the index *i*? What about the memory access? Note the arrays are *2D* in Fortran
- is it possible to collapse some loops? Combining nested loops can reduce overhead and improve memory access patterns, leading to better GPU performance.
- what is the best memory access in a GPU? Review memory access patterns in the code. Minimize global memory access by utilizing shared memory or registers where appropriate. Ensure memory access is coalesced and aligned, maximizing GPU memory throughput

Refactored code!

- Registers are limited and the larger the kernel use more registers registers resulting in less active threads (small occupancy).
- In order to compute *soap_rad_der(is,k2)* the CUDA thread needs access to all the previous values *soap_rad_der(1:nsoap,k2)*.
- In order to compute *soap_cart_der(1, 1:n_soap, k3)* it is required to have access to all values *(k3+1:k2+n_neigh(i))*.
- Note the indices in the first part. The matrices are transposed for better access patterns.

```
  !omp target teams distribute parallel do private (i)
  do k2 = 1, k2_max
    i=list_of_i(k2)
    counter = 0
    counter2 = 0
    do n = 1, n_max
      do np = n, n_max
        do l = 0, l_max
          if( skip_soap_component(l, np, n) ) then
            cycle
          endif
          counter = counter+1
          do m = 0, l
            k = 1 + l*(l+1)/2 + m
            counter2 = counter2 + 1
            multiplicity = multiplicity_array(counter2)
            tsoap_rad_der(k2,counter) = tsoap_rad_der(k2,counter) + multiplicity
* real( tcnk_rad_der(k2,k,n) * conjg(tcnk(i,k,np)) + tcnk(i,k,n) *
conjg(tcnk_rad_der(k2,k,np)) )
            tsoap_azi_der(k2,counter) = tsoap_azi_der(k2,counter) + multiplicity
* real( tcnk_azi_der(k2,k,n) * conjg(tcnk(i,k,np)) + tcnk(i,k,n) *
conjg(tcnk_azi_der(k2,k,np)) )
            tsoap_pol_der(k2,counter) = tsoap_pol_der(k2,counter) + multiplicity
* real( tcnk_pol_der(k2,k,n) * conjg(tcnk(i,k,np)) + tcnk(i,k,n) *
conjg(tcnk_pol_der(k2,k,np)) )
          end do
        end do
      end do
    end do
  end do

! Before the next part the variables are transposed again to their original
layout.

 !omp target teams  distribute private(i)
 do k2 = 1, k2_max
   i=list_of_i(k2)
   locdot=0.d0

   !omp parallel do reduction(+:locdot_rad_der,locdot_azi_der,locdot_pol_der)
   do is=1,nsoap
     locdot_rad_der=locdot_rad_der+soap(is, i) * soap_rad_der(is, k2)
     locdot_azi_der=locdot_azi_der+soap(is, i) * soap_azi_der(is, k2)
     locdot_pol_der=locdot_pol_der+soap(is, i) * soap_pol_der(is, k2)
   enddo
   dot_soap_rad_der(k2)= locdot_rad_der
   dot_soap_azi_der(k2)= locdot_azi_der
   dot_soap_pol_der(k2)= locdot_pol_der
 end do

 !omp target teams distribute
 do k2 = 1, k2_max
   i=list_of_i(k2)

   !omp parallel do
   do is=1,nsoap
     soap_rad_der(is, k2) = soap_rad_der(is, k2) / sqrt_dot_p(i) -   soap(is, i)
/ sqrt_dot_p(i)**3 * dot_soap_rad_der(k2)
     soap_azi_der(is, k2) = soap_azi_der(is, k2) / sqrt_dot_p(i) -   soap(is, i)
/ sqrt_dot_p(i)**3 * dot_soap_azi_der(k2)
     soap_pol_der(is, k2) = soap_pol_der(is, k2) / sqrt_dot_p(i) -   soap(is, i)
/ sqrt_dot_p(i)**3 * dot_soap_pol_der(k2)
   end do
```

```
 end do

 !omp teams distribute private(k3)
 do k2 = 1, k2_max
   k3=list_k2k3(k2)

   !omp parallel do private (is)
   do is=1,n_soap
     if( k3 /= k2)then
       soap_cart_der(1, is, k2) = dsin(thetas(k2)) * dcos(phis(k2)) *
soap_rad_der(1:n_soap, k2) - dcos(thetas(k2)) * dcos(phis(k2)) / rjs(k2) *
soap_pol_der(1:n_soap, k2) - dsin(phis(k2)) / rjs(k2) * soap_azi_der(is, k2)
       soap_cart_der(2, is, k2) = dsin(thetas(k2)) * dsin(phis(k2)) *
soap_rad_der(1:n_soap, k2) - dcos(thetas(k2)) * dsin(phis(k2)) / rjs(k2) *
soap_pol_der(1:n_soap, k2) + dcos(phis(k2)) / rjs(k2) * soap_azi_der(is, k2)
       soap_cart_der(3, is, k2) = dcos(thetas(k2)) * soap_rad_der(is, k2) +
dsin(thetas(k2)) / rjs(k2) * soap_pol_der(is, k2)
     end if
   end do
 end do

 !omp teams distribute private(k3)
 do i = 1, n_sites
   k3=list_k3(i)

   !omp parallel do private(is, k2)
   do is=1,n_soap
     do k2=k3+1,k3+n_neigh(i)
       soap_cart_der(1, is, k3) = soap_cart_der(1, is, k3) - soap_cart_der(1,
is, k2)
       soap_cart_der(2, is, k3) = soap_cart_der(2, is, k3) - soap_cart_der(2,
is, k2)
       soap_cart_der(3, is, k3) = soap_cart_der(3, is, k3) - soap_cart_der(3,
is, k2)
     end do
   end do
 end do
```

Keypoints

- Identify equivalent GPU libraries for CPU-based libraries and utilizing them to ensure efficient GPU utilization.
- Importance of identifying the computationally intensive parts of the code that contribute significantly to the execution time.
- The need to refactor loops to suit the GPU architecture.
- Significance of memory access optimization for efficient GPU execution, including coalesced and aligned memory access patterns.

Porting between different GPU frameworks

You might also find yourself in a situation where you need to port a code from one particular GPU framework to another. This section gives an overview of different tools that enable converting CUDA and OpenACC codes to HIP and OpenMP, respectively. This conversion

process enables an application to target various GPU architectures, specifically, NVIDIA and AMD GPUs. Here we focus on hipify and clacc tools. This guide is adapted from the NRIS documentation.

Translating CUDA to HIP with Hipify

In this section, we cover the use of `hipify-perl` and `hipify-clang` tools to translate a CUDA code to HIP.

Hipify-perl

The `hipify-perl` tool is a script based on perl that translates CUDA syntax into HIP syntax (see .e.g. here for more details). For instance, in a CUDA code that incorporates the CUDA functions `cudaMalloc`` and `cudaDeviceSynchronize`, the tool will substitute `cudaMalloc` with the HIP function `hipMalloc`. Similarly the CUDA function `cudaDeviceSynchronize` will be substituted with the HIP function `hipDeviceSynchronize`. We list below the basic steps to run `hipify-perl` on LUMI-G.

- **Step 1**: Generating `hipify-perl` script

```
$ module load rocm/6.0.3
$ hipify-clang --perl
```

- **Step 2**: Running the generated `hipify-perl`

```
$ hipify-perl program.cu > program.cu.hip
```

- **Step 3**: Compiling with `hipcc` the generated HIP code

```
$ hipcc --offload-arch=gfx90a -o program.hip.exe program.cu.hip
```

Despite the simplicity of the use of `hipify-perl`, the tool might not be suitable for large applications, as it relies heavily on substituting CUDA strings with HIP strings (e.g. it substitutes `*cuda*` with `*hip*`). In addition, `hipify-perl` lacks the ability of distinguishing device/host function calls. The alternative here is to use the `hipify-clang` tool as will be described in the next section.

Hipify-clang

As described in the HIPIFY documentation, the `hipify-clang` tool is based on clang for translating CUDA sources into HIP sources. The tool is more robust for translating CUDA codes compared to the `hipify-perl` tool. Furthermore, it facilitates the analysis of the code by providing assistance.

In short, `hipify-clang` requires `LLVM+CLANG` and `CUDA`. Details about building `hipify-clang` can be found here. Note that `hipify-clang` is available on LUMI-G. The issue however might be related to the installation of CUDA-toolkit. To avoid any eventual issues with the installation procedure we opt for CUDA singularity container. Here we present a step-by-step guide for running `hipify-clang`:

- **Step 1**: Pulling a CUDA singularity container e.g.

```
$ singularity pull docker://nvcr.io/nvidia/cuda:11.4.3-devel-ubuntu20.04
```

- **Step 2**: Loading a rocm module and launching the CUDA singularity

```
$ module load rocm/6.0.3
$ singularity shell -B $PWD,/opt:/opt cuda_11.4.0-devel-ubuntu20.04.sif
```

  where the current directory `$PWD` in the host is mounted to that of the container, and the directory `/opt` in the host is mounted to the that inside the container.

- **Step 3**: Setting the environment variable `$PATH`. In order to run `hipify-clang` from inside the container, one can set the environment variable `$PATH` that defines the path to look for the binary `hipify-clang`.

```
$ export PATH=/opt/rocm-6.0.3/bin:$PATH
```

  Note that the rocm version we used is `rocm-6.0.3`.

- **Step 4**: Running `hipify-clang` from inside the singularity container

```
$ hipify-clang program.cu -o hip_program.cu.hip --cuda-path=/usr/local/cuda-11.4 -I
/usr/local/cuda-11.4/include
```

  Here the cuda path and the path to the `*includes*` and `*defines*` files should be specified. The CUDA source code and the generated output code are *program.cu* and *hip_program.cu.hip*, respectively.

  The syntax for the compilation process of the generated hip code is similar to the one described in the previous section (see the **Step 3** in the hipify-perl section).

Code examples for the `Hipify` exercises can be accessed in the *content/examples/exercise_hipify* subdirectory by cloning this repository:

```
$ git clone https://github.com/ENCCS/gpu-programming.git
$ cd gpu-programming/content/examples/exercise_hipify
$ ls
```

✍ Exercise I : Translate an CUDA code to HIP with `hipify-perl`

1.1 Generate the `hipify-perl` tool.

1.2 Convert the CUDA code `vec_add_cuda.cu` located in `/exercise_hipify/Hipify_perl` with the `Hipify-perl` tool to HIP.

1.3 Compile the generated HIP code with the `hipcc` compiler wrapper and run it.

✍ Exercise II : Translate an CUDA code to HIP with `hipify-clang`

2.1 Convert the CUDA code `vec_add_cuda.cu` located in `/exercise_hipify/Hipify_clang` with the `Hipify-clang` tool to HIP.

2.2 Compile the generated HIP code with the `hipcc` compiler wrapper and run it.

Translating OpenACC to OpenMP with Clacc

Clacc is a tool to translate an OpenACC application to OpenMP offloading with the Clang/LLVM compiler environment. Note that the tool is specific to OpenACC C, while OpenACC Fortran is already supported on AMD GPU. As indicated in the GitHub repository the compiler `Clacc` is the `Clang`'s executable in the subdirectory `\bin` of the `\install` directory as described below.

In the following we present a step-by-step guide for building and using *Clacc*:

- **Step 1**: Building and installing Clacc.

```
$ git clone -b clacc/main https://github.com/llvm-doe-org/llvm-project.git
$ cd llvm-project
$ mkdir build && cd build
$ cmake -DCMAKE_INSTALL_PREFIX=../install     \
   -DCMAKE_BUILD_TYPE=Release            \
   -DLLVM_ENABLE_PROJECTS="clang;lld"    \
   -DLLVM_ENABLE_RUNTIMES=openmp         \
   -DLLVM_TARGETS_TO_BUILD="host;AMDGPU" \
   -DCMAKE_C_COMPILER=gcc                \
   -DCMAKE_CXX_COMPILER=g++              \
   ../llvm
$ make
$ make install
```

- **Step 2**: Setting up environment variables to be able to work from the `/install` directory, which is the simplest way. We assume that the `/install` directory is located in the path `/project/project_xxxxxx/Clacc/llvm-project`.

For more advanced usage, which includes for instance modifying `Clacc`, we refer readers to "Usage from Build directory"

```
$ export PATH=/scratch/project_465002387/clacc/llvm-project/install/bin:$PATH
$ export LD_LIBRARY_PATH=/scratch/project_465002387/clacc/llvm-
project/install/lib:$LD_LIBRARY_PATH
```

- **Step 3**: Source to source conversion of the *openACC_code.c* code to be printed out to the file *openMP_code.c*:

```
$ clang -fopenacc-print=omp -fopenacc-structured-ref-count-omp=no-ompx-hold
openACC_code.c > openMP_code.c
```

  Here the flag `-fopenacc-structured-ref-count-omp=no-ompx-hold` is introduced to disable the `ompx_hold` map type modifier, which is used by the OpenACC `copy` clause translation. The `ompx_hold` is an OpenMP extension that might not be supported yet by other compilers.

- **Step 4** Compiling the code with the cc compiler wrapper

```
module load CrayEnv
module load PrgEnv-cray
module load craype-accel-amd-gfx90a
module load rocm/6.0.3

cc -fopenmp -o executable openMP_code.c
```

Access exercise material

Code examples for the `Clacc` exercise can be accessed in the *content/examples/exercise_clacc* subdirectory by cloning this repository:

```
$ git clone https://github.com/ENCCS/gpu-programming.git
$ cd gpu-programming/content/examples/exercise_clacc
$ ls
```

✍ Exercise : Translate an OpenACC code to OpenMP

1. Convert the OpenACC code `openACC_code.c` located in `/exercise_clacc` with the `Clacc` compiler.
2. Compile the generated OpenMP code with the `cc` compiler wrapper and run it.

Translating CUDA to SYCL/DPC++ with SYCLomatic

Intel offers a tool for CUDA-to-SYCL code migration, included in the Intel oneAPI Basekit.

It is not installed on LUMI, but the general workflow is similar to the HIPify Clang and also requires an existing CUDA installation:

```
$ dpct program.cu
$ cd dpct_output/
$ icpx -fsycl program.dp.cpp
```

SYCLomatic can migrate larger projects by using `-in-root` and `-out-root` flags to process directories recursively. It can also use compilation database (supported by CMake and other build systems) to deal with more complex project layouts.

Please note that the code generated by SYCLomatic relies on oneAPI-specific extensions, and thus cannot be directly used with other SYCL implementations, such as AdaptiveCpp (hipSYCL). The `--no-incremental-migration` flag can be added to `dpct` command to minimize, but not completely avoid, the use of this compatibility layer. That would require manual effort, since some CUDA concepts cannot be directly mapped to SYCL.

Additionally, CUDA applications might assume certain hardware behavior, such as 32-wide warps. If the target hardware is different (e.g., AMD MI250 GPUs, used in LUMI, have warp size of 64), the algorithms might need to be adjusted manually.

Conclusion

This concludes a brief overview of the usage of available tools to convert CUDA codes to HIP and SYCL, and OpenACC codes to OpenMP offloading. In general the translation process for large applications might be incomplete and thus requires manual modification to complete the porting process. It is however worth noting that the accuracy of the translation process requires that applications are written correctly according to the CUDA and OpenACC syntaxes.

See also

- Hipify GitHub
- HIPify Reference Guide v5.1
- HIP example
- Porting CUDA to HIP
- Clacc Main repository README
- SYCLomatic main mage
- SYCLomatic documentation

Keypoints

- Useful tools exist to automatically translate tools from CUDA to HIP and SYCL and from OpenACC to OpenMP, but they may require manual modifications.

Recommendations

❓ Questions

- Which GPU programming framework is right for me and my project?

Instructor note

- 30 min teaching
- 15 min exercises

Portability

One of the critical factors when diving into GPU programming is the portability of the chosen framework. It's crucial to ensure that the framework you decide to utilize is compatible with the GPU or GPUs you intend to use. This might seem like a basic step, but it's essential to avoid unnecessary hardware-software mismatches that could lead to performance bottlenecks or, worse, a complete failure of the system.

Moreover, if you're targeting multiple platforms or GPUs, it's wise to consider using frameworks that support portable kernel-based models or those that come with high-level language support. The benefit of these frameworks is that they allow for efficient execution of your code on a variety of hardware configurations without needing significant alterations.

Programming Effort

The amount of programming effort required is another factor to consider when choosing a GPU programming framework. It's advisable to select a framework that supports the programming language you're comfortable with. This consideration will ensure a smoother learning curve and a more efficient development process.

Furthermore, it's important to check the availability of supportive resources for the chosen framework. Comprehensive documentation, illustrative examples, and an active community are important when learning a new framework or troubleshooting issues. They not only minimize the time spent on resolving bugs but also foster continuous learning and mastery of the framework.

Performance Requirements

Every application or project has unique performance requirements. Therefore, it's crucial to evaluate the performance characteristics and optimization capabilities of the potential frameworks before choosing one. Some frameworks offer extensive optimization features and can automatically tune your code to maximize its performance. Knowing how well a framework can handle your specific workload requirements can save you considerable time and resources in the long run.

Cost-benefit Analysis

Before finalizing your choice of a GPU programming framework, it's recommended to perform a cost-benefit analysis. Consider the specific requirements of your project, like the processing power needed, the complexity of the tasks, the amount of data to be processed, and the cost associated with the potential framework. Understanding these factors will help you determine the most suitable and cost-effective framework for your needs.

Choosing Between Frameworks

The decision of choosing between different GPU programming frameworks often depends on several factors, including:

- **The specifics of the problem**: Different problems might need different computational capabilities. Understand your problem thoroughly and evaluate which framework is best equipped to handle it.
- **Starting point**: If you're starting from scratch, you might have more flexibility in choosing your framework than if you're building on top of existing code.
- **Background knowledge of the programmer**: The familiarity of the programmer with certain programming languages or frameworks plays a big role in the decision-making process.
- **Time investment**: Some frameworks may have a steeper learning curve but offer more extensive capabilities, while others might be easier to grasp but provide limited features.

- **Performance needs**: Some applications require maximum computational power, while others might not. The performance capabilities of the framework should align with the needs of your project.

By keeping these considerations in mind, you can make a more informed decision and choose a GPU programming framework that best suits your needs.

💬 Question and discussion time

- Has your mental model of how GPUs work and how they are programmed changed?
- Do you have a better idea about what framework is right for your code?
- What questions do you have? Ask us anything!

Example: putting it all together

❓ Questions

- How do I compile and run code developed using different programming models and frameworks?
- What can I expect from the GPU-ported programs in terms of performance gains / trends and how do I estimate this?

❗ Objectives

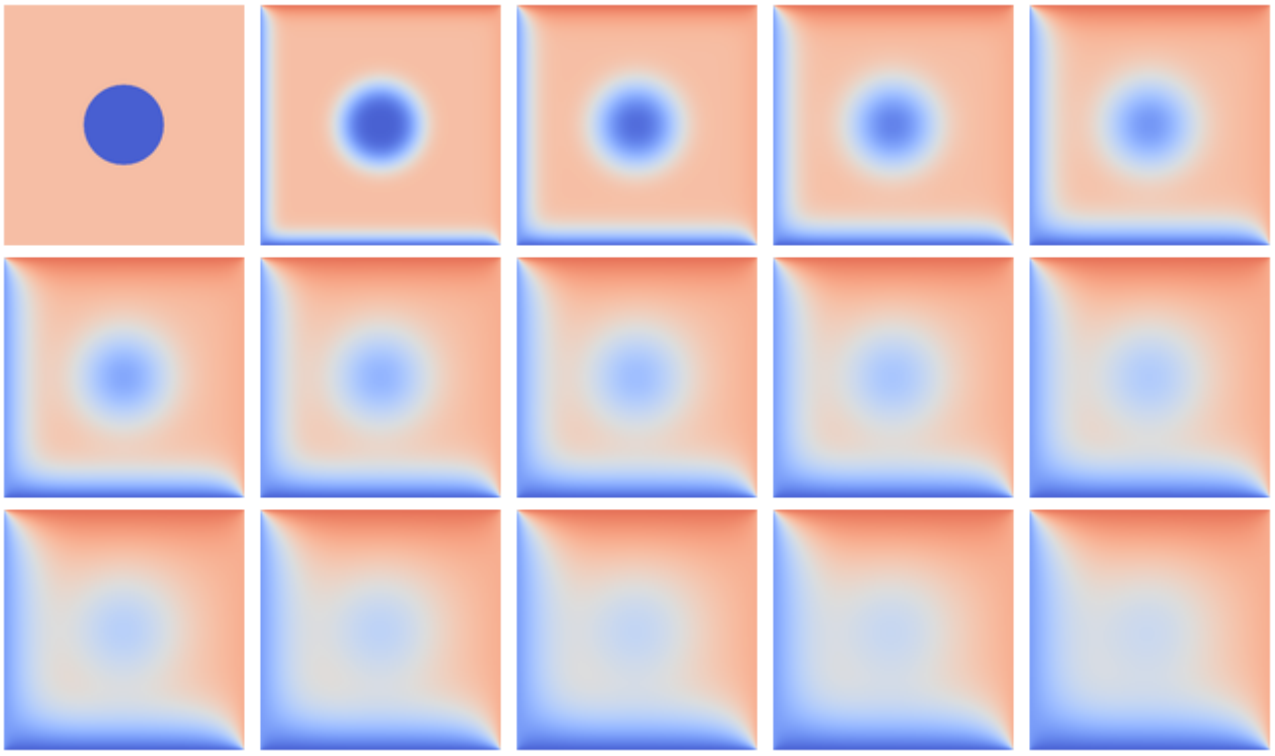
- To show a self-contained example of parallel computation executed on CPU and GPU using different programming models
- To show differences and consequences of implementing the same algorithm in natural "style" of different models/ frameworks
- To discuss how to assess theoretical and practical performance scaling of GPU codes

Instructor note

- 35 min teaching
- 30 min exercises

Problem: heat flow in two-dimensional area

Heat flows in objects according to local temperature differences, as if seeking local equilibrium. The following example defines a rectangular area with two always-warm sides (temperature 70 and 85), two cold sides (temperature 20 and 5) and a cold disk at the center. Because of heat diffusion, temperature of neighboring patches of the area is bound to equalize, changing the overall distribution:



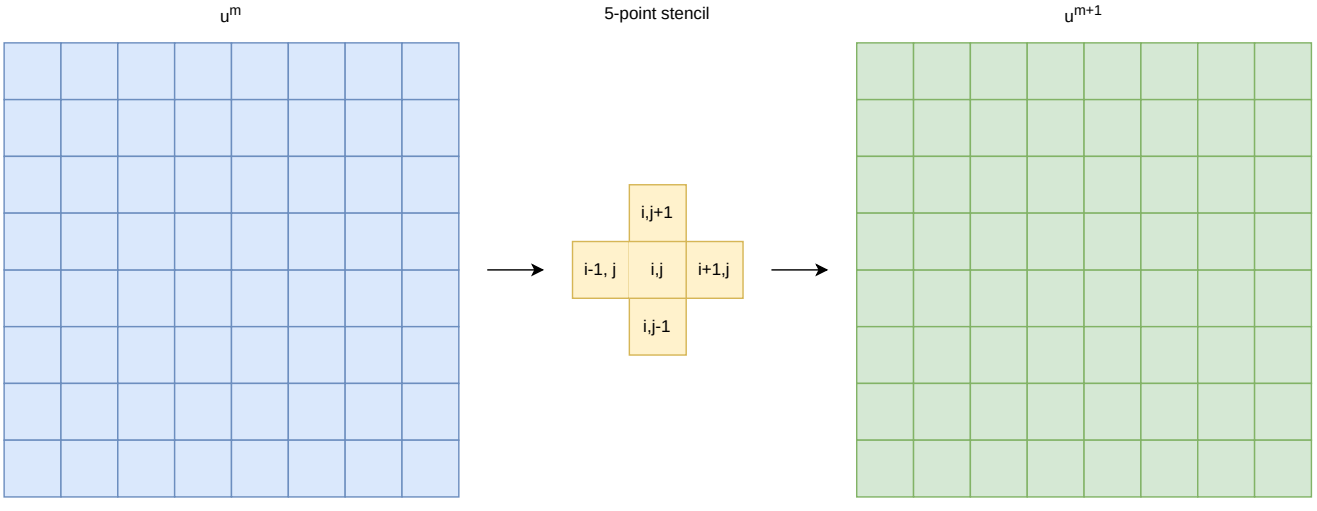
Over time, the temperature distribution progresses from the initial state toward an end state where upper triangle is warm and lower is cold. The average temperature tends to (70 + 85 + 20 + 5) / 4 = 45.

Technique: stencil computation

Heat transfer in the system above is governed by the partial differential equation(s) describing local variation of the temperature field in time and space. That is, the rate of change of the temperature field $u(x, y, t)$ over two spatial dimensions x and y and time t (with rate coefficient α) can be modelled via the equation

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

The standard way to numerically solve differential equations is to *discretize* them, i. e. to consider only a set/ grid of specific area points at specific moments in time. That way, partial derivatives $({\partial u}$ are converted into differences between adjacent grid points $(u^{m}(i,j)$, with m, i, j denoting time and spatial grid points, respectively. Temperature change in time at a certain point can now be computed from the values of neighboring points at earlier time; the same expression, called *stencil*, is applied to every point on the grid.



This simplified model uses an 8x8 grid of data in light blue in state \(m\), each location of which has to be updated based on the indicated 5-point stencil in yellow to move to the next time point \(m+1\).

✍ Question: stencil applications

Stencil computation is a common occurrence in solving numerical problems. Have you already encountered it? Can you think of a problem that could be formulated this way in your field / area of expertise?

✔ Solution

One obvious choice is *convolution* operation, used in image processing to apply various filter kernels; in some contexts, “convolution” and “stencil” are used almost interchangeably. Other related use is for averaging/ pooling adjacent values.

Technical considerations

1. How fast and/ or accurate can the solution be?

Spatial resolution of the temperature field is controlled by the number/ density of the grid points. As the full grid update is required to proceed from one time point to the next, stencil computation is the main target of parallelization (on CPU or GPU).

Moreover, in many cases the chosen time step cannot be arbitrarily large, otherwise the numerical differentiation will fail, and dense/ accurate grids imply small time steps (see inset below), which makes efficient spatial update even more important.

✔ Optional: stencil expression and time-step limit

Differential equation shown above can be discretized using different schemes. For this example, temperature values at each grid point \(u^{m}(i,j)\) are updated from one time point (\(m\)) to the next (\(m+1\)), using the following expressions:

$$u^{m+1}(i,j) = u^m(i,j) + \Delta t \alpha \nabla^2 u^m(i,j) ,$$

where

$$\begin{split}\nabla^2 u &= \frac{u(i-1,j)-2u(i,j)+u(i+1,j)}{(\Delta x)^2} \\ &+ \frac{u(i,j-1)-2u(i,j)+u(i,j+1)}{(\Delta y)^2} ,\end{split}$$

and \(\Delta x\), \(\Delta y\), \(\Delta t\) are step sizes in space and time, respectively.

Time-update schemes often have a limit on the maximum allowed time step \(\Delta t\). For the current scheme, it is equal to

$$\Delta t_{max} = \frac{(\Delta x)^2 (\Delta y)^2}{2 \alpha ((\Delta x)^2 + (\Delta y)^2)}$$

2. What to do with area boundaries?

Naturally, stencil expression can't be applied directly to the outermost grid points that have no outer neighbors. This can be solved by either changing the expression for those points or by adding an additional layer of grid that is used in computing update, but not updated itself – points of fixed temperature for the sides are being used in this example.

3. How could the algorithm be optimized further?

In an earlier episode, importance of efficient memory access was already stressed. In the following examples, each grid point (and its neighbors) is treated mostly independently; however, this also means that for 5-point stencil each value of the grid point may be read up to 5 times from memory (even if it's the fast GPU memory). By rearranging the order of mathematical operations, it may be possible to reuse these values in a more efficient way.

Another point to note is that even if the solution is propagated in small time steps, not every step might actually be needed for output. Once some *local* region of the field is updated, mathematically nothing prevents it from being updated for the second time step – even if the rest of the field is still being recalculated – as long as \(t = m-1\) values for the region boundary are there when needed. (Of course, this is more complicated to implement and would only give benefits in certain cases.)

The following table will aid you in navigating the rest of this section:

Episode guide

- Sequential and OpenMP-threaded code in C++, including compilation/ running instructions
- Naive GPU parallelization, including SYCL compilation instructions
- GPU code with device data management (OpenMP, SYCL)
- Python implementation, including running instructions on Google Colab
- Julia implementation, including running instructions

Sequential and thread-parallel program in C++

Trying out code examples

Source files of the examples presented for the rest of this episode are available in the content/examples/stencil/ directory. To download them to your preferred directory on the cluster (f.e. `/scratch/project_<#>/<your_folder>/`), you can use Git:

```
$ git clone https://github.com/ENCCS/gpu-programming.git
$ cd gpu-programming/content/examples/stencil/
$ ls
```

Warning

Don't forget to `git pull` for the latest updates if you already have the content from the first day of the workshop!

If we assume the grid point values to be truly independent *for a single time step*, stencil application procedure may be straightforwardly written as a loop over the grid points, as shown below in tab "Stencil update". (General structure of the program and the default parameter values for the problem model are also provided for reference.) CPU-thread parallelism can then be enabled by a single OpenMP `#pragma`:

stencil/base/

Stencil update | Main function | Default params

core.cpp

```
// (c) 2023 ENCCS, CSC and the contributors
#include "heat.h"

// Update the temperature values using five-point stencil
// Arguments:
//   curr: current temperature values
//   prev: temperature values from previous time step
//   a: diffusivity
//   dt: time step
void evolve(field *curr, field *prev, double a, double dt)
{
  // Help the compiler avoid being confused by the structs
  double *currdata = curr->data.data();
  double *prevdata = prev->data.data();
  int nx = prev->nx;
  int ny = prev->ny;

  // Determine the temperature field at next time step
  // As we have fixed boundary conditions, the outermost gridpoints
  // are not updated.
  double dx2 = prev->dx * prev->dx;
  double dy2 = prev->dy * prev->dy;

  // Use OpenMP threads for parallel update of grid values
  #pragma omp parallel for
  for (int i = 1; i < nx + 1; i++) {
    for (int j = 1; j < ny + 1; j++) {
      int ind = i * (ny + 2) + j;
      int ip = (i + 1) * (ny + 2) + j;
      int im = (i - 1) * (ny + 2) + j;
      int jp = i * (ny + 2) + j + 1;
      int jm = i * (ny + 2) + j - 1;
      currdata[ind] = prevdata[ind] + a*dt*
            ((prevdata[ip] - 2.0*prevdata[ind] + prevdata[im]) / dx2 +
             (prevdata[jp] - 2.0*prevdata[ind] + prevdata[jm]) / dy2);
    }
  }
}
```

✔ Optional: compiling the executables

To compile executable files for the OpenMP-based variants, follow the instructions below:

```
salloc -A project_465002387 -p small-g -N 1 -c 8 -n 1 --gpus-per-node=1 -t 1:00:00

module load LUMI/24.03
module load partition/G
module load rocm/6.0.3
module load PrgEnv-cray/8.5.0

cd base/
make all
```

Afterwards login into a compute node and test the executables (or just `srun <executable>` directly):

```
$ srun --pty bash

$ ./stencil
$ ./stencil_off
$ ./stencil_data

$ exit
```

If everything works well, the output should look similar to this:

```
$ ./stencil
Average temperature, start: 59.763305
Average temperature at end: 59.281239
Control temperature at end: 59.281239
Iterations took 0.566 seconds.
$ ./stencil_off
Average temperature, start: 59.763305
Average temperature at end: 59.281239
Control temperature at end: 59.281239
Iterations took 3.792 seconds.
$ ./stencil_data
Average temperature, start: 59.763305
Average temperature at end: 59.281239
Control temperature at end: 59.281239
Iterations took 1.211 seconds.
$
```

CPU parallelization: timings

(**NOTE**: for thread-parallel runs it is necessary to request multiple CPU cores. In LUMI-G partitions, this can be done by asking for multiple GPUs; an alternative is to use -C partitions.)

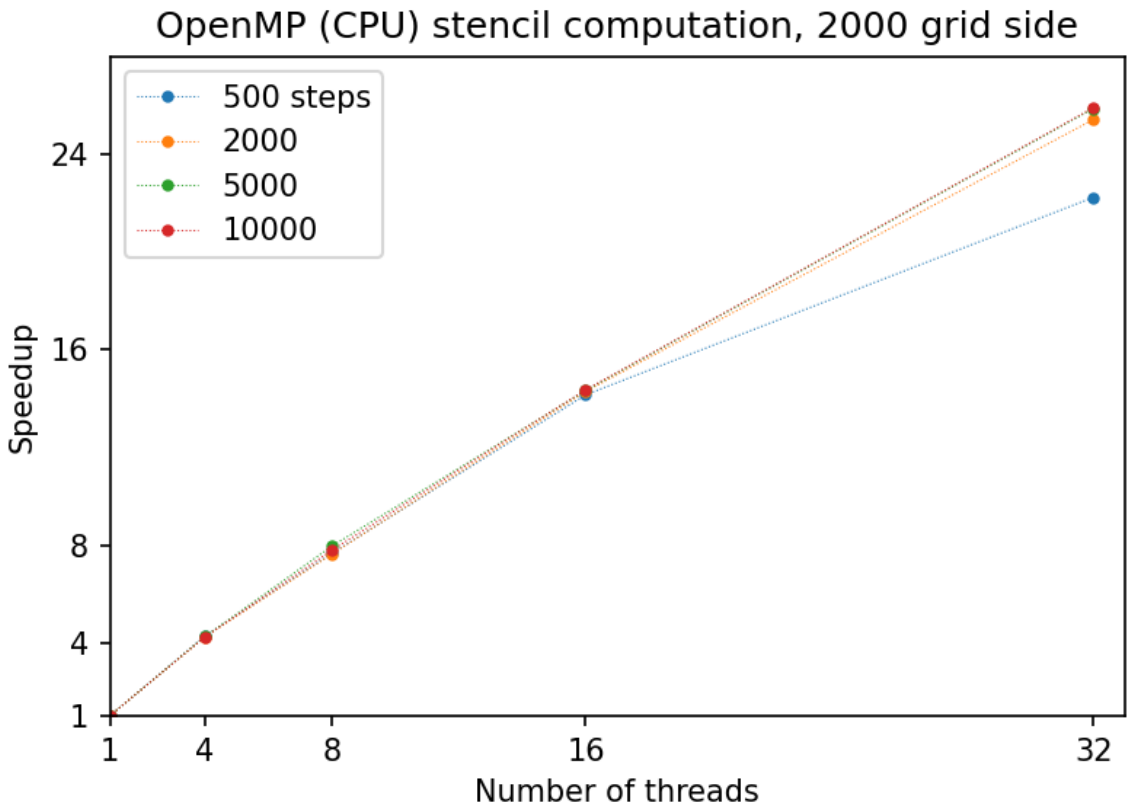
For later comparison, some benchmarks of the OpenMP thread-parallel implementation are provided below:

Run times of OpenMP-enabled executable, s

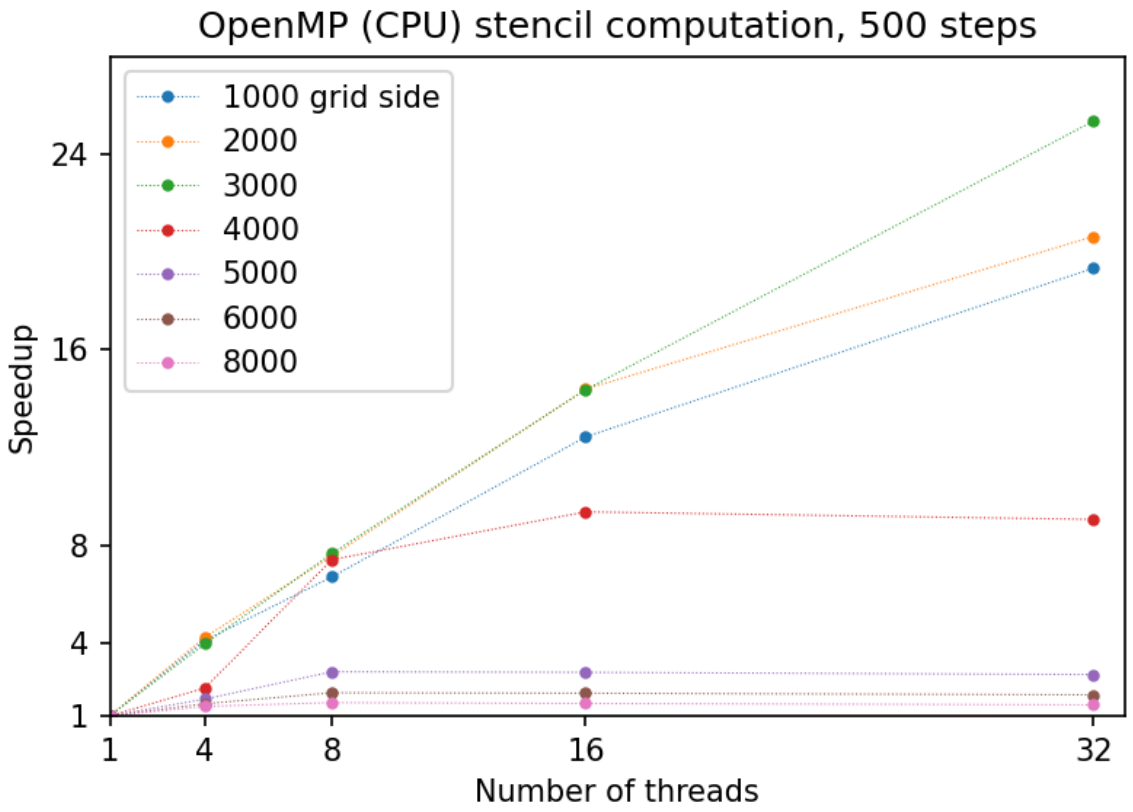
Job size	1 CPU core	32 CPU cores
S:2000 T:500	1.402	0.064
S:2000 T:5000	13.895	0.538
S:2000 T:10000	27.753	1.071

Job size	1 CPU core	32 CPU cores
S:4000 T:500	5.727	0.633
S:8000 T:500	24.130	16.616

A closer look reveals that the computation time scales very nicely with increasing **time steps**:



However, for larger **grid sizes** the parallelization becomes inefficient – as the individual chunks of the grid get too large to fit into CPU cache, threads become bound by the speed of RAM reads/writes:



✍ Discussion: heat flow computation scaling

1. How is heat flow computation **expected** to scale with respect to the number of time steps?
   a. Linearly
   b. Quadratically
   c. Exponentially
2. How is stencil application (grid update) **expected** to scale with respect to the size of the grid side?
   a. Linearly
   b. Quadratically
   c. Exponentially
3. (Optional) Do you expect GPU-accelerated computations to follow the above-mentioned trends? Why/ why not?

✓ Solution

1. The answer is a: since each time-step follows the previous one and involves a similar number of operations, then the update time per step will be more or less constant.
2. The answer is b: since stencil application is independent for every grid point, the update time will be proportional to the number of points, i.e. side * side.

GPU parallelization: first steps

Let's apply several techniques presented in previous episodes to make stencil update run on GPU.

OpenMP (or OpenACC) offloading requires to define a region to be executed in parallel as well as data that shall be copied over/ used in GPU memory. Similarly, SYCL programming model offers convenient ways to define execution kernels, as well as context to run them in (called queue).

Changes of stencil update code for OpenMP and SYCL are shown in the tabs below:

stencil/

OpenMP (naive) | SYCL (naive)

base/core-off.cpp

```
// (c) 2023 ENCCS, CSC and the contributors
#include "heat.h"

// Update the temperature values using five-point stencil
// Arguments:
//   curr: current temperature values
//   prev: temperature values from previous time step
//   a: diffusivity
//   dt: time step
void evolve(field *curr, field *prev, double a, double dt)
{
  // Help the compiler avoid being confused by the structs
  double *currdata = curr->data.data();
  double *prevdata = prev->data.data();
  int nx = prev->nx;
  int ny = prev->ny;

  // Determine the temperature field at next time step
  // As we have fixed boundary conditions, the outermost gridpoints
  // are not updated.
  double dx2 = prev->dx * prev->dx;
  double dy2 = prev->dy * prev->dy;

  // Offload value update to GPU target (fallback to CPU is possible)
  #pragma omp target teams distribute parallel for \
  map(currdata[0:(nx+2)*(ny+2)],prevdata[0:(nx+2)*(ny+2)])
  for (int i = 1; i < nx + 1; i++) {
    for (int j = 1; j < ny + 1; j++) {
      int ind = i * (ny + 2) + j;
      int ip = (i + 1) * (ny + 2) + j;
      int im = (i - 1) * (ny + 2) + j;
      int jp = i * (ny + 2) + j + 1;
      int jm = i * (ny + 2) + j - 1;
      currdata[ind] = prevdata[ind] + a*dt*
            ((prevdata[ip] - 2.0*prevdata[ind] + prevdata[im]) / dx2 +
             (prevdata[jp] - 2.0*prevdata[ind] + prevdata[jm]) / dy2);
    }
  }
}
```

Loading SYCL modules on LUMI

As SYCL is placed on top of ROCm/HIP (or CUDA) software stack, running SYCL executables may require respective modules to be loaded. On current nodes, it can be done as follows:

```
# salloc -A project_465002387 -p small-g -N 1 -c 8 -n 1 --gpus-per-node=1 -t
1:00:00

module load LUMI/24.03
module load partition/G
module load rocm/6.0.3
module use  /appl/local/csc/modulefiles
module load acpp/24.06.0
```

✔ Optional: compiling the SYCL executables

As previously, you are welcome to generate your own executables:

```
$ cd ../sycl/
(give the following lines some time, probably a couple of min)
$ acpp -O2 -o stencil_naive core-naive.cpp io.cpp main-naive.cpp pngwriter.c
setup.cpp utilities.cpp
$ acpp -O2 -o stencil_data core.cpp io.cpp main.cpp pngwriter.c setup.cpp
utilities.cpp

$ srun stencil_naive
$ srun stencil_data
```

If everything works well, the output should look similar to this:

```
$ srun stencil_naive
Average temperature, start: 59.763305
Average temperature at end: 59.281239
Control temperature at end: 59.281239
Iterations took 2.086 seconds.
$ srun stencil_data
Average temperature, start: 59.763305
Average temperature at end: 59.281239
Control temperature at end: 59.281239
Iterations took 0.052 seconds.
```

✍ Exercise: naive GPU ports

Test your compiled executables `base/stencil`, `base/stencil_off` and `sycl/stencil_naive`. Try changing problem size parameters:

- `srun stencil_naive 2000 2000 5000`

Things to look for:

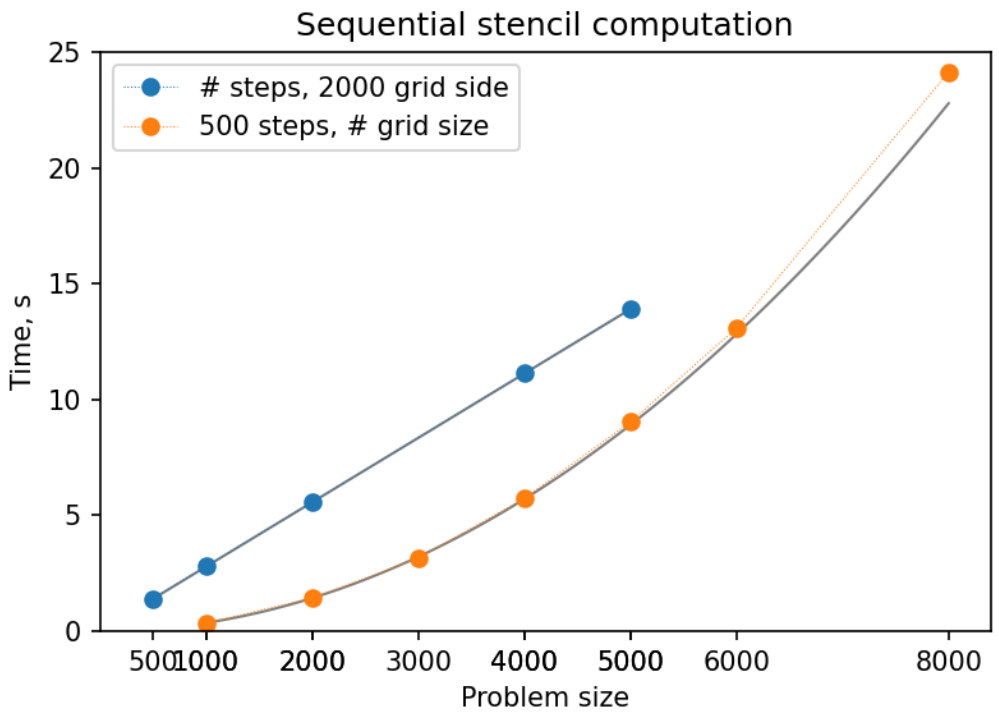
- How computation times change?
- Do the results align to your expectations?

✔ Solution

You might notice that the GPU-“ported” versions actually run slower than the single-CPU-core version! In fact, the scaling behavior of all three variants is similar and expected, which is a good sign; only the “computation unit cost” is different. You can

compare benchmark summaries in the tabs below:

Sequential | OpenMP (naive) | SYCL (naive)



GPU parallelization: data movement

Why the porting approach above seems to be quite inefficient?

On each step, we:

- re-allocate GPU memory,
- copy the data from CPU to GPU,
- perform the computation,
- then copy the data back.

But overhead can be reduced by taking care to minimize data transfers between *host* and *device* memory:

- allocate GPU memory once at the start of the program,
- only copy the data from GPU to CPU when we need it,
- swap the GPU buffers between timesteps, like we do with CPU buffers. (OpenMP does this automatically.)

Changes of stencil update code are shown in tabs below (also check out the respective main() functions for calls to persistent GPU buffer creation, access, and deletion):

stencil/

OpenMP | SYCL

base/core-data.cpp

```
// (c) 2023 ENCCS, CSC and the contributors
#include "heat.h"

// Update the temperature values using five-point stencil
// Arguments:
//   curr: current temperature values
//   prev: temperature values from previous time step
//   a: diffusivity
//   dt: time step
void evolve(field *curr, field *prev, double a, double dt)
{
  // Help the compiler avoid being confused by the structs
  double *currdata = curr->data.data();
  double *prevdata = prev->data.data();
  int nx = prev->nx;
  int ny = prev->ny;

  // Determine the temperature field at next time step
  // As we have fixed boundary conditions, the outermost gridpoints
  // are not updated.
  double dx2 = prev->dx * prev->dx;
  double dy2 = prev->dy * prev->dy;

  // Offload value update to GPU target (fallback to CPU is possible)
  #pragma omp target teams distribute parallel for
  for (int i = 1; i < nx + 1; i++) {
    for (int j = 1; j < ny + 1; j++) {
      int ind = i * (ny + 2) + j;
      int ip = (i + 1) * (ny + 2) + j;
      int im = (i - 1) * (ny + 2) + j;
      int jp = i * (ny + 2) + j + 1;
      int jm = i * (ny + 2) + j - 1;
      currdata[ind] = prevdata[ind] + a*dt*
            ((prevdata[ip] - 2.0*prevdata[ind] + prevdata[im]) / dx2 +
             (prevdata[jp] - 2.0*prevdata[ind] + prevdata[jm]) / dy2);
    }
  }
}

// Start a data region and copy temperature fields to the device
void enter_data(field *curr, field *prev)
{
  double *currdata = curr->data.data();
  double *prevdata = prev->data.data();
  int nx = prev->nx;
  int ny = prev->ny;

  // adding data mapping here
  #pragma omp target enter data \
  map(to: currdata[0:(nx+2)*(ny+2)], prevdata[0:(nx+2)*(ny+2)])
}

// End a data region and copy temperature fields back to the host
void exit_data(field *curr, field *prev)
{
  double *currdata = curr->data.data();
  double *prevdata = prev->data.data();
  int nx = prev->nx;
  int ny = prev->ny;

  // adding data mapping here
  #pragma omp target exit data \
  map(from: currdata[0:(nx+2)*(ny+2)], prevdata[0:(nx+2)*(ny+2)])
}
```

```
// Copy a temperature field from the device to the host
void update_host(field *heat)
{
  double *data = heat->data.data();
  int nx = heat->nx;
  int ny = heat->ny;

  // adding data mapping here
  #pragma omp target update from(data[0:(nx+2)*(ny+2)])
}
```

✍️ Exercise: updated GPU ports

Test your compiled executables `base/stencil_data` and `sycl/stencil_data`. Try changing problem size parameters:

- `srun stencil 2000 2000 5000`

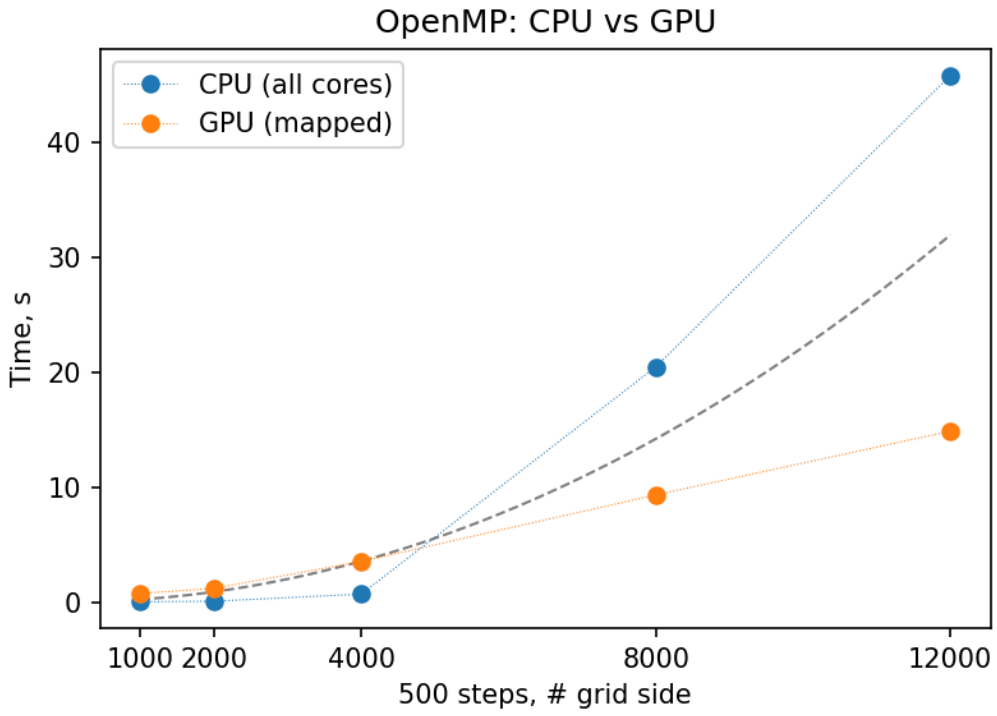
Things to look for:

- How computation times change this time around?
- What largest grid and/or longest propagation time can you get in 10 s on your machine?

✔ Solution

OpenMP data mapping | SYCL device buffers

Using GPU offloading with mapped device data, it is possible to achieve performance gains compared to thread-parallel version for larger grid sizes, due to the fact that the latter version becomes essentially RAM-bound, but the former does not.



Python: JIT and GPU acceleration

As mentioned previously, Numba package allows developers to just-in-time (JIT) compile Python code to run fast on CPUs, but can also be used for JIT compiling for (NVIDIA) GPUs. JIT seems to work well on loop-based, computationally heavy functions, so trying it out is a nice choice for initial source version:

stencil/python-numba

Stencil update | Data generation | Stencil update in GPU

core.py

```
from numba import jit


# Update the temperature values using five-point stencil
# Arguments:
#   curr: current temperature field object
#   prev: temperature field from previous time step
#   a: diffusivity
#   dt: time step
def evolve(current, previous, a, dt):
    dx2, dy2 = previous.dx**2, previous.dy**2
    curr, prev = current.data, previous.data
    # Run (possibly accelerated) update
    _evolve(curr, prev, a, dt, dx2, dy2)


@jit(nopython=True)
def _evolve(curr, prev, a, dt, dx2, dy2):
    nx, ny = prev.shape # These are the FULL dims, rows+2 / cols+2
    for i in range(1, nx-1):
        for j in range(1, ny-1):
            curr[i, j] = prev[i, j] + a * dt * ( \
              (prev[i+1, j] - 2*prev[i, j] + prev[i-1, j]) / dx2 + \
              (prev[i, j+1] - 2*prev[i, j] + prev[i, j-1]) / dy2 )
```

The alternative approach would be to rewrite stencil update code in NumPy style, exploiting loop vectorization.

Trying out Python examples

You can run follow the links below for instructions from the Setup episode. You may choose to run the provided code examples either on

- on a LUMI GPU node, or
- your local machine, or LUMI CPU node, or
- Google Colab.

To run the example in a GPU node via the container,

```
$ # skip the git clone step, if you have done that already
$ git clone https://github.com/ENCCS/gpu-programming.git
$ # allocate a GPU node using `salloc`
$ # execute the `srun --pty singularity exec ...` command
Singularity> cd /work/gpu-programming/content/examples/stencil/python-numba
Singularity> . /.venv/bin/activate
Singularity> python3 main.py
```

To run the example in a CPU node,

```
$ git clone https://github.com/ENCCS/gpu-programming.git
$ cd gpu-programming/content/examples/stencil/python-numba
$ # make sure you have active allocation
$ srun python3 main.py
```

Short summary of a typical Colab run is provided below:

Run times of Numba JIT-enabled Python program, s

Job size	JIT (LUMI)	JIT (Colab)	Job size	no JIT (Colab)
S:2000 T:500	1.648	8.495	S:200 T:50	5.318
S:2000 T:200	0.787	3.524	S:200 T:20	1.859
S:1000 T:500	0.547	2.230	S:100 T:50	1.156

Numba's `@vectorize` and `@guvectorize` decorators offer an interface to create CPU- (or GPU-) accelerated *Python* functions without explicit implementation details. However, such functions become increasingly complicated to write (and optimize by the compiler) with increasing complexity of the computations within.

Numba also offers direct CUDA-based kernel programming, which can be the best choice for those already familiar with CUDA. Example for stencil update written in Numba CUDA is shown in the above section, tab "Stencil update in GPU". In this case, data transfer functions `devdata = cuda.to_device(data)` and `devdata.copy_to_host(data)` (see `main_cuda.py`) are already provided by Numba package.

✍ Exercise: CUDA acceleration in Python

Using Google Colab (or your own machine), run provided Numba-CUDA Python program. Try changing problem size parameters:

- `args.rows, args.cols, args.nsteps = 2000, 2000, 5000` for notebooks,
- [`srun`] `python3 main.py 2000 2000 5000` for command line.

Things to look for:

- How computation times change?
- Do you get better performance than from JIT-compiled CPU version? How far can you push the problem size?
- Are you able to monitor the GPU usage?

✔ Solution

Some numbers from Colab:

Run times of Numba CUDA Python program, s

Job size	JIT (LUMI)	JIT (Colab)	CUDA (Colab)
S:2000 T:500	1.648	8.495	1.079
S:2000 T:2000	6.133	36.61	3.931
S:5000 T:500	9.478	57.19	6.448

Julia GPU acceleration

A Julia version of the stencil example above can be found below (a simplified version of the HeatEquation module at https://github.com/ENCCS/HeatEquation.jl). The source files are also available in the content/examples/stencil/julia directory of this repository.

To run the example on LUMI CPU partition, type:

```
$ # interactive CPU node
$ srun --account=project_465002387 --partition=standard --nodes=1 --cpus-per-task=32 --
ntasks-per-node=1 --time=01:00:00 --pty bash
$ # load Julia env
$ module purge
$ module use /appl/local/csc/modulefiles
$ module load julia
$ module load julia-amdgpu
$ # in directory with Project.toml and source files, instantiate an environment to
install packages
$ julia --project -e "using Pkg ; Pkg.instantiate()"
$ # finally run
$ julia --project main.jl
```

To run on the GPU partition, use instead the `srun` command

```
$ srun --account=project_465002387 --partition=standard-g --nodes=1 --cpus-per-task=1 -
-ntasks-per-node=1 --gpus-per-node=1 --time=1:00:00 --pty bash
```

Optional dependency

Note that the `Plots.jl` dependency is commented out in `main.jl` and `Project.toml`. This saves ~2 minute precompilation time when you first instantiate the Julia environment. To generate plots, just uncomment the commented `Plots.jl` dependency in `Project.toml`, instantiate again, and import and use `Plots` in `main.jl`.

main.jl | core.jl | heat.jl | Project.toml

```
#using Plots
using BenchmarkTools

include("heat.jl")
include("core.jl")


"""
    visualize(curr::Field, filename=:none)

Create a heatmap of a temperature field. Optionally write png file.
"""
function visualize(curr::Field, filename=:none)
    background_color = :white
    plot = heatmap(
        curr.data,
        colorbar_title = "Temperature (C)",
        background_color = background_color
    )

    if filename != :none
        savefig(filename)
    else
        display(plot)
    end
end


ncols, nrows = 2048, 2048
nsteps = 500

# initialize current and previous states to the same state
curr, prev = initialize(ncols, nrows)

# visualize initial field, requires Plots.jl
#visualize(curr, "initial.png")

# simulate temperature evolution for nsteps
simulate!(curr, prev, nsteps)

# visualize final field, requires Plots.jl
#visualize(curr, "final.png")
```

✍ Exercise: Julia port to GPUs

Carefully inspect all Julia source files and consider the following questions:

1. Which functions should be ported to run on GPU?
2. Look at the `initialize!()` function and how it uses the `arraytype` argument. This could be done more compactly and elegantly, but this solution solves scalar indexing errors. What are scalar indexing errors?

3. Try to start sketching GPU-ported versions of the key functions.
4. When you have a version running on a GPU (your own or the solution provided below), try benchmarking it by adding `@btime` in front of `simulate!()` in `main.jl`. Benchmark also the CPU version, and compare.

✔ Hints

- create a new function `evolve_gpu!()` which contains the GPU kernelized version of `evolve!()`
- in the loop over timesteps in `simulate!()`, you will need a conditional like `if typeof(curr.data) <: ROCArray` to call your GPU-ported function
- you cannot pass the struct `Field` to the kernel. You will instead need to directly pass the array `Field.data`. This also necessitates passing in other variables like `curr.dx^2`, etc.

✔ More hints

- since the data is two-dimensional, you'll need `i = (blockIdx().x - 1) * blockDim().x + threadIdx().x` and `j = (blockIdx().y - 1) * blockDim().y + threadIdx().y`
- to not overindex the 2D array, you can use a conditional like `if i > 1 && j > 1 && i < nx+2 && j < ny+2`
- when calling the kernel, you can set the number of threads and blocks like `xthreads = ythreads = 16` and `xblocks, yblocks = cld(curr.nx, xthreads), cld(curr.ny, ythreads)`, and then call it with, e.g., `@roc threads=(xthreads, ythreads) blocks = (xblocks, yblocks) evolve_rocm!(curr.data, prev.data, curr.dx^2, curr.dy^2, nx, ny, a, dt)`.

✔ Solution

1. The `evolve!()` and `simulate!()` functions need to be ported. The `main.jl` file also needs to be updated to work with GPU arrays.
2. "Scalar indexing" is where you iterate over a GPU array, which would be excruciatingly slow and is indeed only allowed in interactive REPL sessions. Without the if-statements in the `initialize!()` function, the `generate_field!()` method would be doing disallowed scalar indexing if you were running on a GPU.
3. The GPU-ported version is found below. Try it out on both CPU and GPU and observe the speedup. Play around with array size to see if the speedup is affected. You can also play around with the `xthreads` and `ythreads` variables to see if it changes anything.

main_gpu.jl | core_gpu.jl

```
#using Plots
using BenchmarkTools
using AMDGPU

include("heat.jl")
include("core_gpu.jl")


"""
    visualize(curr::Field, filename=:none)

Create a heatmap of a temperature field. Optionally write png file.
"""
function visualize(curr::Field, filename=:none)
    background_color = :white
    plot = heatmap(
        curr.data,
        colorbar_title = "Temperature (C)",
        background_color = background_color
    )

    if filename != :none
        savefig(filename)
    else
        display(plot)
    end
end


ncols, nrows = 2048, 2048
nsteps = 500

# initialize data on CPU
curr, prev = initialize(ncols, nrows, ROCArray)
# initialize data on CPU
#curr, prev = initialize(ncols, nrows)

# visualize initial field, requires Plots.jl
#visualize(curr, "initial.png")

# simulate temperature evolution for nsteps
@btime simulate!(curr, prev, nsteps)

# visualize final field, requires Plots.jl
#visualize(curr, "final.png")
```

See also

This section leans heavily on source code and material created for several other computing workshops by ENCCS and CSC and adapted for the purposes of this lesson. If you want to know more about specific programming models / framework, definitely check these out!

- OpenMP for GPU offloading
- Heterogeneous programming with SYCL

- Educational implementation of heat flow example (incl. MPI-aware CUDA)

Quick Reference

Glossary

thread

Definition. otherframework: **workitem**

workitem

Definition. otherframework: **thread**

Abbreviations

Abbreviations	Full names
CUDA	compute unified device architecture
DAG	directed acyclic graph
FPGAs	field-programmable gate arrays
GPU	graphics processing units
HIP	heterogeneous-computing interface for portability
NLP	natural language processing
SIMD	single instruction multiple data
SIMT	single instruction multiple threads
SP	streaming processors
SMP	streaming multi-processors
SVM	shared virtual memory
USM	unified shared memory

Instructor's guide

Updated schedule for a three-day workshop (2024)

Day 1

Time	Section
9:00-9:15	Welcome
9:15-9:40	Why GPUs?
9:40-10:20	The GPU hardware and software ecosystem

Time	Section
10:20-10:30	Break
10:30-11:00	What problems fit to GPU?
11:00-11:30	GPU programming concepts
11:30-12:00	Introduction to GPU programming models
12:00-13:00	Lunch break
13:00-14:20	Directive-based models
14:20-14:30	Break
14:30-16:00	Non-portable kernel-based models

Day 2

Time	Section
9:00-10:30	Portable kernel-based models
10:30-10:40	Break
10:40-12:00	Exercises for various programming models
12:00-13:00	Lunch break
13:00-14:15	High-level language support
14:14-14:30	Break
14:30-15:50	Multi-GPU programming with MPI
15:50-16:00	Buffer time

Day 3

Time	Section
09:00-10:00	Preparing code for GPU porting
10:00-10:30	Recommendations and discussions
10:30-10:45	Break
10:45-11:50	Problem example
11:50-12:00	Wrap-up
12:00-13:00	Lunch break
13:00-15:50	Bring your code and get expert advice
15:50-16:00	Summary of this workshop

Suggested two-day schedule (2023)

Day 1

Time	Section
9:00-9:15	Welcome
9:15-9:30	Why GPUs?
9:30-9:50	The GPU hardware and software ecosystem
9:50-10:10	What problems fit to GPU?
10:10-10:25	Break
10:25-10:50	GPU programming concepts
10:50-11:10	Introduction to GPU programming models
11:10-11:50	High-level language support
11:50-12:50	Lunch break
12:50-13:40	Directive-based models
13:40-14:30	Multi-GPU programming with MPI
14:30-14:45	Break
14:45-16:00	Non-portable kernel-based models

Day 2

Time	Section
9:00-10:15	Portable kernel-based models
10:15-10:30	Break
10:30-11:20	Preparing code for GPU porting
11:20-12:00	Recommendations and discussions
12:00-13:00	Lunch break
13:00-14:30	Problem example
14:30-14:50	Break
14:50-16:00	Buffer time

Who is the course for?

This material is most relevant to researchers and engineers who already develop software which runs on CPUs in workstations or supercomputers, but also to decision makers or project managers who don't write code but make strategic decisions in software projects, whether it's in academia, industry or the public sector.

About the course

This training material is the result of a multilateral effort by GPU programming experts from:

- Aalto University in Finland
- Aarhus University in Denmark
- CSC in Finland
- ENCCS in Sweden
- HPC2N centre in Umeå, Sweden
- KTH Royal Institute for Technology in Sweden
- NRIS in Norway
- Vilnius University in Lithuania and NCC Lithuania

See also

Links to additional resources and tutorials can be found in the lesson episodes.

Credits

Several sections in this lesson have been adapted from the following sources created by ENCCS and CSC, which are all distributed under a Creative Commons Attribution license (CC-BY-4.0):

- OpenMP for GPU offloading
- High Performance Data Analytics in Python
- Julia for HPC

The lesson file structure and browsing layout is inspired by and derived from work by CodeRefinery licensed under the MIT license. We have copied and adapted most of their license text.

Instructional Material

This instructional material is made available under the Creative Commons Attribution license (CC-BY-4.0). The following is a human-readable summary of (and not a substitute for) the full legal text of the CC-BY-4.0 license. You are free to:

- **share** - copy and redistribute the material in any medium or format
- **adapt** - remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow these license terms:

- **Attribution** - You must give appropriate credit (mentioning that your work is derived from work that is Copyright (c) ENCCS and individual contributors and, where practical, linking to https://enccs.github.io/sphinx-lesson-template), provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **No additional restrictions** - You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

With the understanding that:

- You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.
- No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Software

Except where otherwise noted, the example programs and other software provided with this repository are made available under the OSI-approved MIT license.