NHR Summer School
May 19th – 23rd, 2025

Friedrich-Alexander-Universität Erlangen-Nürnberg

Introduction to GPU Programming: Models, Techniques, and Hands-On Exploration

Sebastian Kuckuk, *Erlangen National High Performance Computing Center (NHR@FAU)*

Introduction

Introduction – Motivation

- Why use GPUs?

- GPUs are widespread in the HPC landscape
- c.f. Top500 list: https://www.top500.org/lists/top500/2024/11/
  - 9 out of the top 10 supercomputers are equipped with GPUs
  - 3 NVIDIA (H100, GH200, A100)
  - 5 AMD (2 x MI300A, 3 x MI250X)
  - 1 Intel (GPU Max Series)

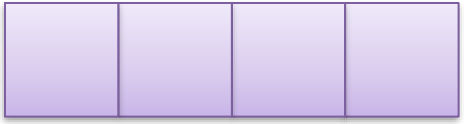
- GPUs promise massive performance
  - Around an order of magnitude more than a CPU

GPU Performance

- If GPUs are mainly about performance – what is performance?
- Two primary factors
  - How fast can the meaningful computation be done?
    Usually given as computational throughput, e.g. FLOP/s
  - How fast can the data be transferred to where the computation is happening (and back)?
    Usually given as sustained bandwidth, e.g. GB/s
- What are GPUs doing differently?

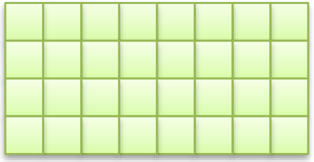
GPU vs CPU

- CPUs have few (~100) powerful cores

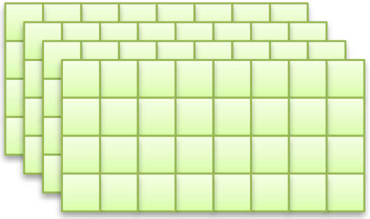


- Organized in sockets

- GPUs have many simplistic 'cores’ (10 000s)

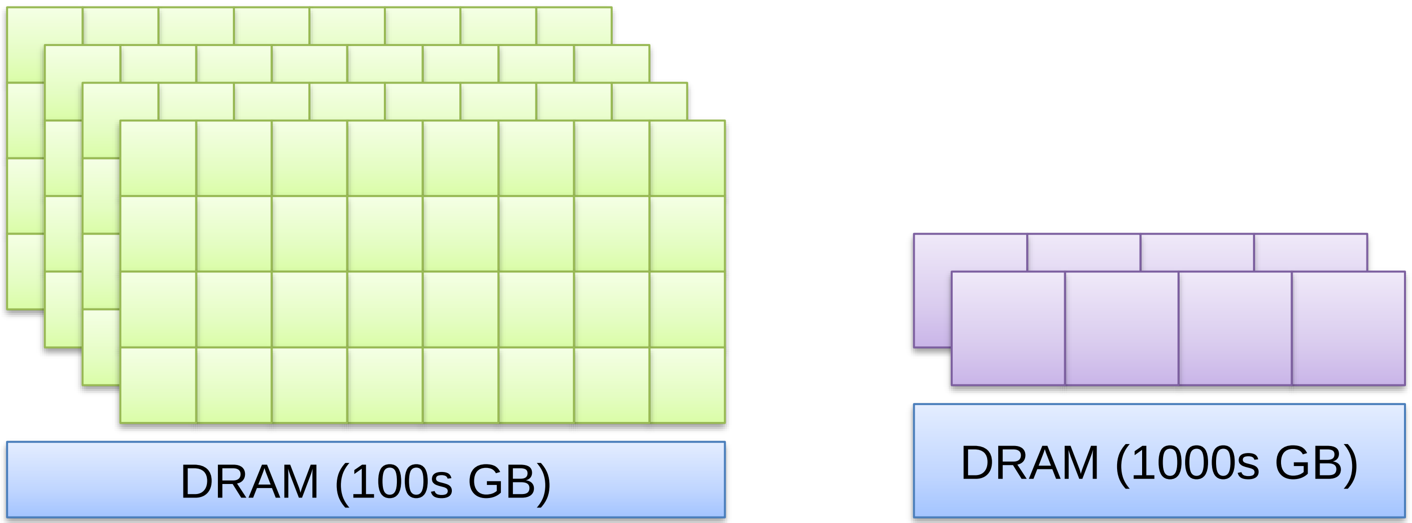


- Organized in 100s of streaming multiprocessors (SMs)



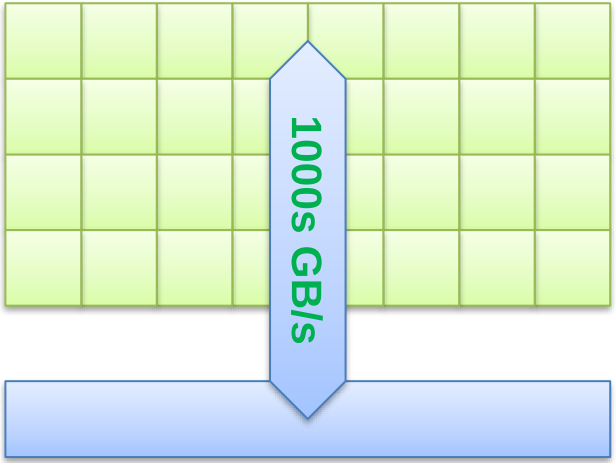
GPU vs CPU

- Both CPU and GPU have a distinct main memory (for most architectures)

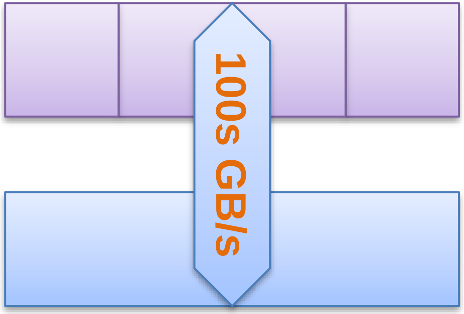


GPU vs CPU

- The memory of GPUs
  - Is optimized for bandwidth
  - Has a high latency

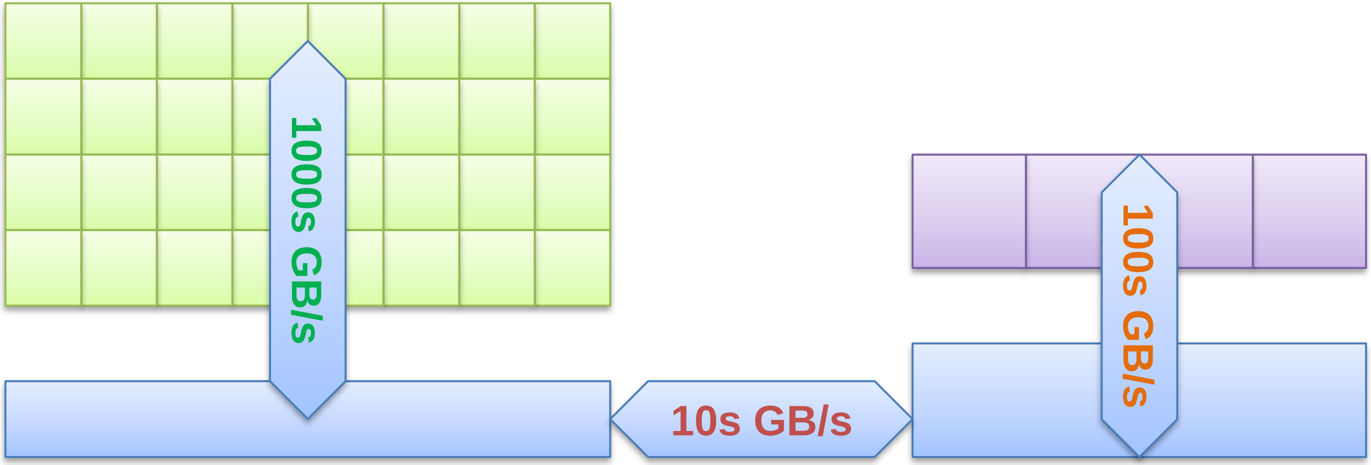


- The memory of CPUs
  - Is optimized for latency
  - Has a high capacity



GPU vs CPU

- Both CPU and GPU have a distinct main memory (for most architectures)
- They communicate via slow interconnects (for most architectures)



Introduction – CPU vs GPU

- Why not use GPUs exclusively?

- Simple benchmark similar to the bandwidth benchmark discussed later
- One A100 40GB GPU (Alex) vs one Sapphire Rapids node (Fritz)
  - ~ 1.3 TB/s vs ~ 260 GB/s => 5x faster

- But: One SM of one A100: ~ 90 GB/s => 3x slower

- Serial execution
  - ~ 0.3 GB/s vs ~ 20 GB/s => 67x slower

Introduction – CPU vs GPU

- Why not use GPUs exclusively?

- GPUs require
  - Massive parallelism
    10 000s of threads to saturate computation
    100 000s of threads to saturate memory

  - Structured computations
    Ideally each thread does the same operation but on different parts of a structured data set

- CPUs deal much better with unstructured, low-parallelism computations

GPU Programming
– without the actual programming

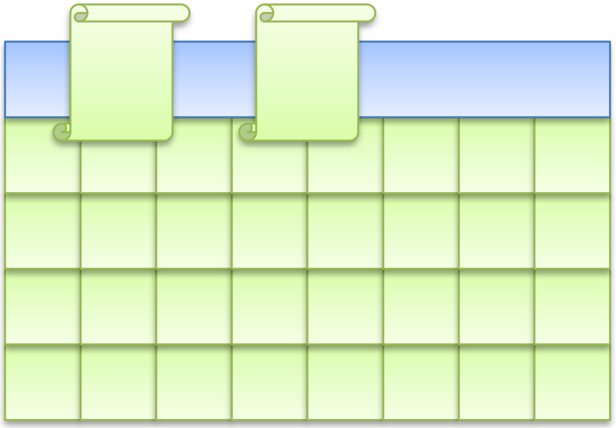
GPU Programming Considerations

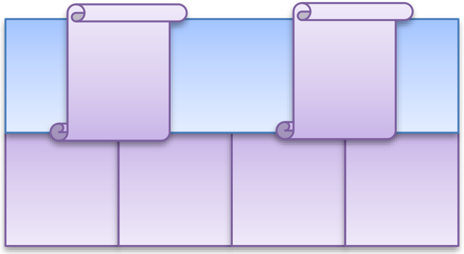
- Generally we require concepts for
- *Execution spaces*: where is the meaningful computation happening and
- *Memory spaces*: where are input and output data located

Data Spaces

- Memory spaces are separate for host and device

- Requirements
  - Allocate/ deallocate data on host/device
  - Move data between host and device

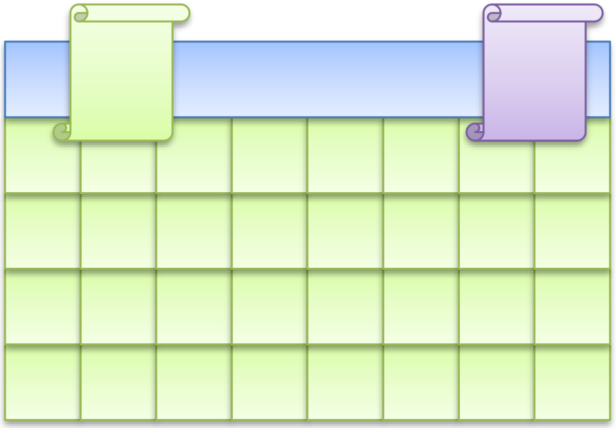


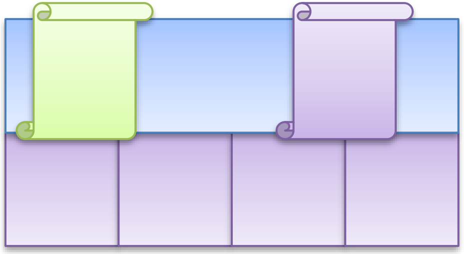


Data Spaces

- Memory spaces are separate for host and device

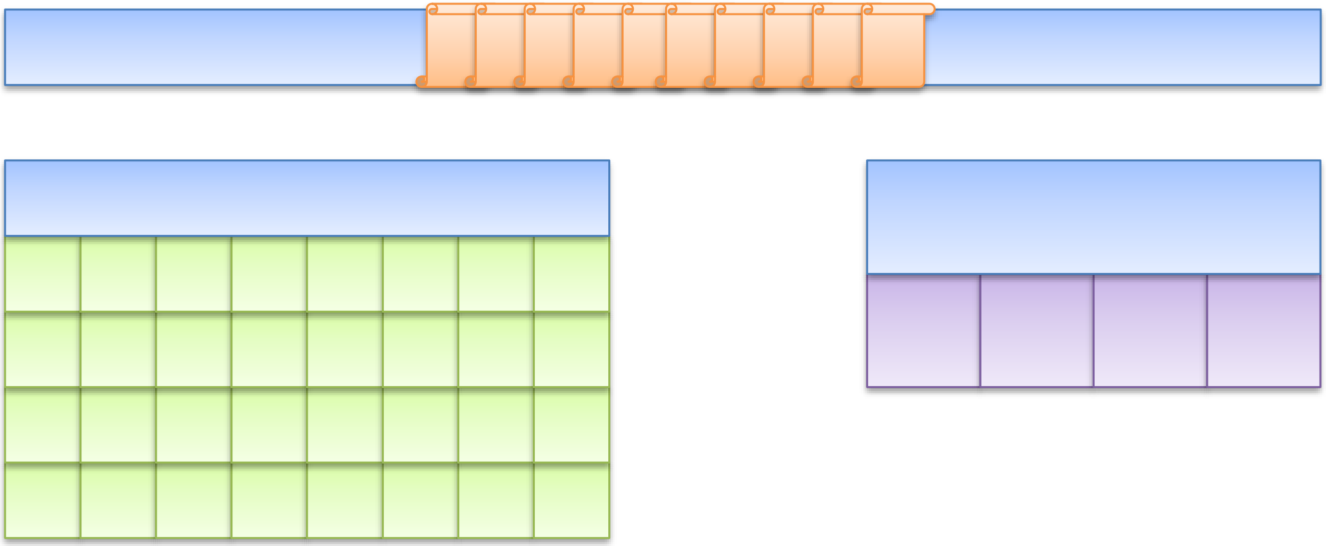
- Requirements
  - Allocate/ deallocate data on host/device
  - Move data between host and device





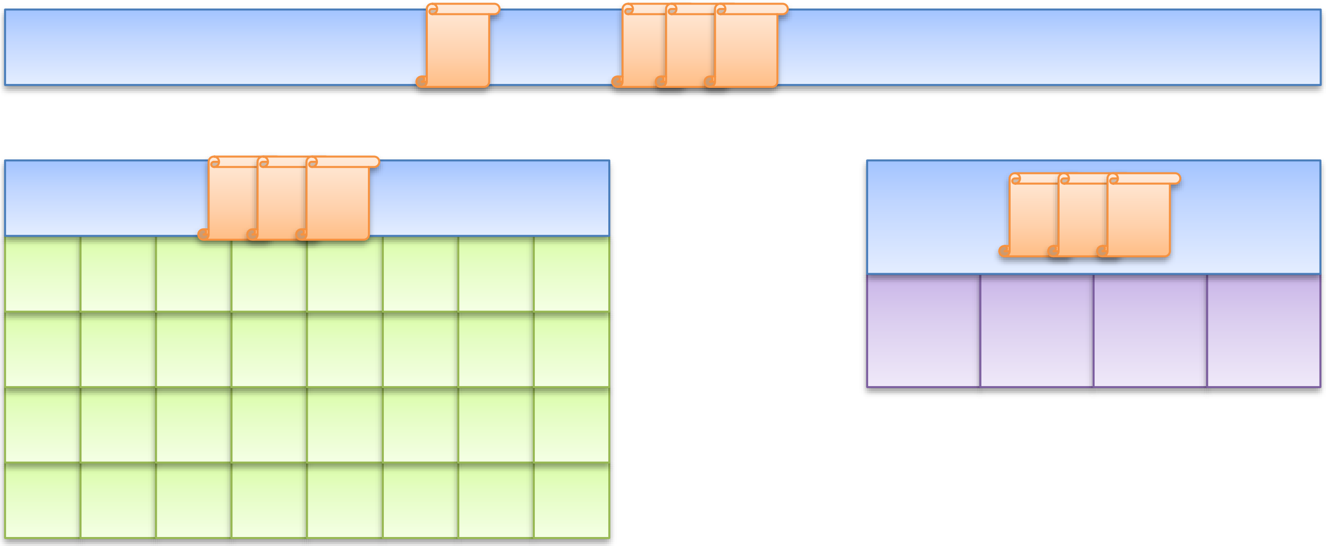
Data Spaces

- Memory spaces are *physically* separate for host and device
- But they can be organized in a *virtual* unified address space
- Data access triggers data *migration* (usually via page faulting)



Data Spaces

- Memory spaces are *physically* separate for host and device
- But they can be organized in a *virtual* unified address space
- Data access triggers data *migration* (usually via page faulting)



Explicit Memory vs Managed Memory

Explicit memory management

- Allocations are done either for host *or* for device
- Data transfers are *explicit*, e.g. via copy operations between host and device allocations
- Data structures are managed as pairs of host and device instances

➢ Both can be modified concurrently

Unified/ managed memory

- Allocations are done for a unified memory space
- Data transfers are *implicit* and follow a given granularity
- Data structures are managed as single instances

➢ Concurrent modification reduces performance

Explicit Memory vs Managed Memory

Explicit memory management

- Allocations are done either for host o for device
- Da
- Data s tures are managed as pairs of ost and device instances

➢ Both can be modified concurrently

Control ↑
Potential Performance ↑
Effort required ↑
Error-proneness ↑

↑ Implementation simplicity
↑ Robustness

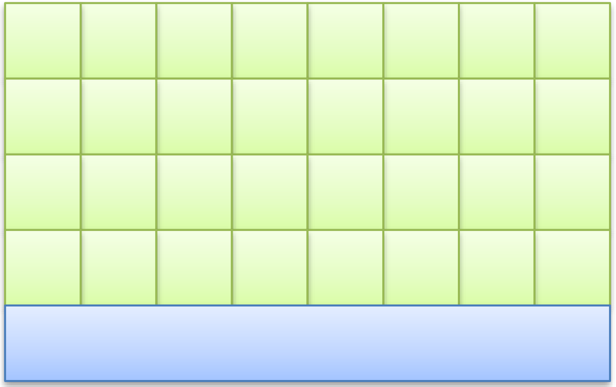
Unified/ managed memory

- Allocations are done for a unified memory spac
- Data structures are ma ged as single instances

➢ Concurrent modification reduces performance

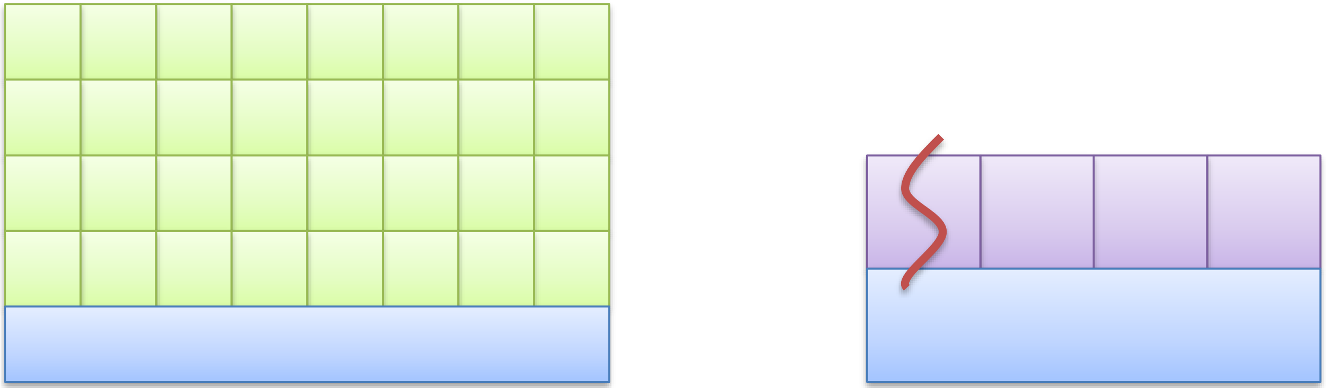
Execution Spaces

- Execution spaces are separate for host and device
  - GPU execution is generally *asynchronous* w.r.t. host execution
  - Host and device can do independent work concurrently



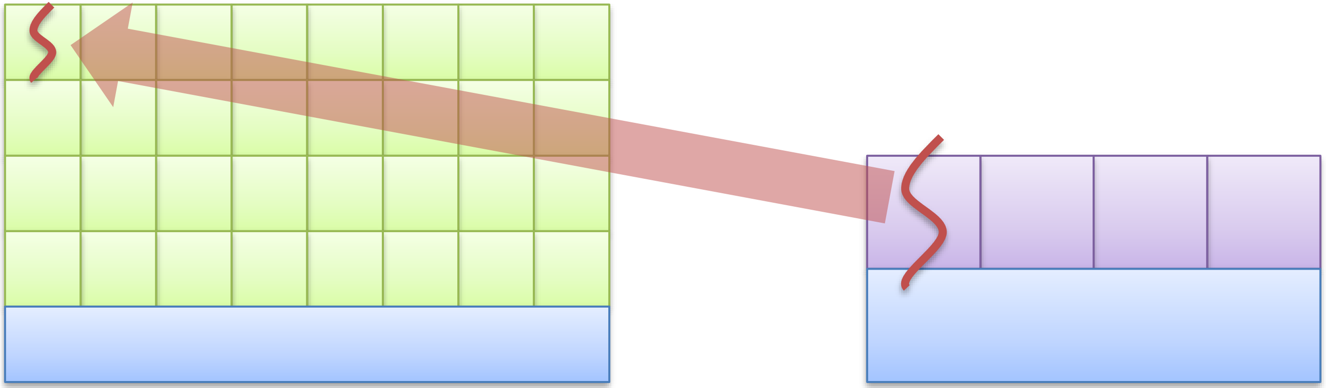
Execution Spaces

- Execution usually starts on the host



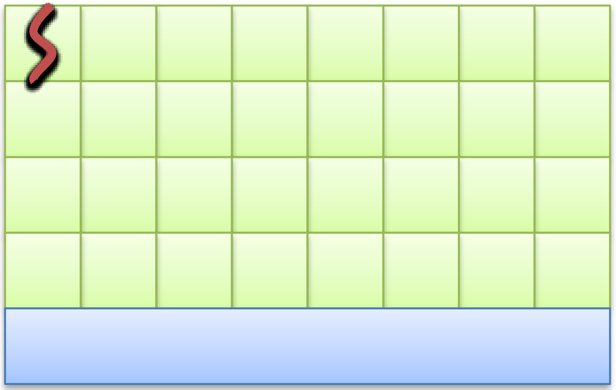
Execution Spaces

- Execution usually starts on the host
- At some point, concurrent execution on the device is started



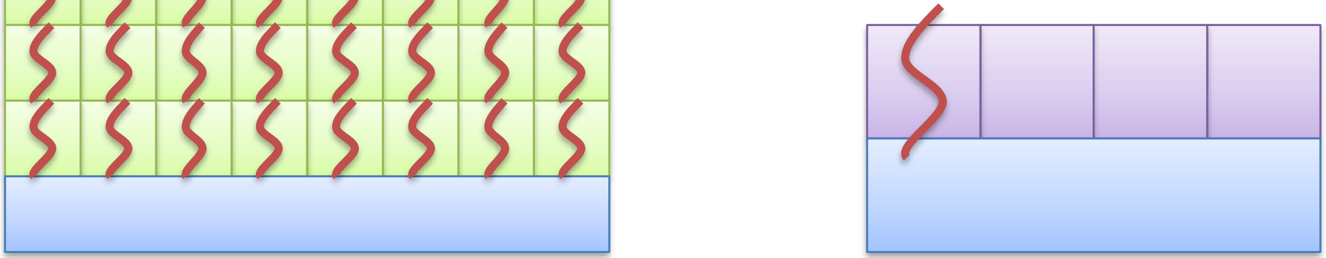
Execution Spaces

- Execution usually starts on the host
- At some point, concurrent execution on the device is started …
- … and threads are spawned for parallel execution



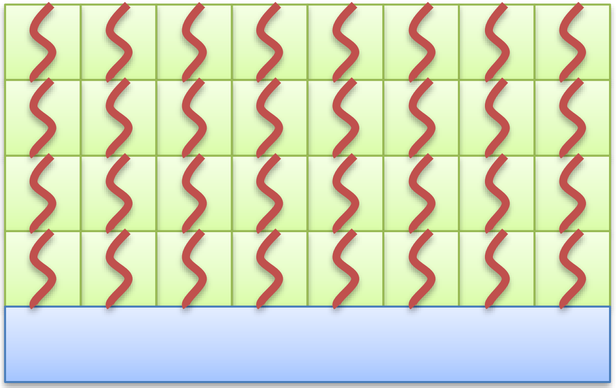
Execution Spaces

- Execution usually starts on the host
- At some point, concurrent execution on the device is started …
- … and threads are spawned for parallel execution



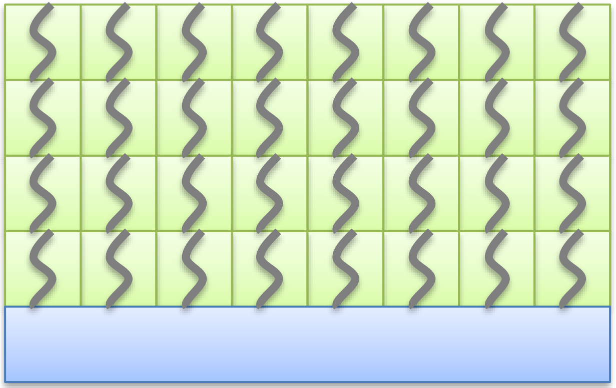
Execution Spaces

- Execution usually starts on the host
- At some point, concurrent execution on the device is started …
- … and threads are spawned for parallel execution
- When the host requires the result of the computation, it has to wait …



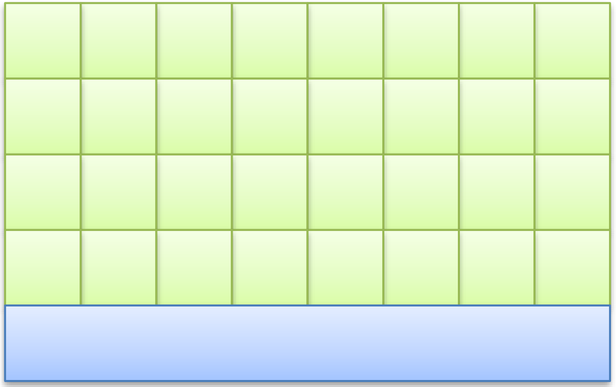
Execution Spaces

- Execution usually starts on the host
- At some point, concurrent execution on the device is started …
- … and threads are spawned for parallel execution
- When the host requires the result of the computation, it has to wait …



Execution Spaces

- Execution usually starts on the host
- At some point, concurrent execution on the device is started …
- … and threads are spawned for parallel execution
- When the host requires the result of the computation, it has to wait …
- … before continuing with its execution



Execution Spaces – Required Concepts

- Initiate execution on GPU
- Parallelization control – how many threads (and how are they organized)?
- Thread mapping – which part of the problem is computed by each thread?
- Synchronization

Workflow of GPU-Accelerated Applications

1. Allocate data for CPU/GPU
2. Initialize data on CPU

3. Copy data from CPU to GPU
4. Launch GPU kernels
5. Do independent work on CPU (optional)
6. Synchronize GPU
7. Copy data from GPU to CPU

8. Post-process data on CPU
9. De-Allocate data

Workflow of GPU-Accelerated Applications

1. Allocate data for CPU/GPU
2. Initialize data on CPU

3. Copy data from CPU to GPU
4. Launch GPU kernels
5. Do independent work on CPU (optional)
6. Synchronize GPU
7. Copy data from GPU to CPU

8. Post-process data on CPU
9. De-Allocate data

GPU Programming Considerations – Summary

- Memory spaces
  - Allocate/ deallocate data on host/device
  - Move data between host and device

- Execution spaces
  - Transfer execution to device
  - Coordinate parallel threads
  - Synchronization

- Fine-grained asynchronicity and overlap

GPU Programming Approaches

GPU Programming Approaches

Options are plentiful

1. Avoid GPU programming all together
   - GPU-accelerated libraries
2. Let the compiler do the job
   - Modern C++
   - Pragma-based approaches (OpenMP, OpenACC)
3. Get your hands dirty and do the technical work
   - CUDA, HIP, oneAPI, …
4. Do all the work – but now with added performance portability
   - Software layers (Kokkos, …)

GPU Programming Approaches

GPU-accelerated libraries and frameworks

- Developed by academia, companies and/ or GPU vendors
- Ranging from very small building blocks to whole applications

- Support for various application areas
- Usually well optimized

- (Performance) Portability can be limited
- Combining frameworks or interfacing own GPU code can be challenging
- Software stability and longevity may be limited

GPU Programming Approaches

Pragma-based approaches (OpenMP, OpenACC)

- Implementation as comments in the source code
- Organized as an open standard
- Works well for codes based on loop nests (and others)

- Easy to integrate into existing code
- No vendor-lock
- Compiler tries to optimize automatically
- Optimization possibilities can be limited
- Compiler support and performance may vary

GPU Programming Approaches

Modern C++

- Implementation directly in C++
- Many parts are already part of the C++ standard
- Works well for codes already using STL algorithms

- Ideal language integration (in extension also IDE support)
- Feature stability

- Limited implementation options – may require algorithm reformulation
- Limited optimization possibilities and potentially lower performance
- Language specification and compiler support is still in development

GPU Programming Approaches

Thrust

- Like the STL – but GPU aware
- Couples with lower-level approaches such as CUDA
- Implementation directly in C++

- All the benefits of modern C++
- Natively GPU aware
- More tuning and optimization options

- Vendor portability may be limited
- May require reformulation of the algorithm to fit existing concepts

GPU Programming Approaches

CUDA, HIP, oneAPI, …

- Implementation using language extensions and API functions

- Exposes all features and capabilities of the hardware
- Full control and maximized optimization potential

- May require substantially more coding effort
- Vendor-specific

GPU Programming Approaches

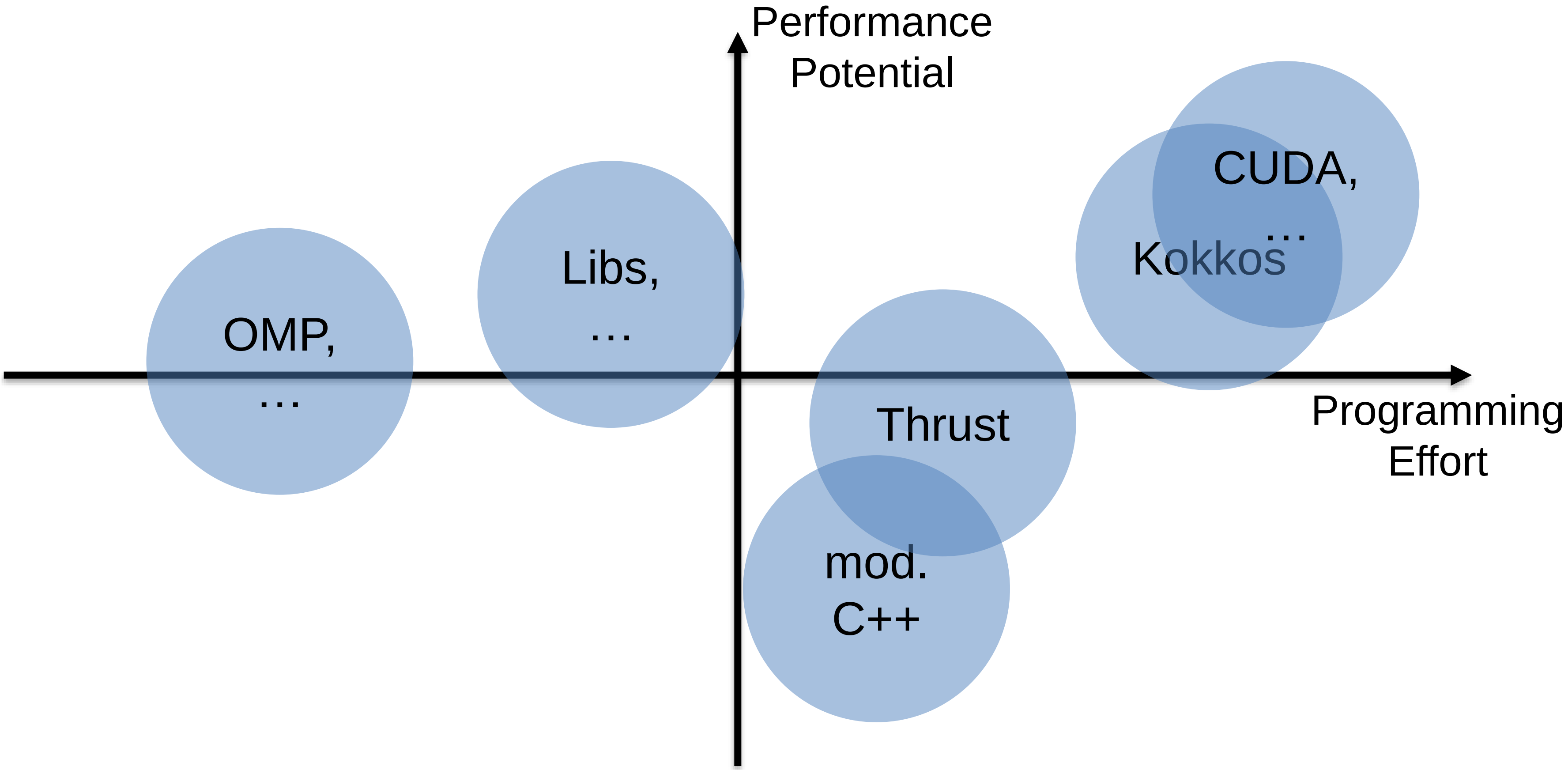
Software layers (Kokkos, …)

- Implementation using a thin abstraction layer

- (Performance) Portability
- More control compared to other approaches

- Requires an abstraction design exposing all vital hardware capabilities
- Performance may be dependent on the quality of the back end

GPU Programming Approaches



GPU Programming Primitives

GPU Programming Primitives

- Data organization
- Parallel kernels
- Next steps

- Programming Challenge

- Access via JupyterHub

- Material is available at
  https://github.com/SebastianKuckuk/gpu-programming-approches

Thank you for your attention