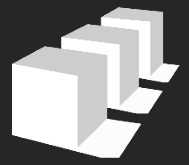
Universidad Zaragoza

1542



Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Instituto Universitario de Investigación en Ingeniería de Aragón Universidad Zaragoza

GPU programming models and languages

Mario Morales Hernández (mmorales@unizar.es)

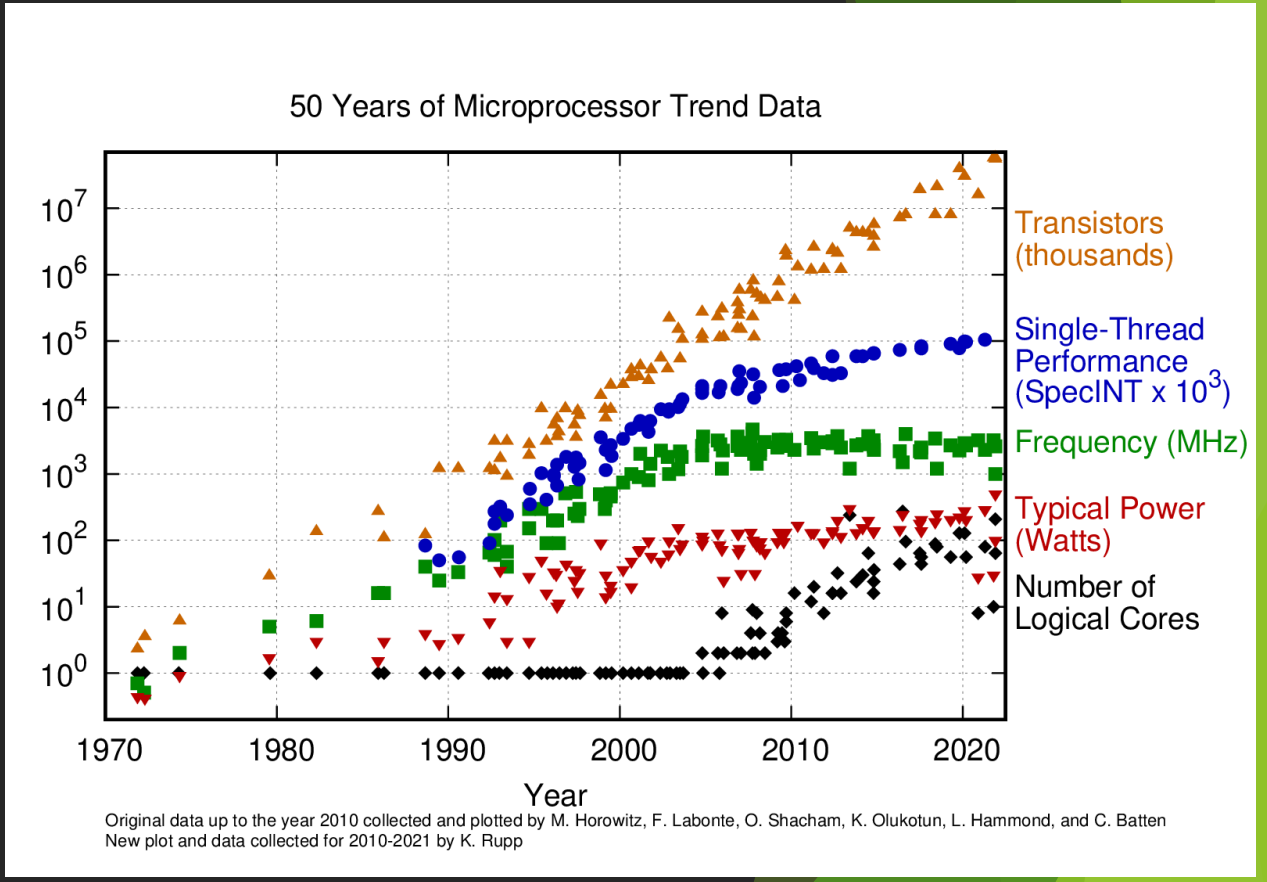
March18 2025

Outline

- Introduction
- Programming models and languages
- CUDA
- OpenACC
- Kokkos

Motivation: Why embrace GPU computing?

- Beyond Moore's law
- Unprecedented speed and efficiency
- Expanding frontiers of research and development
- Cost-effective scalability
- Versatility across fields
- Future-proofing projects



Moving to GPU

"I have my code in CPU (C, C++, Fortran, Python).
How can I run it on GPUs?"

Moving to GPU

Step 1 | Assessment

Identify portions of the code that could benefit from parallelization on a GPU

Step 2 | Porting/Adapting

Choose a suitable GPU programming model/language

Step 3 | Parallel algorithm design

Redesign algorithms to exploit the parallel processing capabilities of GPUs: rethinking data structures, memory access patterns...

Step 4 | Implementation

Translate the code to the chosen programming model

Step 5 | Testing and validation

Ensure correctness to verify that no errors were introduced

Moving to GPU

"Considering the effort to port code to GPUs, does it pay off in terms of performance gains?"

Moving to GPU

Performance gain

Significant speedups possible but depends on problem nature.

Effort consideration

Initial porting and optimization can be intricate and time consuming: redesign algorithms, optimizing memory access and tuning performance. Also, validation and verification.

Trade-offs

Not all code is suited for GPUs. And it requires and investment in learning and development.

Bigger picture

Future-proofing and scalability. Expanding computational horizons.

Programming models and languages

- Diverse ecosystem with multiple vendors and languages

Native

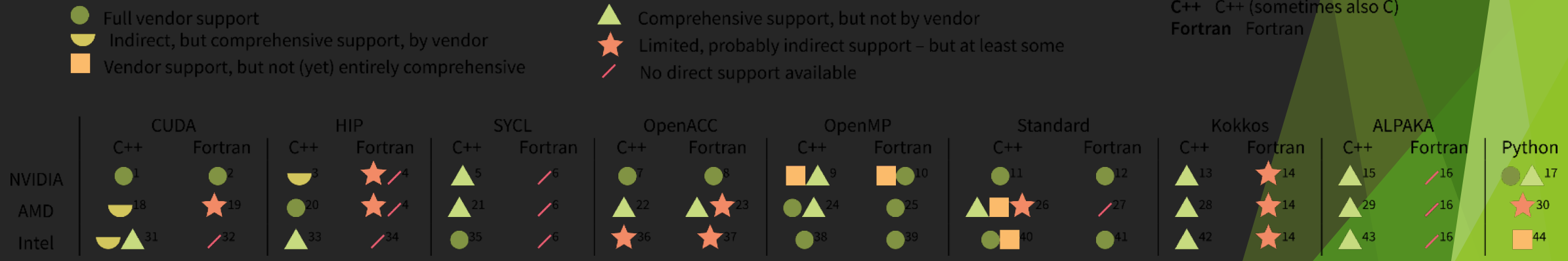
- CUDA (NVIDIA)
- HIP (AMD)
- SYCL (Intel)

Directive-based

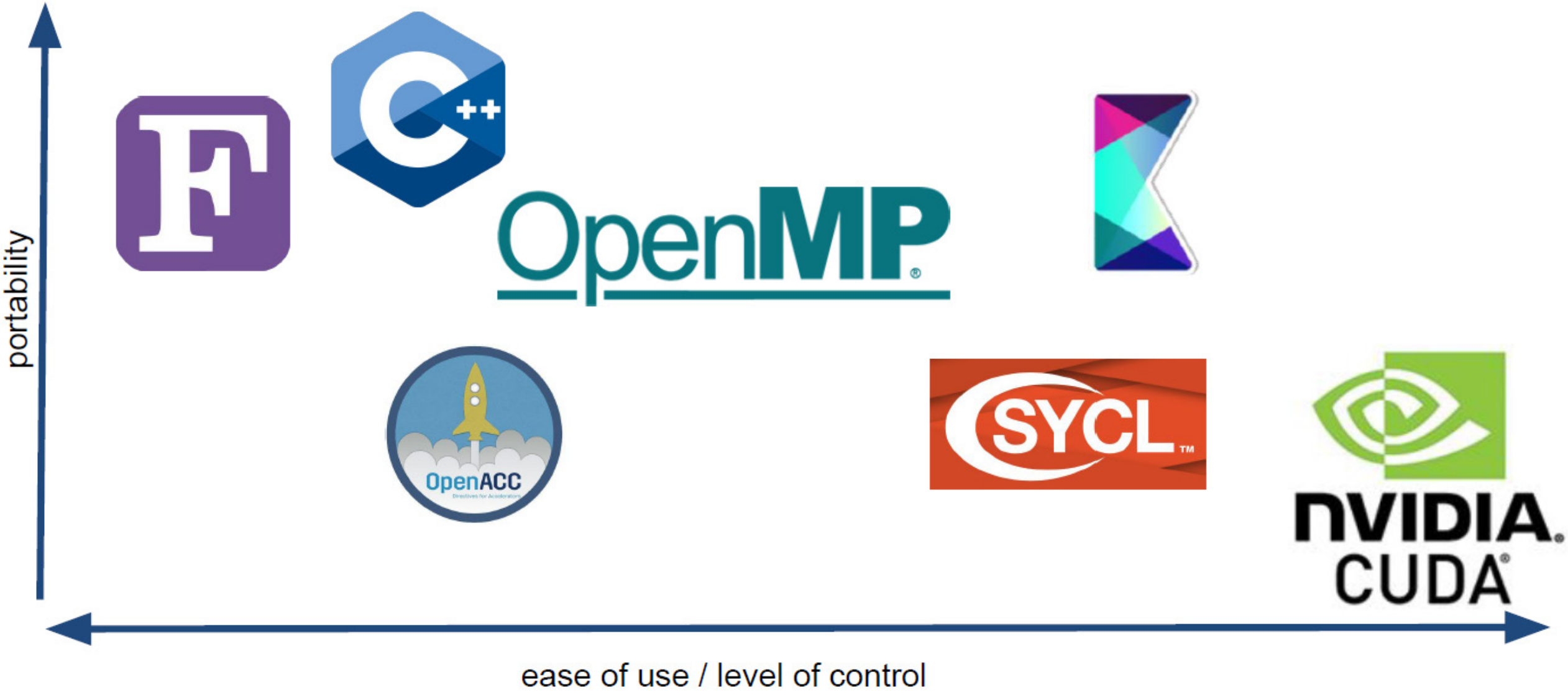
- OpenACC
- OpenMP

Higher-level abstraction

- Kokkos
- RAJA
- ALPAKA



Herten, A. (2023). Many Cores Many Models: GPU Programming Model vs. Vendor Compatibility Overview. arXiv:2309.05445v3 [cs.DC]. Retrieved from https://arxiv.org/abs/2309.05445



Cook. B (2022). GPU Programing Models. NERSC Perlmutter Training

Programming models and languages

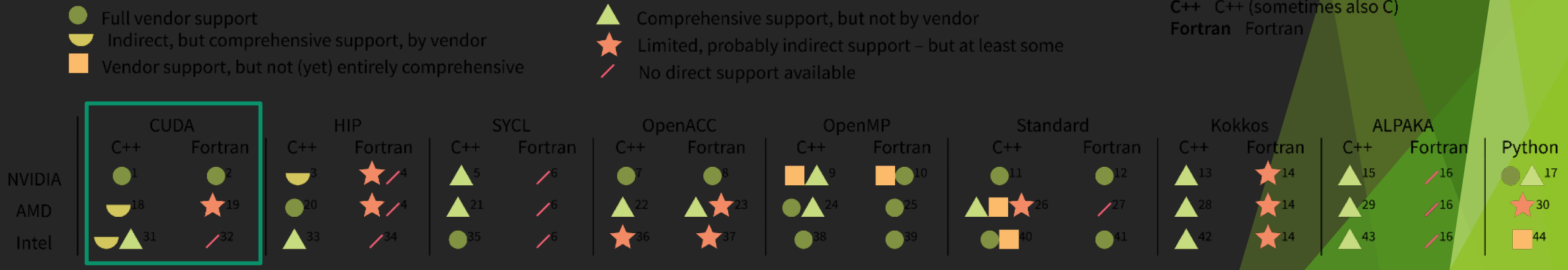
- Diverse ecosystem with multiple vendors and languages

Native
- CUDA (NVIDIA)
- HIP (AMD)
- SYCL (Intel)

1

Directive-based
- OpenACC
- OpenMP

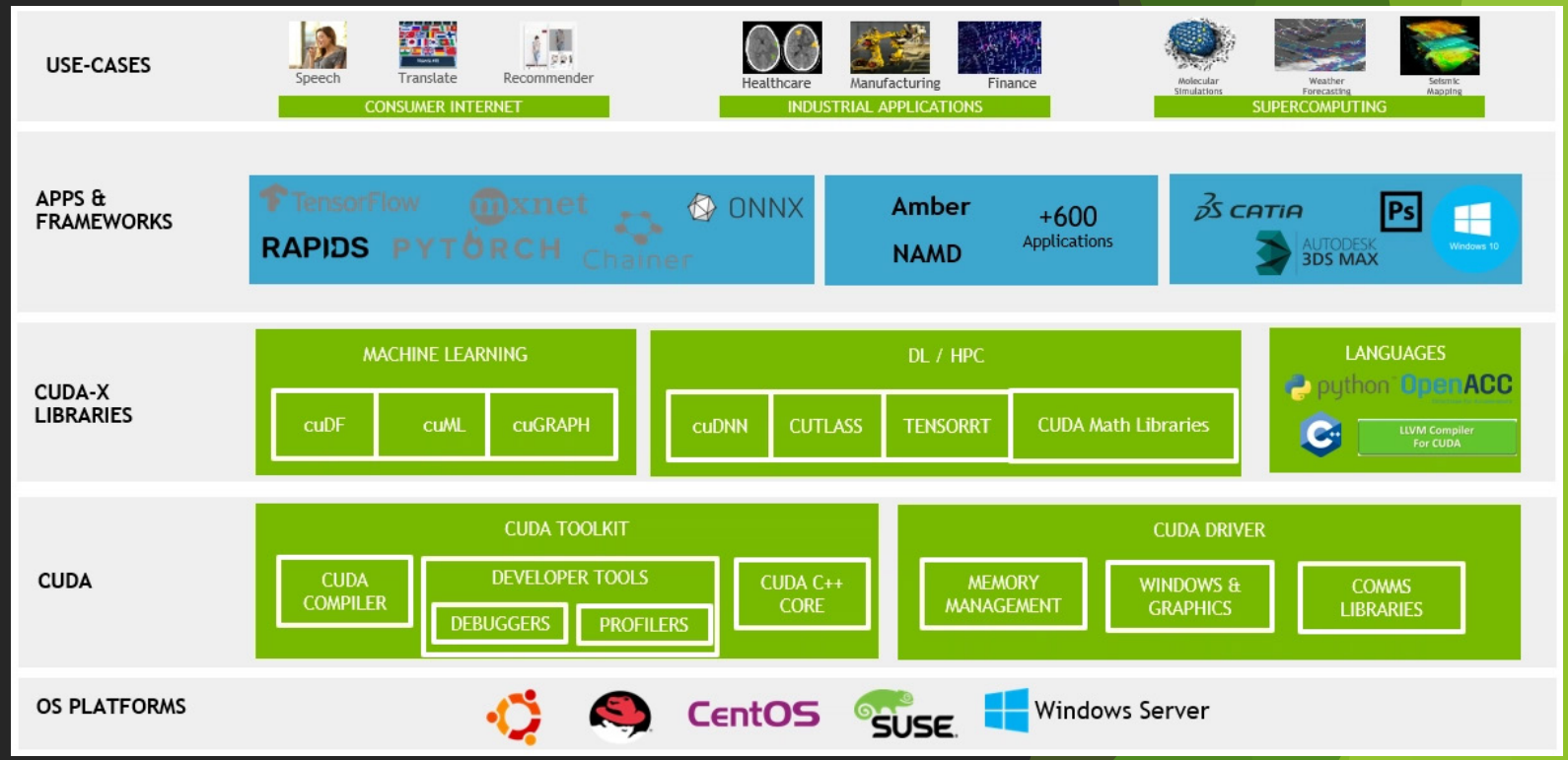
Higher-level abstraction
- Kokkos
- RAJA
- ALPAKA



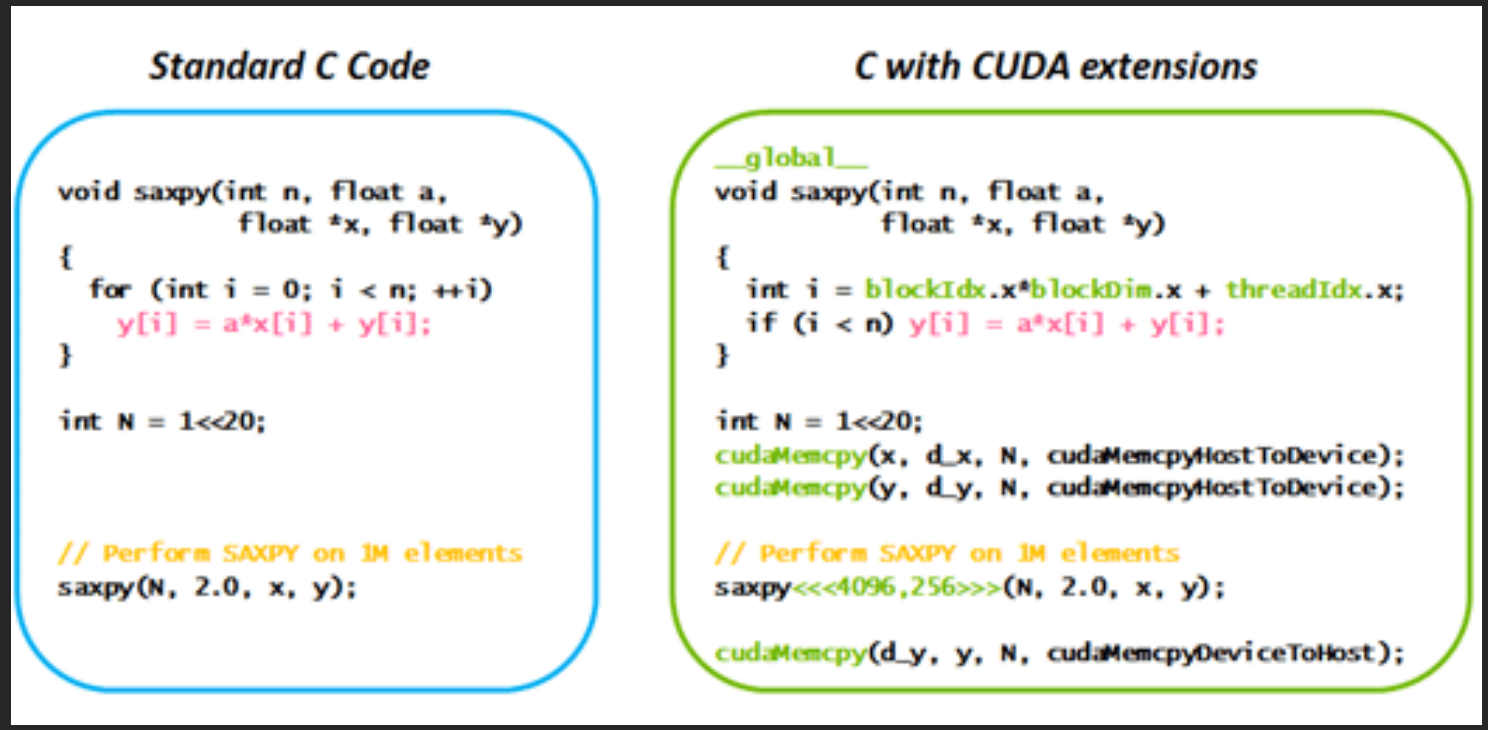
Herten, A. (2023). Many Cores Many Models: GPU Programming Model vs. Vendor Compatibility Overview. arXiv:2309.05445v3 [cs.DC]. Retrieved from https://arxiv.org/abs/2309.05445

What is CUDA

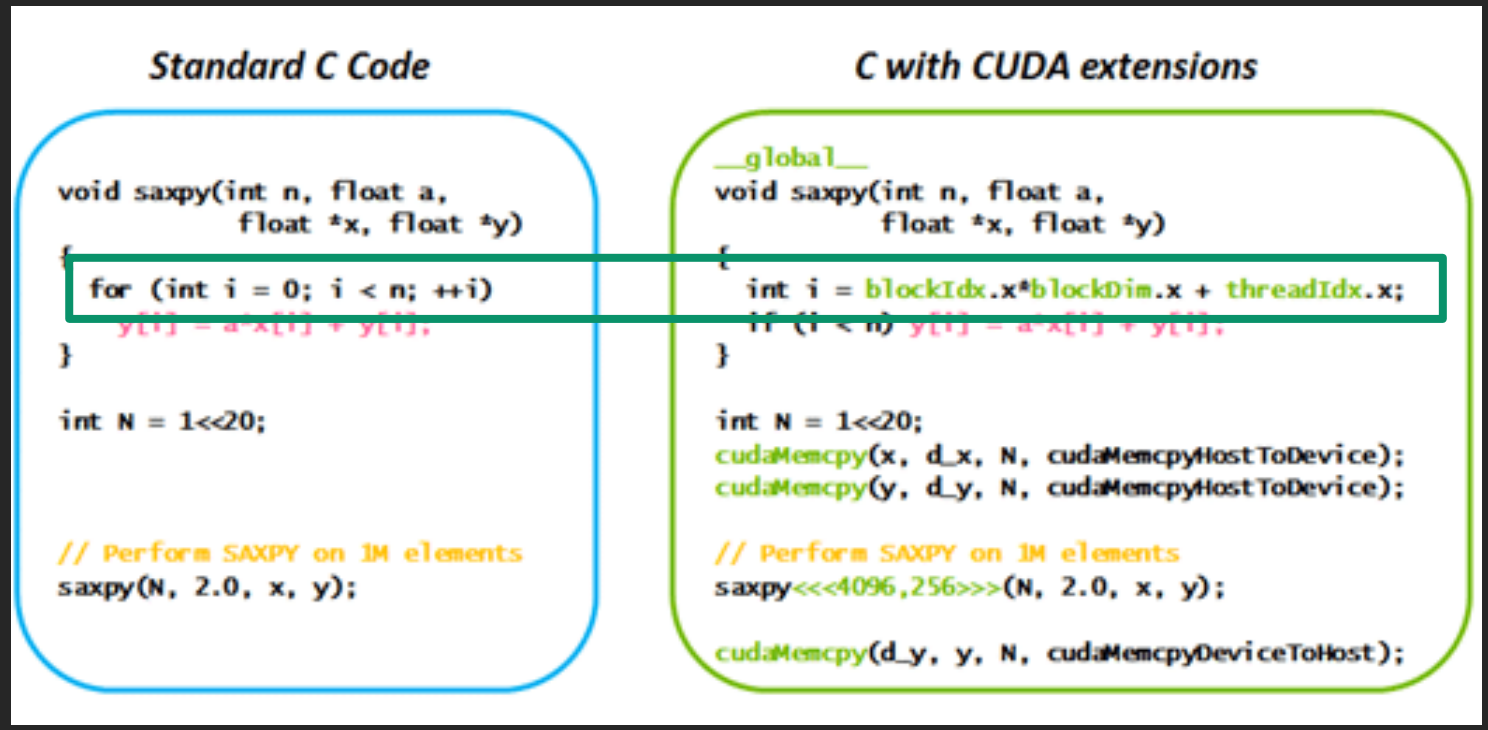
- CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA in 2006.
- It includes a comprehensive development toolkit with compilers, libraries, and debugging tools designed to assist in developing software that runs on NVIDIA GPUs.
- Support for various programming languages (C, C++, Fortran, Python...)
- Wide-ranging use cases: HPC, deep learning, big data ...



CUDA: Implementation of loops

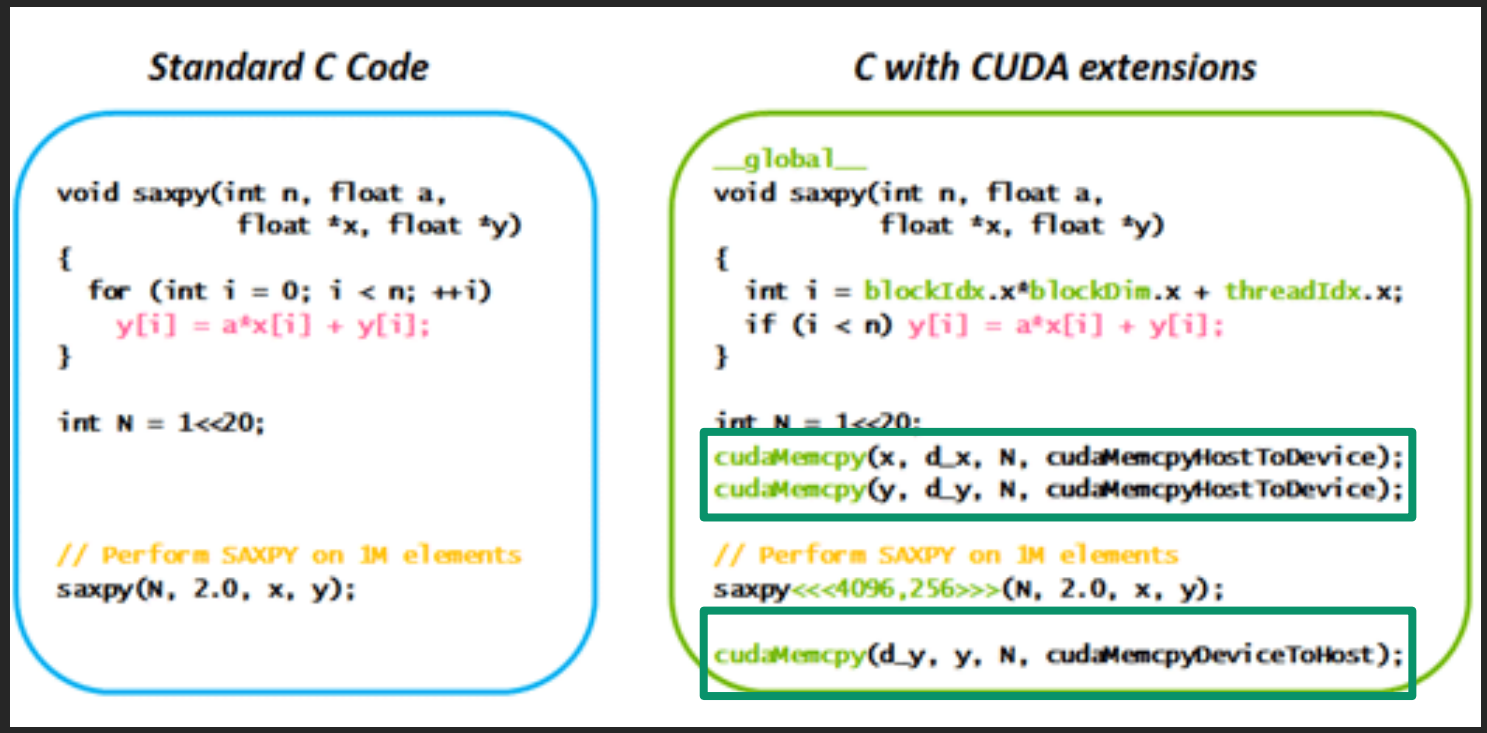


CUDA: Implementation of loops



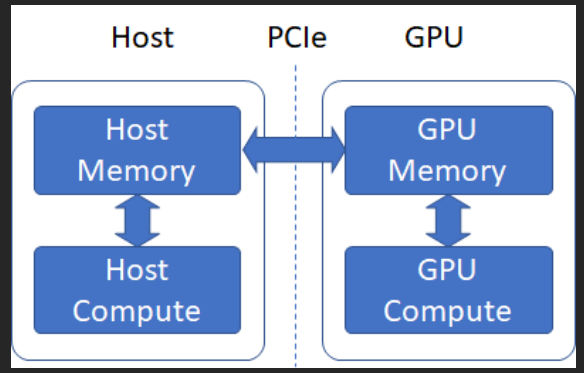
- for **i=1..nElements**
- A[i]=...
- end for

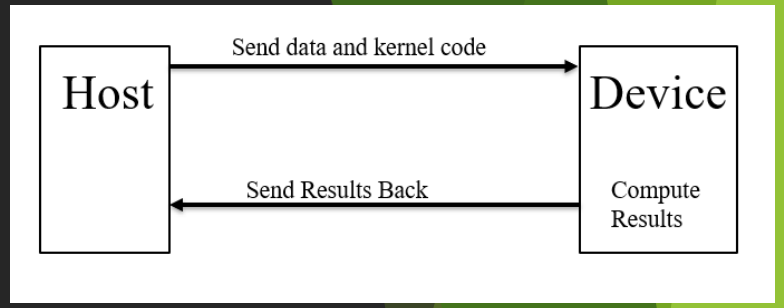
CUDA: Implementation of loops

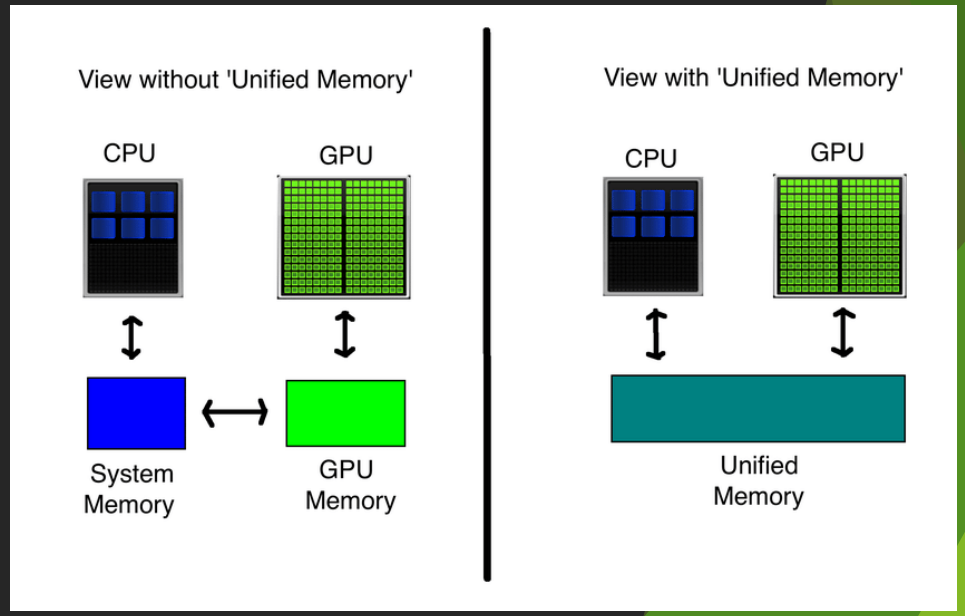


CUDA: Implementation of loops

- **Dynamic Allocation**: Use *cudaMalloc*() for creating space in GPU memory and *malloc*() for CPU memory before data transfer.
- **Memory Copy:** *cudaMemcpy*() transfers data between CPU and GPU. Opt for *cudaMemcpyAsync*() for non-blocking transfers.
- **Unified Memory**: *cudaMallocManaged*() allows data to be automatically shared between CPU and GPU, simplifying code but watch for performance.
- **Key Tips:**
  - Opt for Unified Memory for simplicity; direct allocation and transfer for performance.
  - Minimize data transfers for speed; use asynchronous operations to overlap computation and transfer.







Programming models and languages

- Diverse ecosystem with multiple vendors and languages

Native

- CUDA (NVIDIA)
- HIP (AMD)
- SYCL (Intel)

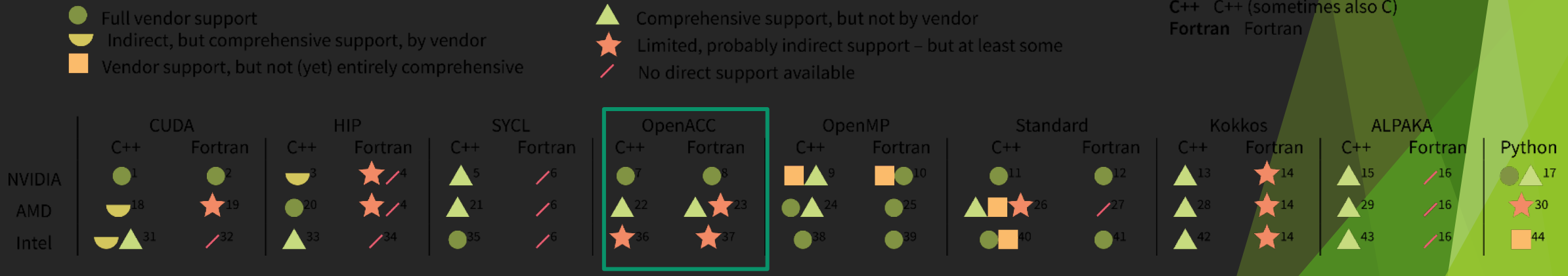
Directive-based

- OpenACC
- OpenMP

2

Higher-level abstraction

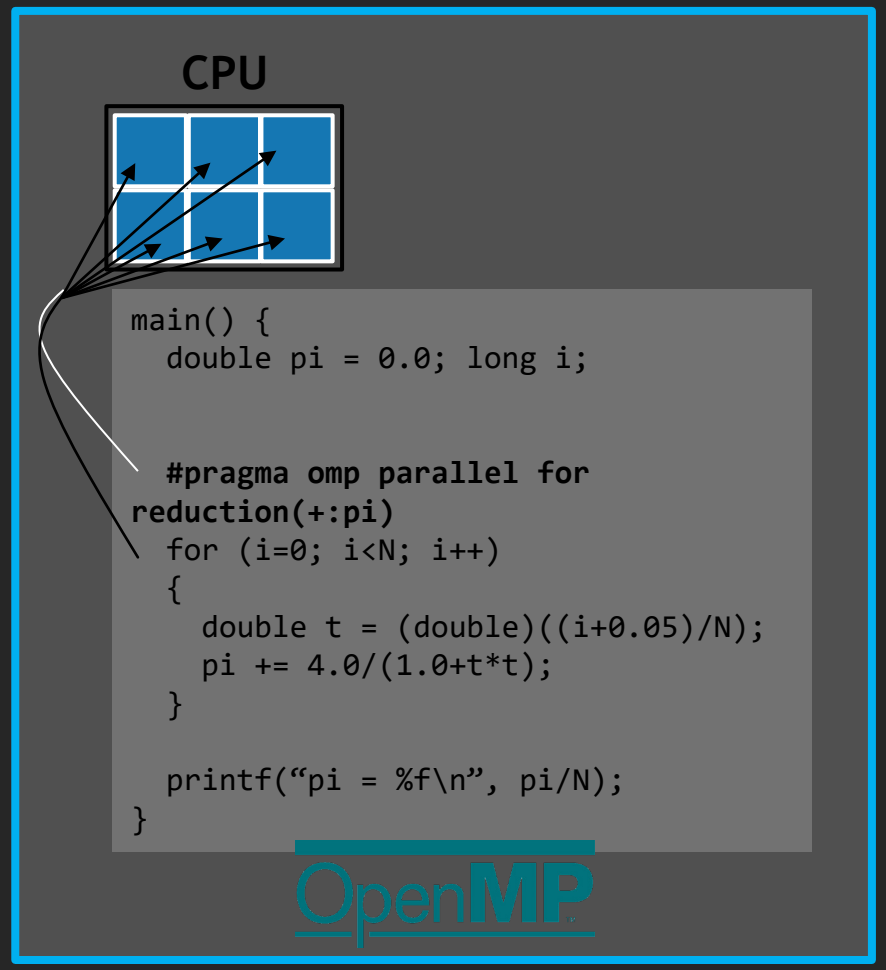
- Kokkos
- RAJA
- ALPAKA

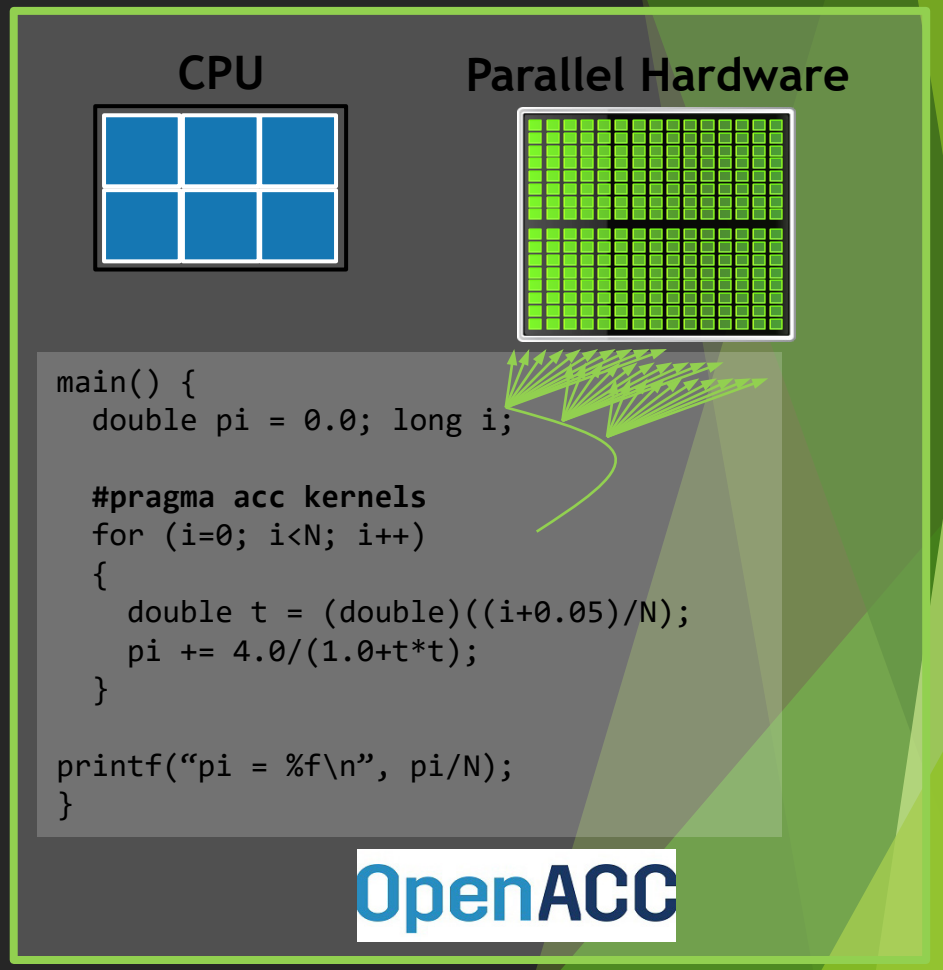


Herten, A. (2023). Many Cores Many Models: GPU Programming Model vs. Vendor Compatibility Overview. arXiv:2309.05445v3 [cs.DC]. Retrieved from https://arxiv.org/abs/2309.05445

OpenACC: Familiar to OpenMP programmers

- A directive-based approach to easily accelerate code on GPUs and CPUs with minimal changes.





What is OpenACC

- A directive-based approach to easily accelerate code on GPUs and CPUs with minimal changes.

- **Simple to Use**: Just add directives to code sections you want to run in parallel—no need for detailed GPU programming knowledge
- **Works on Various Devices**: Designed for multi-platform compatibility, including NVIDIA and AMD GPUs.
- **Gradual Optimization**: Start optimizing critical code sections first for noticeable performance boosts.
- **Compiler Friendly**: Supported by multiple compilers for straightforward code compilation and optimization.
- Ideal for developers seeking performance gains without deep diving into complex GPU code

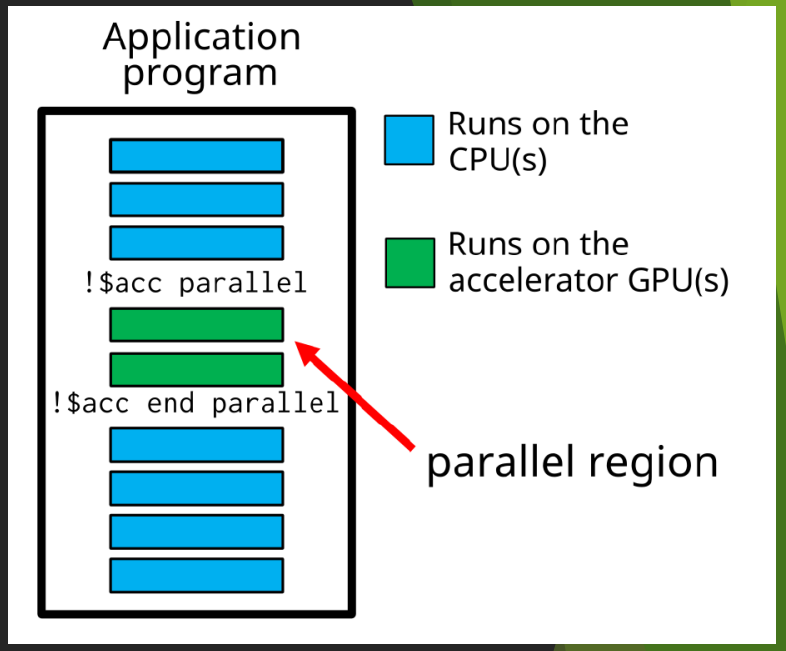
Add Simple Compiler Directive

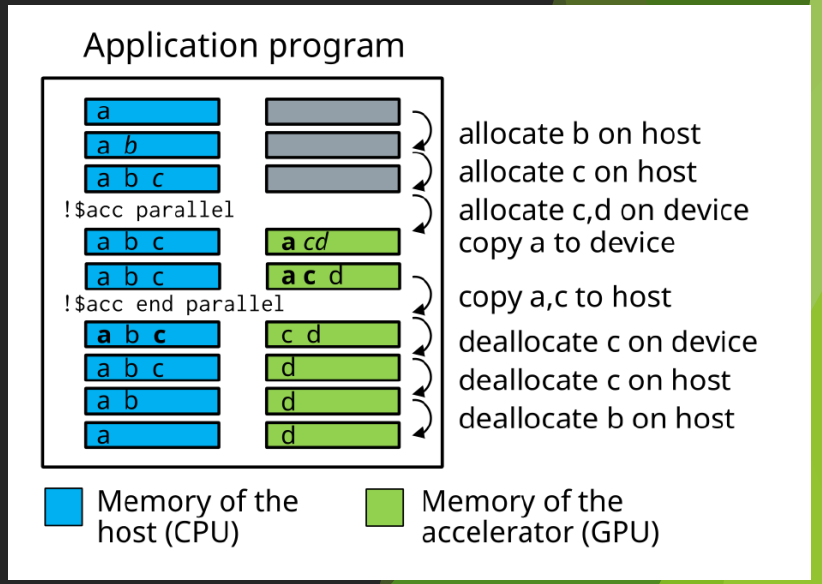
```
main()
 {
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

OpenACC execution model

1. Program runs on the host CPU
2. Host offloads compute-intensive regions (kernels) and related data to the accelerator GPU
3. Compute kernels are executed by the GPU

OpenACC exposes the separate memories through data environment that defines the memory management and needed copy operations





OpenACC: compute directives and data environment

- OpenACC uses compiler directives for defining compute regions (and data transfers) that are to be performed on a GPU
- Important constructs: parallel, kernels, data, loop, update, host_data, wait
- Often used clauses: if (condition), async(handle)

	sentinel	construct	clauses
C/C++	#pragma acc	kernels	copy(data)
Fortran	!$acc	kernels	copy(data)

It supports devices which either share memory with or have a separate memory from the host

Constructs and clauses for:

- defining the variables on the device
- transferring data to/from the device

C/C++

```
int a[100], d[3][3], i, j;

#pragma acc data copy(a[0:100])
{
    #pragma acc parallel loop present(a)
    for (int i=0; i<100; i++)
        a[i] = a[i] + 1;
    #pragma acc parallel loop \
          collapse(2) copyout(d)
    for (int i=0; i<3; ++i)
        for (int j=0; j<3; ++j)
            d[i][j] = i*3 + j + 1;
}
```

Fortran

```
integer a(0:99), d(3,3), i, j

!$acc data copy(a(0:99))
  !$acc parallel loop present(a)
  do i=0,99
     a(i) = a(i) + 1
  end do
  !$acc end parallel loop
  !$acc parallel loop collapse(2) copyout(d)
  do j=1,3
     do i=1,3
        d(i,j) = i*3 + j + 1
     end do
  end do
  !$acc end parallel loop
!$acc end data
```

OpenACC vs. CUDA/HIP

Why OpenACC and not CUDA/HIP?

- Easier to work with
- Porting of existing software requires less work
- Same code can be compiled to CPU and GPU versions easily

Why CUDA/HIP and not OpenACC?

- Can access all features of the GPU hardware
- More optimization possibilities

Programming models and languages

- Diverse ecosystem with multiple vendors and languages

Native

- CUDA (NVIDIA)
- HIP (AMD)
- SYCL (Intel)

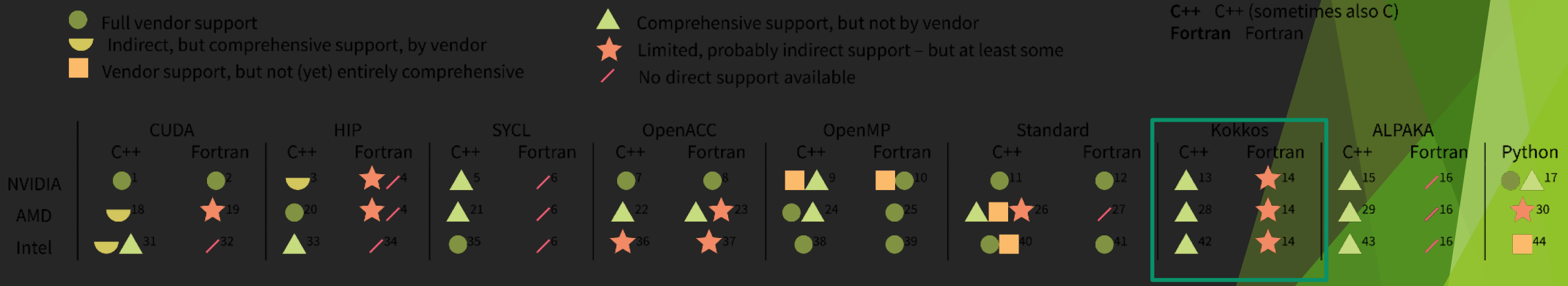
Directive-based

- OpenACC
- OpenMP

Higher-level abstraction

- Kokkos
- RAJA
- ALPAKA

3



Herten, A. (2023). Many Cores Many Models: GPU Programming Model vs. Vendor Compatibility Overview. arXiv:2309.05445v3 [cs.DC]. Retrieved from https://arxiv.org/abs/2309.05445

What is Kokkos

- A programming model for performance portability

Performance Portability

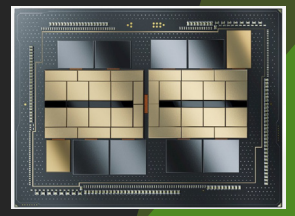
"Achieving a consistent ratio of the actual time to solution to either the best-known or the theoretical best time to solution on each platform with minimal platform specific code required."

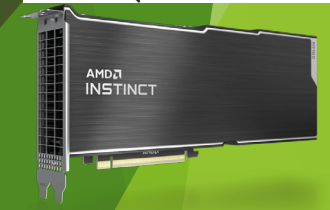
Source: https://performanceportability.org/

Future-proofing

"To design software, a computer, etc. so that it can still be used in the future, even when technology changes."

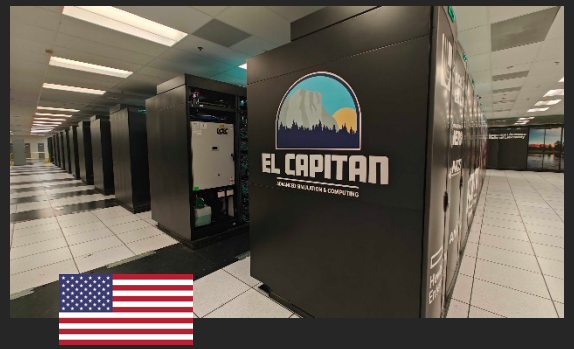
Source: Cambridge Dictionary





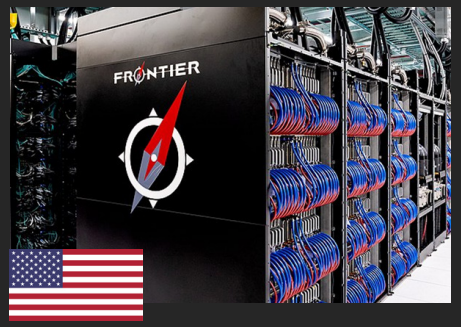
- Lifetime of software is much longer than lifetime of hardware
- Hardware technology changes, nowadays into a variety of new solutions
- For scientists and developers the code needs to be readable, maintainable, and compact, avoiding code duplications and hiding technicalities (separation of concerns)
- Use your code on CPUs and GPUs (and more), form laptops to workstations, clusters and HPC systems.
- Avoid re-coding and multiple hardware-specific code bases.

Exascale and performance-portability

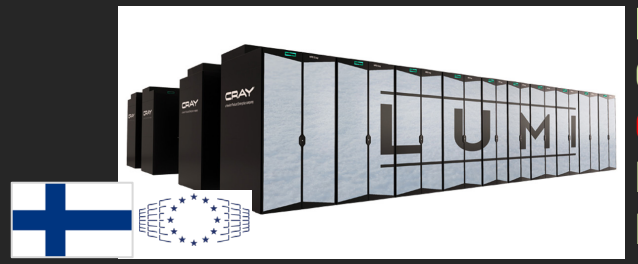


El Capitán@LLNL (USAn)
CPU: AMD Epyc
GPU: AMD Instinct
Perf. peak: 1,74 Eflops
Deployment: 2024
#1 in TOP500

Fugaku @ Reiken (Japan)
CPU: ARM
GPU: none
Perf. Peak: 0.44 Eflops
Deployment: 2021
#6 in TOP500



Frontier@ORNL (USA)
CPU: AMD EPYC
GPU: AMD Instinct
Perf. peak: 1.3 Eflops
Deployment: 2022
#2 in TOP500



Lumi Consortium (Finland)
CPU: AMD EPYC
GPU: AMD Instinct
Perf peak: 0.38 Eflops
Deployment: 2021/2022
#8 in TOP500

Aurora@Argonne (USA)
CPU: Intel Xeon Sapphire Rapids
GPU: Intel Xe
Perf. peak: 1,01 Eflops
Deployment: 2023
#3 in TOP500

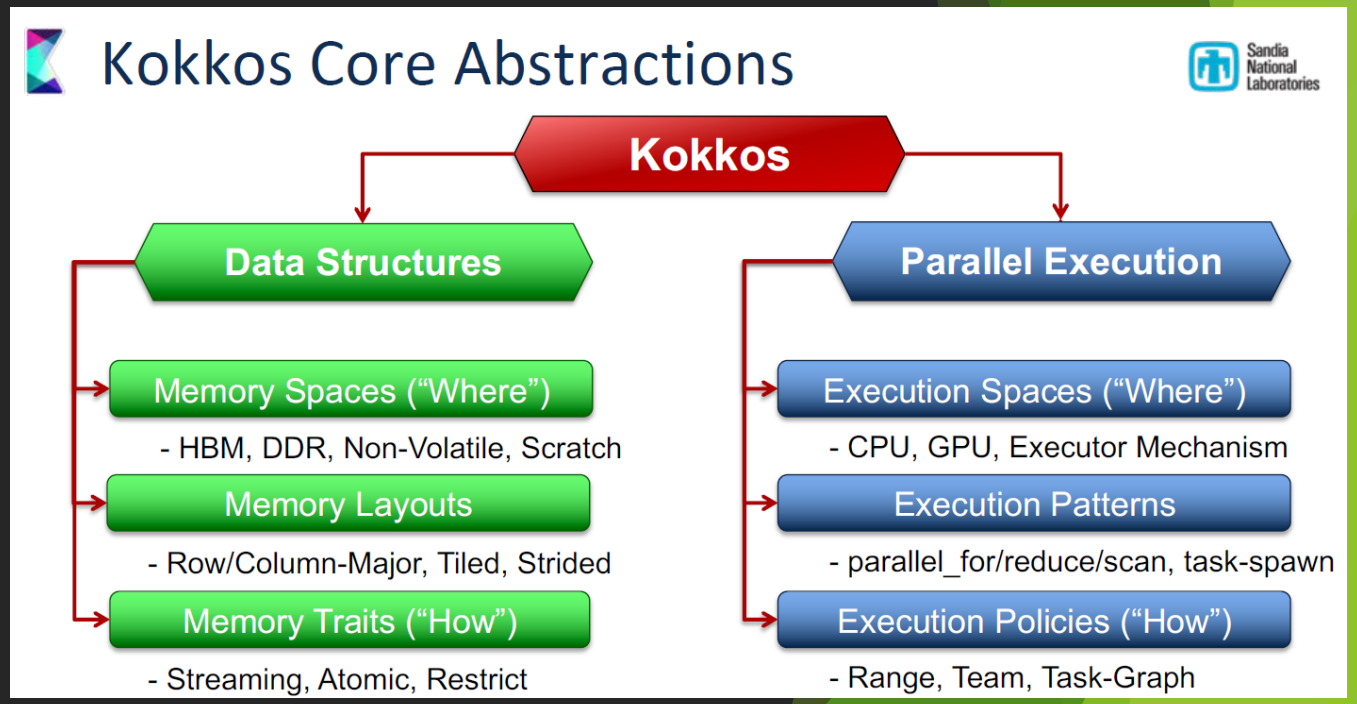
Leonardo@CINECA (Italy)
CPU: Intel Xeon Ice Lake & Intel Xeon Sapphire
GPU: Nvidia A100
Perf. peak: 0.24 Eflops
Deployment: 2022
#9 in TOP500

November 2024: **1/10** no GPUs, **3/10** Nvidia GPUs, **5/10** AMD GPUs, **1/10** Intel GPUs
November 2023: **1/10** no GPUs, **6/10** Nvidia GPUs, **2/10** AMD GPUs, **1/10** Intel GPUs
November 2022: **3/10** no GPUs, **5/10** Nvidia GPUs, **2/10** AMD GPUs, **0/10** Intel GPUs
November 2021: **3/10** no GPUs, **7/10** Nvidia GPUs, **0/10** AMD GPUs, **0/10** Intel GPUs

What is Kokkos

- Ensuring performance portability and future proofing
  - Open Source C++ library
  - Developed at Sandia National Lab (USA).
  - Community project
  - Programming model that targets all major HPC platforms
  - Avoid the cost of porting codes
  - It combines data structures with parallel execution
    - Memory spaces and layouts
    - Execution patterns (for, reduction, atomic…)
    - Data allocation, transfer and execution spaces
  - Math libraries and tools based on Kokkos
    - Profiling
    - Debugging
  - Currently supported backends and devices (03.2024):
    - OpenMP, CUDA, HIP
    - Intel, Nvidia, ARM, IBM, AMD

https://github.com/kokkos/kokkos



Kokkos: typical computational kernel

Serial C/C++ (single CPU execution space)

```
inline void computeNewState(State &state , const Domain &dom, const SourceSinkData &ss) {
for (int j=0; j<dom.ny; j++) {
for (int i=0; i<dom.nx; i++) {
  int ii = dom.getHaloExtension(i,j,dom.nx);

  real z=state.z(ii);
  real hold=state.h(ii);
  real huold=state.hu(ii);
  real hvold=state.hv(ii);
  bool nodata=state.isnodata(ii);

  // lots of computationally intensive code

}
}
```

Kokkos: typical computational kernel

Serial C/C++ (single CPU execution space)

```
inline void computeNewState(State &state , const Domain &dom, const SourceSinkData &ss) {
for (int j=0; j<dom.ny; j++) {
for (int i=0; i<dom.nx; i++) {
  int ii = dom.getHaloExtension(i,j,dom.nx);

  real z=state.z(ii);
  real hold=state.h(ii);
  real huold=state.hu(ii);
  real hvold=state.hv(ii);
  bool nodata=state.isnodata(ii);

  // lots of computationally intensive code

}
}
```

Kokkos C++

```
inline void computeNewState(State &state , const Domain &dom, const SourceSinkData &ss) {

  Kokkos::parallel_for( dom.nCellDomain , KOKKOS_LAMBDA (int iGlob) {
  int ii = dom.getIndex(iGlob);

  real z=state.z(ii);
  real hold=state.h(ii);
  real huold=state.hu(ii);
  real hvold=state.hv(ii);
  bool nodata=state.isnodata(ii);

  // lots of computationally intenstive code

  })
```

Kokkos: typical computational kernel

Serial C/C++ (single CPU execution space)

```
inline void computeNewState(State &state , const Domain &dom, const SourceSinkData &ss) {
for (int j=0; j<dom.ny; j++) {
for (int i=0; i<dom.nx; i++) {
  int ii = dom.getHaloExtension(i,j,dom.nx);

  real z=state.z(ii);
  real hold=state.h(ii);
  real huold=state.hu(ii);
  real hvold=state.hv(ii);
  bool nodata=state.isnodata(ii);

  // lots of computationally intensive code

}
}
```

This code works for single CPUs, threaded CPUs (OpenMP) and GPUs (CUDA)

Implementing Kokkos is minimally invasive (if parallelism is already well-exposed).

Kokkos C++

```
inline void computeNewState(State &state , const Domain &dom, const SourceSinkData &ss) {

 Kokkos::parallel_for( dom.nCellDomain , KOKKOS_LAMBDA (int iGlob) {
 int ii = dom.getIndex(iGlob);

 real z=state.z(ii);
 real hold=state.h(ii);
 real huold=state.hu(ii);
 real hvold=state.hv(ii);
 bool nodata=state.isnodata(ii);

 // lots of computationally intenstive code

 })
```

Serial code is executed in host, parallel execution space determined at compilation or as a policy

Kokkos views

- Multidimensional array abstraction to simplify data management across diverse architectures

Defining all arrays as Kokkos Views

```
#include <Kokkos_Core.hpp>

typedef double          real;
typedef unsigned long ulong;
typedef unsigned int  uint;

#ifdef __NVCC__
  typedef Kokkos::View<real* ,Kokkos::Device<Kokkos::Cuda,Kokkos::CudaUVMSpace>> realArr;
  typedef Kokkos::View<int*  ,Kokkos::Device<Kokkos::Cuda,Kokkos::CudaUVMSpace>> intArr;
  typedef Kokkos::View<bool* ,Kokkos::Device<Kokkos::Cuda,Kokkos::CudaUVMSpace>> boolArr;
#else
  typedef Kokkos::View<real* ,Kokkos::LayoutRight> realArr;
  typedef Kokkos::View<int*  ,Kokkos::LayoutRight> intArr;
  typedef Kokkos::View<bool* ,Kokkos::LayoutRight> boolArr;
#endif
```

Unified memory visible for host and device

No memory space is provided: the data resides in the default memory space of the default execution space (in this case host).

These are convenient general definitions, but can be fine-grained for each specific need

Kokkos vs. CUDA and OpenACC

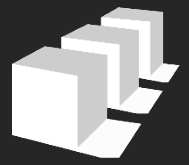
- Avoid deep copies between CPU and GPU (CUDA, OpenACC)
- Avoid re-writing code: human mistakes, reproducibility, readability (CUDA, OpenACC)
- Require target-specific adaptations of data structures to achieve good performance on both CPUs and GPUs (OpenACC)
- Portability between different architectures (CUDA)
- Acceptable performance for both CPU and GPUs (CUDA, OpenMP)

criterion	OpenACC (CPU+GPU)	CUDA (GPU)	Kokkos (CPU+GPU)
code clarity	high	low	medium
productivity	medium	low	medium
portability	medium	low	high
performance	high	high	high

Artigues, V. et al., Evaluation of performance portability frameworks for the implementation of a particle-in-cell code. Concurrency Computat Pract Exper. 2020; 32:e5640.

Universidad Zaragoza

1542



Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Instituto Universitario de Investigación en Ingeniería de Aragón Universidad Zaragoza

GPU programming models and languages

Mario Morales Hernández (mmorales@unizar.es)

March18 2025