3rd NatESM Training Workshop

Two ESM Experiences of Performance Portability: Born Into It or Maturing Into It

Daniel **Caviedes-Voullième**
SDL Terrestrial Systems (JSC/FZJ)

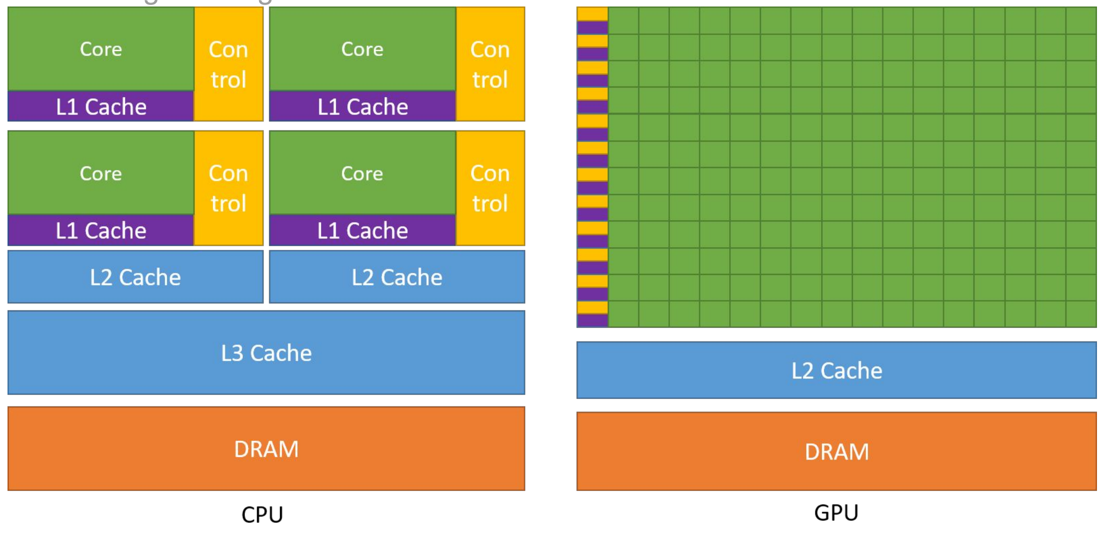
06.11.2024

Take home messages

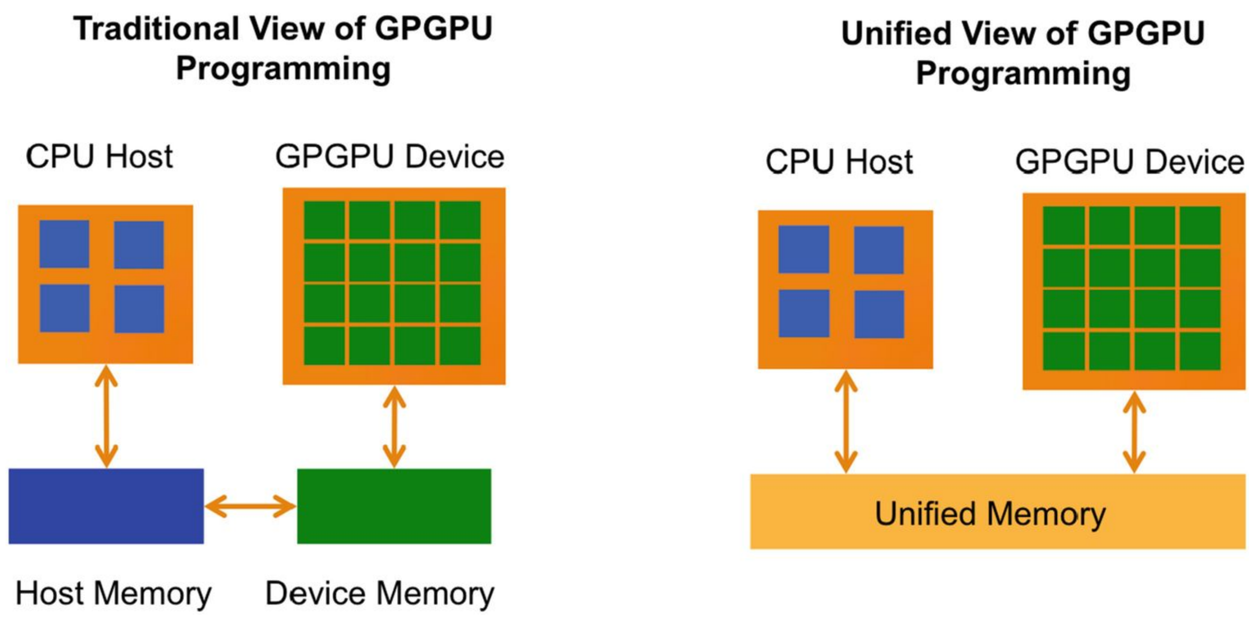
- There is a growing variety of hardware which requires a **variety of programming models**
- Most of ESM codes still struggle with **legacy effects**
- **No one-size-fits-all** solution available, but good solutions are available
- Porting in some way is likely unavoidable, **get started!**
- **Key question**: how to write parallel code and how to allocate memory in a hardware/vendor-agnostic way? **Two ideas in this talk.**
- **Where to start**: port (some part of) your code
- Manage your **expectations:** you will get speed up, don't expect optimal performance (but that's ok)
- Assume you will have to port again: **separate and abstract**
- We should **learn the lesson**: develop assuming disruptive changes may happen again

The problem we are now all too familiar with…

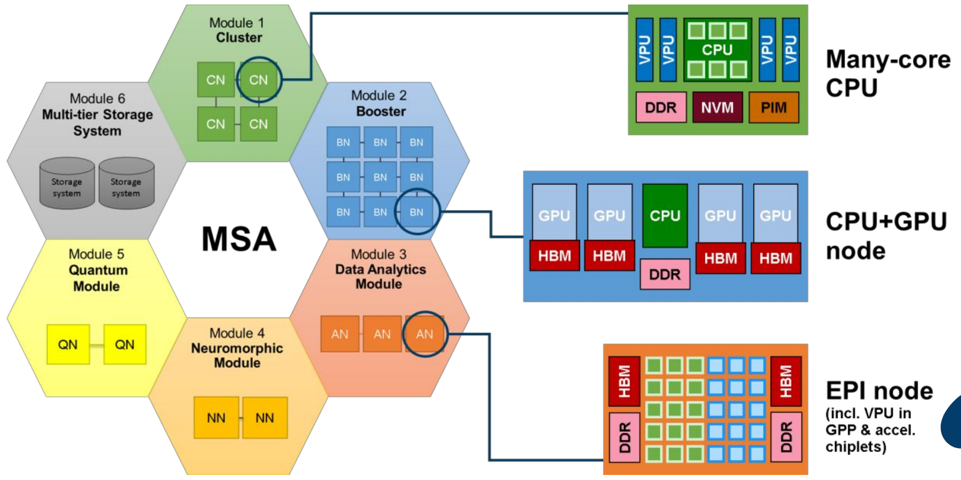
Nvidia Programming Guide



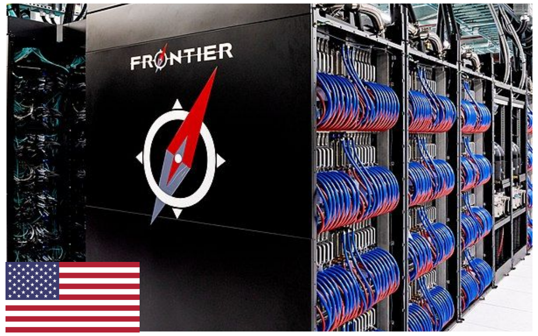
Pallipuram and Gao. "The realm of Graphical Processing Unit (GPU) Computing. Topics in Parallel and Distributed Computing. 2018.



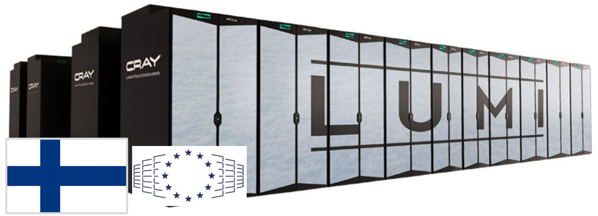
- We cannot rely on in an increase of computational power simply because of faster clocks
- Miniaturisation limit, high energy usage, efficiency
- Devices with different architecture
- Programmer cannot expect compiler to figure out everything alone
- Optimisation still necessary, but now also significant porting efforts, and optimisation again
- Modularity: best hardware and software for the job


Module 1 Cluster
CN
Module 2 Booster
BN
Module 3 Data Analytics Module
AN
Module 4 Neuromorphic Module
NN
Module 5 Quantum Module
QN
Module 6 Multi-tier Storage System
Storage system
MSA
VPU
CPU
DDR
NVM
PIM
Many-core CPU
GPU
HBM
CPU+GPU node
EPI node (incl. VPU in GPP & accel. chiplets)

The *other* problem: evolving HPC ecosystems



Frontier@ORNL (USA)
CPU: AMD EPYC
GPU: **AMD Instinct**
Perf. peak: 1.1 Eflops
Deployment: 2022
#1 in TOP500

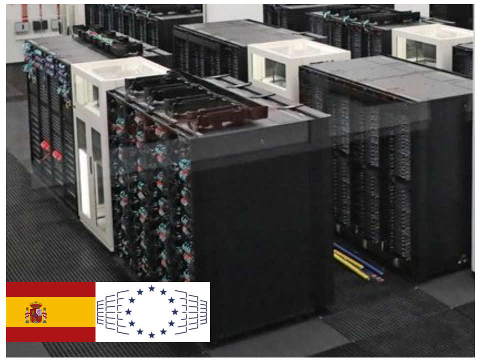


Lumi Consortium (Finland)
CPU: AMD EPYC
GPU: **AMD Instinct**
Perf peak: 0.38 Eflops
Deployment: 2021/2022
#5 in TOP500

Aurora@Argonne (USA)
CPU: Intel Xeon Sapphire Rapids
GPU: **Intel Xe**
Perf. peak: 0.58 Eflops
Deployment: 2023
#2 in TOP500

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

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

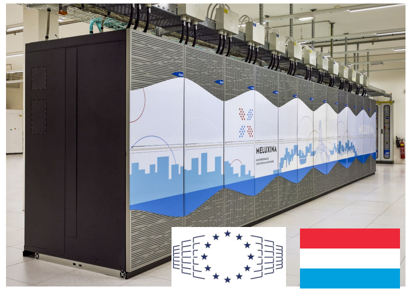


MareNostrum5@BSC (Spain)
CPU: Intel Xeon
GPU: **Nvidia H100**
Perf. peak: 0.14 Eflops
Deployment: 2023
#8 in TOP500

November 2023 (May 2024): **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

JÜLICH Forschungszentrum

EuroHPC: European infrastructure for all

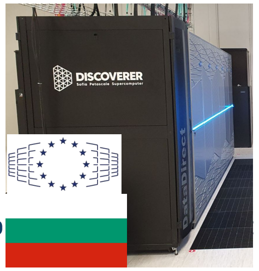


MeluXina (Luxembourg)
CPU: AMD EPYC
GPU: **Nvidia A100**
Performance peak: 10-15 Pflops
Deployment: 2021

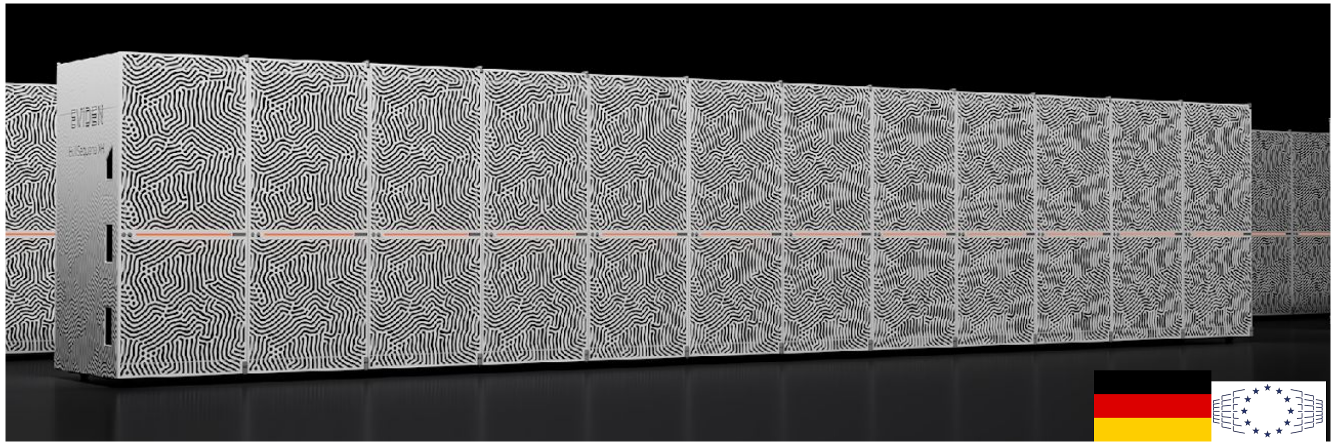
Deucalion (Portugal)
CPU: ARM, AMD EPYC
GPU: **Nvidia A100**
Performance peak: 10 Pflops
Deployment: 2021/2022

Karolina (Czech Rep.)
CPU: AMD 7H12
GPU: **Nvidia A100**
Performance peak: 15.7 Pflops
Deployment: 2021

Vega (Slovenia)
CPU: AMD EPYC
GPU: **Nvidia A100**
Performance peak: 10.1 Pflops
Deployment: 2021



Discoverer (Bulgaria)
CPU: AMD EPYC
GPU: **None**
Performance peak: 6 Pflops
Deployment: 2021



JUPITER@JSC (Germany)
CPU: Arm
GPU: **Nvidia GH200 (Grace Hopper)**
Perf. peak: >1 Eflops
Deployment: 2024/2025
#1? in TOP500
(JEDI is #189, and #1 in Green500)

Memb Association

What is performance-portability and future-proofing?

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

- **Software lifetime** is much longer than hardware lifetime
- **Evolving hardware** technology into a variety of new solutions offered by many vendors
- For scientists and developers the **code needs to be readable**, maintainable, and compact, avoiding code duplications and hiding technicalities (separation of concerns)
- **Avoid re-coding** and/or multiplication of hardware-specific code.
- **Portability**: CPUs and GPUs (and more), form workstations to clusters and HPC systems.

How to achieve performance portability?

- Hardware-specific languages
- Directives
- Portability frameworks
- Domain specific languages
- Libraries
- Very high level languages

NVIDIA CUDA

ROCM ON

kokkos

alpaka

OpenACC

python

julia

PETSc

GridTools

JÜLICH Forschungszentrum

How to choose from this zoo?

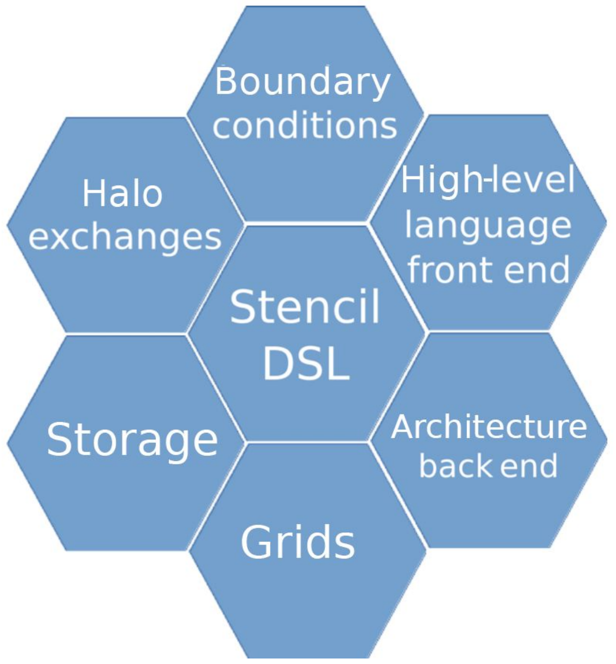
Some common aspects:

- Abstraction to high(er) levels
- Avoid deep copies between CPU and GPU
- Avoid re-writing code: human mistakes, reproducibility, readability
- Safe code: definition of private, shared variables for each subroutine
- Portability between different architectures
- Acceptable performance for both CPU and GPUs

criterion	OpenMP (CPU)	CUDA (GPU)	Kokkos (CPU+GPU)
code clarity	high	low	medium
productivity	high	low	medium
portability	low	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.

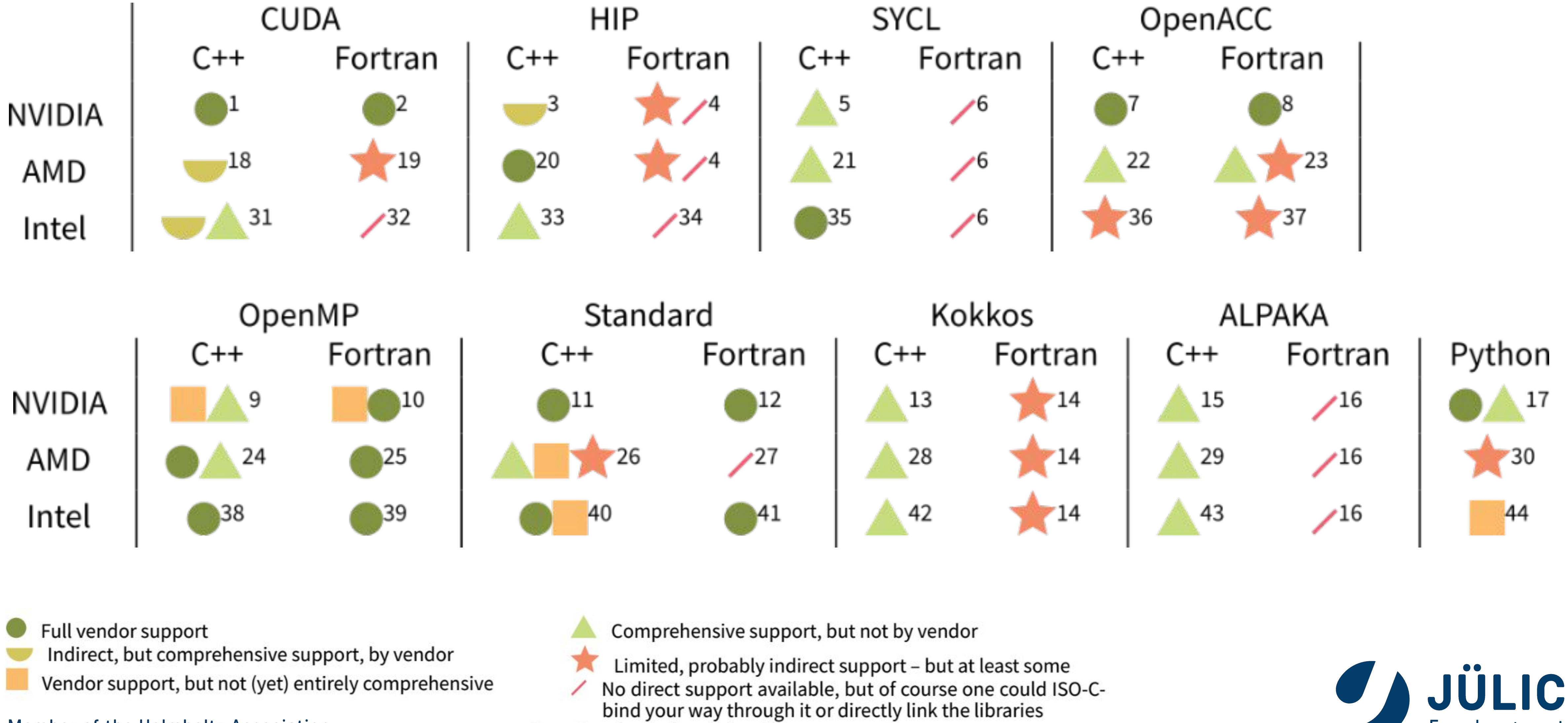
GridTools



Lawrence et al., Crossing the chasm: how to develop weather and climate models for next generation computers? Geoscientific Model Development. 2018

How to choose from this zoo?

Andreas Herten.
GPU Vendor/Programming Model Compatibility Table.
JSC Accelerating Devices Lab Blog.
https://doi.org/10.34732/xdvblg-r1bvif



	CUDA		HIP		SYCL		OpenACC	
	C++	Fortran	C++	Fortran	C++	Fortran	C++	Fortran
NVIDIA	● 1	● 2	◒ 3	★ / 4	▲ 5	/ 6	● 7	● 8
AMD	◒ 18	★ 19	● 20	★ / 4	▲ 21	/ 6	▲ 22	▲ ★ 23
Intel	◒ ▲ 31	/ 32	▲ 33	/ 34	● 35	/ 6	★ 36	★ 37

	OpenMP		Standard		Kokkos		ALPAKA		
	C++	Fortran	C++	Fortran	C++	Fortran	C++	Fortran	Python
NVIDIA	■ ▲ 9	■ ● 10	● 11	● 12	▲ 13	★ 14	▲ 15	/ 16	● ▲ 17
AMD	● ▲ 24	● 25	▲ ■ ★ 26	/ 27	▲ 28	★ 14	▲ 29	/ 16	★ 30
Intel	● 38	● 39	● ■ 40	● 41	▲ 42	★ 14	▲ 43	/ 16	■ 44

● Full vendor support
◒ Indirect, but comprehensive support, by vendor
■ Vendor support, but not (yet) entirely comprehensive
▲ Comprehensive support, but not by vendor
★ Limited, probably indirect support – but at least some
/ No direct support available, but of course one could ISO-C-bind your way through it or directly link the libraries

JÜLICH Forschungszentrum

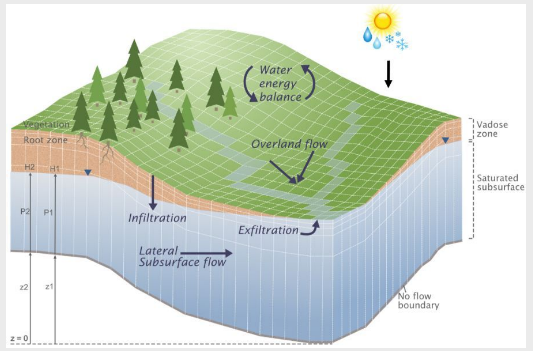
Two examples on how to achieve performance portability

SERGHEI

- C++
- 4+ years old (no legacy issues)
- Developed from scratch based on Kokkos
- Hybrid parallelisation, MPI+Kokkos

ParFlow

- C
- 30+ years old (plenty of legacy issues)
- Originally MPI only
- embedded Domain Specific Language (eDSL)
- Hybrid parallelisation: MPI+CUDA and MPI+Kokkos

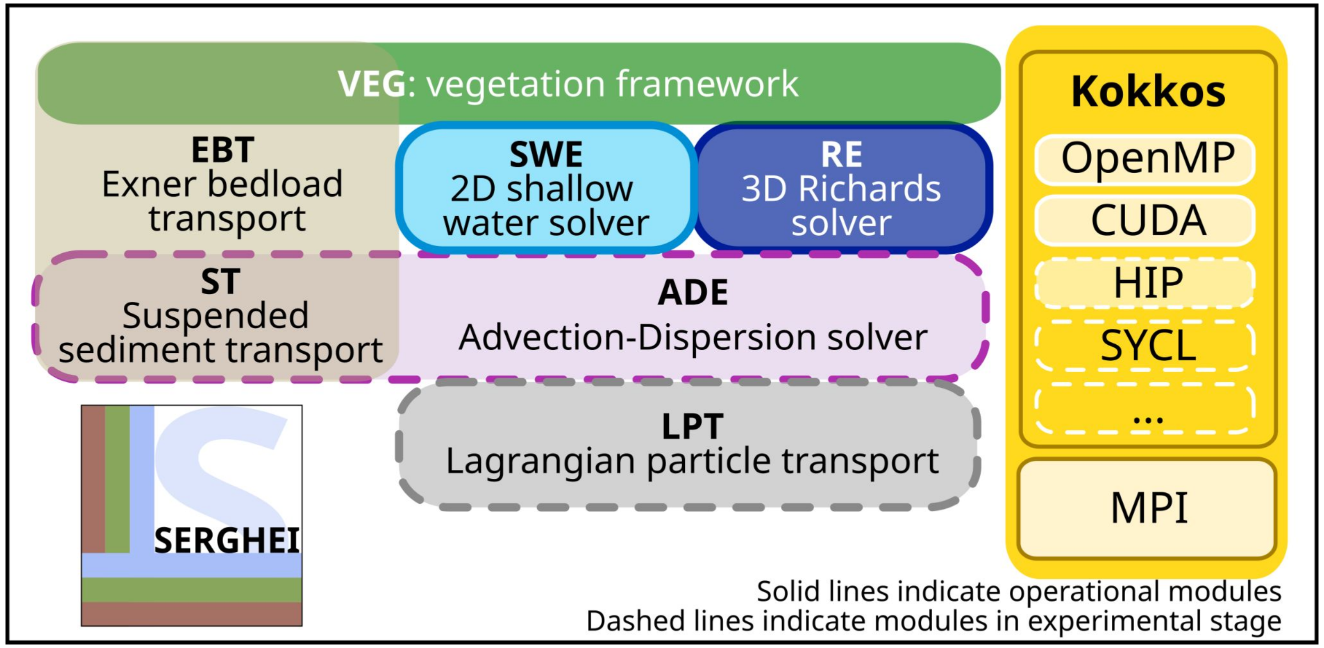


- ● Very lean and readable (mostly C/C++) code on top of Kokkos
- ● Experience on a new code
- ● Potential pitfall when/if Kokkos does not (quickly) support new backends

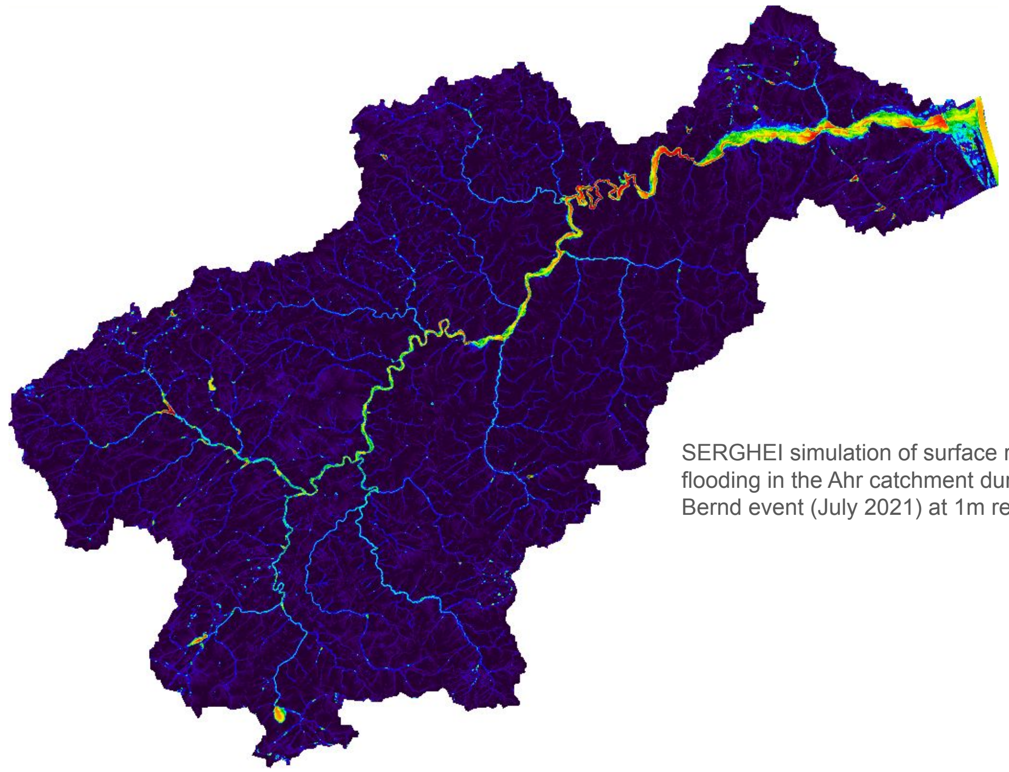
- ● Very flexible: you can implement arbitrary backends, including other portability layers
- ● Possibly well suited for legacy codes
- ● You have to implement the eDSL and backend

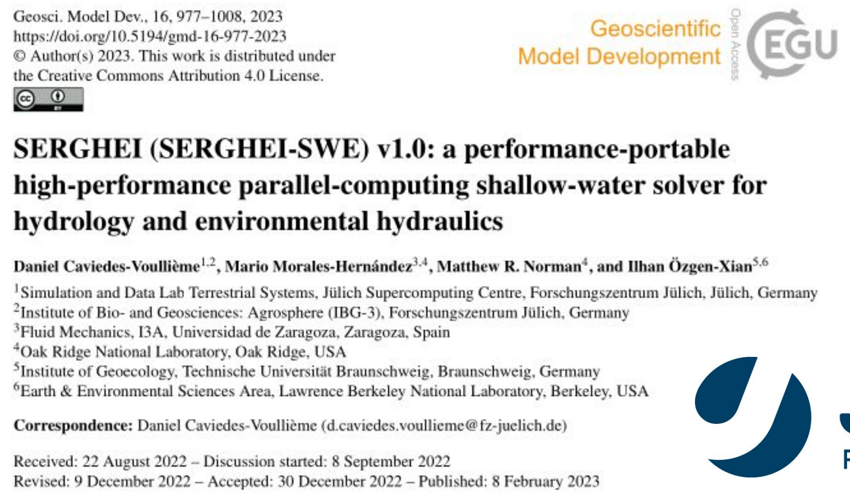
JÜLICH Forschungszentrum

SERGHEI Simulation EnviRonment for Geomorphology, Hydrodynamics, and Ecohydrology in Integrated form


VEG: vegetation framework
EBT
Exner bedload transport
SWE
2D shallow water solver
RE
3D Richards solver
ST
Suspended sediment transport
ADE
Advection-Dispersion solver
LPT
Lagrangian particle transport
SERGHEI
Kokkos
OpenMP
CUDA
HIP
SYCL
...
MPI
Solid lines indicate operational modules
Dashed lines indicate modules in experimental stage

- Modular, HPC-ready, open source
- Consolidating the last decade of mature numerical technology for shallow water
- Future-proofing and sustainability
- C++, performance-portability via Kokkos
- Hybrid MPI + (OpenMP, CUDA) operational
- Hybrid MPI + (HIP, SYCL) is experimental
- Applications in flooding, landscape function, transport & ESM


SERGHEI simulation of surface runoff and flooding in the Ahr catchment during the Bernd event (July 2021) at 1m resolution.


Geosci. Model Dev., 16, 977–1008, 2023
https://doi.org/10.5194/gmd-16-977-2023

Geoscientific Model Development

Open Access

EGU

SERGHEI (SERGHEI-SWE) v1.0: a performance-portable high-performance parallel-computing shallow-water solver for hydrology and environmental hydraulics

Daniel Caviedes-Voullième[1,2], Mario Morales-Hernández[3,4], Matthew R. Norman[4], and Ilhan Özgen-Xian[5,6]

[1]Simulation and Data Lab Terrestrial Systems, Jülich Supercomputing Centre, Forschungszentrum Jülich, Jülich, Germany
[2]Institute of Bio- and Geosciences: Agrosphere (IBG-3), Forschungszentrum Jülich, Germany
[3]Fluid Mechanics, I3A, Universidad de Zaragoza, Zaragoza, Spain
[4]Oak Ridge National Laboratory, Oak Ridge, USA
[5]Institute of Geoecology, Technische Universität Braunschweig, Braunschweig, Germany
[6]Earth & Environmental Sciences Area, Lawrence Berkeley National Laboratory, Berkeley, USA

Correspondence: Daniel Caviedes-Voullième (d.caviedes.voullieme@fz-juelich.de)

Received: 22 August 2022 – Discussion started: 8 September 2022
Revised: 9 December 2022 – Accepted: 30 December 2022 – Published: 8 February 2023

Li, Z., Rickert, G., Zheng, N., Zhang, Z., Özgen-Xian, I., Caviedes-Voullième, D. *SERGHEI v2.0: introducing a performance-portable, high-performance three-dimensional variably-saturated subsurface flow solver (SERGHEI-RE).* GMD. 2024. DOI: 10.5194/egusphere-2024-2588 **PREPRINT**

JÜLICH Forschungszentrum

SERGHEI and kokkos

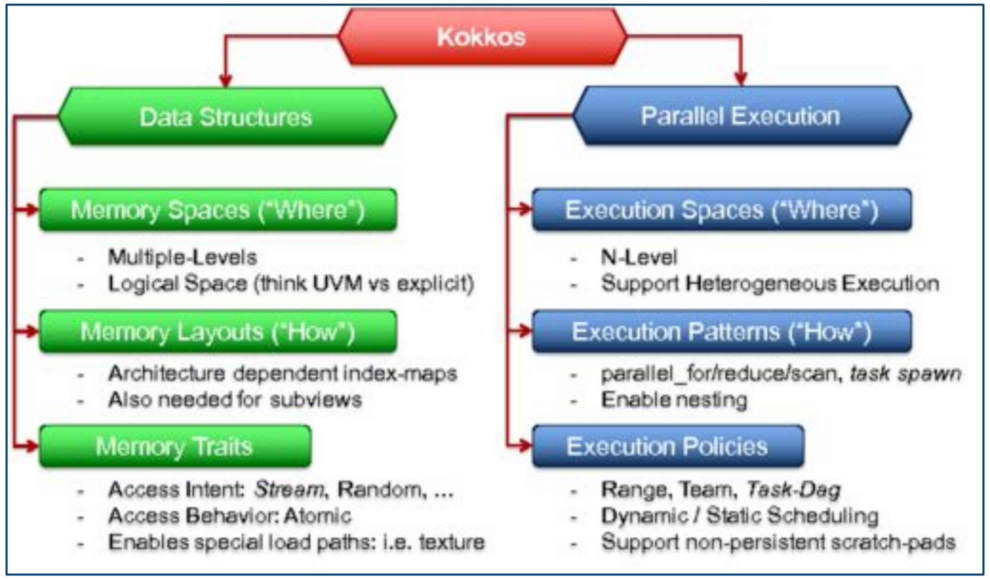
Serial (single CPU execution space)

```
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 (portable)

```
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

 })
```

Source: Kokkos documentation



Kokkos
Data Structures
Parallel Execution
Memory Spaces ("Where")
- Multiple-Levels
- Logical Space (think UVM vs explicit)
Memory Layouts ("How")
- Architecture dependent index-maps
- Also needed for subviews
Memory Traits
- Access Intent: Stream, Random, ...
- Access Behavior: Atomic
- Enables special load paths: i.e. texture
Execution Spaces ("Where")
- N-Level
- Support Heterogeneous Execution
Execution Patterns ("How")
- parallel_for/reduce/scan, task spawn
- Enable nesting
Execution Policies
- Range, Team, Task-Dag
- Dynamic / Static Scheduling
- Support non-persistent scratch-pads

- Serial regions are executed sequentially in host
- Parallel regions have an execution space determined at compilation.

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

Hybrid parallelisation: for multi-GPUs and/or multi-nodes, we still rely on MPI

SERGHEI and kokkos

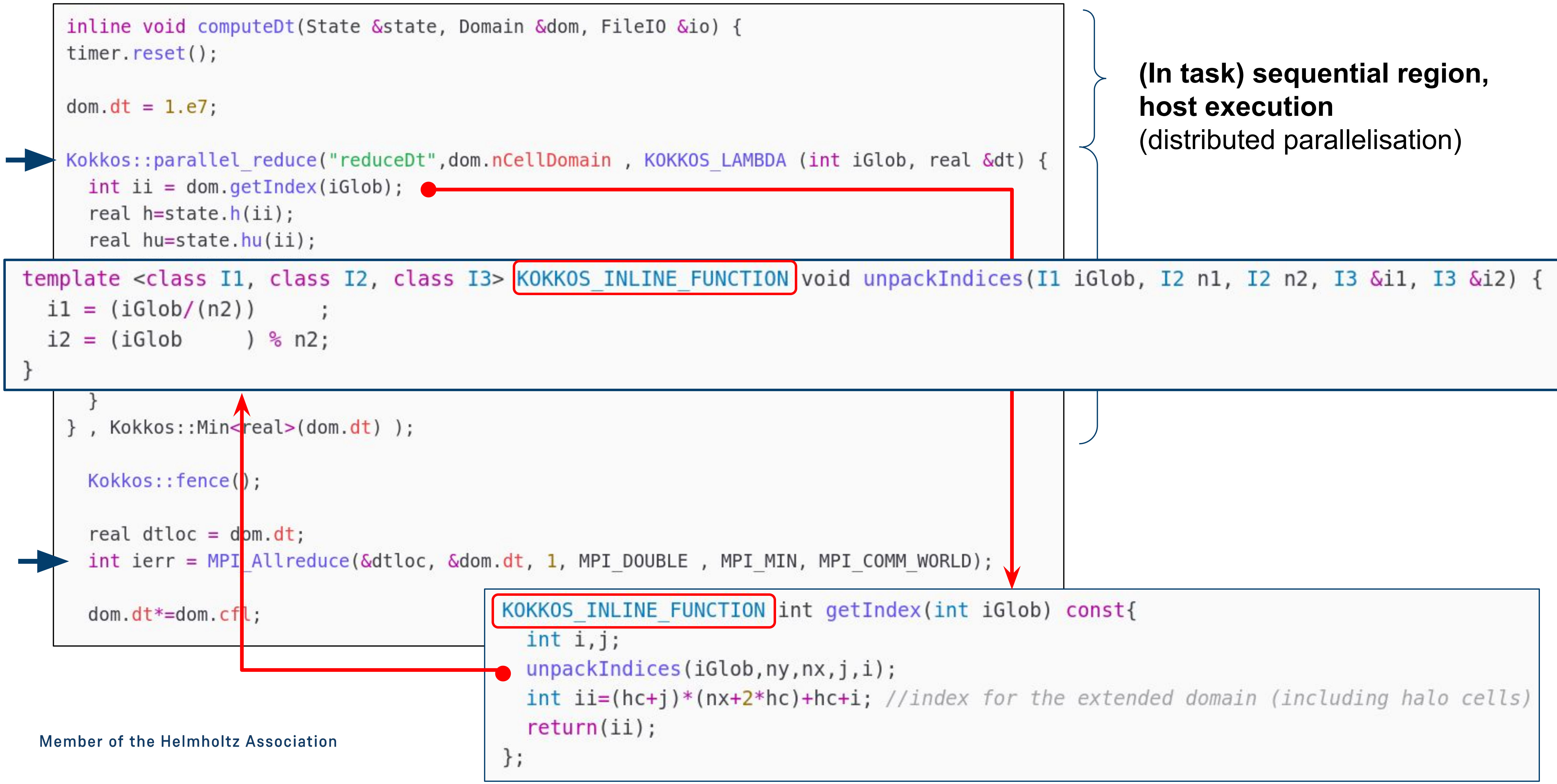
```cpp
inline void computeDt(State &state, Domain &dom, FileIO &io) {
timer.reset();

dom.dt = 1.e7;

Kokkos::parallel_reduce("reduceDt",dom.nCellDomain , KOKKOS_LAMBDA (int iGlob, real &dt) {
  int ii = dom.getIndex(iGlob);
  real h=state.h(ii);
  real hu=state.hu(ii);
  real hv=state.hv(ii);
  dt=fmin(dt,1.e6);
  if(h>TOL12){
    dt=fmin(dt,dom.dx/(fabs(hu/h)+sqrt(GRAV*h)));
    dt=fmin(dt,dom.dx/(fabs(hv/h)+sqrt(GRAV*h)));
  }
} , Kokkos::Min<real>(dom.dt) );

  Kokkos::fence();

  real dtloc = dom.dt;
  int ierr = MPI_Allreduce(&dtloc, &dom.dt, 1, MPI_DOUBLE , MPI_MIN, MPI_COMM_WORLD);

  dom.dt*=dom.cfl;
```

(In task) sequential region, host execution
(distributed parallelisation)

Parallel region
`Kokkos::parallel_reduce`

```cpp
KOKKOS_INLINE_FUNCTION int getIndex(int iGlob) const{
   int i,j;
   unpackIndices(iGlob,ny,nx,j,i);
   int ii=(hc+j)*(nx+2*hc)+hc+i; //index for the extended domain (including halo cells)
   return(ii);
};
```

SERGHEI and kokkos



```
inline void computeDt(State &state, Domain &dom, FileIO &io) {
timer.reset();

dom.dt = 1.e7;

Kokkos::parallel_reduce("reduceDt",dom.nCellDomain , KOKKOS_LAMBDA (int iGlob, real &dt) {
  int ii = dom.getIndex(iGlob);
  real h=state.h(ii);
  real hu=state.hu(ii);
```

```
template <class I1, class I2, class I3> KOKKOS_INLINE_FUNCTION void unpackIndices(I1 iGlob, I2 n1, I2 n2, I3 &i1, I3 &i2) {
  i1 = (iGlob/(n2))      ;
  i2 = (iGlob       ) % n2;
}
```

```
  }
} , Kokkos::Min<real>(dom.dt) );

  Kokkos::fence();

  real dtloc = dom.dt;
  int ierr = MPI_Allreduce(&dtloc, &dom.dt, 1, MPI_DOUBLE , MPI_MIN, MPI_COMM_WORLD);

  dom.dt*=dom.cfl;
```

```
KOKKOS_INLINE_FUNCTION int getIndex(int iGlob) const{
  int i,j;
  unpackIndices(iGlob,ny,nx,j,i);
  int ii=(hc+j)*(nx+2*hc)+hc+i; //index for the extended domain (including halo cells)
  return(ii);
};
```

(In task) sequential region, host execution
(distributed parallelisation)

SERGHEI and kokkos

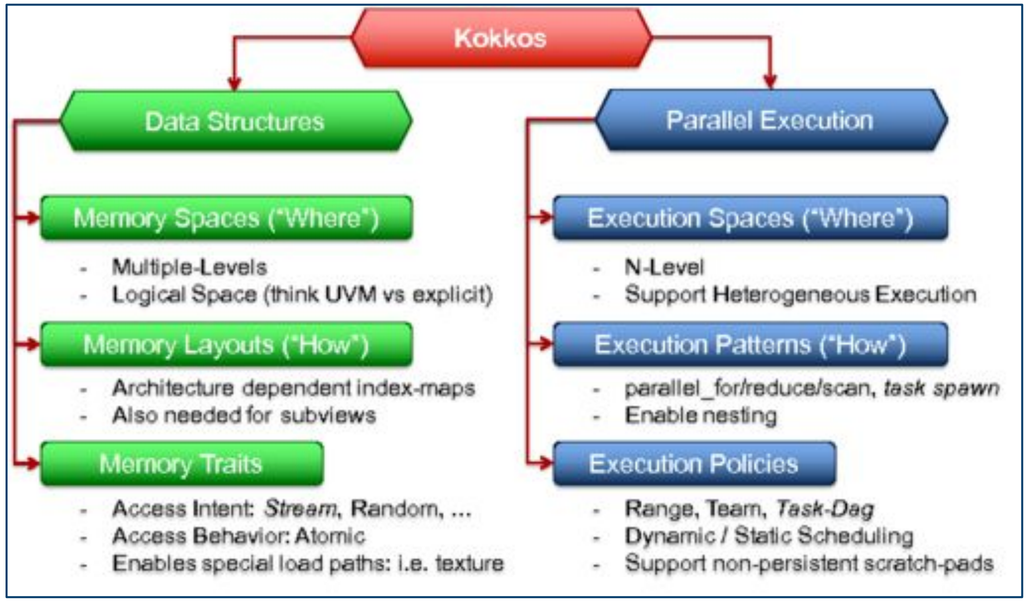
```
class ExtBC {
// this class is safe to invoke in a parallel region
public:
  real hzMin=1E6; // lowest water surface in boundary cross section
  real zMin=1E6; // lowest bed elevation in boundary cross section
  real zMax=-1E6;
  int nzMin=0;
```

```
// public member function of ExtBC

void inline getMinBedElevation(State &state){
  Kokkos::parallel_reduce("swe_bc_z_min", ncellsBC, KOKKOS_CLASS_LAMBDA(int iGlob, real &zMin){
    int ii = bcells[iGlob];
    real z = state.z(ii);
    zMin = min(zMin,z);
  }, Kokkos::Min<real>(zMin) );

  real zMin_all;
  MPI_Allreduce(&zMin, &zMin_all, 1, SERGHEI_MPI_REAL, MPI_MIN, comm);
  zMin = zMin_all;

  Kokkos::parallel_reduce("swe_bc_z_max", ncellsBC, KOKKOS_CLASS_LAMBDA(int iGlob, real &zMax){
    int ii = bcells[iGlob];
    real z = state.z(ii);
    zMax = max(zMax,z);
  }, Kokkos::Max<real>(zMax) );

  real zMax_all;
  MPI_Allreduce(&zMax, &zMax_all, 1, SERGHEI_MPI_REAL, MPI_MAX, comm);
  zMax = zMax_all;

// etc...
```

SERGHEI and kokkos

Kokkos::View and memory spaces

Source: Kokkos documentation


Kokkos
Data Structures
Parallel Execution
Memory Spaces ("Where")
- Multiple-Levels
- Logical Space (think UVM vs explicit)
Memory Layouts ("How")
- Architecture dependent index-maps
- Also needed for subviews
Memory Traits
- Access Intent: Stream, Random, ...
- Access Behavior: Atomic
- Enables special load paths: i.e. texture
Execution Spaces ("Where")
- N-Level
- Support Heterogeneous Execution
Execution Patterns ("How")
- parallel_for/reduce/scan, task spawn
- Enable nesting
Execution Policies
- Range, Team, Task-Dag
- Dynamic / Static Scheduling
- Support non-persistent scratch-pads

Explicitly defining Views and aliasing with typdef-ing

```
#if SERGHEI_REAL == SERGHEI_DOUBLE
  typedef double real;
#endif
#if SERGHEI_REAL == SERGHEI_FLOAT
typedef float real;
#endif

typedef unsigned long ulong;
typedef unsigned int uint;
```

```
#if defined(KOKKOS_ENABLE_CUDA)
  #include <cuda_runtime.h>
  typedef Kokkos::View<real*, Kokkos::LayoutRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> realArr;
  typedef Kokkos::View<int*, Kokkos::LayoutRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> intArr;
  typedef Kokkos::View<bool*, Kokkos::LayoutRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> boolArr;
  typedef Kokkos::View<double*, Kokkos::LayouRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> doubleArr;
#else
  typedef Kokkos::View<real*, Kokkos::LayoutRight> realArr;
  typedef Kokkos::View<int*, Kokkos::LayoutRight> intArr;
  typedef Kokkos::View<bool*, Kokkos::LayoutRight> boolArr;
  typedef Kokkos::View<double*, Kokkos::LayoutRight> doubleArr;
#endif
```

`Kokkos::View` is a (potentially) reference counted multi dimensional array with compile time layouts and **memory space** (which then needs to match the **execution space**).

SERGHEI and kokkos

```
typedef Kokkos::View<real*,   Kokkos::LayoutRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> realArr;
  typedef Kokkos::View<int*,    Kokkos::LayoutRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> intArr;
  typedef Kokkos::View<bool*,   Kokkos::LayoutRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> boolArr;
  typedef Kokkos::View<double*, Kokkos::LayouRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> doubleArr;
```

```
// SW variables
realArr h;
realArr hu;
realArr hv;

//elevation
realArr z;

//roughness
realArr roughness;
real hmin;
```

```
boolArr isnodata; //contains 0 if is a regular cell, 1 if is nodata cell

intArr isBound; //positive values for inlet boundaries, negative values for outlet
```

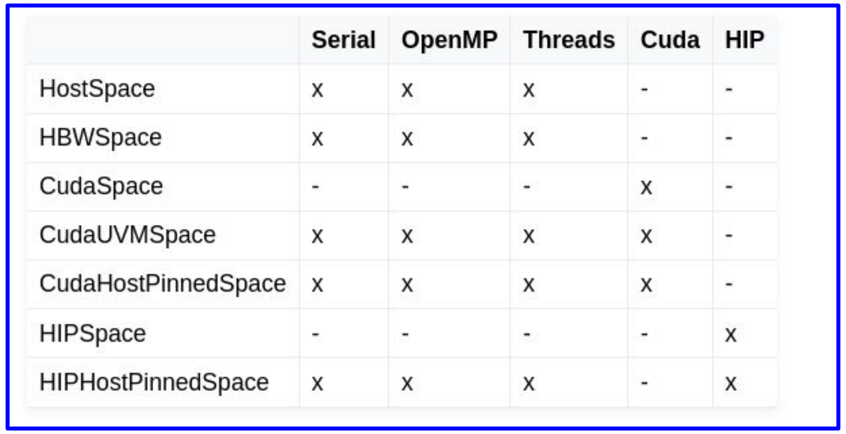
```
inline void computeNewState(State &state , const Domain &dom, const SourceSinkData &ss) {

  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

  })
```

```
inline void allocate(Domain &dom){
    h = realArr("h", dom.nCellMem);
    hu = realArr("hu", dom.nCellMem);
    hv = realArr("hv", dom.nCellMem);
    z = realArr("z", dom.nCellMem);
    roughness = realArr("roughness", dom.nCellMem);
    Kokkos::deep_copy(h, 0);
    Kokkos::deep_copy(hu, 0);
    Kokkos::deep_copy(hv, 0);
    Kokkos::deep_copy(z, 0);
    Kokkos::deep_copy(roughness, 0);
```

SERGHEI: reaching also HIP and SYCL via kokkos

Performance-portable view definitions of basic types in SERGHEI

```
#if defined(KOKKOS_ENABLE_CUDA)
  #include <cuda_runtime.h>
  typedef Kokkos::View<real*,   Kokkos::LayoutRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> realArr;
  typedef Kokkos::View<int*,    Kokkos::LayoutRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> intArr;
  typedef Kokkos::View<bool*,   Kokkos::LayoutRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> boolArr;
  typedef Kokkos::View<double*, Kokkos::LayouRight, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> doubleArr;
#elif defined(KOKKOS_ENABLE_HIP)
  #include <hip_runtime.h>
  typedef Kokkos::View<real*,   Kokkos::LayoutRight, Kokkos::Device<Kokkos::HIP, Kokkos::HIPManagedSpace>> realArr;
  typedef Kokkos::View<int*,    Kokkos::LayoutRight, Kokkos::Device<Kokkos::HIP, Kokkos::HIPManagedSpace>> intArr;
  typedef Kokkos::View<bool*,   Kokkos::LayoutRight, Kokkos::Device<Kokkos::HIP, Kokkos::HIPManagedSpace>> boolArr;
  typedef Kokkos::View<double*, Kokkos::LayoutRight, Kokkos::Device<Kokkos::HIP, Kokkos::HIPManagedSpace>> doubleArr;
#elif defined(KOKKOS_ENABLE_SYCL)
  #include <CL/sycl.hpp>
  typedef Kokkos::View<real*,   Kokkos::LayoutRight, Kokkos::Device<Kokkos::Experimental::SYCL, Kokkos::Experimental::SYCLSharedUSMSpace>> realArr;
  typedef Kokkos::View<int*,    Kokkos::LayoutRight, Kokkos::Device<Kokkos::Experimental::SYCL,Kokkos::Experimental::SYCLSharedUSMSpace>> intArr;
  typedef Kokkos::View<bool*,   Kokkos::LayoutRight, Kokkos::Device<Kokkos::Experimental::SYCL, Kokkos::Experimental::SYCLSharedUSMSpace>> boolArr;
  typedef Kokkos::View<double*, Kokkos::LayoutRight, Kokkos::Device<Kokkos::Experimental::SYCL, Kokkos::Experimental::SYCLSharedUSMSpace>> doubleArr;
#else
  typedef Kokkos::View<real*,   Kokkos::LayoutRight> realArr;
  typedef Kokkos::View<int*,    Kokkos::LayoutRight> intArr;
  typedef Kokkos::View<bool*,   Kokkos::LayoutRight> boolArr;
  typedef Kokkos::View<double*, Kokkos::LayoutRight> doubleArr;
#endif
```



	Serial	OpenMP	Threads	Cuda	HIP
HostSpace	x	x	x	-	-
HBWSpace	x	x	x	-	-
CudaSpace	-	-	-	x	-
CudaUVMSpace	x	x	x	x	-
CudaHostPinnedSpace	x	x	x	x	-
HIPSpace	-	-	-	-	x
HIPHostPinnedSpace	x	x	x	-	x

SERGHEI: reaching also HIP and SYCL via kokkos

Views of classes

```
#if defined(KOKKOS_ENABLE_CUDA)
  typedef Kokkos::View<ObservationGauge*, Kokkos::LayoutRight,Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> obsGaugeView;
#elif defined(KOKKOS_ENABLE_HIP)
  typedef Kokkos::View<ObservationGauge*, Kokkos::LayoutRight,Kokkos::Device<Kokkos::HIP, Kokkos::HIPManagedSpace>> obsGaugeView;
#elif defined(KOKKOS_ENABLE_SYCL)
  typedef Kokkos::View<ObservationGauge*, Kokkos::LayoutRight,Kokkos::Device<Kokkos::Experimental::SYCL,Kokkos::Experimental::SYCLSharedUSMSpace>> obsGaugeView;
#else
  typedef Kokkos::View<ObservationGauge*, Kokkos::LayoutRight> obsGaugeView;
#endif
```

```
class ObservationGauge{
  // this class MUST have a default constructor/destructor because it is used in a Kokkos::View
  // here we have an implicit constructor/destructor (the compiler will create one)
public:

  KOKKOS_FUNCTION ObservationGauge(){}
  int ii; // to store the index which relates to the memory index count
  int ic; // to store the physical cell index
  int id; // to store the subdomain where to find the gauge

  KOKKOS_INLINE_FUNCTION void linkDomain(const Domain &dom){
    ii = dom.getIndexForPoint(x);
    ic = dom.getCellForPoint(x);
    id = dom.id;
  };

  KOKKOS_INLINE_FUNCTION void fetchSurfaceState(const State &state){
    if (ii < 0){ // gauge is undefined, values should be zero, so they can be reduced with MPI_SUM
      sw.h = sw.hu = sw.hv = sw.z = 0.;
    }else{
      sw.h = state.h(ii);
      sw.hu = state.hu(ii);
      sw.hv = state.hv(ii);
      sw.z = state.z(ii);
    }
  };

// and more...
```

SERGHEI: reaching also HIP and SYCL via kokkos

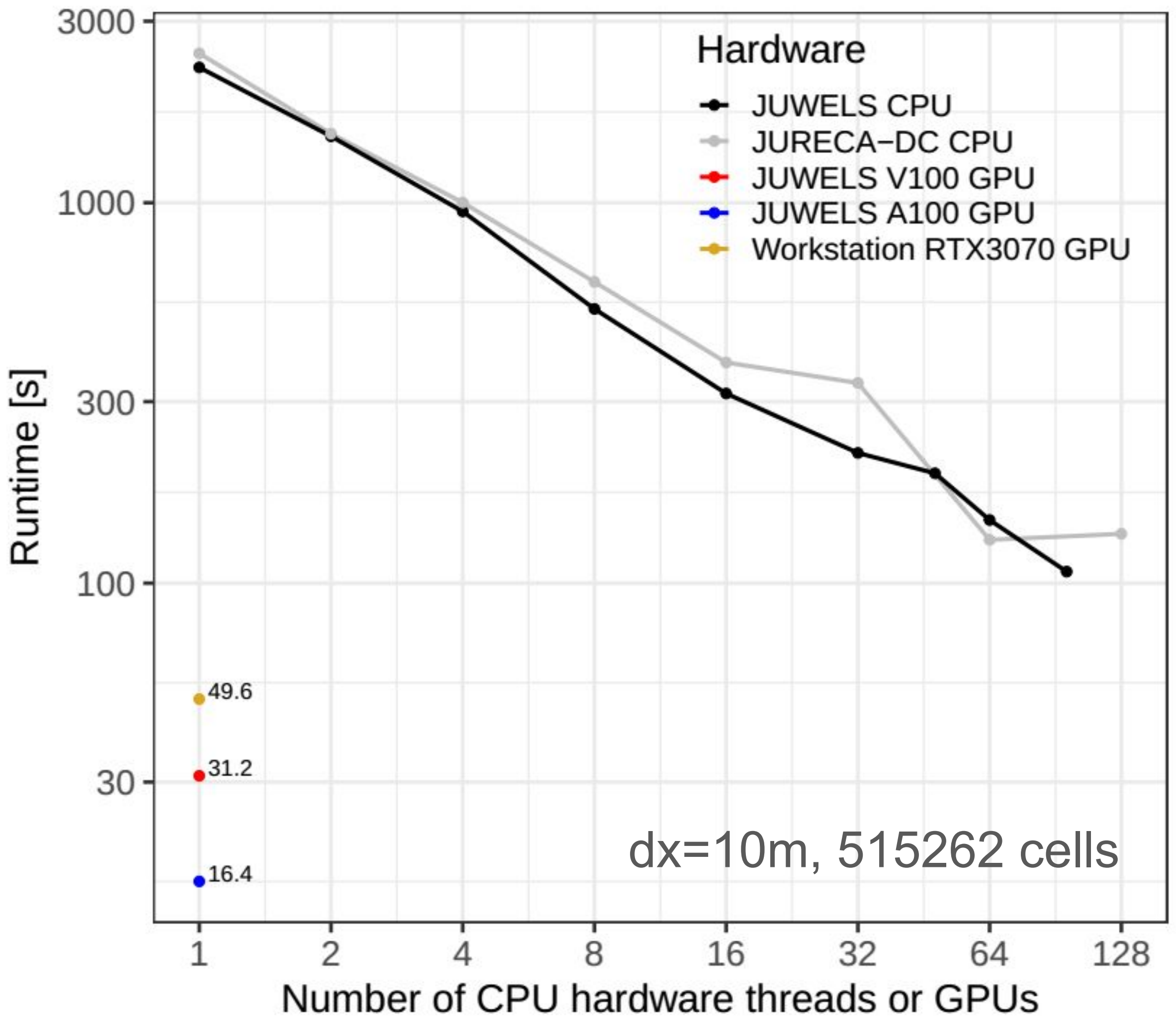
Views of template classes

```
#include "SArray.h"
#if defined(KOKKOS_ENABLE_CUDA)
  typedef Kokkos::View<SArray<real,2>*, Kokkos::LayoutRight,Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpace>> realS2Arr;
#elif defined(KOKKOS_ENABLE_HIP)
  typedef Kokkos::View<SArray<real,2>*, Kokkos::LayoutRight,Kokkos::Device<Kokkos::HIP, Kokkos::HIPManagedSpace>> realS2Arr;
#elif defined(KOKKOS_ENABLE_SYCL)
  typedef Kokkos::View<SArray<real,2>*, Kokkos::LayoutRight,Kokkos::Device<Kokkos::Experimental::SYCL, Kokkos::Experimental::SYCLSharedUSMSpace>> realS2Arr;
#else
  typedef Kokkos::View<SArray<real,2>*, Kokkos::LayoutRight> realS2Arr;
#endif
```

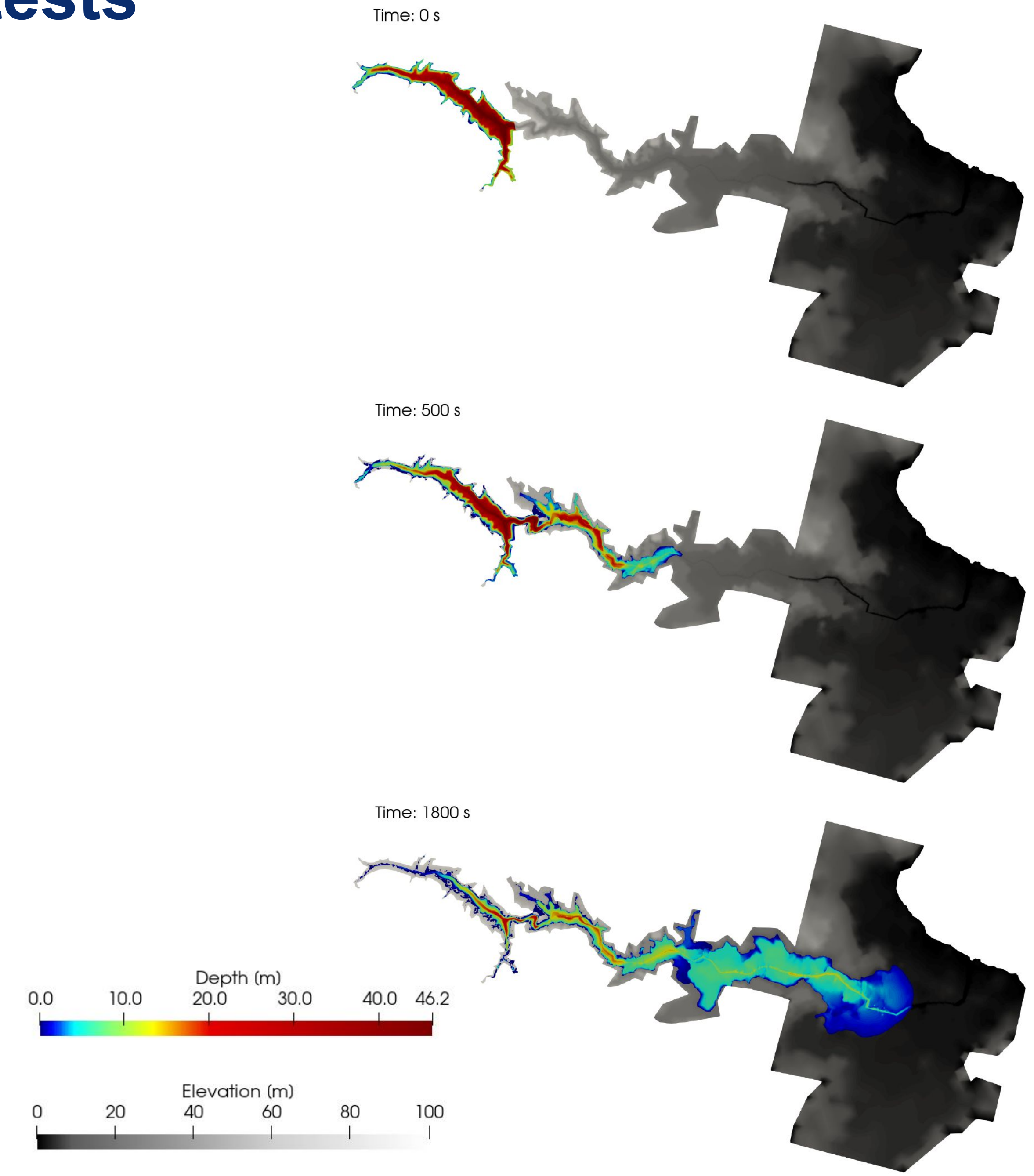
```
template <class T, unsigned long D0, unsigned long D1=1, unsigned long D2=1, unsigned long D3=1> class SArray {
protected:
  typedef unsigned long ulong;

  T data[D0*D1*D2];

public :
  KOKKOS_INLINE_FUNCTION SArray() { }
  KOKKOS_INLINE_FUNCTION ~SArray() { }
  KOKKOS_INLINE_FUNCTION T &operator()(ulong const i0)       {
    #ifdef ARRAY_DEBUG
      if (D1*D2*D3 > 1) {std::cout << "SArray: Using 2D or higher array as 1D array\n";}
      if (i0>D0-1) { printf("i0 > D0-1"); exit(-1); }
    #endif
    return data[i0];
  }
// and more...
```

SERGHEI: performance portability tests

Single node (no MPI) performance benchmark test

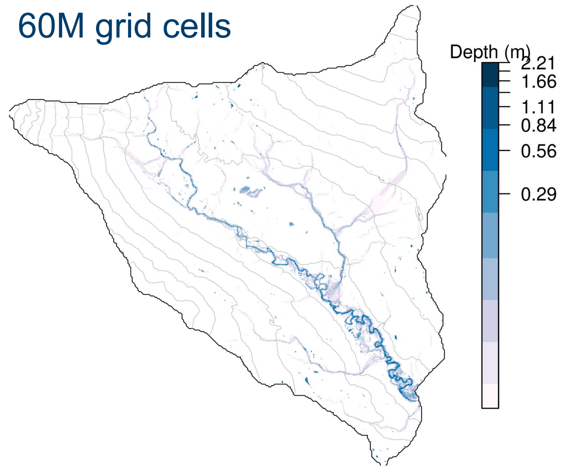


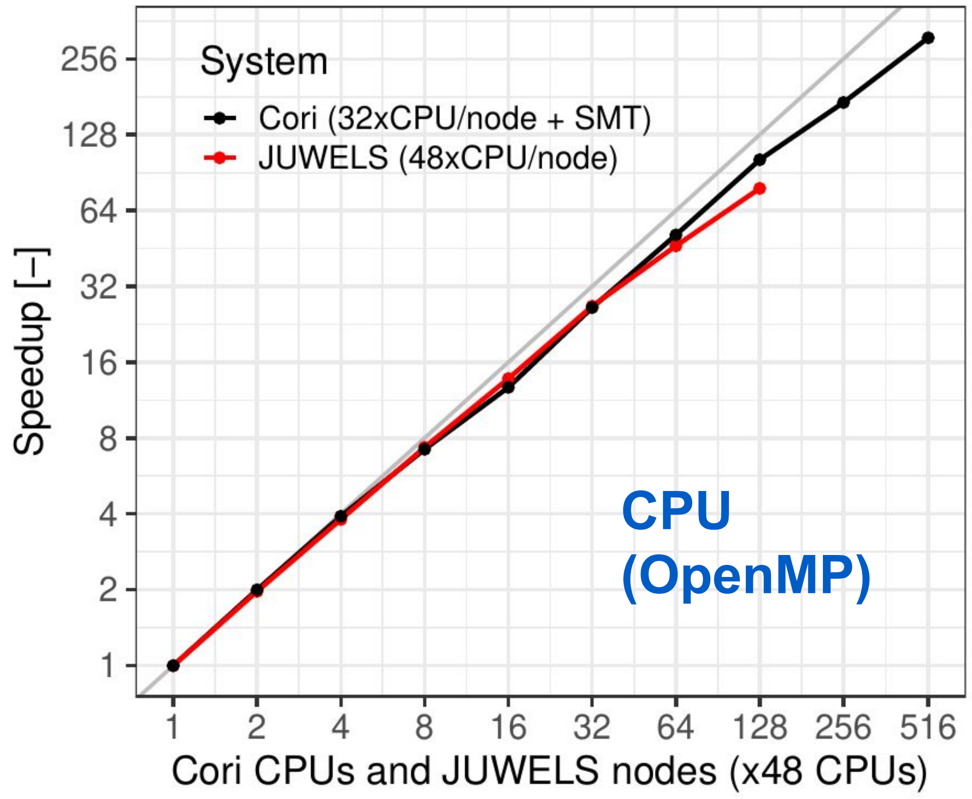
Malpasset dam break event



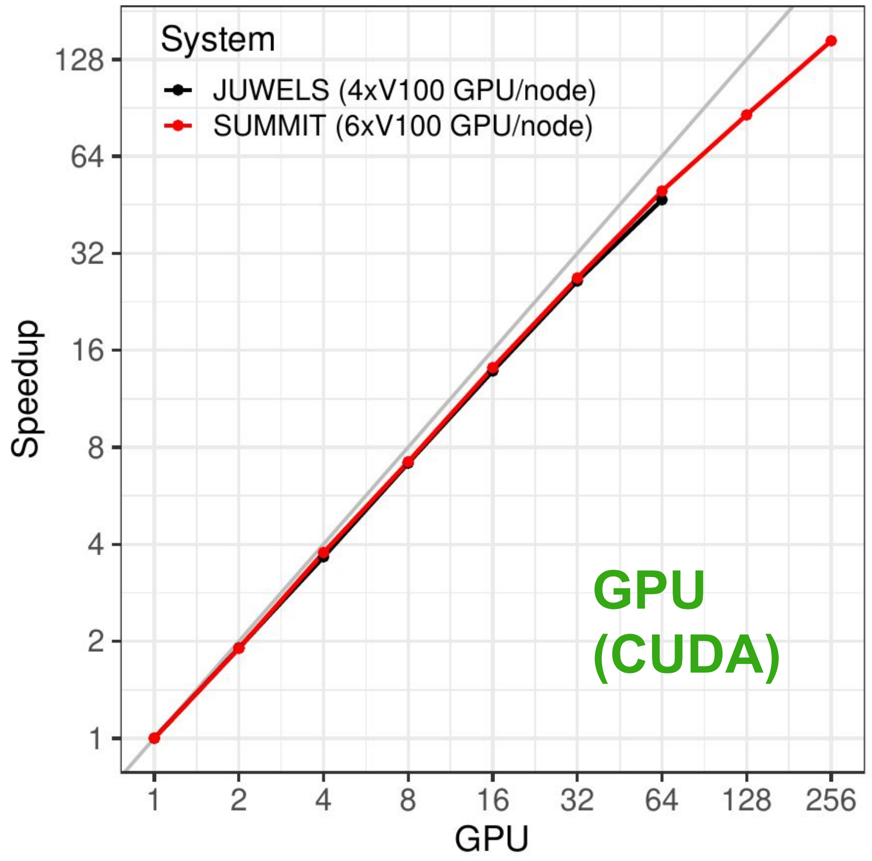
SERGHEI: performance portability tests

Real catchment,
60M grid cells





CPU (OpenMP)



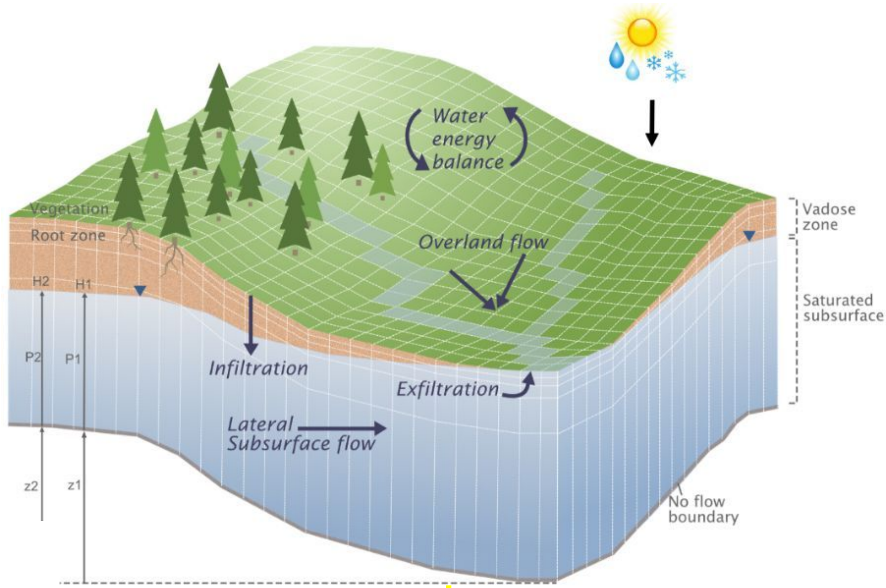
GPU (CUDA)

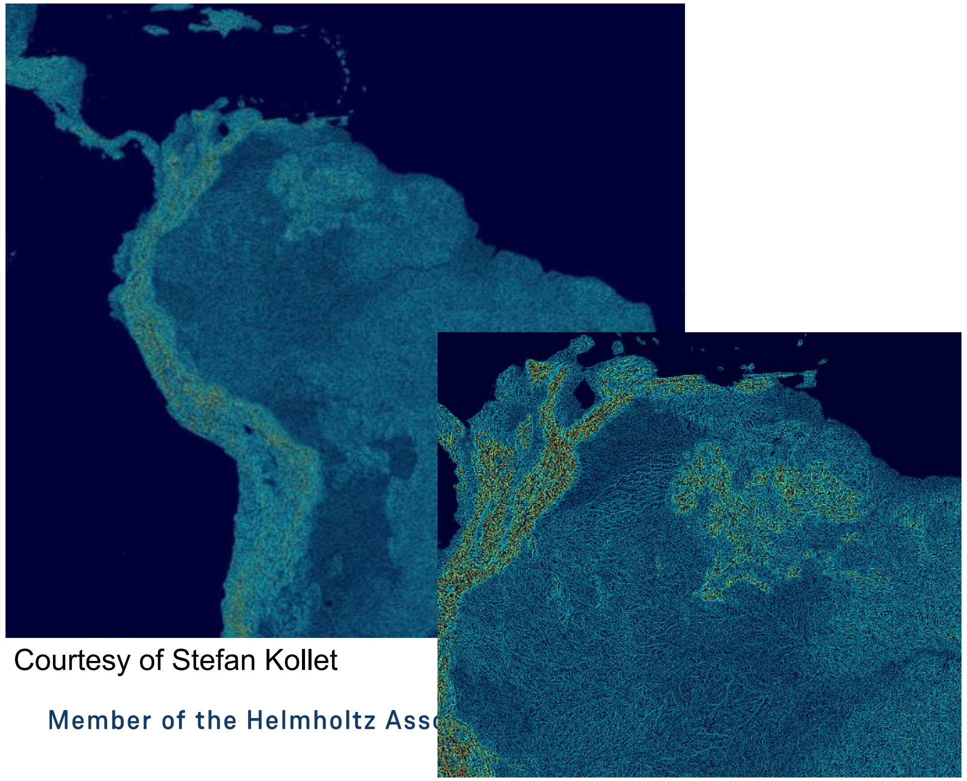
JÜLICH Forschungszentrum

Caviedes-Voullième et al. *SERGHEI (SERGHEI-SWE) v1.0: a performance-portable high-performance parallel-computing shallow-water solver for hydrology and environmental hydraulics.* GMD. 2023. DOI: 10.5194/gmd-16-977-2023

ParFlow https://github.com/parflow

- Integrated hydrological model (also part of TSMP)
- Solves 3D Richards equation + 2D kinematic/diffusive surface flow
- Embedded Domain Specific Language (eDSL)
- CUDA, Kokkos-CUDA, Kokkos-HIP

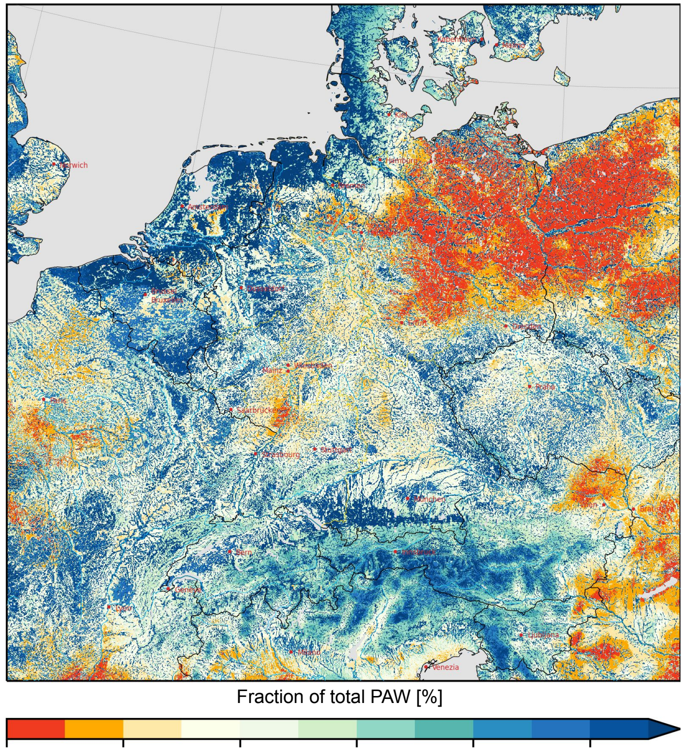




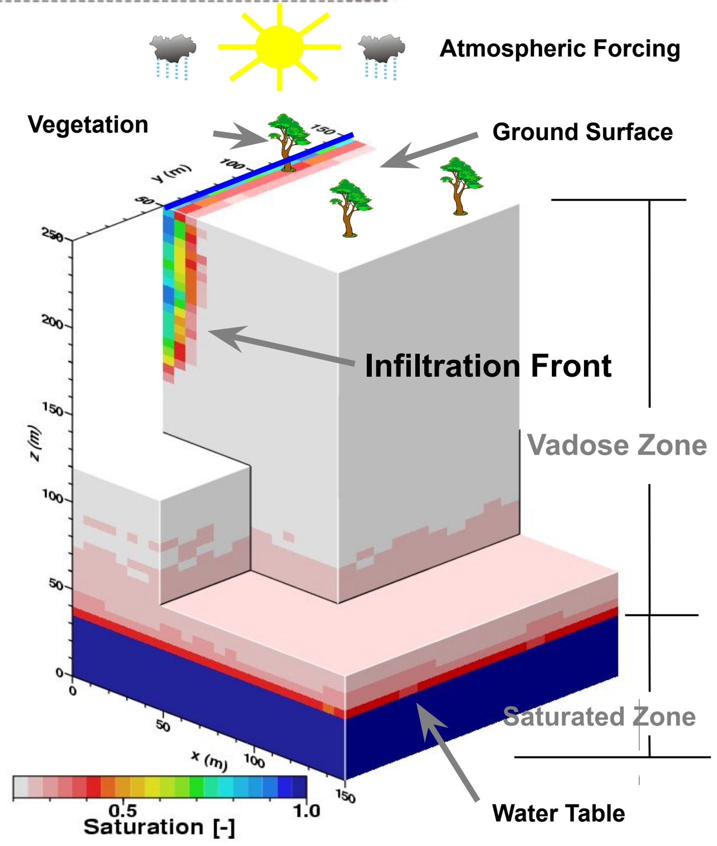
Courtesy of Stefan Kollet

Plant available water

2021-08-18 daily sum, 30cm depth

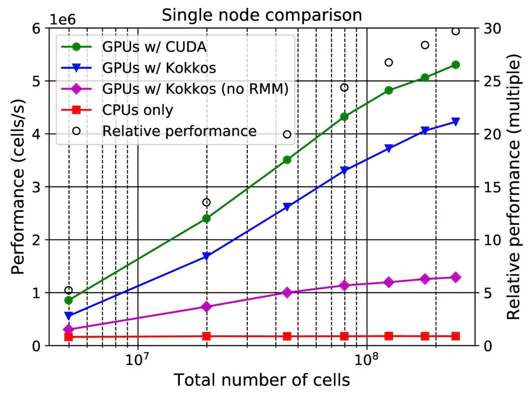


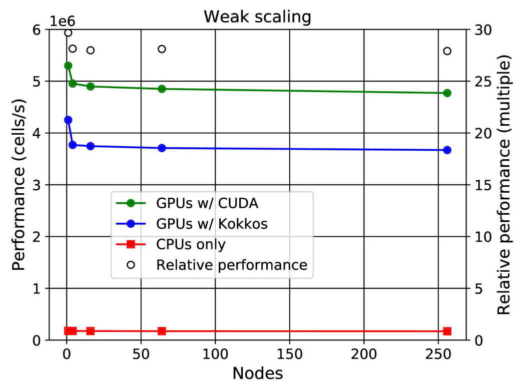
Courtesy of Klaus Görgen and Alexandre Belleflamme



ParFlow's performance-portability story

```
double *fp;
double *fp;
double value;
Subvector *f_sub;

/* some code missing here*/

for(k = iz; k < iz + nz; k++)
    for(j = iy; j < iy + ny; j++)
        for(i = ix; i < ix + nx; i++)
        {
            int ip = SubvectorEltIndex(f_sub, i, j, k);
            fp[ip] = pp[ip] - value;
        }
```





MPI + eDSL → MPI + eDSL (CUDA) → MPI + eDSL (CUDA, Kokkos[CUDA]) → MPI + eDSL (CUDA, Kokkos[CUDA,HIP])

2020 2021 2022 2023

NVIDIA CUDA kokkos natESM ROCm

```
double *fp;
double *fp;
double value;
Subvector *f_sub;

/* some code missing here*/

BoxLoopI0(i, j, k, ix, iy, iz, nx, ny, nz,
        {
            int ip = SubvectorEltIndex(f_sub, i, j, k);
            fp[ip] = pp[ip] - value;
        });
```

eDSL

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  auto lambda_body = [=] __host__ __device__
    (const int i, const int j, const int k)
      loop_body;

  /* some code missing for grid & block sizes */

  BoxKernelI0<<<grid, block>>>(lambda_body,
    ix, iy, iz, nx, ny, nz);
}
```

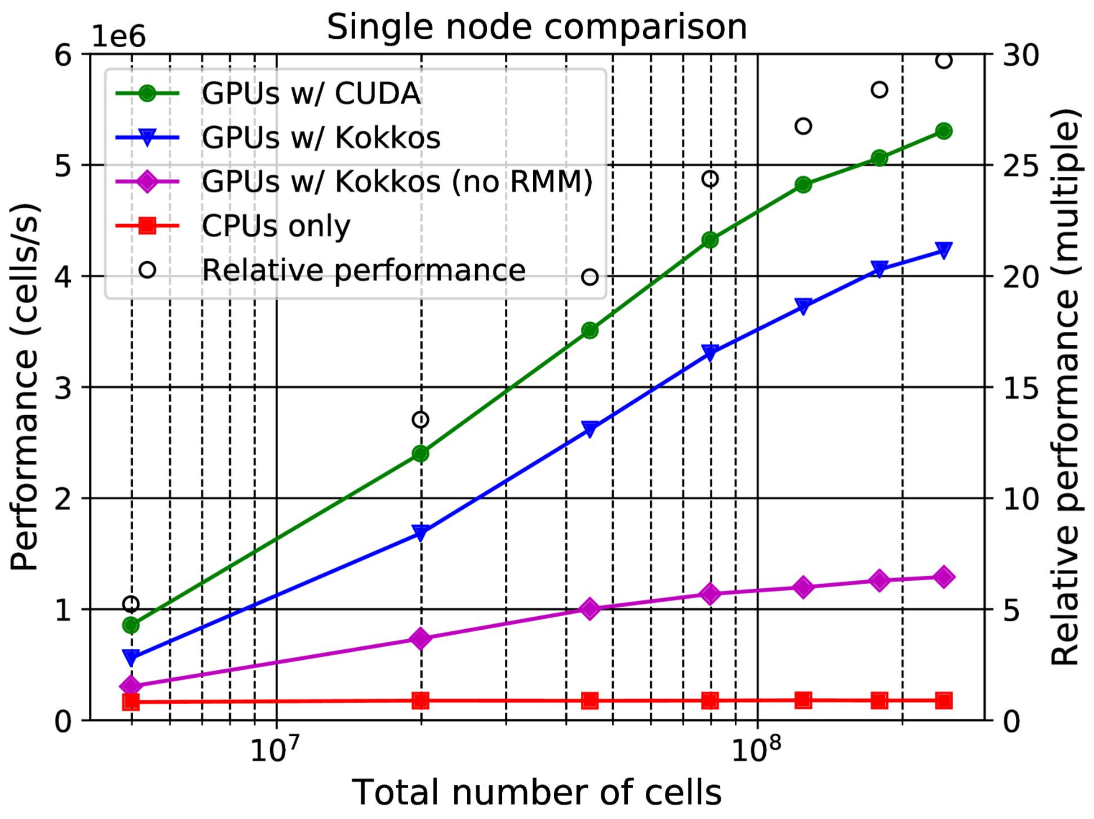
CUDA

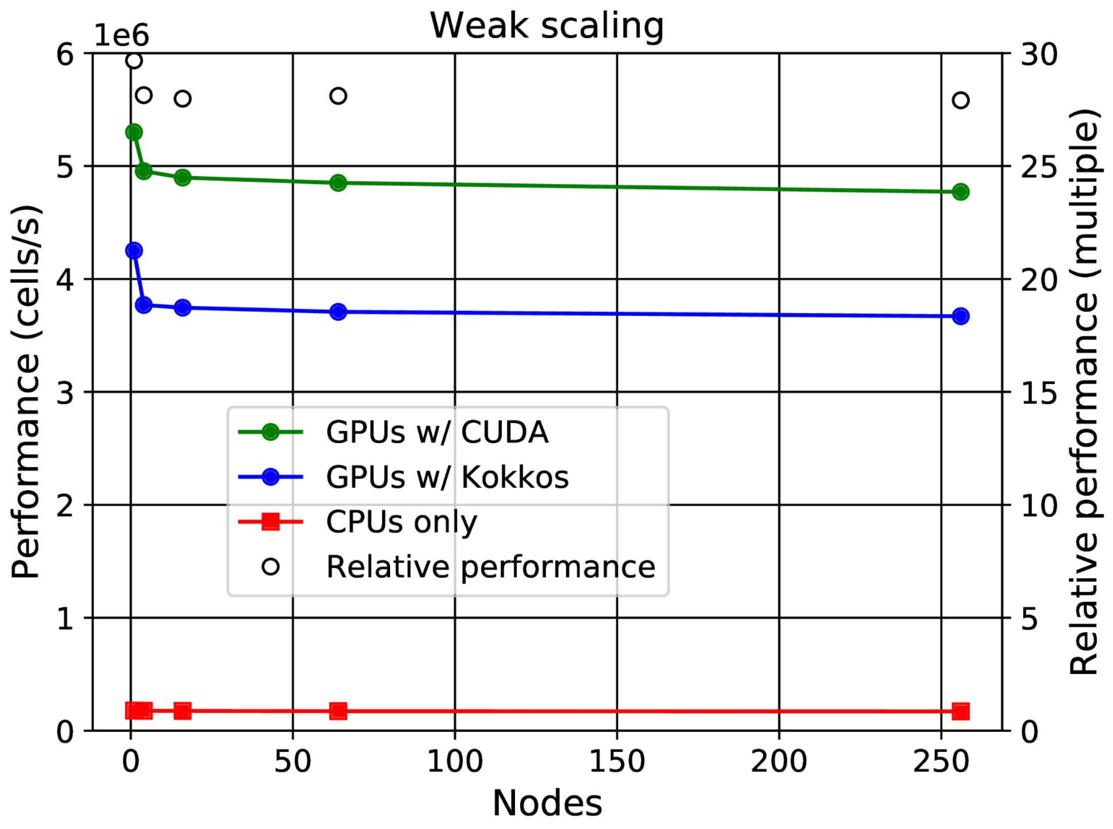
Hokkanen et al. 2021. doi: 10.1007/s10596-021-10051-4

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  auto lambda_body = KOKKOS_LAMBDA(int i, int j, int k)
  {
    i += ix; j += iy; k += iz;
    loop_body;
  }

  MDPolicyType_3D mdpolicy_3d({{0, 0, 0}},{{nx, ny, nz}});
  Kokkos::parallel_for(mdpolicy_3d, lambda_body);
}
```

Kokkos

ParFlow: performance portability tests





ParFlow eDSL (embedded Domain Specific Language)

Key idea: abstract code structures which repeat throughout the code into some macros

A typical loop in 3D space spanning indices i,j,k

```
double *fp;
double *fp;
double value;
Subvector *f_sub;

/* some code missing here*/

for(k = iz; k < iz + nz; k++)
    for(j = iy; j < iy + ny; j++)
        for(i = ix; j < ix + nx; i++)
        {
            int ip = SubvectorEltIndex(f_sub, i, j, k);
            fp[ip] = pp[ip] - value;
        }
```

Same loop, abstracted into the `BoxLoopI0` macro

```
double *fp;
double *fp;
double value;
Subvector *f_sub;

/* some code missing here*/

BoxLoopI0(i, j, k, ix, iy, iz, nx, ny, nz,
        {
            int ip = SubvectorEltIndex(f_sub, i, j, k);
            fp[ip] = pp[ip] - value;
        });
```

eDSL macro definition for `BoxLoopI0`

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  for (k = iz; k < iz + nz; k++)
    for (j = iy; j < iy + ny; j++)
      for (i = ix; i < ix + nx; i++)
      {
        loop_body;
      }
}
```

Hokkanen et al. 2021. Leveraging HPC accelerator architectures with modern techniques — hydrologic modeling on GPUs with ParFlow. Computational Geosciences. Doi: 10.1007/s10596-021-10051-4

ParFlow eDSL and portability

Key idea: write all hardware dependent code inside the eDSL macros

eDSL macro definition - sequential (host)

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  for (k = iz; k < iz + nz; k++)
    for (j = iy; j < iy + ny; j++)
      for (i = ix; i < ix + nx; i++)
      {
        loop_body;
      }
}
```

eDSL macro definition - CUDA (device)

NVIDIA. CUDA.

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  auto lambda_body = [=] __host__ __device__
    (const int i, const int j, const int k)
      loop_body;

  /* some code missing for grid & block sizes */

  BoxKernelI0<<<grid, block>>>(lambda_body,
    ix, iy, iz, nx, ny, nz);
}
```

eDSL macro definition - Kokkos (host & device / sequential, parallel)

kokkos

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  auto lambda_body = KOKKOS_LAMBDA(int i, int j, int k)
  {
    i += ix; j += iy; k += iz;
    loop_body;
  }

  MDPolicyType_3D mdpolicy_3d({{0, 0, 0}},{{nx, ny, nz}});
  Kokkos::parallel_for(mdpolicy_3d, lambda_body);
}
```

JÜLICH Forschungszentrum

Hokkanen et al. 2021. Leveraging HPC accelerator architectures… Computational Geosciences. Doi: 10.1007/s10596-021-10051-4

ParFlow eDSL: resolving backends

Somewhere in Parflow we call **BoxLoopI1**

```
cell_volume = dx * dy * dz;

perm_x_elt = SubvectorElt(perm_x_sub, ix, iy, iz);
perm_y_elt = SubvectorElt(perm_y_sub, ix, iy, iz);
perm_z_elt = SubvectorElt(perm_z_sub, ix, iy, iz);

pi = 0;
BoxLoopI1(i, j, k,
          ix, iy, iz, nx, ny, nz,
          pi, nx_p, ny_p, nz_p, 1, 1, 1,
{
  perm_average_x += perm_x_elt[pi] * (cell_volume / well_volume);
  perm_average_y += perm_y_elt[pi] * (cell_volume / well_volume);
  perm_average_z += perm_z_elt[pi] * (cell_volume / well_volume);
});

FreeSubgrid(tmp_subgrid);       /* done with temporary subgrid */
```

CUDA

BoxLoopI1_cuda

```
#define BoxLoopI1_cuda(i, j, k,
  ix, iy, iz, nx, ny, nz,
  i1, nx1, ny1, nz1, sx1, sy1, sz1,
  loop_body)
{
  if(nx > 0 && ny > 0 && nz > 0)
  {
    DeclareInc(PV_jinc_1, PV_kinc_1, nx, ny, nz, nx1, ny1, nz1, sx1, sy1, sz1);

    dim3 block, grid;
    FindDims(grid, block, nx, ny, nz, 1);

    const auto &ref_i1 = i1;

    auto lambda_body =
      GPU_LAMBDA(int i, int j, int k)
      {
        const int i1 = k * PV_kinc_1 + (k * ny + j) * PV_jinc_1
          + (k * ny * nx + j * nx + i) * sx1 + ref_i1;

        i += ix;
        j += iy;
        k += iz;
        loop_body;
      };

    BoxKernel<<<grid, block>>>(lambda_body, nx, ny, nz);
    CUDA_ERR(cudaPeekAtLastError());

    typedef function_traits<decltype(lambda_body)> traits;
    if(!std::is_same<traits::result_type, struct SkipParallelSync>::value)
      CUDA_ERR(cudaStreamSynchronize(0));
  }
  (void)i;(void)j;(void)k;
}
```

HOST

```
#define BoxLoopI1_default(i, j, k,
                ix, iy, iz, nx, ny, nz,
                i1, nx1, ny1, nz1, sx1, sy1, sz1,
                body)
 {
   DeclareInc(PV_jinc_1, PV_kinc_1, nx, ny, nz, nx1, ny1, nz1, sx1, sy1, sz1);
   for (k = iz; k < iz + nz; k++)
   {
     for (j = iy; j < iy + ny; j++)
     {
       for (i = ix; i < ix + nx; i++)
       {
         body;
         i1 += sx1;
       }
       i1 += PV_jinc_1;
     }
     i1 += PV_kinc_1;
   }
 }
```

BoxLoopI1_default

ParFlow eDSL: resolving backends

Check it out in: `parflow/pfsimulator/parflow_lib/backend_mapping.h`

Macro magic tricks 3

```
#define EMPTY()
#define DEFER(x) x EMPTY()
#define PASTER(x,y) x ## y
#define EVALUATOR(x,y) PASTER(x,y)
#define CHOOSE_BACKEND(name, id) EVALUATOR(name, id)
```

Define the **ACC_ID**, after resolving build flags 1

```
#ifdef PARFLOW_HAVE_KOKKOS

  #define ACC_ID _kokkos

  #include "pf_devices.h"

  #if PF_COMP_UNIT_TYPE == 1
    #include "pf_kokkosloops.h"
  #elif PF_COMP_UNIT_TYPE == 2
    #include "pf_kokkosmalloc.h"
  #endif

#elif defined(PARFLOW_HAVE_CUDA)

  #define ACC_ID _cuda

  #include "pf_devices.h"

  #if PF_COMP_UNIT_TYPE == 1
    #include "pf_cudaloops.h"
  #elif PF_COMP_UNIT_TYPE == 2
    #include "pf_cudamalloc.h"
  #endif
```

Resolving the **loop name** 2

```
#if defined(BoxLoopI1_cuda) || defined(BoxLoopI1_kokkos) || defined(BoxLoopI1_omp)
  #define BoxLoopI1 CHOOSE_BACKEND(DEFER(BoxLoopI1), ACC_ID)
#else
  #define BoxLoopI1 BoxLoopI1_default
#endif
```

4

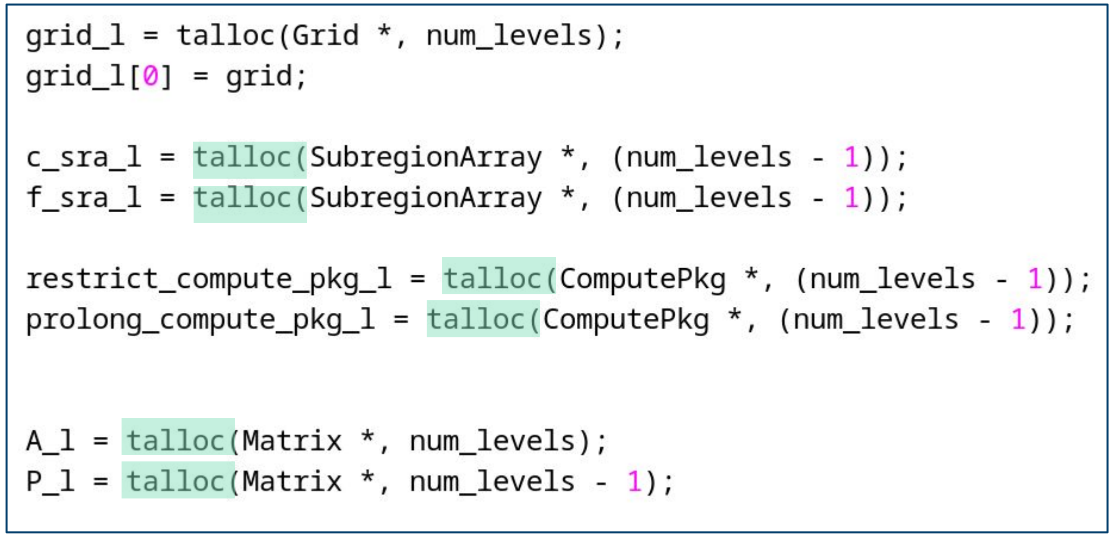
```
#define BoxLoopI1_cuda(i, j, k,
  ix, iy, iz, nx, ny, nz,
  i1, nx1, ny1, nz1, sx1, sy1, sz1,
  loop_body)
{
  if(nx > 0 && ny > 0 && nz > 0)
  {
    DeclareInc(PV_jinc_1, PV_kinc_1, nx, ny, nz, nx1, ny1, nz1, sx1, sy1, sz1);

    dim3 block, grid;
    FindDims(grid, block, nx, ny, nz, 1);

    const auto &ref_i1 = i1;

    auto lambda_body =
      GPU_LAMBDA(int i, int j, int k)
      {
        const int i1 = k * PV_kinc_1 + (k * ny + j) * PV_jinc_1
          + (k * ny * nx + j * nx + i) * sx1 + ref_i1;

        i += ix;
        j += iy;
        k += iz;
        loop_body;
      };

    BoxKernel<<<grid, block>>>(lambda_body, nx, ny, nz);
    CUDA_ERR(cudaPeekAtLastError());
```

4b

```
#define BoxLoopI1_default(i, j, k,
                ix, iy, iz, nx, ny, nz,
                i1, nx1, ny1, nz1, sx1, sy1, sz1,
                body)
  {
    DeclareInc(PV_jinc_1, PV_kinc_1, nx, ny, nz, nx1, ny1, nz1, sx1, sy1, sz1);
    for (k = iz; k < iz + nz; k++)
    {
      for (j = iy; j < iy + ny; j++)
      {
        for (i = ix; i < ix + nx; i++)
        {
          body;
          i1 += sx1;
        }
        i1 += PV_jinc_1;
      }
      i1 += PV_kinc_1;
    }
  }
```

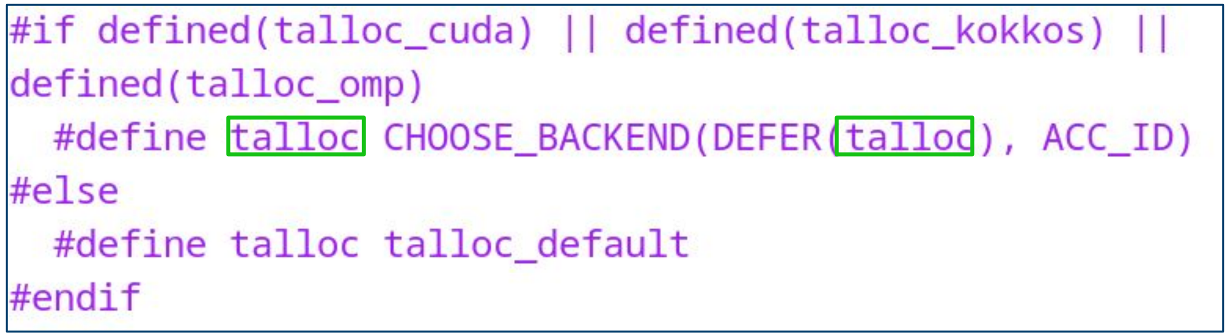
ParFlow eDSL: example of memory (de)allocators

From `parflow/pfsimulator/parflow_lib/mg_semi.c`



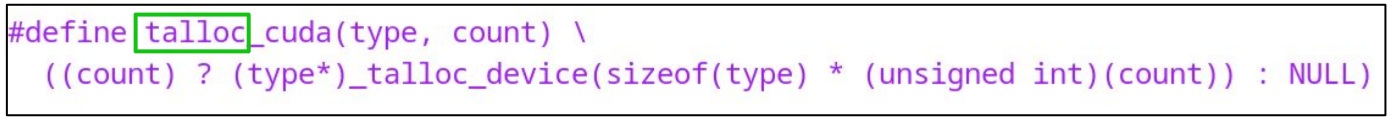
```c
grid_l = talloc(Grid *, num_levels);
grid_l[0] = grid;

c_sra_l = talloc(SubregionArray *, (num_levels - 1));
f_sra_l = talloc(SubregionArray *, (num_levels - 1));

restrict_compute_pkg_l = talloc(ComputePkg *, (num_levels - 1));
prolong_compute_pkg_l = talloc(ComputePkg *, (num_levels - 1));


A_l = talloc(Matrix *, num_levels);
P_l = talloc(Matrix *, num_levels - 1);
```

From `parflow/pfsimulator/parflow_lib/backend_mapping.h`

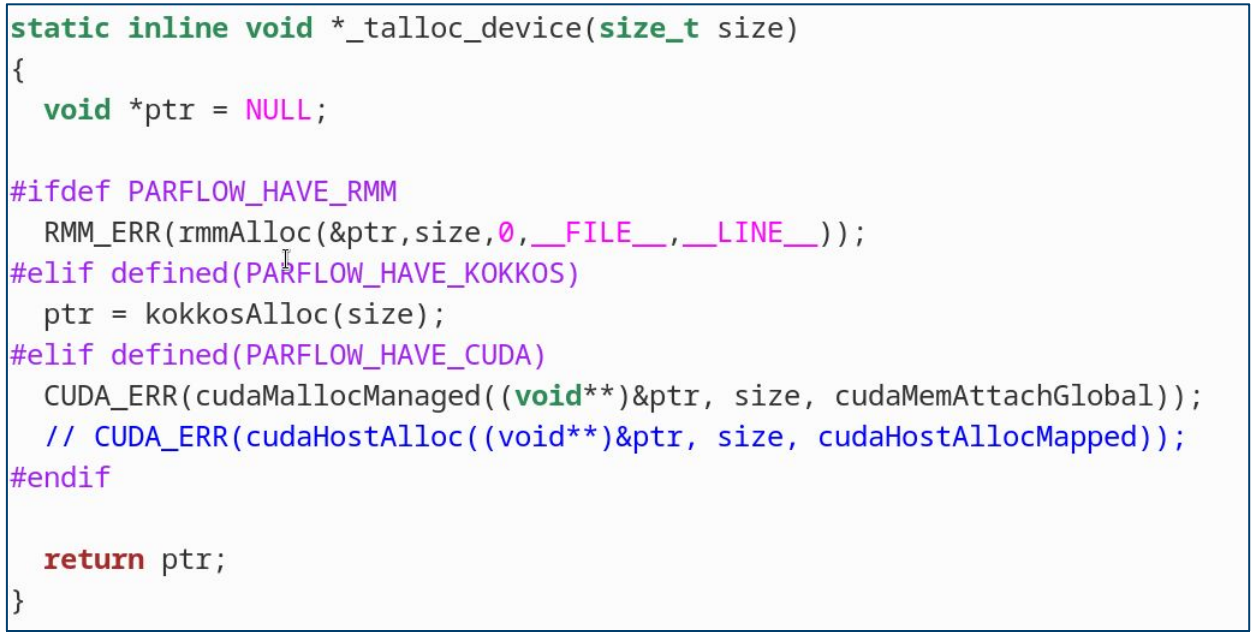


```c
#if defined(talloc_cuda) || defined(talloc_kokkos) ||
defined(talloc_omp)
  #define talloc CHOOSE_BACKEND(DEFER(talloc), ACC_ID)
#else
  #define talloc talloc_default
#endif
```

From `parflow/pfsimulator/parflow_lib/pf_cudamalloc.h`



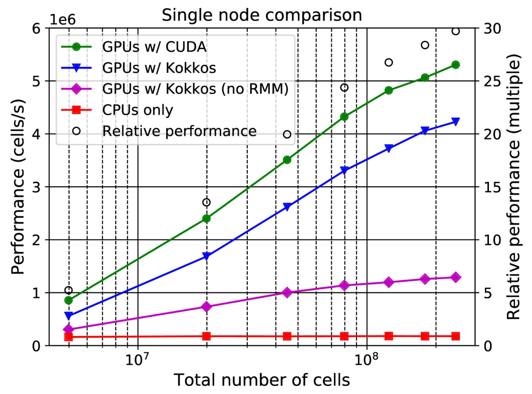
```c
#define talloc_cuda(type, count) \
  ((count) ? (type*)_talloc_device(sizeof(type) * (unsigned int)(count)) : NULL)
```

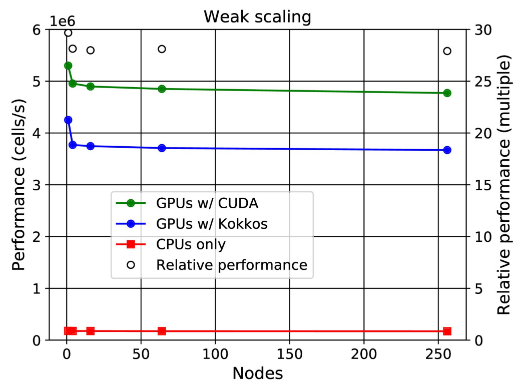


```c
static inline void *_talloc_device(size_t size)
{
  void *ptr = NULL;

#ifdef PARFLOW_HAVE_RMM
  RMM_ERR(rmmAlloc(&ptr,size,0,__FILE__,__LINE__));
#elif defined(PARFLOW_HAVE_KOKKOS)
  ptr = kokkosAlloc(size);
#elif defined(PARFLOW_HAVE_CUDA)
  CUDA_ERR(cudaMallocManaged((void**)&ptr, size, cudaMemAttachGlobal));
  // CUDA_ERR(cudaHostAlloc((void**)&ptr, size, cudaHostAllocMapped));
#endif

  return ptr;
}
```

ParFlow's performance-portability story

```
double *fp;
double *fp;
double value;
Subvector *f_sub;

/* some code missing here*/

for(k = iz; k < iz + nz; k++)
    for(j = iy; j < iy + ny; j++)
        for(i = ix; i < ix + nx; i++)
        {
            int ip = SubvectorEltIndex(f_sub, i, j, k);
            fp[ip] = pp[ip] - value;
        }
```





MPI + eDSL → MPI + eDSL (CUDA) → MPI + eDSL (CUDA, Kokkos[CUDA]) → MPI + eDSL (CUDA, Kokkos[CUDA,HIP])

2020 2021 2022 2023

NVIDIA CUDA kokkos natESM ROCm

```
double *fp;
double *fp;
double value;
Subvector *f_sub;

/* some code missing here*/

BoxLoopI0(i, j, k, ix, iy, iz, nx, ny, nz,
        {
            int ip = SubvectorEltIndex(f_sub, i, j, k);
            fp[ip] = pp[ip] - value;
        });
```

eDSL

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  auto lambda_body = [=] __host__ __device__
    (const int i, const int j, const int k)
      loop_body;

  /* some code missing for grid & block sizes */

  BoxKernelI0<<<grid, block>>>(lambda_body,
    ix, iy, iz, nx, ny, nz);
}
```

CUDA

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  auto lambda_body = KOKKOS_LAMBDA(int i, int j, int k)
  {
    i += ix; j += iy; k += iz;
    loop_body;
  }

  MDPolicyType_3D mdpolicy_3d({{0, 0, 0}},{{nx, ny, nz}});
  Kokkos::parallel_for(mdpolicy_3d, lambda_body);
}
```

Kokkos

Hokkanen et al. 2021. doi: 10.1007/s10596-021-10051-4

Take home messages

- There is a growing variety of hardware which requires a **variety of programming models**
- Most of ESM codes still struggle with **legacy effects**
- **No one-size-fits-all** solution available, but good solutions are available
- Porting in some way is likely unavoidable, **get started!**

- **Key question**: how to write parallel code and how to allocate memory in a hardware/vendor-agnostic way? **Two ideas in this talk.**
- **Where to start**: port (some part of) your code
- Manage your **expectations:** you will get speed up, don't expect optimal performance (but that's ok)
- Assume you will have to port again: **separate and abstract**
- We should **learn the lesson**: develop assuming disruptive changes may happen again

3rd NatESM Training Workshop

Two ESM Experiences of Performance Portability: Born Into It or Maturing Into It

Daniel **Caviedes-Voullième**
SDL Terrestrial Systems (JSC/FZJ)

06.11.2024