Performance Portability for Fortran CFD Software with GALÆXI

Spencer Starr
University of Stuttgart
09 December 2025

CEEC
Centre of Excellence in Exascale CFD

- Numerics Research Group (NRG) from the Institute of Aerodynamics and Gasdynamics, University of Stuttgart
- Research Interests:
  - Numerical methods for scale-resolving CFD
  - CFD software for HPC
- ***FLEXI*** and ***GALAEXI***
  - High-order discontinuous Galerkin spectral element method (DGSEM) CFD codes
  - Computations on CPU (***FLEXI*** and ***GALAEXI***) and NVIDIA/AMD GPUs (***GALAEXI***)

https://numericsresearchgroup.org

Starting Point

CEEC

You Have:

A CPU-only Fortran scientific code

Or more than one vendor of GPUs

A need to run it on GPUs

And can't use "off the shelf" solution

Basic knowledge of GPU computing

Overview

CEEC

PART 1

- GPU hardware overview
  - Compute and memory hierarchy
- GPU programming concepts
  - Kernels, threads, etc.
- Managing memory in CUDA/HIP C++
- Writing kernels in CUDA/HIP C++
- Porting process overview

PART 2

- Introduction to GALÆXI
- Portability with CUDA/HIP C++
- Managing memory and device state
- Multiple backends
- MPI

Part 1

Overview of GPU Computing

CEEC

Centre of Excellence in Exascale CFD

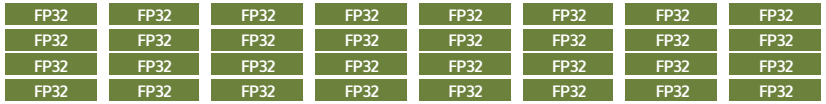
GPU Threads - **Hardware**

1 Core

FP32

==

1 Thread



- **Warp / Wavefront**
  - ***Hardware*** group of threads
    - 32 for NVIDIA, 64 for AMD
  - Warps created on kernel launch and assigned to cores
    - # of warps running limited by architecture
      - Limiting factor can be compute or memory usage
    - You always get the ***FULL*** warp

GPU Threads - **Software**

CEEC

THREAD

Single stream of execution on a GPU

1 software thread == 1 hardware thread

Execute independently

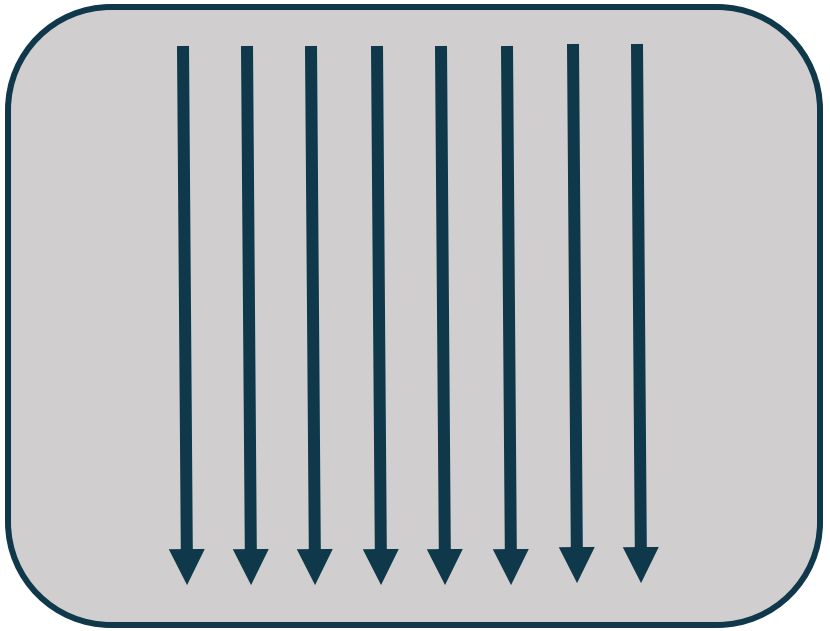
GPU Threads - **Software**

CEEC

BLOCK

Software group of threads

Block ≠ Warp



GPU Threads - **Software**

Blocks

- *Software* abstraction
- Arbitrary number of threads
- Requested by developer
- Can consist of multiple warps

Warps

- *Hardware* abstraction
- Always have 32/64 threads
- Created by hardware

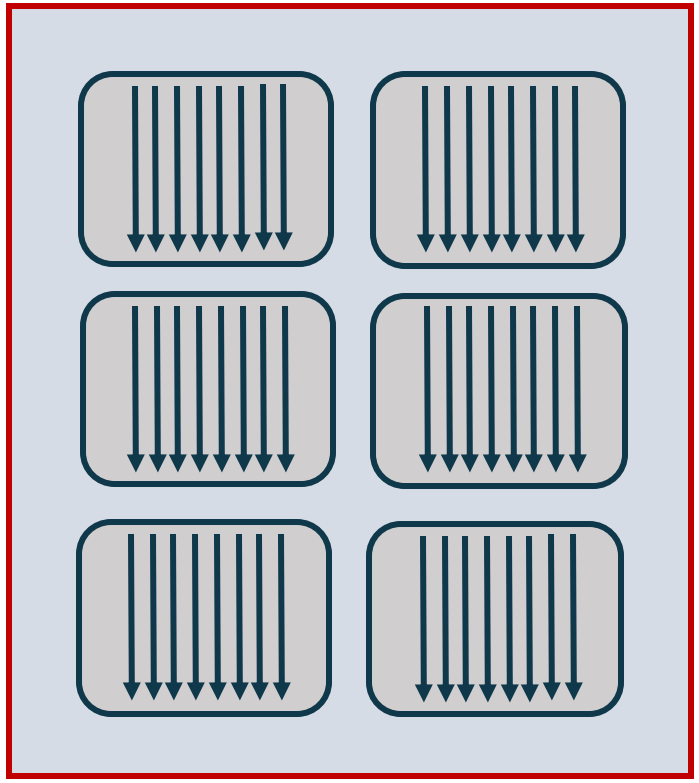
GPU Threads - **Software**

GRID

Group of blocks

All threads in a ***kernel***

Abstraction useful for mapping problems to hardware



GPU Memory

CEEC

Registers

- Local to a kernel
- Very limited in number (register spilling)
- ***Ex.*** Local temporary variable inside kernel function

Fastest

Shared Memory

- Allocated in caches
- Shared among all threads in a ***thread block***
- ***Ex.*** Any variable with the `__shared__` decorator

Fast

Global Memory

- L2 cache & HBM VRAM
- Shared among all threads on device
- ***Ex.*** Global device variable allocated with `cuda/hipMalloc`

Slow

Memory Latency on GPUs

CEEC

GPUs are *throughput machines*

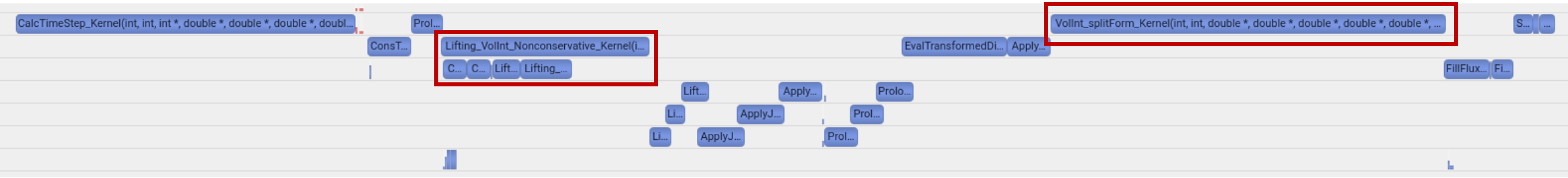
- GPUs compute ***far*** faster than they load data
- Must ***oversubscribe*** with compute to hide memory latency
  - Warp scheduler replaces warps waiting on memory with ones ready to compute

Occupancy

Measure of how much work the GPU is currently assigned

$$\text{Occupancy} = \frac{\text{Active Warps}}{\text{Max Warps Device}}$$

- **Streams**
  - Modern GPUs allow *multiple streams* of computation at once
  - Multiple *kernels can run in parallel* on single GPU
    - ***More oversubscription!***



Profile of GALÆXI with multiple compute streams
Blue blocks** are **running kernels

Launching Kernels in CUDA/HIP C++

- **Kernel**
  - Function that runs on the GPU
  - Looks like a regular function in code
  - Each thread runs the kernel ***independently***
    - Except for global data modifications and serializations (e.g. atomics)

```
KernelFuncName<<<num_blocks,num_threads_block,block_shared_mem_size,streamID>>>(ARGS);
```

Number of thread blocks

Number of threads per block

Amount of dynamic shared memory per block (optional)

ID of the device stream (optional)

Writing Kernels in CUDA/HIP C++

CUDA/HIP C++ code for an FMA operation on GPUs

```
__global__ void fma_kernel(int n, double alpha, double *x, double *y)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  y[i] = alpha*x[i] + y[i];
}
```

Fortran code for an FMA operation on CPUs

```
subroutine fma(n, alpha, x, y)

    do i=1,n
        y(i) = alpha*x(i) + y(i)
    end do

end subroutine fma
```

- `__global__` function decorator
  - Called from CPU, runs on GPU
  - `__device__` function decorator
- Arguments
  - All arguments are passed by value
  - Pointers are *pre-allocated pointers to global VRAM*
- Thread index
  - Above code assumes number of threads matches size of data ***exactly***

Part 2

Application to Scientific Software with GALÆXI

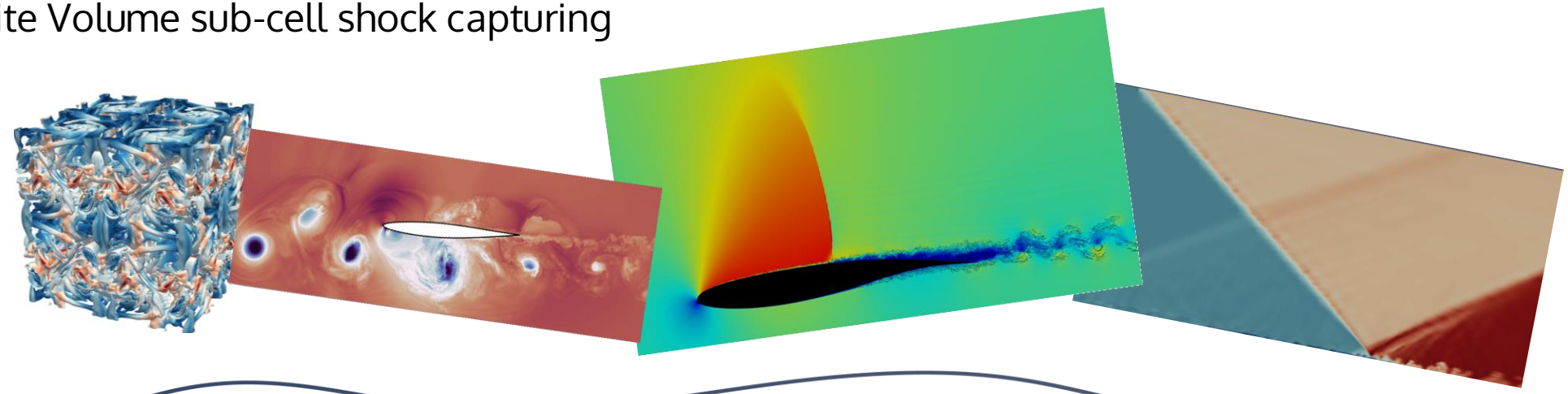
CEEC

Centre of Excellence in Exascale CFD

GALÆXI

CEEC

- *Open source* HPC solver for *unsteady, compressible* Navier-Stokes eqns.
- High-order discontinuous Galerkin spectral element method (*DGSEM*)
- Original CPU code (FLEXI) written in *Fortran*
- GPU offloading using *MPI-aware CUDA/HIP C++* for NVIDIA and AMD GPUs
- Some current features:
  - Variable polynomial degree (spatial order)
  - Several Riemann solver options
  - Finite Volume sub-cell shock capturing



Portability Approaches

• OpenMP • OpenACC	Pragma based
• Kokkos • RAJA • ALPAKA	Abstraction libs

- OpenCL
- Standard parallelism (coarray Fortran)
- Vendor-native languages (CUDA & HIP)
  - CUDA available in Fortran, HIP is not

Require Fortran-to-C interface

Options for Fortran Codes

CEEC

OpenMP

- *Stay all Fortran*
- *Not as easy as sold to be*
- *Performance harder to achieve*

Fortran-C Interfaces

- *C more flexible on GPUs*
- *Easier to get performance*
- *High effort to implement*

Fortran + CUDA/HIP C++: **Decisions**

>> Available personnel, timeline, work hours, skill sets?

>> How much code you want to keep/change?

>> Do you want to retain CPU-only computations?

>> Could you use an existing CUDA/HIP library (e.g. cuBLAS)?

Fortran + CUDA/HIP C++: **Decisions**

CEEC

How will you …

- Add GPU dependencies to your build system?
- Allocate/manage GPU memory?
- Pass data from Fortran to C?
- Map data to GPU threads?
- Do inter-GPU parallelism?
- Split between CPU and GPU computations?

GALÆXI Design Principles

1. Retain general data structure and parallelization strategy of original CPU code
2. Retain majority of non-solver, Fortran code base
3. All routines called during the time-stepping are executed on the accelerator without the need for data transfers (except for file I/O)

Minimize invasive changes, reduce effort, streamline validation

- Retain full support for CPU computing
- *Data to thread mapping?*
  - Each GPU threads works on a single DOF
- *Inter-GPU parallelization?*
  - Distributed approach with GPU-aware MPI
- *Intra-GPU parallelization?*
  - Use devices streams to launch multiple kernels concurrently

High effort

CUDA/HIP & GPU hardware limitations

Fortran to C interface

GALÆXI Approach

CEEC

Fortran == **CPU**

C (& CUDA / HIP) == **GPU**

GALÆXI - GPU Memory Management

```fortran
ALLOCATE( U(PP_nvar,0:PP_N,0:PP_N,0:PP_N,nElems) )

#if (USE_ACCEL != ACCEL_OFF)
    CALL AllocateDeviceMemory(d_U, SIZE_C_DOUBLE, SIZE(U))
#endif
```

Similar interfaces used for all CUDA/HIP API calls (e.g. synchronizes)

- Allocate multi-dim Fortran arrays as 1D on GPU
  - Indexing functions for GPU threads
- Store pointers to GPU memory in hash map on host
  - Retrieve via key created at allocation
  - Keys stored in Fortran variables that mirror variable name (e.g. `Ut` and `d_Ut`)
- Can use any allocator on C++ side (CUDA, HIP, OpenCL, etc.)

```fortran
void AllocateDeviceMemory_Device(int dVarKey, size_t typeSize_bytes, int arraySize)
{
    // Initialize temporary device pointer
    void* d_arr;

    // Call memAlloc API for specific vendor
#if (USE_ACCEL == ACCEL_CUDA)
    DEVICE_ERR_CHECK( cudaMalloc(&d_arr, typeSize_bytes*arraySize) );
#elif (USE_ACCEL == ACCEL_HIP)
    DEVICE_ERR_CHECK( hipMalloc(&d_arr, typeSize_bytes*arraySize) );
#end

    // Store allocated device pointer
    DeviceVars[dVarKey] = d_arr;
}
```

GALÆXI - CPU Backends

```fortran
SUBROUTINE VAXPB_ADD(nTotal,VecOut,VecIn,d_Out,d_In)
IMPLICIT NONE
!-----------------------------------------------------------------
! INPUT/OUTPUT VARIABLES
INTEGER,INTENT(IN)        :: nTotal         !< vector length
REAL,INTENT(INOUT)        :: VecOut(nTotal) !< output vector
REAL,INTENT(IN)           :: VecIn(nTotal)  !< input vector
INTEGER(C_INT),INTENT(IN) :: d_Out,d_In     !< device pointer keys
!=================================================================

#if (USE_ACCEL == ACCEL_OFF)
  CALL VAXPB_ADD_Host(nTotal,VecOut,VecIn)
#else
  CALL VAXPB_ADD_Device(nTotal,d_Out,d_In)
#endif

END SUBROUTINE VAXPB_ADD
```

```fortran
SUBROUTINE VAXPB_ADD_Host(nTotal,VecOut,VecIn)
! MODULES
IMPLICIT NONE
!-----------------------------------------------------------------
! INPUT/OUTPUT VARIABLES
INTEGER,INTENT(IN)    :: nTotal           !< vector length
REAL,INTENT(INOUT)    :: VecOut(nTotal)   !< output vector
REAL,INTENT(IN)       :: VecIn(nTotal)    !< input vector
!-----------------------------------------------------------------
! LOCAL VARIABLES
INTEGER               :: i
!=================================================================

  DO i=1,nTotal
    VecOut(i)=VecOut(i)+VecIn(i)
  END DO

END SUBROUTINE VAXPB_ADD_Host
```

GALÆXI - GPU Backends

```fortran
SUBROUTINE VAXPB_ADD(nTotal,VecOut,VecIn,d_Out,d_In)
IMPLICIT NONE
!-----------------------------------------------------------------
! INPUT/OUTPUT VARIABLES
INTEGER,INTENT(IN)        :: nTotal         !< vector length
REAL,INTENT(INOUT)        :: VecOut(nTotal) !< output vector
REAL,INTENT(IN)           :: VecIn(nTotal)  !< input vector
INTEGER(C_INT),INTENT(IN) :: d_Out,d_In     !< device pointer keys
!=================================================================

#if (USE_ACCEL == ACCEL_OFF)
  CALL VAXPB_ADD_Host(nTotal,VecOut,VecIn)
#else
  CALL VAXPB_ADD_Device(nTotal,d_Out,d_In)
#endif

END SUBROUTINE VAXPB_ADD
```

```
void VAXPB_ADD_Device(int nTotal,int d_VecOut,int d_VecIn)
{
    INVOKE_KERNEL(VAXPB_ADD_Kernel, nTotal/256+1, 256, 0, streams[0], nTotal,
                  (double*)DeviceVars[d_VecOut], (double*)DeviceVars[d_VecIn]
                 );
}
```

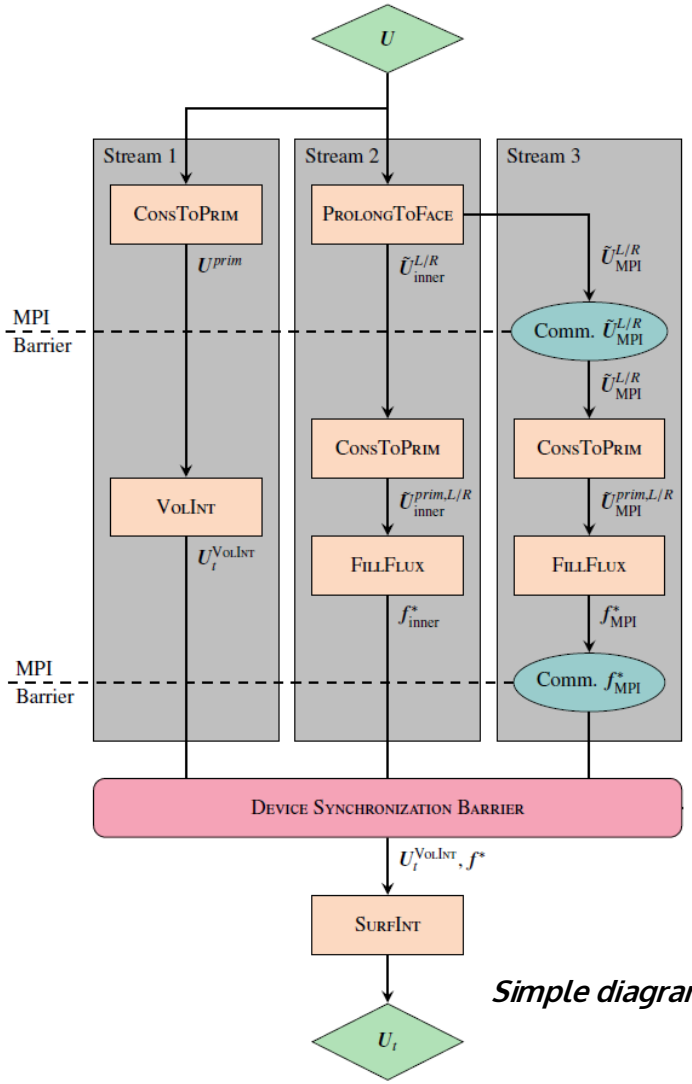
```
__global__ void VAXPB_ADD_Kernel(int nTotal, double* VecOut, double* VecIn)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i > nTotal) return;

    VecOut[i]=VecOut[i]+VecIn[i];
}
```

GALÆXI - Execution Strategy

1. Start computation
2. CPU allocates memory for itself and GPU
3. CPU initializes the solution and copies data to the GPU
4. GPU calculates solutions
    - GPU passes data back to CPU periodically for output to file
5. End computation

- ***Single CPU core*** pinned to ***single GPU***
    - Pinning handled using MPI and `cuda`/`hipSetDevice` in sofware
    - MPI calls made from C side and use device pointers

GALÆXI - Device Streams



Simple diagram of GALAEXI device streams

- Threes streams (HIGH, MED and LOW)
  - Kernels computing data for *MPI communication* given *highest priority*
  - Large, internal *volume-wise operations* given *lowest priority*

- Streams synchronized with ***Events***
  - Attach event to each kernel and track status
  - Downstream kernels wait for work on their data dependencies to complete

GALÆXI - Porting Process

Write unit test for existing CPU code

Useful for validation later!

Refactor/update existing CPU code

You're changing things already, might as well.

CPU code ➡ Host backend

Make CPU backend have same function call as GPU backend

Add GPU memory allocation/copies

Fortran-C interfaces for GPU backends

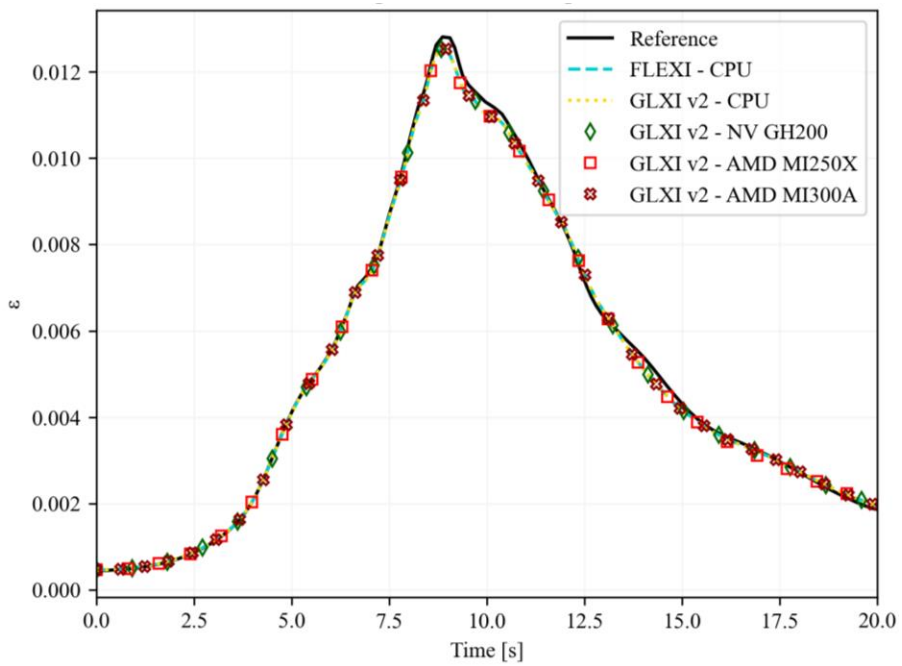
Write GPU kernel

Verify and validate

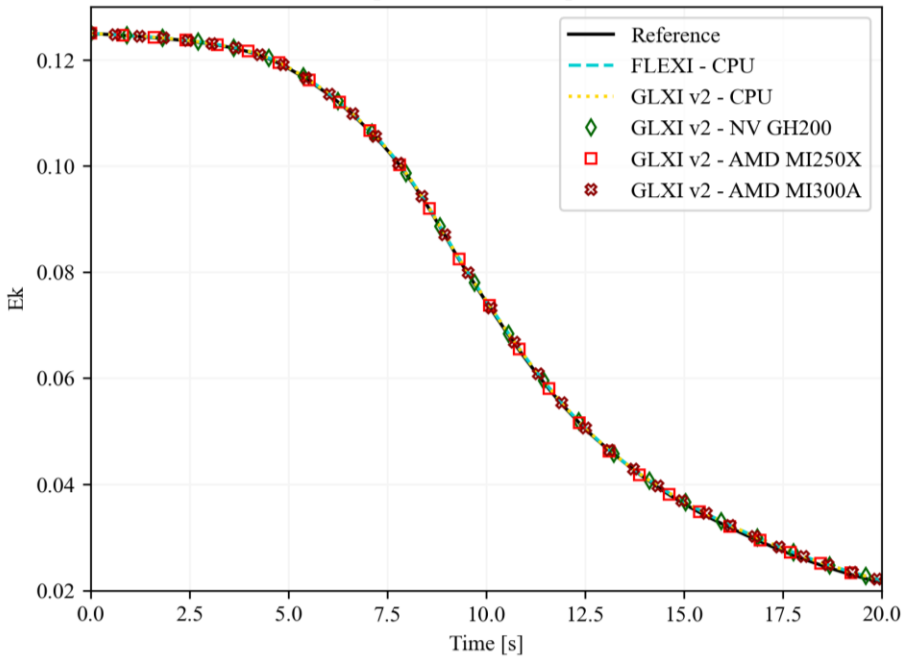
Remember those unit tests?

Validation – Taylor-Green Vortex

- ***Conditions***
  - Re = 1600 (Incompressible), viscous
  - ~17 million DOFs
    - 32x32x32 mesh at N=7
  - Gauss-Lobatto nodes
  - Split flux formulation

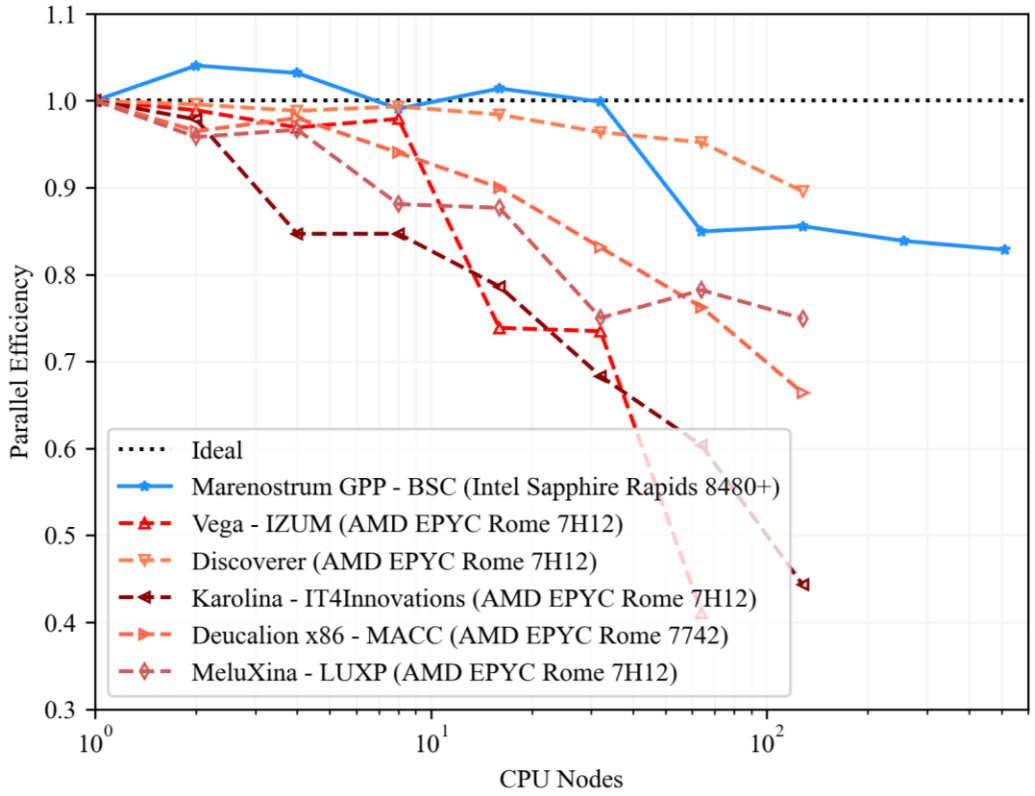


*Comparison of **dissipation rate** results for an incompressible TGV from* FLEXI *on CPUs and* GALÆXI *on 5 different architectures*

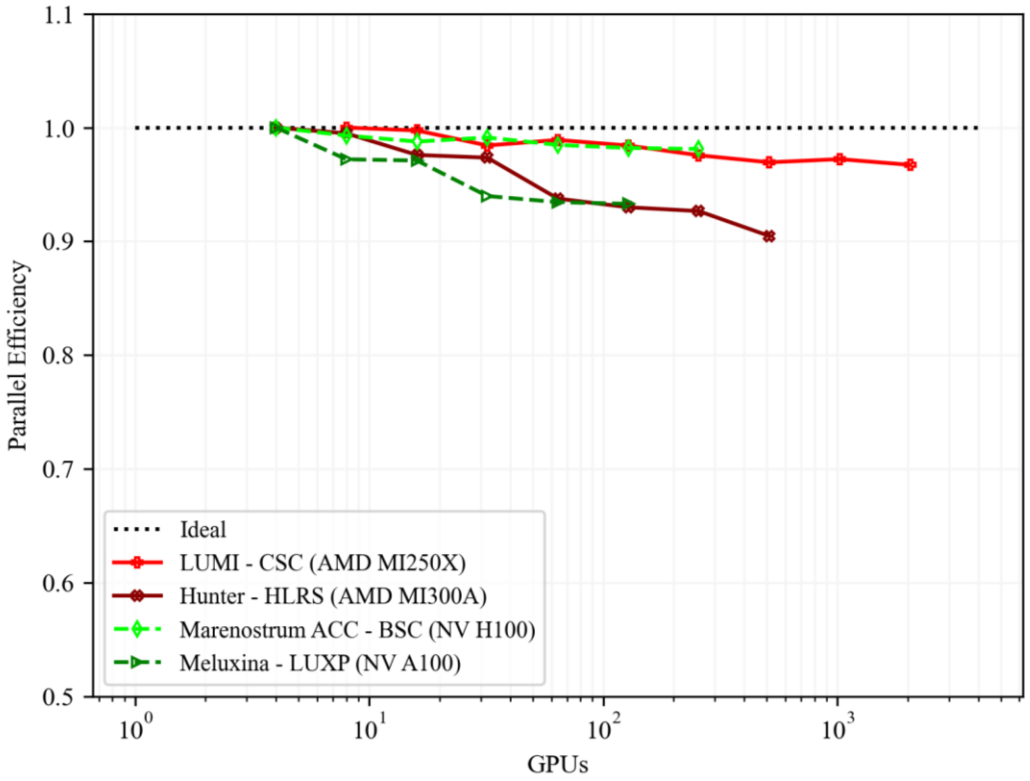


*Comparison of **turbulent kinetic energy** results for an incompressible TGV from* FLEXI *on CPUs and* GALÆXI *on 5 different architectures*

Performance – Weak Scaling

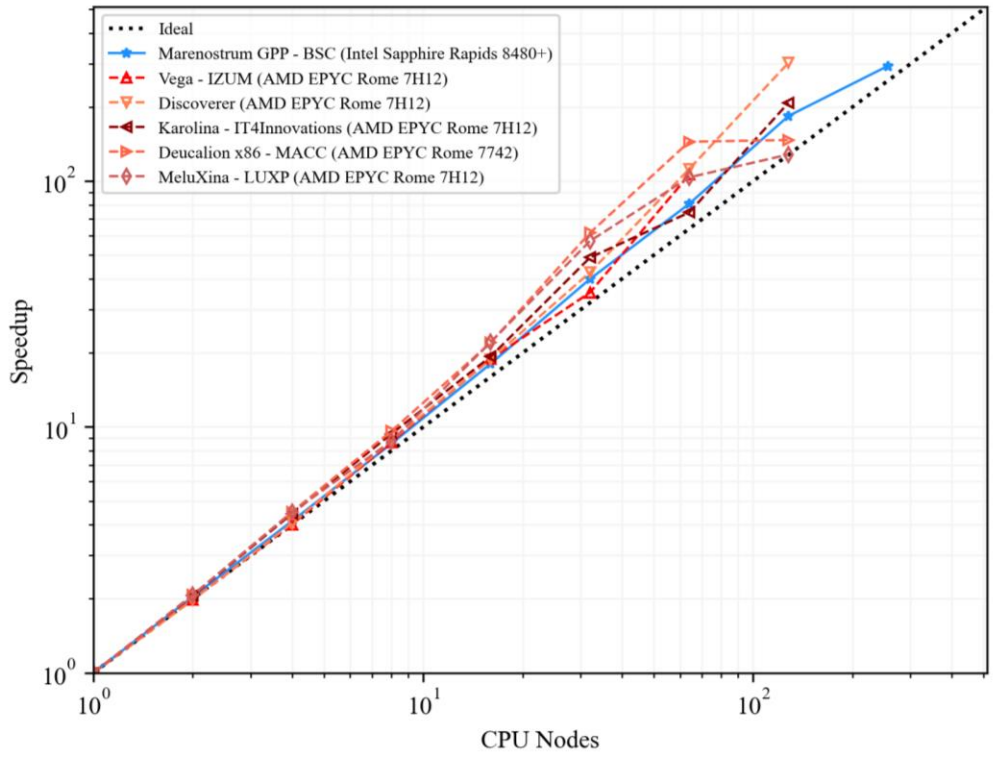


*GALÆXI **CPU** weak scaling results on various EuroHPC systems*

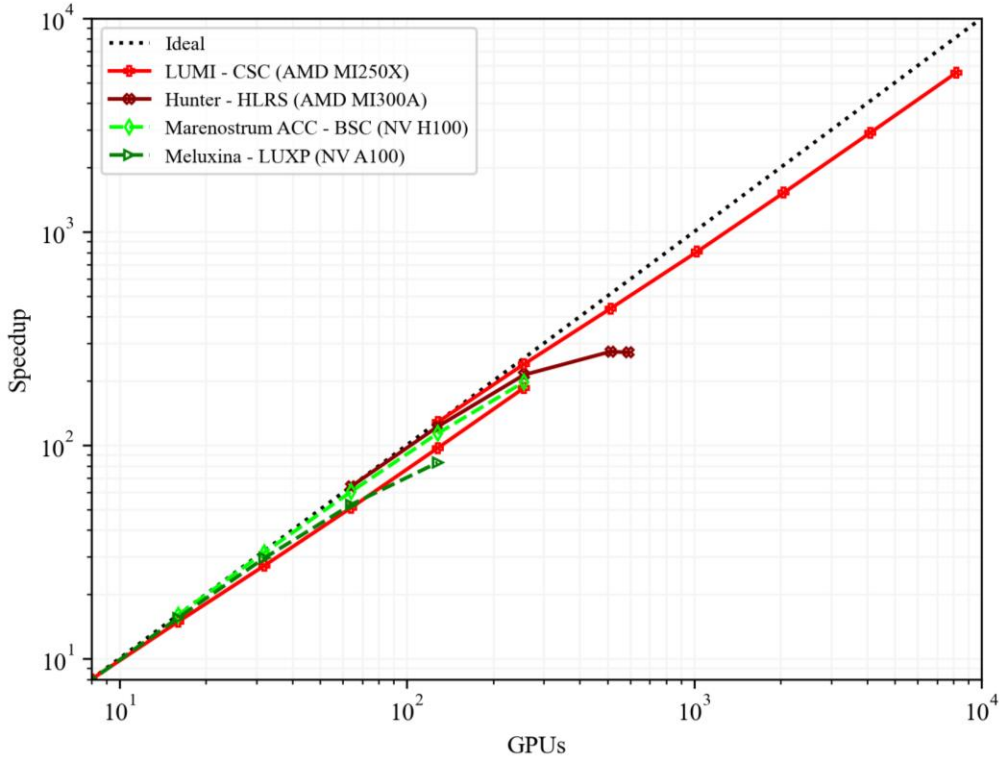


*GALÆXI **GPU** weak scaling results on various EuroHPC systems*

Performance – Strong Scaling

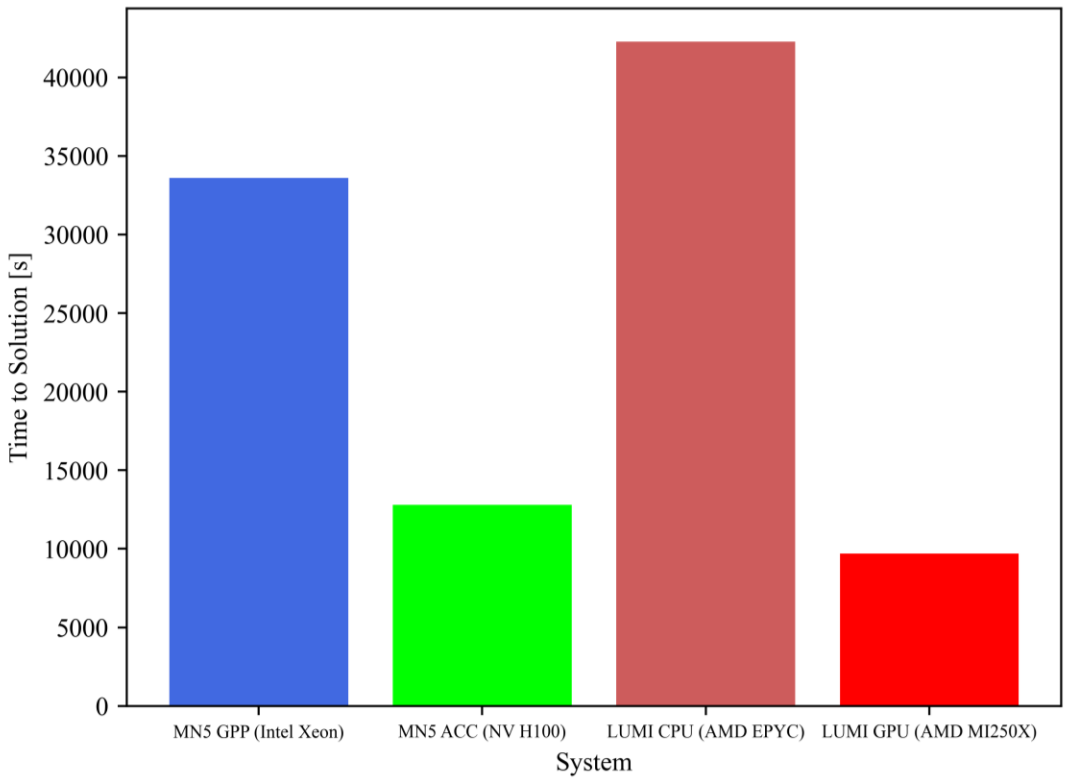


*GALÆXI **CPU** strong scaling results on various EuroHPC systems*

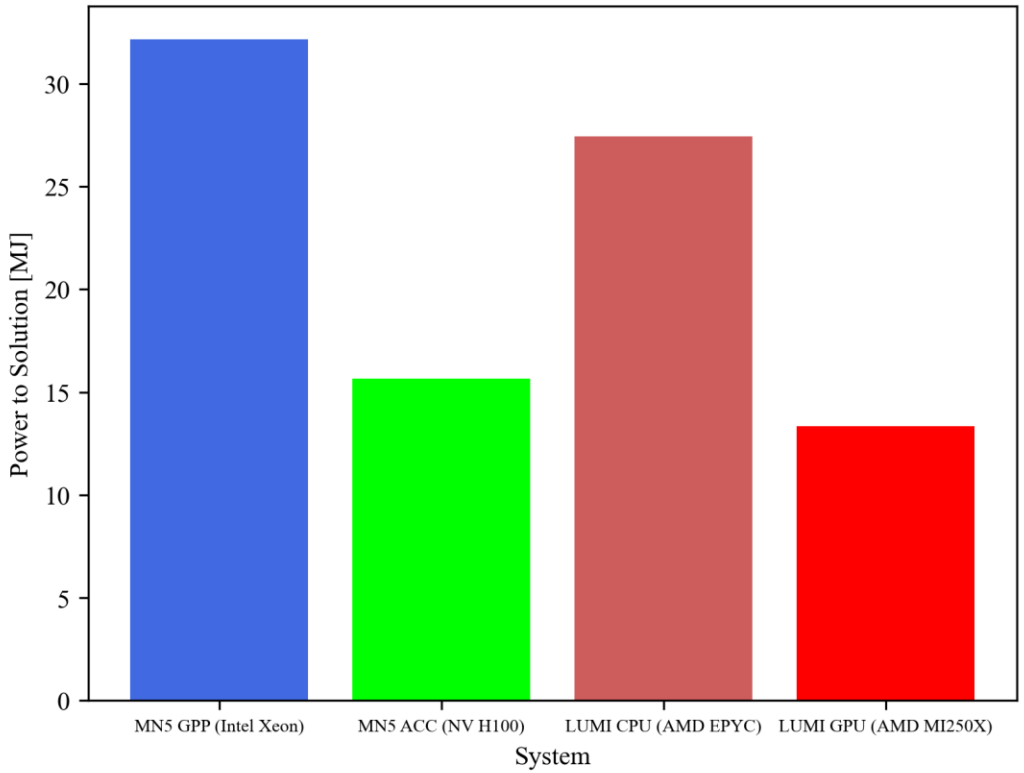


*GALÆXI **GPU** strong scaling results on various EuroHPC systems*

Performance – Time/Energy To Solution



*GALÆXI **time** to solution on single node of various EuroHPC systems for problem with 16 million DOFs*



*GALÆXI **energy** to solution on single node of various EuroHPC systems for problem with 16 million DOFs*

Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Sweden, Germany, Spain, Greece, and Denmark under grant agreement No 101093393.

Co-funded by the European Union

EuroHPC Joint Undertaking

Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European High Performance Computing Joint Undertaking (JU) and Sweden, Germany, Spain, Greece, and Denmark. Neither the European Union nor the granting authority can be held responsible for them.

CEEC

Centre of Excellence in Exascale CFD

Thank you
for your attention!

CEEC

Centre of Excellence in Exascale CFD

Where to Find Us!

https://numericsresearchgroup.org

https://github.com/flexi-framework