Extreme-scale High-Fidelity Computational Fluid Dynamics with NEKO

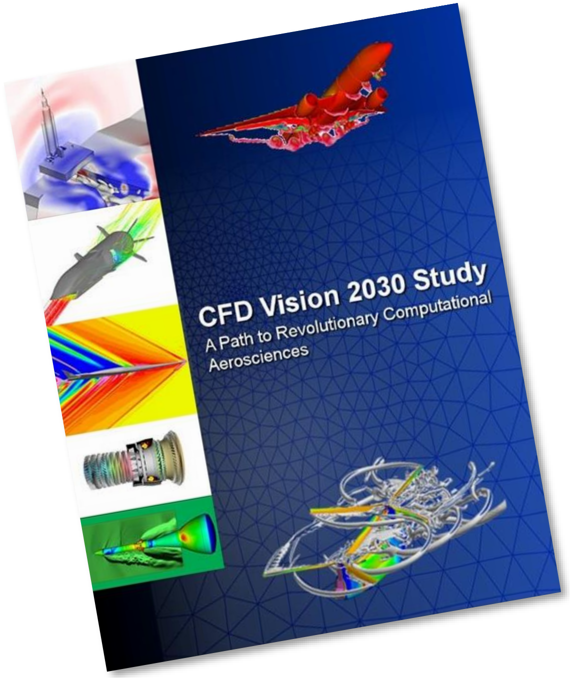
Niclas Jansson
KTH Royal Institute of Technology

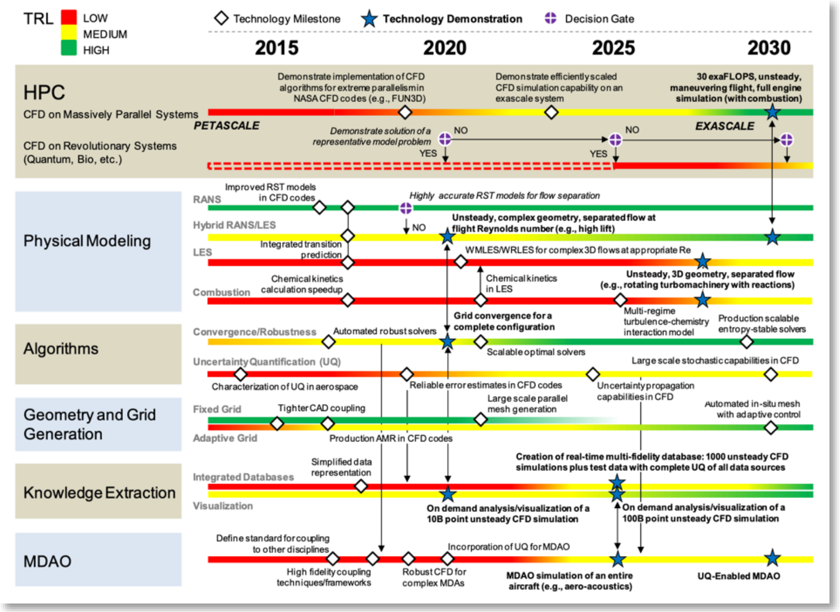
CEEC
Centre of Excellence in Exascale CFD

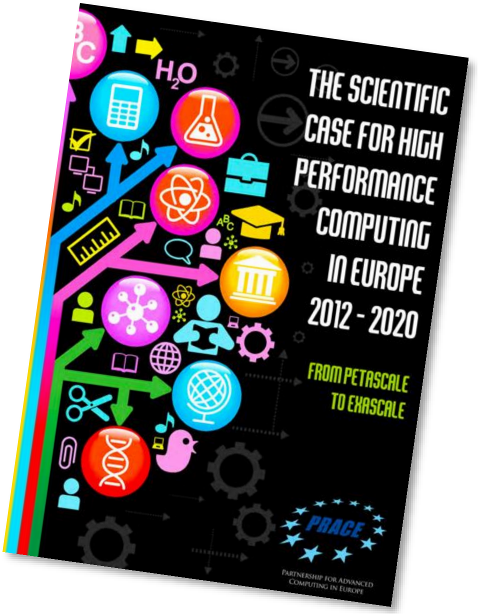
Introduction

CEEC

About 10% of the energy use in the world is spent overcoming turbulent friction







No upper limit in fluid dynamics to the size of the systems to be studied via simulations

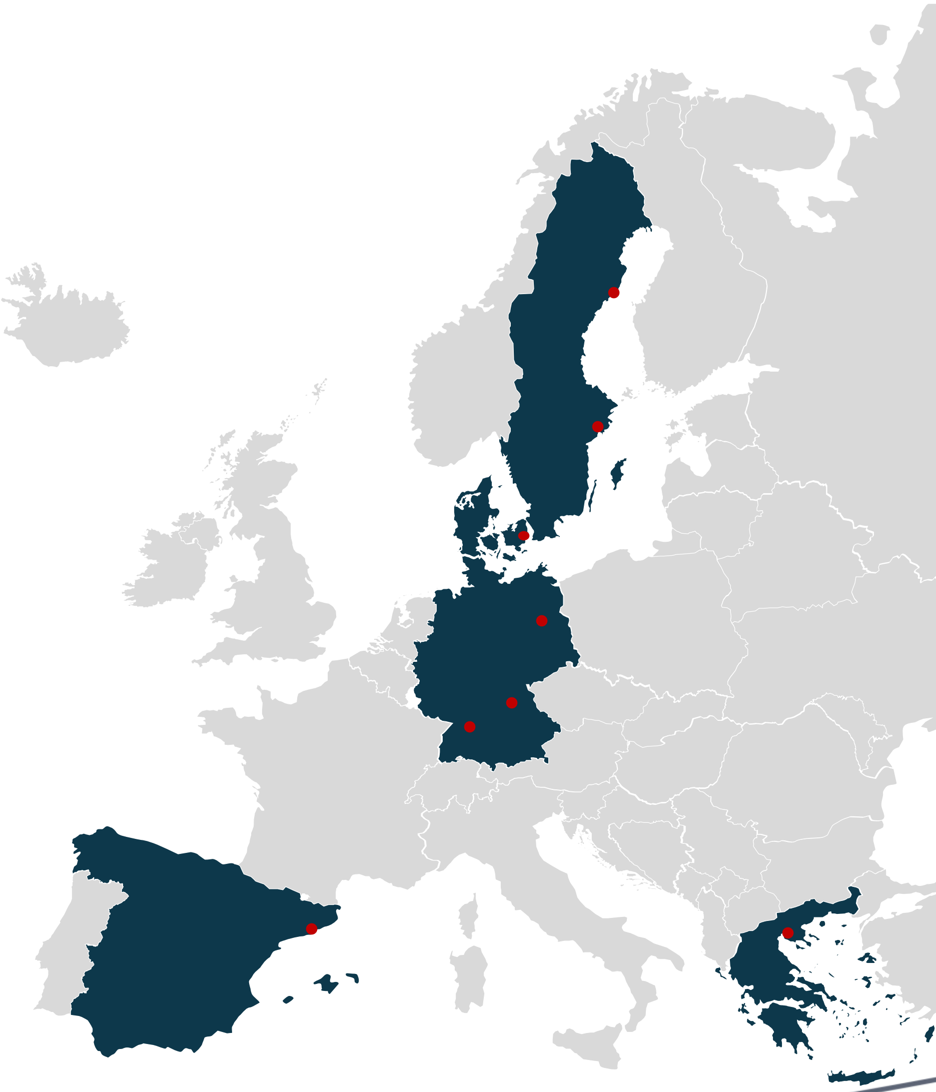
Computational Fluid Dynamics is one of the areas with a clear need and **great potential to reach exascale**

CEEC

Centre of Excellence in Exascale CFD

The main goal of CEEC is to address the extreme-scale computing challenge to enable the use of accurate and cost-efficient high fidelity computational fluid dynamics (CFD) simulations at exascale

- Implement **exascale-ready workflows** for addressing grand challenge scientific problems
- Develop **new or improved algorithms** that can efficiently exploit exascale systems.
- Significantly improve **energy efficiency** of simulations
- Demonstrate workflows on **lighthouse cases** relevant for both academia and industry



KTH VETENSKAP OCH KONST
Universität Stuttgart
FAU Friedrich-Alexander-Universität Erlangen-Nürnberg
DTU
BAM
BSC Barcelona Supercomputing Center Centro Nacional de Supercomputación
UMEÅ UNIVERSITET

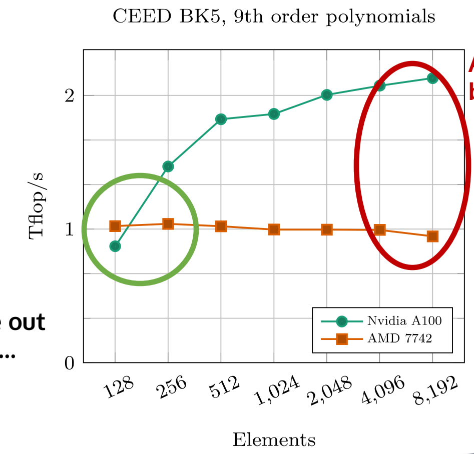
ceec-coe.eu

Introduction

- Exascale will require either **unreasonably large problem** sizes or **significantly improved efficiency** of current methods
  - Finite-Volume LES of a full car on the entire K computer (京) required **more than 100 billion grid points** to run efficiently
  - What problem size is needed to fill the 379 PFlop/s LUMI...
- High-order methods
  - Attractive numerical properties, **small dispersion** errors and more "accuracy" per degree of freedom
  - Better suited to take advantage of **modern hardware** (accelerators)

京: 82944 nodes, 663552 Cores, **10 PFlop/s**

Dardel: 56 nodes, 448 MI250X GCDs, ≈**10 PFlop/s**



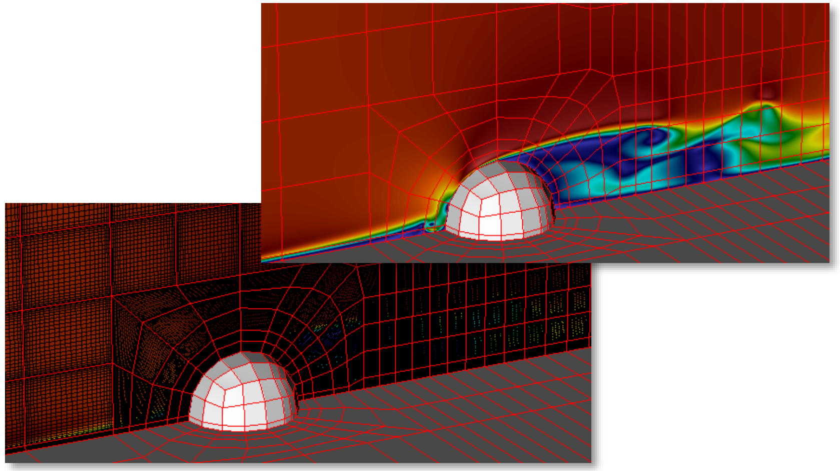
Accelerators works best with a lot of data!

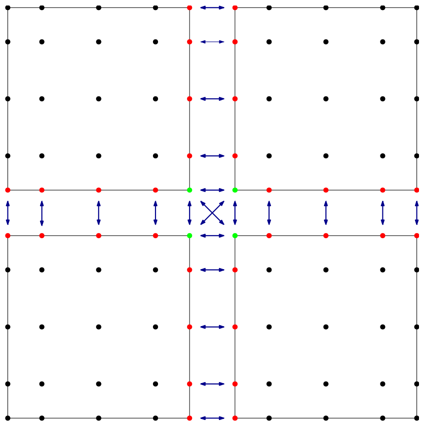
...but we rather scale out our problems in CFD...

Spectral Elements

CEEC

- Finite Elements with high-order basis functions
  - N-th order Legendre-Lagrange polynomials $l_i(\xi)$
  - Gauss-Lobatto-Legendre quadrature points ξ_i
  - Fast tensor product formulation
    - $u^e(\xi,\eta,\gamma) = \sum_{i,j,k}^{N} u_{i,j,k}^e \, l_i(\xi) l_j(\eta) l_k(\gamma)$
  - High-order at low cost! (**Level 3 BLAS!**)
- Too expensive to assemble matrices
  - Element stiffness matrices $A_{i,j}^k$ with $\boldsymbol{O}(\boldsymbol{N}^6)$ **non-zeros**
- Matrix free formulation, key to achieve good performance in SEM
  - Unassembled matrix $A_L = \mathrm{diag}\{A^1, A^2, \dots, A^E\}$ and functions $u_L = \{u^e\}_{e=1}^E$
  - Operation count is **only** $\boldsymbol{O}(\boldsymbol{N}^4)$ **not** $\boldsymbol{O}(\boldsymbol{N}^6)$
  - Boolean gather/scatter matrix Q^T and Q
    - Ensure continuity of functions on the element level $u = Q^T u_L$ and $u_L = Qu$
- Q nor Q^T formed, only the action QQ^T is used
  - Matrix-vector product $w = Au \Rightarrow w_L = QQ^T A_L u_L$



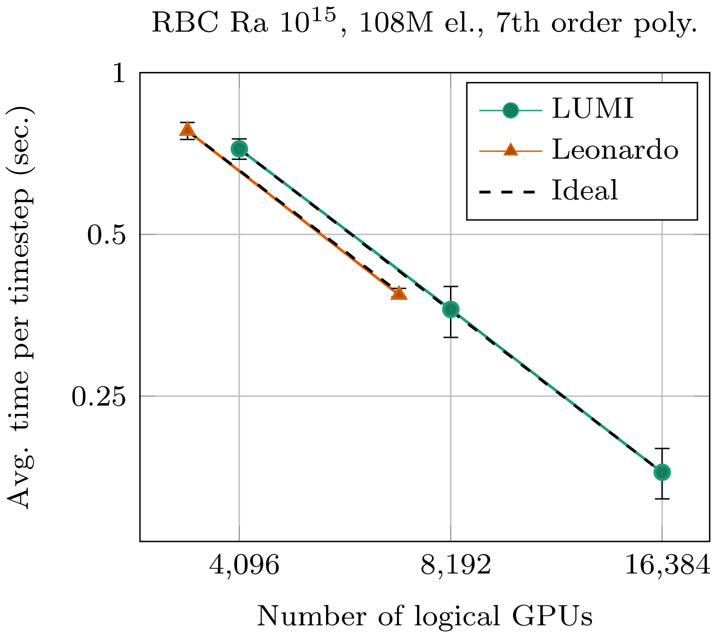


Portable Spectral Element Framework NEKO

- High-order spectral element flow solver
  - Incompressible Navier-Stokes equations
  - Matrix-free formulation, **small tensor products**
  - **Gather-scatter** operations between elements
  - Modern **object-oriented** approach (Fortran 2008)
  - Support for various hardware-backends
    - **Device abstraction** layer for accelerators (CUDA/HIP/OpenCL)
  - **Insitu/Intransit** interfaces (visualization/post-processing)
  - Python based (offline) post-processing suite (**pySEMTools**)
- Recent developments
  - Turbulence modelling (LES models)
  - Immersed boundaries
  - API
- Modern Software Engineering (pFUnit, ReFrame, **Spack**)

```
> spack install neko+cuda
```

ExtremeFLOW/neko



Sustained near **perfect strong scalability** on up to 80% of the pre-exascale supercomputer LUMI

www.neko.cfd

ACM Gordon-Bell prize finalist 2023

Device Abstraction Layer

CEEC

How to interface Fortran with accelerators?

- Native CUDA/HIP/OpenCL implementation via C-interfaces
- Device pointers in each derived type

```
type field_t
   real(kind=rp), allocatable :: x(:,:,:,:) !< Field data
   type(space_t), pointer :: Xh     !< Function space
   type(mesh_t), pointer :: msh     !< Mesh
   type(dofmap_t), pointer :: dof   !< Dofmap
   type(c_ptr) :: x_d = C_NULL_PTR !< Device pointer
end type field_t
```

- Abstraction layer hiding memory management
- Hash table associating x with x_d
- Kernels invoked from the object hierarchy via C interfaces (Ax, vector ops)
  - **Wrapper functions** for each supported accelerator backend
  - **Templated** (CUDA/HIP) or **pre-processor macros** (OpenCL) for runtime parameters
- **Auto/runtime tuning** based on polynomial order

```
src/
|
|-- math
|   `-- bcknd
|       |-- cpu
|       |-- device
|       |   |-- cuda
|       |   |-- hip
|       |   `-- opencl
|       |-- sx
|       `-- xsmm
```

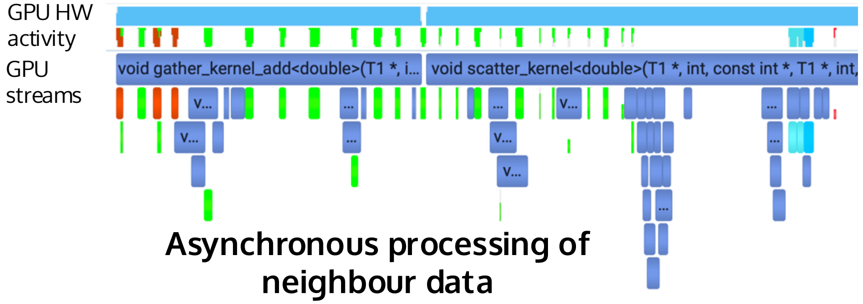
```
!> Enum @a hipError_t
enum, bind(c)
   enumerator :: hipSuccess = 0
   ...
end enum

!> Enum @a hipMemcpyKind
enum, bind(c)
   enumerator :: hipMemcpyHostToHost = 0
   enumerator :: hipMemcpyHostToDevice = 1
   ...
end enum

interface
   integer (c_int) function hipMalloc(ptr_d, s) &
        bind(c, name='hipMalloc')
     use, intrinsic :: iso_c_binding
     implicit none
     type(c_ptr) :: ptr_d
     integer(c_size_t), value :: s
   end function hipMalloc
end interface
```

```
subroutine field_init(f,…)
type(field_t) :: f
...
call allocate(f%x(…,…,…,…,)
call device_alloc(f%x_d, size)
call device_associate(f%x, f%x_d)
```

cudaMalloc | hipMalloc | clCreateBuffer

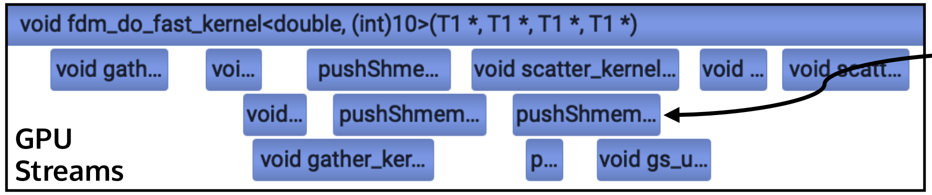
Gather-Scatter

- Neko supports various communication backends for gather-scatter
  - **MPI**, **NVSHMEM**, **NCCL/RCCL** (point-to-point)
- Multiple levels of overlapping communication and computation
  - Overlapping with **non-blocking MPI** (device aware)
  - **Asynchronous** GPU kernels (neighbours in streams)
  - **Auto/runtime** tuning of all combinations

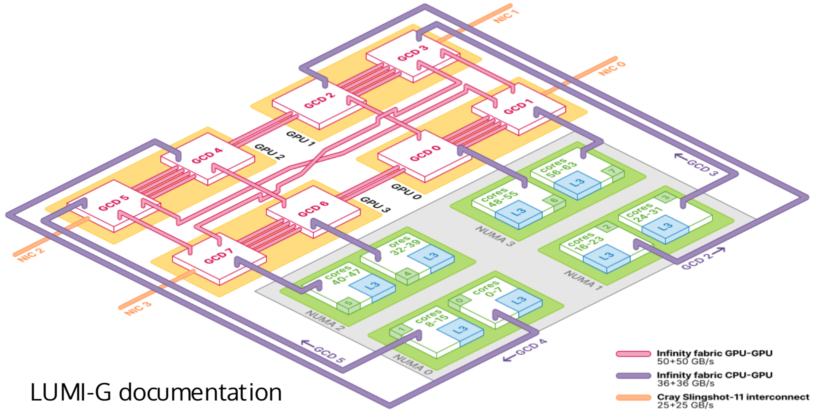


Asynchronous processing of neighbour data

- Using NVSHMEM, both computation and communication "**scheduled by the accelerator**"



SHMEM comm. as device kernels



LUMI-G documentation

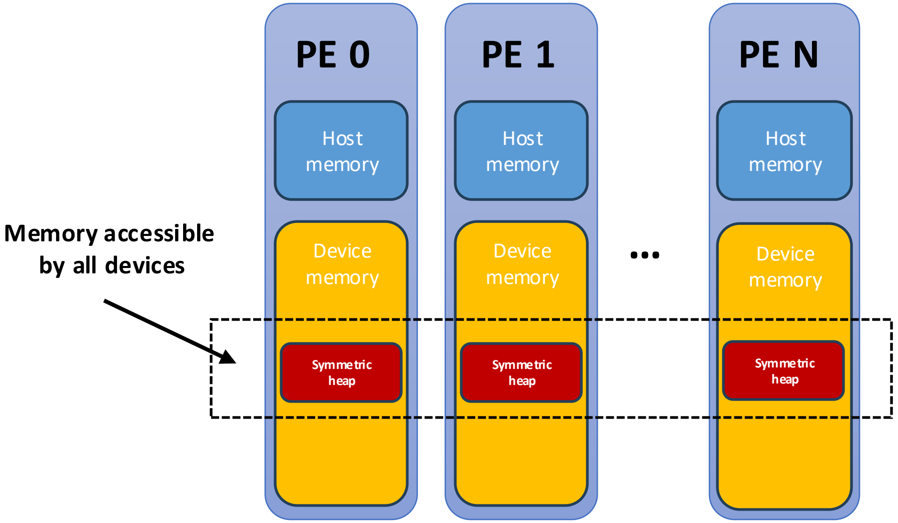


Illustration of SHMEM

Numerical Method $P_N - P_N$

- Time integration is performed using an implicit-explicit scheme (BDFk/EXTk)

$$\sum_{j=0}^{k} \frac{b_j}{dt} u^{n-j} = -\nabla p^n + \frac{1}{Re} \nabla^2 u^n + \sum_{j=1}^{k} a_j \left(u^{n-j} \cdot \nabla u^{n-j} + f^n \right)$$

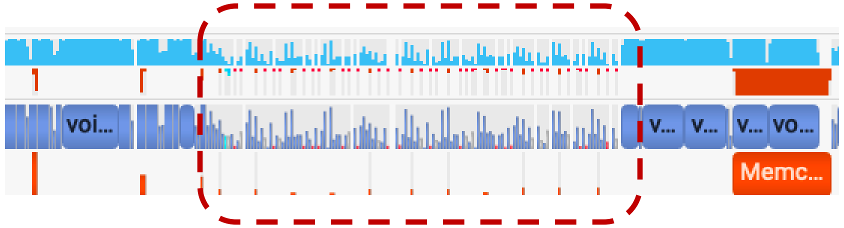
with b_k and a_k coefficients of the implicit-explicit scheme, solving at time-step n

$$\nabla^2 p^n = \nabla \cdot \left(\sum_{j=1}^{k} a_j \left(u^{n-j} \cdot \nabla u^{n-j} + f^n \right) \right)$$

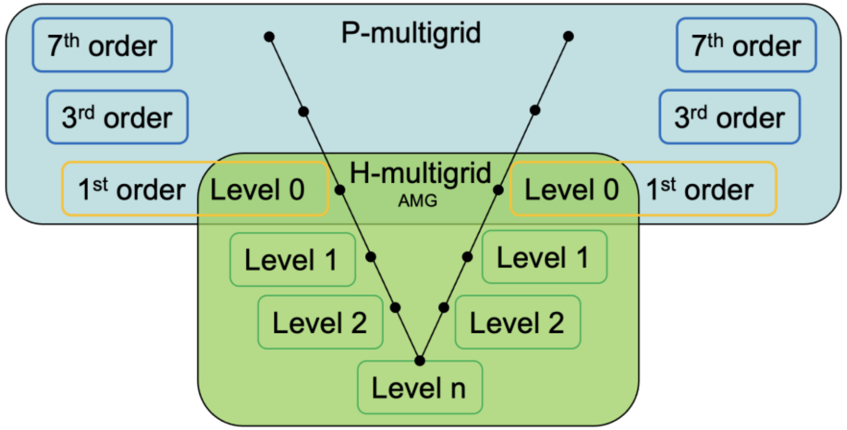
$$\frac{1}{Re} \nabla^2 u^n - \frac{b_0}{dt} u^n = \nabla p^n + \sum_{j=1}^{k} \left(\frac{b_j}{dt} u^{n-j} + a_j \left(u^{n-j} \cdot \nabla u^{n-j} + f^n \right) \right)$$

Scalable Linear Solvers

- Communication is the **major bottleneck** for iterative solvers
- **Horizontal** communication (between PEs)
  - Global reductions: dot products, norms etc, $\boldsymbol{O(P \log P)}$ complexity
- Pipelined formulations
  - **Overlap** communication with computation
- **Vertical** communication (between levels in memory)
  - Model cost Q with a CDAG $G = (V, E)$
    - $u, v \in V$: result of a computation, $(u, v) \in E$: dependency on a result
  - **Re-materialization** or re-computation of e.g., A_L (flops are for free)
- Fused (**kernel fusion**) and/or coupled formulations (**multiple vectors**)
- Current Krylov solvers
  - GMRES, CG, PipeCG, FusedCG
- Efficient preconditioners
  - Additive overlapping Schwarz multigrid
  - Matrix-Free Algebraic hp-Multigrid



Fuse kernels (if possible)



Installation

To build Neko, you will need a Fortran compiler supporting the Fortran-08 standard, autotools, libtool, pkg-config, MPI with Fortran 2008 bindings (mpi_f08), BLAS/LAPACK and JSON-Fortran.

Manual installation

```
<path-to-neko>/configure FC=<Fortran compiler> CC=<C compiler> \
                         MPIFC=<MPI Fortran compiler> MPICC=<MPI C ompiler> \
                         FCFLAGS=<Fortran compiler flags> CFLAGS=<C compiler flags> \
                         --prefix=<installation path> [options]
```

[options] refers to either optional features or packages, enabled or disabled using

```
--enable-FEATURE[=arg] or --disable-FEATURE

--with-FEATURE[=arg] or --without-FEATURE
```

For more details see the installation section of the Neko manual

Installation – examples

CEEC

CUDA installation with device-aware MPI

```
./configure --with-cuda=/usr/local/cuda CUDA_CFLAGS=-O3  CUDA_ARCH=-arch=sm_90 --enable-device-mpi
```

HIP installation

```
./configure --with-hip=/opt/rocm/hip HIPCC=CC HIP_HIPCC_FLAGS="-O3 -x hip --offload-arch=gfx90a"
```

Single precision (FP32) CPU installation

```
./configure CFLAGS="-O3" FCFLAGS="-O3" --enable-real=sp --prefix=/opt/neko/1.0
```

For more examples see installation section of the manual

Setting up a case

- The **`time`** object
  - Variables related to simulation time e.g Time-stepping, step size and start/end time
- The **`numerics`** object
  - Defines properties of the numerical discretization e.g. polynomial order
- The **`fluid`** object
  - Setup of the fluid solver and flow problem
- The **`scalar`** object
  - Setup additional scalar(s) transport
- The **`simulation_components`** object
  - Defines additional components not necessary for running a solver e.g. computing vorticy, lambda2 etc.

Detailed documentation: [Neko user guide]

```
{
  "version": 1.0
  "case": {
    "time": {}
    "numerics": {}
    "fluid": {}
    "scalar": {}
    "simulation_components": []
  }
}
```

Setting up a case

Mesh file to load

Various I/O options

Time object

Numerics object

```
{
    "version": 1.0,
    "case": {
        "mesh_file": "cylinder.nmsh",
        "output_at_end": true,
        "output_boundary": true,
        "output_checkpoints": true,
        "checkpoint_control": "simulationtime",
        "checkpoint_value": 50,
        "time": {
            "end_time": 200.0,
            "variable_timestep": true,
            "target_cfl": 0.5,
            "max_timestep": 1e-1
        },
        "numerics": {
            "time_order": 3,
            "polynomial_order": 5,
            "dealias": true
        },
...
```

Setting up a case

CEEC

Scheme and material properties

Initial conditions

Linear solvers

Boundary conditions

Detailed documentation in the manual

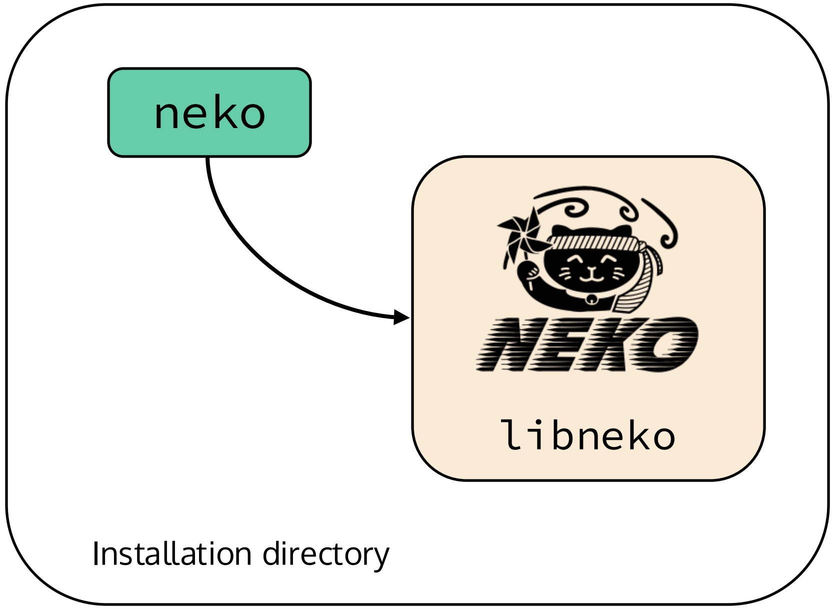
```
"fluid": {
  "scheme": "pnpn",
  "Re": 200,
  "initial_condition": {
      "type": "uniform",
      "value": [ 1.0, 0.0, 0.0 ]
  },
  "velocity_solver": {
      "type": "cg",
      "preconditioner": {
          "type": "jacobi"
      },
      "absolute_tolerance": 1e-7
  },
  "pressure_solver": {
      "type": "gmres",
      "preconditioner": {
          "type": "hsmg"
      },
      "absolute_tolerance": 1e-3
  },
  "boundary_conditions": [
      {
          "type": "velocity_value",
          "value": [ 1, 0, 0 ],
          "zone_indices": [ 1 ]
      },
      {
          "type": ”no_slip",
          "zone_indices": [ 2 ]
      },
      {
          "type": ”outflow",
          "zone_indices": [ 3 ]
      }
  ],
},
```

Running a case

- Running a case in Neko can be done in several ways
- Installation will build and install **`libneko`**
  - The core library containing the Neko framework
- A standalone **`neko`** binary will also be installed
  - This is a full solver, based on `libneko`
    - Internally referred to as "`turboneko`"
  - The binary can be used to solve cases where all necessary functionalities exist in the Neko framework

```
mpirun -np 8 <path-to-neko-installation>/neko cylinder.case
```

For a more detailed example, look at Neko's provided cylinder example in the repository.



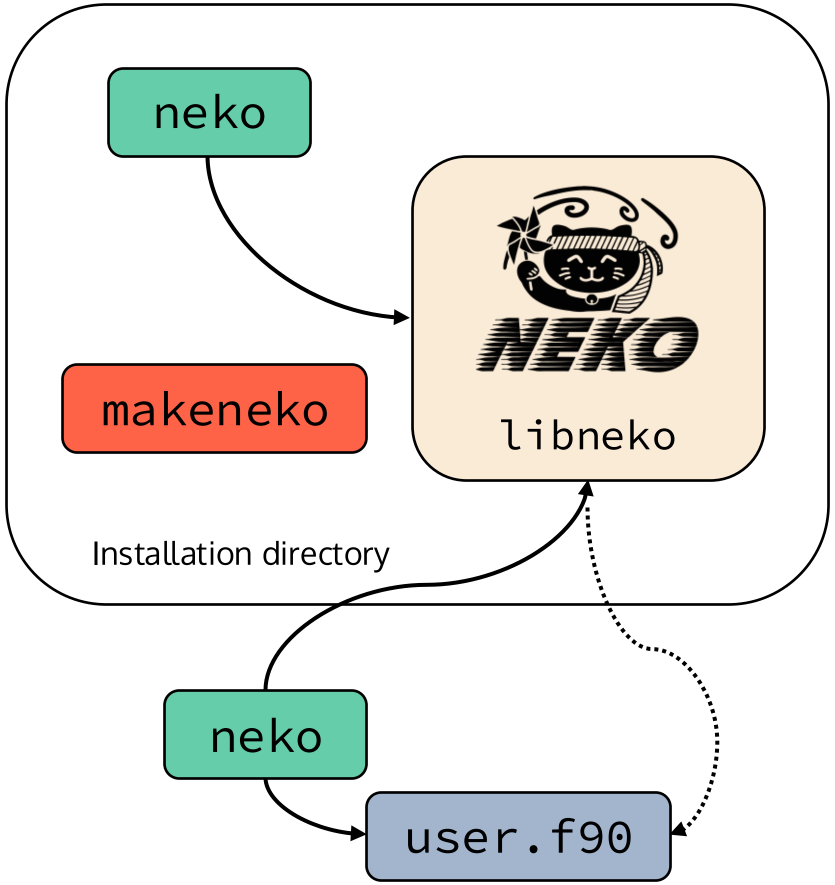
Running a case

- A **user file** is needed if the case needs to extend `libneko`
  - E.g. setting advanced initial/boundary conditions, source terms
- The user file is a normal Fortran file
  - Providing (registering) the user defined functions to Neko
  - The user file must be compiled using `makeneko`

```
> makeneko user.f90
N E K O build tool, Version 1.99.1
(build: 2025-08-01 on x86_64-pc-linux-gnu using gnu)
Building user NEKO ...
Detected the module named 'user' in user.f90
No custom modules detected.
No custom modules register types.
Done!
```

  - This will generate **another** `neko` binary which can run the case (this binary is internally referred to as "`usrneko`")

```
mpirun -np 8 <path-to-case>/neko user_extened.case
```



For a more detailed example, look at Neko's provided Taylor-Green vortex example in the repository.

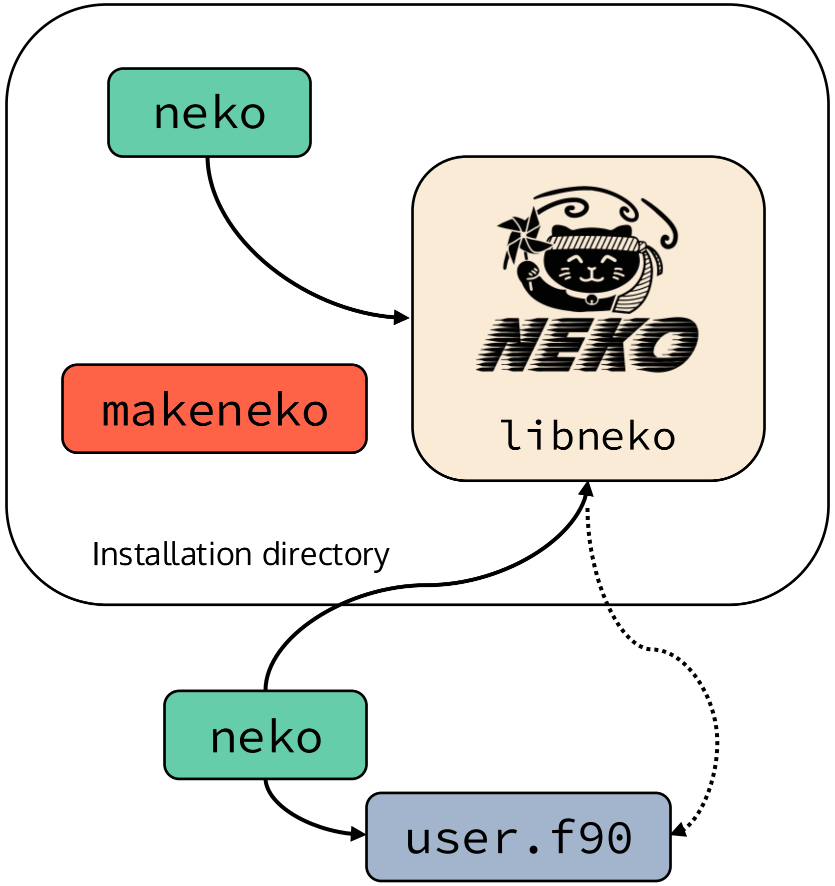
Running a case

Register the callback

Function called by Neko, if enabled in the case file

```
"case": {
   "fluid": {
      "initial_condition": {
         "type": "user"
      }
   }
}
```

```fortran
module user
  use neko
  implicit none
contains
  subroutine user_setup(user)
    type(user_t), intent(inout) :: user
    user%initial_conditions => initial_conditions
. . .
  end subroutine user_setup

  ! User-defined initial condition
  subroutine initial_conditions(scheme_name, fields)
    character(len=*), intent(in) :: scheme_name
    type(field_list_t), intent(inout) :: fields

    u => fields%get_by_name("u")
    v => fields%get_by_name("v")
    w => fields%get_by_name("w")
    p => fields%get_by_name("p")
…

  end subroutine initial_conditions
end module user
```



Detailed documentation in the [manual]

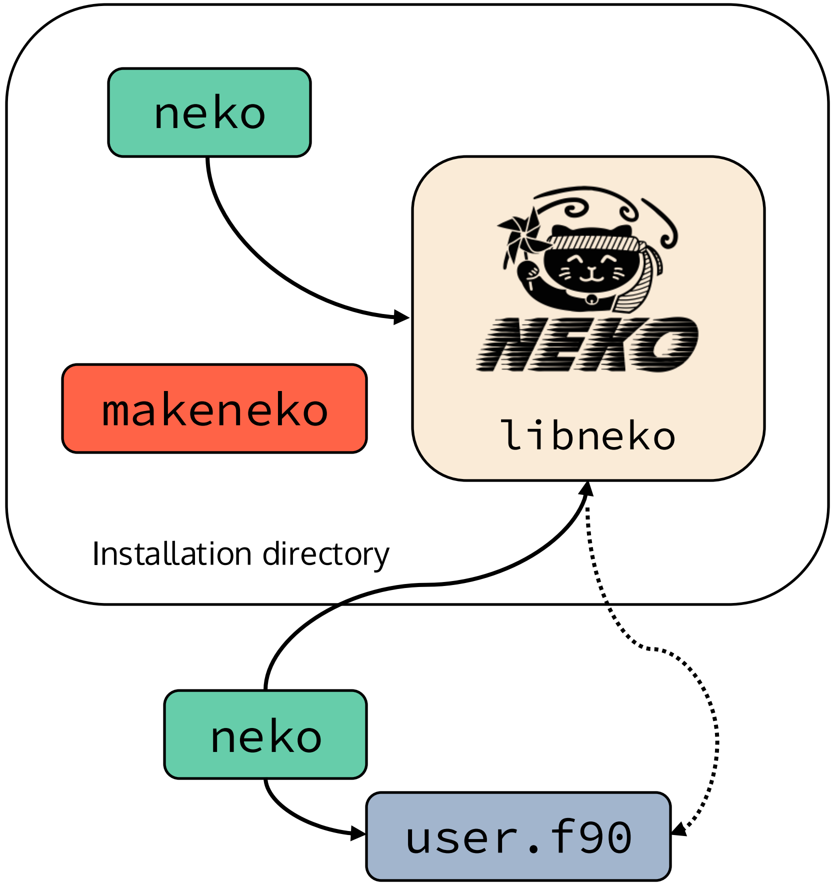
Extending Neko

- In addition to user files, `makeneko` can also be used to compile `.f90` and `.cu/.hip` files
  - Either functions called from the user file
  - Containing components extending Neko's core functionality
    - Selected from the case file (as core components)
    - Currently limited to a subset of types inside Neko

How to extend Neko is explained in the manual

- The last option is to use `libneko` as a SEM framework
  - Build everything from scratch using components inside Neko

This is illustrated in Neko's Poisson example

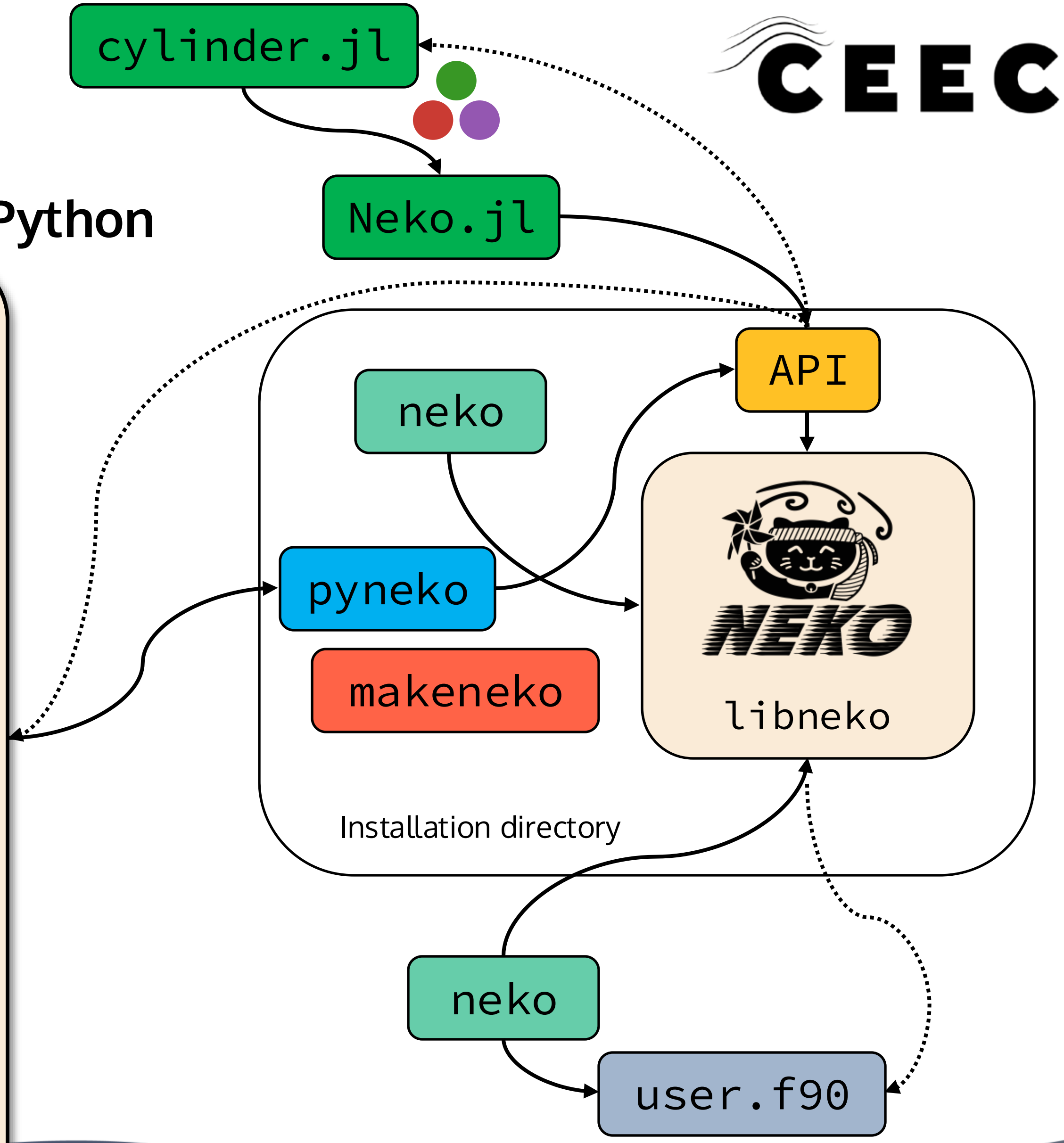


Running a case

Neko v1.0 adds a C-API with bindings for Julia and Python

cylinder.py

```python
import pyneko
import json
import ctypes

# Initialise Neko
pyneko.init()
pyneko.job_info()

cylinder_json = json.load(open('cylinder.case'))

# Define initial conditions
def initial(name, len):
    scheme_name = ctypes.string_at(name, len).decode()

    if (scheme_name == "fluid"):
          u = pyneko.field(b'u')
        for i in range(0, u.dm.ntot):
            u.x[i] = 1.0

# Create a Neko callback for the initial condition
cb_cylinder_ic = pyneko.initial_condition(initial)

# Create a Neko case from a JSON file and provied (optional) callbacks
cylinder_case = pyneko.case_init(cylinder_json,
                                 cb_initial_condition = cb_cylinder_ic)

# To manually step forward in time, call step()
while pyneko.time(cylinder_case) < pyneko.end_time(cylinder_case):
    pyneko.step(cylinder_case)
```



Summary

- Computational Fluid Dynamics is one of the areas with a clear need **and great potential to reach exascale**
- High-order methods are essential on current HPC machines
  - **Better suited for current hardware**, improved accuracy for "free"
- **Neko**, is a portable framework for high-order spectral element-based simulations of turbulent fluid flows
  - Demonstrated **excellent performance and scalability** across various hardware architectures
  - Neko version 1.0 soon to be released (third release candidate 24/10)
- Getting started with Neko
  - Documentation on www.neko.cfd
  - Discussions on GitHub https://github.com/ExtremeFLOW/neko
  - Public Zulip channel https://nekodev.zulipchat.com

CEEC
SeRC Swedish e-Science Research Centre
Swedish Research Council
DEEP-SEA
EXCELLERAT
INNO4SCALE