Neko - Compiling

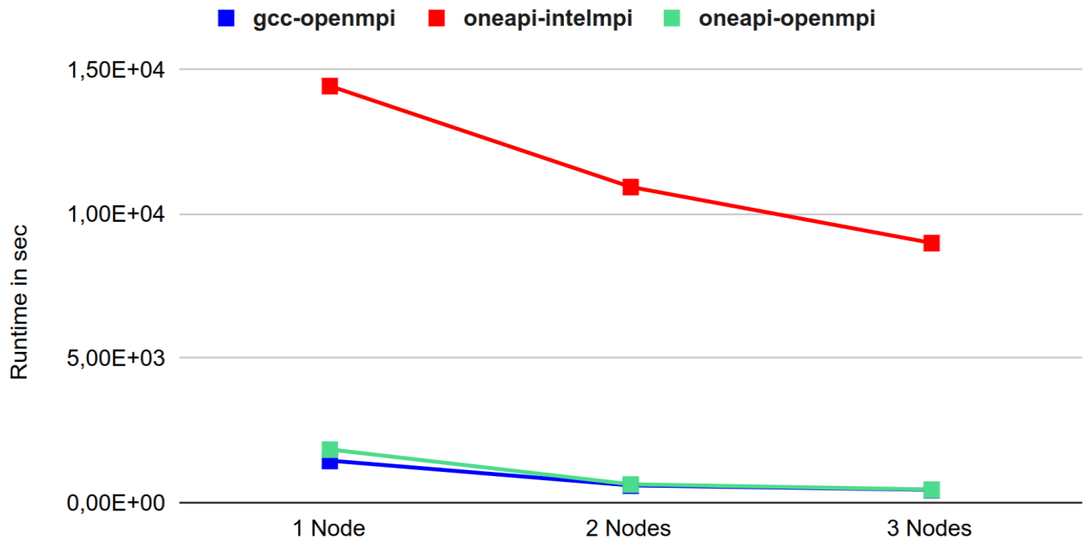
Compiling

- **Neko is hard to build and compile**
- **Many possible compilers**
  - gcc, oneapi, nvhpc, etc.
- **Many possible dependencies**
  - **Math libraries:** openblas, intel-mkl, nvpl-blas, nvpl-lapack
  - **MPI implementations:** openmpi, intel-mpi, mpich
  - **Optional dependencies:** metis, parmeits, etc.
  - **For GPU support:** cuda, nccl
- **We tried different combinations of Compilers x MPI for GPU & CPU version**
  - **Many combinations + different clusters + GPU/CPU -> container**

CPU/GPU - Comparing performance and scalability of different builds

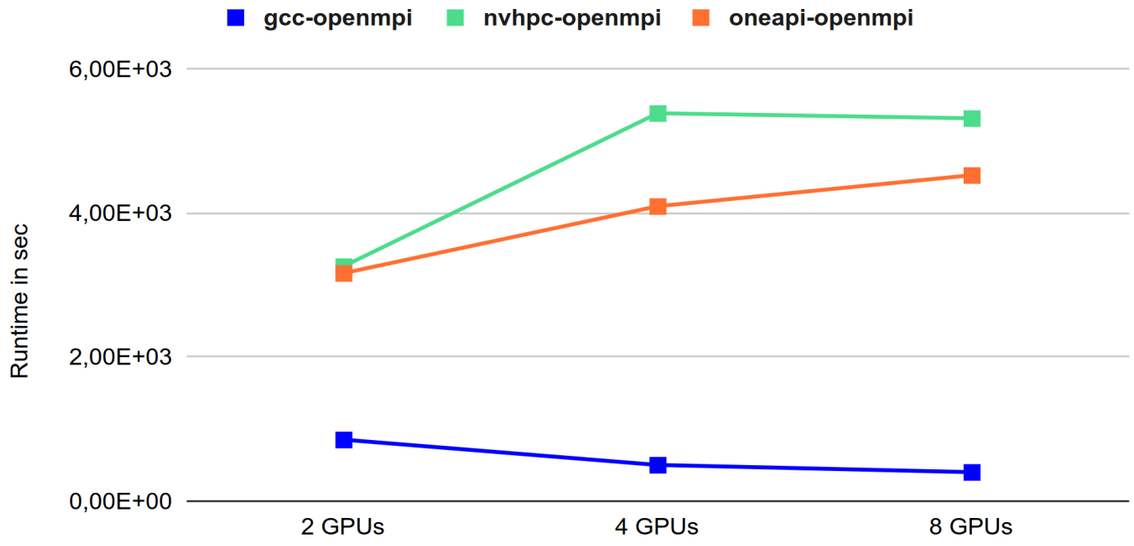
CPU - Comparison of different builds

Hardware: NVIDIA A100 80GB, AMD EPYC 7773X 64-Core Processor



GPU - Comparison of different builds

Hardware: NVIDIA A100 80GB, AMD EPYC 7773X 64-Core Processor



Neko - Container

Building Neko inside a container

- CPU & GPU version based on gcc-openmpi
- 24.3-devel-cuda_multi-rockylinux9 of NVIDIA HPC SDK
- Why?
  - We wanted to test multiple builds
  - Struggle with building GPU version on bare metal: must compile everthing for Nvidia HPC SDK
  - Overall better performance and reliability
- Apptainer was used to enable bare metal HPC performance

Bare metal vs container

- CPU:
  - Bare metal bridges2: 1.93E+03 [s], fully optimized
  - Container bridges2: 4.43E+02 [s], fully optimized -> 4.3x speedup :)
- GPU:
  - Bare metal: n/a
  - Container bridges2: 8.29+02 [s], fully optimized

Neko – CPU Version

Compilers & Dependencies:

```
specs:
  - openmpi%gcc@13.2.0 +atomics +cuda +cxx +legacylaunchers cuda_arch=70 fabrics=ucx schedulers=slurm
    ^pmix%gcc@13.2.0
    ^ucx@1.13%gcc@11.4.1 ~assertions ~debug +cma +cuda +dc +dm +gdrcopy +ib_hw_tm +mlx5_dv +optimizations +rc +rdmacm +ud +verbs ~xpmem cuda_arch=70
    ^gdrcopy%gcc@11.4.1 +cuda cuda_arch=70
    ^cuda@12.3%gcc@11.4.1 +allow-unsupported-compilers
  - openblas%gcc@13.2.0
  - json-fortran%gcc@13.2.0
  - metis%gcc@13.2.0
  - parmetis%gcc@13.2.0
```

- Openmpi performed best in our tests
- Openblas
  - neko relies mostly on small matrix multiplications which fall out of were BLAS/LAPACK implementations perform well
  - Developers have handwritten their math kernels
  - Only place were those libraries are used to invert a small matrix
  - Thus no performance benefit from multithreaded BLAS/LAPACK implementations
- Metis & Parmetis
  - Metis & Parmetis can partition the finite element mesh into smaller subdomains, which can be processed in parallel by different computing units + partitioning of large graphs
  - no observable speedup, but we expect performance gain for larger input

Neko – CPU Version

Configurations & Compiler Flags:

FLAGS="-03 –march=znver2 –mtune=znver2 –funroll-loops"

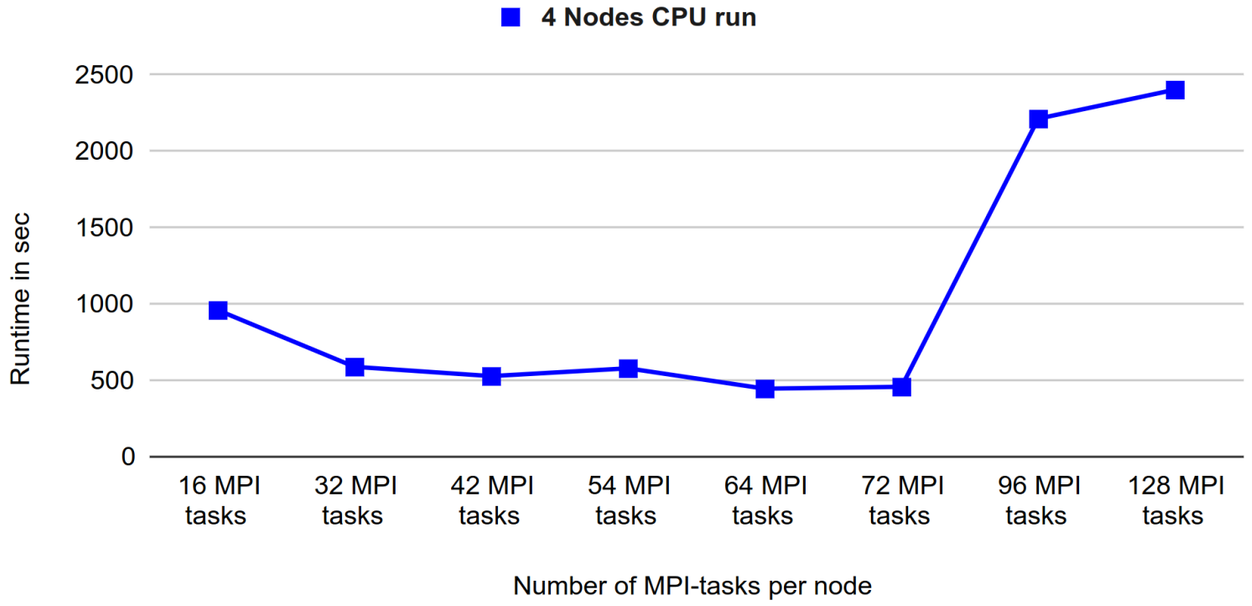
./configure CC=gcc FC=gfortran MPICC=mpicc MPIFC=mpif90 FCFLAGS="${FLAGS}" CFLAGS="${FLAGS}" LDFLAGS="-L${LAPACK_HOME}/lib" --with-parmetis=${PARMETIS_HOME} --with-metis=${METIS_HOME} --prefix=${P_NEKO_INSTALL}

- **GNU compiler collection performed best in our tests**
- **Optimized code for Bridges2 hardware**
  - **AMD znver2**
- **Optimized for loop unrolling**
  - **neko relies on many handwritten small matrix multiplications**
  - **Unroll loops for more performance**

Neko – CPU Version

Run Command:

mpirun –n 256 –H rxzy:64,rxzy:64,rxzy:64,rxzy:64 --cpus-per-proc 2 apptainer exec –nv –no-mount home –bind $P_INPUT:/scratch --pwd=/scratch $P_CONTAINER /sratch/neko /scratch/tgv_Re1600.case



- **We noticed that reducing #MPI-tasks yields better performance**
- **Minimum at 64 MPI-tasks per node were each MPI-task is explicitly bound to 2 cores**
- **Reasoning:**
  - **Small problem size, communication is bottleneck**
  - **128 MPI-tasks per node is an overkill which leads to large communication overhead**
  - **By specifying 64 MPI-tasks per node we get good balance between communication overhead and workload per core**
  - **Bind one MPI-task to two cores to prevent overheating**

Neko – GPU Version

Compilers & Dependencies:

```
specs:
  - openmpi%gcc@13.2.0 +atomics +cuda +cxx +legacylaunchers cuda_arch=70 fabrics=ucx schedulers=slurm
    ^pmix%gcc@13.2.0
    ^ucx@1.13%gcc@11.4.1 ~assertions ~debug +cma +cuda +dc +dm +gdrcopy +ib_hw_tm +mlx5_dv +optimizations +rc +rdmacm +ud +verbs ~xpmem cuda_arch=70
    ^gdrcopy%gcc@11.4.1 +cuda cuda_arch=70
    ^cuda@12.3%gcc@11.4.1 +allow-unsupported-compilers
  - openblas%gcc@13.2.0
  - json-fortran%gcc@13.2.0
  - metis%gcc@13.2.0
  - parmetis%gcc@13.2.0
```

- Same dependencies and reasoning as for CPU version
- Cuda support for Nvidia GPUs

Neko – GPU Version

Configurations & Compiler Flags:

```
./configure FC=gfortran FCFLAGS="-O3 –march=znver2 –mtune=znver2 –funroll-loops" \
        CC="gcc –O3 –march=znver2 –mtune=znver2 –funroll-loops" CFLAGS="-O3" \
        LDFLAGS="-L${LAPACK_HOME}/lib -L${NCCL_HOME}/lib" \
        --with-parmetis=${PARMETIS_HOME} \
        --with-metis=${METIS_HOME} \
        --with-cuda=${CUDA_HOME} CUDA_ARCH=-arch=sm_70 \
        NVCC=${NVCC_HOME} CUDA_CFLAGS="-O3 –allow-unsupported-compiler" \
        --with-nccl=${NCCL_HOME} \
        --enable-device-mpi \
        --prefix=${P_NEKO_INSTALL}
```

- **GNU compiler collection performed best in our tests, optimized for bridges2 hardware**
- **Metis, parmetis might be benefitial for larger input cases**
- **NCCL:**
  - **Nvidia collective communication library; optimize communication between GPUs**
  - **No observed speedup**
- **Enable device MPI: to avoid unnecessary device-host copies**

Neko – GPU Version

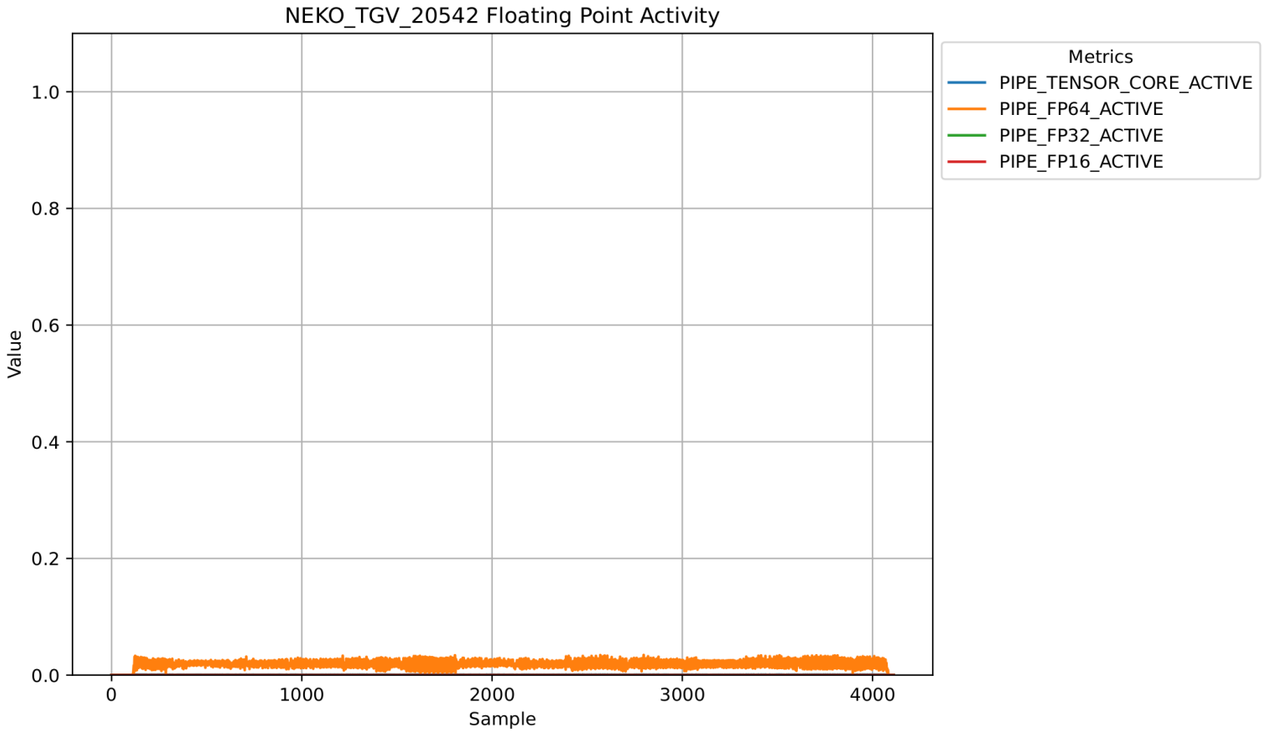
Run command:

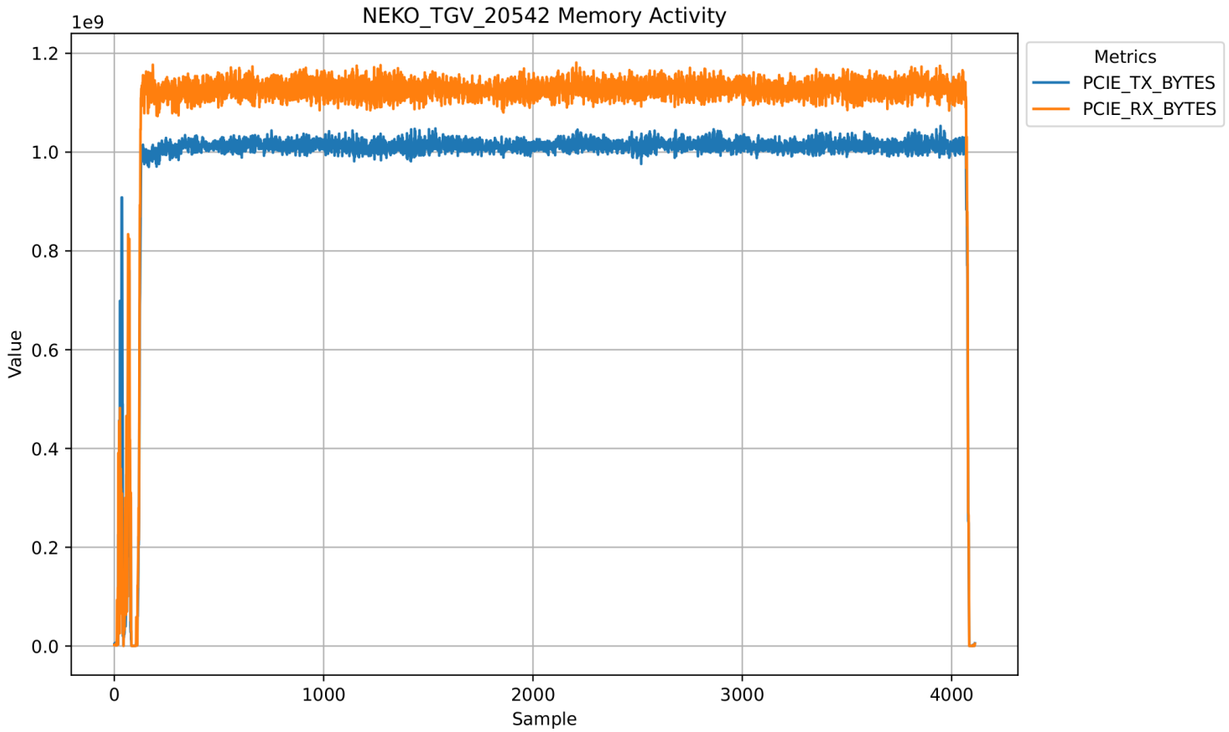
```
mpirun –np 4 \
    –bind—to none \
    –map-by node \
    –rank-by slot:span \
    –mca btl^tcp,vader,openib –mca pml ucx –mca rankfile slot=0:0,1:1,2:2,3:3 \
    apptainer exec –nv –no-mount home –bind $P_INPUT:/scratch --pwd=/scratch $P_CONTAINER \          bash –c ' export CUDA_VISIBLE_DEVICES=$((OMPI_COMM_WORLD_RANK % 4)); \
    /scratch/neko /scratch/tgv_Re1600.case' | tee $OUTPUT
```

- **Explicitly bind each MPI-task to one GPU device**
- **Disabling tcp, vader, openib as components of BTL (Byte Transfer Layer)**
  - --mca btl ^tcp,vader,openib
- **Selects ucx as PML (Point-to-Point Messaging Layer)**
  - --mca pml ucx

Neko – Problems

GPU Profiling with Input tgv_Re1600.case

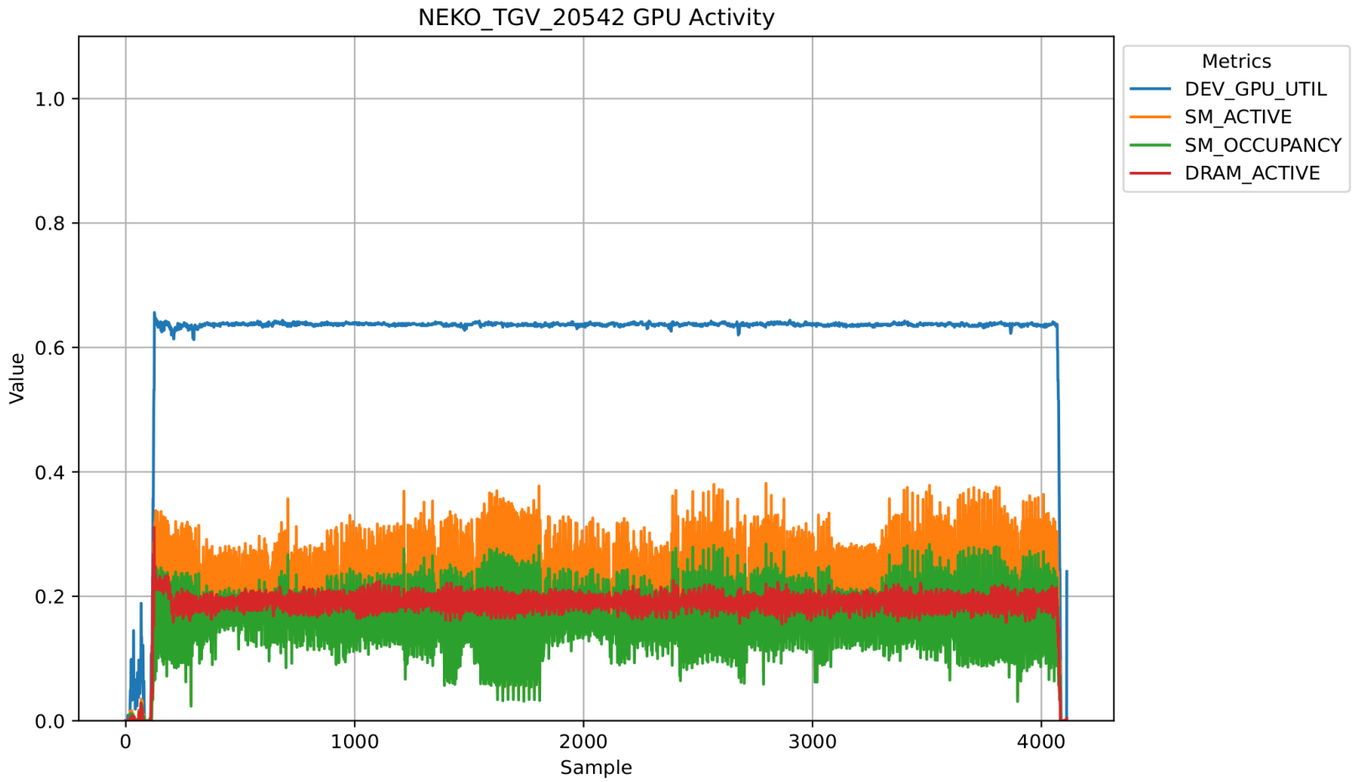




Profiling Tool: https://github.com/eth-cscs/MLp-system-performances-analysis-tool (Thanks to Marcel Ferrari, CSCS)

Neko – Problems

GPU Profiling with Input tgv_Re1600.case



- GPU sitting idle ~80% of the time
- SM activity: GPU is only doing computation ~50% of the time
- FP64 engine activity: math engine is only running at < 5% of its theoretical peak
- Disproportional amount of memory operations compared to floating point activity
- Host->device and device->host connections are not fully utilized
- Neko might scale well compared to CPU implementations, but it is not using the available GPU resources efficiently

Neko – Problems

MPI Profiling:

- Spent 1 week to MPI profile Neko
- Tried hpcx and mpiP
- Both could not even generate profiler output

Conclusion:

- MPI profilers work all the same in the core fundamentals
- They "Overload" the MPI profiling interface:
  - done by linking custom functions that keep track of metrics related to MPI calls
  - MPI profiling interface provides set of "twin" functions to the normal MPI functions, e.g. MPI_Alltoall -> PMPI_Alltoall
  - Functions are overloaded at linktime
  - Collect metrics every time MPI_Alltoall is called
- Problem:
  - Either "twin" functions are not linked correctly
  - Or "twin" functions are overwritten by the standard ones
  - Both would result in no profiler output :(