

GPU Acceleration at Scale with OpenPower platforms in *Code_Saturne*



Samuel Antao¹, Charles Moulinec², Yvan Fournier³, Robert Sawko¹,
Malgorzata Zimon¹, Christopher Thompson¹, Alex Skillen², Juan Uribe⁴, David R. Emerson²

¹IBM Research, ²SCD, STFC Daresbury Laboratory UK, ³EDF R&D FR, ⁴EDF R&D Centre UK



Main accomplishments

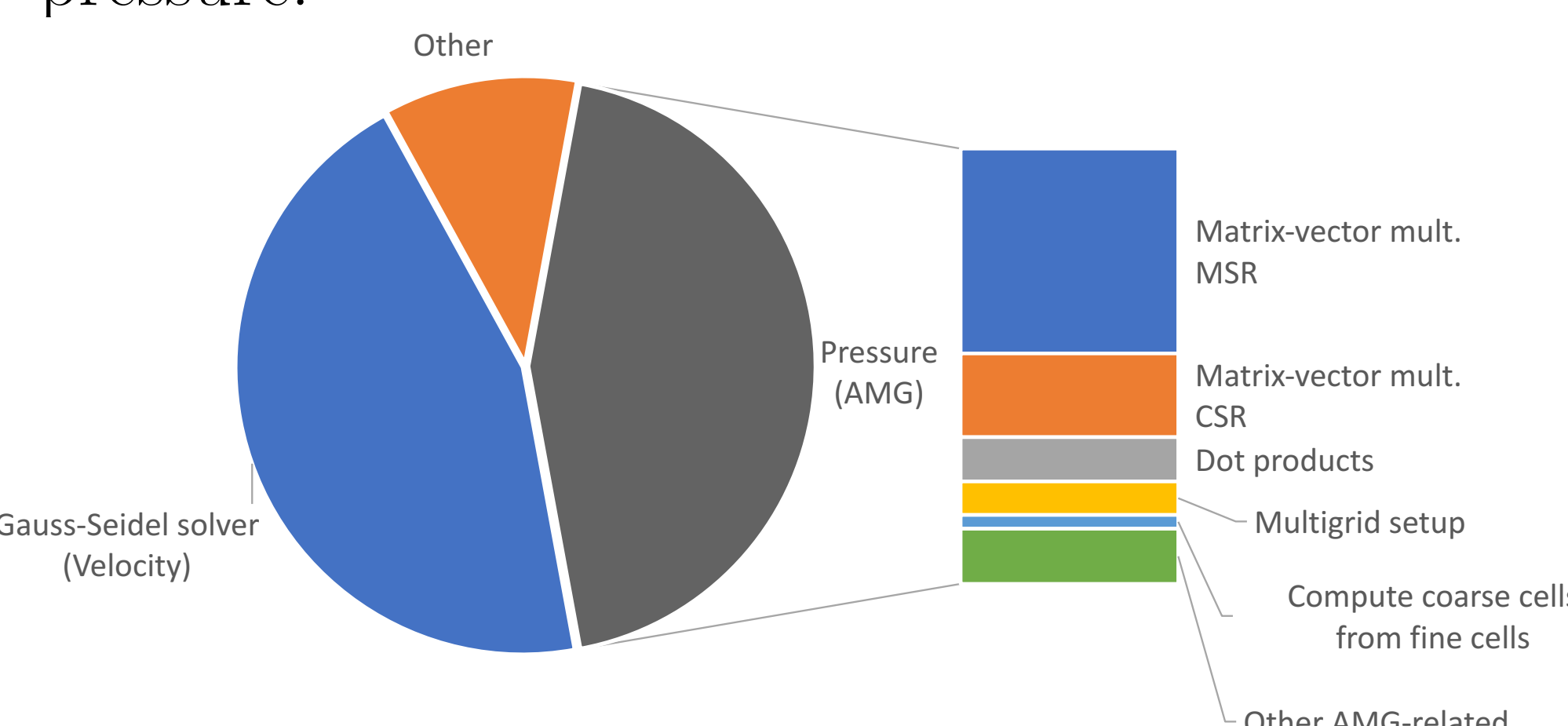
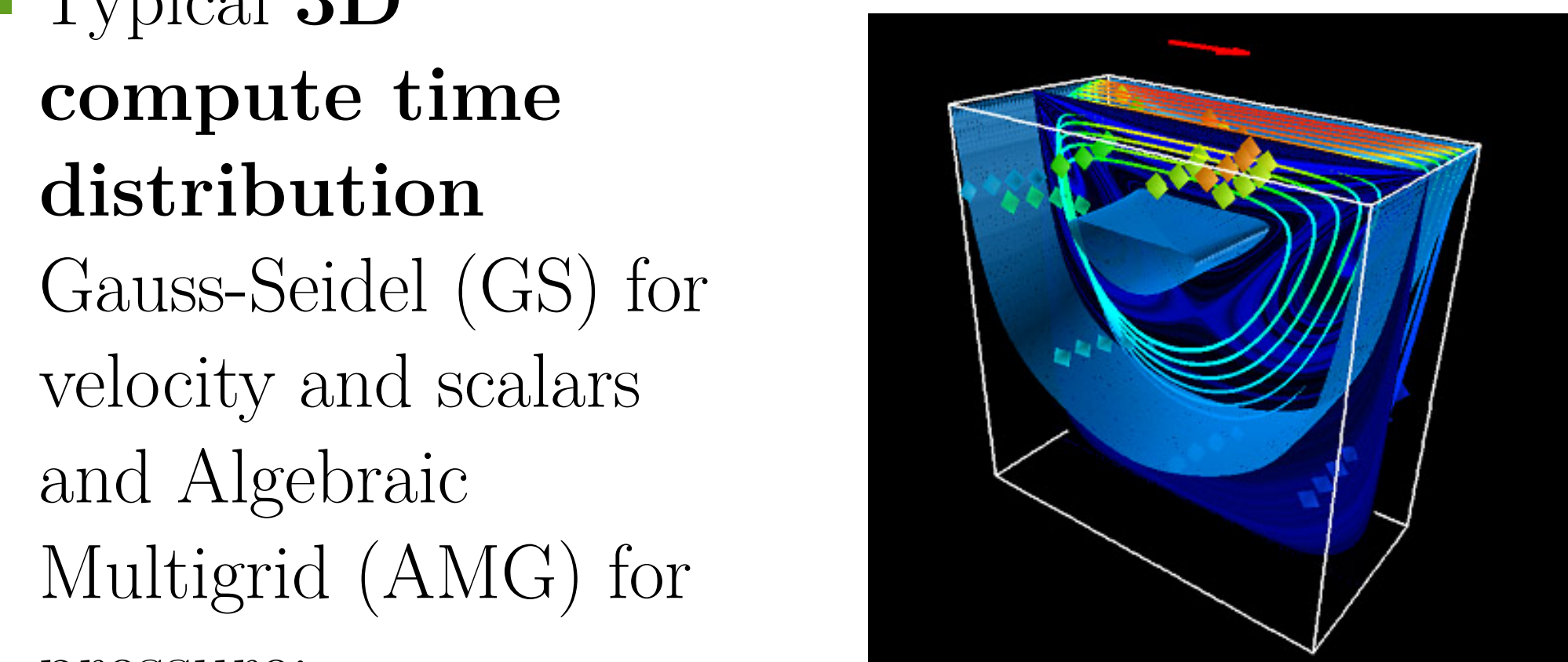
- Library to provide **GPU acceleration at scale** for *Code_Saturne* running on top of OpenPower machines using CUDA.
- Manage **data environments**: control scope of data in the device.
- Single **templated device entry function** to run application kernels back to back and **mitigate kernel launching latencies**.
- Leverage **NVLINK™ fast interconnect** to shift bottle necks from data-movement towards compute.
- 2.9x speedup over CPU-only simulation** on 256-node POWER9™ machine - **ORNL Summit Supercomputer**.
- Enable **multi-billion-cell unstructured mesh** simulations.

What is *Code_Saturne*?



- www.code-saturne.org
- Open-source
- Computational fluid dynamics (CFD) software package
- Developed by EDF-R&D

- Demonstrated **good scaling characteristics**: tailored to tackle extra large problems, e.g. **whole thermo-nuclear reactors**.
- OpenMP + MPI - C/C++, Fortran and Python
- Typical **3D compute time distribution**



What is the challenges?



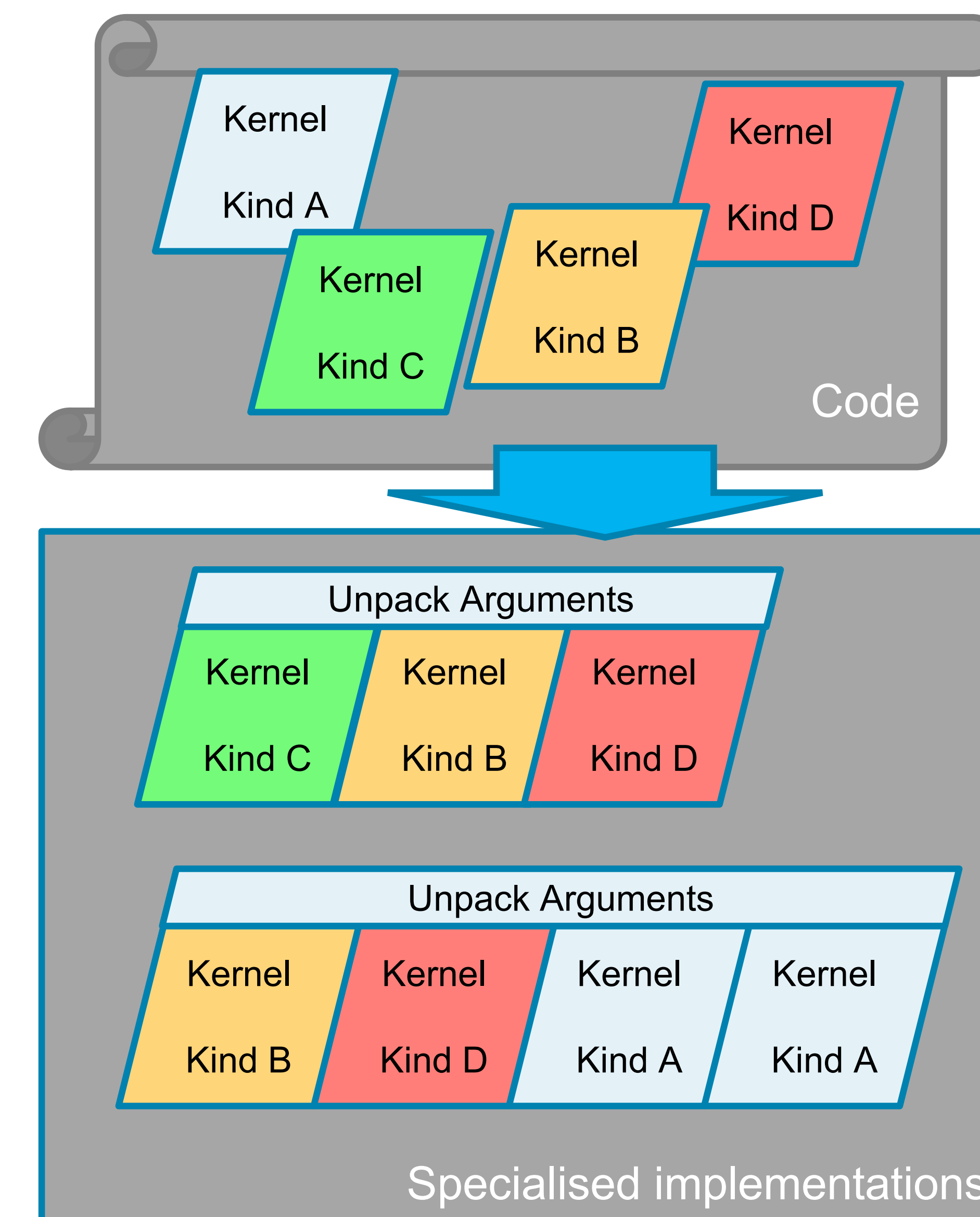
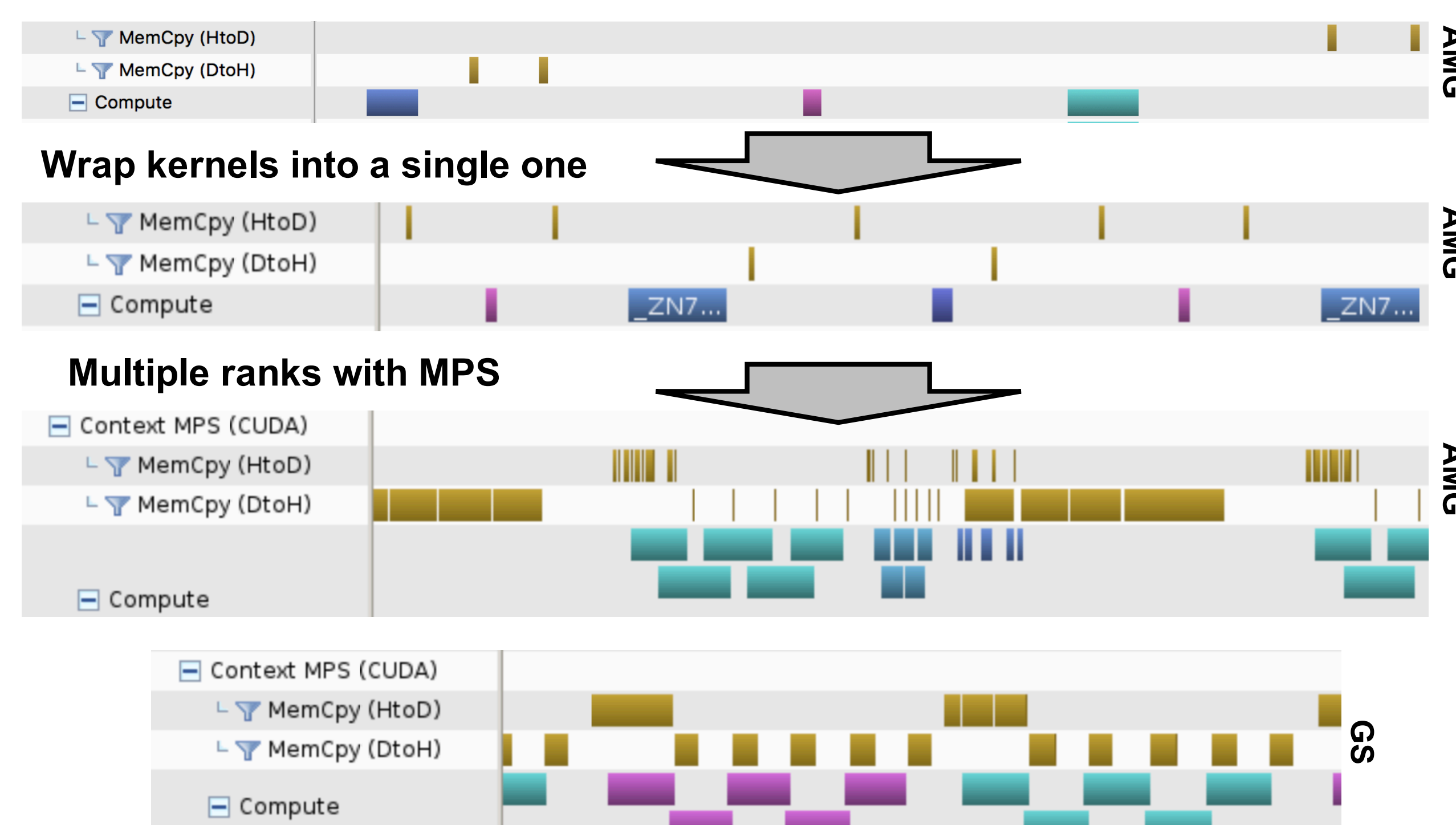
How do we tackle them?

- Mitigate latencies for launching kernels back-to-back:
 - Packing arguments of various kernels (SpMV, dot products, etc) into a single data structure.
 - Leverage template pack expansion to wrap multiple kernels in a single CUDA kernel.

```
template<KernelKinds Kind>
__device__ int any_kernel(KernelArgsBase &Arg, unsigned n_rows_per_block) {
    switch(Kind) {
        // Matrix-vector multiplication (MSR format):
        case MV_MSR:
            matrix_vector_multiplication<Kind>(<
                /* dev_row_index */ Arg.getArg<cs_lnum_t >*(0),
                /* dev_col_id */ Arg.getArg<cs_lnum_t >*(1),
                /* dev_val */ Arg.getArg<cs_real_t >*(2),
                /* dev_d_val */ Arg.getArg<cs_real_t >*(3),
                /* dev_x */ Arg.getArg<cs_real_t >*(4),
                /* dev_y */ Arg.getArg<cs_real_t >*(5),
                /* n_rows */ Arg.getArg<cs_lnum_t >*(6),
                /* n_cols */ Arg.getArg<cs_lnum_t >*(7),
                /* n_rows_per_block */ n_rows_per_block);
            break;
        // ...
        case DP_xx:
            dot_product<Kind>(<
                /* version */ Arg.getArg<cs_lnum_t >*(0),
                /* n_rows */ Arg.getArg<cs_lnum_t >*(1),
                /* x */ Arg.getArg<cs_real_t >*(2),
                /* y */ nullptr,
                /* z */ nullptr,
                /* res */ Arg.getArg<cs_real_t >*(3),
                /* n_rows_per_block */ n_rows_per_block);
            break;
        // ...
    }
    __syncthreads();
    return 0;
}

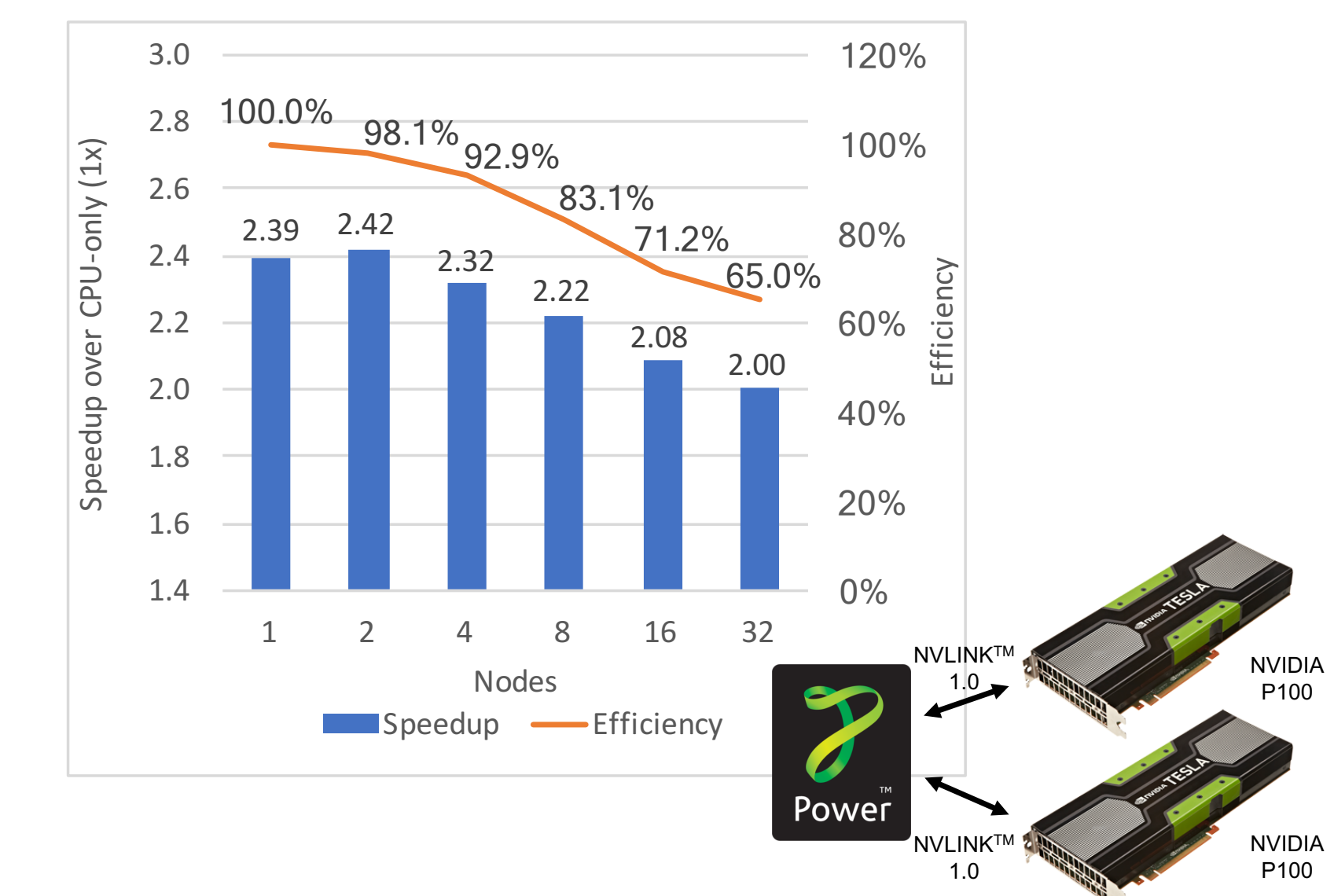
template<KernelKinds... Kinds>
__global__ void any_kernels(void) {
    auto *KA = reinterpret_cast<KernelArgsSeries *>(&KernelArgsSeriesGPU[0]);
    const unsigned n_rows_per_block = KA->RowsPerBlock;
    unsigned idx = 0;

    int dummy [] = { any_kernel<Kinds>(KA->Args[idx++], n_rows_per_block)... };
    (void) dummy;
}
```

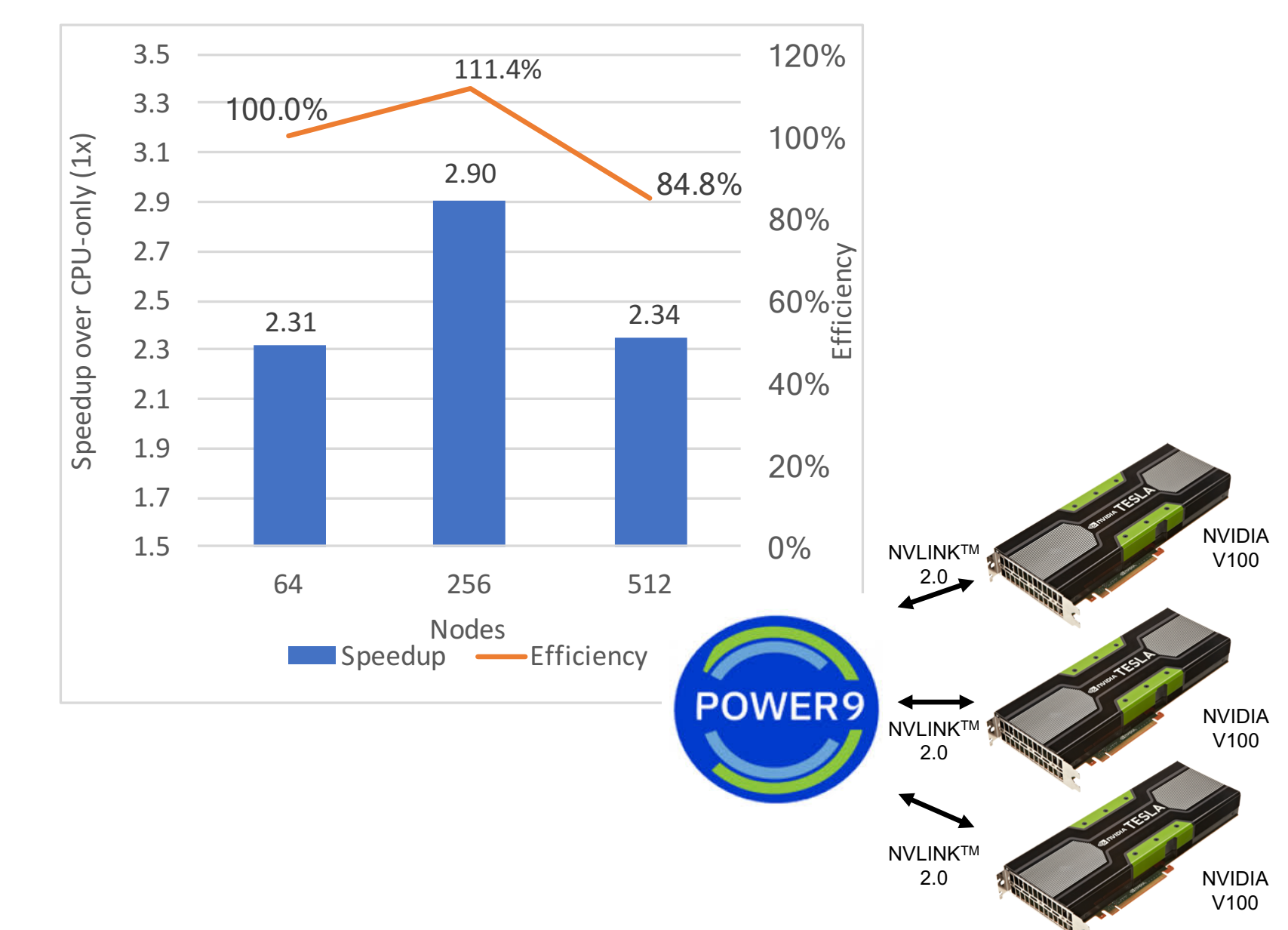


Results

CPU+GPU speedup over CPU-only and efficiency (strong scale)



(a) POWER8 + P100 GPUs (111M-cells).



(b) POWER9 + V100 GPUs (889M-cells)¹.

- Compute and data movements overlap** almost perfectly for the GS algorithm.
- Consistent **speedup over 2x** with only ~100K-cells per rank.
- NVLINK 2.0** available in **POWER9** machines enables reaching better strong scaling than its previous generation:
 - Better efficiency** when using 16x more resources for a problem that is only 8x larger.

Acknowledgements

POWER9 scaling experiments were only possible with Summit Early Access programme sponsored by Oak Ridge Leadership Computing Facility. Work partially funded by UKTC consortium grants EP/L000261/1 and EP/R029326/1.

More information:
samuel.antao@ibm.com



APPENDIX A

ARTIFACT DESCRIPTION APPENDIX: GPU ACCELERATION AT SCALE WITH OPENPOWER PLATFORMS IN *Code_Saturne*

A. Abstract

This description contains the information on how the results presented in SC18 poster “GPU Acceleration at Scale with OpenPower platforms in Code_Saturne” were obtained and how they can be reproduced.

We present details on the dependences, Code_Saturne build-process, case configuration, and job submission.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Velocity-pressure coupling with Gauss-Seidel (velocity) and Algebraic Multigrid with Conjugate gradient at each level (pressure).
- **Program:** C/C++, Fortran and Python source. For GPU runs, CUDA source code is used. Solver implemented in C99 (host) and CUDA (GPUs).
- **Compilation:** gcc 6.4.0 and CUDA 9.1 (POWER8) or CUDA 9.2 (POWER9).
- **Binary:** Native PPC64LE ELF64 binary. For GPU runs, CUDA source code is used and NVIDIA GPU binary is embedded in the host binary.
- **Data set:** 111M-cell mesh and 889M-cell meshes. A 13M-cell mesh was generated using icem-cfd, and mesh multiplication was applied to get these large meshes.
- **Run-time environment:** IBM Spectrum LSF (POWER8) and IBM CSM (POWER9)
- **Hardware:** POWER8 with P100 NVIDIA GPUs and POWER9 with V100 NVIDIA GPUs.
- **Execution:** OpenMPI mpirun launching (POWER8) and jsrun launching (on top of IBM Spectrum MPI) (POWER9).
- **Output:** Simulation Wall time of the 3rd time step.
- **Experiment workflow:** Build binary that implements selected testcase and execute it with MPI launching wrapper.
- **Publicly available?:** Binary yes, source no (arrangements are being made to contribute it to *Code_Saturne* main repository).

2) How software can be obtained (if available):

Code_Saturne is an open-source software. The CPU-only code can be obtained from github.com/code-saturne/code_saturne. We used a snapshot from September 2017 - e5a72a6fa. The GPU port was rebased on the same snapshot and its code is not upstreamed (and therefore not publicly available) as of yet. Nevertheless, the binary of the GPU-port is available to users of the POWER8 Paragon system at Hartree Centre, STFC, UK. It is made available as a *beta* LMOD module that can be conveniently used by any users with access to the system.

3) *Hardware dependencies:* The current GPU-port implementation requires a POWER8 or POWER9 machine with NVIDIA GPUs with CUDA compute capability 3.5 or higher.

The vanilla *Code_Saturne* implementation is known to work on other hardware including IBM BlueGene Q machines.

4) *Software dependencies:* *Code_Saturne* is known to be successfully built on different dependencies and different versions of these dependencies [1]. For the results presented in the poster these are the software dependences:

- POWER8 and NVIDIA GPU P100:
 - GNU compilers 6.4.0
 - OpenMPI 3.0.0

```
./configure \
--disable-shared \
--enable-static \
--enable-openmp \
--enable-cuda-offload \
--enable-long-gnum \
--host=ppc64le \
--build=ppc64le \
--without-modules \
--disable-gui \
--without-libxml2 \
--without-hdf5 \
--without-salome-kernel \
--without-salome-gui \
--prefix=<user selected prefix> \
CC=mpicc CFLAGS="-g_-O3" \
CXX=mpic++ CXXFLAGS="-g_-O3" \
FC=mpifort FCFLAGS="-g_-O3" && make install
```

Listing 1. Command to configure and build Code Saturne with CUDA offload support.

– CUDA 9.1

- POWER9 and NVIDIA GPU V100:
 - GNU compilers 6.4.0
 - IBM spectrumMPI 10.1
 - jsrun 1.1.0
 - CUDA 9.2

5) *Datasets:* The input data set consists mainly of the mesh of a cubic cavity that uses 111M or 889M tetrahedral cells for the POWER8 and POWER9 runs, respectively. The mesh was obtained using icem-cfd for a 13M cell mesh and mesh multiplication was then applied to get these large meshes.

C. Installation

As part of the GPU port work, we also updated the build system, so building with and without GPU offload support is straightforward - see Listing 1. To disable GPU offload and get a CPU-only implementation, one has to remove the line `--enable-cuda-offload`.

D. Experiment workflow

The first step of the experiment is to generate the case. In our setup, we provide a template case for the users - see Section A-E to see how to customise it. The template contains a SRC folder with several files as in Listing 2. The file `mesh_input` is the input mesh and the remaining files specify customisations to the tool. These files are compiled together with *Code_Saturne* to change its default behaviour. Running `code_saturne compile` in that folder compiles these files along with the *Code_Saturne* implementation libraries and creates the binary `cs_solver`. This binary is the executable that performs the simulation.

Listings 3 and 4 show the content of submission scripts of the GPU runs for POWER8 and POWER9, respectively. In both cases, the job is submitted by piping it into `bsub` command. In these listings there are two wrapper scripts: `cs_mps`, and `cs_ompi`. The former starts 4 MPS servers per node (one for each GPU) and set the GPU visibility so that a set of 5 consecutive ranks use the same MPS server instance. The latter just sets OpenMPI compatible environment from the

```
cs_user_boundary_conditions.f90
cs_user_parameters.c
cs_user_parameters.f90
cs_user_performance_tuning.c
cs_user_postprocess.c
mesh_input
```

Listing 2. Contents of *Code_Saturne* case source folder.

```
#!/bin/bash
#
#BSUB -J <job-name>
#BSUB -W 01:30          # wall-clock time
#BSUB -q paragon        # queue
#BSUB -eo errors.log    # error file name
#BSUB -oo output.log    # output file name
#BSUB -x                # exclusive mode
#BSUB -n 512            # number of tasks in job
#BSUB -R "span[ptile=16]" # ranks per node
#BSUB -gpu "num=4:mode=shared" # activate the 4 GPUs
#-----

ulimit -s 10240
export OMP_NUM_THREADS=8

mpirun --map-by socket -bind-to core \
./cs_mps ./cs_solver
```

Listing 3. Submission script for POWER8 system running IBM Spectrum LSF - 32 node run (16 ranks per node) - 4 GPUs per node.

```
#!/bin/bash
#
#BSUB -P <project-name>
#BSUB -J <job-name>
#BSUB -W 00:30          # wall-clock time
#BSUB -q batch          # queue
#BSUB -eo errors.log    # error file name
#BSUB -oo output.log    # output file name
#BSUB -nnodes 512       # number of nodes
#BSUB -alloc_flags "gpumps smt4" # MPS and SMT=4
#-----

ulimit -s 10240
export OMP_NUM_THREADS=4

jsrun --rs_per_host 6 --gpu_per_rs 1 \
--tasks_per_rs 3 --cpu_per_rs 6 --nrs 3072 \
-D CUDA_VISIBLE_DEVICES --smpiargs="-gpu" \
-d plane:3 -b packed:2 ./cs_ompi ./cs_solver
```

Listing 4. Submission script for POWER9 system running IBM CSM - 512 node run (18 ranks per node) - 6 GPUs per node.

```
export OMPI_COMM_WORLD_RANK=$JSM_NAMESPACE_RANK
export OMPI_COMM_WORLD_SIZE=$JSM_NAMESPACE_SIZE
export OMPI_COMM_WORLD_LOCAL_RANK=\
$JSM_NAMESPACE_LOCAL_RANK
export OMPI_COMM_WORLD_LOCAL_SIZE=\
$JSM_NAMESPACE_LOCAL_SIZE
$@
```

Listing 5. Contents of the *cs_ompi* wrapper

existing CSM environment prior to invoking the *cs_solver*-see Listing 5 - CSM handles the MPS server itself.

E. Evaluation and expected result

Once the simulation completes, the time-step execution Wall time is extracted from the generated *timer_stats.csv* file. We do not use the first and last time-step times because that also includes initialisation and clean up actions that are not relevant for the performance assessment.

For the GPU port, results are verified by comparing the content of the generated *listing* file with the CPU-only run. The different quantities (e.g. velocity, pressure) and number of interactions must be similar - they will not be exactly the same as the rounding error as well as the loop scheduling by threads is different.

F. Experiment customisation

Customisation happens at two levels, application and execution environment.

1) *Application Customisation*: Application customisation is performed by editing the files in Listing 2. We used the exact same files for both CPU and GPU runs using default values.

2) *Execution Environment Customisation*: Customisation here has to do with the number and distribution of ranks and threading in each rank to get better performance. The application is OpenMP enabled, so we use all the threads in a core for all configurations, i.e., 8 threads for POWER8 and 4 threads for POWER9.

For POWER8, we use 1 rank per core, which means we use 20 ranks per node. For GPU runs that stays the same, and each group of 5 ranks use one GPU.

For POWER9 we use 21 ranks per node, which is half the number of cores in the node. So, we have a pair of cores for each rank. As L3-cache is shared by each pair of cores, this resulted in better performance. For GPU runs we use only 18 ranks per node, so as it can be divided by 6, the number of GPUs in the system. Each 3 ranks share the same GPU.

REFERENCES

- [1] Code Saturne project website, <https://www.code-saturne.org>.