GPU-accelerated OpenFOAM simulations using PETSc4FOAM

S. Zampini[1], S. Bnà[2], M. Valentini[3], I. Spisso[4]

[1] *Research Scientist, Extreme Computing Research Center, KAUST (Saudi Arabia)*
stefano.zampini@kaust.edu.sa
[2,3,4] *SuperComputing Applications and Innovation (SCAI) Department, CINECA, Italy;*
simone.bna@cineca.it, *m.valentini@cineca.it*, *i.spisso@cineca.it*

Table of Contents

Abstract

The aim of the present work is a preliminary comparison between the OpenFOAM embedded linear algebra solvers and a selection of external linear solvers provided by the PETSc library [1] and other packages of linear algebra solvers such as hypre [2] and ML [3], with and without GPU support. The current work is used as a test-bed for the Petsc4FOAM library [4, 5] by comparing i) standard solvers on CPUs and ii) algebraic multi-grid based solvers running on hybrid CPU-GPU. Software comparisons will be done on two different architectures (Intel Skylake and IBM Power 9 + Nvidia V100) in order to cross-check the results and the software stack.

Preliminary results show that in the serial case (1CPU vs 1CPU+1GPU) the gain in performance is about 5x with minor differences between the GPU library used (cuda or viennaCL) while in the full node case (32 CPU cores vs 32 CPU cores + 4 GPU), the speedup due to the GPU is about 2x. We also tested the caching, a feature that allows us to compute the preconditioner one single time at the beginning of the simulation [5]. In this case, not representative of a general simulation, the speedup is about 3x.

During the test campaign, it is noted an important correlation between the flavour (both software version and toolchain/configuration flags) of the BLAS library and the overall performance delivered. After testing some different configurations we kept the results related to the best performing BLAS flavour (netlib version).

Methodology

The software used for the test campaign is OpenFOAM vanilla versus a petsc-enabled OpenFOAM version. In particular:

- OpenFOAM (v1912) standard solver:
  - **ldu_foam_dic_pcg**: is an iterative solver provided by the OpenFOAM package that combines a diagonal-based incomplete-Cholesky preconditioner (DIC) with a Conjugate Gradient solver (PCG). DIC is a common preconditioner in OpenFOAM for its easy configuration and high efficiency;
- OpenFOAM (v1912) + petsc4foam + external linear algebra solver.

A comprehensive list of external linear algebra solvers embedded in Petsc or external packages that have been tested is reported below:

1. **aij_icc_pcg:** is the PETSc counterpart of **ldu_foam_dic_pcg,** provides a CG implementation that performs both inner products needed in the algorithm with a single MPI_Allreduce call. ICC is the Incomplete Cholesky factorization implementation of PETSc.
2. **aij_hypre_cg:** PETSc provides a CG implementation, and uses HYPRE, a library of high performance preconditioners and solvers featuring multigrid methods for the solution of large, sparse linear systems of equations on massively parallel computers developed at Lawrence Livermore National Lab (https://github.com/hypre-space/hypre). In this case the PETSc matrix format is used, hence a conversion is needed to setup the Hypre solver.
3. **aijhypre_hypre_cg**: PETSc provides a CG implementation, and uses Livermore's HYPRE. In this case the matrix is assembled natively in the Hypre matrix format.
4. **aij_gamg_cheby_cg**: PETSc provides a CG implementation with a geometric-algebraic multigrid preconditioner (GAMG). Cheby is the preconditioned Chebyshev iterative method used as smoother in the preconditioner. This method runs only on the CPUs.
5. **aijcl_gamg_cheby_cg:** PETSc provides a CG implementation with a geometric-algebraic multigrid preconditioner (GAMG). Cheby is the preconditioned Chebyshev iterative method used as smoother in the preconditioner which is the default for PETSc. This method is setup with the preconditioner on the CPU and it runs on the GPU using the viennaCL (openCL) wrapper (https://github.com/viennacl/viennacl-dev).
6. **aijcuda_gamg_cheby_cg:** PETSc provides a CG implementation with a geometric-algebraic multigrid preconditioner (GAMG). Cheby is the preconditioned Chebyshev iterative method used as smoother in the preconditioner which is the default for PETSc. This method is setup with the preconditioner on the CPU and it runs on the GPU using the cuSPARSE API from NVIDIA.
7. **aijcuda_ml_cg**: PETSc provides a CG implementation, and the ML (Multilevel Preconditioning package) from Trilinos (https://docs.trilinos.org/dev/packages/ml/doc/html/index.html) has been used as preconditioner for the linear system. This method is setup on the CPU and it runs on the GPU using directly the cuSPARSE API from NVIDIA.
8. **aijcuda_gamg_sor_cg:** PETSc provides a CG implementation with a geometric-algebraic multigrid preconditioner (GAMG). Sor is the Gauss-Seidel smoother used in the preconditioner (Successive Over Relaxation). This method runs on CPU and GPU using directly the cuSPARSE drivers from NVIDIA; sor preconditioning is applied on the CPU.
9. **aij_gamg_sor_cg**: PETSc provides a CG implementation with a geometric-algebraic multigrid preconditioner (GAMG). Sor is the Gauss-Seidel smoother used in the preconditioner (Successive Over Relaxation). This method runs only on CPU.

10. **aijcl_gamg_sor_cg:** PETSc provides a CG implementation with a geometric-algebraic multigrid preconditioner (GAMG). Sor is the Gauss-Seidel smoother used in the preconditioner (Successive Over Relaxation). This method runs on CPU and GPU using the viennaCL wrapper (https://github.com/viennacl/viennacl-dev); sor preconditioning is applied on the CPU.

Platform and software stack

The test platform is the Marconi100 machine at CINECA, an accelerated cluster based on IBM Power9 architecture and NVIDIA Volta GPUs (https://www.hpc.cineca.it/hardware/marconi100).

All the tests have been performed on a single node with the following architecture reported in Fig. 1.

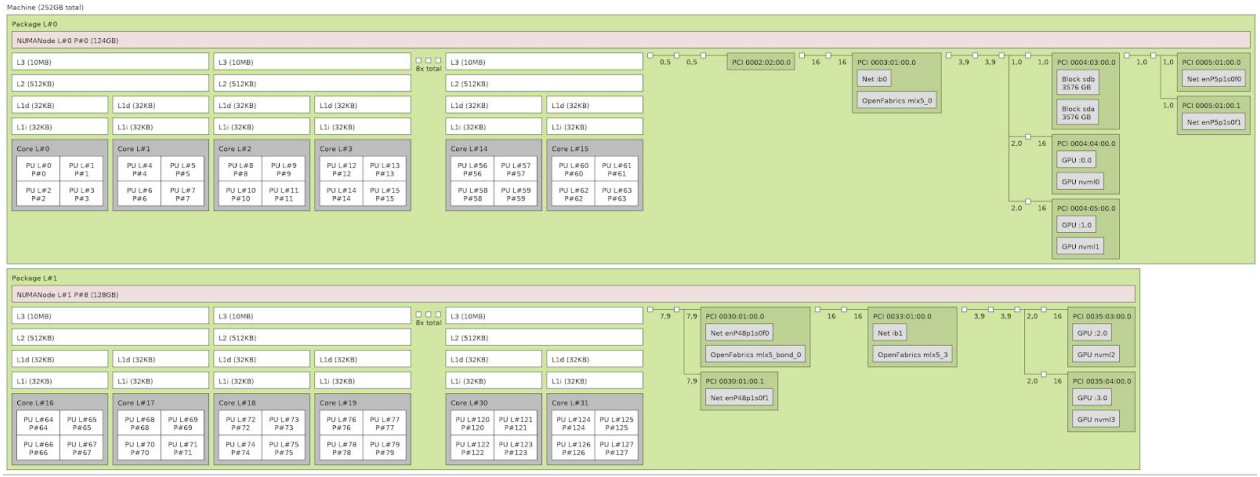

Fig. 1: sketch of the microprocessor architecture for a single M100 node which consists of 2 Power9 cpus and 4 Nvidia V100 GPUs.

All the software and libraries have been built on top of the following toolchain, reported in Table 1:

Linux kernel	**4.14.0**
c compiler	**gnu gcc 8.4.0**
nvidia sdk	**cuda version: 10.1.243** **driver version: 418.116.00**
MPI library	**spectrum MPI 10.3.1**
BLAS Library	**downloaded openBLAS: 0.3.8-dev** **netlib blas (system): 0.3.8 (used for the computations)** **openblas (system): 0.3.9**
LAPACK library	**(system): 3.9.0**
PETSc Library (Master)	**3.13.4 - c5fa24d0279b5bbc232c16e2ebaee358316c51d0**
ML Library	**6.2**
viennaCL sdk	**1.7.0**
Hypre	**2.17.0**

C_FLAGS	**-O3 -mcpu=power9 -fPIC**
CXX FLAGS	**-O3 -mcpu=power9 -std=c++11 -fPIC**
CUDA_FLAGS	**-O3 -Xcompiler -mcpu=power9 -gencode arch=compute_70,code=sm_70 -ccbin g++ -mno-float128**

Note: starting from PETSc 3.13.4 the flag "-mno-float128" is needed to build with cuda 10.1.

Table 1: software stack used in the tests reported in following section

Test-case

The test-case chosen is the 3-D version of the Lid-driven cavity flow tutorial, taken from the code repository of the High Performance Computing Technical Committee [7]. This test-case has a simple geometry boundary conditions, involving transient, isothermal, incompressible laminar flow in a

three-dimensional box domain. The icoFoam solver is used in such test-case. It is intended to stress test the linear algebra solver, most of the time being spent in solving the pressure equation. In this simple geometry, all the boundaries of the box are walls. The top wall moves in the x-direction at the speed of 1 m/s while the other five are stationary. The iterative method chosen is at fixedNORM, with a fixed exit norm value of 10^{-4} for the pressure solver.

Performance results

Figure 2 shows the serial execution time comparison of standard OpenFOAM (*ldu_foam_dic_pcg*) versus 9 different external linear solvers packages set-up in the S-case of the Lid-driven cavity flow benchmark. Two different hardware configurations have been tested: Intel Skylake + V100 GPUs (blu column) and Power9 + V100 GPUs (red column). Figure 1 shows that the Intel Skylake based architecture slightly outperforms IBM Power9 based architecture for the different configurations under analysis. The fastest solver results in the serial *aijcuda_gamg_cheby_cg*, with a speed-up more than 2x compared to standard OpenFOAM solver CPU-based (*ldu_foam_dic_pcg*).

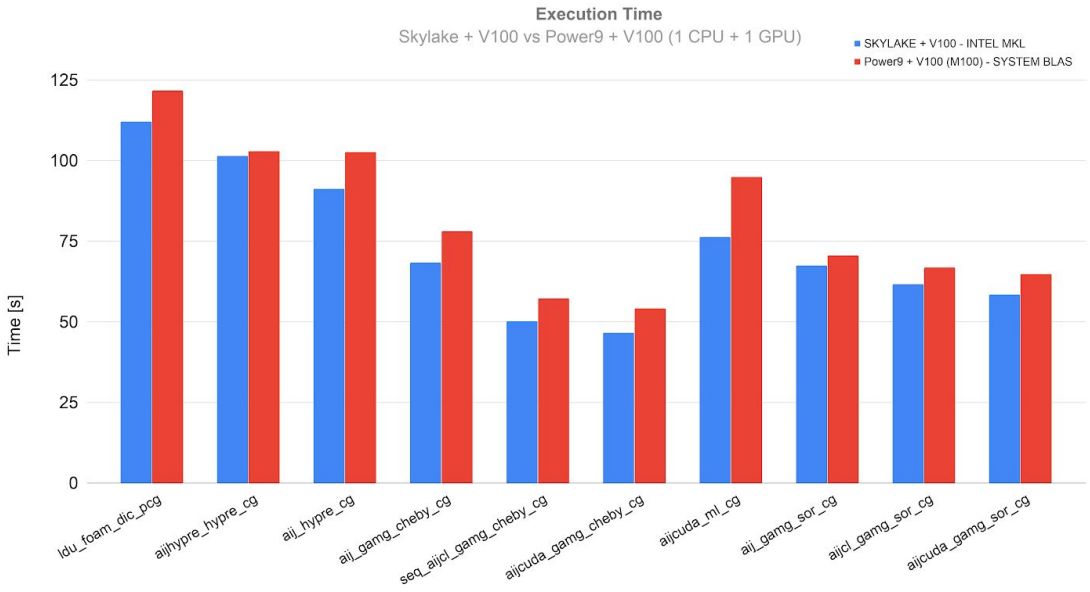

Figure 2: Serial execution time comparison of standard OpenFOAM versus 9 different external linear solvers packages set-up. Two different hardware configurations have been tested: Intel Skylake + V100 GPUs (blu column) and Power9 + V100 GPUs (red column).

Figure 3 shows the comparison of Execution Time (Left y-axis) and Speed-up (right y-axis) of standard OpenFOAM and two external solvers (GPU-based) in two different configurations: 1 core vs 1 core + 1 GPU (a) and 1 node vs 1 core + 1 GPU (b). The test-case is the M version of the Lid-driven cavity flow benchmark.S v

Figure 3-A shows that in scalar mode the GPU-based solvers outperform the CPU based solver of a factor between 4 and 5 x. Figure 3-B shows that with a full node CPU configuration (i.e. 32 cores Power9), versus 1 core + 1 GPUs configuration, the GPU-based solvers are slower of a factor around 0,27-0,28 x.

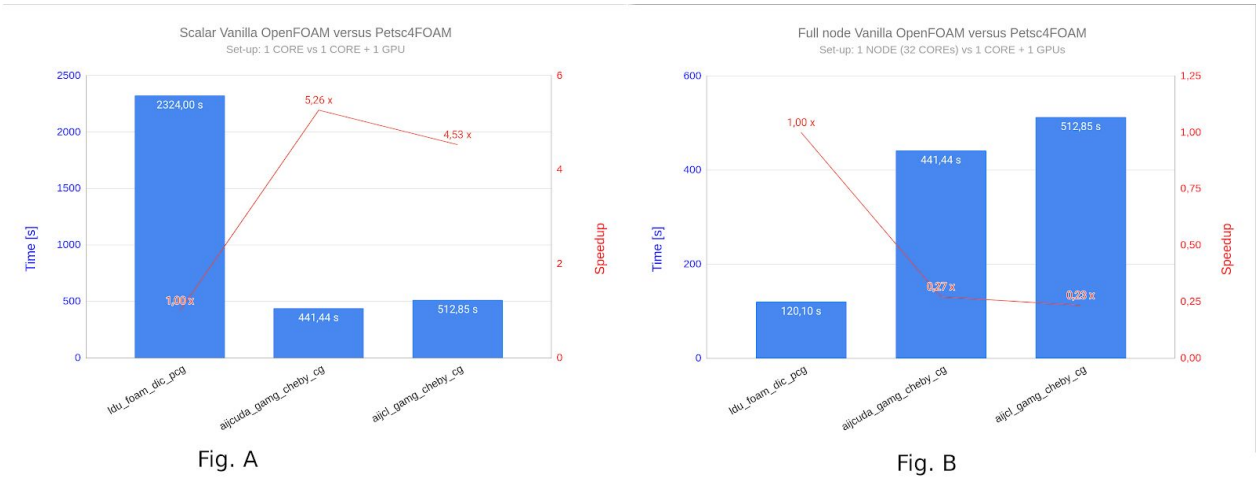

Figure 3: Comparison of Execution Time (Left y-axis) and Speed-up (right y-axis) of standard OpenFOAM and two external solvers (GPU-based) in two different configurations: 1 core vs 1 core + 1 GPU (a) and 1 node vs 1 core + 1 GPU (b).

Figure 4 shows the comparison of Execution Time and Speed-up of standard OpenFOAM and four external solvers (GPU-based) in a full node configuration (32 cores + 4 GPUs) and in the M-case of the

Lid-driven cavity flow benchmark. In such cases, the GPU-based solvers outperform the standard OpenFOAM of a factor 2x without caching and up to a factor 3,5 x when the caching is enabled.

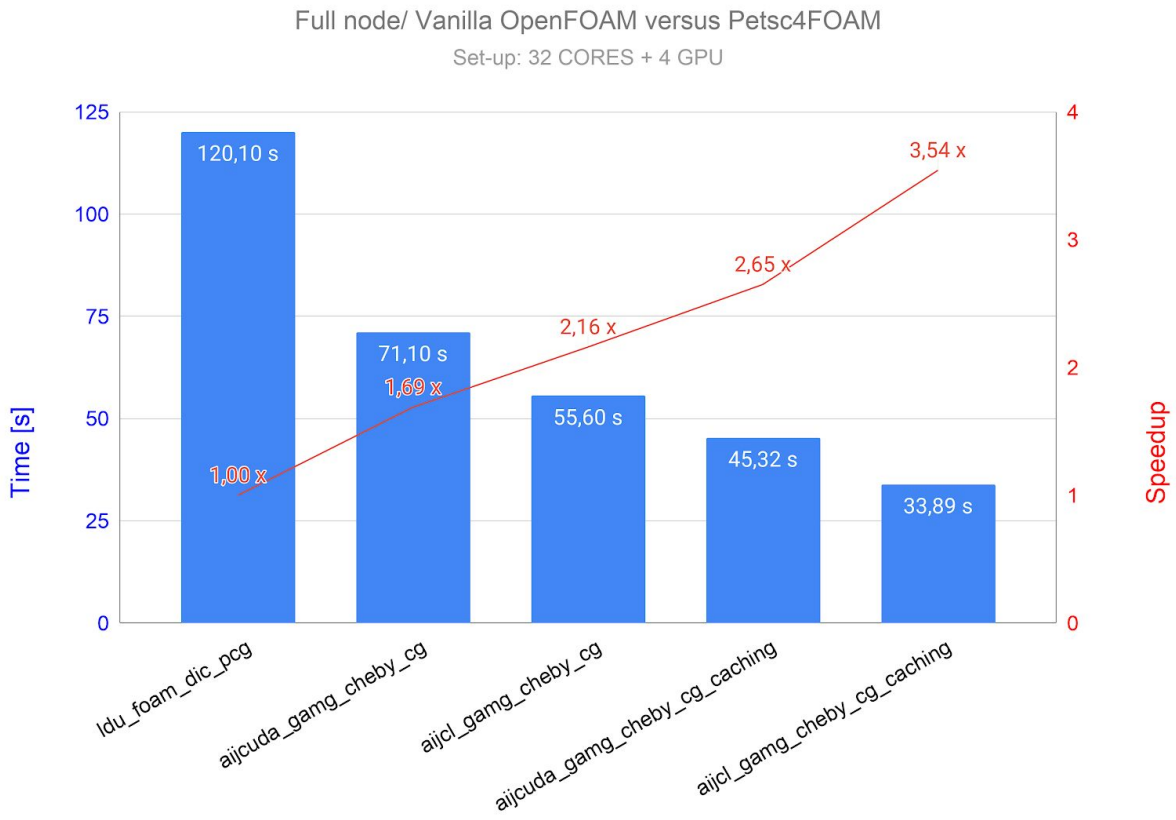

Figure 4: Comparison of Execution Time and Speed-up of standard OpenFOAM and four external solvers (GPU-based) in a full node configuration (32 cores + 4 GPUs).

Further work

In the future we plan to:

- Add the FOAM-GAMG-PCG in the list of solver to be tested; it is an OpenFOAM iterative solver that uses a Conjugate Gradient (CG) accelerated by a generalized geometric-algebraic multigrid (GAMG) method that supports both geometrical and algebraic multigrid.
- Deeply profile the GPU-PETSc implementation in order to investigate and possibly fix the bottlenecks.
- Add AMGx [7], (using the PETSc-wrapper [8]) to the set of solvers we want to compare.
- Use architecture similar to M100 for hardware comparison, that is SUMMIT or SIERRA from OKNL.

References

1. Satish Balay and Shrirang Abhyankar and Mark F. Adams and Jed Brown and Peter Brune, and Kris Buschelman and Lisandro Dalcin and Alp Dener and Victor Eijkhout and William D. Gropp, and Dmitry Karpeyev and Dinesh Kaushik and Matthew G. Knepley and Dave A. May and Lois Curfman McInnes, and Richard Tran Mills and Todd Munson and Karl Rupp and Patrick Sanan, and Barry F. Smith and Stefano Zampini and Hong Zhang and Hong Zhang, PETSc Web page (2019). URL https://www.mcs.anl.gov/petsc
2. HYPRE: Scalable Linear Solvers and Multigrid Methods — Computing (2020). URL https://computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods
3. ML: A Massive Parallel Algebraic Multigrid Solver Library for Solving Sparse Linear Systems. URL https://docs.trilinos.org/dev/packages/ml/doc/html/index.html
4. modules/external-solver PETSc4FOAM· GitLab (2020). URL https://develop.openfoam.com/Community/external-solver
5. S. Bnà, I. Spisso, M. Olesen, G. Rossi “PETSc4FOAM: A Library to plug-in PETSc into the OpenFOAM Framework” PRACE White paper
6. Committees / HPC · GitLab (2020). URL https://develop.openfoam.com/committees/hpc

7. AmgX: Multi-Grid Accelerated Linear Solvers for Industrial Applications
https://developer.nvidia.com/blog/amgx-multi-grid-accelerated-linear-solvers-industrial-applications/

8. AmgXWrapper a wrapper to use AmgX from PETSc
https://github.com/barbagroup/AmgXWrapper