The FEniCS Project on AWS Graviton3

Michal Habera [1,2] and Jack S. Hale [2]

Abstract. We show initial performance results executing the FEniCS Project finite element software on Amazon Web Services (AWS) c7g instances with Graviton3 processors. Graviton3 processors are based on the ARM64 instruction set and provide Scalable Vector Extensions (SVE) for single instruction, multiple data (SIMD) operations. The c7g instances include a fast Elastic Fabric Adaptor (EFA) interconnect for low-latency high-bandwidth Message Passing Interface (MPI) based parallel communication. Comparing clang 15 and GCC 12 series compilers for compiling a high-order elasticity finite element kernel our results show that GCC emitted more vectorised loops with variable width SVE instructions than clang. The runtime performance of the GCC compiled kernel was 20 % faster than the clang compiled kernel. We also tested multi-node weak scalability of a Poisson solver on the EFA interconnect up to 512 MPI processes. We find that overall performance and weak scalability of the AWS provisioned cluster is similar to a dedicated AMD EPYC x86-64 HPC installed at the University of Luxembourg.

1. Introduction

The FEniCS Project has been used to write finite element solvers for problems arising in all fields that involve the solution of partial differential equations, including mathematics, biology, physics, engineering, geophysics and mechanics.

A key enabling technology in the FEniCS Project is the use of automatic code generation (compilation) to reduce the tedious and error-prone work associated with building a finite element solver for a given problem. The user is able to express their finite element problem in a high-level domain specific language and have it automatically translated into a low-level C kernel that computes the finite element tensor on any given cell of the problem mesh. An important aspect for achieving good runtime performance is the ability of the C compiler to produce highly efficient assembly code from the C kernel. In order to take advantage of the single instruction multiple data (SIMD) capabilities of modern CPUs the compiler should emit assembly code with appropriate intrinsics using automatic vectorisation.

The FEniCS Project also supports parallel execution using the Message Passing Interface (MPI). The finite element mesh is partitioned between MPI processes and each process is responsible for assembling contributions from its own partition of the mesh. Communication between is then required to ensure a consistent representation of the finite element linear system across processes.

1: Rafinex Sarl, 2: University of Luxembourg.

	Aion node	AWS c7g instance
Processor	2 x (AMD Epyc ROME 7H12, 64 cores @ 2.6 GHz)	1 x (Graviton3, 64 cores @ 2.6 GHz)
Architecture	x86_64, Zen 2 (AVX2)	ARMv8.5, Neoverse V1 (SVE)
Memory	256 GB DDR4 3200 MT/s = 25.6 GB/s 8 NUMA nodes	128 GB DDR5 4800 MT/s = 38.4 GB/s Unified Memory Access (no NUMA)
Total mem. bandwidth	2 x 200 GB/s	1 x 300 GB/s

Table 1. Configuration of the Aion nodes (University of Luxembourg HPC) and AWS c7g (Amazon) instances.

AWS Graviton3-based instances aim to provide a cost effective compute resources, particularly for scientific computing and machine-learning applications. Particularly appealing for running scientific computing codes such as the FEniCS Project are Graviton3's support for Scalable Vector Extension (SVE) instructions and inclusion of Elastic Fabric Adaptor (EFA) interconnect for high-bandwidth low-latency communications between instances.

In this report we show results that aim to answer the following questions:

(1) Do the latest compilers automatically emit ARM SIMD instructions (Scalable Vector Extensions (SVE) and/or Neon) when compiling the generated C finite element kernels, and what is the runtime performance of the compiled finite element kernels?
(2) Does a Poisson solver implemented using the FEniCS Project scale when running with MPI-based distributed memory parallelism across the AWS Elastic Fabric Adapter?

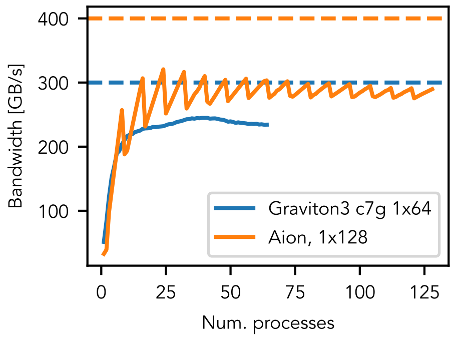
2. Results

In this section we present benchmark results. AWS c7g instances are compared to Aion computing instances available at the University of Luxembourg HPC facilities [**7**]. These instances have different hardware configuration, see Table 1. The purpose of the comparison to ascertain an ARM-based cloud compute cluster can be broadly competitive with a relatively modern x86-64 cluster.

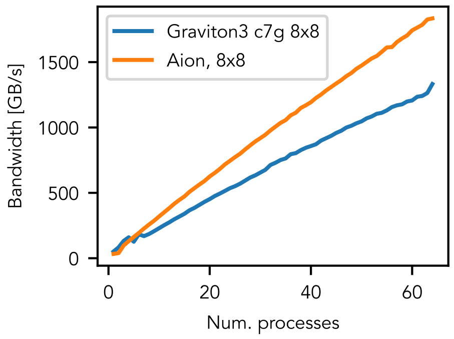
The FEniCS Project components are written in a mixture of Python, modern-style C++20 and ANSI C. All computationally intensive parts (core data structures and algorithms for finite element linear system assembly) of DOLFINx are written in C++20 or ANSI C.

We built the FEniCS Project with the Spack package manager using GCC 12.2.0. We setup Spack to use a version of OpenMPI provided by AWS which includes the appropriate libfabric with support for the EFA interconnect. For the finite element kernel benchmarks we also built LLVM/clang 15.0.7 for the purpose of comparing the performance against assembly code generated with GCC 12.2.0.

2.1. Memory bandwidth. Low-order finite element methods are typically memory bandwidth constrained as the time taken to load and store data from main memory (e.g. the mesh geometry) dominates the time taken to compute the finite element cell tensor itself. Understanding the memory bandwidth characteristics of a processor is therefore important for ensuring optimal performance.



(a) Single-node.



(b) Multi-node.

Figure 1. STREAM benchmark.

STREAM and STREAM MPI [**6, 5**] are the industry standard benchmarks for measuring sustained memory bandwidth performance. It estimates memory bandwidth from memory intense operations (copy, scale, add) on large arrays. In Figure 1a results for the copy operation for single-node benchmark are shown. For the single-node benchmark theoretical peak memory bandwidth of $400\,\mathrm{GB\,s^{-1}}$ for Aion and $300\,\mathrm{GB\,s^{-1}}$ for AWS c7g are reached within 80 %, which is considered a reasonable outcome of the STREAM benchmark. Bandwidth saturation is observed around 20 % of the node utilisation. Both curves show different characteristics of the saturation point due to different memory access configuration. On the Aion instances there are 8 non-unified memory access (NUMA) nodes of 16 cores each, while AWS c7g was setup with unified memory access.

In terms of multi-node scalability, low utilisation (8 processes on 8 nodes each) helps to escape the bandwidth limitations as expected, see Figure 1b.

2.2. Finite element kernels. In order to measure the performance of a standard FEniCS user finite element code we used the Local Finite Element Operator Benchmarks repository [**2**]. The benchmark measures execution time for local finite element kernel generated by the FEniCS Form Compiler (FFCx), [**3**]. Two types of kernels were generated for the purpose of this paper. Matrix-free three-dimensional elasticity kernel represents a finite element discretisation of the action of elasticity (stiffness) operator with spatially varying material property $\kappa(x)$, i.e.

$$v_i = \sum_j A_{ij} w_j = \sum_j w_j \int_K \kappa(x) \mathrm{sym}(\nabla \phi_i) : \mathrm{sym}(\nabla \phi_j) \mathrm{d}x, \tag{1}$$

where K is a fixed reference tetrahedron and $w_j \in \mathbb{R}^n$ is a fixed, prescribed vector. The generated kernel assembles a double precision vector $v_i \in \mathbb{R}^n$, where $n = 30$ for second-order discretization (low-order) and $n = 84$ for sixth-order discretization (high-order). Low-order kernels are expected to be memory bandwidth limited, while high-order kernels have higher arithmetic intesity. In addition, the matrix-free version requires fewer copy operations in comparison to the assembly of a matrix, increasing the ratio of floating-point operations to memory loads and stores. Consequently for the higher-order kernels there is the scope for significant performance increases if the compiler can automatically emit SIMD instructions.

	Compiler	Aion	AWS c7g
Ofast, native, vectorized	gcc 12.2.0	-Ofast -march=znver2 -mtune=znver2	-Ofast -mcpu=neoverse-v1
	clang 15.0.7	-Ofast -march=znver2 -mtune=znver2	-Ofast -mcpu=neoverse-v1
Ofast, native, no vec.	gcc 12.2.0	-Ofast -march=znver2 -mtune=znver2 -fno-tree-vectorize	-Ofast -mcpu=neoverse-v1 -fno-tree-vectorize
	clang 15.0.7	-Ofast -march=znver2 -mtune=znver2 -fno-slp-vectorize -fno-vectorize	-Ofast -mcpu=neoverse-v1 -fno-slp-vectorize -fno-vectorize
O2, no vec.	gcc 12.2.0	-O2 -fno-tree-vectorize	-O2 -fno-tree-vectorize
	clang 15.0.7	-O2 -fno-slp-vectorize -fno-vectorize	-O2 -fno-slp-vectorize -fno-vectorize

Table 2. Compiler versions and compilation flags used for finite element kernel benchmarks.

Different compilers and compiler options were used to assess the performance differences on the AWS c7g instances, see Table 2.

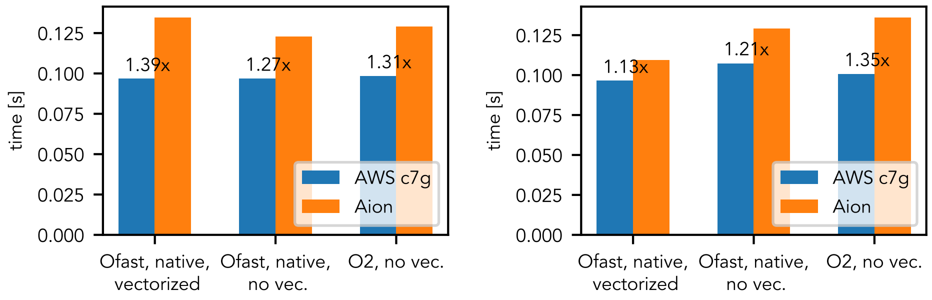
Results for kernel benchmarks are included in Figure 2 and Figure 3. Low-order kernels show no dependence on compiler setup. On the other hand, AWS c7g shows 1.3x speed-up over Aion due to higher memory bandwidth for a single process (DDR5 vs. DDR4).

High-order kernels, which are expected to benefit from compiler optimization, show this trend clearly. GCC 12.2.0 has the most consistent performance for both tested instances. Both Clang and GCC auto-vectorizers perform well, producing a noticeable speed-up (> 2x) in the most optimized setting. Clang 15.0.7 appears to deliver sub-optimal loop auto-vectorization compared to GCC 12.2.0. Optimization reports (`-Rpass=loop-vectorize` for Clang and `-fopt-info-vec-optimized`) revealed that GCC managed to produce Scalable Vector Extensions (SVE) with variable width for three loops in the generated kernel, while Clang produced only one. The other two loops were vectorized, but using fixed vectorization width of two.

2.3. Parallel scalability. Results for the parallel scalability were produced using performance test codes for FEniCSx [**8**] built against DOLFINx 0.6.0 and PETSc 3.18 [**1**].

The Poisson equation solver benchmark consists of the following measured steps:

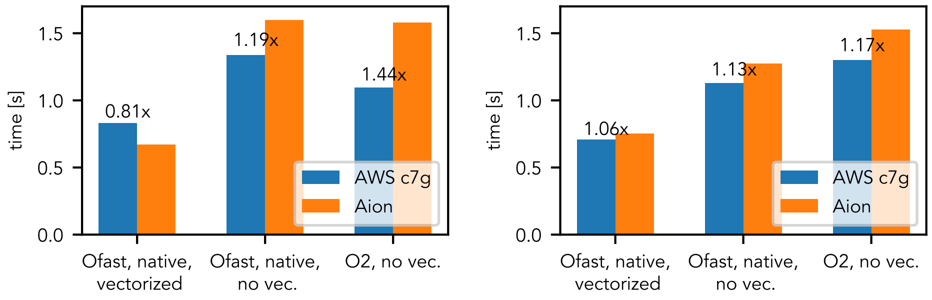
(1) Create mesh. Create a unit cube mesh and discretise using linear tetrahedral cells. Partition the mesh with Parmetis partitioner and distribute. Compute cell-to-edge connectivities.



(a) Clang 15.0.7.

(b) GCC 12.2.0.

Figure 2. Low-order Elasticity operator assembly. Relative speed-up of AWS c7g is included above bars.



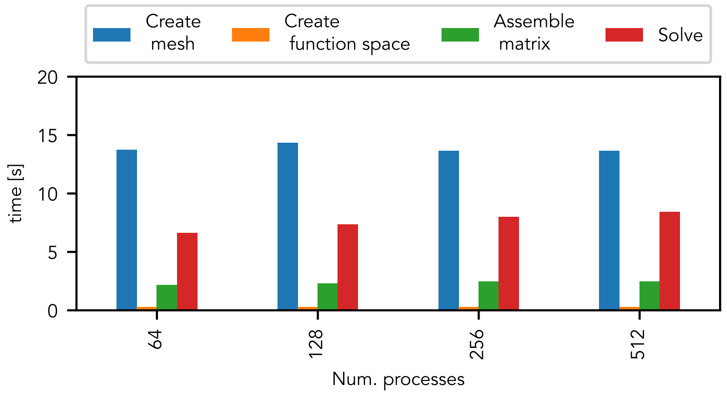
(a) Clang 15.0.7.

(b) GCC 12.2.0.

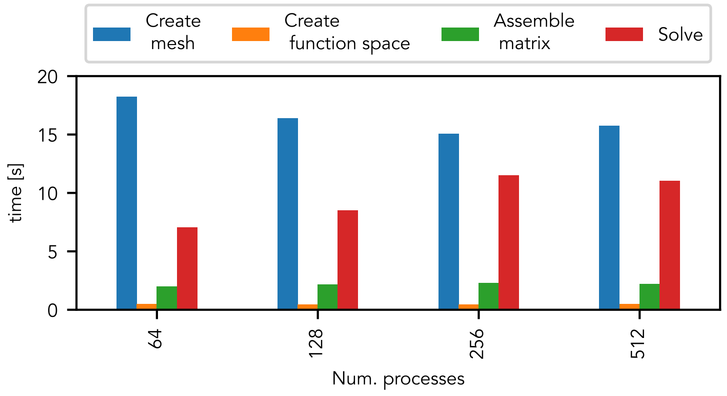
Figure 3. High-order Elasticity operator assembly. Relative speed-up of AWS c7g is included above bars.

(2) Create function space. Create scalar-valued, globally continuous, piecewise linear function space on the mesh.
(3) Assemble matrix. Execute the local Poisson equation kernel over the mesh and assemble PETSc MATMPIAIJ (distributed compressed sparse row) matrix.
(4) Solve. Run Conjugate Gradient (CG) solver with a classical algebraic multigrid (BoomerAMG [**4**]) preconditioner.

Weak scaling results (constant workload of approx. 5×10^5 degrees-of-freedom per process) are shown in Figure 4. Both Aion and AWS c7g show almost constant times for mesh and function space creation. Moreover, matrix assembly has the most ideal weak parallel scalability due to the cell-local nature of the assembly loop and negligible amount of MPI communication during matrix finalisation. The solution step shows small increase (30 %) for both benchmarked instances.



(a) Aion, 5×10^5 degrees-of-freedom per process, 25 % utilisation (32 processes per node).



(b) AWS c7g, 5×10^5 degrees-of-freedom per process, 50 % utilisation (32 processes per node).

Figure 4. Weak parallel scalability of the Poisson equation solver.

3. Conclusion

Benchmarks for memory bandwidth, local finite element kernels and parallel scalability of Poisson solver were executed on nodes Aion and on AWS c7g instances.

Memory bandwidth measured using STREAM MPI confirms higher memory transfer rate of AWS c7g, but superior total bandwidth of $310\,\mathrm{GB\,s^{-1}}$ per Aion node.

In terms of auto-vectorization capabilities of GCC and clang, both produced optimized instructions for the targeted microarchitectures (Zen 2 for Aion and Neoverse V1 for AWS c7g). This observation was confirmed with performance benchmarks based on local finite element kernels for Elasticity (stiffness) operator for low (memory bound) and high (compute bound) orders. GCC 12.2.0 emitted different loop vectorization based on variable width vector instructions (SVE) than clang 15.0.7.

MPI-based distributed memory Poisson equation solver shows good weak scaling on both instances, with results comparable to the in-house University of Luxembourg Aion system.

Executing the FEniCS Project on ARM64, and more specifically on the Graviton3 CPU, has proved to be straightforward. There were no ARM64, or Graviton3, specific adjustments required. Credit for this can largely be attributed to the dedicated work of the Open Source community in ensuring that the entire HPC toolchain is ready for the ARM64 transition, and the engineering work done by AWS on their Graviton3-based instances.

4. Acknowledgements

This project has received compute resources from Amazon Web Services (AWS) through the first and second University of Luxembourg/AWS collaborative Graviton3 call.

This research was funded in whole, or in part, by the National Research Fund (FNR), grant reference COAT/17205623. For the purpose of open access, and in fulfillment of the obligations arising from the grant agreement, the author has applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.

References

[1] Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E. M., Dalcin, L., Dener, A., Eijkhout, V., Faibussowitsch, J., Gropp, W. D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M. G., Kong, F., Kruger, S., May, D. A., McInnes, L. C., Mills, R. T., Mitchell, L., Munson, T., Roman, J. E., Rupp, K., Sanan, P., Sarich, J., Smith, B. F., Zampini, S., Zhang, H., Zhang, H., and Zhang, J. PETSc Web page. `https://petsc.org/`, 2023.

[2] Baratta, I., Richardson, C., Dokken, J. S., and Hermano, A. Local Finite Element Operator Benchmarks, 2023.

[3] Habera, M., Hale, J. S., Richardson, C., Ring, J., Rognes, M., Sime, N., and Wells, G. N. Fenicsx: A sustainable future for the fenics project, 2020.

[4] *hypre*: High performance preconditioners. `https://llnl.gov/casc/hypre`, `https://github.com/hypre-space/hypre`.

[5] McCalpin, J. D. Stream: Sustainable memory bandwidth in high performance computers. Tech. rep., University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[6] McCalpin, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[7] Varrette, S., Cartiaux, H., Peter, S., Kieffer, E., Valette, T., and Olloh, A. Management of an Academic HPC & Research Computing Facility: The ULHPC Experience 2.0. In *Proc. of the 6th ACM High Performance Computing and Cluster Technologies Conf. (HPCCT 2022)* (Fuzhou, China, July 2022), Association for Computing Machinery (ACM).

[8] Wells, G., and Richardson, C. Performance test codes for FEniCSx, 2023.