An Evaluation of the Fujitsu A64FX for HPC Applications

Andrei Poenaru, Tom Deakin, Simon McIntosh-Smith
Department of Computer Science
University of Bristol
Bristol, UK
Email: {andrei.poenaru, tom.deakin, S.McIntosh-Smith}@bristol.ac.uk

Simon D. Hammond, Andrew J. Younge
Center for Computing Research
Sandia National Laboratories
Albuquerque, New Mexico, USA
Email: {sdhammo, ajyoung}@sandia.gov

***Abstract*—Recent generations of supercomputers have adopted different strategies in their attempts to remain competitive in the race to Exascale. In most cases, they rely on accelerators such as GPUs to deliver high arithmetic performance and memory bandwidth. But accelerators come with their own challenges due to their programming models, which can be hard for applications to exploit.**

The current leader in the TOP500 list, the Fugaku system in Japan, has chosen a different route: instead of offloading to accelerators, this system relies on a new generation of general-purpose CPUs to deliver GPU-class performance while maintaining the ease of use of a traditional CPU. This is the Fujitsu A64FX, a design purpose-built for high-performance computing (HPC) based on the Arm AArch64 architecture. It is able to deliver up to 1 TB/s of memory bandwidth by using the same HBM2 technology found in top-end GPUs, and it offers 512-bit-wide vectors through the Scalable Vector Extension (SVE). It is the first CPU to integrate either HBM2 or SVE.

In this paper we evaluated the performance of the A64FX processor on a range of common scientific workloads. We used compute-bound and memory-bandwidth-bound mini-apps, and widely utilised full-scale scientific applications. These benchmarks have been successfully used in the past to quantify performance characteristics in other emerging HPC processors, such as the Arm-based Marvell ThunderX2 and the many-core Intel Xeon Phi. As part of this evaluation, we looked not only at raw application performance, but also at the maturity of the tools available for the A64FX. We uniquely compared all four major HPC compilers that can target the A64FX, including Cray, GNU, Arm and Fujitsu's own compiler.

We found the A64FX to be a strong competitor to mainstream HPC processors. In memory-bandwidth-bound benchmarks, it exceeded 800 GB/s and delivered more than twice the performance of a top-end Xeon or ThunderX2 dual-socket node. We observed particularly good vectorisation performance from the Fujitsu compiler, which was also able to further tune the code for this microarchitecture through techniques such as software pipelining.

***Keywords*-benchmarking; performance analysis; Arm SVE; vectorisation**

I. Introduction

Arm-based processors have been investigated for use in HPC systems since the early 2010s [1]. Initially based on mobile designs, dedicated high-performance cores have been used in recent years to provide performance similar to high-end Intel and AMD x86-based processors. Likewise, the tools ecosystem is mature, stable, production-ready, making Arm a first-class citizen in HPC [2]. Due to their flexibility, an increasing number of vendors are integrating Arm-based cores into their upcoming exascale-era products [3].

Since 2017, a number of systems have deployed Arm in production HPC using the Marvell ThunderX2 (TX2), one of the first Arm-based processors designed specifically for HPC [4], [5]. Studies on these systems have helped identify the types of workloads that are suited for these processors, as well as what their weakness are: TX2 offered a large number of cores and high memory bandwidth, but its short 128-bit-wide vectors make it less suited for compute-intensive applications.

In 2020, the Fugaku system in Japan deployed a new Arm-based design: the A64FX built by Fujitsu [6]. The A64FX improved on both aspects compared to the TX2, implementing for the first time in a CPU HBM2 memory that offers a peak of 1 TB/s of bandwidth and 512-bit vectors based on the Arm Scalable Vector Extension (SVE) [7]. This system was ranked #1 in the TOP500 list in June 2020, and since then several other HPC centres have been adopting the A64FX processors for their own deployments.

In this paper, we evaluate the performance of the Fujitsu A64FX processor on a range of scientific mini-apps and full applications. We compare the A64FX with the other mainstream HPC processors at the time of writing, and we devote special attention to its other Arm-based competitors.

II. Background

The A64FX is the new HPC-first processor designed for the Japanese supercomputer Fugaku. Its core design is custom-made by Fujitsu based on the ARMv8.2 architecture with extensions. The chips contain 48 cores running at up to 2.2 GHz, without simultaneous multithreading (SMT). They are used in single-socket configurations, connected to either TofuD or 100 Gbps InfiniBand networking [8].

An A64FX chips houses four stacks of HBM2 memory. It is the first CPU to utilise HBM2 memory, which had only been used on GPUs before. Each stack is directly attached to a subset of 12 cores, known as a *Core Memory Group (CMG)*. Each core has a private Level 1 cache, but Level 2 (the Last-Level Cache) is shared between cores in a CMG. To the operating system, each CMG appears as a separate NUMA node, and in order to achieve high performance the latency between these nodes needs to be carefully considered.

The A64FX is also the first hardware implementation of SVE. SVE is a vector-length-agnostic (VLA) instruction architecture, allowing each implementation to choose its desired vector length, while ensuring that the same code remains compatible with all implementations. In the A64FX, the native vector width is 512 bits, chosen after experiments in simulation have suggested it is efficient for a range of applications important for the Fugaku supercomputer [9]. As a successor to the NEON ASIMD vector instruction set used in previous Arm-based processors, it offers a wider range of instructions, including gather loads and scatter stores, and per-lane predication for all operations. These features are important for the tuning of low-level optimised math libraries [10].

III. Performance Evaluation Methodology

To evaluate the performance of the A64FX for HPC workloads, we used mini-apps representative of common classes of HPC applications, as well as full-scale codes that are widely used in supercomputing centres around the world. We chose these benchmarks because their performance profiles closely resemble real workloads, and hence should provide a good indication of the real-world performance achievable by these processors. We split them according to the type of resource they depend on most heavily: memory bandwidth or raw compute performance.

Using these benchmarks, we compared the performance achieved by the A64FX with that of other common HPC processors. The platforms we compare against are the Arm-based Marvell ThunderX2, AWS Graviton 2 (in an `M6g.metal` EC2 instance), and Ampere Altra, and the x86-based Intel Cascade Lake (CLX) and AMD EPYC Rome. At the time of writing, these represent the top offerings from the most widely utilised vendors in HPC. The specifications of these processors are given in Table I. Note that the A64FX and Graviton 2 can only be used in single-socket configurations, but the other processors were used in dual-socket nodes.

On all platforms, we used the latest versions of the common HPC compilers: GCC 11.1 supports all the platforms in this study, Arm Compiler for Linux (ACfL) 21.0 supports all the Arm-based targets, Intel Compiler 19.1 (part of the 2020.4 package) supports all the x86-based processors, Cray Compilation Environment (CCE) 11.0 supports all the platforms except the Graviton 2 and the Altra, and Fujitsu Compiler 4.3 supports the A64FX only. There were two exceptions to the above:

- The latest version of CCE available for the A64FX is a pre-release version based on 10.0. This uses the legacy Cray-proprietary frontend instead of the Clang-based frontend used in CCE 11.0;
- There was a regression in the performance of the TeaLeaf benchmark with CCE 11.0, so 10.0 was used to obtain the fastest results for this application.

A. Bandwidth-Bound Benchmarks

To evaluate the best-case achievable memory bandwidth, we used **BabelStream** [11], a C++ implementation of the *de facto* memory bandwidth benchmark, STREAM [12]. BabelStream contains implementations in many programming models, and for this work we used the baseline OpenMP version.

We used the mini-apps **TeaLeaf** [13] and **CloverLeaf** [14] as representative bandwidth-bound workloads. These are both written in Fortran, using hybrid MPI and OpenMP, and they solve equations for heat diffusion and hydrodynamics, respectively. We have studied these extensively in the past and found that their performance correlates well with STREAM performance. Of the two, CloverLeaf is slightly more computationally intensive, as it includes divisions and trigonometry functions.

Finally, we evaluated the performance of **OpenFOAM** [15], a well-known computational fluid dynamics (CFD) application and one of the top 10 most heavily used applications on ARHCER, the UK's national supercomputer. We used version 2006 of the code, the DrivAer open-source test-case [16], and the standard `simpleFoam` solver, applied for 50 time steps. Because the time reported for the first step includes some initialisation overhead, we excluded it from the final benchmark times.

B. Compute-Bound Benchmarks

We used **miniBUDE** for a compute-bound mini-app. This is a molecular docking benchmark developed at the University of Bristol which has previously been shown to achieve close to 60% of peak arithmetic performance on contemporary HPC hardware [17]. The code is implemented in several programming models, of which we used the standard OpenMP implementation here. The performance reported for miniBUDE is in the number of poses computed per unit time.

Another benchmark studied is **SPARTA**, a Direct Simulation Monte Carlo (DSMC) mini-app from Sandia National Laboratories designed for large systems [18]. It is implemented in C++ and MPI, with optional support for threading through the Kokkos library [19]. As a Monte Carlo application, this code is challenging to vectorise and its memory access patterns are irregular.

We took **MiniFMM** [20] to represent applications that use task-based parallelism instead of traditional loop-based parallelism. For this benchmark, we used the provided input set based on a Plummer distribution and recorded the total time taken.

Table I: Hardware specifications of the processors benchmarked.

CPU	Cores	Clock Speed (GHz)		Compute Peak (DP TFLOP/s)	Bandwidth Peak (GB/s)
		Base	Boost		
AMD Rome 7742	2 × 64	2.25	3.4	6.9	410
Ampere Altra Q80-30	2 × 80	3.0	—	3.8	410
AWS Graviton 2 M6g.metal	64	2.5	—	1.3	205
Fujitsu A64FX	48	1.8	—	2.8	1,024
Intel Cascade Lake 6230	2 × 20	2.1	3.9	2.0	375
Marvell ThunderX2	2 × 64	2.2	2.5	1.3	320

A good example of a widely used compute-bound application is **GROMACS**. We used GROMACS 2021.1 and two different benchmarks to evaluate the performance of the systems tested under different conditions:

- The integrated `nonbonded-benchmark`, which runs in flat OpenMP mode and is heavily compute bound. This does not require any input files and runs a Particle Mesh Ewald (PME) simulation; the size parameter for this benchmark was set to 64;
- The `ion_channel_vsites` benchmark, which simulates a membrane protein system comprising around 145,000 atoms. It uses FFTs and represents a realistic use-case for GROMACS in modelling drug molecules. Compared to `nonbonded-benchmark`, PME calculations in `ion_channel_vsites` only take about ⅓ of the total time. We ran this benchmark for 5000 steps of 5 fs.

The 2021.1 release includes initial support for SVE through the GROMACS SIMD abstraction layer [21], although at the time of writing this can only be used with the GNU compiler.

IV. Results and Performance Analysis

A. Benchmark Results

BabelStream

We were able to achieve 824 GB/s in the Triad run on BabelStream on the A64FX, which represents more than 80% of the platform's peak memory bandwidth. This result is more than double that of the next best platform for memory bandwidth. High memory bandwidth is of course expected due to the use of HBM2 on A64FX compared to traditional DDR-DRAM used by the other platforms.

We achieved this result using the Fujitsu compiler, which utilises zero-fill (`zfill`) instructions to zero cache lines before writing to them. This prevents the hardware from first loading the data from memory, because it will be overwritten anyway; it essentially emulates streaming stores even though the architecture doesn't support it explicitly. The other compilers do not use this procedure, and so observed memory bandwidth there was lower at around 600 GB/s.

It was also important for this benchmark to set the `XOS_MMM_L_PAGING_POLICY` environment variable to `demand:demand:demand`. This controls how memory pages are allocated between the four NUMA domains in the A64FX, ensuring they are placed in the same CMG where they are needed, as opposed to that of the core that first started running the program.

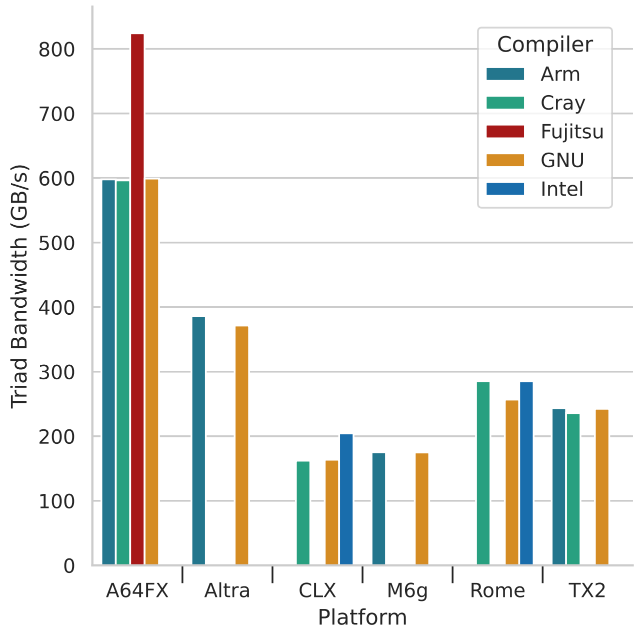


Figure 1: Achieved bandwidth in BabelStream Triad. Higher numbers show better results.

The Ampere Altra obtained the second-highest result with its two sockets of 8-channel DDR4-3200. Even though the TX2 also has 8 channels of DDR4 and in dual-socket configuration, its slower DDR4-2400 memory put its result closer to that of a single-socket Graviton 2 with DDR-3200. The TX2 achieved a lower fraction of peak bandwidth in the BabelStream benchmark, and we observed a regression with CCE 11.0: reverting to version 10.0 produces a result higher by about 15%. The fastest results obtained on each platform are shown in Figure 1. Where a result for a compiler is not shown this is due to that compiler not supporting the platform.

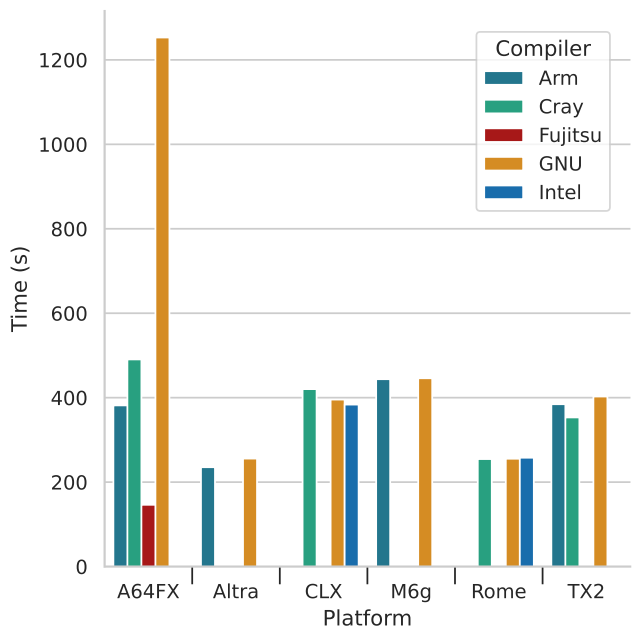


Figure 2: CloverLeaf `bm16` benchmark time. Lower numbers show better results.

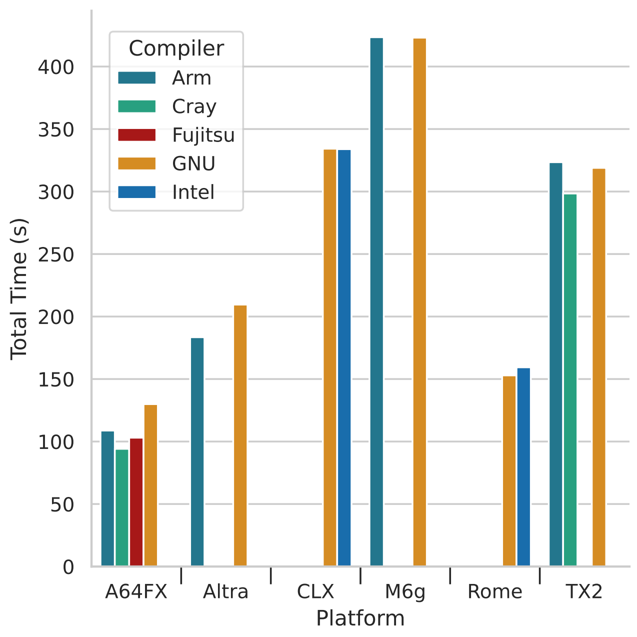


Figure 3: TeaLeaf `bm5` benchmark time. Lower numbers show better results.

CloverLeaf and TeaLeaf

Due to their memory-bandwidth-bound nature, we expected the CloverLeaf and TeaLeaf results to follow similar distributions between platforms as we saw for BabelStream. We largely observed this behaviour, but there were some important differences.

These two applications can be run in hybrid MPI–OpenMP mode, and we tested all viable combinations. On most platforms, we have previously found that running in flat MPI mode, i. e. setting the number of OpenMP threads to 1 and filling all the cores with MPI ranks, generally provides the best performance in single-node configurations [22]. Where there was a difference between flat MPI, flat OpenMP, and hybrid MPI–OpenMP, it was below 10%. On the A64FX, however, we have found larger differences between these run configurations. This section discusses the fastest results obtained, regardless of the run configuration, but Section IV-B goes into more details about the differences.

TeaLeaf contains relatively fewer arithmetic operations compared to CloverLeaf, so memory bandwidth is even more important. In descending order, starting with the fastest result, first was the A64FX, then the Altra at just under twice the run time, then TX2, closely followed by the Graviton 2. Where available, the Cray compiler produced the fastest results. On A64FX, the Fujitsu compiler was a close second, and the Arm and GNU compilers performed similarly on all the platforms.

CloverLeaf includes division operations, which on the A64FX have high execution latency. To work around this, some compilers can replace division with an iterative reciprocal approximation, which is much faster at a slight cost of accuracy. The Arm, Cray, and Fujitsu compilers are all able to apply this optimisation — Cray and Fujitsu do it automatically when targetting the A64FX, and with Arm the user can specify the `-fiterative-reciprocal` flag. The GNU compiler does not apply this optimisation, which results in almost $10\times$ slower performance compared to Fujitsu. Fujitsu is further able to optimise this benchmark by using software pipelining of instructions, a technique which carefully schedules operations such that the processor's out-of-order resources are utilised as efficiently as possible.

Due to all the optimisations it applied, the Fujitsu compiler on A64FX produced the fastest time in this benchmark. However, the Ampere Altra and AMD Rome benefited from their large number of cores and obtained results faster than when using the A64FX with other compilers. The Graviton 2 and the TX2 performed almost identically, suggesting that the newer out-of-order architecture in the Graviton 2 was able to make up for the slightly lower overall memory bandwidth.

The results obtained for CloverLeaf and TeaLeaf are shown in Figures 2 and 3, respectively. These figures show run time, so lower numbers correspond to better performance.

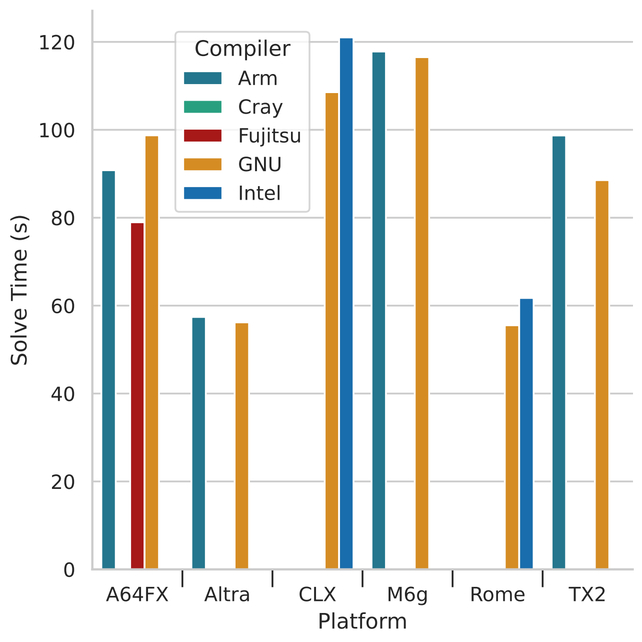


Figure 4: OpenFOAM DrivAer solve time after 50 time steps. The time taken for the first step is excluded. Lower numbers show better results.

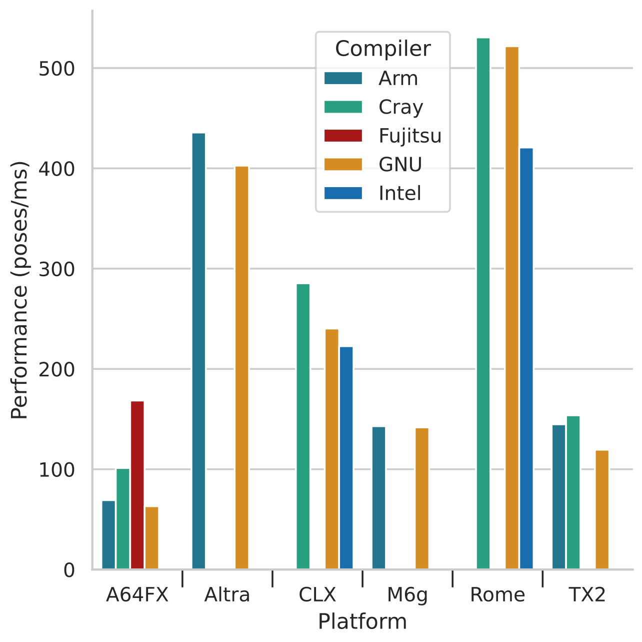


Figure 5: Achieved performance in miniBUDE. Higher numbers show better results.

OpenFOAM

When run on a single-node, OpenFOAM is generally bound by memory bandwidth and does not benefit greatly from vectorisation [4]. These two effects work for and against the A64FX, respectively: it should see good performance from the HBM2 memory, but the 512-bit SVE may not bring a significant improvement over NEON. The results showed that the fastest processor in this benchmark was the AMD Rome, followed by the Ampere Altra, suggesting that the large amount of total L2 cache — 1 MB/core in both the these processors — helped more than HBM2 did on A64FX. The Fujitsu compiler was again the fastest choice on the A64FX, but this time the differences to the other compilers were smaller; Arm and GNU produced similar results on A64FX and Graviton 2. Figure 4 shows the results on all platforms.

miniBUDE

miniBUDE scales very well to many-core architectures — the full BUDE application is routinely run on GPUs. As expected, the results for this benchmark followed the peak compute performance of the processors: TX2 and Graviton 2 achieved similar results, Cascade Lake was more than twice as fast, and the Altra obtained the highest result of the Arm processors, only surpassed by the AMD Rome. On the A64FX, the Fujitsu compiler was able produce better optimised code compared to other compilers, reaching almost $3\times$ the performance obtained with ACfL and GCC, and more than $1.5\times$ the performance of the CCE-compiled binary. However, the performance achieved by the Altra was over twice that of the A64FX, despite their difference in peak performance being lower than a factor of 2, showing that its high core count can rival higher vector width as long as the application parallelises well. The results for miniBUDE are presented in Figure 5.

SPARTA

The performance of SPARTA scaled very well with the number of cores available. There was virtually no vectorised code on any of the platforms, and the choice of compiler made little difference towards the final run time on this benchmark. The data access patterns of this application were not cache-friendly, with only 58.5% of the requests hitting L2 cache, so a lot of time was spent fetching data from main memory.

In general, GCC offered the highest performance on most platforms, being only slightly slower than the Intel compiler on Cascade Lake and Rome. The TX2, Graviton 2, and Cascade Lake achieved similar results, despite the narrower vectors available on the Arm-based platforms. On the A64FX, the Fujitsu compiler failed to link the benchmark, in either Trad or Clang mode, and without its aggressive optimisations the platform's low out-of-order resources led to a slower benchmark time. Figure 6 shows the results on all platforms.

MiniFMM

We found the vectorisation efficiency of the MiniFMM benchmark to be low on all the Arm-based platforms. With NEON, a lot of the code was not vectorised, and although

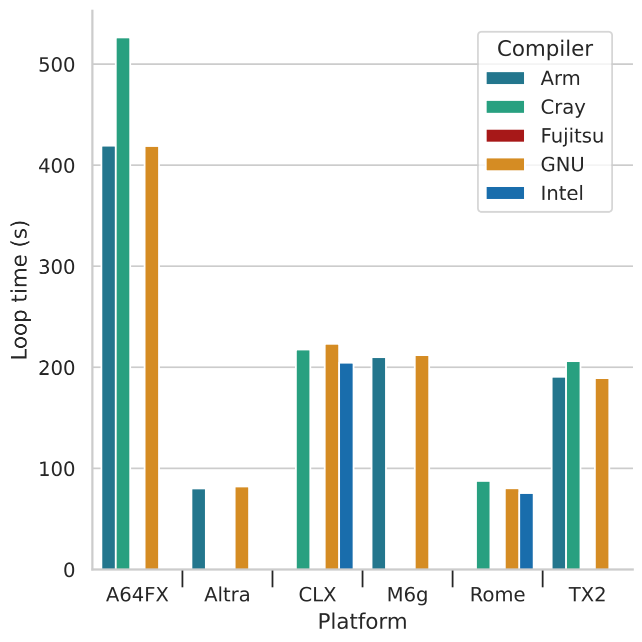


Figure 6: SPARTA benchmark time using the collisional flow input, 10M cells, and 5000 iterations. Lower numbers show better results.

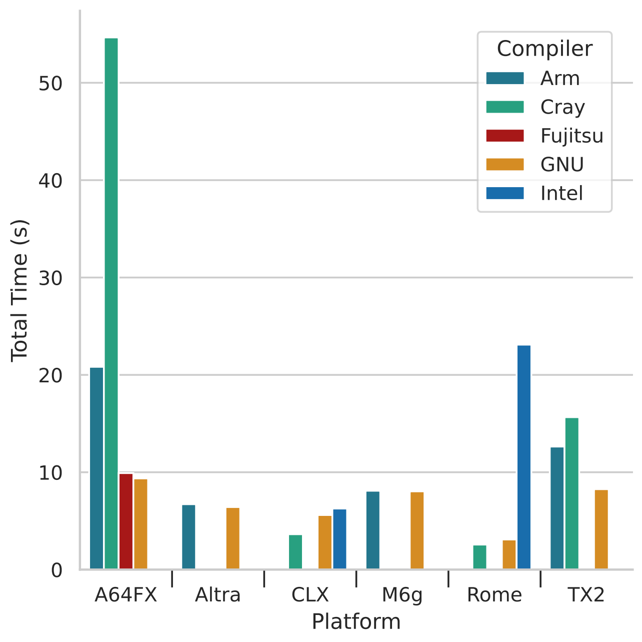


Figure 7: MiniFMM benchmark time using a Plummer and the OpenMP tasks implementation. Lower numbers show better results.

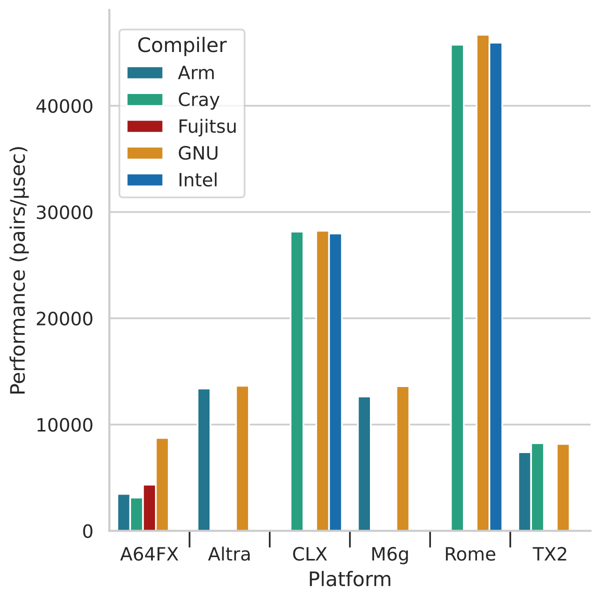
SVE was able to address that to an extent, many operations were masked and utilised only a fraction of the available vector width [23]. In addition, it did not scale well to high core counts: beyond 60 cores, the run time stopped decreasing, and above 80 it started *increasing*. For the Altra, this meant that fewer than half of the available cores were utilised. On the x86-based platforms, there was more benefit from vectorisation, but the high core count of Rome again did not show a tangible benefit in this benchmark.

The best result was similar on all the Arm platforms, with a slight advantage to Altra due to its high clock speed. We found that the Cray and Arm compilers were less efficient at exploiting parallelism in this task-based benchmark compared to GCC, which was the best compiler choice even on the A64FX. The Intel compiler performed well on Cascade Lake, but it was significantly slower compared to Cray and GNU on Rome. The results are presented in Figure 7.

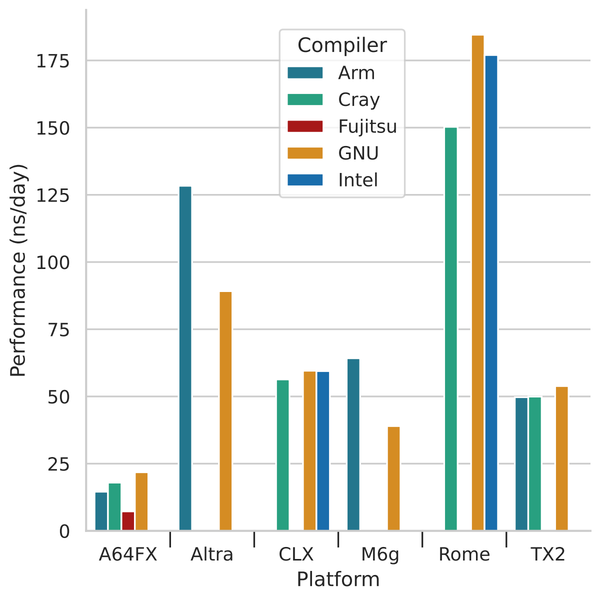
GROMACS

With `nonbonded-benchmark`, there were significant performance differences between the x86 platforms, where AVX2 and AVX-512 could be used, compared to the Arm platforms. This workload is heavily compute-bound, so the wider vector length constituted a significant advantage. This benchmark cannot be used with MPI and the maximum number of OpenMP threads allowed in GROMACS is 64, which limited the performance achieved by the Altra and the Rome platforms, and resulted in similar performance on Altra and Graviton 2, since both use the same Neoverse N1 cores. Even though the early SVE implementation for A64FX — which was only usable with the GNU compiler — achieved almost twice the performance of the NEON implementation on the Fujitsu platform, it still only produced results similar to a ThunderX2 running NEON; with more optimised code, it should be possible for the A64FX to produce results several times faster than this. On the other platforms, there were virtually no differences between the compilers, because the performance-critical PME kernels in GROMACS are written in hand-tuned intrinsics.

However, the more realistic `ion_channel_vsites` test case revealed different behaviour. On the one hand, the change in performance profile to place more emphasis on the memory system brought the results of a 64-core TX2 node very close to that of a 40-core Cascade Lake node, despite the difference in native vector length between the two processors. On the other hand, the benefit from the early SVE implementation on the A64FX was lower and closed the performance gap to the other compilers, which still used the NEON implementation. With core usage no longer limited to 64, the Altra's performance increased relative to the other platforms with lower core counts. We observed that the optimised FFT implementations in the Arm Performance Libraries performed significantly better on the Neoverse N1, granting as $1.43\times$ speed-up over FFTW; on the other platforms, FFTW built from source, Cray's Optimised FFTW and ArmPL (on TX2) performed similarly. For OpenMP parallelism, the best choice of number of



(a) `nonbonded-benchmark`



(b) `ion_channel_vsites`

Figure 8: Achieved performance in two GROMACS benchmarks. The open-source FFTW library was used with GCC and Fujitsu, ArmPL was used with the ACfL, MKL with the Intel compiler, and Cray's optimised build of FFTW was used with CCE. Higher numbers show better results.

threads was 2 or 4 on all the platforms, with enough MPI rank run to fill all the available hardware threads, i. e. utilising SMT where available.

The results for the two GROMACS benchmarks on all the platforms are shown in Figure 8.

B. Thread Placement on the A64FX

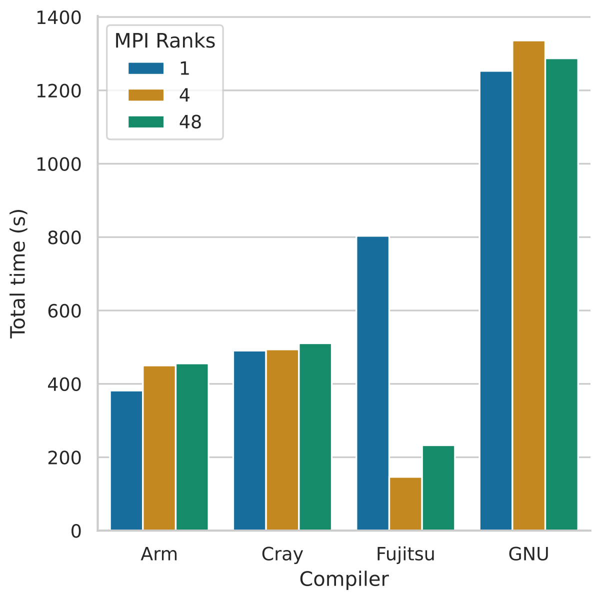
Due to the four NUMA node configuration, placement and binding of MPI ranks and OpenMP threads are particularly important on the A64FX. Three of the benchmarks in this study combine MPI with OpenMP and allow the user to divide parallelism between the two levels: CloverLeaf, TeaLeaf, and SPARTA.

CloverLeaf and TeaLeaf behaved similarly, in that the fastest configuration differed with the compiler used: hybrid MPI–OpenMP, running one rank per CMG and filling all its 12 cores with OpenMP threads, was fastest with all compilers except for Arm, where flat OpenMP was the fastest configuration. The difference between the performance of hybrid MPI and flat MPI was around 5% wth GCC and Cray, and around 15% with ACfL. However, the results were very different when using the Fujitsu compiler: flat OpenMP was the *slowest* configuration, achieving less than 20% the performance of the hybrid configuration, and flat MPI was second, at 62% of the performance of the hybrid run. Placement results for CloverLeaf with all the compilers available on the A64FX are shown in Figure 9a.

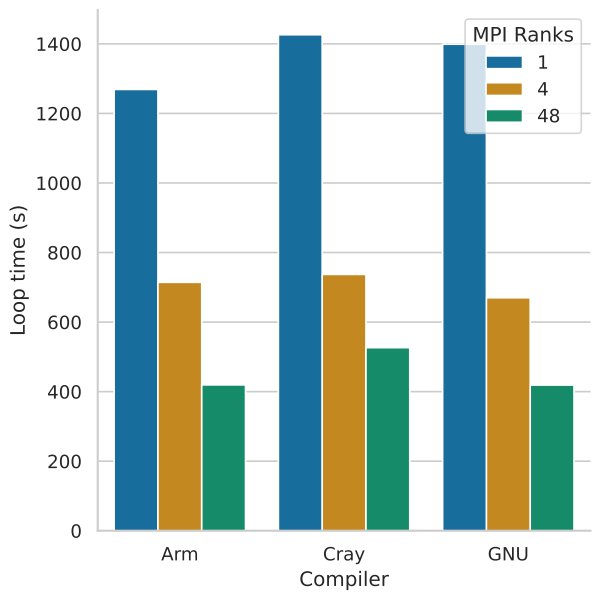
SPARTA failed to build with the Fujitsu compiler; the other compilers all performed similarly to each other. For this benchmark, flat MPI was the fastest configuration, but here switching some of the parallelism to Kokkos threads *reduced* performance by up to 2×. Even though we used Kokkos 3.4, the latest at the time of writing and which supports the A64FX target, the code it generates may not yet be as optimal as OpenMP produced directly by a compiler. Figure 9b shows the run time of SPARTA under the three different placement strategies.

We found that the four compilers that can target the A64FX have different default semantics for binding threads, and sometimes these are different from the optimal configuration. The following settings reliably produced correct rank and thread placement on all the compilers tested:

- For flat MPI, set `OMP_NUM_THREADS` to 1 and bind each rank to a core, e. g. using the `-bind-to core` argument to `mpirun`;
- For flat OpenMP, disable binding of MPI ranks, in order to prevent all threads from being bound to the same object, using `-bind-to none`, then explicitly split threads between all the NUMA nodes using `OMP_PLACES=cores OMP_PROC_BIND=spread`;
- For hybrid OpenMP–MPI, fill all NUMA nodes equally with rank using `-map-by numa`, bind all its threads to the NUMA node with `-bind-to numa`, then spread the OpenMP threads onto the available cores with `OMP_PLACES=cores OMP_PROC_BIND=close`.



(a) CloverLeaf



(b) SPARTA

Figure 9: Comparison of MPI–OpenMP run configurations on A64FX. As many OpenMP threads were used as needed in each case to fill all 48 cores. Lower numbers show better results.

V. Future Work

In this study we have investigated the performance of the A64FX using single-node benchmarks. We have identified strong and weak points of this processor, but when running at scale these may manifest differently. In particular, compute-bound applications can become network-bound, thus increasing the benefits of using A64FX in a large-scale system.

One of the points for improvement that we have identified is around the compiler support for the A64FX. Because of its relatively lightweight microarchitecture, this processor relies on a good optimising compiler with an accurate cost model to schedule instructions well. There is currently a significant gap between the performance of binaries compiled with the Fujitsu Compiler and open-source alternatives, so there is room for further studies on this architecture to suggest and implement compiler improvements.

Finally, when looking at the next generations of high-performance processors, it is essential to understand how microarchitectural design decision affect the performance of applications. This process is known as the *co-design* of hardware and applications and it is a way to ensure that future hardware will provide adequate performance for its intended use cases. Such experiments are generally hard, because modelling hypothetical architectures accurately is an involved task that requires specialised tools. Still, it is essential in the co-design process, which is one of the main motivating factors for the upcoming SimEng simulation framework [24]. SimEng aims to enable fast, accurate, flexible simulations through a simple interface for extending existing processor designs with hypothetical additions[1].

VI. Reproducibility

Instructions on running the benchmarks in this study are available online[2]. The scripts provided obtain the code and any input data, build the applications with the specified compiler, and provide run configurations for the platforms used in this paper.

VII. Conclusion

In this paper, we explored the performance of the Fujitsu A64FX processor on a range of scientific benchmarks. The benchmarks were chosen to cover several important classes of HPC applications, and the results were compared to other common high-performance processors at the time of writing. We gave special attention to Arm-based alternatives, of which we covered the previous-generation Marvell ThunderX2 and the newer AWS Graviton 2 and Ampere Altra. We also compared to the best-in-class x86 processors available at the time of writing.

We found the A64FX to be a competitive processor for HPC. It performed particularly well for memory-bandwidth-bound applications, where its HBM2 with a peak of 1 TB/s was utilised to its full potential. The results on compute-bound benchmarks were mixed: performance was good when using the Fujitu Compiler, which was specifically

[1]https://uob-hpc.github.io/SimEng-Docs/index.html
[2]https://github.com/UoB-HPC/benchmarks

developed to target the A64FX, but with other compilers that do not apply optimisations such as software pipelining, the relatively lower out-of-order capacity of the A64FX led to reduced performance compared to more heavyweight cores.

Acknowledgement

This work used the Isambard UK National Tier-2 HPC Service[3] operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1).

References

[1] N. Rajovic, P. Carpenter, I. Gelado, N. Puzovic and A. Ramirez, 'Are mobile processors ready for HPC?', in *IEEE/ACM Supercomputing Conference*, 2013.

[2] A. Rico, J. A. Joao, C. Adeniyi-Jones and E. Van Hensbergen, 'ARM HPC Ecosystem and the Reemergence of Vectors', in *Proceedings of the Computing Frontiers Conference*, ser. CF'17, Siena, Italy: Association for Computing Machinery, 2017, pp. 329–334, ISBN: 9781450344876. DOI: 10 . 1145 / 3075564 . 3095086.

[3] NVIDIA. 'NVIDIA Announces CPU for Giant AI and High Performance Computing Workloads'. (12 Apr. 2021), [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-announces-cpu-for-giant-ai-and-high-performance-computing-workloads (visited on 09/05/2021).

[4] S. McIntosh-Smith, J. Price, T. Deakin and A. Poenaru, 'A performance analysis of the first generation of HPC-optimized Arm processors', *Concurrency and Computation: Practice and Experience*, vol. 31, no. 16, e5110, 2019. DOI: 10.1002/cpe.5110.

[5] K. Pedretti, A. J. Younge, S. D. Hammond *et al.*, 'Chronicles of Astra: Challenges and Lessons from the First Petascale Arm Supercomputer', in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20, Atlanta, Georgia: IEEE Press, 2020, ISBN: 9781728199986.

[6] M. Sato, 'The Supercomputer "Fugaku" and Arm-SVE enabled A64FX processor for energy-efficiency and sustained application performance', in *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2020, pp. 1–5. DOI: 10.1109/ISPDC51135.2020.00009.

[7] N. Stephens, S. Biles, M. Boettcher *et al.*, 'The ARM scalable vector extension', *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017. DOI: 10.1109/MM.2017.35.

[8] M. Sato, Y. Ishikawa, H. Tomita *et al.*, 'Co-Design for A64FX Manycore Processor and "Fugaku"', in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15. DOI: 10.1109/SC41405.2020.00051.

[9] Y. Kodama, T. Odajima, M. Matsuda, M. Tsuji, J. Lee and M. Sato, 'Preliminary Performance Evaluation of Application Kernels Using ARM SVE with Multiple Vector Lengths', in *2017 IEEE International Conference on Cluster Computing*, IEEE, 2017, pp. 677–684.

[10] C. L. Alappat, N. Meyer, J. Laukemann *et al.*, 'ECM modeling and performance tuning of spmv and lattice QCD on A64FX', *CoRR*, vol. abs/2103.03013, 2021. arXiv: 2103.03013. [Online]. Available: https://arxiv.org/abs/2103.03013.

[11] T. Deakin, J. Price, M. Martineau and S. McIntosh-Smith, 'GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models', in *International Conference on High Performance Computing*, Springer, 2016, pp. 489–507, ISBN: 978-3-319-46079-6. DOI: 10.1007/978-3-319-46079-6_34.

[12] J. D. McCalpin, 'Memory bandwidth and machine balance in current high performance computers', *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19–25, 1995.

[13] M. Martineau, S. McIntosh-Smith and W. Gaudin, 'Assessing the performance portability of modern parallel programming models using TeaLeaf', *Concurrency and Computation: Practice and Experience*, vol. 29, no. 15, 2017, ISSN: 15320626. DOI: 10.1002/cpe.4117.

[14] A. Mallinson, D. Beckingsale, W. Gaudin, J. Herdman, J. Levesque and S. Jarvis, 'CloverLeaf: Preparing Hydrodynamics Codes for Exascale', in *Cray User Group*, Napa Valley, California, USA, May 2013.

[15] H. Jasak, A. Jemcov, Z. Tukovic *et al.*, 'OpenFOAM: A C++ library for complex physics simulations', in *International workshop on coupled methods in numerical dynamics*, IUC Dubrovnik Croatia, vol. 1000, 2007, pp. 1–20.

[16] A. I. Heft, T. Indinger and N. A. Adams, 'Introduction of a new realistic generic car model for aerodynamic investigations', SAE Technical Paper, Tech. Rep., 2012. DOI: 10.4271/2012-01-0168.

[17] A. Poenaru, W.-C. Lin and S. McIntosh-Smith, 'A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application', in *36th International Conference, ISC High Performance 2021*, Frankfurt, Germany, 2021, in press.

[18] M. A. Gallis, J. R. Torczynski, S. J. Plimpton, D. J. Rader and T. Koehler, 'Direct Simulation Monte Carlo: The Quest for Speed', in *29th Intl Symposium on Rarefied Gas Dynamics*, vol. 1628, Xi'an, China: AIP Conference Proceedings, 2014.

[3]https://gw4.ac.uk/isambard/

[19] H. C. Edwards and C. R. Trott, 'Kokkos: Enabling performance portability across manycore architectures', in *2013 Extreme Scaling Workshop (xsw 2013)*, IEEE, 2013, pp. 18–24.

[20] P. Atkinson and S. McIntosh-Smith, 'On the Performance of Parallel Tasking Runtimes for an Irregular Fast Multipole Method Application', in *Scaling OpenMP for Exascale Performance and Portability*, Springer International Publishing, 2017, pp. 92–106, ISBN: 978-3-319-65578-9.

[21] S. Páll and B. Hess, 'A flexible algorithm for calculating pair interactions on simd architectures', *Computer Physics Communications*, vol. 184, no. 12, pp. 2641–2650, 2013, ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2013.06.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465513001975.

[22] T. Deakin, S. McIntosh-Smith, J. Price *et al.*, 'Performance Portability across Diverse Computer Architectures', in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Denver, CO, USA: IEEE, Nov. 2019, pp. 1–13, ISBN: 978-1-72816-003-0. DOI: 10.1109/P3HPC49587.2019.00006.

[23] A. Poenaru and S. McIntosh-Smith, 'Evaluating the Effectiveness of a Vector-Length-Agnostic Instruction Set', in *Euro-Par 2020: Parallel Processing*, (Warsaw, Poland, 24–28 Aug. 2020), M. Malawski and K. Rzadca, Eds., Cham: Springer International Publishing, 2020, pp. 98–114.

[24] S. McIntosh-Smith, J. Jones, H. Waugh and A. Poenaru, 'SimEng: a fast, easy to use, open source processor simulation framework', presented at the ModSim 2021: Workshop on Modeling and Simulation of Systems and Applications, Seattle, WA, USA, 2021, In Review.