Accepted to be Published in: Proceedings of the 2021 IEEE International Conference on Cluster Computing (CLUSTER), EAHPC Workshop, Sept. 7-10, 2021 Portland, Oregon, USA

A64FX – Your Compiler You Must Decide!

Jens Domke
RIKEN Center for Computational Science (R-CCS)
Kobe, Hyogo, 650-0047 Japan
Email: http://domke.gitlab.io/#contact

***Abstract*—The current number one of the TOP500 list, Supercomputer Fugaku, has demonstrated that CPU-only HPC systems aren't dead and CPUs can be used for more than just being the host controller for a discrete accelerators. While the specifications of the chip and overall system architecture, and benchmarks submitted to various lists, like TOP500 and Green500, etc., are clearly highlighting the potential, the proliferation of Arm into the HPC business is rather recent and hence the software stack might not be fully matured and tuned, yet. We test 3 state-of-the-art compiler suite against a broad set of benchmarks. Our measurements show that orders of magnitudes in performance can be gained by deviating from the recommended usage model of the A64FX compute nodes.**

1. Introduction

The HPC community has been testing Arm-based architectures for a few years now [1], [2], [3], and Supercomputer Fugaku [4] is the first large-scale system in the top-end of the TOP500 list, which demonstrates the competitiveness of Arm in a space which recently had been dominated by Intel, AMD, and Nvidia. The benefits of Arm CPUs paired with high bandwidth memory, as in the case of Fujitsu's A64FX processor [5], for the HPC field are clear: (1) Arm CPUs are highly customizable, energy efficient, and there is an existing ecosystem of software, compilers, tools, etc., which is readily available (unlike for the K computer with its SPARC CPU); and (2) most applications executed on HPC systems tend to be memory-bandwidth-bound, as we have shown in a previous study [6]. Although, a different compute-to-bandwidth ratio, as found in A64FX, might challenge this view in individual cases resulting in a greater influence by the compiler onto the performance. Furthermore, despite the dominance of Arm chips in the embedded and low-power space, the system software and compilers, such as the widely used GNU Compiler Collection for embedded systems, might be tuned for metrics other than wide vectors (e.g., Arm's Scalable Vector Extension) and high performance.

It is known among benchmarkers and performance analysts, that Intel's Parallel Studio or AMD's AOCC compilers, depending on the CPU vendor, usually yield a high baseline performance. However, for Fujitsu's A64FX, this choice is not that obvious, yet, and options such as Fujitsu's compiler suite, GNU Compiler Collection, Arm compilers, and HPE/Cray compilers exist, and need to be evaluated.

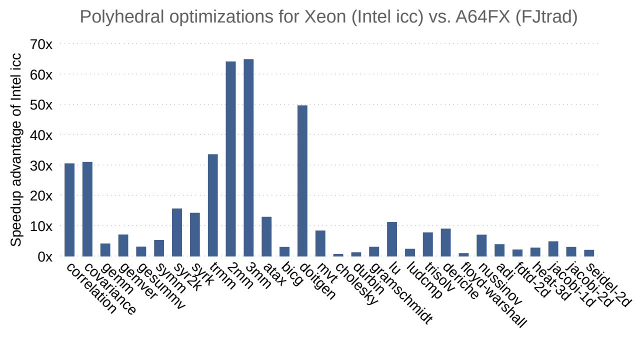


Figure 1. **Unexpected advantage of Xeon vs. A64FX** in PolyBench[large], which **prompted this study**. Recommended compiler/flags used for both.

For example, after Supercomputer Fugaku—or short Fugaku hereafter—was put into production, we experimented with micro kernels, namely PolyBench [7], to debug some unexpected performance discrepancies, especially when compared to an Intel Xeon E5-2650v4 reference CPU. While a compiler-based core-to-core comparison for these single-threaded benchmarks is inherently inaccurate (due to ISA, caches, GHz, etc.), we did not expect the Xeon to execute some tests up to two orders of magnitude faster, see Figure 1. This obviously does not make sense, especially when looking at the *2mm* and *3mm* matrix multiplication cases which should be compute-bound. We will analyze the reasons in greater depth in Section 3.1, and it initiated this study in which we seek to answer the following questions:

- Is the recommended usage model for A64FX, i.e., compiler+flags as well as number of MPI/OMP ranks and threads, ideal or just a starting point?
- Is there a "silver bullet" compiler choice for A64FX?
- Can performance differences, compared to similar x86-based hardware, be attributed to the compiler?

To answer these questions, our contributions are as follows:

- We execute a broad set of micro benchmarks, procurement proxy applications, and real-world application benchmarks under 5 different compiler environments to determine performance trends.
- A detailed discussion of the performance opportunities arising from different compiler options (which can also create challenges for ordinary users), and recommendations for operators and users of A64FX.

2. Measurement Methodology

The performance discrepancy for the *2mm* benchmark was quickly identified. Intel's C compiler reordered the nested loop construct, while Fujitsu's C compiler (fcc) failed to do so, resulting in a 64x speedup on the Xeon core which has less than ½ of a A64FX cores's theoretical peak flop/s. Hence, we started investigating alternative compilers to improve PolyBench and also real-world codes on Fugaku.

2.1. Compiler and Compiler Flags

FUJITSU Software Technical Computing Suite (v4.5.0) is the recommended compiler infrastructure for Fugaku supporting C/C++ and Fortran. It supports two modes, *traditional* and *clang*; the latter being based on an enhanced version of LLVM 7. We utilize both modes in this study, and link to Fujitsu's SSL2 library for linear algebra operations whenever necessary. Most application come with build scripts tuned for individual compilers, and we augment them with *-Kfast,ocl,largepage,lto* for performance, optimization control line (OCL) support, hugepages, and link-time-optimization, respectively.

LLVM Compiler Infrastructure (v12) supports C/C++ via clang; however, flang (LLVM's Fortran frontend) requires currently a host compiler and we experienced many errors using it, and hence we skip flang and directly utilize Fujitsu's *frt* compiler. We test two settings with LLVM, the first being *-Ofast -ffast-math -flto=thin* and the second specifically targeting polyhedral optimizations with *-mllvm -polly -mllvm -polly-vectorizer=polly* and replacing the thin linker with the full linker, since *thin* interfered with *polly*.

GNU Compiler Collection (v10.2.0) supports C/C++ and Fortran, and we use *-O3 -march=native -flto* in addition to the benchmark-specific compiler flags whenever possible.

Hence, we have 5 variations, identified hereafter with *FJtrad*, *FJclang*, *LLVM*, *LLVM+Polly*, and *GNU*. Unless otherwise stated, the flags listed above (or minor variations to avoid compile/runtime issues) are in effect[1].

Other commercial compilers from Arm (a fork of LLVM with additional optimizations and native Fortran-support) and HPE/Cray exist; however, these are currently not available on Fugaku and we could not install them ourselves without acquiring a license. We refer an interested reader to our related work in Section 4 which includes comparison for these compilers as well, but on other benchmarks/systems.

2.2. Benchmarks – From Micro to Macro Level

We test over 100 different kernels and scientific codes from 7 benchmark suites, as outlined hereafter:

Micro Kernels are a collection of 22 kernels[2] extracted from RIKEN priority applications (see later in this Section), which have been used during the Fugaku development for testing and validation. These kernels are OpenMP-parallelized, primarily written in Fortran (except 5), and test various performance-relevant aspects of one core memory group (CMG) of the A64FX processor, i.e., 12 cores (+1 assistant/OS core) and a 8 GiB HBM2 module.

Polyhedral Benchmark suite (in short, PolyBench) is a collection of 30 single-threaded scientific kernels written in C. The input sizes can be tuned for different memory hierarchy levels, and we use the *LARGE* input (exc.: *MEDIUM* for *floyd-warshall*) to stress all memory levels of A64FX.

HPL, HPCG, and BabelStream are commonly known to test the system's compute [8] and memory performance [9], [10], and are used, for example, to rank supercomputers in the TOP500 list. HPL's and HPCG's problem sizes are configured to 36,864 and 120^3, respectively, while we use 2 GiByte long vectors for the stream benchmark.

ECP proxy-apps and RIKEN Fiber mini-apps are collections of so called *proxy applications* which are smaller representative codes and inputs for production applications commonly executed on supercomputers in the USA and Japan. We have studied these codes previously [6], [11], and we refer the reader to these publications for details.

SPEC CPU[speed] and OMP are two suites used by HPC centers and vendors to test compute node capabilities. The former comprises 20 tests. One half are single-threaded, integer-intensive computations and the other half tests multi-threaded, floating-point-heavy scientific applications. The latter are 14 science workloads which are OpenMP-parallelized, too. The benchmarks are implemented in C, Fortran, and C++, or a mix thereof. For our (non-compliant) SPEC runs, we universally select the *train*-ing input sizes.

2.3. Evaluation Environment

All test are performed on 2.2 Ghz A64FX-based nodes of Fugaku [4], [5], and the benchmark's files are cached to the first-layer storage (a SSD shared among 16 nodes) prior to its execution. We have disabled all power-saving features, and other settings which could limit the performance of individual benchmarks. Furthermore, we submitted all runs to the batch system with the *--mpi max-proc-per-node=<num>* setting, to instruct the Fujitsu's MPI runtime to appropriately map the ranks and threads to the CMGs and cores of A64FX, i.e., *spread* and *close*, respectively. The exception to this rule is PolyBench, whose tests are pinned to one core, and SPEC which comes with its own execution environment. For SPEC, we followed a colleague's recommendation to further tune the large page settings via *XOS_MMM_L_PAGING_POLICY=demand:demand:demand* and *XOS_MMM_L_ARENA_LOCK_TYPE=0*, and specify *OMP_PROC_BIND=close*, but left these settings at default values for all other benchmarks.

While theoretically possible, we note that tuning all our 100+ benchmarks individually with the full range of compiler flags, runtime parameters, and manual code refactoring, etc., for optimal performance is outside the scope of this work, which seeks to identify or disproof the existence of a "silver bullet" compiler for the A64FX processor.

1. Compilation scripts & inputs available: gitlab.com/domke/a64fxCvC
2. Source code: github.com/RIKEN-RCCS/fs2020-tapp-kernels; We use earlier snapshot; Referencing them with Kernel 1...22 to avoid confusion.

2.4. Measurement Approach and Metrics

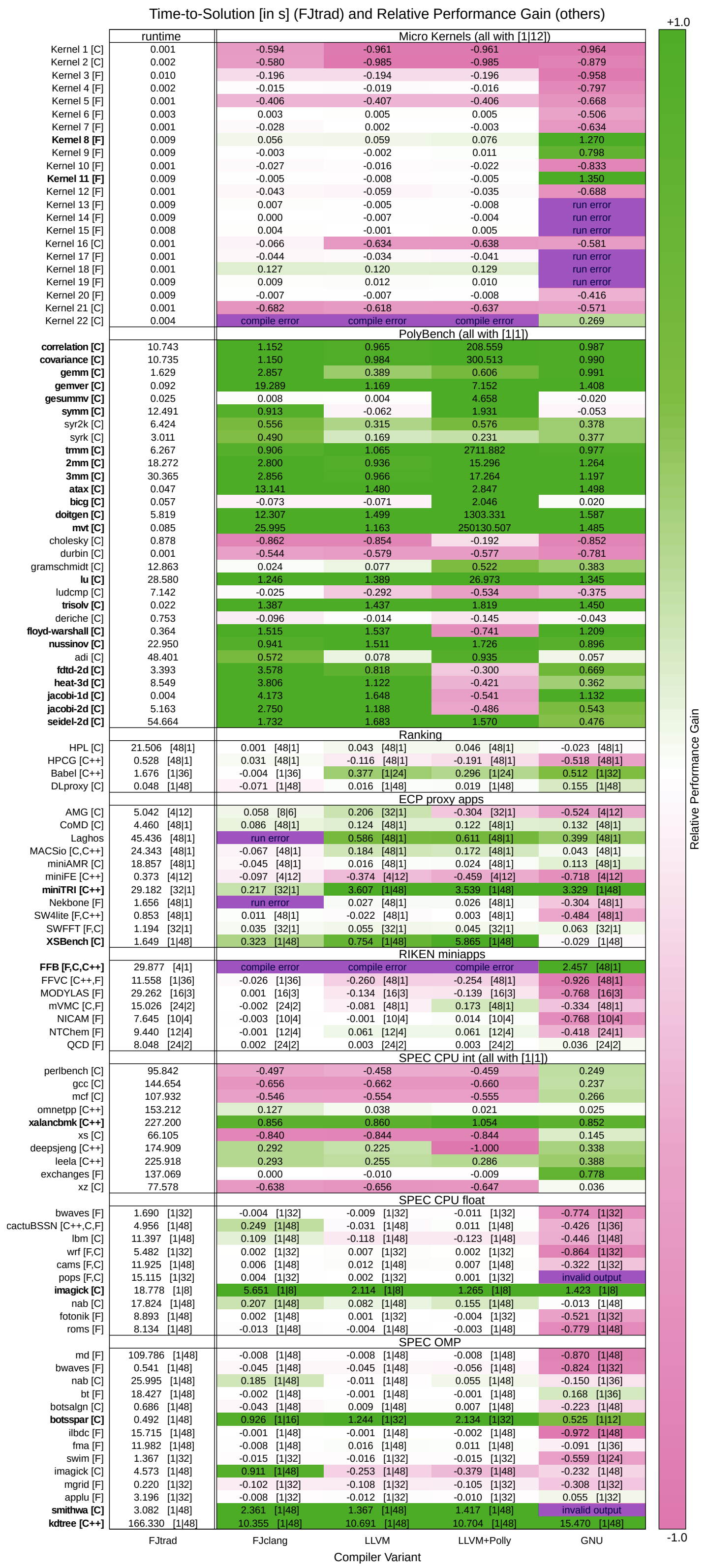
While Fujitsu's and RIKEN's recommendation for Fugaku and A64FX is 4 ranks (one per CMG) and 12 OpenMP threads per rank per node, this may not always be ideal, since some codes prefer or even require a power-of-2 for ranks/threads (e.g., *SWFFT*) or do not scale with number of threads (e.g., SPEC *imagick*'s sweet spot is 8 threads). Hence, we employ a exploration phase for each compiler and test various MPI and/or OMP combinations for all parallelized, strong-scaling benchmarks (except: weak-scaling *MiniAMR* and *XSBench*), using 3 trial runs each. The fastest time-to-solution determines the final MPI/OMP setting (individual per compiler) for the performance runs, which we run again 10 times. We manually instrumented all benchmarks to only report the time-to-solution of the region of interest, i.e., excluding any pre-/post-processing phases; except for SPEC CPU/OMP where we rely on SPEC's runtime reporting. Performing 10 runs should suffice, since we experience low run-to-run variability on A64FX. For example, AMG's coefficient of variation (CV) in runtime was below 0.114%, and we only see high variability in BabelStream with a CV of up to 22% which is still noticeably smaller than the gap between compilers.

3. Evaluation and Result Discussion

We report the fastest runtime across 10 *performance runs* (cf. Sec. 2.4), and relative comparisons to the recommended compiler, i.e., using Fujitsu's compiler suite in *trad* mode. Figure 2 is additionally color-coded with the relative performance gain (see [12]) of each compiler over the *FJtrad* baseline, with white for similar runtime and dark green indicating 2x speedup. Benchmarks exhibiting over 2x speedup are further signalized with a **bold** name. Furthermore, unsolvable compilation errors, runtime errors, and invalid runs are encoded in dark pink. Additionally, we added the best parallelization configuration, i.e., shown via [#MPI ranks | #OMP threads] for each compiler/benchmark combination, except for the micro, PolyBench, and SPEC CPUint for which it is universal and written in the header.

3.1. Micro Kernels and Polybench

For the Micro Kernels representing RIKEN's priority apps, we clearly see the results of the co-design efforts for Fugaku. Fujitsu's compiler in traditional mode outperforms all other compilers in nearly all test. Only the GNU compiler is able to noticeably beat *FJtrad* in 4 of the 22 tests, but also produces 6 executables which result in runtime errors. Assuming that we switch always to the best compiler option, then we could reduce the runtime by 17% on average, with a median of 0%, and peak of 2.4x improvement. However, for PolyBench the roles reverse, with *LLVM+Polly* showing the best results, followed by *FJclang* in some cases. Especially for *mvt* the polyhedral optimizations resulted in over 250.000x speedup. Choosing the best compiler over *FJtrad*



Time-to-Solution [in s] (FJtrad) and Relative Performance Gain (others)

	FJtrad	FJclang	LLVM	LLVM+Polly	GNU
	runtime	Micro Kernels (all with [1|12])			
Kernel 1 [C]	0.001	-0.594	-0.961	-0.961	-0.964
Kernel 2 [C]	0.002	-0.580	-0.985	-0.985	-0.879
Kernel 3 [F]	0.010	-0.196	-0.194	-0.196	-0.958
Kernel 4 [F]	0.002	-0.015	-0.019	-0.016	-0.797
Kernel 5 [F]	0.001	-0.406	-0.407	-0.406	-0.668
Kernel 6 [F]	0.003	0.003	0.005	0.005	-0.506
Kernel 7 [F]	0.001	-0.028	0.002	-0.003	-0.634
Kernel 8 [F]	0.009	0.056	0.059	0.076	1.270
Kernel 9 [F]	0.009	-0.003	-0.002	0.011	0.798
Kernel 10 [F]	0.001	-0.027	-0.016	-0.022	-0.833
Kernel 11 [F]	0.009	-0.005	-0.008	-0.005	1.350
Kernel 12 [F]	0.001	-0.043	-0.059	-0.035	-0.688
Kernel 13 [F]	0.009	0.007	-0.005	-0.008	run error
Kernel 14 [F]	0.009	0.000	-0.007	-0.004	run error
Kernel 15 [F]	0.008	0.004	-0.001	0.005	run error
Kernel 16 [C]	0.001	-0.066	-0.634	-0.638	-0.581
Kernel 17 [F]	0.001	-0.044	-0.034	-0.041	run error
Kernel 18 [F]	0.001	0.127	0.120	0.129	run error
Kernel 19 [F]	0.009	0.009	0.012	0.010	run error
Kernel 20 [F]	0.009	-0.007	-0.007	-0.008	-0.416
Kernel 21 [C]	0.001	-0.682	-0.618	-0.637	-0.571
Kernel 22 [C]	0.004	compile error	compile error	compile error	0.269
		PolyBench (all with [1|1])			
correlation [C]	10.743	1.152	0.965	208.559	0.987
covariance [C]	10.735	1.150	0.984	300.513	0.990
gemm [C]	1.629	2.857	0.389	0.606	0.991
gemver [C]	0.092	19.289	1.169	7.152	1.408
gesummv [C]	0.025	0.008	0.004	4.658	-0.020
symm [C]	12.491	0.913	-0.062	1.931	-0.053
syr2k [C]	6.424	0.556	0.315	0.576	0.378
syrk [C]	3.011	0.490	0.169	0.231	0.377
trmm [C]	6.267	0.906	1.065	2711.882	0.977
2mm [C]	18.272	2.800	0.936	15.296	1.264
3mm [C]	30.365	2.856	0.966	17.264	1.197
atax [C]	0.047	13.141	1.480	2.847	1.498
bicg [C]	0.057	-0.073	-0.071	2.046	0.020
doitgen [C]	5.819	12.307	1.499	1303.331	1.587
mvt [C]	0.085	25.995	1.163	250130.507	1.485
cholesky [C]	0.878	-0.862	-0.854	-0.192	-0.852
durbin [C]	0.001	-0.544	-0.579	-0.577	-0.781
gramschmidt [C]	12.863	0.024	0.077	0.522	0.383
lu [C]	28.580	1.246	1.389	26.973	1.345
ludcmp [C]	7.142	-0.025	-0.292	-0.534	-0.375
trisolv [C]	0.022	1.387	1.437	1.819	1.450
deriche [C]	0.753	-0.096	-0.014	-0.145	-0.043
floyd-warshall [C]	0.364	1.515	1.537	-0.741	1.209
nussinov [C]	22.950	0.941	1.511	1.726	0.896
adi [C]	48.401	0.572	0.078	0.935	0.057
fdtd-2d [C]	3.393	3.578	0.818	-0.300	0.669
heat-3d [C]	8.549	3.806	1.122	-0.421	0.362
jacobi-1d [C]	0.004	4.173	1.648	-0.541	1.132
jacobi-2d [C]	5.163	2.750	1.188	-0.486	0.543
seidel-2d [C]	54.664	1.732	1.683	1.570	0.476
		Ranking			
HPL [C]	21.506 [48|1]	0.001 [48|1]	0.043 [48|1]	0.046 [48|1]	-0.023 [48|1]
HPCG [C++]	0.528 [48|1]	0.031 [48|1]	-0.116 [48|1]	-0.191 [48|1]	-0.518 [48|1]
Babel [C++]	1.676 [1|36]	-0.004 [1|36]	0.377 [1|24]	0.296 [1|24]	0.512 [1|32]
DLproxy [C]	0.048 [1|48]	-0.071 [1|48]	0.016 [1|48]	0.019 [1|48]	0.155 [1|48]
		ECP proxy apps			
AMG [C]	5.042 [4|12]	0.058 [8|6]	0.206 [32|1]	-0.304 [32|1]	-0.524 [4|12]
CoMD [C]	4.460 [48|1]	0.086 [48|1]	0.124 [48|1]	0.122 [48|1]	0.132 [48|1]
Laghos	45.436 [48|1]	run error	0.586 [48|1]	0.611 [48|1]	0.399 [48|1]
MACSio [C,C++]	24.343 [48|1]	-0.067 [48|1]	0.184 [48|1]	0.172 [48|1]	0.043 [48|1]
miniAMR [C]	18.857 [48|1]	-0.045 [48|1]	0.016 [48|1]	0.024 [48|1]	0.113 [48|1]
miniFE [C++]	0.373 [4|12]	-0.097 [4|12]	-0.374 [4|12]	-0.459 [4|12]	-0.718 [4|12]
miniTRI [C++]	29.182 [32|1]	0.217 [32|1]	3.607 [1|48]	3.539 [1|48]	3.329 [1|48]
Nekbone [F]	1.656 [48|1]	run error	0.027 [48|1]	0.026 [48|1]	-0.304 [48|1]
SW4lite [F,C++]	0.853 [48|1]	0.011 [48|1]	-0.022 [48|1]	0.003 [48|1]	-0.484 [48|1]
SWFFT [F,C]	1.194 [32|1]	0.035 [32|1]	0.055 [32|1]	0.045 [32|1]	0.063 [32|1]
XSBench [C]	1.649 [1|48]	0.323 [1|48]	0.754 [1|48]	5.865 [1|48]	-0.029 [1|48]
		RIKEN miniapps			
FFB [F,C,C++]	29.877 [4|1]	compile error	compile error	compile error	2.457 [48|1]
FFVC [C++,F]	11.558 [1|36]	-0.026 [1|36]	-0.260 [48|1]	-0.254 [48|1]	-0.926 [48|1]
MODYLAS [F]	29.262 [16|3]	0.001 [16|3]	-0.134 [16|3]	-0.139 [16|3]	-0.768 [16|3]
mVMC [C,F]	15.026 [24|2]	-0.002 [24|2]	-0.081 [48|1]	0.173 [48|1]	-0.334 [48|1]
NICAM [F]	7.645 [10|4]	-0.003 [10|4]	-0.001 [10|4]	0.014 [10|4]	-0.768 [10|4]
NTChem [F]	9.440 [12|4]	-0.001 [12|4]	0.061 [12|4]	0.061 [12|4]	-0.418 [24|1]
QCD [F]	8.048 [24|2]	0.002 [24|2]	0.003 [24|2]	0.003 [24|2]	0.036 [24|2]
		SPEC CPU int (all with [1|1])			
perlbench [C]	95.842	-0.497	-0.458	-0.459	0.249
gcc [C]	144.654	-0.656	-0.662	-0.660	0.237
mcf [C]	107.932	-0.546	-0.554	-0.555	0.266
omnetpp [C++]	153.212	0.127	0.038	0.021	0.025
xalancbmk [C++]	227.200	0.856	0.860	1.054	0.852
xs [C]	66.105	-0.840	-0.844	-0.844	0.145
deepsjeng [C++]	174.909	0.292	0.225	-1.000	0.338
leela [C++]	225.918	0.293	0.255	0.286	0.388
exchanges [F]	137.069	0.000	-0.010	-0.009	0.778
xz [C]	77.578	-0.638	-0.656	-0.647	0.036
		SPEC CPU float			
bwaves [F]	1.690 [1|32]	-0.004 [1|32]	-0.009 [1|32]	-0.011 [1|32]	-0.774 [1|32]
cactuBSSN [C++,C,F]	4.956 [1|48]	0.249 [1|48]	-0.031 [1|48]	0.011 [1|48]	-0.426 [1|36]
lbm [C]	11.397 [1|48]	0.109 [1|48]	-0.118 [1|48]	-0.123 [1|48]	-0.446 [1|48]
wrf [F,C]	5.482 [1|32]	0.002 [1|32]	0.007 [1|32]	0.002 [1|32]	-0.864 [1|32]
cams [F,C]	11.925 [1|48]	0.006 [1|48]	0.012 [1|48]	0.007 [1|48]	-0.322 [1|32]
pops [F,C]	15.115 [1|32]	0.004 [1|32]	0.002 [1|32]	0.001 [1|32]	invalid output
imagick [C]	18.778 [1|8]	5.651 [1|8]	2.114 [1|8]	1.265 [1|8]	1.423 [1|8]
nab [C]	17.824 [1|48]	0.207 [1|48]	0.082 [1|48]	0.155 [1|48]	-0.013 [1|48]
fotonik [F]	8.893 [1|48]	0.002 [1|48]	0.001 [1|32]	-0.004 [1|32]	-0.521 [1|32]
roms [F]	8.134 [1|48]	-0.013 [1|48]	-0.004 [1|48]	-0.003 [1|48]	-0.779 [1|48]
		SPEC OMP			
md [F]	109.786 [1|48]	-0.008 [1|48]	-0.008 [1|48]	-0.008 [1|48]	-0.870 [1|48]
bwaves [F]	0.541 [1|48]	-0.045 [1|48]	-0.045 [1|48]	-0.056 [1|48]	-0.824 [1|32]
nab [C]	25.995 [1|48]	0.185 [1|48]	-0.011 [1|48]	0.055 [1|48]	-0.150 [1|36]
bt [F]	18.427 [1|48]	-0.002 [1|48]	-0.001 [1|48]	-0.001 [1|48]	0.168 [1|36]
botsalgn [C]	0.686 [1|48]	-0.043 [1|48]	0.009 [1|48]	0.007 [1|48]	-0.223 [1|48]
botsspar [C]	0.492 [1|48]	0.926 [1|16]	1.244 [1|32]	2.134 [1|32]	0.525 [1|12]
ilbdc [F]	15.715 [1|48]	-0.001 [1|48]	-0.001 [1|48]	-0.002 [1|48]	-0.972 [1|48]
fma [F]	11.982 [1|48]	-0.008 [1|48]	0.016 [1|48]	0.011 [1|48]	-0.091 [1|36]
swim [F]	1.367 [1|32]	-0.015 [1|32]	-0.016 [1|32]	-0.015 [1|32]	-0.559 [1|24]
imagick [C]	4.573 [1|48]	0.911 [1|48]	-0.253 [1|48]	-0.379 [1|48]	-0.232 [1|48]
mgrid [F]	0.220 [1|32]	-0.102 [1|32]	-0.108 [1|32]	-0.105 [1|32]	-0.308 [1|32]
applu [F]	3.196 [1|32]	-0.008 [1|32]	-0.012 [1|32]	-0.010 [1|32]	0.055 [1|32]
smithwa [C]	3.082 [1|48]	2.361 [1|48]	1.367 [1|48]	1.417 [1|48]	invalid output
kdtree [C++]	166.330 [1|48]	10.355 [1|48]	10.691 [1|48]	10.704 [1|48]	15.470 [1|48]

Figure 2. Absolute time-to-solution (numbers; *FJtrad* column) and relative gain over *FJtrad* (numbers; color coding; non-*FJtrad* columns) for all benchmarks; Invalid entries explain (e.g. "compiler error", see Kernel 22); Programming language indicated after benchmark name; Benchmark names with >2x speedup in bold; Number of MPI rank & OMP threads in brackets

results in a median speedup of 3.8x. It remains to be seen if *polly*'s gains translate to "real" scientific codes hereafter.

3.2. TOP500, ECP, and Fiber Proxys

HPL gains a minor advantage ($\approx$5%) by using LLVM over Fujitsu's compilers, despite that most of the calculations are performed within SSL2, which holds true for the convolution kernel of the deep learning proxy, too. BabelStream shows the largest gain from switching to LLVM or GNU with up to 51% lower runtime. With a few exceptions, like FFB and mVMC, Fujitsu dominates the other compilers on Fiber mini-apps, which is consistent with the Micro Kernel results shown earlier. For ECP proxy-apps the conclusions reverses, and the user would be advised to switch to LLVM or GNU in almost all cases. Such a change leads to an average speedup of 1.65x (median 1.09x). The 6.7x speedup for XSBench is salient, because it also demonstrates that *polly* can have an impact on real workloads.

3.3. SPEC CPU and OMP

The SPEC measurements reveal multiple interesting insights. Firstly, *FJtrad* outperforms any Clang-based alternative on A64FX on integer-intensive codes; however, the GNU compiler almost universally beats *FJtrad* at the same single-threaded workloads. We speculate that this advantage is partially a result of GNU's prevalence in the embedded space where many of the Arm CPUs have no floating-point units, and also a result of Arm's continued investments into the open-source GNU compilers [13]. By contrast, for multi-threaded and floating-point-based SPEC CPU workloads, as well as SPEC OMP, the GNU compiler is currently the worst choice on A64FX. Furthermore, many of the applications are written in Fortran, and hence there is little benefit (apart from maybe link-time-optimizations) for switching to LLVM. For C/C++ applications on the other hand, LLVM-based compilers (incl. *FJclang*), and *GNU* in a few cases, can yield a runtime benefit over *FJtrad*. We see speedup as high as 16.5x in SPEC OMP simply by switching compilers (e.g., for kdtree), with an average improvement of 49% in SPEC CPU and 2.5x speedup in SPEC OMP. The median runtime improvement from choosing the best compiler across both SPEC suites is 14%.

Overall, across all 108 benchmarks and realistic workloads, we see that a median runtime improvement of 16% is possible by selecting an appropriate compiler, without any changes to the source code or other tuning methods.

4. Related Work

Various publications and reports of A64FX evaluations have been released recently. For example, [14] tested LLVM and its SVE code generation capabilities for DCA++, and [15] compared a limited set of applications with LLVM, GNU, ARM, and Cray compilers and focused on SVE and multi-node scaling. Similarly, [16] investigated OpenMP-scaling of 3 proxy apps on A64FX while comparing 5 compilers, and [17] measured nearly a dozen proxy apps (different from ours) on ARM and x86 for multiple compilers, but lacked LLVM. In [18], the authors analyzed the achievable bandwidth for a set of memory-bandwidth-bound kernels using GNU, and [19] reported a 2x performance advantage of GNU compilers for the E3SM climate code. Lastly, the studies [4], [20], [21] compare A64FX with Fujitsu's traditional mode to ARM-based ThunderX2 and Intel Xeon CPUs using various priority apps and proxies. All these studies are complementary to our comprehensive compiler comparison for a wide variety of workloads, since they use different apps, compilers, or evaluation approaches.

5. Conclusion

In conclusion, we demonstrate a clear benefit from exploring alternative compilers for the A64FX CPU as a valid first-order tuning method before investing a considerable amount of effort into testing "exotic" compiler flags, environment variables, and performing manual code refactoring. Especially, the performance discrepancy for PolyBench, which we show in Figure 1, was solved by switching from the recommended *FJtrad* to LLVM 12 compiler, but the *polly* optimizations seem rarely applicable or beneficial outside this benchmark set. To revisit our initial question, we could not identify a "silver bullet" compiler for A64FX, but our measurements give indications for which compilers work well in which situations, i.e., Fujitsu for Fortran codes, GNU for integer-intensive apps, and any clang-based compilers for C/C++. Furthermore, we noticed that for "legacy" applications, the recommended usage model of 4 ranks and 12 threads per A64FX node results in suboptimal time-to-solution more often than not, and that the Arm software ecosystem for HPC is not as mature as for x86, yet.

Our work is just one among many similar, early explorations of the newly introduced Arm-base CPU for high-performance computing, but it gives reason to believe that potential performance deficiencies, when directly compared against x86 for the same applications, are most likely the results of immature compilers. Hence, our recommendation to administrators and users of A64FX-based supercomputers is to install and test as many different, available compilers as possible to extract the true performance potential from the A64FX CPU. Similarly, it could be worthwhile to revisit how various system libraries, such as for searching, sorting, routing, or MPI libraries, etc., and other OS packages are (pre-)compiled for the A64FX processor.

Acknowledgments

The authors would like to thank various colleagues from RIKEN R-CCS, in particular members of the NG-HPA, HPAI, and Operations team, for their feedback, discussions, and assistance with software installation and application debugging. This work was supported by the Japan Society for the Promotion of Science KAKENHI Grant Number JP19H04119; and by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero, "Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC?" in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: Association for Computing Machinery, 2013, event-place: Denver, Colorado.

[2] N. Rajovic, A. Rico, F. Mantovani, D. Ruiz, J. O. Vilarrubi, C. Gomez, L. Backes, D. Nieto, H. Servat, X. Martorell, J. Labarta, E. Ayguade, C. Adeniyi-Jones, S. Derradji, H. Gloaguen, P. Lanucara, N. Sanna, J.-F. Mehaut, K. Pouget, B. Videau, E. Boyer, M. Allalen, A. Auweter, D. Brayford, D. Tafani, V. Weinberg, D. Brömmel, R. Halver, J. H. Meinke, R. Beivide, M. Benito, E. Vallejo, M. Valero, and A. Ramirez, "The Mont-Blanc Prototype: An Alternative Approach for HPC Systems," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 444–455.

[3] A. Rico, J. A. Joao, C. Adeniyi-Jones, and E. Van Hensbergen, "ARM HPC Ecosystem and the Reemergence of Vectors: Invited Paper," in *Proceedings of the Computing Frontiers Conference*, ser. CF'17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 329–334.

[4] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu, "Co-Design for A64FX Manycore Processor and "Fugaku"," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020, event-place: Atlanta, Georgia.

[5] Fujitsu Limited, "FUJITSU Processor A64FX." [Online]. Available: https://www.fujitsu.com/downloads/SUPER/a64fx/a64fx_datasheet_en.pdf

[6] J. Domke, K. Matsumura, M. Wahib, H. Zhang, K. Yashima, T. Tsuchikawa, Y. Tsuji, A. Podobas, and S. Matsuoka, "Double-precision FPUs in High-Performance Computing: an Embarrassment of Riches?" in *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, Rio de Janeiro, Brazil, May 2019.

[7] L.-N. Pouchet and M. Taylor, "PolyBench/C 4.2.1 (beta)," May 2016. [Online]. Available: https://sourceforge.net/projects/polybench/

[8] J. Dongarra, "The LINPACK Benchmark: An Explanation," in *Proceedings of the 1st International Conference on Supercomputing*. London, UK, UK: Springer-Verlag, 1988, pp. 456–474. [Online]. Available: http://dl.acm.org/citation.cfm?id=647970.742568

[9] J. Dongarra, M. Heroux, and P. Luszczek, "HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems," University of Tennessee, Tech. Rep. ut-eecs-15-736, Jan. 2015. [Online]. Available: https://library.eecs.utk.edu/pub/594

[10] 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 *High Performance Computing*, M. Taufer, B. Mohr, and J. M. Kunkel, Eds. Cham: Springer International Publishing, 2016, pp. 489–507.

[11] J. Domke, E. Vatai, A. Drozd, C. Peng, Y. Oyama, L. Zhang, S. Salaria, D. Mukunoki, A. Podobas, M. Wahib, and S. Matsuoka, "Matrix Engines for High Performance Computing: A Paragon of Performance or Grasping at Straws?" in *2021 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, Oregon, USA, May 17-21, 2021*, Portland, Oregon, USA, May 2021, p. 10.

[12] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, Austin, TX, USA, Nov. 2015, pp. 73:1–73:12.

[13] T. Christina, "Blog Post: Significant performance improvements in GCC 10 through Vectorization and In-lining," May 2020. [Online]. Available: https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/gcc-10-better-and-faster

[14] J. Huber, W. Wei, G. Georgakoudis, J. Doerfert, and O. R. Hernandez, "A Case Study of LLVM-Based Analysis for Optimizing SIMD Code Generation," Tech. Rep. arXiv:2106.14332v1, 2021. [Online]. Available: https://arxiv.org/pdf/2106.14332.pdf

[15] A. Burford, A. C. Calder, D. Carlson, B. M. Chapman, F. CoSKun, T. Curtis, C. Feldman, R. J. Harrison, Y. Kang, B. Michalow-Icz, E. Raut, E. Siegmann, D. G. Wood, R. L. DeLeon, M. Jones, N. A. Simakov, J. P. White, and D. Oryspayev, "Ookami: Deployment and Initial Experiences," Tech. Rep. arXiv:2106.08987v1, 2021. [Online]. Available: https://arxiv.org/pdf/2106.08987.pdf

[16] B. Michalowicz, E. Raut, Y. Kang, T. Curtis, B. M. Chapman, and D. Oryspayev, "Comparing the behavior of OpenMP Implementations with various Applications on two different Fujitsu A64FX platforms," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[17] A. Poenaru, "An Evaluation of the Fujitsu A64FX for HPC Applications," Presentation in AHUG ISC 21 Workshop, Jul. 2021. [Online]. Available: https://a-hug.org/isc-2021-event/

[18] C. Alappat, J. Laukemann, T. Gruber, G. Hager, G. Wellein, N. Meyer, and T. Wettig, "Performance Modeling of Streaming Kernels and Sparse Matrix-Vector Multiplication on A64FX," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020, pp. 1–7.

[19] S. Sreepathi, O. Guba, and M. Taylor, "E3SM Pathfinding on Fugaku," May 2021. [Online]. Available: https://e3sm.org/e3sm-pathfinding-on-fugaku/

[20] A. Jackson, M. Weiland, N. Brown, A. Turner, and M. Parsons, "Investigating Applications on the A64FX," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. Los Alamitos, CA, USA: IEEE Computer Society, Sep. 2020, pp. 549–558.

[21] T. Odajima, Y. Kodama, M. Tsuji, M. Matsuda, Y. Maruyama, and M. Sato, "Preliminary Performance Evaluation of the Fujitsu A64FX Using HPC Applications," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 523–530.