A64FX performance: experience on Ookami

Md Abdullah Shahneous Bari, Barbara Chapman, Anthony Curtis, Robert J. Harrison, Eva Siegmann
Institute for Advanced Computational Science, Stony Brook University
Stony Brook, NY, USA
{MdAbdullah.ShahneousBari, Barbara.Chapman, Anthony.Curtis, Robert.Harrison, Eva.Siegmann}@stonybrook.edu

Nikolay A. Simakov, Matthew D. Jones
Center for Computational Research, University at Buffalo
Buffalo, NY, USA
nikolays@buffalo.edu, jonesm@buffalo.edu

***Abstract*—We examine the performance of scientific and engineering kernels on the Fujitsu A64FX processor, both out-of-the-box using various toolchains and with processor-specific optimizations. While nearly all applications port with little to no modification, significant performance variation is observed between the multiple tool chains. This variation depends heavily upon characteristics of the application (most notably its use of mathematical functions) and is also constrained by the most performant toolchains having limited support for recent language standards. As expected, high performance demands that a kernel is vectorized, multi-threaded, and localizes memory references. Detailed optimizations, including use of intrinsics, are also examined to understand performance gaps and what is necessary to attain peak performance. This article employs the Ookami computer technology testbed funded by the American National Science Foundation. The system provides researchers worldwide with access to 176 Fujitsu A64FX compute nodes as well as other state-of the-art technology.**

***Index Terms*—high-performance computing**

I. Introduction

The A64FX processor deployed in Fugaku [1], currently the world's currently fastest supercomputer, was developed by Fujitsu in collaboration with the RIKΞN Center for Computational Science. Our interest in A64FX is motivated by the unique success of Fugaku in leading five major benchmarks: TOP500 [2], Graph500 [3], HPCG-benchmark [4], HPL-AI [5], Green [6]. The Fujitsu A64FX 48-core, 64-bit ARM processor is the first to deploy the Scalable Vector Extension (SVE) SIMD-vector instruction set. Its vector performance is matched with 32 GB of high-bandwidth memory (1 TB/s) and promises to retain familiar and successful programming models while achieving high performance for a wide range of applications. The Ookami testbed [7] at Stony Brook University enables exploration of the performance and relevance of A64FX and related technologies to computational science and engineering in multiple settings outside the leadership scale of Fugaku. Researchers worldwide can request access and, if approved, use the system free of charge.

In previous work by ourselves [8], [9] and others [10], the platform has already demonstrated excellent portability with most applications compiling and running out of the box, after addressing the mundane issues of library names and compiler flags. Software that is both vectorized and threaded, or employs optimized library routines, can immediately achieve competitive performance [8]. However, for some applications the path to performance is longer. In this paper, we explore how the choice of toolchain impacts performance through the ability to vectorize, the underlying math libraries, and aspects of the OpenMP parallel runtime. We start by examining the performance of vector kernels, explore the detailed implementation of the exponential function, examine the NAS parallel benchmarks that emphasize the performance of compiled code, and conclude with the HPC Challenge benchmarks that emphasize the performance of library routines.

II. The Ookami System

The Ookami HPE Apollo 80 system funded by the National Science Foundation (NSF) under grant OAC 1927880 provides researchers with state-of-the-art computing technology. It includes 176 A64FX compute nodes running at 1.8GHz, each with 32GB high-bandwidth memory, 48 cores and a 512GB SSD. The operating system is standard Linux, CentOS 8. A high-performance Lustre filesystem provides 0.8TB of storage served by a Cray ClusterStor E1000. The HDR 200 GB/s Infiniband network is configured as a full fat tree. The system's two login nodes are dual-socket ThunderX2 with 256 GB memory. The ARM-based login nodes eliminate most cross-compilation issues, and their very high scalar performance (64 cores or 256 threads at 2.3GHz) accelerates the development cycle. Other nodes available on Ookami are a dual socket AMD Milan (64 cores) with 512GB memory, an Intel Skylake (32 cores) with 192 GB memory including two Nvidia V100 GPUs. This enables users to directly compare portability and performance across all current major architectures (Intel x86, AMD x86, ARM+NEON, ARM+SVE, and NVIDIA GPU).

The A64FX-700 series processor used in Ookami has 48 cores arranged in four groups of 12, termed core memory groups (CMG) that are fully connected to each other. Each CMG forms a non-uniform memory access (NUMA) region with 8 Gbyte of on-package, stacked, high-bandwidth memory (256 Gbyte/s). Each core has 64KB L1 cache, and there is an 8 MB L2 cache shared between the cores within a CMG. Peak execution rate is achieved by issuing 2x512-bit wide SIMD FMA + predicate op + 4x ALU instructions per cycle.

Theoretical peak double precision speed is computed as 1.8 GHz×2 FMA/cycle×2 FLOPs/FMA×8 64-bit words/vector = 57.6 GFLOP/s/core.

A. The Software Environment

Ookami runs on standard Linux, specifically CentOS 8. The Bright Cluster Manager provides the operating system, and workload manager from a unified interface. Job scheduling is done using SLURM (Simple Linux Utility for Resource Management, version 19.05.7).

III. Compiler and Math Library Vectorization

In teaching computer performance and software optimization to physical scientists [11], team members have employed the example of a simple Monte Carlo algorithm to compute an integral over the exponential function. The naive loop body is just 3 lines of code.

```
x = 23.0*rand(); sum=0.0; // initialize
while (1) {
   // sample and accept/reject new point
   xnew = 23.0*rand();
   if (exp(-xnew) > exp(-x)*rand()) x = xnew;
   sum += x;  // accumulate statistics
}
```

However, on a CPU, in addition to being unvectorized and unthreaded, this loop is completely serial — it exposes nearly the full latency of most of the operations in the loop. In contrast, on a GPU, nearly the same code is fully parallel due to the implicit data parallelism of CUDA and the fully-predicated instruction set of the GPU. Hence, comparison of the performance of this tiny kernel reveals over a 500-fold performance advantage for GPUs over CPUs. While being a fair comparison of what is possible with minimal effort, this is not a valid comparison of the performance of the underlying hardware. Remedying the gap involves introducing an additional loop over independent samples, splitting that loop to serve both thread and vector parallelism, interchanging loops, and promoting scalars to vectors. However, until about a decade ago, mainstream compilers could not vectorize the resulting inner loop due to (1) the if-test, (2) the exponential function, and (3) the random number generator. Thus, additional required optimizations were loop splitting, and directly invoking vectorized math library operations. On modern X86 (AVX and beyond), all major tool chains can vectorize the if-test and exponential functions (for GCC assuming a recent version of glibc), but a manual call to a vectorized random number generator is still necessary. A Cray compiler from the 1980s had no such need.

To understand the situation On ARM+SVE, we developed a small test suite to explore the ability of toolchains to vectorize code and the resulting performance. The small loops also permit examining and understanding the generated code. The test loops are

- simple: `y[i] = 2*x[i] + 3*x[i]*x[i]`
- predicate: `if (x[i]>0) y[i] = x[i]`
- math functions: reciprocal, square root, exponential, sine, power
- gather: `y[i] = x[index[i]]`
- scatter: `y[index[i]] = x[i]`
- short gather and scatter (see text)

. The sizes of working vectors were adjusted to collectively fill the L1 cache. The simple loop is representative of many scientific or engineering applications that perform basic arithmetic operations on vectors, matrices, and grids. The predicate is motivated by the Monte Carlo application above, but occurs widely. Many scientific kernels (e.g., evaluation of bonded interactions in molecular dynamics, basis functions in quantum chemistry, equations of state in astrophysics, etc.) can be dominated by evaluation of mathematical functions. Gather and scatter operations occur in many settings (interpolation tables, splitting/merging vectors to avoid divergent execution paths, unstructured meshes, etc.). For the gather/scatter test, the index vector was constructed as a random permutation of the entire index space (i.e., the integers $i = 0, 1, \ldots, n-1$, where n is the vector length). In contrast, the index vector for the short gather/scatter tests was constructed by randomly permuting within 128 byte windows (i.e., 16 doubles). This is to explore the performance of the A64FX implementation of the SVE non-contiguous load that has special optimizations within a 128-byte window, similar to coalescing of memory accesses on a GPU by the threads in a (half-) warp.

The Intel, Fujitsu, Cray and ARM compilers vectorized all loops, whereas the GNU compiler did not vectorize `exp`, `sin`, and `pow` (compiler flags are in Table I). The Intel compiled versions were only run on Intel Skylake (Xeon Gold 6140, 2.1GHz base, 3.7GHz boost), whereas all others used an A64FX on Ookami (1.8GHz fixed). For these single-core tests, the clock speed ratio leads to an expected circa 2x ratio of runtime between A64FX and Skylake unless other architectural or software features intervene. Figures 1 and 2 show the run-times relative to Skylake for the different loops and compilers — the rationale for this choice being that the Intel compiler and its associated math library are sufficiently mature and tuned as to be regarded as optimal for practical purposes, whereas the ARM+SVE stack is clearly less mature.

On A64FX, the Fujitsu toolchain delivers the highest performance for all loops, followed by Cray, and ARM/GNU. For the simple loops, the ARM and GNU compilers are fairly competitive, but their performance on the ”simple” and predicate loops are up to 2 times slower. However, we note on similar kernels, the GNU compiler can sometimes outperform the Cray compiler. Relative to Skylake, we note that the Fujitsu tool chain performance hovers at the factor of 2 expected from the ratio of the clock speeds, except for the predicate operation that is 3-fold slower than Skylake and the short gather that is only circa 1.5-fold slower. The A64FX Microarchitecture Manual [12] indicates that if loads of pairs of elements of a gather operation fit within an aligned 128-byte window, then they are not split, resulting in a 2-fold speed up. No such acceleration is indicated for scatter operations, however, we note that the short scatter test localizes pairs of 128-byte windows within a single 256 byte cache line, whereas the cache line is only 64 bytes on Skylake.

TABLE I
COMPILER FLAGS USED IN LOOP VECTORIZATION TESTS.

Compiler	Version	Flags
Fujitsu	1.0.20	-Kfast -KSVE -Koptmsg=2
Arm	21	–std=c++17 -Ofast -ffp-contract=fastv -ffast-math -Wall -Rpass=loop-vectorize -march=armv8.2-a+sve -mcpu=a64fx -armpl -fopenmp
Cray	10.0.2	-O3 -h aggress,flex_mp= tolerant,msgs,negmsgs,vector3,omp
GNU	11.1.0	-Ofast -ffast-math -Wall -mtune=a64fx -mcpu=a64fx -march=armv8.2-a+sve -fopt-info-vec -fopt-info-vec-missed -fopenmp
Intel	19.1.2.254	-xHOST -O3 -ipo -no-prec-div -fp-model fast=2 -qopt-report=5 -qopt-report-phase=vec -mkl=sequential -qopt-zmm-usage=high -qopenmp

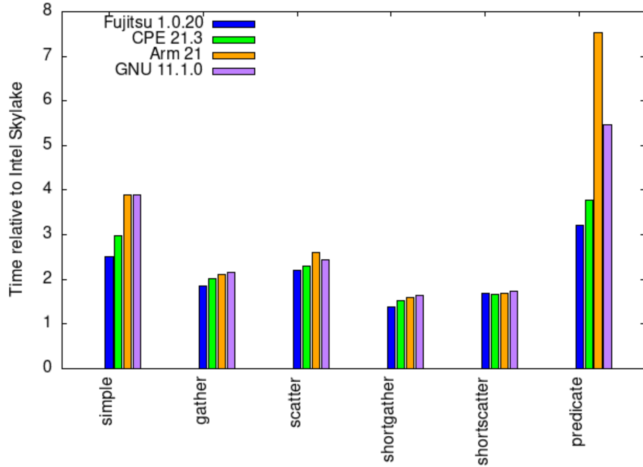

Fig. 1. Runtime on A64FX of simple vector loops compiled with different compilers relative to Intel compiler on Skylake.

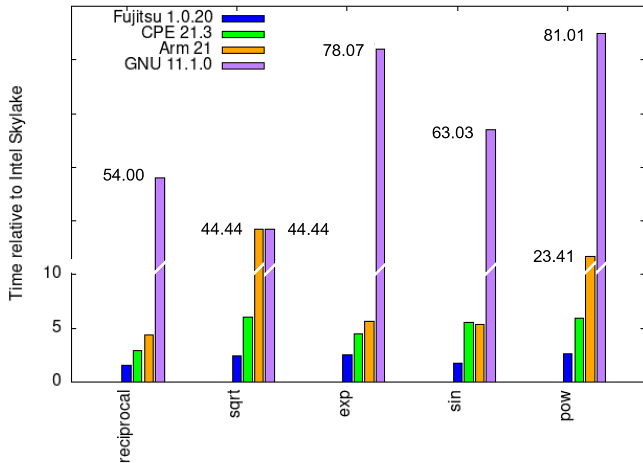

Fig. 2. Runtime on A64FX of vectorized math functions compiled with different compilers relative to Intel compiler on Skylake.

For the math functions, the Fujitsu tool chain is consistently close to the anticipated 2-fold ratio relative to Skylake, indicating that the A64FX can deliver cycle-level equivalent throughput on these hand-coded kernels. The Cray math library is fairly consistently another factor of 1.5-2 slower, with the AMD library slightly slower still except for 10x slower on `pow` and 20x on square root. The latter is due to both the AMD and GNU compilers selecting the SVE `FSQRT` instruction that on A64FX is blocking with a 134 cycle latency for a 512-bit vector. The Cray and Fujitsu compilers instead employ a Newton algorithm. The previous ARM compiler version 20 also made a similar bad choice for reciprocal (as do the current GNU compilers), leading us to hope square root will be fixed in an upcoming release. It is important to note that the both the GNU and AMD compilers report fully vectorizing the reciprocal and square root loops even though the performance could be very far from anticipated. Some components of the AMD vector math library apparently employ `Sleef` [13]. Finally, we note that a complete evaluation of math library performance must include accuracy, which will be the topic of another paper.

The state of the GNU vector math library on ARM+SVE deserves special discussion. Not only is there no vector math library within `glibc` for ARM+SVE, there presently appears to be no activity to develop one, and so we must anticipate this situation persisting for possibly years. Thus, unless an application computes primarily with floating-point multiplication and addition (which fortunately includes most linear algebra, finite-difference stencils, and FFT), or integer operations, the GNU toolchain must be avoided for high-performance kernels. This is a very unfortunate state of affairs since this toolchain is the *de facto* standard for Linux platforms and is the first that most new users on Ookami attempt to use.

IV. EVALUATION OF THE EXPONENTIAL FUNCTION

This section explores the gap in math library performance. The serial GNU implementation of the exponential function on A64FX takes nearly 32 cycles per evaluation. The vectorized ARM, Cray, and Fujitsu compilers take 6, 4.2, and 2.1 cycles, respectively, and the Intel compiler on Skylake takes 1.6 cycles.

A standard approach to computing $\exp x$ starts by finding integer m and value r such that $|r| < \frac{1}{2}\log 2$ and $x = m\log 2 + r$, leading to $\exp x = 2^m \exp r$. Exponentiating r can be done using a series expansion, with 13 terms being required to obtain the required accuracy in double precision arithmetic. Multiplication by 2^m is accomplished efficiently by adding m to the binary exponent. Unless extended precision is used or some refinement is performed, the last bit(s) will not

be correctly rounded. An error of between 1 and 4 ulps (units in the last place) is common in vectorized libraries, whereas the slow serial libraries typically guarantee correct rounding.

The SVE instruction `FEPXA` accelerates this process by reducing the number of terms in the series expansion to 5 by reducing the range of r by a factor of 64. We start by writing $x = (m + i/64)\log 2 + r$, but now with integer m, $0 \le i < 64$, and $|r| < \frac{1}{128}\log 2$. Then, $\exp x = 2^{m+i/64}\exp r$. `FEPXA` computes $2^{m+i/64}$, taking 17 bits as input (i in the lower 6 bits and $m + 1023$ in the upper 11).

There are 15 floating-point instructions in the loop body, with more instructions necessary for incrementing the loop counter and pointers, and for testing and branching. The processor issues all of these in about 16 cycles — with 8 elements per vector this gives a cost of 2 cycles per element (in more detail, using the SVE vector-length agnostic loop structure we obtain 2.2 cycles/element and 2.0 cycles/element with a fixed-width register). Unrolling once decreased this to 1.9 cycles/element. Empirically, the Estrin form for the polynomial that reveals more parallelism at the expense of more multiplications is slightly faster than the Horner form.

This is not a production-quality implementation — it has not been tested at the edges of permissible input values and some additional mask manipulation is necessary to ensure out of range large positive values are evaluated to be either NaN or infinity. Limited testing suggests that it yields about 6 ulp precision, which is good enough for many applications, but better is possible without compromising speed too much (an estimated 0.25 additional cycles/element) by correcting the last FMA operation. This would make the performance of this implementation comparable with that of Fujitsu.

In light of this experience, we hypothesize that the main performance gap for the non-Fujitsu math libraries is porting from other platforms standard function approximation algorithms that do not take full advantage of the features of the SVE instruction set, as well as not having the detailed architecture knowledge of the Fujitsu team.

V. NAS Parallel Benchmarks (NPB)

The NAS Parallel Benchmarks (NPB) [21] are well-known problems for measuring the performance of parallel computers and parallelization tools. Most of the benchmarks are derived from computational fluid dynamics (CFD) codes and are widely recognized as a good indicator of parallel computer performance.

We used six applications from the C implementation of the NAS Parallel Benchmark (NPB) suite [22] to evaluate the performance of different compilers in the Ookami system. BT, LU, and SP are pseudo applications, while the other three, CG, EP, and UA, are based on computational kernels. The benchmark applications can be run with six different datasets; S, A, B, C, D, and E. S is the smallest, and E is the largest data set. We used dataset C for our experimentation. Here is a brief overview of these applications:

BT: A simulated CFD computational kernel that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y, and z dimensions. The resulting systems are Block-Tridiagonal of 5×5 blocks and are solved sequentially along each dimension. This application shows good load balancing behavior. We used dataset size C (($162\times162\times162$ grid size, and 200 iterations)) for experimentation.

CG: Uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, and unstructured matrix. It tests unstructured vector computations and communications. It has a large amount of cache misses due to its usage of a matrix with randomly generated locations of entries. Dataset C employs 150000 rows, 15 non-zeros, and 75 iterations.

EP: An Embarrassingly Parallel Benchmark. It generates pairs of Gaussian random deviates according to a specific scheme. The goal of this benchmark is to establish a reference point for platforms' peak performance. Dataset C employs 2^{32} pairs of random numbers.

LU: Solves a 3D seven-block-diagonal system using lower-upper triangular systems solution. This application works with regular sparse matrices, and it uses symmetric successive over relaxation (SSOR) operations. Dataset size C employs $162\times162\times162$ grid size, and 250 iterations.

SP: A simulated CFD computational kernel that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y, and z dimensions. The resulting system has Scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension. It shows good load balancing behavior but poor cache behavior. Dataset C employs $162\times162\times162$ grid size, and 400 iterations.

UA: Provides the solution of a stylized heat transfer problem in a cubic domain, discretized on an adaptively refined, and unstructured mesh. The benchmark features irregular, dynamic memory accesses. Dataset C employs 33500 elements, 1262100 mortar points, 8 levels of refinements, and 200 iterations.

We designed several experiments to test the A64FX system performance and the performance of available compilers (ARM, Cray, Fujitsu, GNU). Since the performance of the A64FX system significantly depends on vectorization, the experiments are designed to test the auto-vectorization capabilities of the compilers. We also compare the scalability of these applications across multiple threads in the A64FX system. Finally, we compare how the single-node performance of A64FX compares to a traditional x86 system (Intel Skylake with 36 cores).

A. Single node performance of different compilers

1) Single Core: We used the serial version of the NPB to test the single-core performance of these benchmarks across different compilers in A64FX and compare their performance to the single-core performance of an Intel x86 system (Intel Skylake) using Intel compiler. We used the same optimization

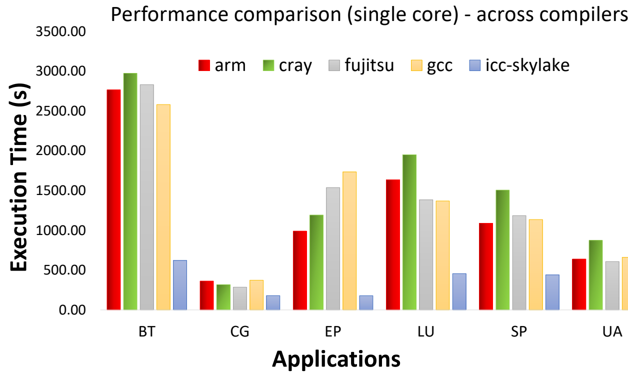

Fig. 3. Single core runtime of NAS parallel benchmark applications compiled with different compilers.

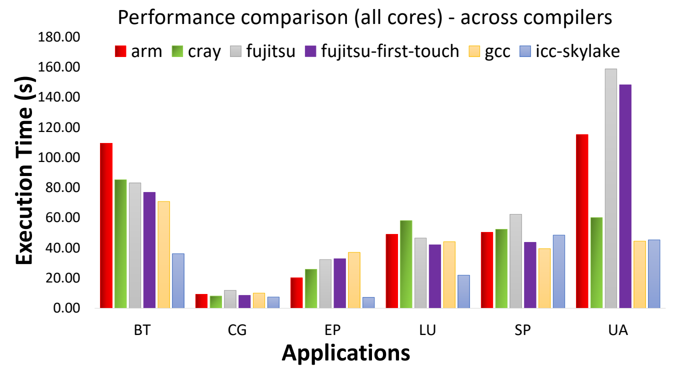

Fig. 4. Runtime of NAS parallel benchmark applications compiled with different compilers running with all available cores.

flags described in Table I except the OpenMP flags to compile the programs. Figure 3 shows the result. These results show the comparison of the optimization and vectorization performance of different compilers (ARM vs. Cray vs. Fujitsu vs. GCC) in the A64FX platform and the comparison of platform performance (A64FX vs. x86). In A64FX, gcc seems to perform the best or comparable for 5 of the 6 apps except for EP, a highly compute-bound embarrassingly parallel application. To figure out why GCC performed worse in EP, we investigated the detailed vectorization report. We found that both compilers vectorized the same portion of the code, yet there is a 3 fold performance difference. We believe the performance difference is due to some other optimization, not vectorization.

A64FX vs. x86: To compare the single-core performance of A64FX against an Intel x86 system (Skylake), we compared the performance of these compilers in A64FX against the Intel compiler in Skylake. Intel compiler outperforms all the compilers in A64FX by a huge margin (from 1.6X to 5.5X). This performance difference can be attributed to both the systems and the compilers. The performance difference is the biggest for the compute-bound applications (5.5X for EP) while it narrows towards the memory-bound applications (1.6X for CG). However, the single-core performance difference in A64FX leaves a lot to be desired. We plan to investigate this phenomenon in depth in the future.

2) All Cores: Although A64FX has a significant single-core performance difference compared to Skylake, when using the whole node (all available cores) using OpenMP, A64FX does a much better job (Figure 4). In fact, in some cases, it outperforms Skylake (SP and UA). Further investigation shows that A64FX performs well in memory-bound applications (CG, SP, UA) while Skylake wins out in compute-bound applications. However, even in compute-bound applications, the performance gap narrows compared to a single core. The trend of A64FXs good performance in memory-bound apps can be attributed to higher memory bandwidth.

As for the performance of different compilers in A64FX, the trend from the single-core experiments continues, with GCC still performing the best in most applications. However, we do observe some interesting results with the ARM (in UA and BT) and the Fujitsu (in UA and SP) compilers where they perform significantly worse than GCC even though they had a comparable single-core performance. This can be attributed to data placement in a multi-threaded environment and the OpenMP library implementation. Further investigation reveals that the data placement strategy is the reason for such performance deviance, at least for the Fujitsu compiler. The Fujitsu compiler has a default policy of allocating all the data in CMG 0. Once we changed the policy to first touch, the Fujitsu compiler showed a much better performance in SP and a slightly better performance in all the apps, including UA. The bar representing `fujitsu-first-touch` shows the performance with first touch binding. However, as seen in figure 4, the performance improvement in UA is still not significant enough for it to be comparable with other compilers. We plan to investigate the reason behind this as well as the reason behind the performance deviance of arm compiler in BT and UA in the future.

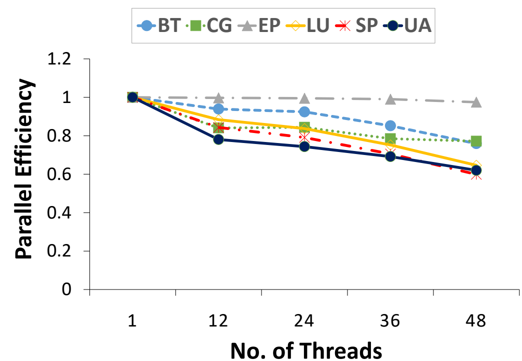

Fig. 5. Scaling (Parallel efficiency) of NAS parallel benchmark applications in A64FX using the GNU compiler.

3) Scaling: An important aspect of HPC system design is the scaling/parallel efficiency it can achieve for different applications. To compare the scaling/parallel efficiency of the NPB applications on the A64FX and the Skylake node, we chose two of the best performing compilers in those systems

(GCC in A64FX and icc in Skylake). Figure 5 and 6 show the parallel efficiency of the NPB applications on A64FX and Skylake respectively. A64FX shows better scaling for all the applications compared to Skylake. In A64FX, across 48 cores, EP (compute-bound) scales almost linearly while other apps scale favorably, with SP (memory-bound) having the least scaling/parallel efficiency of 0.6 across all 48 cores. Compared to that, Skylake has a scaling/parallel efficiency between 0.7 (in EP) and 0.25 (in SP).

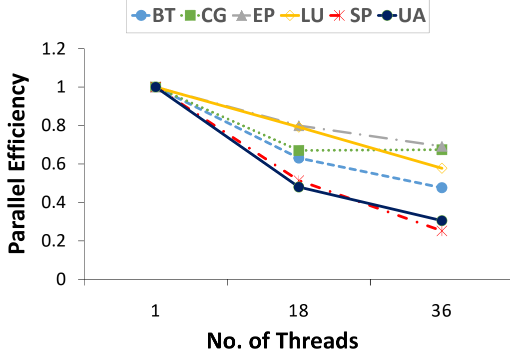

Fig. 6. Scaling (Parallel efficiency) of NAS parallel benchmark applications in x86 (Skylake) using the Intel compiler.

B. Discussion

During this experimentation, we focused on the out-of-the-box experience of an application developer. Hence, we used the compiler and runtime defaults for the experiments. As a result, the result presented here may not represent the best performance that can be achieved by different toolchains through use of more detailed options and environment variables.

VI. LULESH

The LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) proxy application was developed as representative of a challenge problem as well as a proxy for Exascale applications [17]. LULESH solves a simplified Sedov blast problem with analytic answers while capturing the numerical essentials of more complex hydrodynamic applications [18]. Here we report early tests of LULESH 1.0 [19] on the Ookami A64FX system, using reasonable defaults for the deployed compiler tool chains. These defaults are shown in Table II and Figure 7, which also contains the timings for single thread (st) and full occupation (mt) of the available cores (48 for A64FX and 32 for the Intel Skylake (Xeon Gold 6130, 16 cores/socket, 32 cores/server, 2.1 GHz base frequency) comparison system). Base represents the reference 1.0 LULESH code, and Vect the result of an available vectorized implementation (done originally for the Intel Sandy Bridge architecture) [19].

Graphically these results are shown in Figure 7. Table II shows promising vectorization for LULESH based on code tuned for Intel architectures.

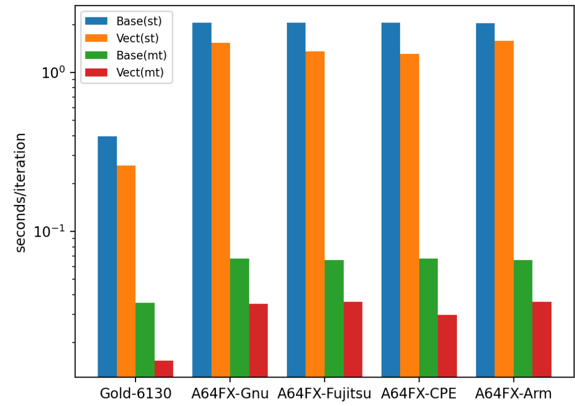

Fig. 7. Comparison of LULESH timings for compilers deployed thus far on Ookami. For comparison results are also shown for an Intel Skylake (Xeon Gold 6130, 16 cores/socket, 32 cores/server, 2.1 GHz base frequency) system.

VII. HPC Challenge Benchmark (HPCC)

The High-Performance Computing Challenge (HPCC) benchmark [15] is built on top of the popular High-Performance LINPACK (HPL) [16] and combines several other benchmarks. Here we concentrate on matrix-matrix multiplication (DGEMM), HPL, and Fast Fourier Transformation (FFT). The first two use linear algebra libraries, and the last one uses the FFT libraries.

Within XDMoD [14], an HPC resource utilization and performance analysis tool, we use HPCC for performance monitoring of several HPC resources from XSEDE and other systems. This allows us to compare the performance across a wide range of resources. We have chosen several HPC resources to make an interesting comparison (Table III). The TACC Stampede 2 system consists of two node types: Intel Skylake X (SKX) and Intel Knights Landing (KNL). Like Ookami, both have a 512-bit wide SIMD instructions set (AVX512), and SKX nodes have 48 cores each (like Ookami). KNL has simpler cores than SKX but has a higher core count, and both nodes have the same theoretical peak FLOP/s. Thus we can compare several different systems, all of which utilize a 512-bit wide SIMD instruction set. PSC Bridges 2 and SDSC Expanse have the same AMD, Zen 2 generation, CPUs and 128 cores/node for the two-socket machine. However, they only have a 256-bit wide SIMD instructions set (AVX2). Thus, it is interesting to see whether the narrower SIMD instruction with a higher core count will be more efficient.

The matrix and vector sizes for the tests were set similarly to weak scaling problems with the fixed number of elements per node. Specifically, in HPL, the matrix size was set to $(20,000\sqrt{N_n})^2$, where N_n is the number of nodes. In the embarrassingly parallel DGEMM test, each MPI process performs a test on the matrix of size $(20,000\sqrt{N_n/N_c})^2$ where N_n and N_c is the number of nodes and cores per node respectively. The FFT was done on a vector of size $20,000^2 N_n$.

We tested a large set of LA and FFT libraries on Ookami. Several of them already provide some SVE optimized routines,

TABLE II
TIMINGS FOR LULESH.

Compiler	Version	Flags	Base(st)	Base(mt)	Vect(st)	Vect(mt)
ARM	21.0	`-O3 -armpl -mcpu=native -fopenmp -fvectorize`	2.030	0.0661	1.575	0.0359
CPE	21.03	`-O3 -h vector3 -h msgs -h negmsgs -h omp`	2.055	0.0677	1.310	0.0298
Fujitsu	1.0.20	`-Kfast -KSVE -Koptmsg=2 -Kopenmp`	2.052	0.0662	1.359	0.0361
GNU	11.1.0	`-Ofast -fopenmp -Wall -mcpu=a64fx`	2.054	0.0674	1.533	0.0351
Intel/x86_64	19.5	`-O3 -qopenmp`	0.395	0.0355	0.260	0.0154

TABLE III
SPECIFICATIONS OF COMPARED HPC SYSTEMS

System	CPU	SIMD Instructions	Cores per Node	Base Frequency, GHz	Peak GFLOP/s per Core	Peak GFLOP/s per Node
Ookami	Fujitsu A64FX	SVE (512 wide)	48	1.8	57.6	2765
TACC Stampede 2	Intel Xeon Platinum 8160, Skylake (SKX)	AVX512	48	1.4 (AVX512, all cores)	44.8	2150
TACC Stampede 2	Intel Xeon Phi 7250, Knights Landing (KNL)	AVX512	68	1.4	44.8	3046
PSC Bridges 2	AMD EPYC 7742 (Zen2)	AVX2	128	2.25	36	4608
SDSC Expanse	AMD EPYC 7742 (Zen2)	AVX2	128	2.25	36	4608

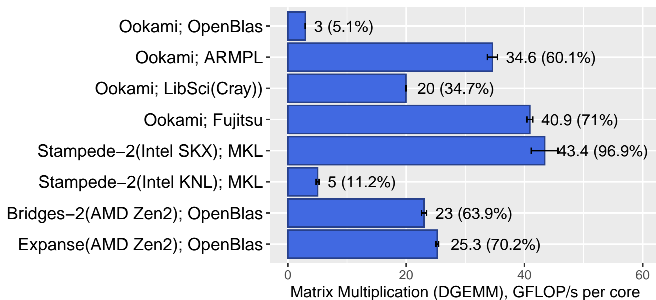

Fig. 8. Double-Precision General Matrix Multiplication (DGEMM) Floating-Point Performance per core, embarrassingly parallel (all cores performs same workload). The number in parenthesis show percentage of theoretical peak performance. Ookami is compared to several XSEDE systems, see Table III for some of their specifications. The error bars are the standard deviation of measurements.

among them: ARM Performance Library (ARMPL), Cray LibSci, Fujitsu BLAS, Cray FFTW, Fujitsu FFTW. OpenBLAS and FFTW currently do not have SVE optimizations but can be built and pass numeric tests. By comparing optimized and non-optimized libraries, we can speculate on the potential gains from specifically optimizing software for SVE instructions.

In the case of matrix-matrix multiplication, Fujitsu BLAS has the highest performance, almost 14 times faster than non-optimized OpenBLAS (Figure 8). The performance percentage of the theoretical peak performance for A64FX core is 71% which is between that for Intel KNL (11.%) and SKX (97%) systems and on par with AMD Zen 2 systems. Thus DGEMM performance is most likely already close to the maximum on A64FX for this matrix size. As for per-core performance compared to other systems, it is close to Intel SKX and 1.6 times faster than AMD Zen 2 cores. That is, the SIMD width is important for DGEMM. The ARM Performance Library and Cray LibSci also show significant speed-up over the non-optimized OpenBLAS library.

For HPL, Fujitsu BLAS also has the highest performance, nearly ten times faster than non-optimized OpenBLAS (Figure 9). Core-wise the A64FX core performance is similar to Intel SKX and 1.6 times faster than AMD Zen-2 cores. Per-node performance is comparable to that of the Intel SKX system and nearly 1.6 smaller than that of the AMD Zen-2 system. Thus overall, 2.7 times more cores and a slightly faster clock speed outperform a twice wider SIMD set. On multiple nodes, HPL does not scale well in the case of Fujitsu BLAS and MPI (Figure 9). ARMPL on the other hand shows better scalability and performance on two or more nodes. We speculate the Fujitsu MPI may not be optimized for our interconnect.

Within the FFT benchmark, we tested several versions of FFTW library and ARMPL (Figure 9). The ARMPL implementation seems to be unoptimized. The Fujitsu version of FFTW shows the best results, which is 4.2 times faster than the non-optimized FFTW. This is smaller than what we see in the LA library comparison, and the performance percentage of the theoretical peak is also below the well-established systems (Intel and AMD). The multi-node parallel performance is also a struggle, and it is relatively flat across all tested nodes count (Figure 9).D). In summary, The HPCC results showed good performance from several LA. On a single node, the Fujitsu BLAS is likely already close to the best possible performance, though lagged on multiple nodes. The FFT library routines seem to have room for improvement.

VIII. CONCLUSION

The Fujitsu and HPE/Cray toolchains stand out as delivering the very best performance. The ARM/LLVM-11 toolchain delivers competitive performance and fully supports current language standards. However, mainline LLVM currently includes limited support for SVE. GCC optimizes for both SVE and A64FX, and can sometimes generate the best vector and OpenMP performance. However, it is let down by the lack of a vector math library and, as a result some kernels might run 30-times slower than if using the Fujitsu or Cray compilers. Thus, high performance for full applications means adopting

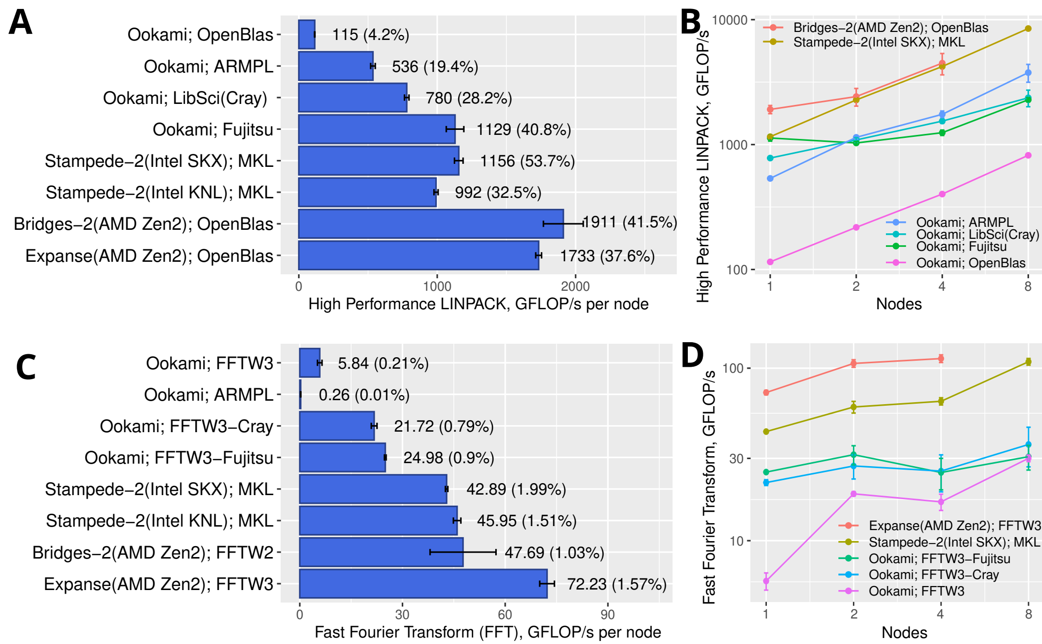

Fig. 9. High Performance LINPACK Floating-Point Performance on (A) single and (B) multiple nodes. Fast Fourier Transform (FFTW) Floating-Point Performance on (C) single and (D) multiple nodes. The number in parenthesis show percentage of theoretical peak performance. The error bars are the standard deviation of measurements.

a commercial toolchain and, possibly, limiting use of modern language features.

Acknowledgment

Ookami [7] is supported by the National Science Foundation (NSF) grant OAC 1927880, XDMoD by NSF OAC 1445806. OpenSHMEM is funded through Los Alamos National Laboratory, Grant #367958. Use of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory was supported by the Office of Science of the DOE under Contract No. DE-AC05-00OR22725. Use of XSEDE systems is supported by XSEDE grant TG-CCR120014. Computational support was also proved by the Center for Computational Research at the University at Buffalo [20]. We thank the MVAPICH and Open-MPI teams for assistance in tuning and deploying their software.

References

[1] RIKEN. 2021. https://www.r-ccs.riken.jp/en/fugaku
[2] Top500.org. 2020. https://www.top500.org/
[3] Graph500.org. 2020. https://graph500.org/
[4] hpcg-benchmark.org. 2020. https://www.hpcg-benchmark.org/
[5] J. Dongarra, P. Luszczek, and Y. Tsai. 2021. HPL-AI benchmark. https://icl.bitbucket.io/hpl-ai/
[6] Green500.org. 2019. http://www.green500.org/
[7] Stony Brook University. 2020. https://www.stonybrook.edu/ookami/
[8] A. Burford, A.C. Calder, D. Carlson, B. Chapman, F. CoŞKun, T. Curtis, C. Feldman, R.J. Harrison, Y. Kang, B. Michalowicz, E. Raut, E. Siegmann, D.G. Wood, R.L. Deleon, M. Jones, N.A. Simakov, J.P. White, D. Oryspayev Ookami: Deployment and Initial Experiences. PEARC 2021, doi:10.1145/3437359.3465578
[9] B. Michalowicz, E. Raut, Y. Kang, T. Curtis, B. Chapman, D. Oryspayev Comparing the behavior of OpenMP Implementations with various Applications on two different Fujitsu A64FX platforms PEARC2021, doi:10.1145/3437359.3465592
[10] R. Bird, N. Tan, S.V. Luedtke, S. Harrell, M. Taufer, B. Albright VPIC 2.0: Next Generation Particle-in-Cell Simulations PrePrint, IEEE Transactions on Parallel and Distributed Systems, doi:10.1109/TPDS.2021.3084795
[11] Summer school and workshop on parallel computing funded by ECP https://github.com/wadejong/Summer-School-Materials
[12] https://github.com/fujitsu/A64FX/tree/master/doc
[13] https://sleef.org/
[14] J.T. Palmer, S.M. Gallo, T.R. Furlani, M.D. Jones, R.L. DeLeon, J.P. White, N. Simakov, A.K. Patra, J. Sperhac, T. Yearke, R. Rathsam, M. Innus, C.D. Cornelius, J.C. Browne, W.L. Barth, R.T. Evans. Open XDMoD: A Tool for the Comprehensive Management of High-Performance Computing Resources. Computing in Science & Engineering, Vol 17, Issue 4, 2015, pp. 52-62. 10.1109/MCSE.2015.68
[15] P. Luszczek and J. Dongarra and et al., Introduction to the HPC Challeng Benchmark Suite, ICL Technical Report ICL-UT-05-01, University of Tennessee - Knoxville, 2005
[16] A. Petitet and R. C. Whaley and J. Dongarra and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://netlib.org/benchmark/hpl
[17] https://asc.llnl.gov/codes/proxy-apps/lulesh
[18] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254, https://asc.llnl.gov/sites/asc/files/2021-01/spec-7.pdf
[19] I. Karlin, A. Bhatele, B. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, D. Wong. LULESH Programming Model and Performance Ports Overview, December 2012, pages 1-17, LLNL-TR-608824, https://asc.llnl.gov/sites/asc/files/2021-01/lulesh_ports1.pdf
[20] Center for Computational Research, University at Buffalo, http://hdl.handle.net/10477/79221
[21] D. H. Bailey et al., "The NAS parallel benchmarks summary and preliminary results," Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, 1991, pp. 158-165, doi: 10.1145/125826.125925.
[22] S. N. U. Center for Manycore Programming, "Implementation of NAS parallel benchmark using C," http://aces.snu.ac.kr/SNU_NPB_Suite.html, 2013.