PROCEEDINGS OF SCIENCE

Software development and performance of Fugaku and ARM architectures

Yoshifumi Nakamura[a,*]

[a] *RIKEN Center for Computational Science, Kobe, Hyogo 650-0047, Japan*

E-mail: nakamura@riken.jp

The supercomputer "Fugaku" was jointly developed by Fujitsu Limited and RIKEN, and is the latest supercomputer installed at the RIKEN Center for Computational Science in Kobe, Japan. In the recent TOP500, HPCG, HPL-AI, and Graph500 benchmark rankings, it has been ranked No. 1 in the world for four consecutive terms. The CPU installed in Fugaku is a 48-core + 2 assistant core processor called A64FX, which is an extension of the Arm v8-A instruction set architecture for high-performance computing, and was developed by Fujitsu as the processor for Fugaku. The CPU consists of four "core memory groups" (CMGs) of 12 cores, and the L2 cache is shared by the 12 cores in the CMG. The main memory per node is 32GiB. The interconnect is a Tofu interconnect D. We will present an overview of Fugaku, development of LQCD code for A64FX and its performance, and large-scale benchmark results on Fugaku.

The 38th International Symposium on Lattice Field Theory, LATTICE2021 26th-30th July, 2021
Zoom/Gather@Massachusetts Institute of Technology

*Speaker

PoS(LATTICE2021)023

1. Introduction

As described in the full paper[1] on the large-scale benchmark test of lattice QCD on the supercomputer Fugaku (hereinafter referred to as Fugaku), we explain the specification of Fugaku and introduce a brief overview optimization for quark solver and the results of the benchmark tests. We also present the development status of lattice QCD software on A64FX.

2. Fugaku

Fugaku is a new japanese supercomputer developed by RIKEN and Fujitsu, which is the successor of the supercomputer K (hereinafter referred to as K). In the recent TOP500, HPCG, HPL-AI, and Graph500 benchmark rankings, it has been ranked No. 1 in the world for four consecutive terms (June 2020, November 2020, and June 2021, November 2021) [2]. It has a total of 432 racks with a total of 158976 nodes (384 nodes $\times$ 396 racks + 192 nodes $\times$ 36 racks). One node has one A64FX processor [3] and 32 GiB main memory (four 8 GiB High Bandwidth Memory 2). The peak memory bandwidth per node is 1024 GB/s. The node has two external interfaces, one is computational network called Tofu interconnect D (TofuD) [4] and another is PCIe Gen3 16 lanes. The node topology is (X, Y, Z, a, b, c) : (24, 23, 24, 2, 3, 2). The combination of XYZabc can be used to create a 3-dimensional torus. It can also perform fast allreduce using Tofu's hardware barrier function, but only up to three floating-point numbers and eight integer numbers. Link bandwidth is 6.8 GB/s. Injection bandwidth is 40.8 GB/s. Concurrent communications with 6 RDMA engines are available. L2$ misses may be reduced by cache injection which is a function to write received data directly to L2$.

The A64FX processor has designed based on Armv8.2-A instruction sets with the Scalable Vector Extension (SVE). A64FX has 48 cores for compute and two or four assistant cores for OS services. It consists of four Core Memory Groups (CMG) connected by ring bus. Each CMG has 12 compute cores which share L2 cache. Two operating frequency modes of normal 2.0 GHz and boost 2.2 GHz are available on Fugaku. While the specification of the Armv8.2-A with SVE allows hardware developers to select a vector length from 128 to 2,048 bits, the 512-bit-width SIMD arithmetic units had been chosen on Fugaku. The processor supports out-of-order execution of instructions. Other specifications are summarized in Table 1. Pictures for CPU die A64FX, Tofu interconnect, Tofu unit, and rack of Fugaku are shown in Fig. 1.

3. QWS

We have developed "Lattice quantum chromodynamics simulation library for Fugaku with wide SIMD" (QWS [5]) to get high performane for computing quark solver on Fugaku. It is an open source software on github. The QWS library contains not only the optimized linear solver for quark solver but also may functions such as the sparse-matrix-vector multiplication routines or linear-algebra routines for quark fields, which can be used as building blocks of algorithms developed by the library users. As a practical example, we introduce the use in the multi-grid solver algorithm implemented in the Bridge++ code set in which the domain-decomposed Clover-Wilson operator in QWS is called alternatively to the original code and achieves considerable acceleration [6]. While

Table 1: A64FX CPU Specifications. TF denotes TFLOPS. Cache performance is at normal mode of 2.0 GHz.

	description		
Architecture	Armv8.2-A SVE (512 bit SIMD)		
Core	48 (+ 2 or 4 assistant cores)		
Perfromance	double prec.	single prec.	half prec.
Normal mode	3.072 TF	6.144 TF	12.288 TF
Boost mode	3.3792 TF	6.7584 TF	13.5168 TF
Cache	L1D/core: 64 KiB, 4way, 256 GB/s (load), 128 GB/s (store) L2/CMG: 8 MiB, 16way L2/node: 4 TB/s (load), 2 TB/s (store) L2/core: 128 GB/s (load), 64 GB/s (store)		
Memory	32 GiB, 1024 GB/s		
Interconect	TofuD (28 Gbps × 2 lane × 10 port)		
PCIe	Gen3 16 lanes		
Technology	7nm FinFET		

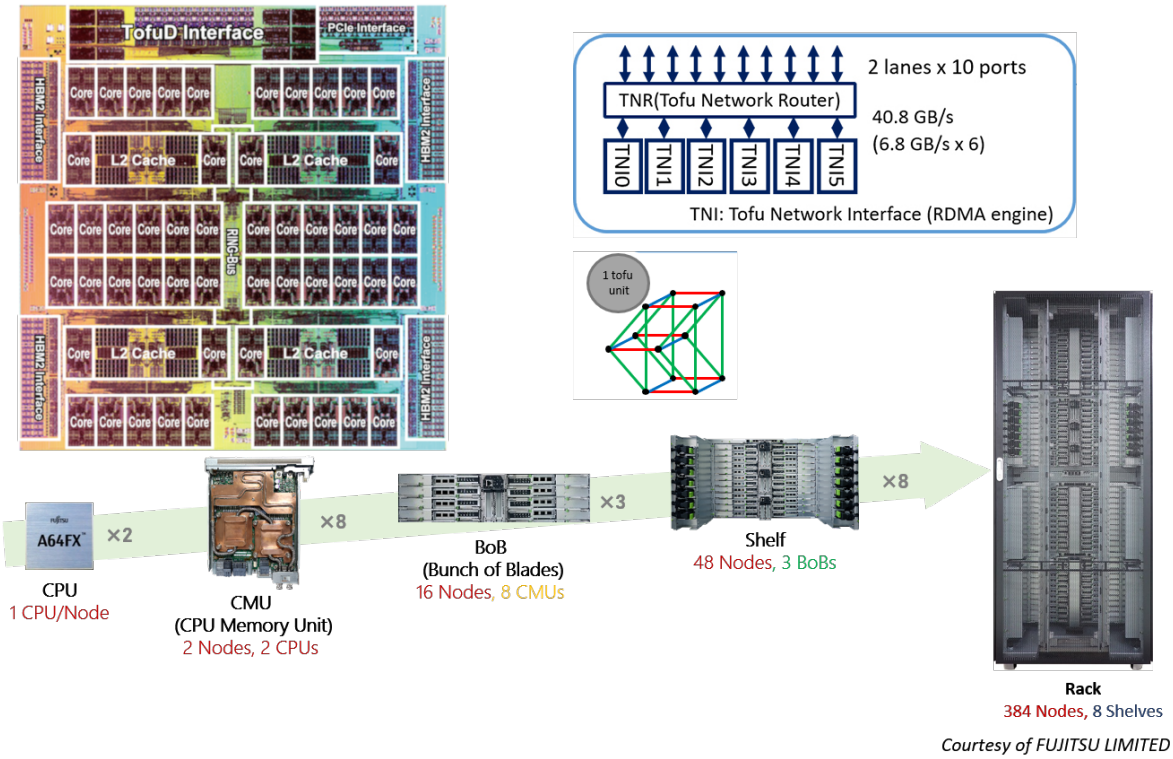

Figure 1: Pictures for CPU die of A64FX, Tofu interconnect, Tofu unit, and rack of Fugaku

the convention of the gamma-matrix and data layout are different in QWS and Bridge++, they are properly converted before and after the QWS functions are called.

We also expect that QWS provides a working example of the optimization techniques for A64FX architecture. While the quantitative evaluation of the components of QWS is specific to the fermion operator and execution setup, the improved implementation is generic and applicable to the other kinds of fermion operators as well as to many ingredients of LQCD simulations. Indeed in the multi-grid solver in [6] the other ingredients, such as the fermion operator on the coarse lattice and the inter-process communication, are accelerated by partly referring to the QWS implementation.

4. Tuning quark solver on Fugaku

To get high performance on Fugaku, effective SIMD vectorization with 512 bits wide SIMD is very important. The original code used for Fugaku co-design was tuned for narrow SIMD width which was used in HPC architectures in early 2000s. On K, the data layout for complex number, (Real-Imag)-(Real-Imag)-..., was used since the SIMD width was 128 bits and operations for complex numbers were supported. In the case of wide SIMD width in recent HPC architectures, a different optimization like for vector computers in 1990s is needed. We employ the following real number data layout (C-style array or Row major order),

$$\begin{aligned} \text{Fugaku(double)} &: [\text{nt}][\text{nz}][\text{ny}][\text{nx}/8][3][4][2][8]\,, \\ \text{Fugaku(single)} &: [\text{nt}][\text{nz}][\text{ny}][\text{nx}/16][3][4][2][16]\,, \\ \text{Fugaku(half)} &: [\text{nt}][\text{nz}][\text{ny}][\text{nx}/32][3][4][2][32]\,, \\ \text{cf. K} &: [\text{nt}][\text{nz}][\text{ny}][\text{nx}][3][4][2]\,, \end{aligned} \tag{1}$$

where nt, nz, ny, nx are the local domain lattice size in t,z,y,x directions, respectively. nx is divided and packed to the SIMD of Fugaku. The factor 3 sized rank corresponds to the color index, the 4 sized rank to the spin index, and the 2 sized rank to the complex real-imaginary index. We used the complex number data major layout on K. For Fugaku, we layout continuous x site index first by blocking to fit with 512 bits wide SIMD vector. We optimize the x–direction calculation by using Arm C Language Extensions (ACLE) [7]. For the stencil computation in the x–direction, the vector element shift operation is required. Instead of shifting data on vector registers, we utilize vector load with mask operation (named predicate operation) functionality of SVE. Fig 2 is a schematic picture of the x–direction shift for double precision data layout by using two load operations with predicate registers and one XOR operation.

We also have applied other general optimizations, e.g., removing temporal arrays, manually prefetching explicitly 256 B for all arrays by software prefetch, OMP Parallel region expansion that we put "omp parallel" on higher level caller routines because making omp parallel region is costly. This is important on many core architectures.

A function `__mult_clvs`, which corresponds to local matrix multiplication of inverted clover term, is used in `ddd_in_s_`, `jinv_ddd_in_s_`, and `ddd_out_s_`. We found that a naive implementation of `__mult_clvs` was inefficient due to load and store operations caused by register spill/fill. It is difficult for the compiler to calculate the optimal instruction schedule for such a large computation. However, we found a pattern on the code that allows an efficient schedule. Then, we

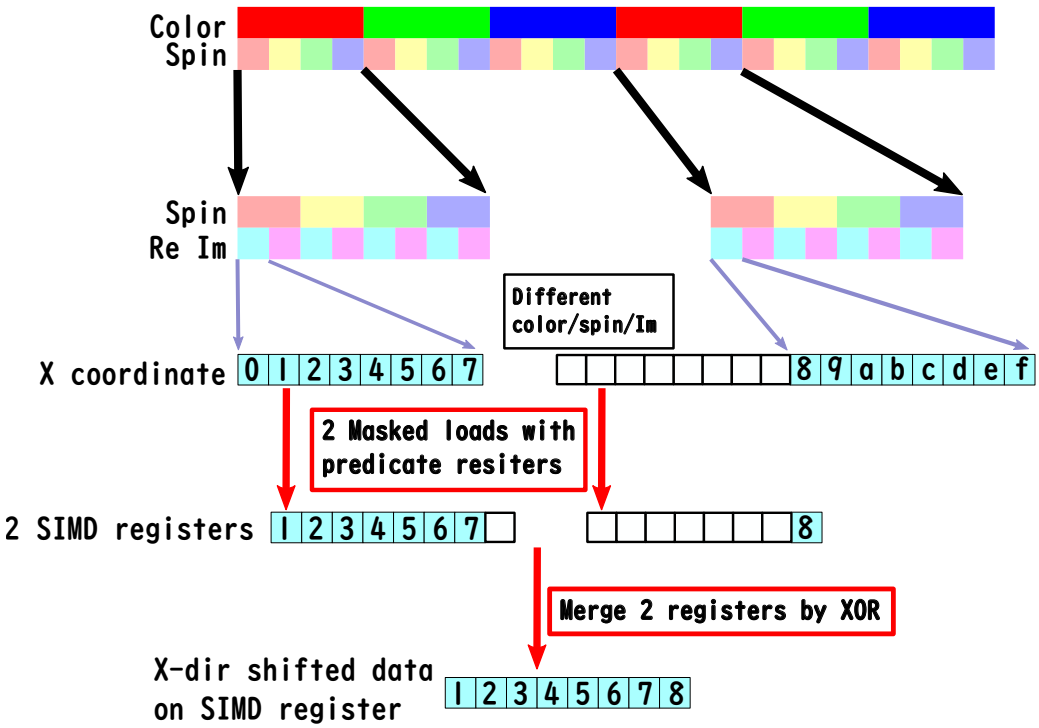

Figure 2: X–direction shift by using two load operations with predicate registers and one XOR operation for data layout in double precision.

reordered the operations at the source code level. Fig. 3 is the outline of the original code. The blocks starting with the red line share four values. So by arranging the rows of these blocks in a round robin fashion, we can minimize the interval of value reuse and get instruction-level parallelism. Fig. 4 is the the optimized code, where 16 streams of chained FMA (Fused Multiply-Add) operations are executed in a round robin fashion. Since the FMA latency is 9 and the number of the FMA pipelines is 2, 18 independent operations are required to fill the pipelines. Therefore, the theoretical efficiency of this code is 89 % (=16/18). In addition, the schedule allows for the reuse of register values at short intervals; the number of registers required does not exceed 32 (the number of registers in the architecture). Since each loaded value is reused once, the number of FMAs and loads are equal. Since the FMA and load pipelines are also equal in number, the load pipeline will not become a bottleneck. We prevented undesired compiler optimization, which is common subexpression elimination for long-distance reuse, by splitting blocks with if statements. We also specified the compiler flags that suppress instruction scheduling (`-Knosch_post_ra -Knosch_pre_ra -Knoeval`). We confirmed that these optimizations reduced the number of spills from the original 512 to the optimized 14.

The above is an overview of the optimization in computational cores and CMGs. On the other hand, optimization of communication is also very important to improve the performance. LQCD needs nearest neighbor stencil communication between surface sites. To minimize communication time, we adopt the double buffering algorithm and implement it by using the uTofu API library to directly use the RDMA engine called Tofu Network Interface (TNI) instead of the MPI communi-

```
void __mult_clvs(rvecs_t sc[3][4][2], rvecs_t a[2][36]) {
  rvecs_t x[2][6][2];
  rvecs_t y[2][2];

  for (int c=0;c<3;c++){
    for (int ri=0;ri<2;ri++){
      for (int v=0;v < VLENS; vv++){
        x[0][0+c][ri].v[v] = sc[c][0][ri].v[v] + sc[c][2][ri].v[v];
        x[0][3+c][ri].v[v] = sc[c][1][ri].v[v] + sc[c][3][ri].v[v];
        x[1][0+c][ri].v[v] = sc[c][0][ri].v[v] + sc[c][2][ri].v[v];
        x[1][3+c][ri].v[v] = sc[c][1][ri].v[v] + sc[c][3][ri].v[v];
      }
    }
  }

  for (i=0;i<2;i++){
    for (int v=0;v < VLENS; vv++){
      y[i][0].v[v] = a[i][ 0].v[v] * x[i][0][0].v[v] +
                     a[i][ 6].v[v] * x[i][1][0].v[v] +
                     a[i][ 8].v[v] * x[i][2][0].v[v] +
                     a[i][10].v[v] * x[i][3][0].v[v] +
                     a[i][12].v[v] * x[i][4][0].v[v] +
                     a[i][14].v[v] * x[i][5][0].v[v] -
                     a[i][ 7].v[v] * x[i][1][1].v[v] -
                     a[i][ 9].v[v] * x[i][2][1].v[v] -
                     a[i][11].v[v] * x[i][3][1].v[v] -
                     a[i][13].v[v] * x[i][4][1].v[v] -
                     a[i][15].v[v] * x[i][5][1].v[v];

      y[i][1].v[v] = a[i][ 0].v[v] * x[i][0][1].v[v] +
                     a[i][ 6].v[v] * x[i][1][1].v[v] +
                     a[i][ 8].v[v] * x[i][2][1].v[v] +
                     a[i][10].v[v] * x[i][3][1].v[v] +
                     a[i][12].v[v] * x[i][4][1].v[v] +
                     a[i][14].v[v] * x[i][5][1].v[v] +
                     a[i][ 7].v[v] * x[i][1][0].v[v] +
                     a[i][ 9].v[v] * x[i][2][0].v[v] +
                     a[i][11].v[v] * x[i][3][0].v[v] +
                     a[i][13].v[v] * x[i][4][0].v[v] +
                     a[i][15].v[v] * x[i][5][0].v[v];
    }
  }

  for (int v=0;v < VLENS; vv++){
    sc[0][0][0].v[v] = y[0][0].v[v] + y[1][0].v[v];
    sc[0][0][1].v[v] = y[0][1].v[v] + y[1][1].v[v];
    sc[0][2][0].v[v] = y[0][0].v[v] - y[1][0].v[v];
    sc[0][2][1].v[v] = y[0][1].v[v] - y[1][1].v[v];
  }
```

The same pattern follows by 6 times

Figure 3: `__mult_clvs` before optimizations.

cation library. The uTofu interface has lower latency than that of MPI and they can be used together in an application, so that global operations such as reduction for vector inner-product are processed by MPI functions while the nearest neighbor 1-to-1 communications are done by uTofu. The details are described in our paper[1].

4.1 Optimization of process mapping and TNI allocation

To minimize neighbor communication in a large-scale benchmark test using the entire Fugaku system, it is not enough to simply use double buffering and uTofu, but it is necessary to perfectly map the MPI processes of 4D QCD to the Tofu of 6D mesh torus, and specify the links and TNI that each neighbor communication uses. This subsection describes a method for searching for process mapping and TNI allocation. This method searches a configuration that minimizes the transfer message length of the Tofu link or TNI. In the following explanation, the space of process allocation

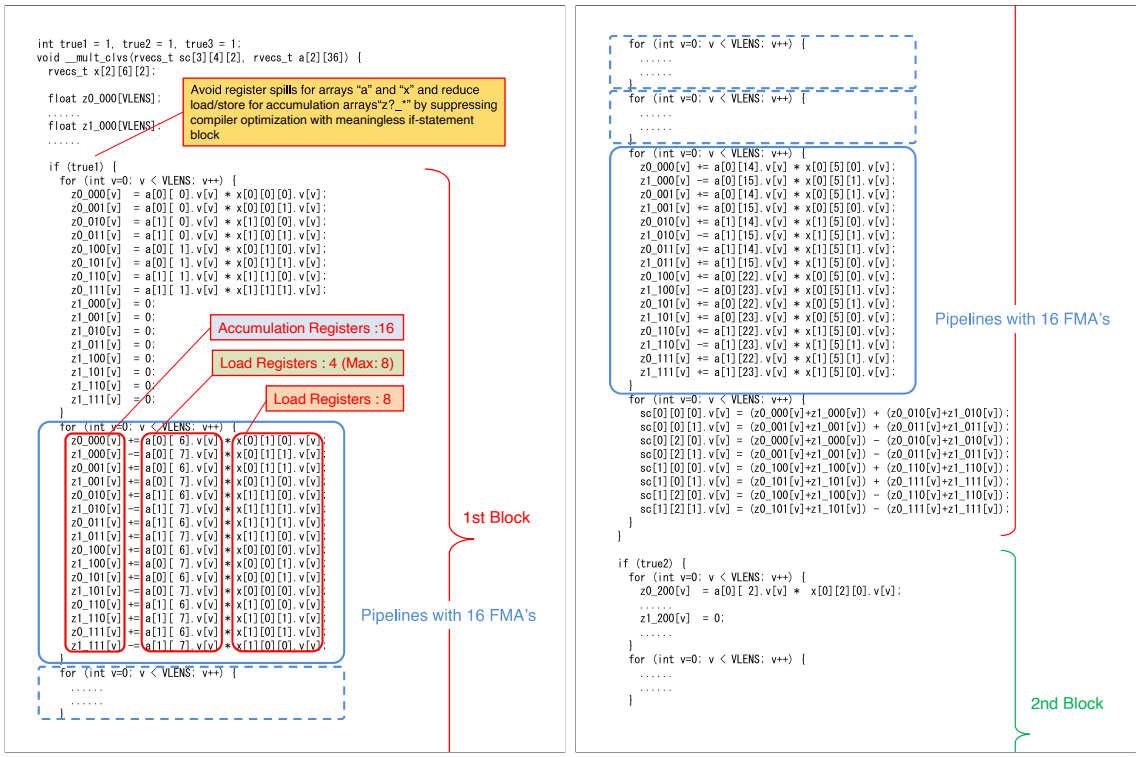

Figure 4: `__mult_clvs` after optimizations.

is expressed as $\{T_1, T_2, \ldots, T_n\}$,

$$
\begin{aligned}
&T_i = (t_i, s_i, o_i, h_i)\,, \\
&t_i \in \{\mathrm{TX}, \mathrm{TY}, \mathrm{TZ}, \mathrm{TA}, \mathrm{TB}, \mathrm{TC}, \mathrm{IN\_NODE}\}\,, \\
&s_i \in \mathbb{N}\,, \quad o_i \in \{\mathrm{torus}, \mathrm{mesh}\}\,, \quad h_i \in \mathbb{N}\,,
\end{aligned}
\tag{2}
$$

where T_i is the respective axis of the network, t_i denotes the physical axis of Tofu (when multiple processes are allocated to one node, the axis is expressed as IN_NODE), s_i is the length of T_i, o_i is torus if the coordinates at both ends of T_i are adjacent to each other, otherwise it is mesh, and h_i is the distance between the adjacent coordinates in the T_i axis as viewed in the t_i axis. For example, a physical X-axis of length 24 (24 being the entire system, it is torus) is represented as (TX, 24, torus, 1).

The physical X-axis of length 24 is hypothetically divided into the axes TXc of length 2 and TXd of length 12, and the coordinate x of TX corresponds to the coordinate [x mod 2] of TXc and the coordinate $[x/2]$ of TXd. they are (TX, 2,mesh, 1) and (TX, 12, torus, 2), respectively.

Step 1 For the input Tofu physical shape, enumerate the combinations of at most two divisions for each physical axis. In other words, for all s_1 and s_2 satisfying $s = s_1 \times s_2$ for a physical axis t of length s and tortuosity o, enumerate $\{(t, s_1, \mathrm{mesh}, 1), (t, s_2, o, s_1)\}$, where if either s_1 or s_2 is 1, then $\{(t, s, o, 1)\}$.

Step 2 For each physical axis, select one from those listed in Step 1 and combine them. Then, list all the combinations as P_2. For example, when the lengths of the physical X, Y, and Z axes are 24, 22, and 24, respectively, an example of $p_2 \in P_2$ is {(TX, 24, torus, 1), (TY, 11, mesh, 2), (TY, 2, mesh,1), (TZ, 8, torus, 3), (TZ, 3, mesh, 1), (TA, 2, mesh, 1), (TB, 3, torus, 1), (TC, 2, mesh, 1), (IN_NODE, 4, mesh,1)}.

Step 3 If all the elements of P_2 have already been processed, terminate the algorithm. For $p_2 \in P_2$, which has not yet been selected, select and combine axes to make a torus from p_2, and list the possible combinations assigned to each axis of the process partition. That is, when the lengths of the four axes of the process partition are (n_1, n_2, n_3, n_4) and we assign the following axesC_i, enumerate all combinations of (C_1, C_2, C_3, C_4).

$$C_i = \{(t_{i,1}, s_{i,1}, o_{i,1}, h_{i,1}), \ldots, (t_{i,n}, s_{i,n}, o_{i,n}, h_{i,n})\} \subset p_2\,, \tag{3}$$

where C_i satisfies $n_i \le \prod_j s_{i,j}$, and two or more $t_{i,j}$ exist except for IN_NODE or $o_{i,j}$ = torus exists. We assume that the torus passing through n_i vertices can be constructed even if n_i is even and $\prod_j s_{i,j}$ (except j for which $t_{i,j}$=IN_NODE) is odd. We can also exclude combinations where there are j, k such that $t_{i,j} = t_{i,k}$, since they are equivalent to combinations that do not split that physical axis. For example, for an axis with process number 6, we can assign {(TY, 2,mesh, 1), (TZ, 3,mesh, 1)}. Let P_3 be the result of the enumeration.

Step 4 If all elements of P_3 have already been processed, return to Step 3. For $p_3 \in P_3$ that has not yet been selected, compute the corresponding F of the message length to be transferred for each combination of the Tofu axis and the LQCD axis of process division. The elements of F are

$$\begin{aligned} &(t, q, d, m) \in F\,, \quad t \in \{\mathrm{TX}, \mathrm{TY}, \mathrm{TZ}, \mathrm{TA}, \mathrm{TB}, \mathrm{TC}, \mathrm{IN\_NODE}\}\,, \\ &q \in \{\mathrm{QX}, \mathrm{QY}, \mathrm{QZ}, \mathrm{QT}\}\,, \quad d \in \{\mathrm{plus}, \mathrm{minus}, \mathrm{both}\}\,, \quad m \in \mathbb{N}\,, \end{aligned} \tag{4}$$

where t is the physical axis, q is the axis of process division of LQCD, d is the direction of transmission as seen in the axis of process division, and m is the message length. The following substeps are used to calculate F for the next assignment to the process partition axis q.

$$\{(t_1, s_1, o_1, h_1), \ldots, (t_n, s_n, o_n, h_n)\} \tag{5}$$

Step 4.1 On the q-axis, find the number of inter-node communication processes n_{inter} and the number of intra-node communication processes n_{intra} per node. Set the overall number of processes per node as n. If t_i = IN_NODE, $n_q = s_i$; otherwise, $n_q = 1$. In this case

$$n_{\mathrm{inter}} = \frac{n}{n_q}\,, \quad n_{\mathrm{intra}} = \frac{(n_q - 1) \times n}{n_q}\,. \tag{6}$$

Step 4.2 Assign the length of the message sent per process at the q-axis to m_q. When $t_i \neq$ IN_NODE, let $(t_i, q, \mathrm{both}, n_{\mathrm{inter}} \times m_q \times h_i) \in F$. Multiply by h_i to include transfers to other nodes due to hops. If t_i = IN_NODE, then (IN_NODE, q, both, $n_{\mathrm{intra}} \times m_q$).

Step 5 For F obtained in Step 4, calculate the total transfer message length for each Tofu physical axis, where IN_NODE is excluded. The total transfer message length for physical axis t is the sum of m for all $(t, q, d, m) \in F$. If the best value among them exceeds the best value from other assignments obtained so far, return to step 4.

Step 6 For F obtained in Step 4, list the possible combinations of the six TNI assignments. For a physical axis t, when $(t, q_i, \text{both}, m_i) \in F$, we can use either Step 6a or 6b to assign TNI a, b ($a = b$ is also acceptable). These methods satisfy the restriction that, from the point of view of a Tofu link, there is only one TNI that uses the Tofu link. Let I_a, I_b be the set of communications assigned to a, b, respectively. Since there is no need to distinguish each of the six TNI, we can express $K = \{(I'_1, n_1), \ldots, (I'_m, n_m)\}$ instead of $(I_1, \ldots, I_6)$, where n_i is the number of j for which $I'_i = I_j$. In this way, when choosing the TNI to assign, we can reduce it to $|K|$ ways.

Step 6a When t is IN_NODE, or when there is exactly one axis of q_i that communicates in both directions in one adjacent communication (i.e., when there is one i such that $q_i \in$ {QY,QZ,QT}), assign $(t, q_i,$ plus, $m_i) \in I_a$, $(t, q_i,$ minus, $m_i) \in I_b$ or $(t, q_i,$ minus, $m_i) \in I_a$, $(t, q_i,$ plus, $m_i) \in I_b$ to each different q_i. This means that in the former case, I_a is used for communication in the positive direction of q_i, regardless of the direction of t, and I_b is used for communication in the negative direction of q_i, regardless of the direction of t. When q_i = QX and $a = b$, the communication of q_i never occurs in both directions at the same time, so only one of plus or minus shall be assigned. For example, when $\{(t, \text{QX}, \text{both}, m_x), (t, \text{QY}, \text{both}, m_y)\} \subset F$, the following assignments can be made respectively.

A) $\{(t, \text{QX}, \text{plus}, m_x), (t, \text{QY}, \text{plus}, m_y)\} \subset I_a$, $\{(t, \text{QX}, \text{minus}, m_x), (t, \text{QY}, \text{minus}, m_y)\} \subset I_b$

B) $\{(t, \text{QX}, \text{plus}, m_x), (t, \text{QY}, \text{minus}, m_y)\} \subset I_a$, $\{(t, \text{QX}, \text{minus}, m_x), (t, \text{QY}, \text{plus}, m_y)\} \subset I_b$

C) $\{(t, \text{QX}, \text{minus}, m_x), (t, \text{QY}, \text{plus}, m_y)\} \subset I_a$, $\{(t, \text{QX}, \text{plus}, m_x), (t, \text{QY}, \text{minus}, m_y)\} \subset I_b$

D) $\{(t, \text{QX}, \text{minus}, m_x), (t, \text{QY}, \text{minus}, m_y)\} \subset I_a$, $\{(t, \text{QX}, \text{plus}, m_x), (t, \text{QY}, \text{plus}, m_y)\} \subset I_b$

In this case, "A and D" and "B and C" are the same if a, b are swapped, respectively, so one of them can be omitted.

Step 6b When t is not an IN_NODE, assign $(t, q_i$,both, $m_i) \in I_a$ for all i. This means that if t has only a one-way link (either TA or TC), the communication through t uses a regardless of the direction of q_i. If t has links in both directions, we further assign $(t, q_i$,both, $m_i) \in I_b$. This means that communication through one direction, positive or negative of t, will use a regardless of the direction of q_i, and communication through the other direction will use b regardless of the direction of q_i. Since we do not need to distinguish between the positive and negative values of T until the end, we assign the same value. When $a = b$, in order to express the existence of both directions, we further use $(\{t\}, q_i$,both, $m_i) \in I_a$.

Step 7 For the allocations listed in Step 6, calculate the transfer message length of the bottleneck TNI. Calculate the transfer message length of the TNI for the allocation $I = \{(t_1, q_1, d_1, m_1), \ldots, (t_n, q_n, d_n, m_n)\}$ in Step 7.1 and Step 7.2. If the transfer message length of the bottleneck TNI is smaller than the best value so far, update the best value. After

completing Step 7, return to Step 4. The calculation of the transfer message length of the TNI by step 7 can be performed in the allocation by step 6, and if it exceeds the best value, the steps for the allocation can be terminated at that point.

Step 7.1 For each i for which t_i = IN_NODE, add m_i.

Step 7.2 In I', excluding those processed in Step 7.1, for each $q \in$ {QX,QY,QZ,QT}, divide it into $I' = \cup_q I_q$, where $I_q = \{(t_{q,1}, q, d_{q,1}, m_{q,1}), \ldots, (t_{q,n_q}, q, d_{q,n_q}, m_{q,n_q})\}$. For each I_q, add the largest $m_{q,i}$ of i that is $d_{q,i} \in$ {plus, both}. Furthermore, add the largest $m_{q,i}$ among i that is $d_{q,i} \in$ {minus, both}. In addition, add the largest $m_{q,i}$ of i that is $d_{q,i} \in$ {minus, both}. The reason why only the largest value is added is that only one link is used for communication in the same direction on the same axis.

5. Benchmark tests

We perform benchmark tests for "the single precision BiCGStab for a Wilson-clover Dirac matrix with Schwarz Alternating Procedure domain decomposition preconditioning [8] using Jacobi iteration for the local domain matrix inversion (hereinafter referred to as QCDJDD)" in quark solver. The lattice size of the target problem is 192^4. QCDJDD is divided to several computation and communication regions as in Table 2.

Table 2: Regions of QCDJDD

Computation regions on K	
`jinv_ddd_in_s_`	static solver in domain
`ddd_in_s_`	matrix vector multiplication in domain
`ddd_out_pre_s_`	preprocess for interdomain matrix vector multiplication
`ddd_out_pos_s_`	postprocess for interdomain matrix vector multiplication
`other_calc`	other calculation in iteration
Computation region on Fugaku	
`all_calc`	all calculation
`overlapped`	region overlapped by communication
Communication regions	
`comlib_irecv_all_c`	start receiving for neighboring communication
`comlib_isend_all_c`	start sending for neighboring communication
`comlib_recv_wait_all_c`	wait receiving for neighboring communication
`comlib_send_wait_all_c`	wait sending for neighboring communication
`s_drbicgstab_dd_hpc_iter_reduc1_`	Allreduce for one float
`s_drbicgstab_dd_hpc_iter_reduc2_`	Allreduce for two floats
`s_drbicgstab_dd_hpc_iter_reduc3_`	two times of Allreduce for three floats

5.1 Baseline performance measured on K

We run the quark solver on the target problem 192^4 using 8 OpenMP threads per node on 82944 nodes of the K computer (hereinafter referred to as K). The measured performance is shown

in Table 3 for single iteration of QCDJDD on K. The computation efficiency is estimated based on the peak double precision performance as 1. The performance on the regions `ddd_in_s_` and `comlib_recv_wait_all_c` are measured with communication-computation overlapping on K.

Table 3: Baseline performace measured on K.

		Exec. Time [ms]	Comp. Eff. (%)
Total time for computationl regions		27.99	34.8
Region name	`jinv_ddd_in_s_`	14.09	41.9
	`ddd_in_s_`	6.52	44.3
	`ddd_out_pre_s_`	0.95	12.6
	`ddd_out_pos_s_`	3.84	16.9
	`other_calc`	2.58	7.1
Total time communication regions		2.66	
Region name	`comlib_irecv_all_c`	0.45	
	`comlib_isend_all_c`	0.17	
	`comlib_recv_wait_all_c`	0.16	
	`comlib_send_wait_all_c`	0.17	
	`s_drbicgstab_dd_hpc_iter_reduc1_`	0.18	
	`s_drbicgstab_dd_hpc_iter_reduc2_`	0.85	
	`s_drbicgstab_dd_hpc_iter_reduc3_`	0.67	
Total time		30.65	31.8

5.2 Benchmark tests on Fugaku

Benchmark tests are performed on the boost mode 2.2 GHz without using Eco modes that one of two floating-point arithmetic pipelines is limited. Elapse time and performance of 500 iterations of QCDJDD and ddd_in_s_ region measured on two MPI processes using two CMGs for several problem sizes per MPI process are listed in Table 4 and Table 5. FLOP indicates a floating-point operation count calculated theoretically. Efficiency indicates floating-point operation efficiency against single precision floating-point operation peak. We see that performance for $32 \times 6 \times 4 \times 6$ is the best in the tests if the communications are not taken into consideration.

Table 4: Elapse time and performance for 500 iterations of QCDJDD measured on two MPI processes using two CMGs for several problem sizes per MPI process. FLOP indicates a floating-point operation count calculated theoretically per MPI process. TFLOPS indicates the performance per node. Efficiency indicates floating-point operation efficiency against the single precision floating-point operation peak.

Size	Elapse [s]	TFLOPS	Efficiency	FLOP
$32 \times 6 \times 4 \times 3$	0.334867	0.8272	12.24%	69254421000
$32 \times 6 \times 4 \times 6$	0.515010	1.0839	16.04%	139560981000
$32 \times 6 \times 8 \times 6$	1.145754	0.9786	14.48%	280304661000
$32 \times 6 \times 8 \times 12$	2.606202	0.8616	12.75%	561369621000
$32 \times 12 \times 8 \times 12$	5.778703	0.7773	11.50%	1122981141000

Table 5: Same as Table 4, but for region of ddd_in_s_ during 500 iterations of QCDJDD.

Size	Elapse [s]	TFLOPS	Efficiency	FLOP
$32 \times 6 \times 4 \times 3$	0.068043	1.1208	16.58%	19065600000
$32 \times 6 \times 4 \times 6$	0.119455	1.3170	19.49%	39329280000
$32 \times 6 \times 8 \times 6$	0.219403	1.4693	21.74%	80593920000
$32 \times 6 \times 8 \times 12$	0.559297	1.1699	17.31%	163584000000
$32 \times 12 \times 8 \times 12$	1.192644	1.1146	16.49%	332328960000

We use a fast Allreduce using the Tofu barrier in quark solver. The Allreduce up to three elements for MPI_DOUBLE and MPI_FLOAT can be performed on the Tofu barrier. In Table 6, we show Allreduce benchmark results on 72 racks, 27648 nodes of $48 \times 12 \times 48$ node shape by using "Intel(R) MPI Benchmarks 2019 Update 6, MPI 1 part (IMB-MPI1)" with and without Tofu barrier. Minimum (min), maximum (max), and average (avg) time for repetition number, 10000 are shown. The number of bytes (byte) is a message length to be reduced per one MPI_Allreuce call. And the number of counts (count) is a number of elements. The data type of MPI_FLOAT is reduced as default of IMB-MPI1. We see that Allreduce up to three elements with the Tofu barrier is about six times faster than one without the Tofu barrier and it is faster to split MPI_Allreduce for 15 elements into five MPI_Allreduce for three elements.

Table 6: Allreduce benchmark by IMB-MPI1.

		with Tofu barrier			without Tofu barrier		
byte	count	min [μs]	max [μs]	avg [μs]	min [μs]	max [μs]	avg [μs]
0	0	0.09	0.14	0.10	0.10	0.16	0.12
4	1	7.60	11.33	9.46	55.69	69.05	62.83
8	2	8.25	10.79	9.50	55.79	68.93	62.91
12	3	8.25	10.93	9.57	55.89	69.02	62.94
16	4	58.99	66.95	62.68	56.42	69.71	63.51
32	8	61.50	72.34	66.32	78.24	97.57	88.14
64	16	61.61	72.38	66.31	78.63	97.84	88.42
128	32	63.70	74.45	68.43	80.46	99.56	90.10

We show a weak scaling plot of the evaluation region in Fig. 5. The vertical line at 147456 nodes denotes the number of nodes used in the benchmark for the target problem size while 158976 nodes is a total nodes of Fugaku. We see a nice weak scaling from 432 nodes to 147456 nodes of the target nodes with a few exceptions caused by OS jitters. The elapse time increases 0.5 [ms] (about 7 %) from 432 nodes to 147456 nodes due to the time for Allreduce. The elapse times of five benchmark tests on 147456 nodes are 0.8000, 0.7998, 0.7982, 0.7989, and 0.7978 [ms], respectively. These are about 38.3 times faster than the elapse time, 30.65 [ms], for same problem setup on the full system of K. Performance is 102 PFLOPS, 10% floating-point operation efficiency against single precision floating-point operation peak. Averaged power is about 20 MW. The power efficiency is 5 GFLOPS/W.

Table 7 shows a breakdown of the total elapse time for the target problem size on 147456 nodes.

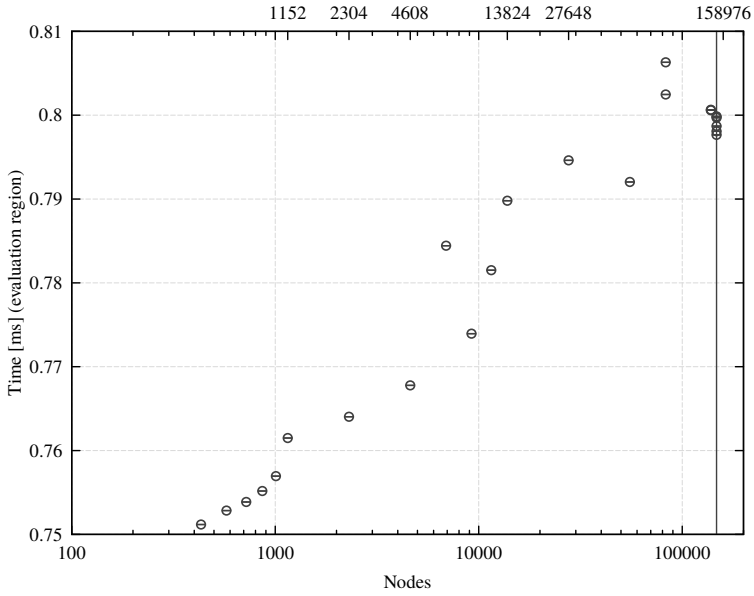

Figure 5: Weak scaling of the evaluation region.

The total time is divided into calculation time (all_calc) and communication time (all_comm). all_comm is divided into three parts for neighboring communication and three parts for Allreduce. Summed total elapse time in the table is slightly longer than 0.8000 [ms] which is measured in peak performance tests because there is a non-negligible overhead to measure elapse times for each region. We see half of time is spent for communication. In usual production runs, we may better use a smaller number of nodes.

Table 7: Elapse time breakdown.

region	Elapse time [ms]
all_calc	0.400
all_comm	0.407
comlib_isend_all_c	0.029
comlib_recv_wait_all_c	0.254
comlib_send_wait_all_c	0.062
s_drbicgstab_dd_hpc_iter_reduc1_	0.015
s_drbicgstab_dd_hpc_iter_reduc1_	0.016
s_drbicgstab_dd_hpc_iter_reduc1_	0.031
total	0.807

5.3 Grid benchmark tests on QPACE4

The performance of the Grid c++ QCD library [9] was measured on QPACE4, which is Fujitsu PRIMEHPC FX700 and deployed at Regensburg University. Meyer *et. al.*, had implemented Grid's lower-level functions by using ACLE and ported Grid to A64FX [10]. The results of the benchmark

tests for the domain wall fermion kernel, Wilson fermion kernel and SU(3) matrix product are shown in Fig. 6. Both the domain wall fermion kernel and the Wilson fermion kernel do not scale very well. This is probably mainly due to the fact that communication is not overlapped by computation. In SU(3) matrix multiplication, a code compiled with GCC can use about 80% of the theoretical bandwidth and achieve about 300 GFLOPS. Grid on A64FX shows that further performance improvement can be expected by changing the data layout of the complex numbers to a QWS-like data layout that separates the real and imaginary parts.

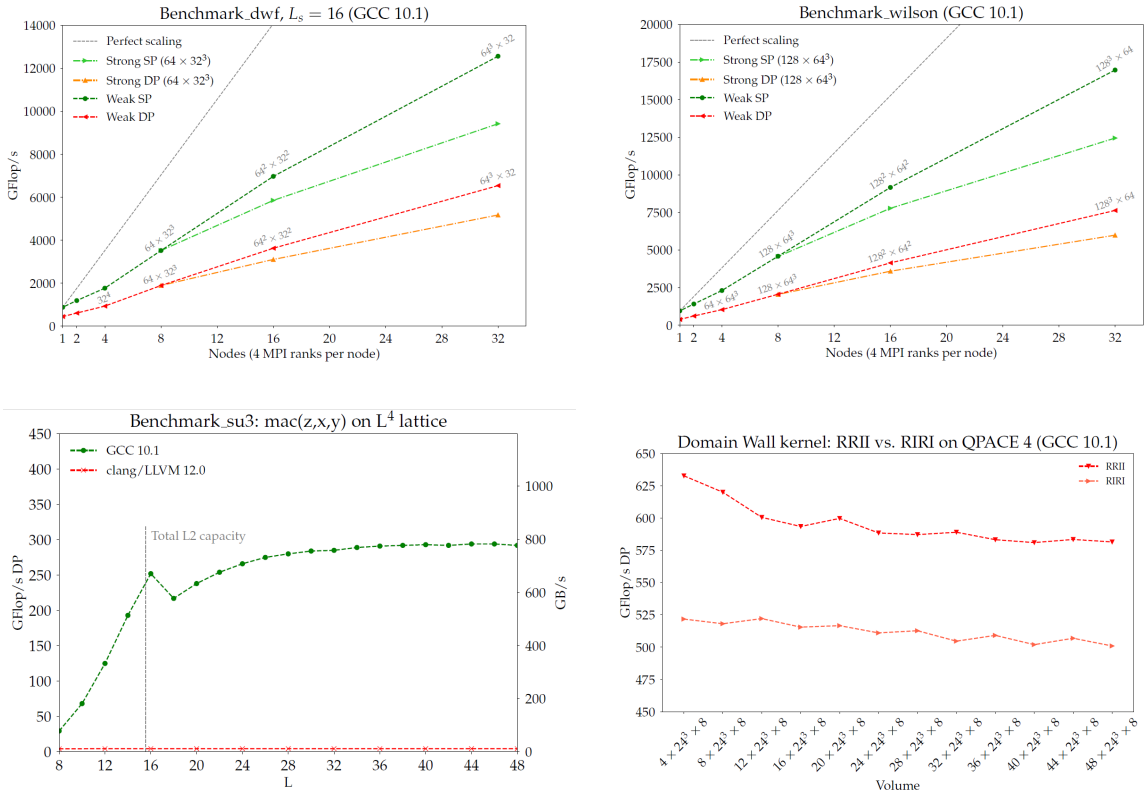

Figure 6: The benchmark tests for the domain wall fermion kernel, Wilson fermion kernel, SU(3) matrix product, and domain wall kernel with different data layout for complex numbers on QPACE4

5.4 Multigrid solver on Fugaku

Kanamori *et al.*, reported an implementation of a multigrid solver on Fugaku [6]. Their multigrid solvers are made from several components so that one can use a part of QWS such as Clover kernel. The code is developed by using Bridge++ code framework [11] and its extension. They compared the performance with LDDHMC, which is a reimplementation of QWS in Domain Decomposed HMC (nested BiCGStab with Domain Decomposition + SAP + mixed precision), using three ensembles (A, B, C), A at $m_\pi = 156$ MeV on $32^3 \times 64$, B at $m_\pi = 512$ MeV on $64^3 \times 96$, C at $m_\pi = 145$ MeV on $96^3 \times 96$. Due to the setup cost of multigrid solver, LDDHMC was faster in

solving one inversion, but in some cases, multigrid solver was faster in solving 12 inversions. It is effective for solving the inverse matrix-vector product of common matrices in measurements at small quark masses. Some optimization seems to be insufficient, and future performance improvement is expected.

6. Summary

We introduced Fugaku and the current status of LQCD software development on A64FX. As a milestone, we have achieved 102 PFLOPS, 10% floating-point operation efficiency against single precision floating-point operation peak, of Clover–Wilson quark solver on 192^4 lattice on Fugaku through the co-design in FS2020 project. It was observed that with proper optimization, execution efficiency of about 10% can be achieved and found that a little more performance improvement can be expected by selecting the problem size. From this co-design activity, several fed backs especially on communication was send to the system implementation. All the benchmark results on Fugaku have been obtained on the evaluation environment in the trial phase. It does not guarantee the performance, power and other attributes of Fugaku at the start of its public use operation. Finally, we would like to thank all the people who were involved in the LQCD working group of the project.

References

[1] K.-I. Ishikawa, I. Kanamori, H. Matsufuru, I. Miyoshi, Y. Mukai, Y. Nakamura, K. Nitadori, M. Tsuji, [arXiv:2109.10687 [hep-lat]].

[2] TOP500, https://www.top500.org.

[3] A64FX, https://github.com/fujitsu/A64FX.

[4] Y. Ajima *et al.*, IEEE Cluster 2018, 2018.

[5] Y. Nakamura, Y. Mukai, K.-I. Ishikawa, I. Kanamori, (https://github.com/RIKEN-LQCD/qws).

[6] I. Kanamori *et al.*, PoS LATTICE2021 (2021).

[7] Arm C Language Extensions, https://developer.arm.com/architectures/system-architectures/software-standards/acle

[8] M. Lüscher, Lattice QCD and the Schwarz alternating procedure, JHEP 0305, 052 (2003); Comput. Phys. Commun. 165, (2005) 119.

[9] P. Boyle *et al.*, Proceedings of LATTICE 15 (2016) 023 [arxiv:1512.03487 [hep-lat]].

[10] N. Meyer *et al.*, PoS LATTICE2021 (2021).

[11] https://bridge.kek.jp/Lattice-code/.