Grid on Vector Architectures

RIKEN R-CCS, LQCD Workshop

December 12, 2019

Peter Georg [1], **Benjamin Huth** [1], **Nils Meyer** [1],
Dirk Pleiter [1], Stefan Solbrig [1], Tilo Wettig [1],
Yuetsu Kodama [2], Mitsuhisa Sato [2]

[1] University of Regensburg (Regensburg, Germany)
[2] RIKEN R-CCS (Kobe, Japan)

Introduction

Grid

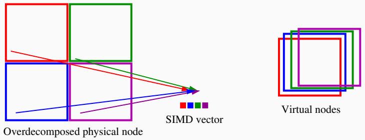
- Open-source Lattice QCD framework written in C++ 11
  maintained by Peter Boyle (Edinburgh, BNL) and co-workers
  GitHub: https://github.com/paboyle/Grid
  arxiv: https://arxiv.org/abs/1512.03487
- Targets massively parallel architectures supporting MPI + OpenMP + SIMD
- Vectorization on CPUs using intrinsics

Intel	SSE4	128 bits
	AVX/AVX2	256 bits
	AVX-512	512 bits
IBM	QPX	256 bits
Arm	NEONv8	128 bits
	SVE	512 bits

- In addition, generic data types are supported (user-defined array length)
- New architecture $\rightarrow$ introduce new intrinsics layer (few 100 lines)
- For QPX and AVX-512: LQCD operators in assembly
- Comes with a series of tests and performance benchmarks

Vectorization

- Grid decomposes local volume into “virtual nodes”
  - Sites from different virtual nodes are assigned to same register (site fusing)
  - This eliminates dependencies between register elements
  - 100% SIMD efficiency



- Intrinsics layer defines operations on arrays of size = SIMD width
  - Arithmetics of real and complex numbers
  - Intra- and inter-register permutations
  - Memory prefetch and streaming memory access
  - Conversion of floating-point precision
- Maximum array size in Grid is 512 bits

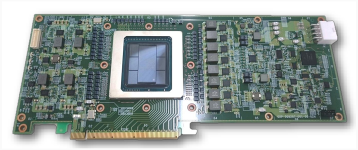
NEC SX-Aurora Tsubasa

NEC SX-Aurora Tsubasa

- SX-Aurora Tsubasa is the newest member of the NEC SX-series
- PCIe card with Gen3 x16 interconnect called vector engine (VE)

Vector engine model	Type 10A	Type 10B	Type 10C
Clock frequency [GHz]	1.6	1.4	1.4
SP/DP peak performance [TFlops/s]	4.91/2.45	4.30/2.15	4.30/2.15
Memory capacity [GB]	48	48	24
Memory throughput [TB/s]	1.20	1.20	0.75

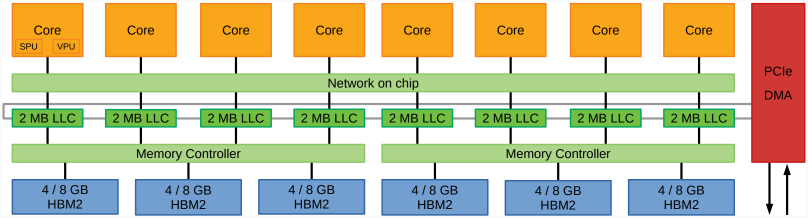
- Multiple platforms are available, ranging from workstation to supercomputer
- Up to 64 vector engines interconnected by InfiniBand fit into one NEC A500 rack, delivering 157 TFlops/s peak in double precision



© by NEC

Vector engine architecture

- Novel vector architecture with vector registers of 16 kbit size each
- Vector processor consists of 8 single-thread out-of-order cores, each with a
  - scalar processing unit (SPU) with L1 and L2 caches
  - vector processing unit (VPU), which processes the (optionally masked) 16-kbit vector registers in 8 chunks of 2 kbits each
  - coherent last-level cache (LLC) directly accessible by the VPU
- 6 HBM2 stacks in-package
- 2d mesh network on chip for memory access
- Ring bus for direct memory access (DMA) and PCIe traffic



Grid on the NEC SX-Aurora Tsubasa

- Extending Grid to registers with $2^n \times 128$ bits
  - Due to implementation of shift/stencil operations, current maximum SIMD layout in Grid is {2,2,2,1} (sites per dimension, complex single precision)
  - For SX-Aurora we need, e.g., {4,4,4,4} $\rightarrow$ Implement shift/stencil with generalized transformation, replacing original implementation
  - Implementation should be 100% SIMD efficient
- Illustration: shift on 2d lattice with real numbers
  - Lattice volume is {8,4}, SIMD width is 8 real numbers with layout {4,2}
  - Data layout of lattice

0	8	**1**	9	**2**	10	**3**	11
16	24	17	25	18	26	19	27
4	12	**5**	13	**6**	14	**7**	15
20	28	21	29	22	30	23	31

Shift 2 to the left $\longrightarrow$

1	9	**2**	10	**3**	11	**0**	8
17	25	18	26	19	27	16	24
5	13	**6**	14	**7**	15	**4**	12
21	29	22	30	23	31	20	28

  - Data layout in memory

[0 1 2 3 4 5 6 7] $\longrightarrow$ **[1 2 3 0 | 5 6 7 4]**

  - Need (same) permutations of partial vectors

SX-Aurora details

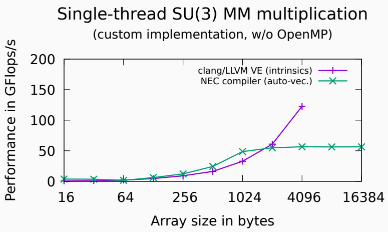
- C/C++ compiler capabilities

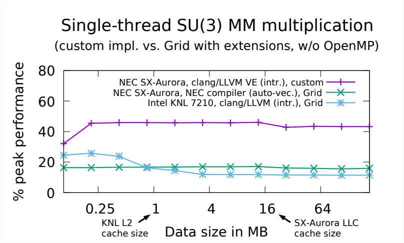
Compiler	Auto-vec.	Intrinsics	Notes
NEC clang/LLVM VE	✗	✓	open source, alpha stage
NEC ncc	✓	✗	closed source

  - Current focus on generic C/C++ and on auto-vectorization using ncc
  - Future: implement intrinsics
    (we have contacts to NEC team responsible for clang/LLVM VE)
- Complex numbers
  - Grid stores complex numbers as array of two-element structures (re, im)
    - Complex multiplication then needs intra-register permutations
  - SX-Aurora supports strided load instruction, which is applied twice to load re and im parts into separate registers (strided store analoguous)
    - Complex multiplication then no longer needs intra-register permutations
  - Auto-vectorization yields strided load/store

Optimal array size

- Test case: SU(3) MMM on SX-Aurora Type 10B and Intel KNL





- Optimal array size in Grid is 4 kB due to strided load/store (register size is 2 kB)
- Custom implementation of SU(3) MMM using intrinsics with strided load/store outperforms auto-vectorization of SU(3) MMM in Grid
  - Using clang, strided load/store requires explicit calls to intrinsics
- Auto-vectorization by ncc performs poorly due to sub-optimal register allocation (excessive load/store operations)

Status of porting Grid to SX-Aurora

- Shift/stencil operations are generalized to support $2^n \times 128$-bit arrays and auto-vectorization by ncc yields good performance
- ncc has matured and some issues have been resolved recently, e.g., issues compiling Grid with OpenMP support
  - Grid tests and benchmarks functionally correct using full vector length (4 kB arrays) and OpenMP on 8 cores
- clang introduces function call overhead compiling with OpenMP, thereby limiting performance
- MPI communications (ongoing work)
- Intrinsics implementation (future work)
- Optimization of LQCD operators (future work)
- GitHub: https://github.com/benjaminhuth/Grid/tree/feature/sx-aurora-0.8.2

ARM Scalable Vector Extension (SVE)

SVE toolchain

- C/C++ compiler capabilities

compiler	Auto-vec. VLA	Auto-vec. fixed-size	Intrinsics
armclang 19	✓	✗	✓
(Arm) gcc 8	✓	✓	✗
Fujitsu fccpx 4.0.0	✓	✓	[redacted]

  - We currently focus on armclang VLA compiler (research contract with Arm UK; direct contact to armclang and Arm gcc compiler developers)
  - We continue to stimulate SVE compiler support for fixed-size intrinsic vector data types
  - Fujitsu compiler [redacted]

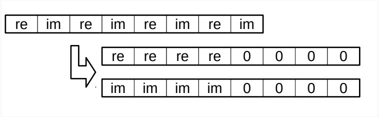
- Auto-vectorization of Grid for 512-bit SVE
  - armclang introduces VLA overhead and SIMD efficiency is limited
  - (Arm) gcc 8 did not compile Grid due to ambiguous function calls
    - Fixed in Grid just recently, initial tests indicate poor performance
  - Fujitsu compiler [redacted]. [redacted]

History: cmult using armclang 18 VLA compiler

- Test case: auto-vectorization of complex multiplication
- Array of std::complex, layout: $(\mathrm{re}_1, \mathrm{im}_1, \mathrm{re}_2, \mathrm{im}_2, \ldots)$

```
constexpr int n = COMPLEX_DOUBLES_FITTING_IN_VECTOR_REGISTER;
std::complex<double> x[n], y[n], z[n];
for(int i = 0; i != n; ++i)
    z[i] = x[i] * y[i];
```

- Massive overhead, branch is never taken
  - VLA not sensible for Grid, need workaround suitable for armclang VLA compiler
- No FCMLA
- Structure load/store
  - real arithmetics, no permutations
  - only 50% SIMD efficiency



```
    mov     x8, xzr
    whilelo p0.d, xzr, x0
    ptrue   p1.d
.LOOP
    lsl     x9, x8, #1
    ld2d    {z0.d, z1.d}, p0/z, [...]
    ld2d    {z2.d, z3.d}, p0/z, [...]
    incd    x8
    whilelo p2.d, x8, x0
    fmul    z4.d, z2.d, z1.d
    fmul    z5.d, z3.d, z1.d
    movprfx z7, z4
    fmla    z7.d, p1/m, z3.d, z0.d
    movprfx z6, z5
    fnmls   z6.d, p1/m, z2.d, z0.d
    st2d    {z6.d, z7.d}, p0, [...]
    brkns   p2.b, p1/z, p0.b, p2.b
    mov     p0.b, p2.b
    b.mi    .LOOP
```

Porting Grid to SVE ACLE

- Implementation details of Grid (example: AVX-512)
  - Data layout determined by SIMD width using `sizeof(SIMD data type)`
  - Architecture-specific data types as function arguments and return type
  - Load/store generated by compiler
  - Example: cmult using permutations and real arithmetics (DP)

```
struct MultComplex {
    inline __m512d operator()(__m512d a, __m512d b) {

        __m512d tmp    = _mm512_permute_pd(b, 0x55)
        __m512d a_real = _mm512_shuffle_pd(a, a, 0x00);
        __m512d a_imag = _mm512_shuffle_pd(a, a, 0xFF);
        a_imag = _mm512_mul_pd(a_imag, tmp);
        __m512d out    = _mm512_fmaddsub_pd(a_real, b, a_imag);

        return out;
    }
};
```

- SVE ACLE: we must respect restrictions on usage of sizeless types
  - Sizeless types allowed in member function headers, bodies and as return types, but we need explicit entry/exit points to/from SIMD land (compiler-generated loads and stores not feasible for Grid)
  - Sizeless types not allowed as member data of classes/structures/unions
  - `sizeof()` not applicable to sizeless types

Porting Grid to SVE ACLE

- Grid SVE ACLE implementation
  - Define SIMD width at compile time
  - Ordinary C-arrays of size = SIMD width as member data
  - Sizeless types only in member function bodies
  - Explicit loads and stores for well-defined entry/exit points to/from SIMD land
  - Compatible with armclang compiler
- Example: complex multiplication using FCMLA (SP + DP)

```
template <typename T>
struct vec {
    alignas(SVE_VECTOR_SIZE) T v[SVE_VECTOR_SIZE / sizeof(T)];
};

struct MultComplex {
    template <typename T>
    inline vec<T> operator()(const vec<T> &x, const vec<T> &y) {

        svbool_t pg1 = acle<T>::pg1();
        typename acle<T>::vt x_v = svld1(pg1, x.v);
        typename acle<T>::vt y_v = svld1(pg1, y.v);

        typename acle<T>::vt z_v = acle<T>::zero();
        typename acle<T>::vt r_v;
        r_v = svcmla_x(pg1, z_v, x_v, y_v, 90);
        r_v = svcmla_x(pg1, r_v, x_v, y_v, 0);

        vec<T> out;
        svst1(pg1, out.v, r_v);
        return out;
    }
};
```

- After expansion of expression templates the compiler must remove superfluous instructions, e.g., load/store and predication

Porting options

- Relevant details of the A64FX architecture are still unknown
  - Performance signatures of SVE instructions
  - Performance signatures of the memory hierarchy
- SVE instruction set allows for 3 implementations for complex multiplication:
  1. Ordinary load/store (ld1/st1) and FCMLA
     (not supported by RIKEN Fugaku processor simulator)
  2. Ordinary load/store (ld1/st1), intra-register permutations and real arithmetics
  3. Structure load/store (ld2/st2) and real arithmetics
- Which instruction mix performs best?
- What is the best performance achievable on simulator/hardware?
- Auto-vectorization vs. SVE ACLE
- Future: sizeless vs. fixed-size data types

Status of porting Grid to SVE ACLE

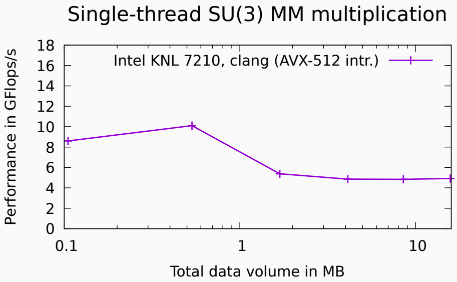
- 512-bit SVE support using 512-bit arrays
  - Ordinary load/store (ld1/st1) and FCMLA, 100% SIMD efficiency: done
  - Ordinary load/store (ld1/st1), permutations and real arithmetics for complex multiplication, 100% SIMD efficiency: done
  - Structure load/store (ld2/st2), real arithmetics for complex multiplication, 50% SIMD efficiency, mixing of ordinary and structure load/store otherwise: done, but mixing of SVE ACLE data types might not be good
  - All implementations compile using armclang, verified using ArmIE and qemu
  - Further optimizations to be applied once hardware is available
  - GitHub: https://github.com/nmeyer-ur/Grid/tree/feature/arm-sve

- Options for future work
  - 512-bit SVE using 1024-bit arrays
    - Grid extension for NEC SX-Aurora serves as a template
    - 100% SIMD efficiency with structure load/store (ld2/st2)
  - Implement fixed-size data types
    - Compiler-generated load/store (hopefully)
    - Potential fix for excessive copy operations
  - [redacted] Fujitsu compiler: [redacted]
  - Optimization of LQCD operators

RIKEN Fugaku Processor Simulator and A64FX Prototype Benchmarks

Grid benchmark

- Test case: Grid SU(3) MMM AVX-512 on Intel KNL 7210 vs. 512-bit SVE in RIKEN Fugaku processor simulator
  - clang 5.0 (AVX-512 intrinsics)
  - armclang 19.2 (512-bit arrays, SVE ACLE intrinsics)
  - fccpx auto-vectorization (512-bit arrays)



- Intel KNL [redacted] RIKEN Fugaku processor simulator
  - ld1/st1 [redacted] [redacted] [redacted] [redacted] [redacted] 100% SIMD efficiency
  - ld2/st2 [redacted] [redacted] [redacted] [redacted] [redacted] 50% SIMD efficiency
  - fccpx [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted]

Schönauer vector triad micro-benchmark

- Schönauer vector triad

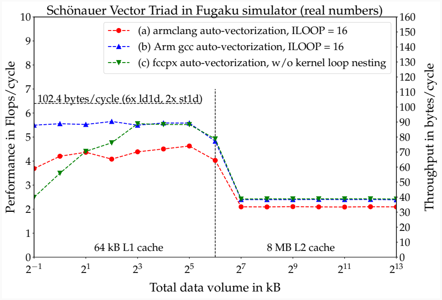
```
for(i=0; i<R; i++)          // R = number of repetitions
    for(j=0; j<N; j+=ILOOP) // vector triad kernel
        for(k=j; k<j+ILOOP; k++)
            A[k] = B[k] + C[k] * D[k];
```

- We test 64-bit `double` and 128-bit `double _Complex` (two-element structure of real and imaginary part)
- 1 addition and 1 multiplication (real), or 4 additions and 4 multiplications (complex) per triad
- 4 or 8 triads computed in parallel using 512-bit vectors
- 3 loads and 1 store per vector triad
- Peak performance is 32 Flops/cycle (dual-issue FMA, DP)

→ Poster session

Schönauer vector triad: real numbers

RIKEN Fugaku processor simulator



A64FX prototype

- armclang auto-vectorization generates VLA loop, performs poorly
- Arm gcc and fccpx auto-vectorization perform loop unrolling
- RIKEN Fugaku processor simulator and A64FX prototype performance [redacted]
  - Achieves up to about 90 bytes/cycle in simulator (5.5 Flops/cycle, 17% peak)
  - Achieves up to about [redacted] on prototype [redacted]

Simple pipeline model: real numbers

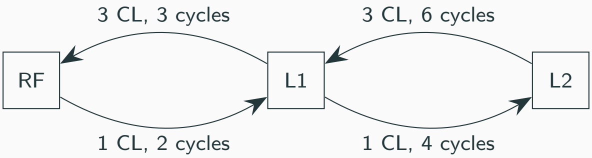
- Minimal A64FX pipeline model: 2 load ports, 1 store port, 2 FPUs (FP0, FP1)
  - $2 \cdot 512$-bit load xor $1 \cdot 512$-bit store from/to L1 per cycle
  - Loads take precedence over stores, no limitations otherwise
  - Assume counters, branches and prefix operations (FMA4) are hidden
  - Assume latency of 1 clock cycle for each instruction
- ld1/st1 for load/store of 512-bit vector operands B_i, C_i, D_i and A_i

Cycle	LOAD0	LOAD1	STORE	FP0	FP1
1	B_1	B_2			
2	C_1	C_2			
3	D_1	D_2			
4	B_3	B_4		$A_1 \leftarrow B_1 + C_1 \cdot D_1$	
5	C_3	C_4		$A_2 \leftarrow B_2 + C_2 \cdot D_2$	
6	D_3	D_4			
7			A_1	$A_3 \leftarrow B_3 + C_3 \cdot D_3$	
8			A_2	$A_4 \leftarrow B_4 + C_4 \cdot D_4$	
9			A_3		
10			A_4		

- Vector triad performance limited by load/store throughput
  - 4 vector triads saturate load/store ports
  - $12 \cdot 512$-bit loads + $4 \cdot 512$-bit stores in 10 cycles: 102.4 bytes/cycle
  - $4 \cdot 8$ FMA (DP) in 10 cycles: 6.4 Flops/cycle (20% peak)

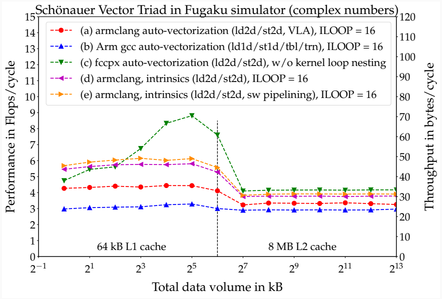
Cache access models: real numbers

- RF – L1 (RF = register file)
  - A64FX core: 2 · 512-bit load xor 1 · 512-bit store from/to L1 per cycle
  - Expected peak throughput RF – L1 for Schönauer vector triad is 102.4 bytes/cycle, assuming optimal use of load and store ports
- RF – L1 – L2
  - Assume L1 misses for all loads
  - Assume exclusive access to L1 by either RF or L2
    $\rightarrow$ Cache line (CL) transfers between RF, L1 and L2 caches proceed sequentially
  - 12 ordinary loads (`ld1d`) of 64 bytes each (6 cache lines) from L2 via L1 to RF take 12 + 6 cycles, 4 ordinary stores (`st1d`, 1 cache line) from RF to L1 take 4 cycles, and eviction to L2 takes 8 cycles
  - Peak throughput is 16 · 64 bytes/30 cycles = 34.1 bytes/cycle
  - Model predicts peak throughput in good approximation


$$\begin{tikzcd}
\boxed{RF} \arrow[r, bend right, "\text{1 CL, 2 cycles}"'] & \boxed{L1} \arrow[l, bend right, "\text{3 CL, 3 cycles}"'] \arrow[r, bend right, "\text{1 CL, 4 cycles}"'] & \boxed{L2} \arrow[l, bend right, "\text{3 CL, 6 cycles}"']
\end{tikzcd}$$

Schönauer vector triad: complex numbers

RIKEN Fugaku processor simulator



A64FX prototype

- armclang auto-vec. generates structure load/store (ld2/st2), performs poorly
- Arm gcc auto-vec. generates ordinary load/store (ld1/st1) and perm., performs poorly
- fccpx auto-vec. generates structure load/store (ld2/st2), loop unrolling and schedules the instructions (software pipelining), performs best
- Intrinsics using structure load/store and two-fold loop unrolling
  - Software pipelining outperforms naive loop unrolling
- RIKEN Fugaku processor simulator and A64FX prototype [redacted] [redacted]
  - Achieves up to about 70 bytes/cycle in simulator (9 Flops/cycle, 30% peak)
  - Achieves up to about [redacted] [redacted] on prototype [redacted] [redacted] [redacted]
  - No adequate pipeline model available due to unknown instruction characteristics

Simple pipeline model: complex numbers

- Minimal A64FX pipeline model: 2 load ports, 1 store port, 2 FPUs (FP0, FP1)
  - $2 \cdot 512$-bit load xor $1 \cdot 512$-bit store from/to L1 per cycle
  - Loads take precedence over stores, no limitations otherwise
  - Assume counters and branches are hidden
  - Assume latency of 1 clock cycle for each instruction
- ld2/st2 for load/store of 512-bit vector operands re/im(B_i), same for C_i, D_i and A_i

Cycle	LOAD0	LOAD1	STORE	FP0	FP1
1	C_1^{re}	C_1^{im}			
2	D_1^{re}	D_1^{im}			
3	B_1^{re}	B_1^{im}		$A_1^{re} \leftarrow C_1^{re} \cdot D_1^{re}$	$A_1^{im} \leftarrow C_1^{re} \cdot D_1^{im}$
4	C_2^{re}	C_2^{im}		$A_1^{re} \leftarrow A_1^{re} - C_1^{im} \cdot D_1^{im}$	$A_1^{im} \leftarrow A_1^{im} + C_1^{im} \cdot D_1^{re}$
5	D_2^{re}	D_2^{im}		$A_1^{re} \leftarrow A_1^{re} + B_1^{re}$	$A_1^{im} \leftarrow A_1^{im} + B_1^{im}$
6	B_2^{re}	B_2^{im}		$A_2^{re} \leftarrow C_2^{re} \cdot D_2^{re}$	$A_2^{im} \leftarrow C_2^{re} \cdot D_2^{im}$
7			A_1^{re}	$A_2^{re} \leftarrow A_2^{re} - C_2^{im} \cdot D_2^{im}$	$A_2^{im} \leftarrow A_2^{im} + C_2^{im} \cdot D_2^{re}$
8			A_1^{im}	$A_2^{re} \leftarrow A_2^{re} + B_2^{re}$	$A_2^{im} \leftarrow A_2^{im} + B_2^{im}$
9			A_2^{re}		
10			A_2^{im}		

- Vector triad performance limited by load/store throughput
  - 2 vector triads saturate load/store ports
  - $12 \cdot 512$-bit loads $+ 4 \cdot 512$-bit stores in 10 cycles: 102.4 bytes/cycle
  - $4 \cdot 8$ MULT, $4 \cdot 8$ FMA and $4 \cdot 8$ ADD (DP) in 10 cycles: 12.8 Flops/cycle (40% peak)

Summary and Outlook

Summary and Outlook

- Good progress porting Grid to vector architectures
  - Enables support for registers with $2^n \times 128$ bits when done
- Diversity of compiler capabilities results in high development effort
- NEC SX-Aurora
  - Current focus is on auto-vectorization of 4 kB arrays using NEC compiler
  - Future work: intrinsics support using strided load/store
- Fujitsu A64FX: multiple implementation options feasible
  - Implementations using 512-bit arrays
    - Ordinary load/store and permutations done, but yields poor performance
    - Structure load/store done, but results in limited SIMD performance
  - We expect 1024-bit arrays and structure load/store to be most efficient
    - Grid extension for SX-Aurora serves as a template
  - Implementation of fixed-size data types reasonable once available
    - Compiler-generated load/store (hopefully)
    - Potential fix for excessive copy operations
- Future work: optimization of LQCD operators

The result of the RIKEN Post-K processor simulator is just an estimated value, and it does not guarantee the performance of the supercomputer Fugaku at the start of its operation. We use the processor simulator compiled on 11th of September 2019. The Fujitsu fccpx compiler is a pre-release version under development.

We acknowledge support from the HPC tools team at Arm.