FUJITSU

shaping tomorrow with you

Programming Guide (Processors)

V1.4

Aug 2021

FUJITSU LIMITED

Introduction

- This document puts together the information about the features and microarchitecture of the A64FX processor, as well as its basic performance data, from the viewpoint of application developers and tuning engineers.
- Refer to the following in conjunction with this document.
  - Fortran User’s Guide
  - C User’s Guide
  - C++ User’s Guide
  - Profiler User's Guide
  - Programming Guide(Programming common part)
  - Programming Guide(Tuning)
  - Programming Guide(Fortran)
- The following abbreviation is used in this document:
  - A64FX Logic Specifications
  - A64FX ® Microarchitecture Manual
  - ARM® Architecture Reference Manual (ARMv8 , ARMv8.1 , ARMv8.2 , ARMv8.3)
  - ARM® Architecture Reference Manual Supplement The Scalable Vector Extension

Trademarks

- Linux® is a trademark or registered trademark of Linus Torvalds in the United States and other countries.
- Red Hat is a trademark or registered trademark of Red Hat Inc. in the United States and other countries.
- ARM is a trademark or registered trademark of ARM Ltd. in the United States and other countries.
- Proper names such as the product name mentioned are trademark or registered trademark of each company.
- Trademark symbols such as ® and ™ may be omitted from system names and product names in this document.

Revision History

Version	Date	Details
1.1	May 14, 2020	- First published
1.2	Sep 30, 2020	- Correcting typographical errors and expressions by reviewing articles
1.3	Mar 31, 2021	- Correcting typographical errors and expressions by reviewing articles

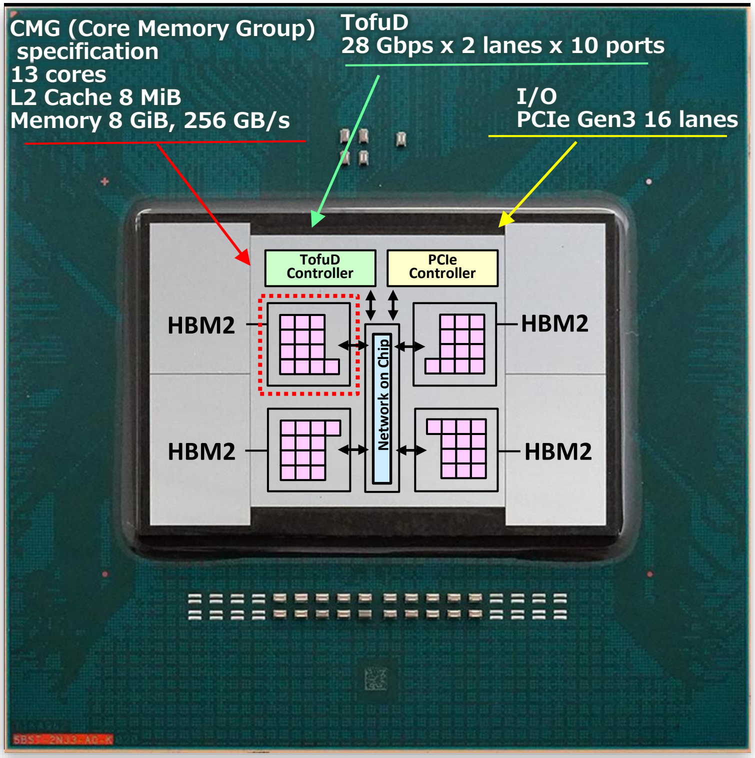
Contents

A64FX Processor Overview

- A64FX Processor Overview
- A64FX Processor Specifications
- Interconnect “Tofu Interconnect D“ Overview

A64FX Processor Overview (1/3)

- High-performance, high-efficiency CPU using Arm SVE
  - Double precision operation performance: 3.072 TFLOPS@2 GHz, 90+%@DGEMM
  - Memory bandwidth: 1024 GB/s, 80+%@STREAM Triad



CMG (Core Memory Group) specification
13 cores
L2 Cache 8 MiB
Memory 8 GiB, 256 GB/s

TofuD
28 Gbps x 2 lanes x 10 ports

I/O
PCIe Gen3 16 lanes

	A64FX
ISA (Base, extension)	Armv8.2-A, SVE
Process technology	7 nm
Double precision peak performance	3.072 TFLOPS@2 GHz
SIMD Width	512-bit 256-bit/128-bit supported as well
Number of cores	48 + 4
Memory size	32 GiB (HBM2 x4)
Memory bandwidth	1024 GB/s
PCIe	Gen3 16 lanes
Interconnect	TofuD(*) integrated

(*1) Tofu interconnect D

A64FX Processor Overview (2/3)

- The A64FX processor (hereinafter referred to as the A64FX) is designed for high performance computing (HPC). It is an out-of-order execution super-scalar processor compliant with the ARMv8-A profile architecture and the Scalable Vector Extension for ARMv8-A.
  The A64FX adopts several distinctive architectures for HPC.

  - Scalable Vector Extension

    The A64FX supports the Scalable Vector Extension (SVE), which is a vector extension of the ARM Instruction Set Architecture.

  - Core Memory Group

    The A64FX has groups called Core Memory Groups (CMGs) in it, each consisting of 13 processor cores, an independent L2 cache, and an independent memory controller.
    The processor has four CMGs and uses the Non-Uniform Memory Access (NUMA) architecture for inter-CMG access.

  - Sector cache

    This function virtually partitions the cache in units of ways, making it possible to specify the sector that can be used at the instruction level. A program can specify a sector by using tagged addresses.
    The L1 cache has two 4-partition groups, and the L2 cache has two 2-partition groups.

A64FX Processor Overview (3/3)

Hardware barrier

This function allows hardware to support synchronization between software processes or threads. It enables high-speed synchronization without any memory operation.
A hardware barrier is effective only within a CMG. Synchronization between CMGs is achieved using a software barrier.

Hardware prefetch assist

This function allows a program to control the behavior of hardware prefetch. The program can give information to the hardware prefetch mechanism using the system register and tagged addresses.

High Bandwidth Memory

High Bandwidth Memory Gen2 (HBM2) is used as the main memory to provide high memory bandwidth.

A64FX Specifications

FUJITSU

Item	Specification
Number of processor cores	52 (13 cores / CMG)
Number of CMGs	4
L1I cache size	64KiB / 4way
L1D cache size	64KiB / 4way
L2 cache size	32MiB / 16way (8MiB / CMG)
Cache line size	256B
Memory size	32GiB(8GiB/CMG)
Interconnect	Tofu Interconnect D
I/O	PCI-Express Gen3 16 Lanes
Instruction set architecture	ARMv8-A, ARMv8.1, ARMv8.2, ARMv8.3 (*1), SVE

(*1) ARMv8.3 supports only complex number support instructions.

A64FX Specifications : Bandwidth and Latency

		A64FX	Remarks
Frequency [GHz]		2.0	
Number of CPUs/node		1	
Number of computing cores/node		48	
Number of CMGs/node		4	
Memory size/node [GiB]		32	
Cache size	L1 [KiB/core]	64(instruction)+64(data)	
	L2 [MiB/CMG]	8	Note: In the case of a node with an assistant core, an application is considered to use 7 MB/14 ways.
Cache latency [cycle]	L1	5 (EX, short) 8 (FL, short) 11 (FL, long)	
	L2	37 to 47	Average 42
Cache bandwidth [B/cycle/core]	L1	When hit: 128	
	L2	42.7	
Operation performance per node (per core) [GFlops]	Double precision Single precision Half precision	3072 (64) 6144 (128) 12288 (256)	
Basic operation latency [cycle]	Integer add instruction Integer mult instruction FMA instruction	1 5 9	
Memory latency [ns]		150	
Theoretical memory bandwidth per node (per CMG) [GB/s]		1024(256)	
Inter-CMG bandwidth		128 GB/s × 2 (two ways)	

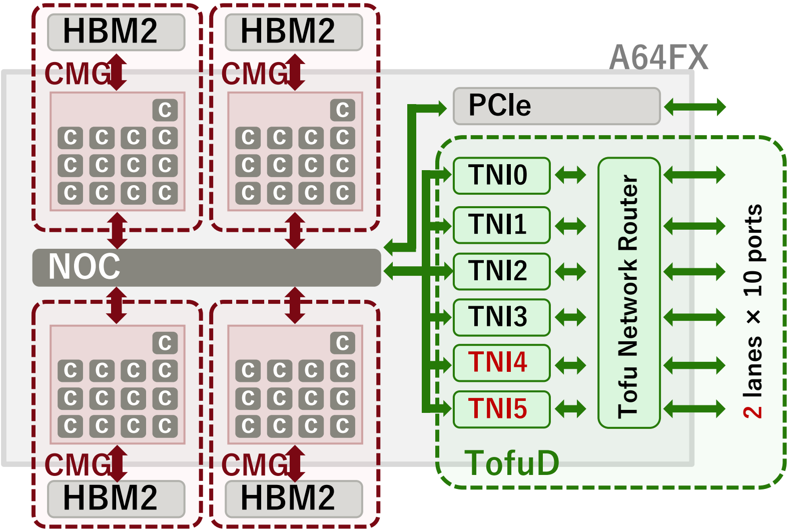
A64FX Specifications : Others

	A64FX	Remarks
Maximum number of decodes per cycle	4	
Hardware prefetch queue	16	

Interconnect “Tofu Interconnect D“(TofuD) Overview

- An interconnect controller is integrated in the CPU.
  - The number of TNIs has been increased to achieve higher injection bandwidth and flexible communication patterns.
  - The barrier resources have also been increased to enable the implementation of flexible collective communication algorithms.
- The memory bypass technology enables low-latency communication.
  - Direct descriptor and cache injection

	TofuD spec
Data rate	28.05 Gbps
Link bandwidth	6.8 GB/s
Injection bandwidth	40.8 GB/s
	Measured
Put throughput	6.35 GB/s
PingPong latency	0.49 to 0.54 μs



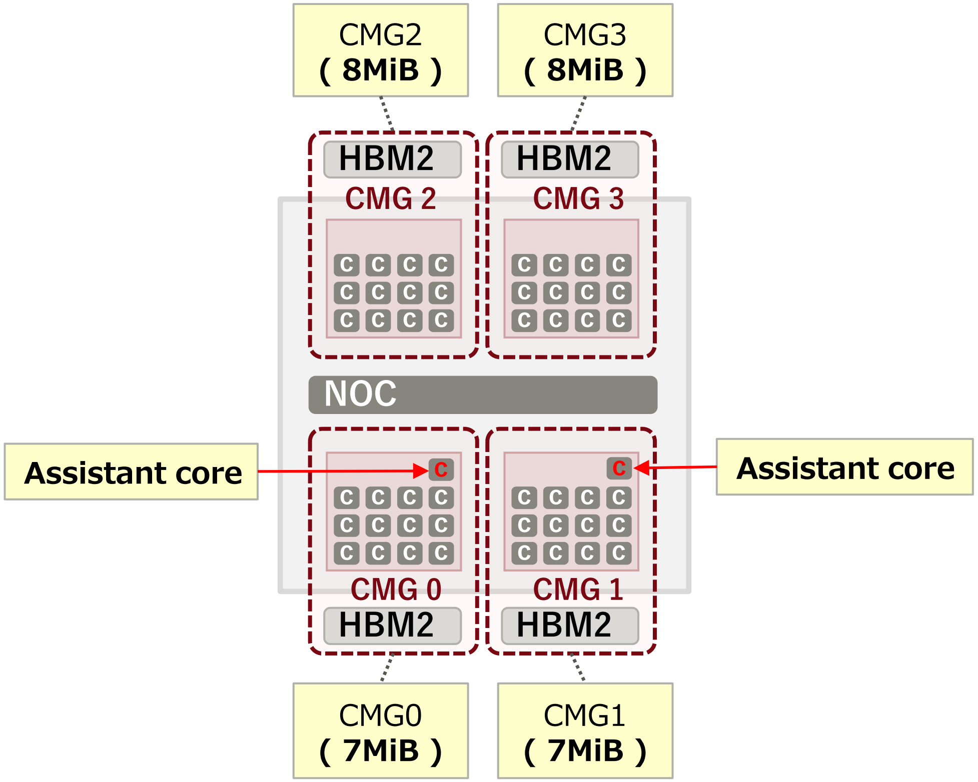
L2 Cache Size Available to Applications

- L2 Cache Size Available to Applications
- Verification: Performance Based on the Presence of an Assistant Core

L2 Cache Size Available to Applications

- In a CMG including an assistant core, part of the L2 cache (two ways = 1 MiB) is used for the assistant core.
  Therefore, when a CMG includes an assistant core, the space of the L2 cache available to a user program is 7 MiB.

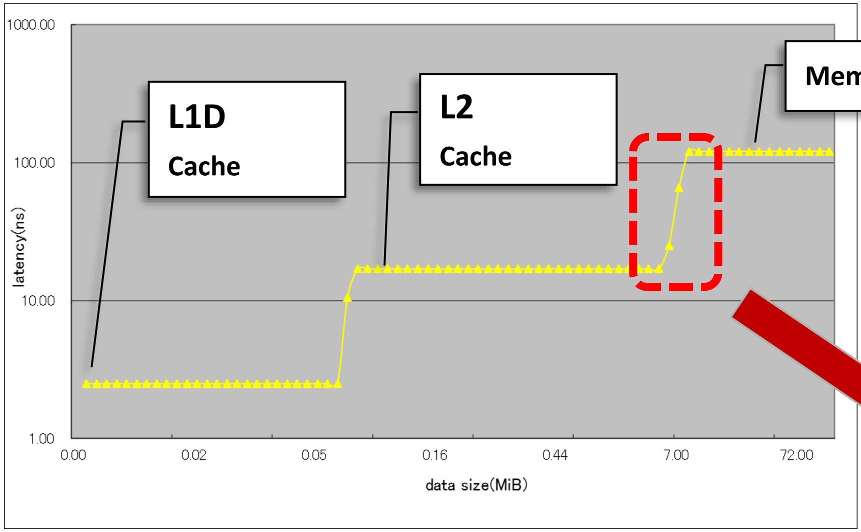
In case of a computing node



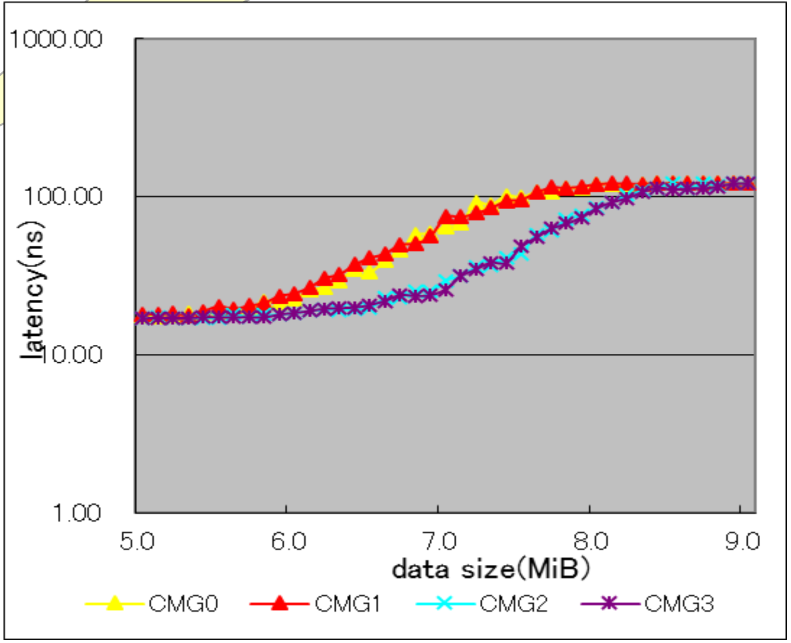
Verification: Performance Based on the Presence of an Assistant Core

Imbench performance (2.0 GHz)

- Results of measuring data access latency using CMG0 Imbench (integer access)



The data size (5 to 9 MiB) at which the change from L2 cache access to memory access occurred was obtained for each CMG.



The change from L2 cache access to memory access occurs more quickly in the CMGs that include assistant cores (CMG0 and CMG1).

- Under the CPU specifications, the L2 cache size per CMG is 8 MiB. However, the cache size must be considered 7 MiB when the application is running.
- Care needs to be exercised in cases where tuning is performed, such as when blocking is done taking the L2 cache size into consideration. (The reference value of L2 cache size is slightly less than 7 MiB.)

Microarchitecture

Prefetch

- About Prefetch
- Hardware Prefetch Operation
- Prefetch Distance Setting Command
- Prefetch Distance Verification
- Hardware Prefetch Evaluation Using an Actual Application (NICAM)
- Prefetch Instructions Provided by the A64FX
- Prefetch Collaboration Between Hardware and Software
- Prefetch Collaboration Verification

About Prefetch (1/2)

- The A64FX supports the following new functions as hardware and software prefetch:
  - Hardware prefetch (HWPF)
    - Hardware prefetch distance setting function
    - Hardware stride prefetch function
  - Software prefetch (SWPF)
    - Automatic adjustment of the prefetch distance
    - SVE Gather prefetch instruction support

- By using these prefetch functions, you can mask data access latency to speed up application execution. (Latency masking)

About Prefetch (2/2)

- Prefetch distance
  Hardware prefetch and software prefetch perform data prefetching on the lines ahead as indicated below.

	Hardware Prefetch		Software Prefetch	
	L1 Prefetch	L2 Prefetch	L1 Prefetch	L2 Prefetch
FX100	2 lines	Up to 16 lines	3 lines	15 lines
A64FX	**Up to 6 lines**	**Up to 40 lines**	**Automatic**	**Automatic**

The hardware prefetch distance can also be set by users.

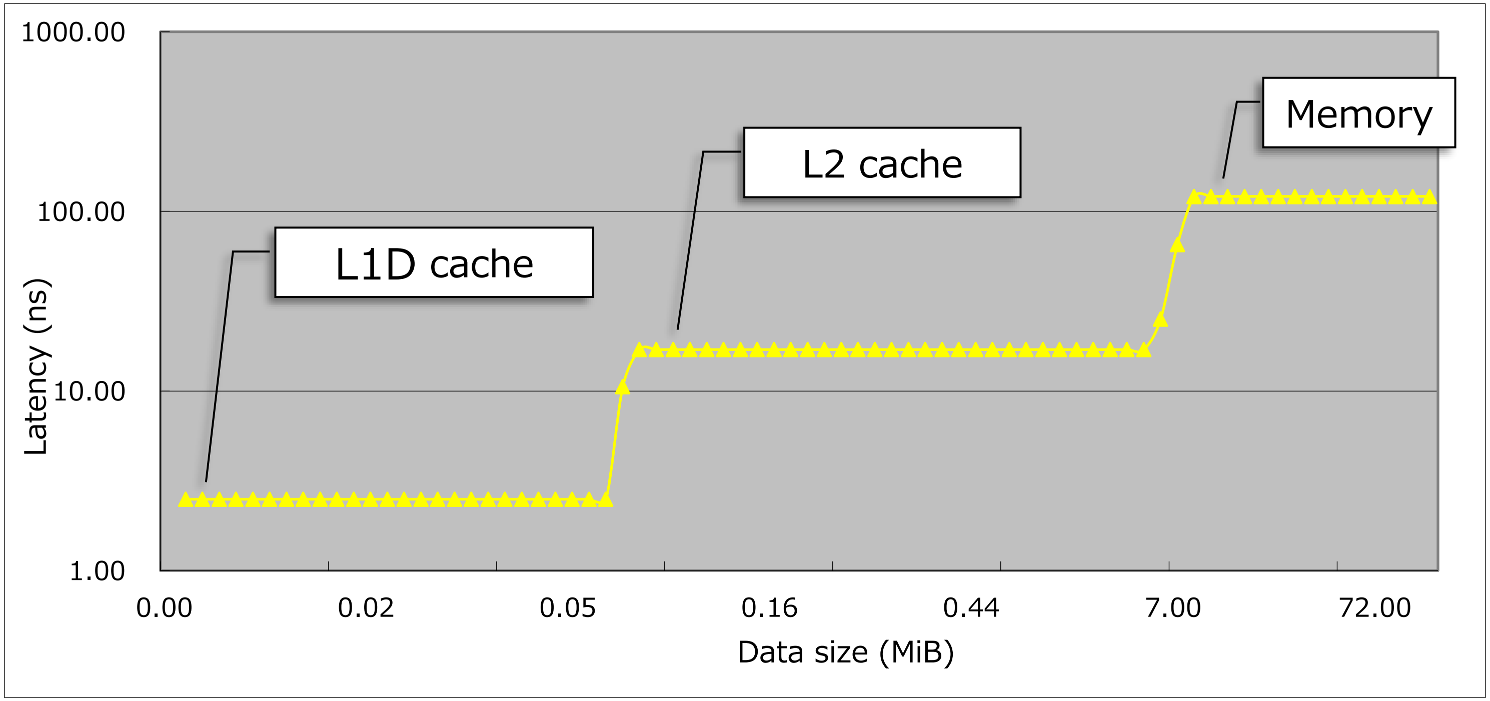
The distance is automatically adjusted for software prefetch.

- When tuning loop blocking or outer loop unrolling, be aware that the hardware prefetch may not work if the inner loop length decreases.

[Reference] What is Latency Masking?

- Latency masking hides data access latency (period from a data transmission request until a response returns) by using prefetch. There are three types of data access: L1D cache, L2 cache, and memory. The prefetch target for latency masking is the L2 cache and memory.

- Data access latency measurement results from LMbench (at integer access)



Hardware Prefetch Operation

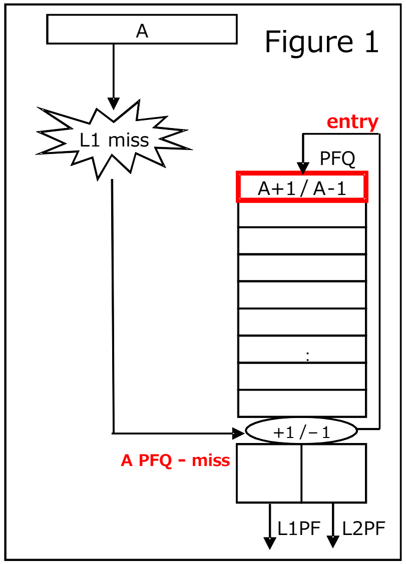
Hardware prefetch conditions

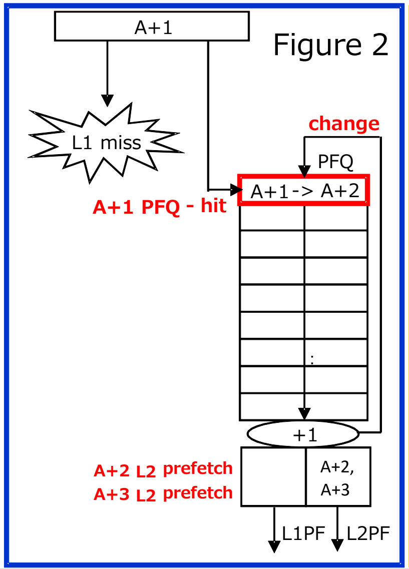
- The prefetch is triggered by missing L1.
- The L1 miss continues to be repeated on a cache line basis.

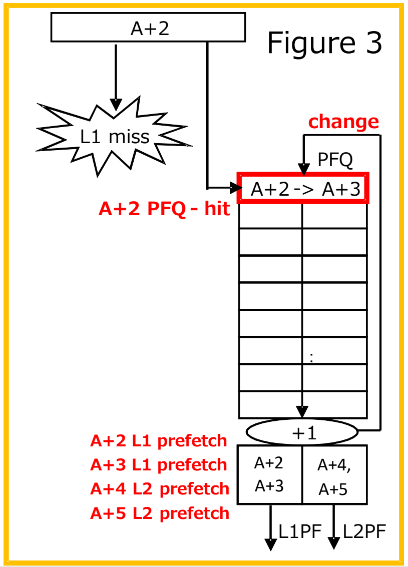
Operation algorithm of hardware prefetch

1. If cache line A is missed, line A+1/A-1 is newly registered in a 16-entry FIFO (queue) called a PFQ (prefetch queue). (Figure 1)
2. When subsequent access is made to line A+1 and hits A+1 registered in the PFQ, it is considered ascending order stream access. From this point on, HWPF begins in the ascending order direction. Also, A+1 in the PFQ is updated to A+2. (Figure 2)
3. After the above stream access is detected, the L2 prefetch is extended up to 40 lines ahead by prefetching the data of two lines at a time. In the A64FX, the L1 prefetch is also extended up to six lines ahead by prefetching the data of two lines at a time, as in the L2 prefetch. (Figure 3)

- Overview of prefetch







Note) The numbers in the table represent lines. In byte notation, 1 line must be regarded as 256 bytes.

- Correspondence between PFQ and prefetch addresses

PFQ-hit	L2HWPF-ADRS	L1PHWF-ADRS
A+ 1	A+ 2, A+ 3	
A+ 2	A+ 4, A+ 5	A+ 2, A+ 3
A+ 3	A+ 6, A+ 7	A+ 4, A+ 5
A+ 4	A+ 8, A+ 9	A+ 6, A+ 7
A+ 5	A+10, A+11	A+ 8, A+ 9
A+ 6	A+12, A+13	A+10
A+ 7	A+14, A+15	A+11
A+ 8	A+16, A+17	A+12
A+ 9	A+18, A+19	A+13
A+10	A+20	A+14
A+11	A+21	A+15
A+12	A+22	A+16
A+13	A+23	A+17
A+14	A+24	A+18
A+15	A+25	A+19

* L2PF = 10 lines ahead, L1PF = 4 line ahead

Hardware Prefetch Distance Setting Command

Provision of the hardware prefetch distance setting command (hwpfctl)

Item	Description

Format	hwpfctl [--disableL1] [--disableL2] [--distL1 lines_l1] [--distL2 lines_l2] [--weakL1] [--weakL2] [--verbose] command {arguments ...}
	hwpfctl --default [--verbose] command {arguments ...}
	hwpfctl --reset [--verbose]
	hwpfctl –help

Explanation	The hwpfctl command changes the behavior of hardware prefetch (stream detect mode) provided in the A64FX. The CPU core to be changed by this command is determined by process affinity.
Option(s)	--disableL1 --disableL2 Disables hardware prefetch for the L1/L2 cache. If these options are omitted, hardware prefetch is enabled. --distL1=lines_l1 --distL2=lines_l2 Specify the lines of the L1/L2 cache to be prefetched, by using the number of cache lines counted from the missed cache line. In lines_l1, you can specify a value from 1 to 15 as the number of lines of the L1 cache to be prefetched. Likewise, in lines_l2, you can specify a value from 1 to 60 as the number of lines of the L2 cache to be prefetched. Note that the value specified in lines_l2 is rounded up to a multiple of 4 when written to the system register. If you specify 0, the command behaves assuming the default value of the CPU. If these options are omitted or an invalid value is specified, 0 is assumed. --weakL1 --weakL2 Specify that the priority of the L1/L2 cache prefetch request is weak. If these options are omitted, the priority is strong. --default Starts the command using the default settings. The options other than --verbose are ignored. --reset Initializes the system register values. The options other than --verbose are ignored. --verbose Outputs the values before and after the system register is changed. --help Shows how to use this command.

Use example of the hardware prefetch distance setting command (hwpfctl)

```
hwpfctl  –distL1=6  –distL2=40  a.out
```

Prefetch Distance Verification: Conditions

- Measurement conditions for prefetch performance evaluation

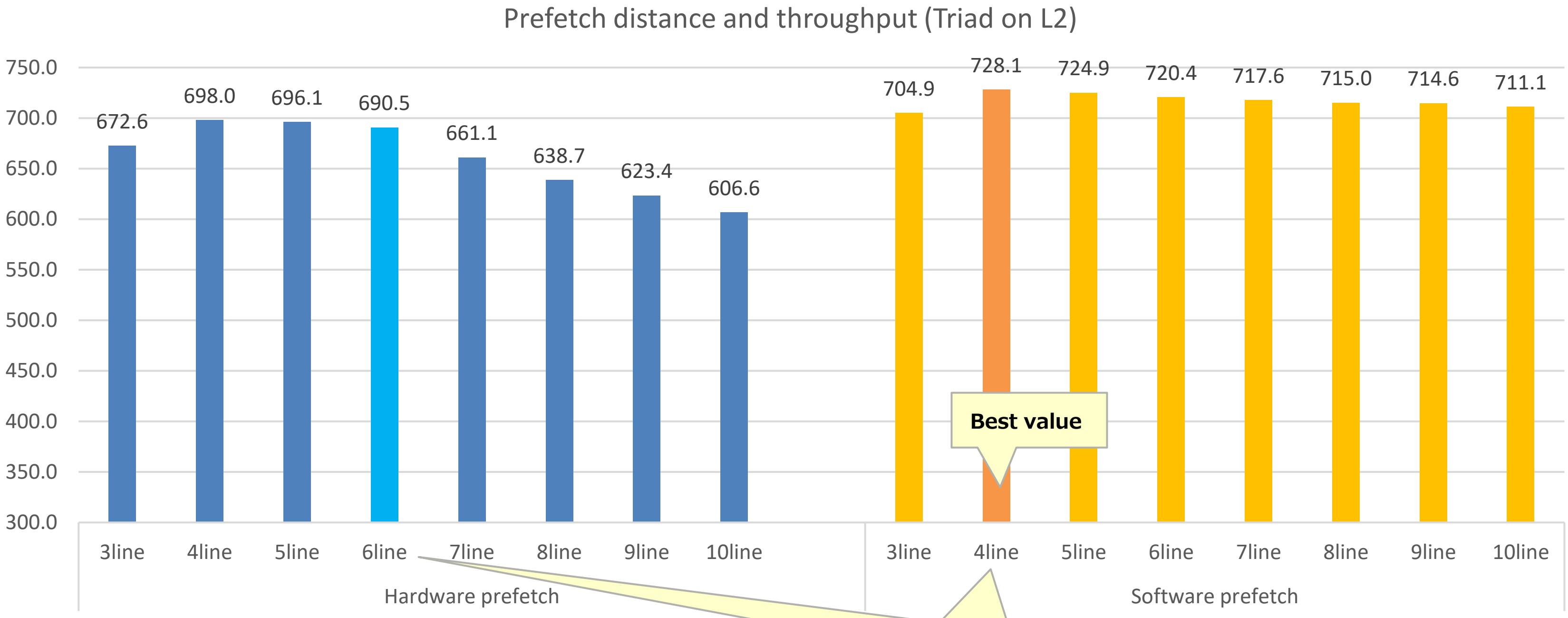
	Pattern
Verification code	- Triad L2 cache access (L1 prefetch distance evaluation) - Triad memory access (L2 prefetch distance evaluation)
Prefetch distance to be evaluated	Hardware prefetch distance and software prefetch distance (strong) - L1 prefetch: 3 to 10 lines ahead - L2 prefetch: 10, 15, 20, 25, 30, 35, or 40 lines ahead (In L1, the default is 6 for HWPF and automatic for SWPF (4 for Triad).)
Cores to be measured	12 cores (CMG0)
Translation option	-Kfast For software prefetch evaluation: -Kprefetch_sequential=soft -Kprefetch_line=? -Kprefetch_line_L2=?
Access range	The conditions for the Triad code evaluation are as follows. -- Double precision operation arrays must be used. -- bss must be used. -- Number of innermost loop iterations, array size (n) - L1 prefetch evaluation: 174720 (The total size of the arrays to be accessed is half the L2 cache size.) - L2 prefetch evaluation: 10485120 (The total size of the arrays to be accessed is 30 times the L2 cache size.) * Each array is 256 byte aligned. -- Number of outer loop iterations (iter) - L1 prefetch evaluation: 10000 - L2 prefetch evaluation: 3000

Prefetch Distance Verification: L1 Prefetch

- L1 prefetch distance evaluation using Triad (L2 cache access)

Triad

```
!$omp parallel
  Do j = 1, iter
!$omp do
     Do i = 1, n
        y(i)=x1(i) + c0 * x2(i)
     End Do
!$omp end do nowait
  End Do
!$omp end parallel
```



Best value

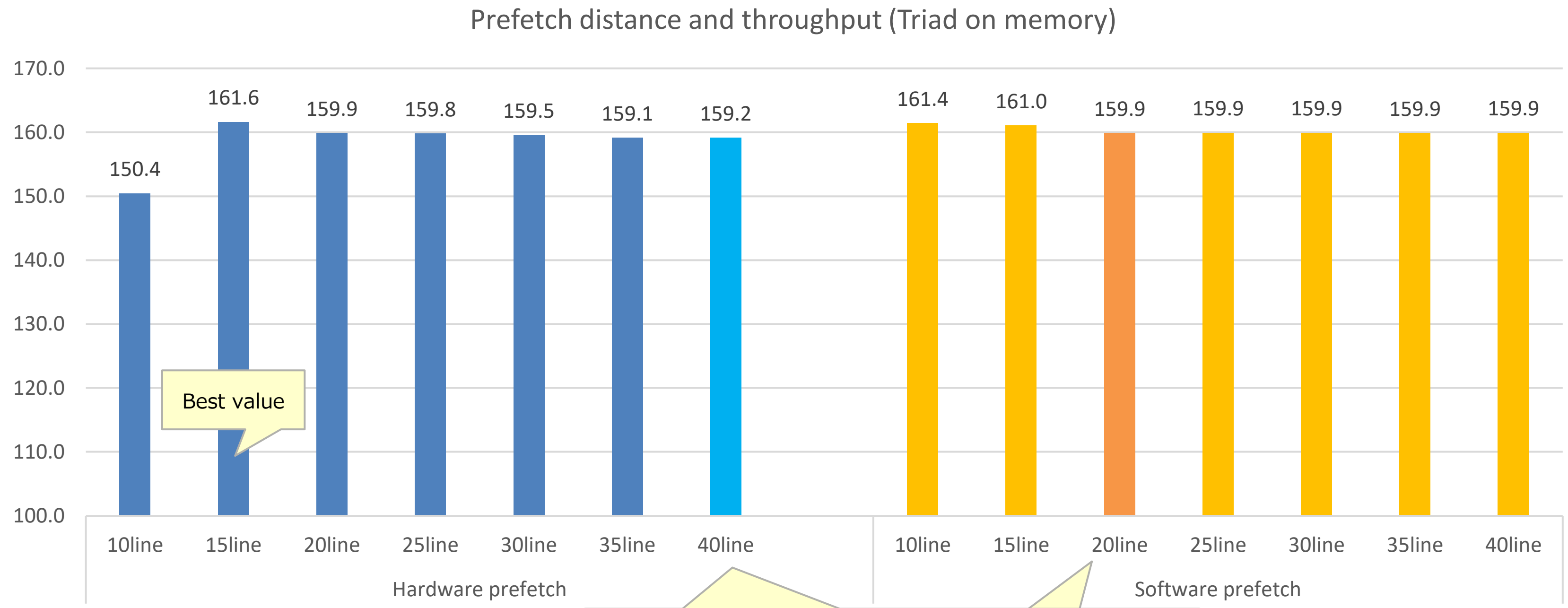
Performance when the distance is not specified

Prefetch Distance Verification: L2 Prefetch

- L2 prefetch distance evaluation using Triad (memory access)

Triad

```
!$omp parallel
  Do j = 1, iter
!$omp do
     Do i = 1, n
        y(i)=x1(i) + c0 * x2(i)
     End Do
!$omp end do nowait
  End Do
!$omp end parallel
```



Best value

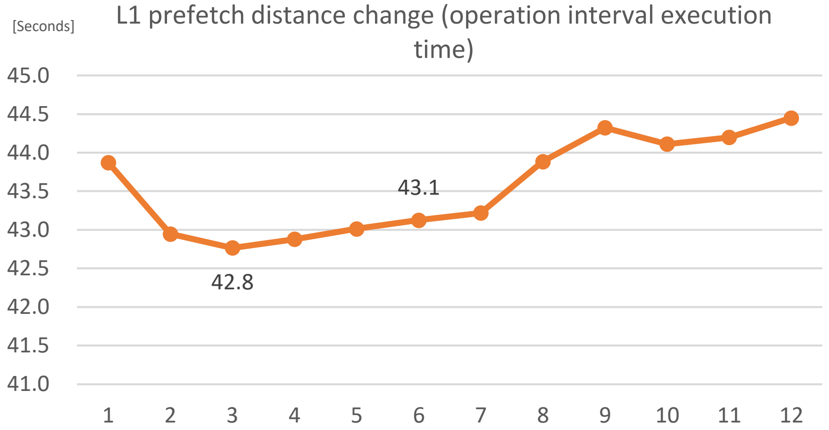
Performance when the distance is not specified

＊ The throughput does not include that of the data read from those cache lines to which data is written.

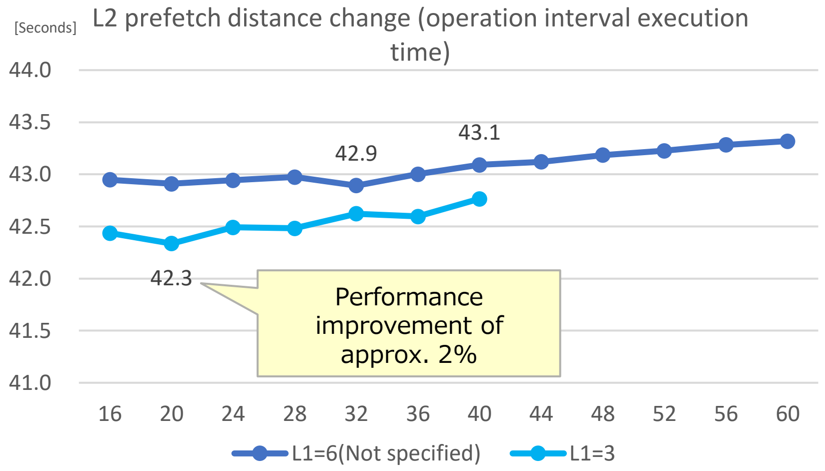
Hardware Prefetch Evaluation Using an Actual Application (NICAM)

- **Evaluation by hardware prefetch distance adjustment**
  The hardware prefetch distance adjustment function was used to do performance evaluation with an actual application (NICAM).

  - L1 prefetch distance evaluation
    As the L2 prefetch distance, the initial value (40) was used. The fastest value was obtained when L1 was 3.



  - L2 prefetch distance evaluation
    First, as the L1 prefetch distance, the initial value (6) was used. It was confirmed that high speed was achieved at short distances (16 to 32).
    Next, based on the L1 prefetch distance evaluation, measurement was conducted at short distances, with 3 specified as L1. An improvement of approx. 2% was confirmed.



Prefetch Instructions Provided by the A64FX (ISA)

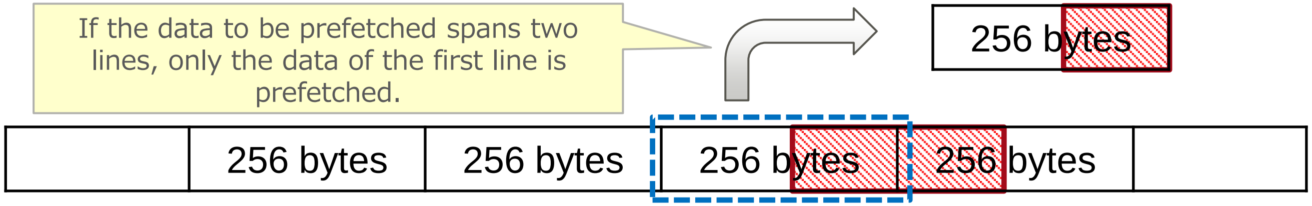
FUJITSU

Prefetch instructions provided by the A64FX

The (1) ARMv8 prefetch instruction and (2) SVE contiguous prefetch instruction are used for contiguous access.

① ARMv8 prefetch instruction

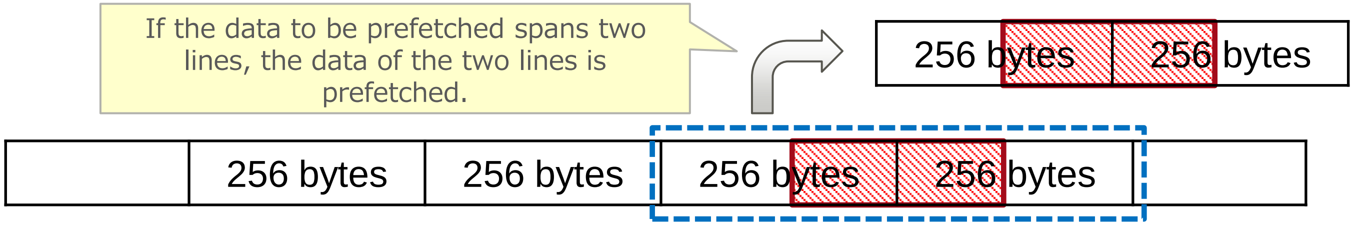
Prefetch instruction corresponding to the contiguous load / store instructions (without considering the prefetching across lines)



If the data to be prefetched spans two lines, only the data of the first line is prefetched.

② SVE contiguous prefetch instruction

Prefetch instruction corresponding to the contiguous load / store instructions (considering the prefetching across lines)

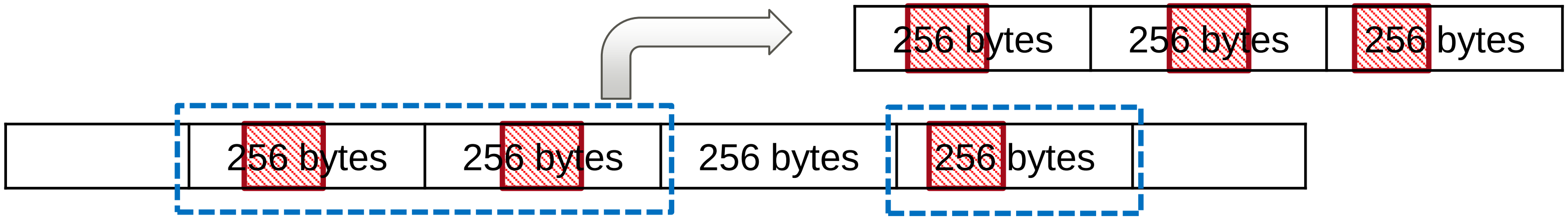


If the data to be prefetched spans two lines, the data of the two lines is prefetched.

③ SVE gather prefetch instruction

Prefetch instruction corresponding to discrete access instructions (gather/scatter)

The (3) SVE gather prefetch instruction is used for indirect access.



Prefetch Collaboration Between Hardware and Software

HWPF may not be generated in the cases below.

Complementing hardware prefetch with software prefetch achieves ideal performance.

- There are more than 16 streams.
- Block stride
  - Loop blocking (effective for array replacement, matrix operation, etc.)
  - Unrolling of an outer loop
- Access by masked SIMD
  When a stream in an if statement is accessed, hardware prefetch may not be generated depending on the true rate of the if statement, preventing contiguous access from occurring.

The term "collaboration" means that software prefetch complements the prefetching process in case hardware prefetch is not generated.

Prefetch Collaboration Verification (1/2)

Evaluation of the case when there are more than 16 streams

If the innermost loop has more than 16 streams, software prefetch complements part of the prefetching process (collaboration).
Software prefetch handles as many streams as the total number of streams minus 16 (when there are 20 streams, software prefetch handles four).
This is automatically recognized by the compiler and done transparently to the user.

- Evaluation cases verified

(1) HWPF only

A case of execution is evaluated where prefetching is performed only by HWPF without SWPF complementing the process even when there are more than 16 streams.
There is the possibility that the PFQ size may become insufficient, making sufficiently effective prefetching difficult.
In a program like a short loop, the startup of HWPF may become remarkable.

(2) SWPF only

A case of execution is evaluated where all prefetching is performed by software prefetch.
An increase in the number of instructions may have an adverse effect, potentially preventing the performance from being improved.

(3) HWPF + SWPF (default for the compiler)

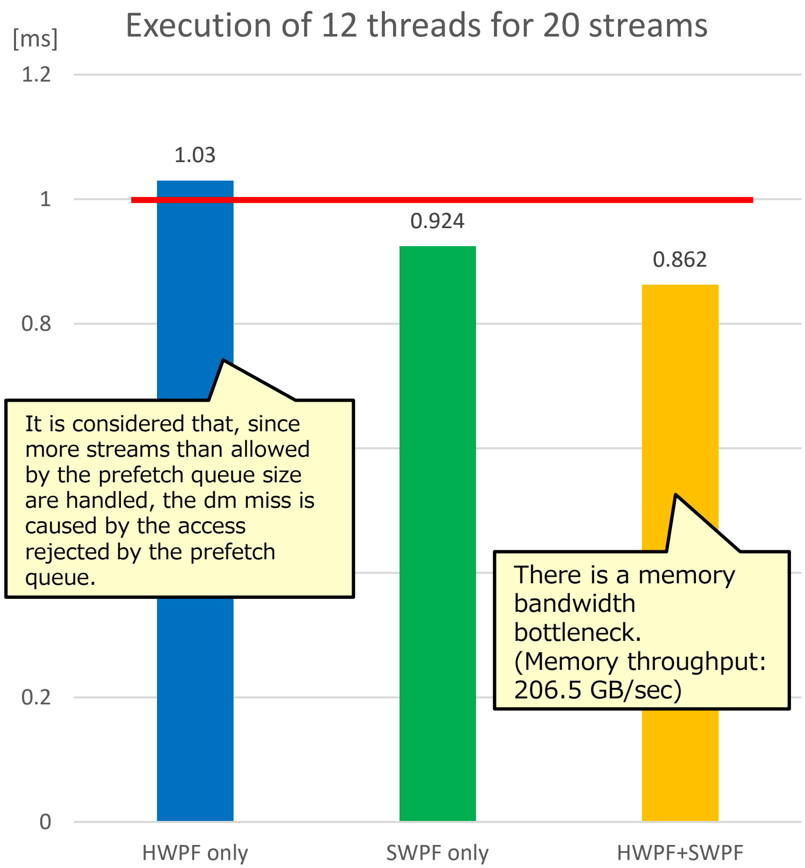
A case of execution is evaluated where SWPF complements the prefetching process when there are more than 16 streams.

Prefetch Collaboration Verification (2/2)

Execution results

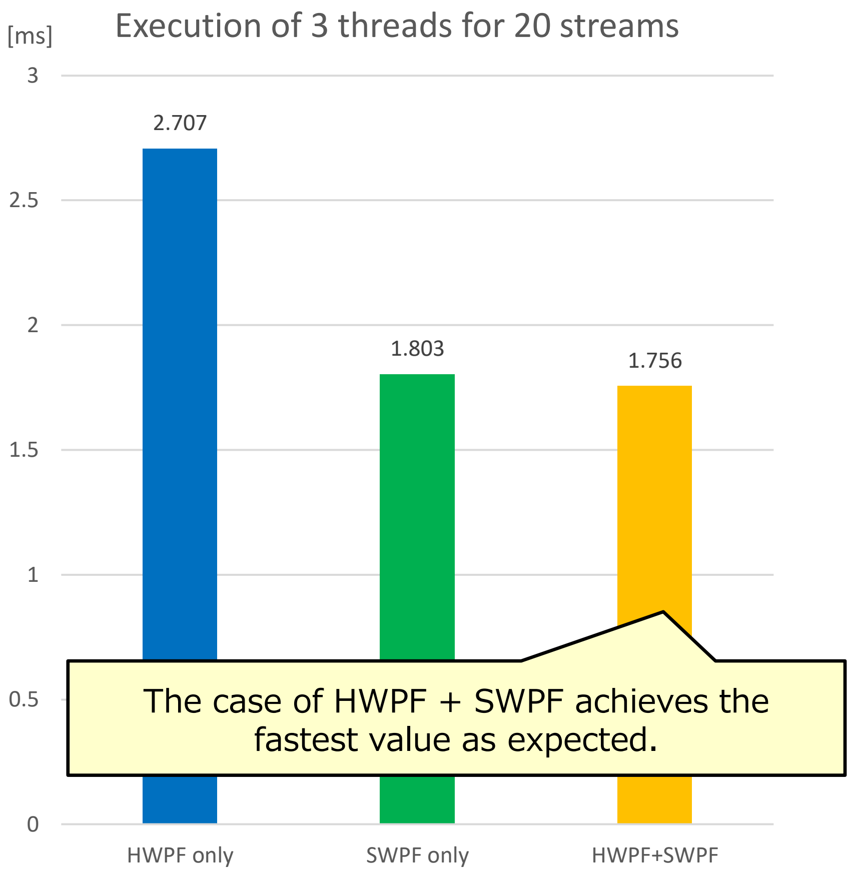
```
do k = 1, iter                                    iter=3 n=323323
!$omp parallel do
     do i = 1, n
       y(i)  = ((((((((x1(i) &
               *c0 +x2(i) ) *c1 +x3(i) ) *c2 +x4(i) ) *c3 +x5(i) ) &
               *c4 +x6(i) ) *c5 +x7(i) ) *c6 +x8(i) ) *c7 +x9(i) )
       y2(i) = ((((((((x11(i) &
               *c0 +x12(i) ) *c1 +x13(i) ) *c2 +x14(i) ) *c3 +x15(i) ) &
               *c4 +x16(i) ) *c5 +x17(i) ) *c6 +x18(i) ) *c7 +x19(i) )
     end do
  end do
```

Three threads are executed to evaluate a state in which there is a memory bandwidth bottleneck (latency bottleneck).



It is considered that, since more streams than allowed by the prefetch queue size are handled, the dm miss is caused by the access rejected by the prefetch queue.

There is a memory bandwidth bottleneck. (Memory throughput: 206.5 GB/sec)



The case of HWPF + SWPF achieves the fastest value as expected.

SFI (Store Fetch Interlock)

- About SFI
- Causes of Excessive SFI and SFI Precheck
- Explanation of Measurement Cases
- Excessive SFI Prevention by the Compiler

About SFI

- SFI (Store Fetch Interlock)
  - This is a control mechanism that applies an interlock so that, when the address of the preceding store instruction is identical to that of the succeeding load instruction, load is not executed before store.
  - Basically, the interlock is applied only to the address where store is executed.

- Excessive SFI

This can be prevented by the compiler. (A case of such prevention will be described later.)

Excessive SFI occurs in the following cases. (Details will be given later.)

- In the case of masked SIMD, the <u>addresses whose mask judgment results are 0 (do not store) are also locked</u>.
- When the gathering function of Gather Load is activated, <u>the SFI target to be checked by Gather Load becomes the cache lines (all included entries). Actually, there may be cases where the SFI of an address for which load is not executed is detected to determine whether the address is locked</u>.

- The occurrence of SFI can be checked based on the CPU analysis report.

Causes of Excessive SFI and SFI Precheck

Causes of excessive SFI

Cause	Description of excessive SFI	Related application
Predicate mask = 0	SFI also applies to the store of an entry whose predicate mask is 0.	ADVENTURE
Gather Load instruction aggregation	When the Gather Load instruction activates the aggregation function, the SFI check target becomes the cache lines.	GENESIS
Load instruction for access across a 4-KB boundary	In the case of a load instruction for access across a 4-KB boundary, the SFI check is always performed on bits 11 to 0 of the physical address.	
Addresses not aligned on the 4-B boundary in all SIMD entry sizes	The SFI check is always performed with the data length aligned on the 4-B boundary.	
Multiple structure instruction	- The SFI check target becomes the cache lines. (Entry size: 4 B/8 B) - Even if VL is 0 or 1, the SFI check target for store is the same as when VL is 3. (Entry size: 1 B/2 B)	

Causes of SFI precheck

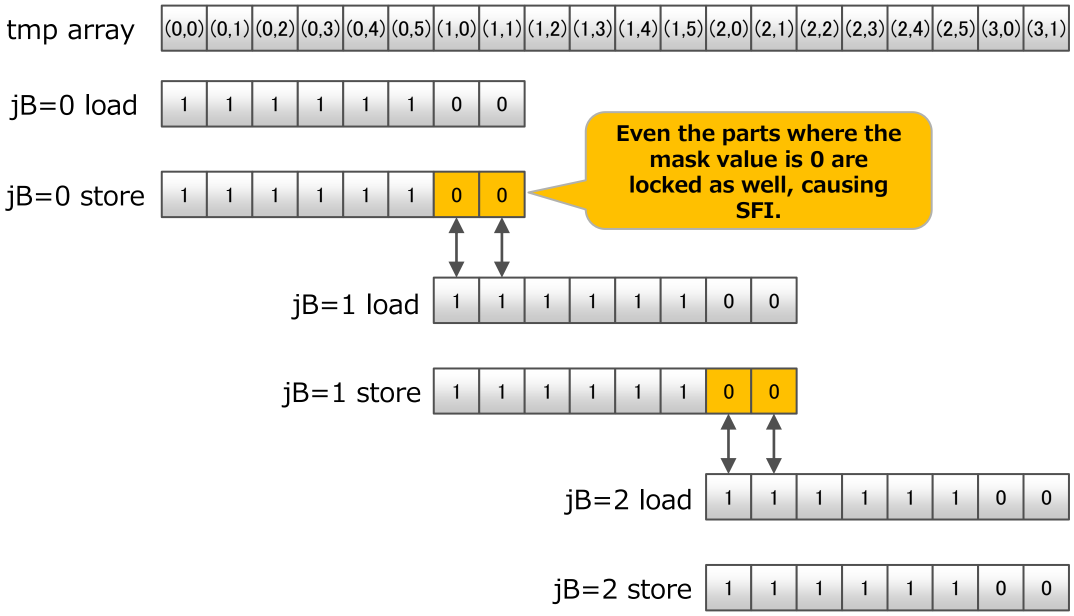
Cause	Description of SFI precheck	Related application
When the preceding store instruction exists in the pipeline	When the preceding store instruction exists in the pipeline, the SFI precheck is performed on bits 11 to 0 of the physical address.	on L1$ array access
When the preceding store instruction misses L1D$	The SFI precheck is performed on bits 11 to 0 of the physical address until the store data arrives at L1D$. The load instruction is not entered again until after the arrival of the store data.	

Explanation of Measurement Cases

- Excessive SFI when the Mask Value of the Access Instruction is 0
- Excessive SFI when the Gather Load Instruction Aggregation Function Makes the Cache Lines the SFI Check Target
- Excessive SFI when the SIMD Address is Not on the 4-B Boundary
- Excessive SFI when the Load Instruction Calls for Access Across a 4-KB Boundary
- Excessive SFI for Multiple Structure
- Causes of SFI Precheck
- Simple Cases when the Mask Value is 0
- Impact on Performance when the Mask Value is 0

Excessive SFI when the Mask Value of the Access Instruction is 0

```
for (jBlock = 0; jBlock < iBlock; jBlock++) {
 for ( iv = 0; iv < 6; iv++ ){
  double v_j = vector2[jBlock][iv];
  double m_i0_j = OffDiagComponents[instanceId][offset + jBlock][0][iv];
  double m_i1_j = OffDiagComponents[instanceId][offset + jBlock][1][iv];
  double m_i2_j = OffDiagComponents[instanceId][offset + jBlock][2][iv];
  double m_i3_j = OffDiagComponents[instanceId][offset + jBlock][3][iv];
  double m_i4_j = OffDiagComponents[instanceId][offset + jBlock][4][iv];
  double m_i5_j = OffDiagComponents[instanceId][offset + jBlock][5][iv];
  tmp[threadId][jBlock][iv]
       += m_i0_j * v_i0 + m_i1_j * v_i1 + m_i2_j * v_i2
        + m_i3_j * v_i3 + m_i4_j * v_i4 + m_i5_j * v_i5;
 }
}
```



tmp array: (0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (1,0) (1,1) (1,2) (1,3) (1,4) (1,5) (2,0) (2,1) (2,2) (2,3) (2,4) (2,5) (3,0) (3,1)

jB=0 load: 1 1 1 1 1 1 0 0

jB=0 store: 1 1 1 1 1 1 0 0

jB=1 load: 1 1 1 1 1 1 0 0

jB=1 store: 1 1 1 1 1 1 0 0

jB=2 load: 1 1 1 1 1 1 0 0

jB=2 store: 1 1 1 1 1 1 0 0

Even the parts where the mask value is 0 are locked as well, causing SFI.

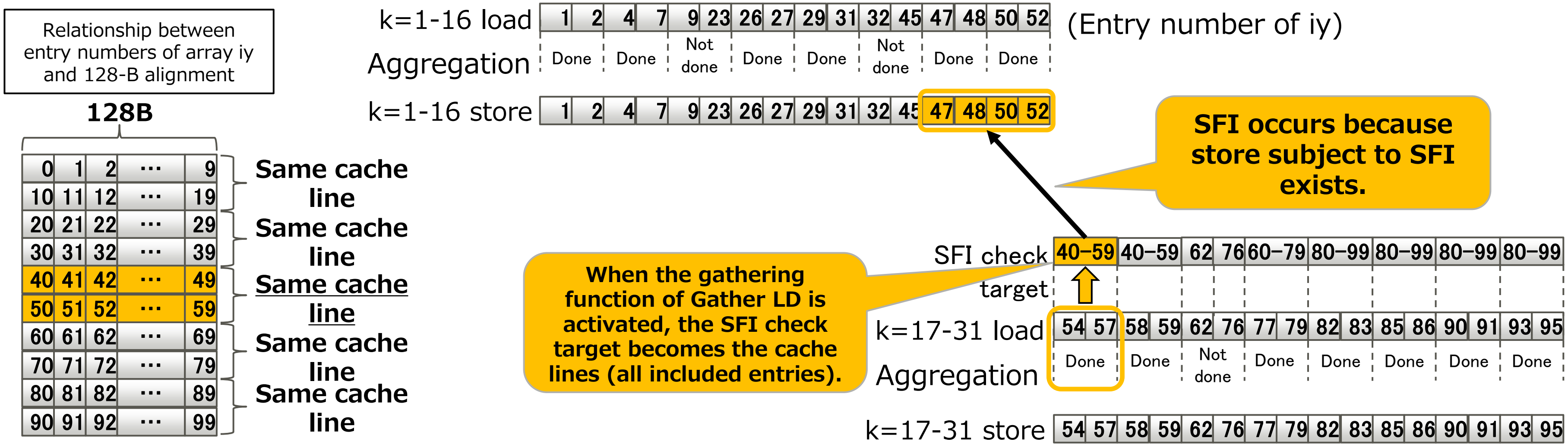
Excessive SFI when the Gather Load Instruction Aggregation Function Makes the Cache Lines the SFI Check Target

```
do k = 1, num_nb15_calc(ix,i,ik)

  ij = nb15_calc_list(k,ix,i,ik)
  j  = int(real(ij)*inv_MaxAtom)
  iy = ij - j*MaxAtom


  ...


  force(1,iy,j,id+1,ik) = force(1,iy,j,id+1,ik) + work(1)
  force(2,iy,j,id+1,ik) = force(2,iy,j,id+1,ik) + work(2)
  force(3,iy,j,id+1,ik) = force(3,iy,j,id+1,ik) + work(3)

end do
```

Notes:

1. The list values of the array are changed to easy-to-explain values for simplicity.

2. Excessive SFI also occurs between force(1,..), force(2,..), and force(3,..), but its description is omitted.



Relationship between entry numbers of array iy and 128-B alignment

128B

0	1	2	···	9	Same cache line
10	11	12	···	19	
20	21	22	···	29	Same cache line
30	31	32	···	39	
40	41	42	···	49	Same cache line
50	51	52	···	59	
60	61	62	···	69	Same cache line
70	71	72	···	79	
80	81	82	···	89	Same cache line
90	91	92	···	99	

(Entry number of iy)

k=1-16 load	1	2	4	7	9	23	26	27	29	31	32	45	47	48	50	52
Aggregation	Done		Done		Not done		Done		Done		Not done		Done		Done	
k=1-16 store	1	2	4	7	9	23	26	27	29	31	32	45	47	48	50	52

SFI occurs because store subject to SFI exists.

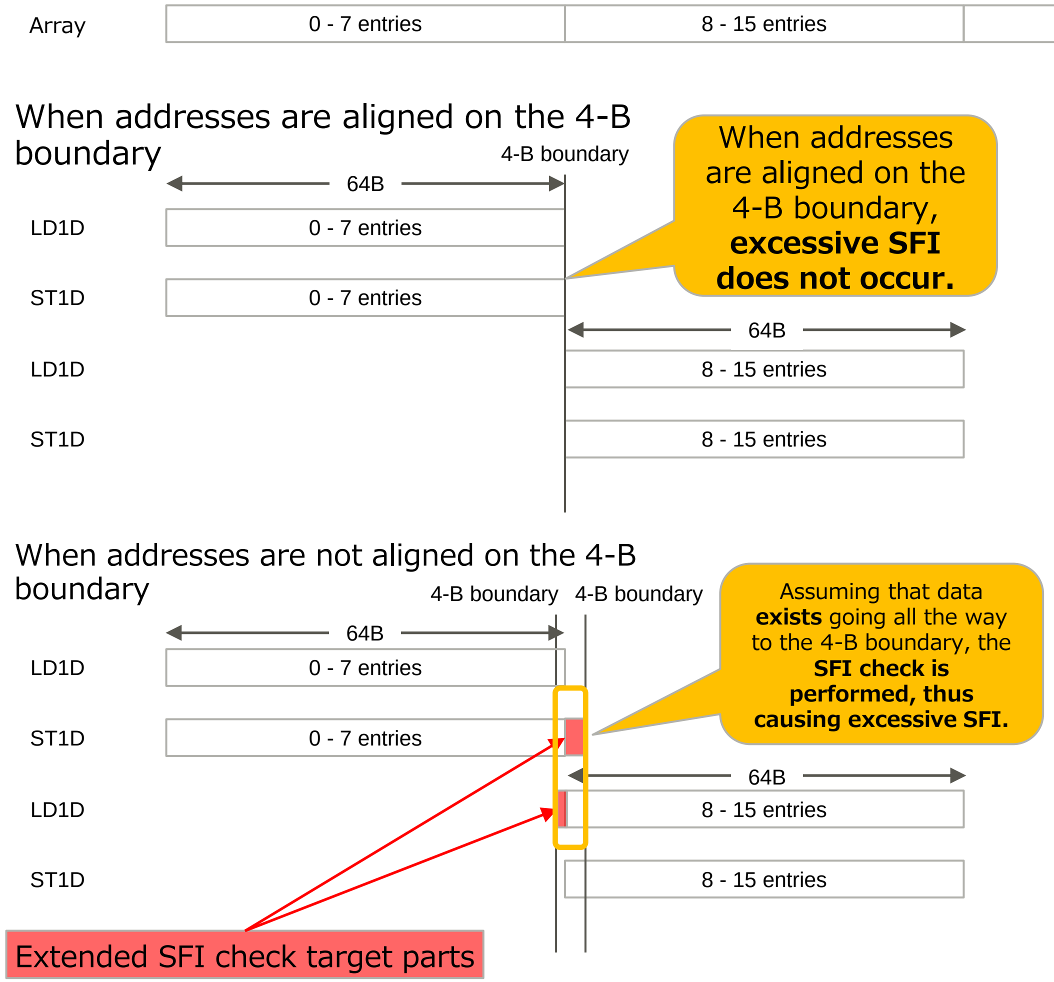
When the gathering function of Gather LD is activated, the SFI check target becomes the cache lines (all included entries).

SFI check target	40–59		40–59		62	76	60–79		80–99		80–99		80–99		80–99	
k=17-31 load	54	57	58	59	62	76	77	79	82	83	85	86	90	91	93	95
Aggregation	Done		Done		Not done		Done		Done		Done		Done		Done	
k=17-31 store	54	57	58	59	62	76	77	79	82	83	85	86	90	91	93	95

Excessive SFI when the SIMD Address is Not on the 4-B Boundary

- When addresses are not aligned on the 4-B boundary in all SIMD entry sizes, the SFI check target always becomes the 4-B boundary.

  Example: When the array entries are read and updated at double precision

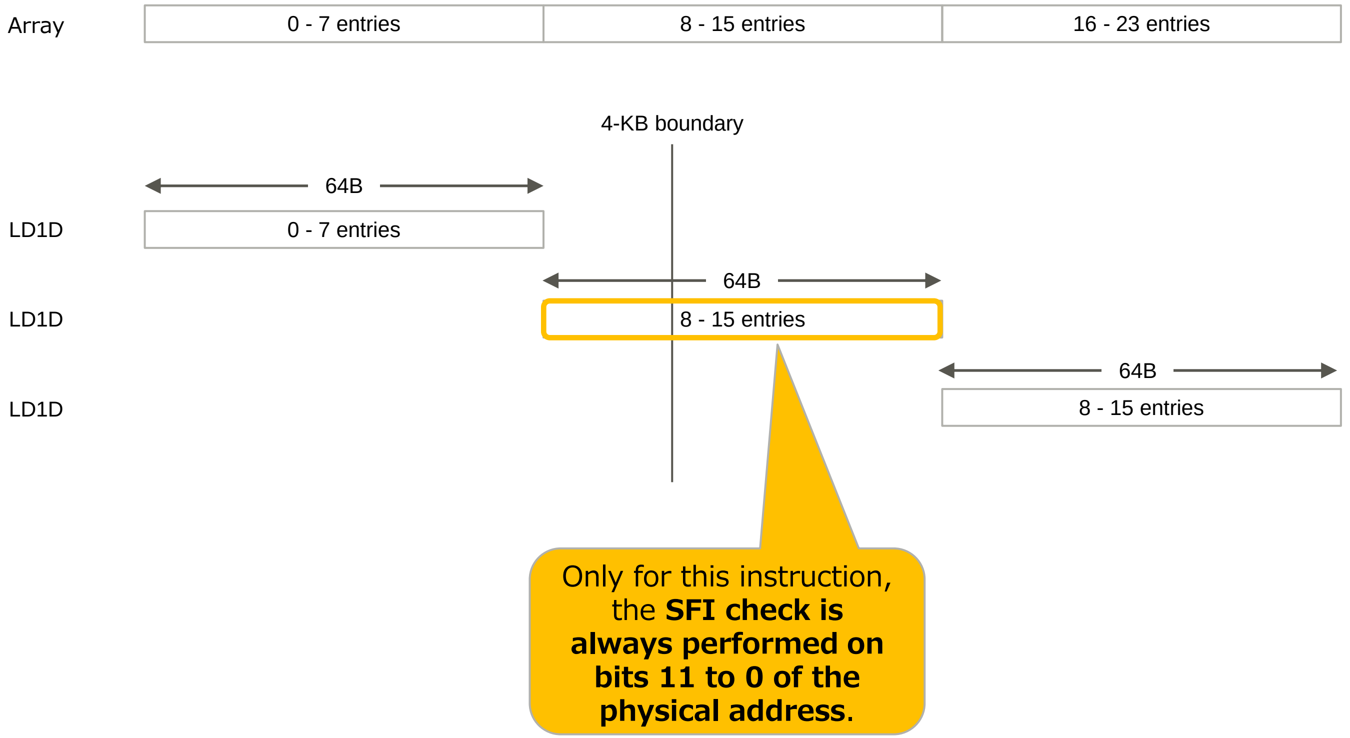

Array
0 - 7 entries
8 - 15 entries
16 - 23 entries
When addresses are aligned on the 4-B boundary
4-B boundary
64B
LD1D
0 - 7 entries
ST1D
0 - 7 entries
64B
LD1D
8 - 15 entries
ST1D
8 - 15 entries
When addresses are aligned on the 4-B boundary, excessive SFI does not occur.
When addresses are not aligned on the 4-B boundary
4-B boundary
4-B boundary
64B
LD1D
0 - 7 entries
ST1D
0 - 7 entries
64B
LD1D
8 - 15 entries
ST1D
8 - 15 entries
Assuming that data exists going all the way to the 4-B boundary, the SFI check is performed, thus causing excessive SFI.
Extended SFI check target parts

- ✓ In the case of 4-B or 8-B entries, addresses are aligned on the 4-B or 8-B boundary in principle and, therefore, there is no problem.
- ✓ Care needs to be exercised about FP16 or integers. Note that the problem may be mitigated by software pipelining.

Excessive SFI when the Load Instruction Calls for Access Across a 4-KB Boundary

- When access across a 4-KB boundary is made with a single load instruction, the SFI check is always performed on bits 11 to 0 of the physical address.

Example: When the array entries are read at double precision



Only for this instruction, the **SFI check is always performed on bits 11 to 0 of the physical address.**

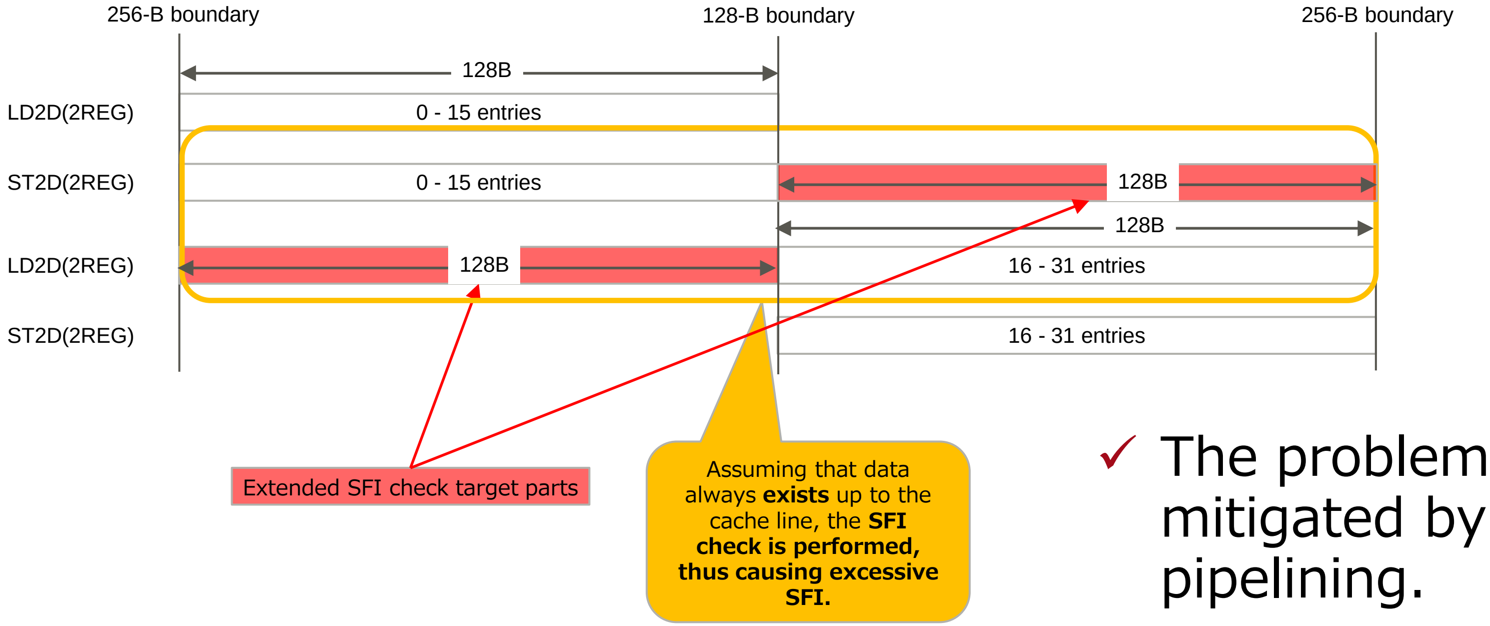
Excessive SFI for Multiple Structure (1/2)

- In a multiple structure store / load instruction whose entry size is 4 B/8 B, the SFI check target becomes the cache lines.

Example: When the array entries are read and updated by a LD2D instruction/ST2D instruction

Array	0 - 15 entries	16 - 31 entries	32 and subsequent entries

- When addresses are aligned on the 256-B boundary



Extended SFI check target parts

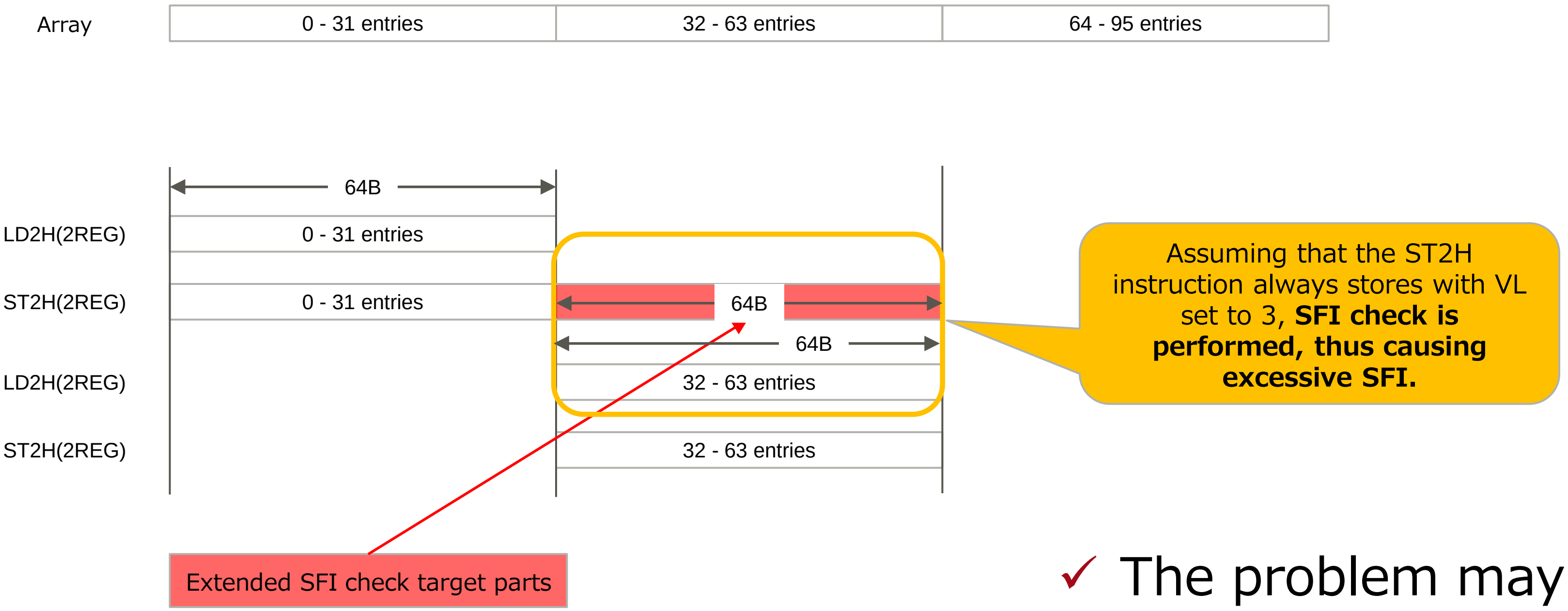
Assuming that data always **exists** up to the cache line, the **SFI check is performed, thus causing excessive SFI.**

✓ The problem may be mitigated by software pipelining.

Excessive SFI for Multiple Structure (2/2)

- In a multiple structure store instruction whose entry size is 1 B/2 B, when the VL (vector length) is 0 (128-bit SIMD) or 1 (256-bit SIMD), the SFI check target is the same as when the VL is 3 (512-bit SIMD).

  Example: When the array entries are read and updated by a LD2H instruction/ST2H instruction with the VL set to 1 (256-bit SIMD)

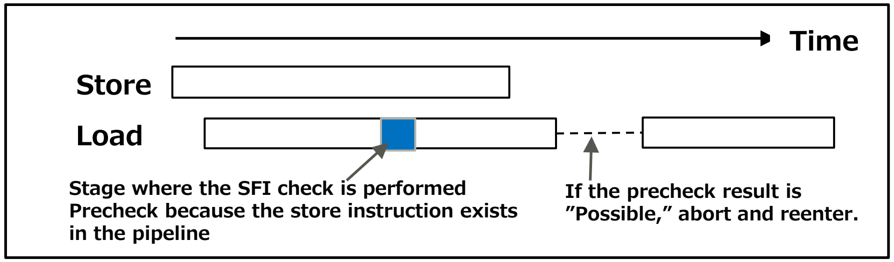


Assuming that the ST2H instruction always stores with VL set to 3, **SFI check is performed, thus causing excessive SFI.**

✓ The problem may be mitigated by software pipelining.

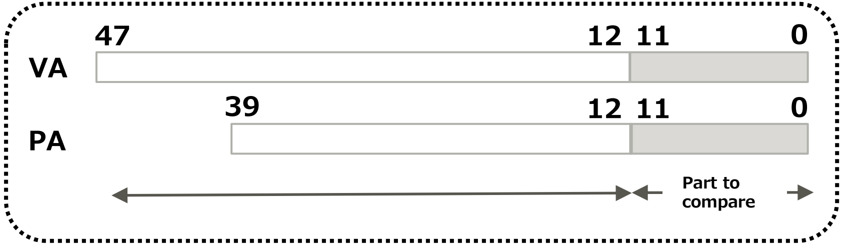
Causes of SFI Precheck

1. When the preceding store instruction exists in the pipeline while the pipelining of the load instruction is in progress

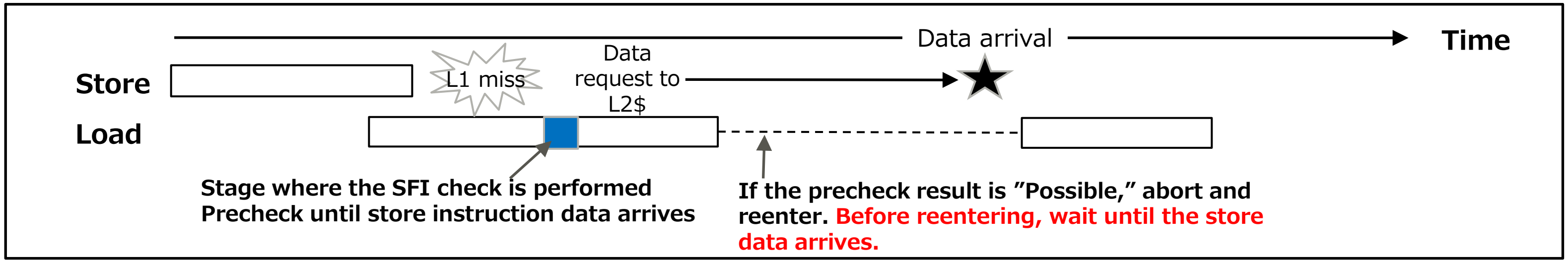


Time
Store
Load
Stage where the SFI check is performed
Precheck because the store instruction exists in the pipeline
If the precheck result is "Possible," abort and reenter.

Addresses used for the SFI precheck

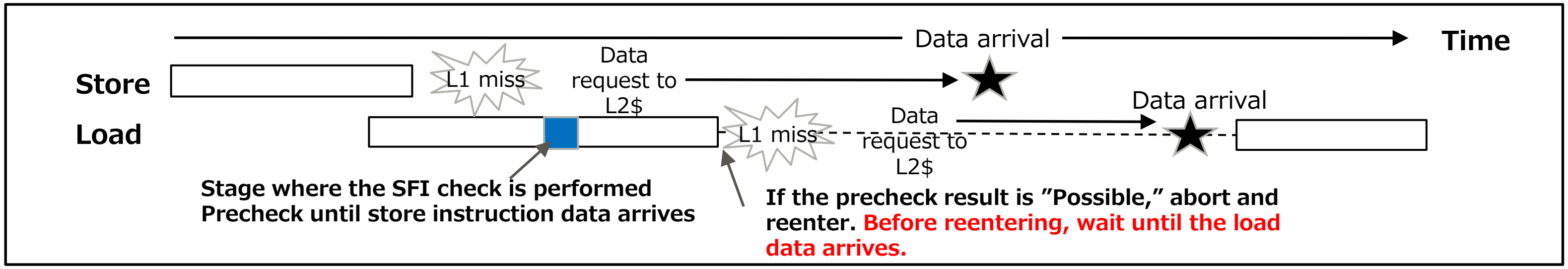


2. When the preceding store instruction misses the L1D$ cache and retrieves data from L2$ (1) (When the load instruction hits L1D$: Precheck and wait until the store data arrives)



Data arrival
Time
Store
L1 miss
Data request to L2$
Load
Stage where the SFI check is performed
Precheck until store instruction data arrives
If the precheck result is "Possible," abort and reenter. Before reentering, wait until the store data arrives.

3. When the preceding store instruction misses the L1D$ cache and retrieves data from L2$ (2) (When the load instruction misses L1D$: Precheck and wait until the store data arrives)



Data arrival
Time
Store
L1 miss
Data request to L2$
Data arrival
Load
L1 miss
Data request to L2$
Stage where the SFI check is performed
Precheck until store instruction data arrives
If the precheck result is "Possible," abort and reenter. Before reentering, wait until the load data arrives.

✓ Since the data request is made to L2$, there is no impact on performance, while the SFI count of the PA is updated.

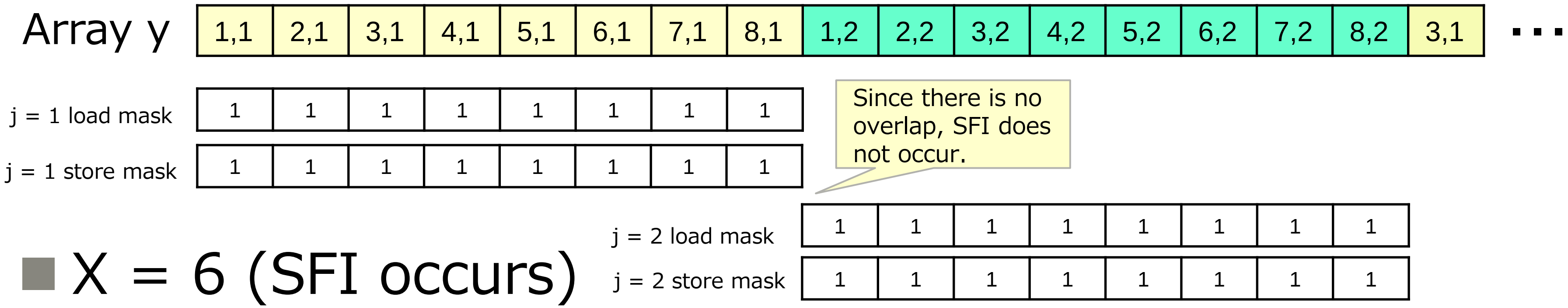
Simple Cases when the Mask Value is 0

Image of access

Code to be measured

```
real*8 y(X,n), x1(X,n)    <- X=8,6
  Do k = 1, iter
    Do j = 1, n
      Do i = 1, X        <- X=8,6
        y(i,j) = y(i,j) + x1(i,j)
      End Do
    End Do
  End Do
```

- X = 8 (SFI does not occur)

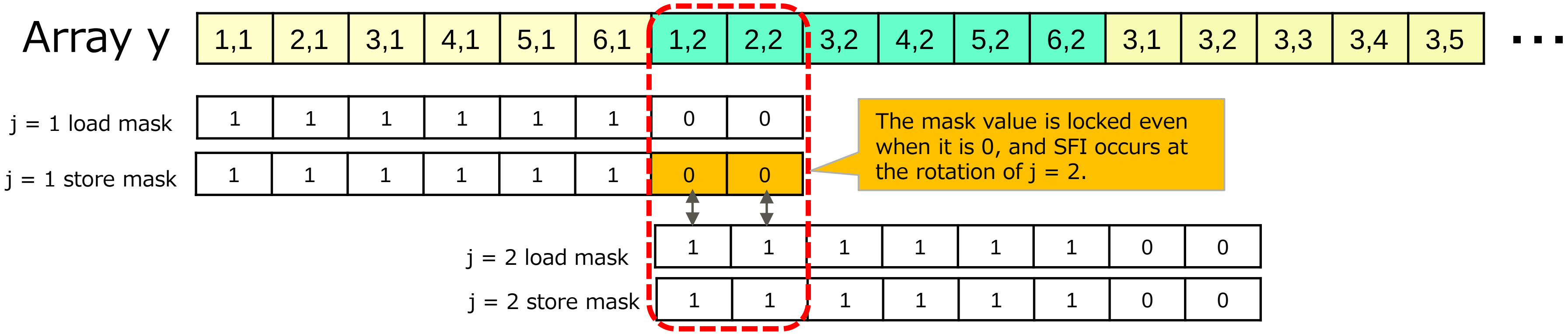


Array y	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	3,1
j = 1 load mask	1	1	1	1	1	1	1	1									
j = 1 store mask	1	1	1	1	1	1	1	1									
j = 2 load mask									1	1	1	1	1	1	1	1	
j = 2 store mask									1	1	1	1	1	1	1	1	

・・・

Since there is no overlap, SFI does not occur.

- X = 6 (SFI occurs)



Array y	1,1	2,1	3,1	4,1	5,1	6,1	1,2	2,2	3,2	4,2	5,2	6,2	3,1	3,2	3,3	3,4	3,5
j = 1 load mask	1	1	1	1	1	1	0	0									
j = 1 store mask	1	1	1	1	1	1	0	0									
j = 2 load mask							1	1	1	1	1	1	0	0			
j = 2 store mask							1	1	1	1	1	1	0	0			

・・・

The mask value is locked even when it is 0, and SFI occurs at the rotation of j = 2.

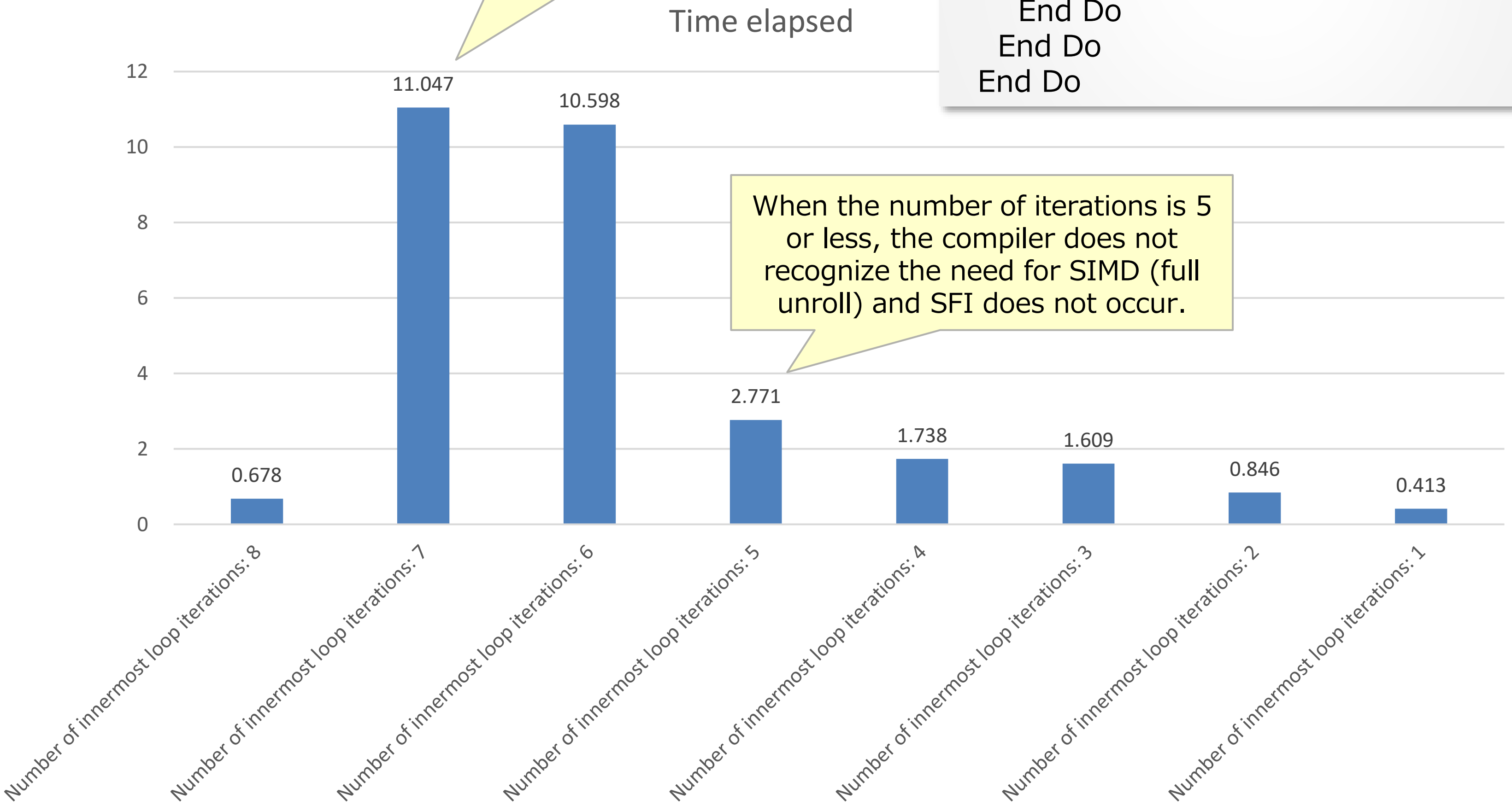
Impact on Performance when the Mask Value is 0

Measurement results

Code to be measured

```
real*8 y(X,n), x1(X,n)    <- X=8 to 1
  Do k = 1, iter
    Do j = 1, n
      Do i = 1, X        <- X=8 to 1
        y(i,j) = y(i,j) + x1(i,j)
      End Do
    End Do
  End Do
```

SFI occurs when the number of iterations is 7 or 6



When the number of iterations is 5 or less, the compiler does not recognize the need for SIMD (full unroll) and SFI does not occur.

Excessive SFI Prevention by the Compiler

- When the Mask Value is 0: Excessive SFI Prevention by the Compiler
- When the Mask Value is 0: Results of Excessive SFI Prevention by the Compiler

When the Mask Value is 0: Excessive SFI Prevention by the Compiler

- Excessive SFI can be prevented by compiler scheduling.

When SIMD is implemented in the innermost loop, scheduling (SWPL) is promoted on the outer loops to prevent excessive SFI.

Before improvement

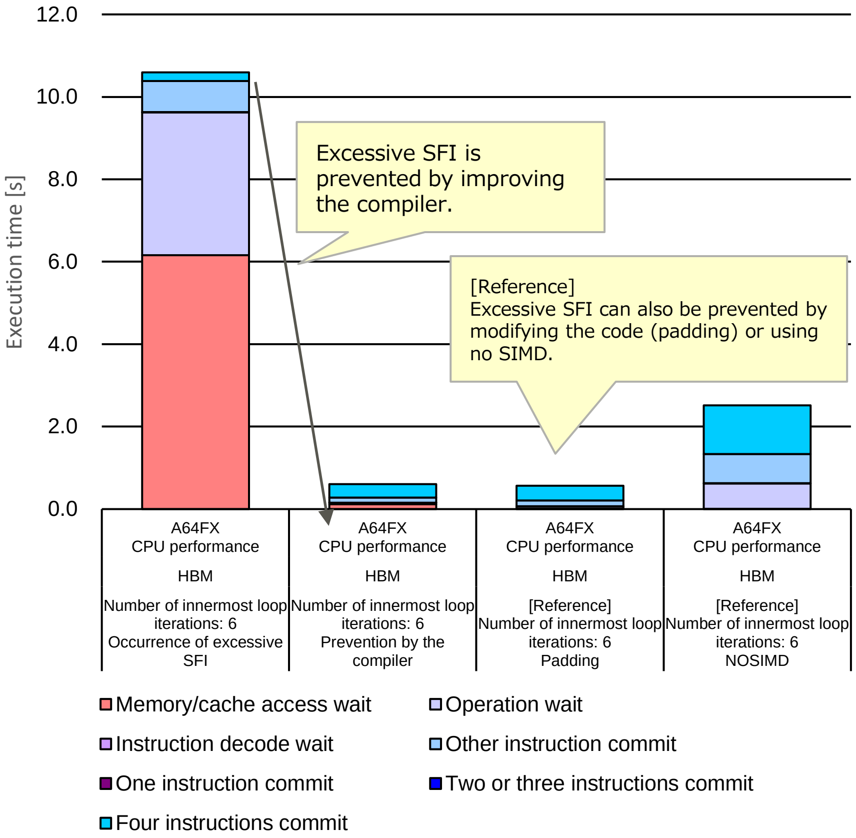
```
 5    1                Do k = 1, iter
 6    2     2            Do j = 1, n
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<< Loop-information  End >>>
 7    3     2v             Do i = 1, 6
 8    3     2v               y(i,j) = y(i,j) + x1(i,j)
 9    3     2v             End Do
10    2     2            End Do
11    1                End Do
```

After improvement
(preventive measure)

```
 5    1                Do k = 1, iter
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SOFTWARE PIPELINING
                  <<< Loop-information  End >>>
 6    2     2            Do j = 1, n
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<< Loop-information  End >>>
 7    3     2v             Do i = 1, 6
 8    3     2v               y(i,j) = y(i,j) + x1(i,j)
 9    3     2v             End Do
10    2     2            End Do
11    1                End Do
```

When the Mask Value is 0: Results of Excessive SFI Prevention by the Compiler

■ Measurement results of prevention by the compiler



Excessive SFI is prevented by improving the compiler.

[Reference]
Excessive SFI can also be prevented by modifying the code (padding) or using no SIMD.

	A64FX CPU performance Actual machine	A64FX CPU performance Actual machine	A64FX CPU performance Actual machine	A64FX CPU performance Actual machine
Source code version	Number of innermost loop iterations: 6 Occurrence of excessive SFI	Number of innermost loop iterations: 6 Prevention by the compiler	[Reference] Number of innermost loop iterations: 6 Padding	[Reference] Number of innermost loop iterations: 6 NOSIMD
Floating-point precision	Double precision	Double precision	Double precision	Double precision
SIMD width	8	8	8	1
Number of threads	1	1	1	1
Aggregation thread number	0	0	0	0
Execution time [s]	1.06E+01	6.02E-01	5.65E-01	2.51E+00
Total number of effective instructions	3.47E+09	2.81E+09	3.09E+09	1.08E+10
GFLOPS (processes)	0.29	5.10	5.44	0.92
Memory throughput [GB/s/process]	0.00	0.00	0.00	0.00
L1 busy rate/thread	42.25%	80.41%	88.26%	99.49%
SFI(/cycle)/thread	0.68	0.03	0.00	0.00

Excessive SFI when the mask value is 0 can be prevented.

PMU Events

- Features of CPU Analysis Report
- Improvements in the CPU Analysis Report from K/FX100 Products
- Examples of the Displayed CPU Analysis Report
  - Image of the Displayed CPU Analysis Report: Overview
  - Image of the Displayed CPU Analysis Report: Graphs
  - Image of the Displayed CPU Analysis Report: Tables
  - DGEMM: Display and Analysis Example
  - STREAM: Display and Analysis Examples
- Notes on FLOPS

Features of CPU Analysis Report

- Display CPU performance information in Excel based on PMU Events.
- The collected major performance data can be displayed just after five measurements.
- 11 times measurements recommended. (Equivalent to FX100)
- Conducting additional measurements can provide more detailed performance data.

Collectable information corresponding to the number of measurements.

Performance data Level	Single report	Simple report	Standard report	Overall report
Number of measurements	**1**	**5**	**11**	**17**
Statistical information	Collected	Collected※	<-	<-
Cycle accounting		Collected	Collected※	<-
Memory/cache busy status		Collected	Collected※	Collected※
Cache miss status		Collected	Collected※	<-
Instruction mix		Collected	Collected※	Collected※
Unbalanced load		Collected	<-	<-
Power consumption			Collected	<-
Hardware prefetch information				Collected
Inter-CMG data transfer status				Collected
Other performance data				Collected

(*) The higher the level is, the more data is collected.

Improvements in the CPU Analysis Report from K/FX100 Products

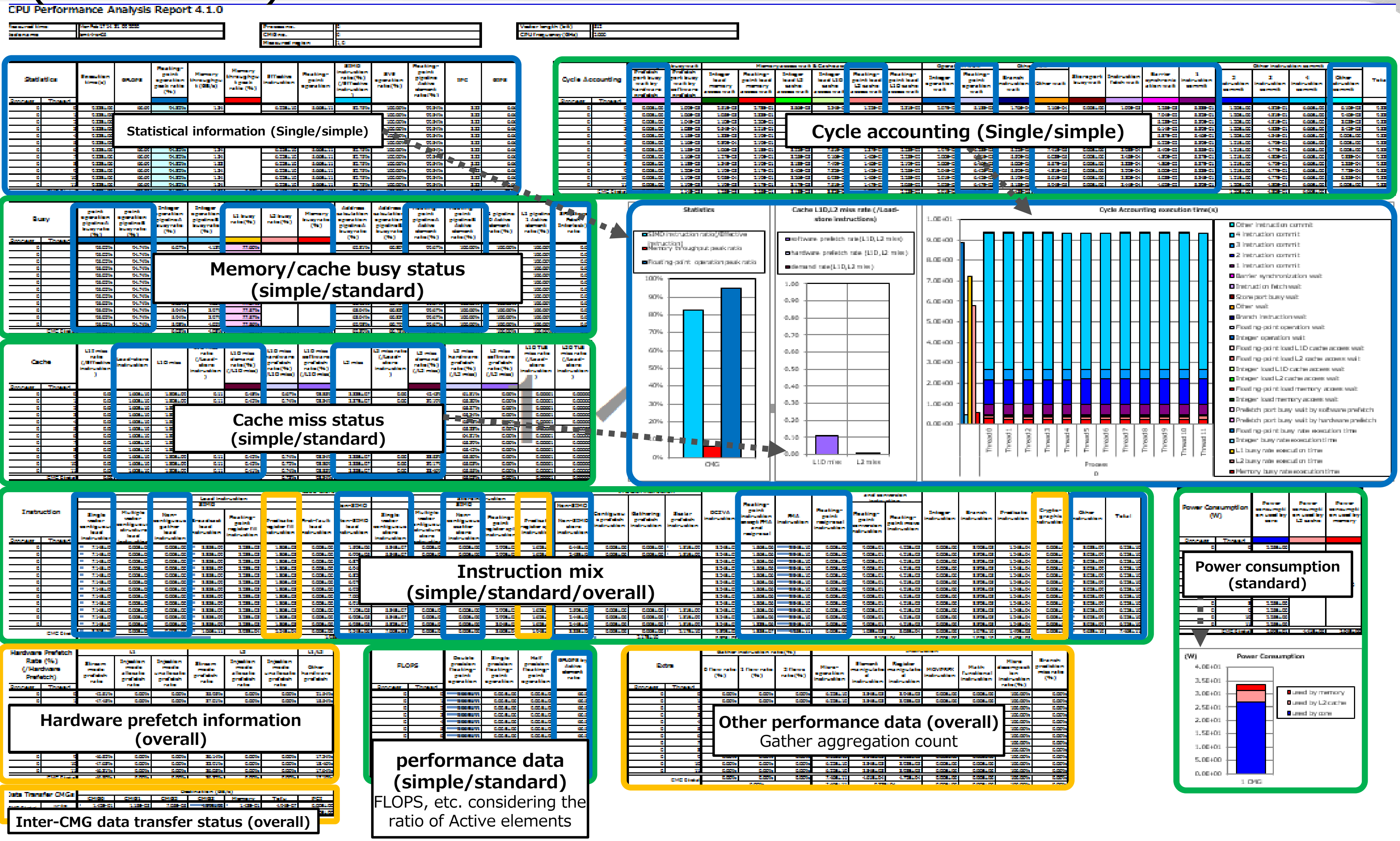
Improvement	K/FX100 products	Improvement for the A64FX
Display of performance data according to the number of measurements	The precision PA visibility function (CPU performance analysis report) cannot be used unless the measurement is conducted at least 7 times for K or 11 times for the FX100.	The CPU performance analysis report can be displayed just after 1 time or 5 times measurements. Additional data can be displayed by conducting the measurement 11 or 17 times.
		Additional measurements of cycle accounting and others can provide more detailed CPU performance data.
Increased amount of data	---	Various kinds of data have been added. - Floating-point and integer operation pipeline busy - L2 cache miss breakdown (dm miss rate, swpf miss rate, hwpf miss rate) - Predicate mask information - Gather aggregation count - Spill/fill counts - Inter-CMG access volume - Power, etc.
Layout	---	While the A64FX handles more data than the previous products, the main data is displayed in a single A3-size sheet.

Examples of the Displayed CPU Analysis Report

- Target applications
  - DGEMM
  - STREAM

- The measurement conditions are as follows.
  - Operating environment
    - A64FX
  - CPU performance analysis data collection level and number of measurements
    - Simple data (five measurements)

Image of the Displayed CPU Analysis Report (Overview)



CPU Performance Analysis Report 4.1.0

Statistical information (Single/simple)

Cycle accounting (Single/simple)

Memory/cache busy status (simple/standard)

Cache miss status (simple/standard)

Instruction mix (simple/standard/overall)

Power consumption (standard)

Hardware prefetch information (overall)

performance data (simple/standard)
FLOPS, etc. considering the ratio of Active elements

Other performance data (overall)
Gather aggregation count

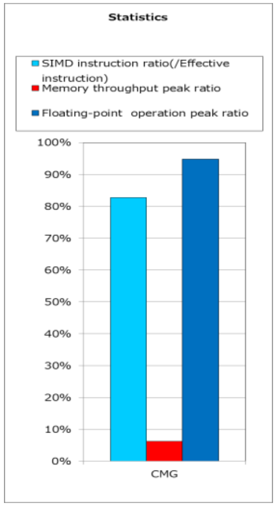
Inter-CMG data transfer status (overall)

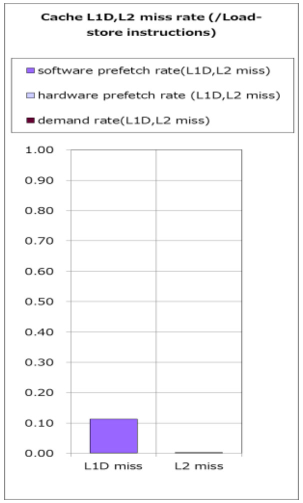
Depending on the number of times data is collected, the tables and graphs are displayed with increasing levels of detail in the following order: simple, standard, and overall.
(The actual CPU performance analysis report does not display colored frames like those shown below.)

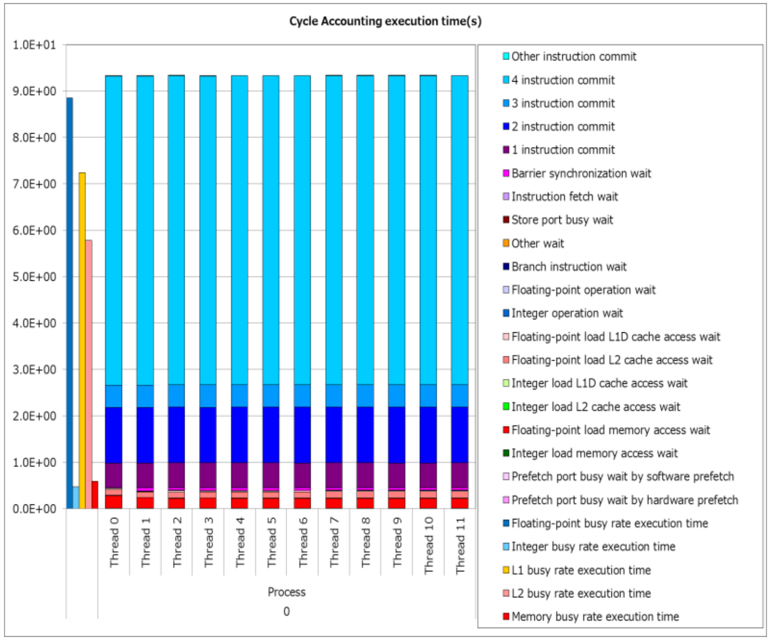
Simple Standard Overall

Image of the Displayed CPU Analysis Report (Graphs)

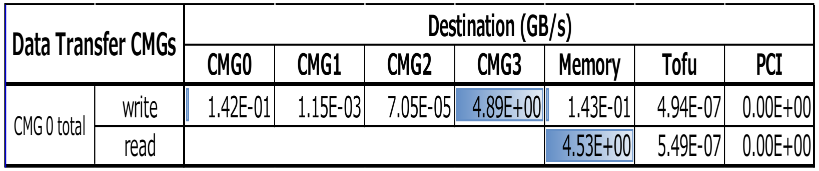
- A look at the following data gives you a rough grasp of the performance.
  - Operation peak, throughput peak, and SIMD ratios
  - Cache status
  - Cycle accounting







- Inter-CMG data transfer status



Data Transfer CMGs		Destination (GB/s)						
		CMG0	CMG1	CMG2	CMG3	Memory	Tofu	PCI
CMG 0 total	write	1.42E-01	1.15E-03	7.05E-05	4.89E+00	1.43E-01	4.94E-07	0.00E+00
	read					4.53E+00	5.49E-07	0.00E+00

- Power consumption

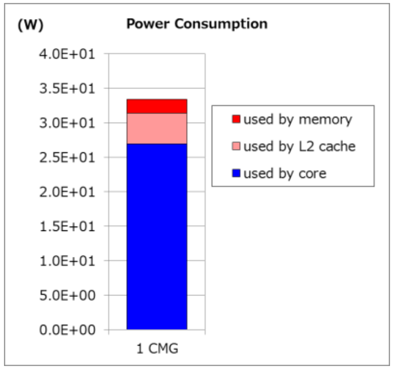
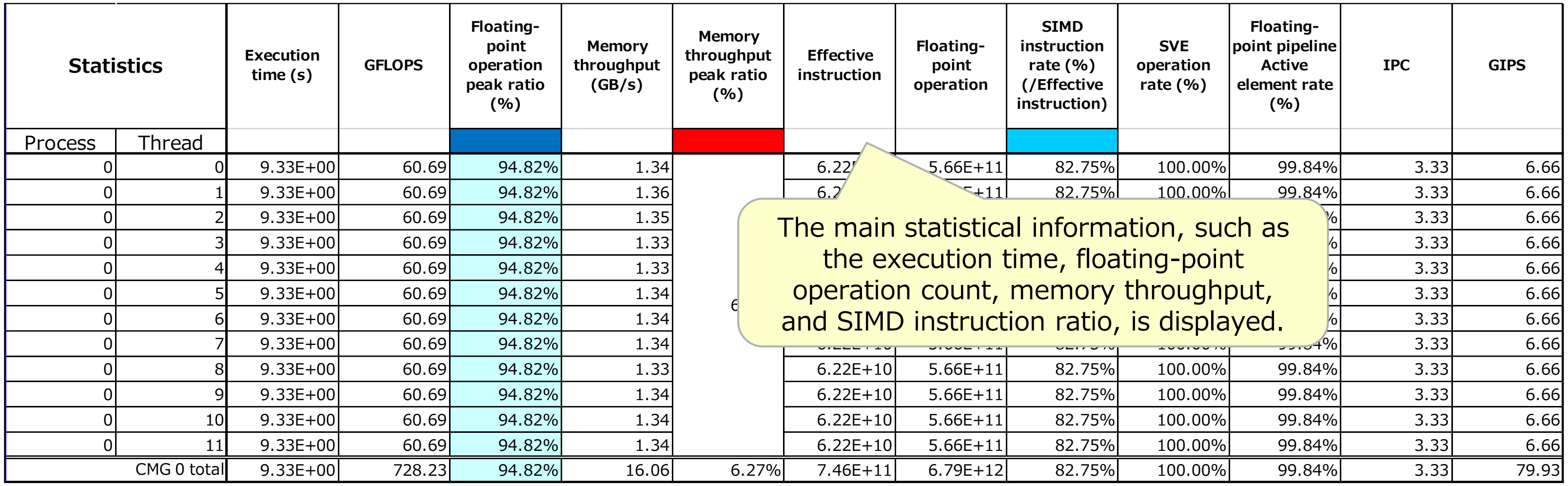


Image of the Displayed CPU Analysis Report (Table 1)

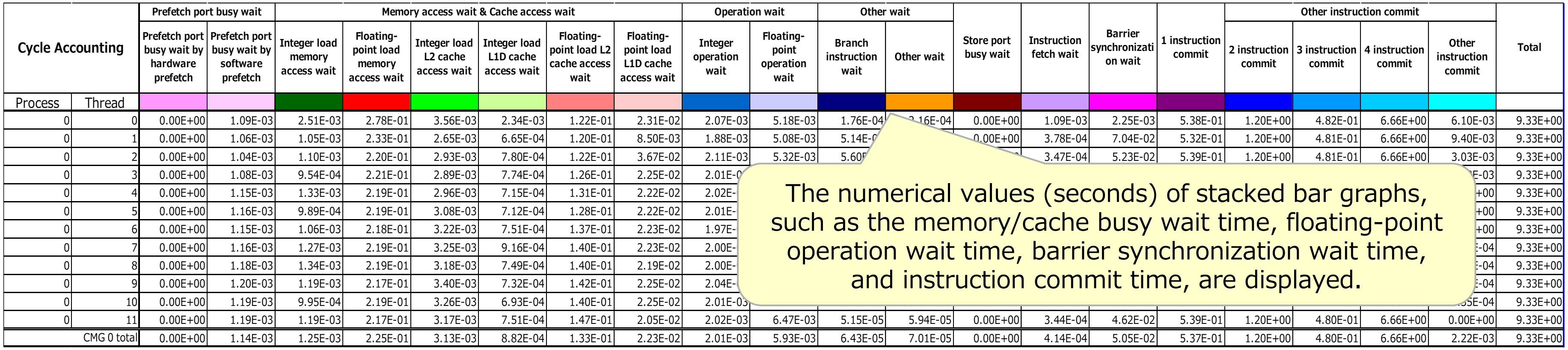
Statistical information



Statistics		Execution time (s)	GFLOPS	Floating-point operation peak ratio (%)	Memory throughput (GB/s)	Memory throughput peak ratio (%)	Effective instruction	Floating-point operation	SIMD instruction rate (%) (/Effective instruction)	SVE operation rate (%)	Floating-point pipeline Active element rate (%)	IPC	GIPS
Process	Thread			🟦		🟥			🟦				
0	0	9.33E+00	60.69	94.82%	1.34	6	6.22	5.66E+11	82.75%	100.00%	99.84%	3.33	6.66
0	1	9.33E+00	60.69	94.82%	1.36		6.2	+11	82.75%	100.00%	99.84%	3.33	6.66
0	2	9.33E+00	60.69	94.82%	1.35		[illegible]	[illegible]	[illegible]	[illegible]	%	3.33	6.66
0	3	9.33E+00	60.69	94.82%	1.33		[illegible]	[illegible]	[illegible]	[illegible]	%	3.33	6.66
0	4	9.33E+00	60.69	94.82%	1.33		[illegible]	[illegible]	[illegible]	[illegible]	%	3.33	6.66
0	5	9.33E+00	60.69	94.82%	1.34		[illegible]	[illegible]	[illegible]	[illegible]	%	3.33	6.66
0	6	9.33E+00	60.69	94.82%	1.34		[illegible]	[illegible]	[illegible]	[illegible]	%	3.33	6.66
0	7	9.33E+00	60.69	94.82%	1.34		6.22E+10	5.66E+11	82.75%	100.00%	99.84%	3.33	6.66
0	8	9.33E+00	60.69	94.82%	1.33		6.22E+10	5.66E+11	82.75%	100.00%	99.84%	3.33	6.66
0	9	9.33E+00	60.69	94.82%	1.34		6.22E+10	5.66E+11	82.75%	100.00%	99.84%	3.33	6.66
0	10	9.33E+00	60.69	94.82%	1.34		6.22E+10	5.66E+11	82.75%	100.00%	99.84%	3.33	6.66
0	11	9.33E+00	60.69	94.82%	1.34		6.22E+10	5.66E+11	82.75%	100.00%	99.84%	3.33	6.66
	CMG 0 total	9.33E+00	728.23	94.82%	16.06	6.27%	7.46E+11	6.79E+12	82.75%	100.00%	99.84%	3.33	79.93

The main statistical information, such as the execution time, floating-point operation count, memory throughput, and SIMD instruction ratio, is displayed.

Cycle accounting

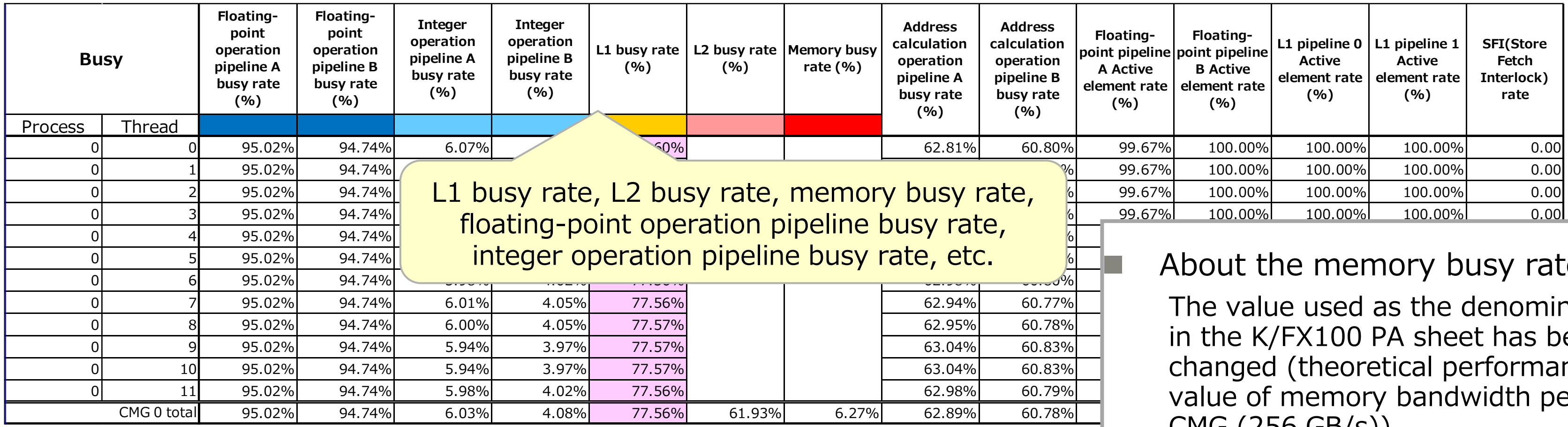


Cycle Accounting		Prefetch port busy wait		Memory access wait & Cache access wait						Operation wait		Other wait						Other instruction commit				
		Prefetch port busy wait by hardware prefetch	Prefetch port busy wait by software prefetch	Integer load memory access wait	Floating-point load memory access wait	Integer load L2 cache access wait	Integer load L1D cache access wait	Floating-point load L2 cache access wait	Floating-point load L1D cache access wait	Integer operation wait	Floating-point operation wait	Branch instruction wait	Other wait	Store port busy wait	Instruction fetch wait	Barrier synchronization wait	1 instruction commit	2 instruction commit	3 instruction commit	4 instruction commit	Other instruction commit	Total
Process	Thread	🟪	🟪	🟩	🟥	🟩	🟩	🟥	🟥	🟦	🟦	🟦	🟧	🟫	🟪	🟪	🟪	🟦	🟦	🟦	🟦	
0	0	0.00E+00	1.09E-03	2.51E-03	2.78E-01	3.56E-03	2.34E-03	1.22E-01	2.31E-02	2.07E-03	5.18E-03	1.76E-04	2.16E-04	0.00E+00	1.09E-03	2.25E-03	5.38E-01	1.20E+00	4.82E-01	6.66E+00	6.10E-03	9.33E+00
0	1	0.00E+00	1.06E-03	1.05E-03	2.33E-01	2.65E-03	6.65E-04	1.20E-01	8.50E-03	1.88E-03	5.08E-03	5.14E-0	[illegible]	0.00E+00	3.78E-04	7.04E-02	5.32E-01	1.20E+00	4.81E-01	6.66E+00	9.40E-03	9.33E+00
0	2	0.00E+00	1.04E-03	1.10E-03	2.20E-01	2.93E-03	7.80E-04	1.22E-01	3.67E-02	2.11E-03	5.32E-03	5.60	[illegible]	[illegible]	3.47E-04	5.23E-02	5.39E-01	1.20E+00	4.81E-01	6.66E+00	3.03E-03	9.33E+00
0	3	0.00E+00	1.08E-03	9.54E-04	2.21E-01	2.89E-03	7.74E-04	1.26E-01	2.25E-02	2.01E-0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E-03	9.33E+00
0	4	0.00E+00	1.15E-03	1.33E-03	2.19E-01	2.96E-03	7.15E-04	1.31E-01	2.22E-02	2.02E-	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	+00	9.33E+00
0	5	0.00E+00	1.16E-03	9.89E-04	2.19E-01	3.08E-03	7.12E-04	1.28E-01	2.22E-02	2.01E-	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	+00	9.33E+00
0	6	0.00E+00	1.15E-03	1.06E-03	2.18E-01	3.22E-03	7.51E-04	1.37E-01	2.23E-02	1.97E-	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	+00	9.33E+00
0	7	0.00E+00	1.16E-03	1.27E-03	2.19E-01	3.25E-03	9.16E-04	1.40E-01	2.23E-02	2.00E-	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E-04	9.33E+00
0	8	0.00E+00	1.18E-03	1.34E-03	2.19E-01	3.18E-03	7.49E-04	1.40E-01	2.19E-02	2.00E-	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E-04	9.33E+00
0	9	0.00E+00	1.20E-03	1.19E-03	2.17E-01	3.40E-03	7.32E-04	1.42E-01	2.25E-02	2.04E-	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E-04	9.33E+00
0	10	0.00E+00	1.19E-03	9.95E-04	2.19E-01	3.26E-03	6.93E-04	1.40E-01	2.25E-02	2.01E-03	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	35E-04	9.33E+00
0	11	0.00E+00	1.19E-03	1.19E-03	2.17E-01	3.17E-03	7.51E-04	1.47E-01	2.05E-02	2.02E-03	6.47E-03	5.15E-05	5.94E-05	0.00E+00	3.44E-04	4.62E-02	5.39E-01	1.20E+00	4.80E-01	6.66E+00	0.00E+00	9.33E+00
	CMG 0 total	0.00E+00	1.14E-03	1.25E-03	2.25E-01	3.13E-03	8.82E-04	1.33E-01	2.23E-02	2.01E-03	5.93E-03	6.43E-05	7.01E-05	0.00E+00	4.14E-04	5.05E-02	5.37E-01	1.20E+00	4.80E-01	6.66E+00	2.22E-03	9.33E+00

The numerical values (seconds) of stacked bar graphs, such as the memory/cache busy wait time, floating-point operation wait time, barrier synchronization wait time, and instruction commit time, are displayed.

Image of the Displayed CPU Analysis Report (Table 2)

Memory/cache busy information



Busy		Floating-point operation pipeline A busy rate (%)	Floating-point operation pipeline B busy rate (%)	Integer operation pipeline A busy rate (%)	Integer operation pipeline B busy rate (%)	L1 busy rate (%)	L2 busy rate (%)	Memory busy rate (%)	Address calculation operation pipeline A busy rate (%)	Address calculation operation pipeline B busy rate (%)	Floating-point pipeline A Active element rate (%)	Floating-point pipeline B Active element rate (%)	L1 pipeline 0 Active element rate (%)	L1 pipeline 1 Active element rate (%)	SFI(Store Fetch Interlock) rate
Process	Thread	🟦	🟦	🟦	🟦	🟧	🟥	🟥							
0	0	95.02%	94.74%	6.07%		60%			62.81%	60.80%	99.67%	100.00%	100.00%	100.00%	0.00
0	1	95.02%	94.74%							%	99.67%	100.00%	100.00%	100.00%	0.00
0	2	95.02%	94.74%							%	99.67%	100.00%	100.00%	100.00%	0.00
0	3	95.02%	94.74%							%	99.67%	100.00%	100.00%	100.00%	0.00
0	4	95.02%	94.74%							%					
0	5	95.02%	94.74%							%					
0	6	95.02%	94.74%												
0	7	95.02%	94.74%	6.01%	4.05%	77.56%			62.94%	60.77%					
0	8	95.02%	94.74%	6.00%	4.05%	77.57%			62.95%	60.78%					
0	9	95.02%	94.74%	5.94%	3.97%	77.57%			63.04%	60.83%					
0	10	95.02%	94.74%	5.94%	3.97%	77.57%			63.04%	60.83%					
0	11	95.02%	94.74%	5.98%	4.02%	77.56%			62.98%	60.79%					
CMG 0 total		95.02%	94.74%	6.03%	4.08%	77.56%	61.93%	6.27%	62.89%	60.78%					

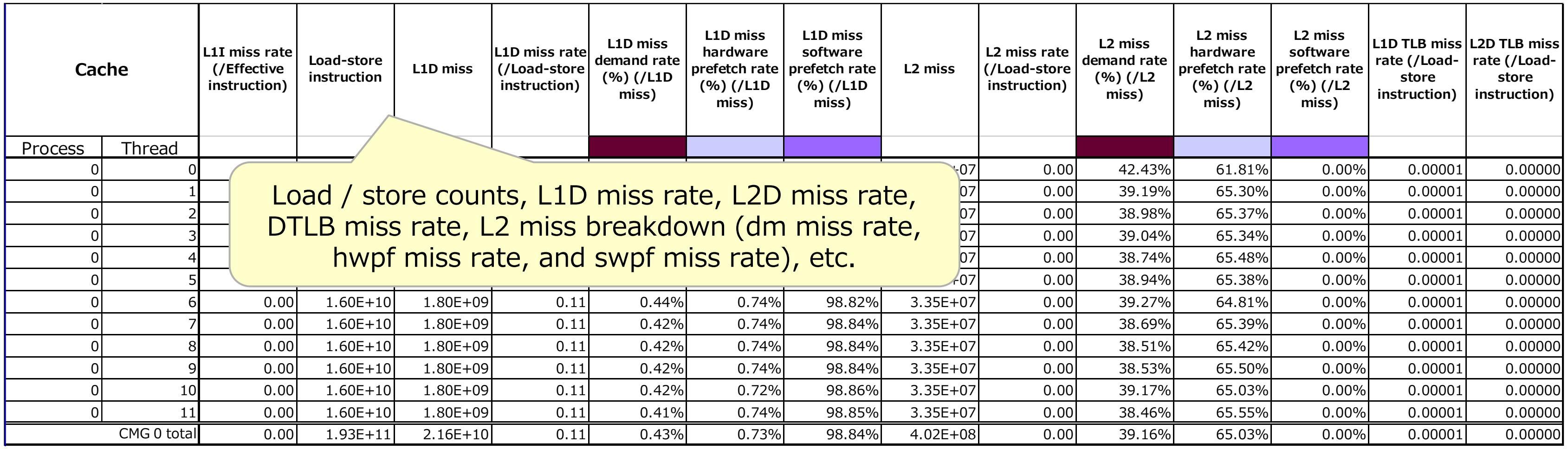
L1 busy rate, L2 busy rate, memory busy rate, floating-point operation pipeline busy rate, integer operation pipeline busy rate, etc.

- About the memory busy rate

  The value used as the denominator in the K/FX100 PA sheet has been changed (theoretical performance value of memory bandwidth per CMG (256 GB/s)).

  A rate of 80% or so is regarded as a busy state.

Cache miss status



Cache		L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)	L1D TLB miss rate (/Load-store instruction)	L2D TLB miss rate (/Load-store instruction)
Process	Thread					🟥	🟪	🟪			🟥	🟪	🟪		
0	0								+07	0.00	42.43%	61.81%	0.00%	0.00001	0.00000
0	1								07	0.00	39.19%	65.30%	0.00%	0.00001	0.00000
0	2								07	0.00	38.98%	65.37%	0.00%	0.00001	0.00000
0	3								07	0.00	39.04%	65.34%	0.00%	0.00001	0.00000
0	4								07	0.00	38.74%	65.48%	0.00%	0.00001	0.00000
0	5								+07	0.00	38.94%	65.38%	0.00%	0.00001	0.00000
0	6	0.00	1.60E+10	1.80E+09	0.11	0.44%	0.74%	98.82%	3.35E+07	0.00	39.27%	64.81%	0.00%	0.00001	0.00000
0	7	0.00	1.60E+10	1.80E+09	0.11	0.42%	0.74%	98.84%	3.35E+07	0.00	38.69%	65.39%	0.00%	0.00001	0.00000
0	8	0.00	1.60E+10	1.80E+09	0.11	0.42%	0.74%	98.84%	3.35E+07	0.00	38.51%	65.42%	0.00%	0.00001	0.00000
0	9	0.00	1.60E+10	1.80E+09	0.11	0.42%	0.74%	98.84%	3.35E+07	0.00	38.53%	65.50%	0.00%	0.00001	0.00000
0	10	0.00	1.60E+10	1.80E+09	0.11	0.42%	0.72%	98.86%	3.35E+07	0.00	39.17%	65.03%	0.00%	0.00001	0.00000
0	11	0.00	1.60E+10	1.80E+09	0.11	0.41%	0.74%	98.85%	3.35E+07	0.00	38.46%	65.55%	0.00%	0.00001	0.00000
CMG 0 total		0.00	1.93E+11	2.16E+10	0.11	0.43%	0.73%	98.84%	4.02E+08	0.00	39.16%	65.03%	0.00%	0.00001	0.00000

Load / store counts, L1D miss rate, L2D miss rate, DTLB miss rate, L2 miss breakdown (dm miss rate, hwpf miss rate, and swpf miss rate), etc.

Image of the Displayed CPU Analysis Report (Table 3)

Instruction mix

Instruction		Load-store instruction													
		Load instruction								Store instruction					
		SIMD							Non-SIMD	SIMD					Non-SIMD
		Single vector contiguous load instruction	Multiple vector contiguous structure load instruction	Non-contiguous gather load instruction	Broadcast load instruction	Floating-point register fill instruction	Predicate register fill instruction	First-fault load instruction	Non-SIMD load instruction	Single vector contiguous store instruction	Multiple vector contiguous structure store instruction	Non-contiguous scatter store instruction	Floating-point register spill instruction	Predicate register spill instruction	Non-SIMD store instruction
Process	Thread														
0	0	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	1.59E+06	5.84E+07	0.00E+00	0.00E+00	2.99E+03	1.62E+03	6
0	1	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	6.99E+05	5.84E+07	0.00E+00	0.00E+00	2.99E+03	1.62E+03	[illegible]
0	2	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	6.87E+05	5.84E+07	0.00E+00	0.00E+	[illegible]	[illegible]	[illegible]
0	3	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	6.94E+05	5.84E+07	0.00E+00	0.00E	[illegible]	[illegible]	[illegible]
0	4	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	6.82E+05	5.84E+07	0.00E+00	0.00E	[illegible]	[illegible]	[illegible]
0	5	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	6.97E+05	5.84E+07	0.00E+00	0.00E	[illegible]	[illegible]	[illegible]
0	6	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	6.92E+05	5.84E+07	0.00E+00	0.00E	[illegible]	[illegible]	[illegible]
0	7	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	7.00E+05	5.84E+07	0.00E+00	0.00E	[illegible]	[illegible]	[illegible]
0	8	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	6.91E+05	5.84E+07	0.00E+00	0.00E	[illegible]	[illegible]	[illegible]
0	9	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	7.19E+05	5.84E+07	0.00E+00	0.00E	[illegible]	[illegible]	[illegible]
0	10	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	6.90E+05	5.84E+07	0.00E+00	0.00E+0	[illegible]	[illegible]	[illegible]
0	11	7.14E+09	0.00E+00	0.00E+00	8.85E+09	3.28E+03	1.86E+03	0.00E+00	6.95E+05	5.82E+07	0.00E+00	0.00E+00	3.04E+03	1.62E+03	2.44E+05
CMG 0 total		8.56E+10	0.00E+00	0.00E+00	1.06E+11	3.93E+04	2.24E+04	0.00E+00	9.24E+06	7.00E+08	0.00E+00	0.00E+00	3.60E+04	1.94E+04	3.35E+06
		1.93E+11													

Instruction		Prefetch instruction			DCZVA instruction	Floating-point instruction			Floating-point move and conversion instruction		Integer instruction	Branch instruction	Predicate instruction	Crypto-graphic instruction	Other instruction	Total
		Contiguous prefetch instruction	Gathering prefetch instruction	Scalar prefetch instruction		Floating-point instruction except FMA and reciprocal	FMA instruction	Floating-point reciprocal instruction	Floating-point conversion instruction	Floating-point move instruction						
Process	Thread															
0	0	[illegible]	0.00E+00	1.81E+09	8.24E+02	1.56E+06	3.54E+10	0.00E+00	9.00E+01	4.22E+03	0.00E+00	8.90E+08	1.24E+04	0.00E+00	8.03E+09	6.22E+10
0	1	[illegible]	[illegible]	1.81E+09	8.24E+02	1.56E+06	3.54E+10	0.00E+00	9.00E+01	4.21E+03	0.00E+00	8.89E+08	1.24E+04	0.00E+00	8.02E+09	6.22E+10
0	2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	89E+08	1.24E+04	0.00E+00	8.02E+09	6.22E+10
0	3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E+08	1.24E+04	0.00E+00	8.02E+09	6.22E+10
0	4	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E+08	1.24E+04	0.00E+00	8.02E+09	6.22E+10
0	5	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E+08	1.24E+04	0.00E+00	8.02E+09	6.22E+10
0	6	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E+08	1.24E+04	0.00E+00	8.02E+09	6.22E+10
0	7	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E+08	1.24E+04	0.00E+00	8.02E+09	6.22E+10
0	8	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E+08	1.24E+04	0.00E+00	8.02E+09	6.22E+10
0	9	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E+08	1.24E+04	0.00E+00	8.02E+09	6.22E+10
0	10	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	.89E+08	1.24E+04	0.00E+00	8.02E+09	6.22E+10
0	11	0.00E+00	0.00E+00	1.81E+09	8.24E+02	1.33E+06	3.54E+10	0.00E+00	9.00E+01	4.21E+03	0.00E+00	8.89E+08	1.24E+04	0.00E+00	8.02E+09	6.22E+10
CMG 0 total		0.00E+00	0.00E+00	2.17E+10	9.89E+03	1.85E+07	4.25E+11	0.00E+00	1.08E+03	5.05E+04	0.00E+00	1.07E+10	1.49E+05	0.00E+00	9.63E+10	7.46E+11
		2.17E+10			9.89E+03	4.25E+11			5.16E+04		0.00E+00	1.07E+10	1.49E+05	0.00E+00	9.63E+10	7.46E+11

Load / store instruction, prefetch instruction, floating-point operation instruction, integer operation instruction, branch instruction, etc.

Performance data

FLOPS		Double precision floating-point operation	Single precision floating-point operation	Half precision floating-point operation	GFLOPS by Active element rate
Process	Thread				
0	0	11	0.00.E+00	0.00.E+00	60.59
0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
0	6	5.66.E+11	0.00.E+00	0.00.E+00	60.59
0	7	5.66.E+11	0.00.E+00	0.00.E+00	60.59
0	8	5.66.E+11	0.00.E+00	0.00.E+00	60.59
0	9	5.66.E+11	0.00.E+00	0.00.E+00	60.59
0	10	5.66.E+11	0.00.E+00	0.00.E+00	60.59
0	11	5.66.E+11	0.00.E+00	0.00.E+00	60.59
CMG 0 total		6.79.E+12	0.00.E+00	0.00.E+00	727.03

GFLOPS values based on floating-point operation rate by type (half precision, single precision, and double precision), predicate mask information

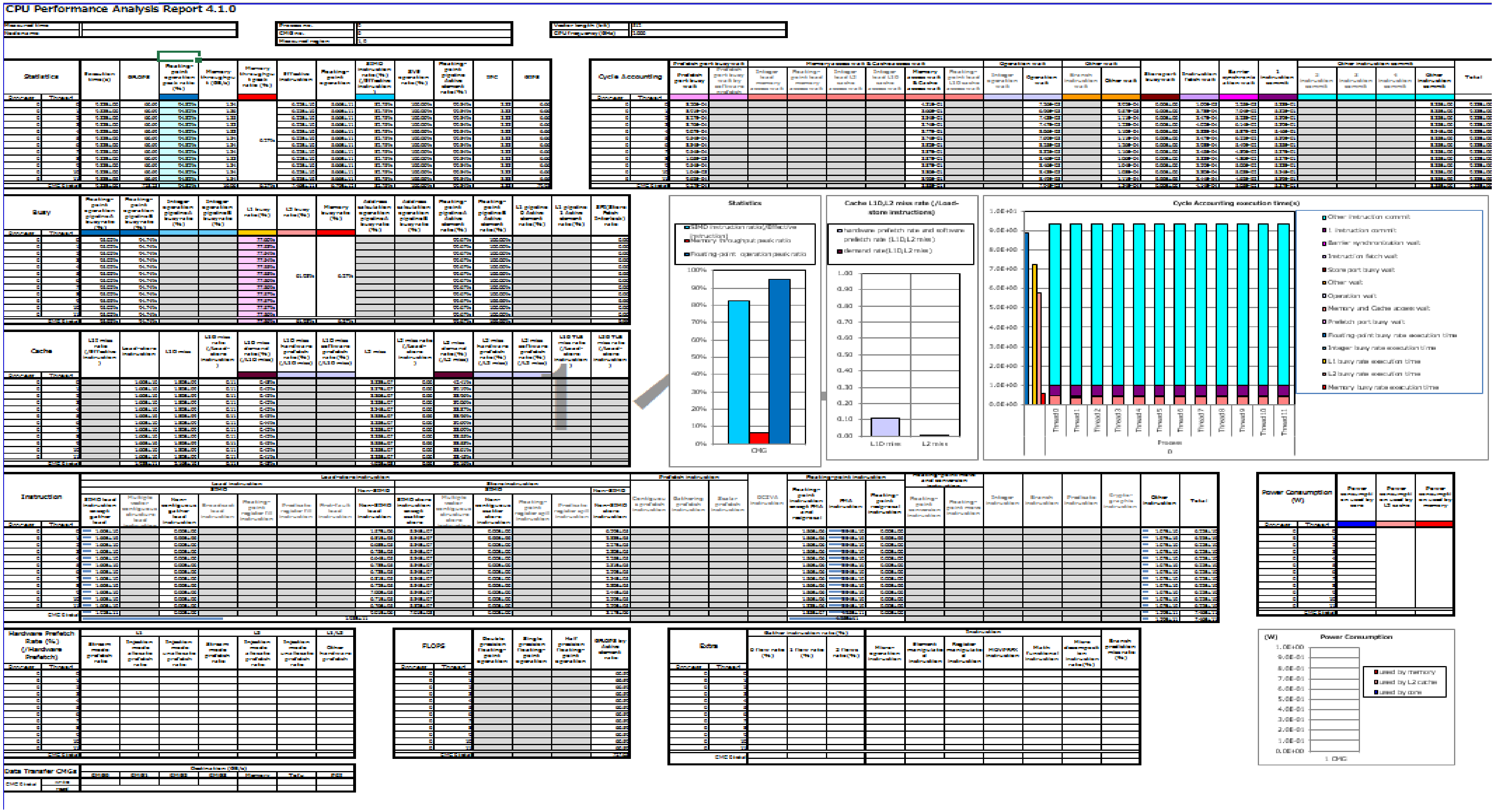
- Floating-point operation rate by type

  Of the floating-point operations performed during measurement intervals, the floating-point operation rate by type (half precision, single precision, or double precision) is displayed as %.

- GFLOPS values based on predicate mask information

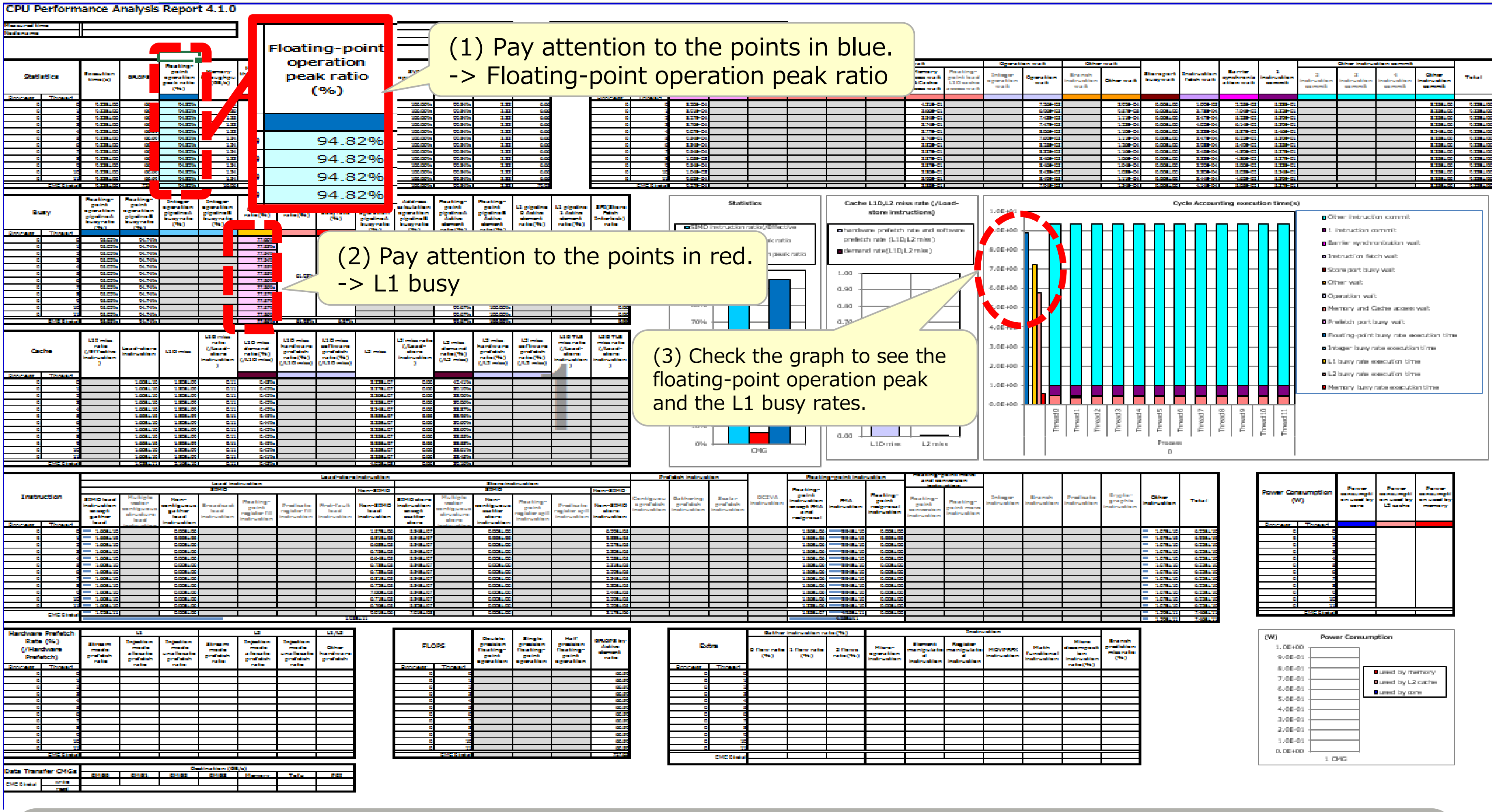
  GFLOPS values calculated using predicate mask information (The predicate mask information values are for reference only since they are not only used for operations.)

Display Example
(DGEMM: Simple Data (Five Measurements))



CPU Performance Analysis Report 4.1.0

Analysis Example (DGEMM: Simple Data (Five Measurements))



CPU Performance Analysis Report 4.1.0

Floating-point operation peak ratio (%)
94.82%
94.82%
94.82%
94.82%

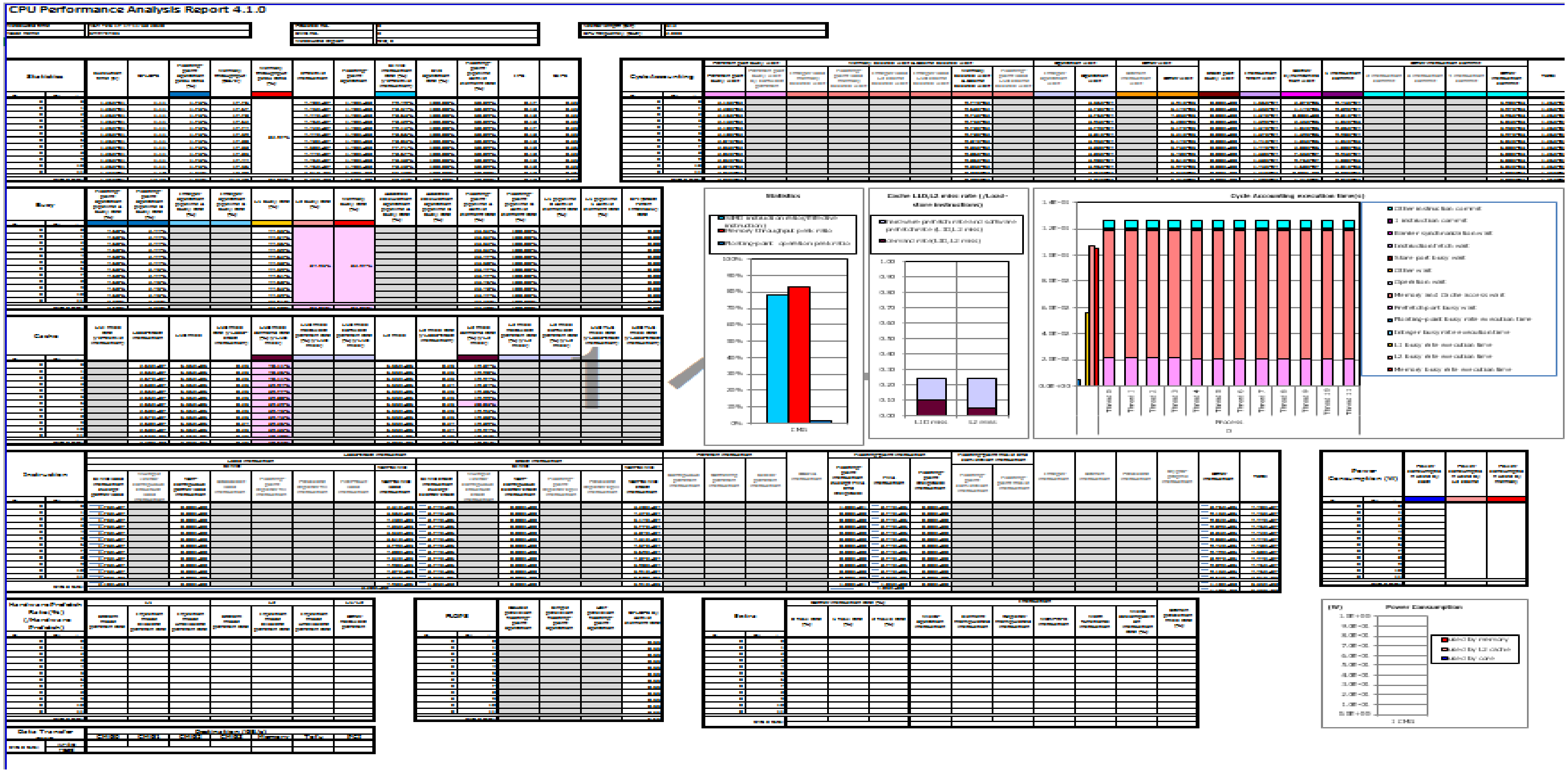
(1) Pay attention to the points in blue.
-> Floating-point operation peak ratio

(2) Pay attention to the points in red.
-> L1 busy

(3) Check the graph to see the floating-point operation peak and the L1 busy rates.

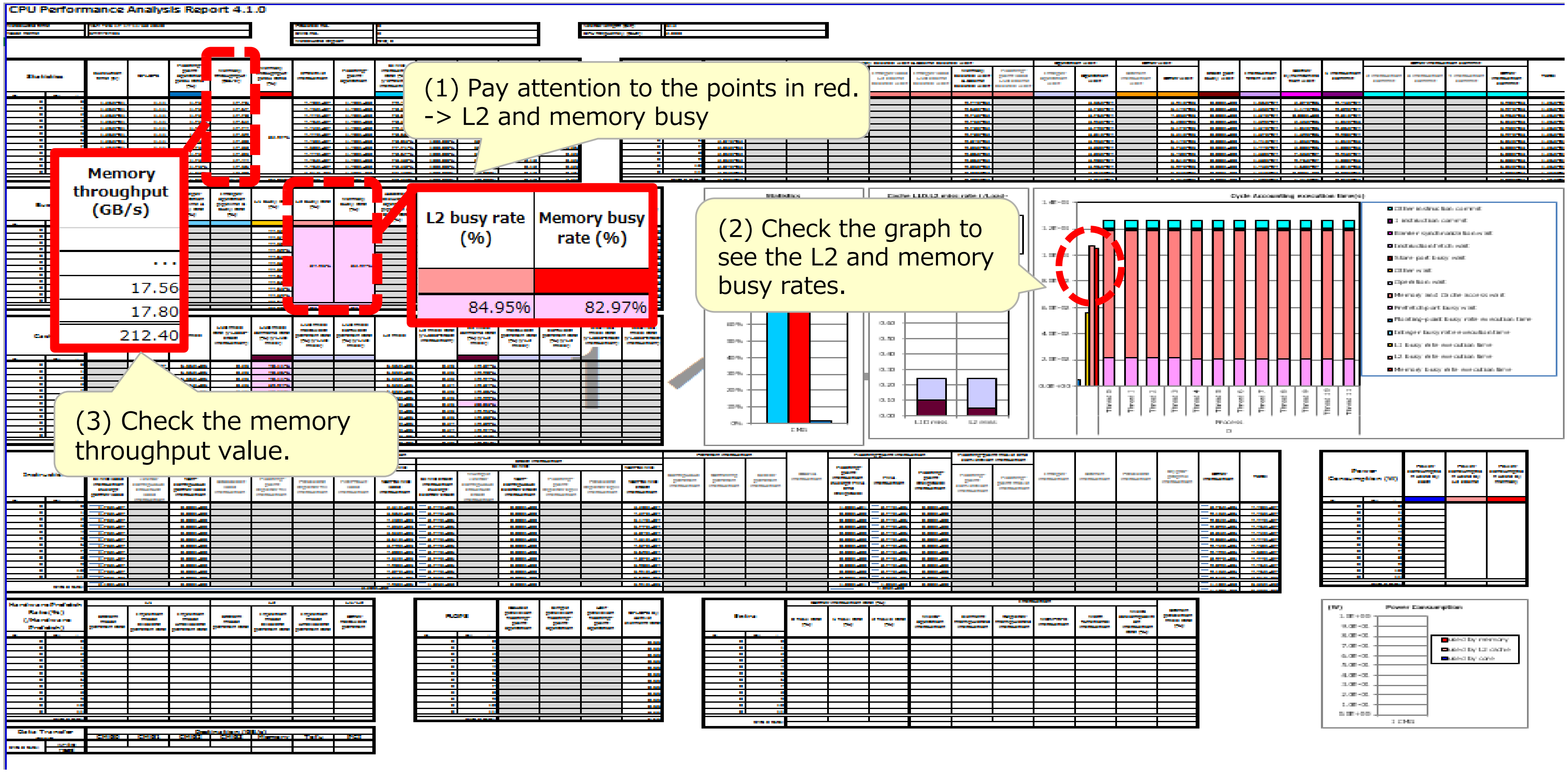
The floating-point peak ratio ((1)) and the L1 busy ratio ((2)) are high. From the graph ((3)), it can be seen that the floating-point peak ratio is higher. Its value is 94.82%, which indicates that high operation performance can be achieved.

Display Example
(STREAM: Simple Data (Five Measurements))



CPU Performance Analysis Report 4.1.0

Analysis Example
(STREAM: Simple Data (Five Measurements))



CPU Performance Analysis Report 4.1.0

(1) Pay attention to the points in red.
-> L2 and memory busy

L2 busy rate (%)	Memory busy rate (%)
84.95%	82.97%

(2) Check the graph to see the L2 and memory busy rates.

Memory throughput (GB/s)
...
17.56
17.80
212.40

(3) Check the memory throughput value.

From the busy information ((1)) and the graph ((2)), it is found that the L2 busy rate and memory busy rate are high.
When the value ((3)) is checked, it is found to be 207.69 GB/s. This indicates that performance equivalent to over 80% of the memory bandwidth per CMG (256 GB/s) is achieved.

Notes on FLOPS

About the count of the number of operations

There are cases in which more operations are counted in the CPU performance analysis report. Therefore, the FLOPS value may become higher than the actual one.
For example, when a masked SIMD instruction is used, all operations are counted as true even if some masks (predicate) are false. In this case, the number of operations becomes greater than it actually is.
The cases in which the count of the number of operations becomes greater than the actual number are summarized below, including the one mentioned above.

Item	Overview	Difference from K/FX100	Prevention method
Floating-point division/SQRT function	Since the compiler replaces an operation with a sequence of multiple operation instructions, more operations are counted than seen on the code.	Same	Specify -Knofp_relaxed.
Mathematical function/numeric function			None
Reduction	If an automatic loop slicing occurs in a loop that includes a reduction operation, more operations are counted than seen on the code.	Same	Specify -Knoreduction.
Conversion of a loop including an IF construct into a SIMD instruction	When a loop including an IF construct is optimized using a masked SIMD instruction, all operations are counted as true even if some masks (predicate) are false.	Same as -Ksimd=2	Specify 1 in -Ksimd.
SIMD conversion through redundant loop execution	When a loop is optimized using a masked SIMD instruction in cases where the number of loop repetitions cannot be divided by the SIMD length, all operations are counted as true even if some masks (predicate) are false.	New	Specify the following OCL. SIMD_NOREDUNDANT_VL

Large Page

- About the Large Page
- Large Page Specifications
- Environment Variables for Large Page Setting
  - Basic Setting/Paging Policy Setting
  - Tuning Setting
- Large Page Evaluation
  - Stream Triad
  - ARENA_FREE Performance
  - PAGING_POLICY Performance
  - ARENA_LOCK_TYPE Performance

About the Large Page

About the large page

- It refers to the allocation of memory in a larger page size (large page) than the normal page to an application that handles a large volume of data. The large page:
  - Reduces the overhead of the CPU address translation processing.
  - Improves the memory access performance.

- In the A64FX system environment, the normal page size is 64 KiB and the size available for the large page is 2 MiB.
  - The following operations can be set with environment variables.
    - ✓ Enabling and disabling of the large page allocation operation
    - ✓ Enabling and disabling of the large page allocation operation in a stack area
    - ✓ Selection of the paging policy (page allocation trigger) for each memory area
  - Various page sizes, such as 32 MiB, 1 GiB, and 16 GiB, can be implemented using McKernel.

Large Page Specifications

Large page specifications

Memory area	MP10/FX10/ FX100	A64FX			
	Page size	Page size			Paging (The default is underlined.)
		Normal page	Large page base	Large page base+stack (default)	
Text (.text)	8KiB	64KiB	64KiB	64KiB	–
Static data (.data)	4MiB (default), 8KiB, 32MiB, 256MiB	64KiB	2MiB	2MiB	Always prepage
Static data (.bss)		64KiB	2MiB	2MiB	demand | prepage
Stack (*1)		64KiB	64KiB	2MiB	demand | prepage
Dynamic memory (*2)		64KiB	2MiB	2MiB	demand | prepage
Shared memory		64KiB	64KiB	64KiB	–

*1: For the process stack, main thread stack, and thread stack areas
*2: For the process heap, main thread heap, thread heap, and mmap areas

Environment Variables for Large Page Setting (1/2)

Basic setting/paging policy setting

Environment variable name	Specifiable value (The default is underlined.)	Explanation
XOS_MMM_L_HPAGE_TYPE	hugetlbfs | none	This setting is used to select whether to enable or disable the large page allocation operation using the large page library. If “hugetlbfs” is specified, HugeTLBfs makes large pagination. If “none” is specified, the large page library is not used for large pagination.
XOS_MMM_L_LPG_MODE	base+stack | base	This setting is used to select whether to enable or disable the large page allocation operation in a stack area and thread stack area. If “base+stack” is specified, large pagination is done not only in static data and dynamic memory allocation areas but also in a stack area and thread stack area. If “base” is specified, large pagination is done only in static data and dynamic memory allocation areas, but not in a stack area or thread stack area.
XOS_MMM_L_PAGING_POLICY	[demand | prepage]: [demand | prepage]: [demand | prepage]	This setting is used to select the paging policy (page allocation trigger) for each memory area. “demand” means the demand paging policy, and “prepage” means the prepaging policy. This variable lets you specify paging policies for three different memory areas using colons (:) as delimiters. The first specified policy is for the .bss area of static data. (The paging policy specification is not applicable to the .data area of static data, and prepage is always specified.) The second specified policy is for the stack area and thread stack area. The third specified policy is for the dynamic memory allocation area. If a value other than the specifiable value is specified, “prepage:demand:prepage” is assumed.

Environment Variables for Large Page Setting (2/2)

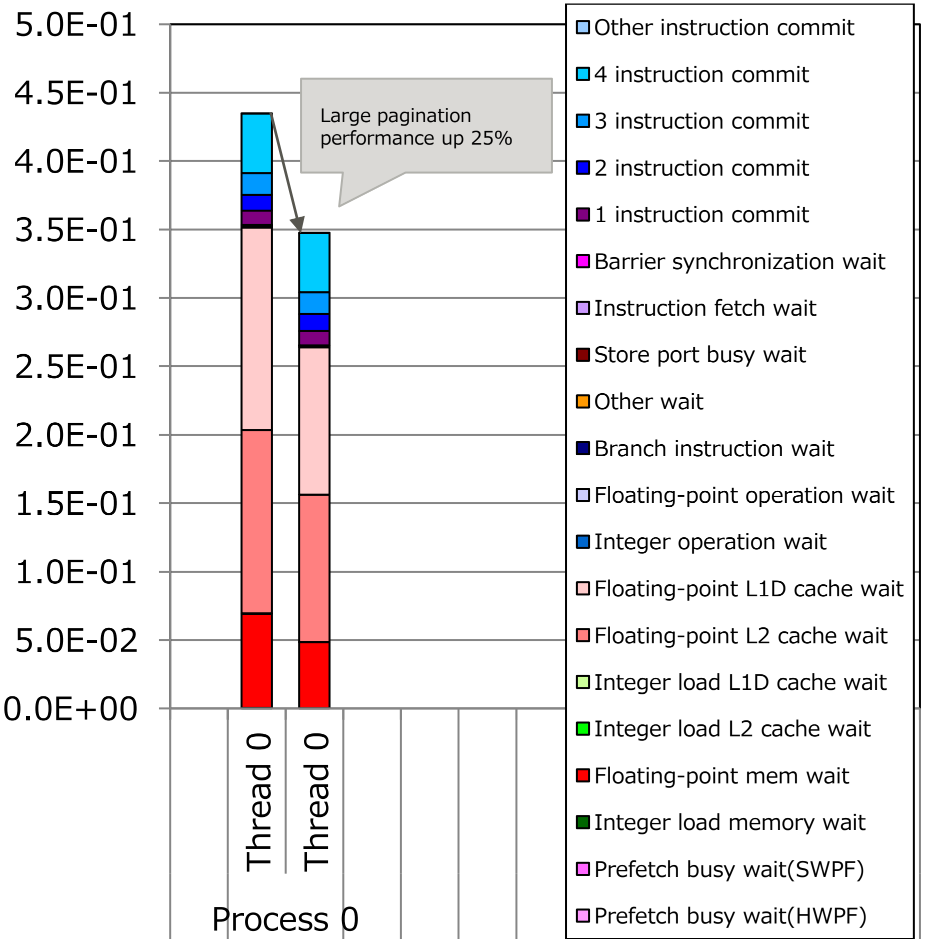
Tuning setting (environment variables unique to the large page library)

Environment variable name	Specifiable value (The default is underlined.)	Explanation
XOS_MMM_L_ARENA_FREE	1 | 2	This setting concerns the handling of the heap area that is freed by free(3). If “1” is specified, the memory that can be freed is immediately freed. If “2” is specified, no memory is freed and all the memory is pooled and reused.
XOS_MMM_L_ARENA_LOCK_TYPE	0 | 1	This setting concerns the memory allocation policy. “0” means that priority is on memory allocation performance. “1” means that priority is on memory usage efficiency.
XOS_MMM_L_MAX_ARENA_NUM	Integer [in decimal notation] from 1 or more to a value equal to or less than INT_MAX	Set the number of arenas that can be generated (the total of the process heap and thread heap areas). This is valid when 0 is specified in XOS_MMM_L_ARENA_LOCK_TYPE.
XOS_MMM_L_HEAP_SIZE_MB	Integer <in MiB> [in decimal notation] from a value twice MALLOC_MMAP_THRESHOLD or more to a value equal to or less than ULONG_MAX	When using a thread heap area, set the size of memory to be allocated for generating or expanding the thread heap area.
XOS_MMM_L_COLORING	0 | 1	Set whether to enable or disable cache coloring. This reduces the conflict of the L1 cache of the processor. If “0” is specified, cache coloring is not performed. If “1” is specified, cache coloring is performed when memory of a size equal to or larger than MALLOC_MMAP_THRESHOLD_ (the default is 128 MiB) is allocated by mmap(2).
XOS_MMM_L_FORCE_MMAP_THRESHOLD	0 | 1	This setting specifies whether to give priority to mmap(2) when allocating memory of a size equal to or larger than MALLOC_MMAP_THRESHOLD_ (the default is 128 MiB). If “0” is specified, priority is not given to mmap(2). First, the heap area is searched for free space. If there is any free space, the free memory space of the heap area is returned. Memory is allocated by mmap(2) only when free space cannot be found in the heap area. If “1” is specified, priority is given to mmap(2). Memory is allocated by mmap(2) without searching the heap area for free space (even if there is free space).

- For the environment variables of glibc (MALLOC_MMAP_THRESHOLD_, etc.), see the *User’s Guide*.

Large Page Evaluation 1

Stream Triad (1 thread execution)



Large pagination performance up 25%

Estimation method	A64FX	A64FX
Kernelization/reduction	n=83880960	
Source code version	**Normal page**	**Large page**
Floating-point precision	Double precision real type	
SIMD width	8	
Aggregation thread number	0	
Execution time [s]	4.35E-01	3.47E-01
GFLOPS/process	3.86	4.83
Memory throughput [GB/s/process]	61.76	77.29
Number of executed instructions/thread [10^9]	5.11E+08	5.11E+08
Number of load / store instructions/thread [10^9]	3.15E+08	3.15E+08
Number of branch instructions/thread [10^9]	1.31E+07	1.31E+07
Number of other instructions/thread [10^9]	7.87E+07	7.87E+07
SIMD instruction rate/thread	82.03%	82.04%
L1D miss rate/thread	25.26%	25.00%
L2 miss rate/thread	25.00%	25.00%
L1D TLB miss rate/thread	0.09786%	0.00324%
L2D TLB miss rate/thread	0.09776%	0.00033%
L1 busy rate/process	73.95%	84.13%
L2 busy rate/process	16.77%	20.75%
Memory busy rate/process	24.13%	30.19%

- Large pagination reduces the L2D TLB miss rate, improving the performance by 25%.

Large Page Evaluation 2: ARENA_FREE Performance (1/2)

Measurement conditions

Condition	Pattern
Verification code	Code shown at the right (excerpt from the manual)
Number of threads to be measured	1 thread
Compiler	Compiler for the A64FX
Compilation option	-Kfast

Access range	N=1024
	MALLOC_CNT=1024

Evaluation conditions	The execution results are compared using the following two conditions. - XOS_MMM_L_ARENA_FREE = 1 (Default: Memory is freed.) - XOS_MMM_L_ARENA_FREE=2 (Memory is reused without being freed.)

Perform a loop of executing malloc and free twice.

Code to be measured

```
while(loop <2){
  printf("malloc start.¥n");
  clock_gettime(CLOCK_REALTIME, &time1);

  for(i=0;i<MALLOC_CNT;i++){
    c[i]=(double *)malloc(sizeof(double)*N*N);
    if (c[i] == NULL) {
      fprintf(stderr, "malloc error: cnt=%d, errno=%d¥n", i, errno);
      exit(1);
    }
  }

  clock_gettime(CLOCK_REALTIME, &time2);
  printf("malloc end.¥n");
  sec = (time2.tv_sec - time1.tv_sec);
  nsec= (time2.tv_nsec-time1.tv_nsec);
  if(nsec<0){
    sec--;
    nsec += 1000000000L;
  }
  printf("MALLOC TIME:%d:%010d¥n", sec, nsec);
  sleep(10);

  printf("free start.¥n");
  clock_gettime(CLOCK_REALTIME, &time1);

  for(i=0;i<MALLOC_CNT;i++){
    free(c[i]);
  }

  clock_gettime(CLOCK_REALTIME, &time2);
  printf("free end.¥n");
  sec = (time2.tv_sec - time1.tv_sec);
  nsec= (time2.tv_nsec-time1.tv_nsec);
  if(nsec<0){
    sec--;
    nsec += 1000000000L;
  }
  printf("FREE TIME:%d:%010d¥n", sec, nsec);
  loop++;
}
```

Repeat malloc 1024 times.

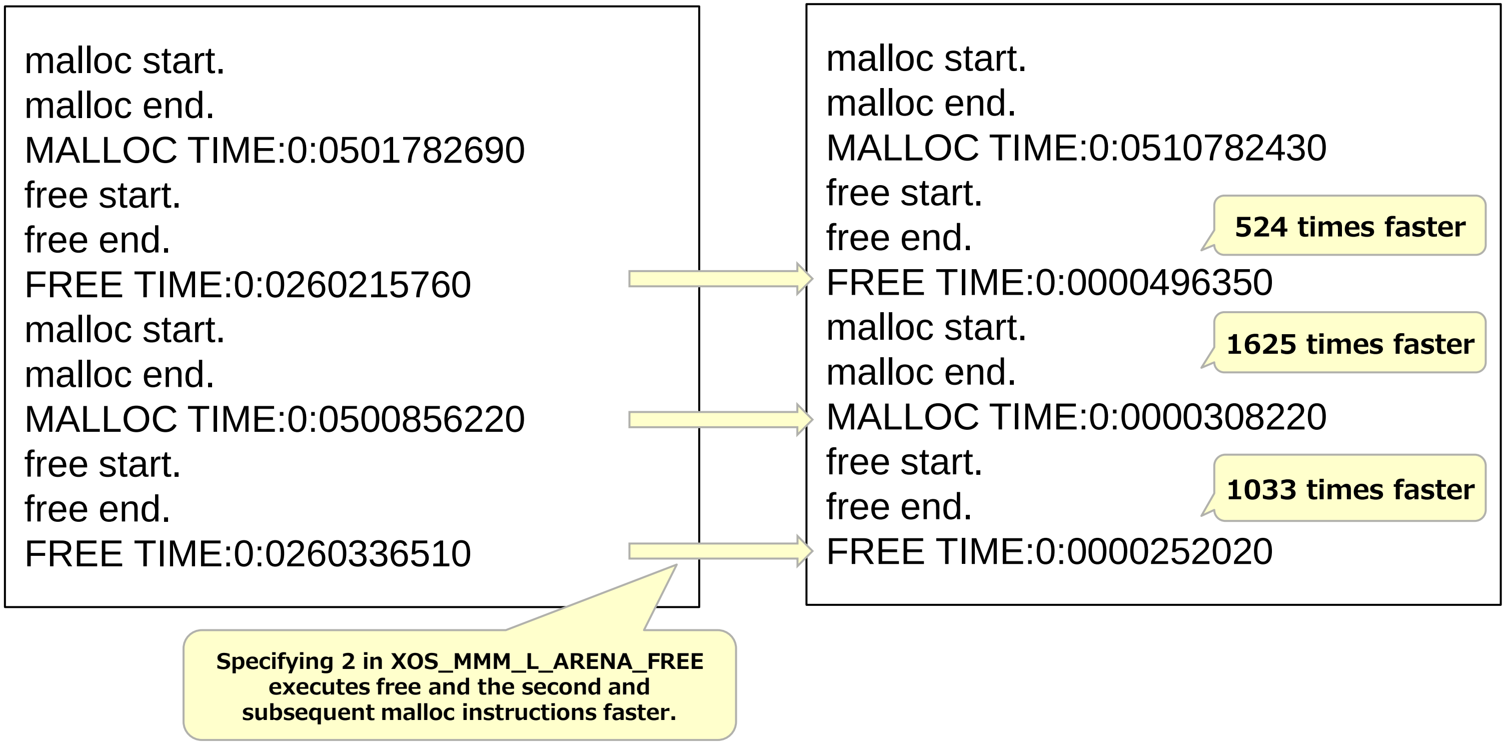
Print the elapsed time of malloc.

Repeat free 1024 times.

Print the elapsed time of free.

Large Page Evaluation 2: ARENA_FREE Performance (2/2)

- XOS_MMM_L_ARENA_FREE = 1 (Default: Memory is freed.)
- XOS_MMM_L_ARENA_FREE = 2 (Memory is reused without being freed.)



```
malloc start.
malloc end.
MALLOC TIME:0:0501782690
free start.
free end.
FREE TIME:0:0260215760
malloc start.
malloc end.
MALLOC TIME:0:0500856220
free start.
free end.
FREE TIME:0:0260336510
```

```
malloc start.
malloc end.
MALLOC TIME:0:0510782430
free start.
free end.
FREE TIME:0:0000496350
malloc start.
malloc end.
MALLOC TIME:0:0000308220
free start.
free end.
FREE TIME:0:0000252020
```

524 times faster

1625 times faster

1033 times faster

Specifying 2 in XOS_MMM_L_ARENA_FREE executes free and the second and subsequent malloc instructions faster.

XOS_MMM_L_ARENA_FREE is effective for a program in which malloc and free instructions of the same size are repeated.

Large Page Evaluation 3: PAGING_POLICY Performance

FUJITSU

Since data comes from CMG0 in prepaging, performance cannot reach that of 48-thread streams.

With the method changed to demand paging, data is put on the running CMG, and performance is significantly higher.

Source

```
14                    Subroutine sub(n,iter,x1,x2,y1)
15                      real(8) :: x1(n), x2(n), y1(n),c0
16                      integer n,i,k
17                      c0=2.0
18
19                      call fapp_start("sub",0,0)
20     1                 do k=1,iter
21     1               !$omp parallel do
                   <<< Loop-information Start >>>
                   <<<  [OPTIMIZATION]
                   <<<    SIMD(VL: 8)
                   <<<    SOFTWARE PIPELINING(IPC: 2.45, ITR:
128, MVE: 2, POL: S)
                   <<<    PREFETCH(SOFT) : 10
                   <<<     SEQUENTIAL : 10
                   <<<      x2: 4, x1: 4, y1: 2
                   <<<    ZFILL          :
                   <<<      y1
                   <<< Loop-information  End >>>
22     2   p   v         do i=1,n
23     2   p   v            y1(i) = x1(i) + c0 * x2(i)
24     2   p   v         end do
25     1                 enddo
 :                       ……
30                      parameter(N=45000000,ITER=100)
31                      real*8 x1(N),x2(N),y1(N)
32                      call init(N,ITER,x1,x2,y1)
33                      call sub(N,ITER,x1,x2,y1)
```

Stream (Data size: About 1 GB)

	Memory throughput (GB/s)
prepage (default)	93 GB/s
demand	804 GB/s

Compiler option: -Kfast,openmp -Kprefetch_sequential=soft -Kprefetch_line=9 -Kprefetch_line_L2=70 -Kzfill=18

Large Page Evaluation 4: LOCK_TYPE Performance

malloc performance is higher when XOS_MMM_L_ARENA_LOCK_TYPE=0 is specified.
(Reduced execution time from 0.56 seconds to 0.35 seconds, a performance increase of 1.60 times)

Source

```
1                  subroutine sub(n,m,iter,x1,x2,y2)
2                    integer(8) :: pZ1(iter)
3                    real(8) :: x1(n), x2(n), y2(n,m),c0
4                    c0=2.0
5
6                  !$omp parallel do shared(n,m,iter,x1,x2,c0,y2) private(pZ1,i,j,k) default(none)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      x1, x2, y2
                <<< Loop-information  End >>>
7    1   p               do k=1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      (unknown)
                <<< Loop-information  End >>>
8    2   p   s             do j=1,iter
9    2   p   m               pZ1(j) = malloc(8 * n)
10    2   p   v            end do
11    1
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.50, ITR: 144, MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      x1, x2, y2
                <<< Loop-information  End >>>
12    2   p  2v            do i=1,n
13    2   p  2v              y2(i,k) = x1(i) + c0 * x2(i)
14    2   p  2v            end do
15    1
16    2   p   s            do j=1,iter
17    2   p   s              call free( pZ1(j))
18    2   p   s            end do
19    1   p              end do
20                 end subroutine sub
21
22                 program main
23                   parameter(N=1048512,ITER=80)
24                   real*8 x1(N),x2(N),y2(N,12)
25                   call sub(N,12,ITER,x1,x2,y2)
26                 end program main
```

FUJITSU

shaping tomorrow with you

Sector Cache

What is the Sector Cache?
How to Use the Sector Cache
Sector Cache: Case 1 (Before Improvement)
Sector Cache: Case 1 (Source Tuning)
Sector Cache: Case 2 (Before Improvement)
Sector Cache: Case 2 (Source Tuning)

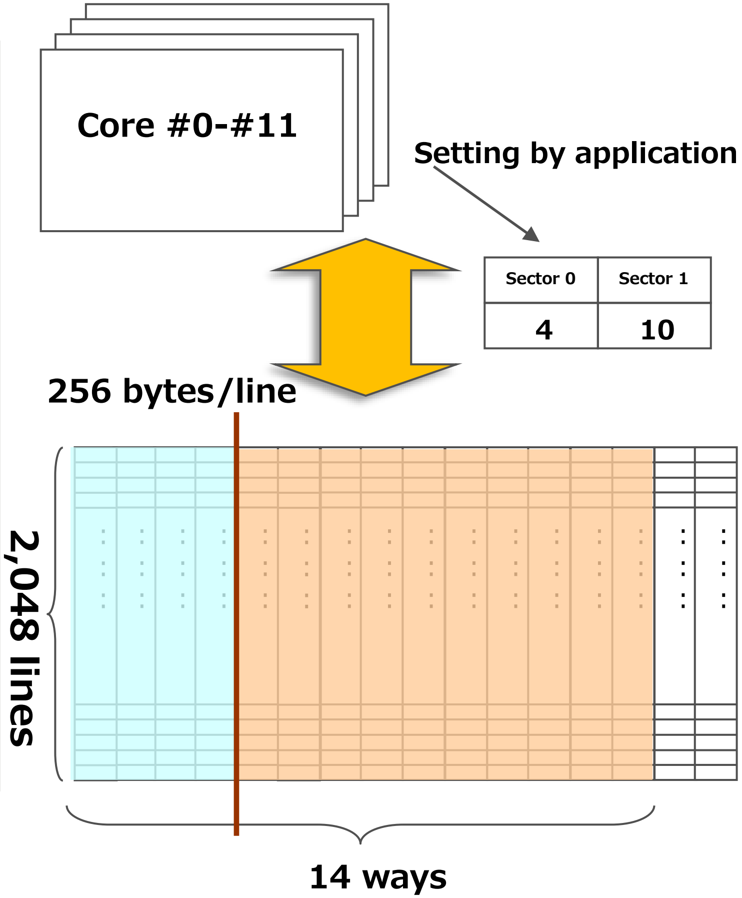
What is the Sector Cache?

The sector cache is a cache mechanism that can prevent non-reusable data from expelling reusable data from the cache. An application can allocate reusable data and non-reusable to different sectors. (Reusable arrays use Sector 1, and others use Sector 0.)

Sector cache details

- You can set multiple sectors in both the L1D cache and L2 cache. The maximum number of sectors is 4 in L1D and 2 in L2.
- The number of ways specifies the capacity of each sector.
- The capacity works as a target value.
  Hardware controls sectors so that they approach the specified capacity at the line replacement time.
  -> Not forcibly disabled even when over the capacity
- Use the LRU (least recently used) algorithm to control expulsion within a sector.
- Applications can decide the usage of sectors 0 and 1.
  However, Sector 0 stores instruction sequences.
- In a secondary cache, the assistant core always uses two ways.

Conceptual image of L2 cache usage



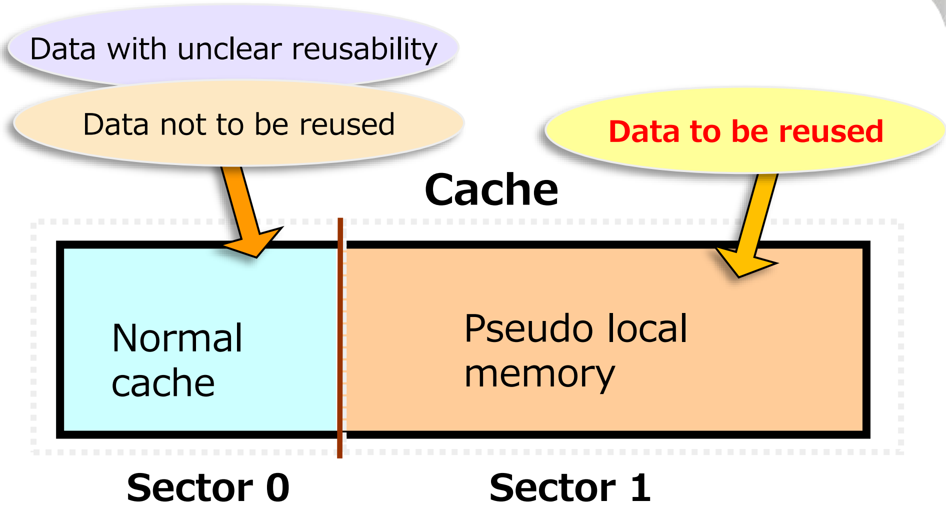
Sector 0	Sector 1
4	10

How to Use the Sector Cache (1/2)

■ Sector cache: Pseudo local memory

Software can use sectors separately according to data reusability.

- **Data used -> Use Sector 1**
- **Other data -> Use Sector 0**
- **Data in Sector 1 is not expelled by other data.**
- **Instruction lines can specify the arrays stored in Sector 1.**



Example using compiler instruction lines to specify the sector cache

```
!OCL SCACHE_ISOLATE_WAY(L2=10)
!OCL SCACHE_ISOLATE_ASSIGN(a)
do j=1,m
   do i=1,n
      a(i) = a(i) + b(i,j)*c(i,j)
   enddo
enddo
!OCL END_SCACHE_ISOLATE_ASSIGN
!OCL END_SCACHE_ISOLATE_WAY
```

How they are specified under the old specifications (K computer, FX100)

```
!OCL CACHE_SECTOR_SIZE(4,10)
!OCL CACHE_SUBSECTOR_ASSIGN(a)
do j=1,m
   do i=1,n
      a(i) = a(i) + b(i,j)*c(i,j)
   enddo
enddo
!OCL END_CACHE_SUBSECTOR
!OCL END_CACHE_SECTOR_SIZE
```

<Purpose>
To prevent Array a, which is reusable, from being expelled from the cache due to access to Arrays b and c during the loop

How to Use the Sector Cache (2/2)

To use the sector cache, specify the following optimization control lines.

Optimization Specifier	Meaning	Optimization Control Line Specifiable?			
		By Program	By DO Loop	By Statement	By Array Assignment Statement

SCACHE_ISOLATE_WAY(L2=n1[,L1=n2])
END_SCACHE_ISOLATE_WAY
	Specifies the maximum number of ways for Sector 1 of the primary cache and secondary cache.	Yes	No	Yes	No

SCACHE_ISOLATE_ASSIGN(array1[,array2]…)
END_SCACHE_ISOLATE_ASSIGN
	Specifies the arrays stored in Sector 1 of the cache.	Yes	No	Yes	No

■ **Note**

- **In the secondary cache, the assistant core always uses two ways. Therefore, the ranges of values that can be specified in n1 and n2 are as follows:**
  0 ≦ *n1* ≦ *maximum number of ways of secondary cache* - 2
  0 ≦ *n2* ≦ *maximum number of ways of primary cache*

- **For a CMG that contains an assistant core, the assistant core uses part (2 ways = 1 MiB) of the L2 cache. Therefore, for the CMG, the maximum number of ways of the secondary cache is 14 and the size is 7 MiB.**

A64FX Specifications	
Number of CMGs	4
L1I cache size	64 KiB/4 ways
L1D cache size	64 KiB/4 ways
L2 cache size	32 MiB/16 ways (8 MiB/CMG)

Sector Cache: Case 1 (Before Improvement)

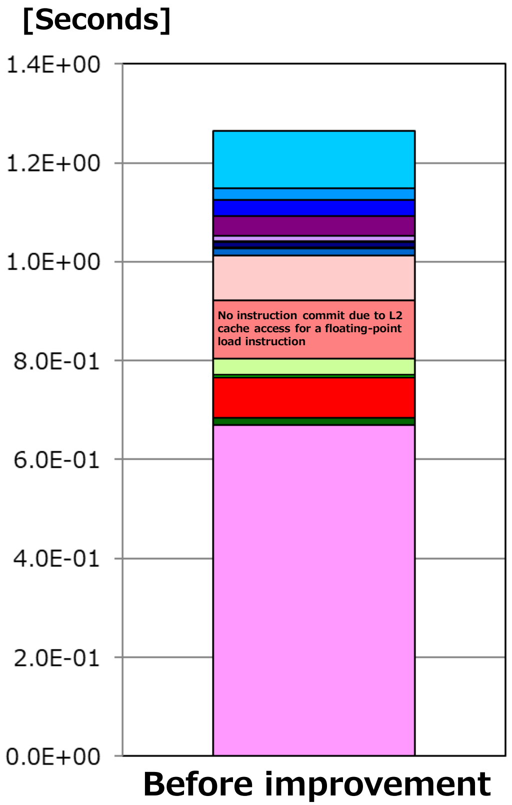
FUJITSU

Array b data is expelled from the cache and thus cannot be reused. Consequently, the "No instruction commit due to L2 cache access for a floating-point load instruction" event occurs many times.

Source Before Improvement

```
66                    parameter(n=8*1024*1024, m=9*512*1024/8)
67                    real*8    a(n), b(m), s
68                    integer*8 c(n)
69                    real*8    dummy1(140),dummy2(140)
70                    common /data/a,dummy1,c,dummy2,b
71
              <<< Loop-information Start >>>
              <<<  [PARALLELIZATION]
              <<<    Standard iteration count: 843
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 176, MVE: 4, POL: S)
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      c, a
              <<< Loop-information  End >>>
72    1  pp   2v          do i=1,n
73    1   p   2v            a(i) = a(i) + s * b(c(i))
74    1   p   2v          enddo
```

Array size
a: 64 MiB
b: 4.5 MiB
c: 64 MiB



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before improvement	0.00	4.76E+09	7.89E+08	0.17	0.89%	99.11%	0.00%	7.34E+08	0.15	0.77%	100.00%	0.00%

	Memory throughput (GB/s)
Before improvement	203.11

Memory throughput is bottleneck

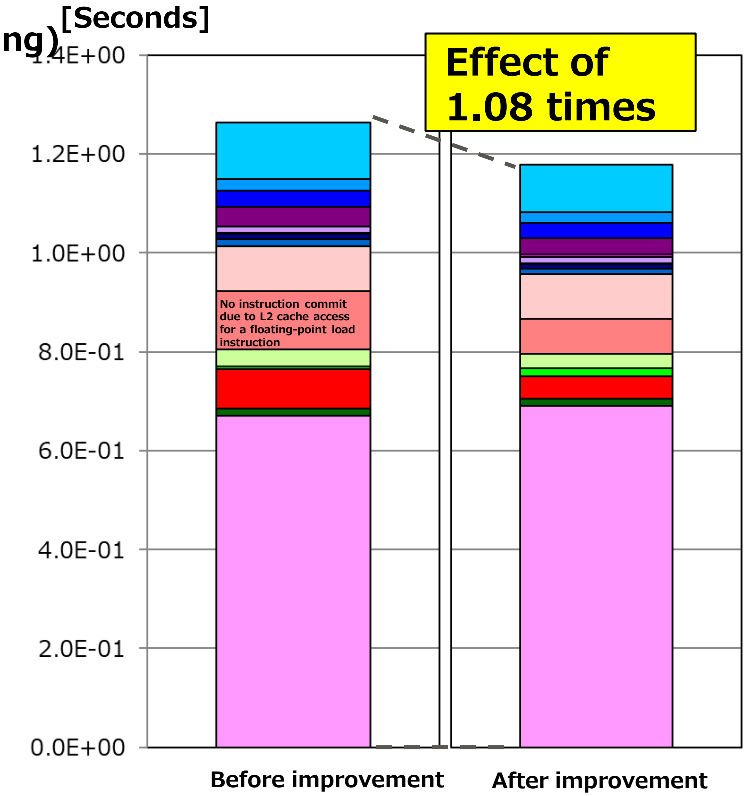
High L2 cache miss rate

Sector Cache: Case 1 (Source Tuning)

Storing Array b in Sector 1 increases cache efficiency. The result is improvement of the "No instruction commit due to L2 cache access for a floating-point load instruction" event.

Source After Improvement (Optimization Control Line Tuning)

```
58                   parameter(n=8*1024*1024, m=9*512*1024/8)
 59                   real*8    a(n), b(m), s
 60                   integer*8 c(n)
 61                   real*8    dummy1(140),dummy2(140)
 62                   common /data/a,dummy1,c,dummy2,b
 63
 64              !OCL SCACHE_ISOLATE_WAY(L2=10)
 65              !OCL SCACHE_ISOLATE_ASSIGN(b)
                 <<< Loop-information Start >>>
                 <<<  [PARALLELIZATION]
                 <<<    Standard iteration count: 843
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 176, MVE: 4, POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      c, a
                 <<< Loop-information  End >>>
 66     1  pp   2v        do i=1,n
 67     1   p   2v          a(i) = a(i) + s * b(c(i))
 68     1   p   2v        enddo
 69              !OCL END_SCACHE_ISOLATE_ASSIGN
 70              !OCL END_SCACHE_ISOLATE_WAY
```



Effect of 1.08 times

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%)(/L1D miss)	L1D miss hardware prefetch rate (%)(/L1D miss)	L1D miss software prefetch rate (%)(/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%)(/L2 miss)	L2 miss hardware prefetch rate (%)(/L2 miss)	L2 miss software prefetch rate (%)(/L2 miss)
Before improvement	0.00	4.76E+09	7.89E+08	0.17	0.89%	99.11%	0.00%	7.34E+08	0.15	0.77%	100.00%	0.00%
After improvement	0.00	5.19E+09	7.93E+08	0.15	1.19%	98.81%	0.01%	5.99E+08	0.12	1.93%	99.69%	0.00%

	Memory throughput (GB/s)
Before improvement	203.11
After improvement	188.07

L2 misses reduced

Sector Cache: Case 2 (Before Improvement)

Array u data is expelled from the cache and thus cannot be reused. Consequently, the "No instruction commit because memory cache is busy" event occurs many times.

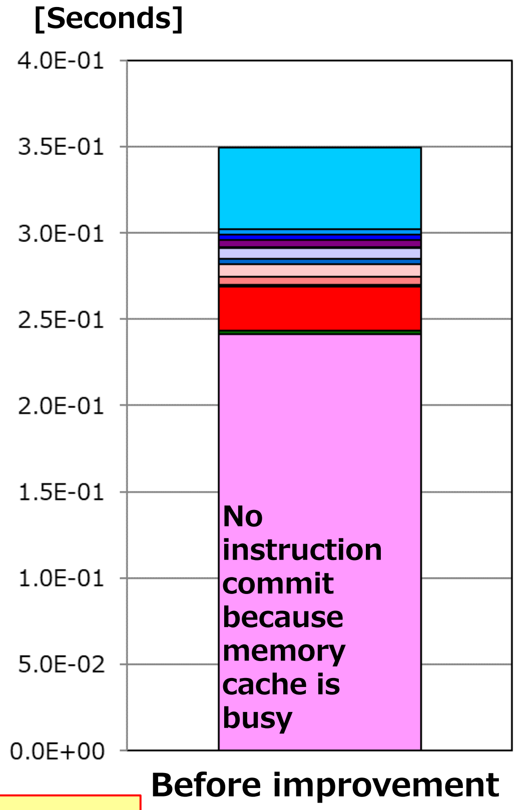
Source Before Improvement

```
167    1   s          do iter = 1, niter
                 <<< Loop-information Start >>>
                 <<<  [PARALLELIZATION]
                 <<<    Standard iteration cou
                 <<< Loop-information  End >
168    2  pp            do k=1,n3-2
                 <<< Loop-information Start >
                 <<<  [OPTIMIZATION]
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      u, rhs, unew
                 <<< Loop-information  End >>
169    3   p              do j=1,n2-2
                 <<< Loop-information Start >>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 3.71, ITR: 136, MVE: 9, POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      u, rhs, unew
                 <<< Loop-information  End >>>
170    4   p   v             do i=1,n1-2
171    4   p   v               unew(i,j,k) = &
172    4                         ((u(i+1,j,k) + u(i-1,j,k)) * h1sqinv  &
173    4                         +(u(i,j+1,k) + u(i,j-1,k)) * h2sqinv &
174    4                         +(u(i,j,k+1) + u(i,j,k-1)) * h3sqinv &
175    4                         -rhs(i,j,k)) * hhhinv
176    4   p   v             end do
177    3   p              end do
178    2   p            end do
179    1          end do
```

n1=452
n2=52
n3=322

Array size
unew: 60.5 MB
u: 60.5 MB
rhs: 60.5 MB

Preferably, Array u is cached so that dimensions i and j of Array u are reusable.



Memory throughput is bottleneck

	Memory throughput (GB/s)
Before improvement	215.62

Cache	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before improvement	2.46E+08	0.15	2.57%	98.19%	0.00%

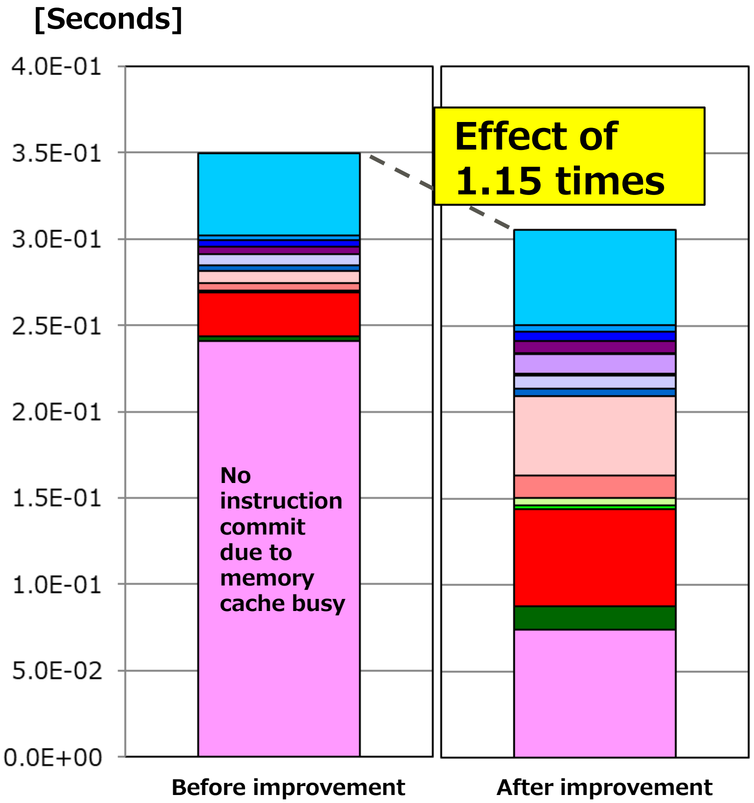
Sector Cache: Case 2 (Source Tuning)

Storing part of dimension k of Array u in Sector 1 increases cache efficiency. The result is improvement of the "No instruction commit because memory cache is busy" event.

Source After Improvement

```
166                !OCL SCACHE_ISOLATE_WAY(L2=13)
167                !OCL SCACHE_ISOLATE_ASSIGN(u)
168    1  s          do iter = 1, niter
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<< Loop-information  End >>>
169    2  pp           do k=1,n3-2
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      u, rhs, unew
                <<< Loop-information  End >>>
170    3  p              do j=1,n2-2
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.71, ITR: 136, MVE: 9, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      u, rhs, unew
                <<< Loop-information  End >>>
171    4  p   v            do i=1,n1-2
172    4  p   v              unew(i,j,k) = &
173    4                       ((u(i+1,j,k) + u(i-1,j,k)) * h1sqinv  &
174    4                       +(u(i,j+1,k) + u(i,j-1,k)) * h2sqinv &
175    4                       +(u(i,j,k+1) + u(i,j,k-1)) * h3sqinv &
176    4                       -rhs(i,j,k)) * hhhinv
177    4  p   v            end do
178    3  p              end do
179    2  p            end do
180    1             end do
181                !OCL END_SCACHE_ISOLATE_ASSIGN
182                !OCL END_SCACHE_ISOLATE_WAY
```

Array u reusability increased



Effect of 1.15 times

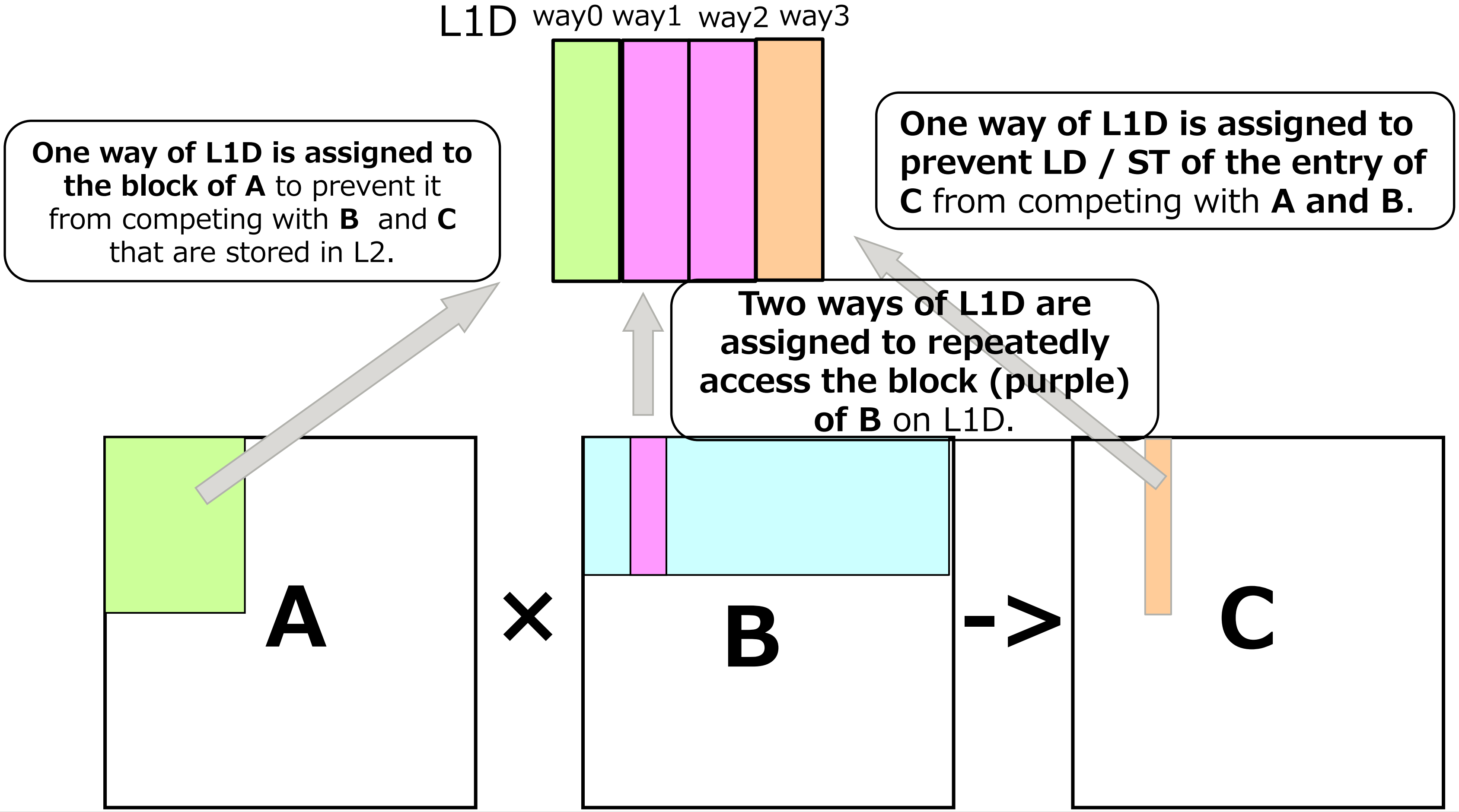
	Memory throughput (GB/s)
Before improvement	215.62
After improvement	205.39

L2 misses reduced

	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
Before improvement	2.46E+08	0.15	2.57%	98.19%	0.00%
After improvement	1.95E+08	0.10	13.43%	87.39%	0.00%

Effects of a Sector Cache on DGEMM (1/2)

- In the matrix product (C = AB), matrices are divided into blocks, stored in the L1D and L2 caches, and calculated.
- The blocks of A and B are copied to the work area to prevent thrashing.
- The sector cache of L1 is used to ensure that A, B, and C are stored in the cache. Particularly, the block of B is important that is repeatedly used on L1D.



One way of L1D is assigned to the block of A to prevent it from competing with **B** and **C** that are stored in L2.

Two ways of L1D are assigned to repeatedly access the block (purple) of B on L1D.

One way of L1D is assigned to prevent LD / ST of the entry of C from competing with **A and B**.

Effects of a Sector Cache on DGEMM (2/2)

- Performance of DGEMM when a sector cache is used and not used
  - 1CMG

	DGEMM efficiency
Sector cache used	94%
Sector cache not used	74%

- The use of a sector cache increases efficiency by 20%.

High-Speed Store (Zfill)

- Image of High-Speed Store (zfill)
- About High-Speed Store (zfill) by the Compiler
- High-Speed Store Instructions
- Operating Conditions of High-Speed Store (zfill)
- Evaluation of High-Speed Store (zfill)
- [Note] About High-Speed Store of On-Cache Data

Image of High-Speed Store (zfill)

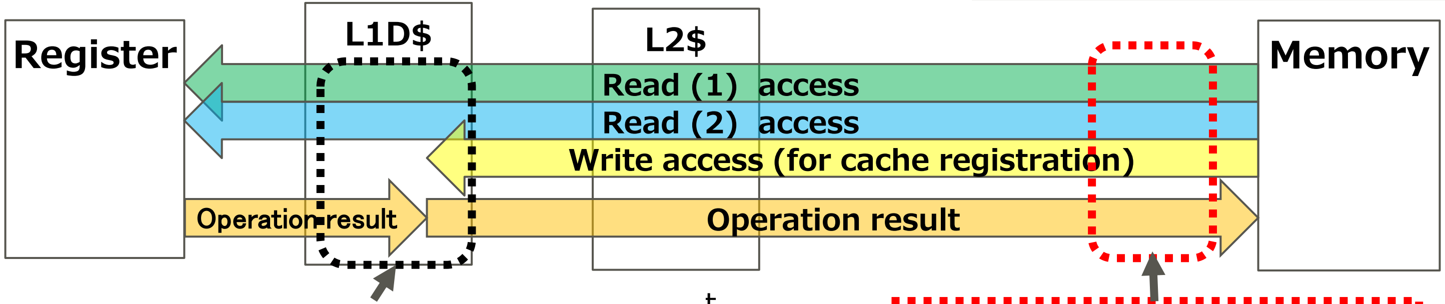
- About high-speed store (zfill)

  This function secures cache lines for writing on the cache. This reduces the cache line read access from the memory and improves the performance of a program whose bottleneck is the memory throughput.

- Conceptual diagram (image of the Stream Triad case)

The A64FX uses the DC ZVA instruction.

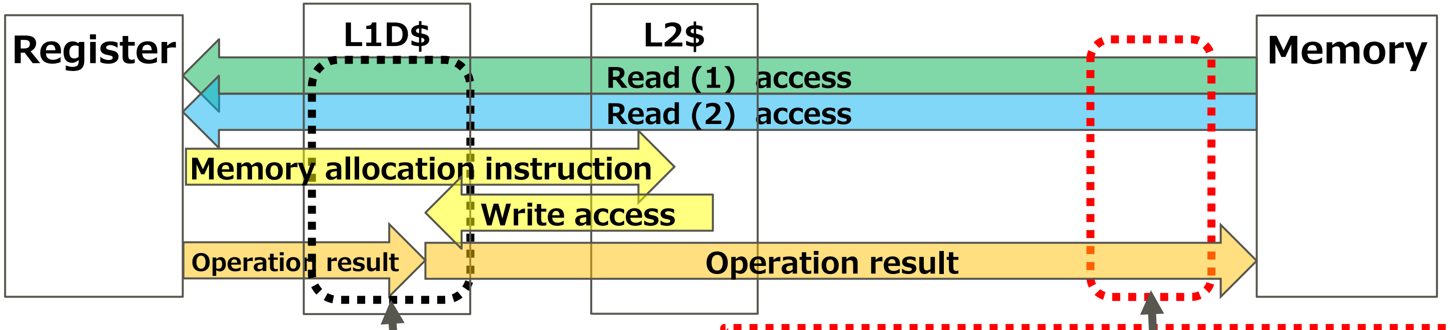
1. When high-speed store is not used



Cache is used for both read and write.

There are four streams between memory and register.

2. When high-speed store is used



Cache is used for both read and write.

The number of streams between memory and register is reduced to three, resulting in improved stream performance.

About High-Speed Store (zfill) by the Compiler

FUJITSU

Option description

Not induced by -Kfast

The option and OCL specifier are renamed.
XFILL -> zfill
* The old option and specifier (-KXFILL and !OCL XFILL) are also compatible.

{ zfill[=*N*] | nozfill } 1 ≦ *N* ≦ 100
For array data that is only written within the loop, the store instruction is accelerated by using an instruction that secures cache lines for writing on the cache without loading data from the memory.
If you specify N, the data of N cache lines ahead is optimized.
The specifiable value range of N is 1 to 100. If you omit N, the compiler automatically decides the value. This option takes effect only when the -O2 and subsequent options are valid. The default is –Knozfill.

Optimization overview

zfill achieves optimization in which data is stored at high speed by using an instruction that secures cache lines for writing on the cache without loading data from the memory. It is applied to array data that is stored in the loop.
Note that optimization does not occur for an array having any reference in the same loop, an array subject to non-sequential access, or an array stored under an IF construct. Also, if zfill is applied, a prefetch instruction is not output to the secondary cache.

Since loop deformation is performed in which all the cache lines secured through optimization by zfill are stored, the following optimization techniques become unavailable. This may result in lower execution performance.
- Loop unrolling
- Loop striping

Even if the zfill optimization occurs, the information resulting from performing the above-mentioned optimization may be output to the compilation message or optimization information when optimization by loop deformation is initiated.
The execution performance may also lower in the following cases.
- Loop with a small number of rotations
- If you specify the number of cache lines in -Kzfill=N, the number of rotations is smaller than the number of entries in the cache lines.
- Program without a memory bandwidth bottleneck

Do not specify -Kzfill if doing so lowers the execution performance. It is desirable to control zfill on a loop-by-loop basis. Therefore, it is recommended to specify the optimization specifier “ZFILL” rather than the option that affects the entire program.

High-Speed Store Instructions

FX100

```
ldd,s   [%xg0+-8],%f34
ldd,s   [%xg1+-8],%f32
subcc   %o1,1,%o1
ldd,s   [%xg0+24],%f40
ldd,s   [%xg1+24],%f38
ldd,s   [%xg0+56],%f44
ldd,s   [%xg1+56],%f42
ldd,s   [%xg0+88],%f48
ldd,s   [%xg1+88],%f46
ldd,s   [%xg0+120],%f52
ldd,s   [%xg1+120],%f50
ldd,s   [%xg0+152],%f56
ldd,s   [%xg1+152],%f54
fmaddd,s        %f36,%f34,%f32,%f34
ldd,s   [%xg0+184],%f60
ldd,s   [%xg1+184],%f58
fmaddd,s        %f36,%f40,%f38,%f40
ldd,s   [%xg0+216],%f64
ldd,s   [%xg1+216],%f62
add     %xg1,256,%xg1
fmaddd,s        %f36,%f44,%f42,%f44
add     %xg0,256,%xg0
fmaddd,s        %f36,%f48,%f46,%f48
fmaddd,s        %f36,%f52,%f50,%f52
fmaddd,s        %f36,%f56,%f54,%f56
fmaddd,s        %f36,%f60,%f58,%f60
fmaddd,s        %f36,%f64,%f62,%f64
stxa    %g0, [%xg2 + %xg6] 0xf4
std,sd  %f34,[%xg2+-8]
std,sd  %f40,[%xg2+24]
std,sd  %f44,[%xg2+56]
std,sd  %f48,[%xg2+88]
std,sd  %f52,[%xg2+120]
prefetch        [%xg2+760],2
std,sd  %f56,[%xg2+152]
std,sd  %f60,[%xg2+184]
std,sd  %f64,[%xg2+216]
add     %xg2,256,%xg2
bne,pt  %icc, .L91
```

* The sxar instruction is omitted.

High-speed store instruction for the FX100

A64FX

```
ld1d    {z1.d}, p0/z, [x4, -3, mul vl]
ld1d    {z0.d}, p0/z, [x9, -3, mul vl]
dc      ZVA, x7
add     x7, x7, 256
ld1d    {z4.d}, p0/z, [x4, -2, mul vl]
ld1d    {z3.d}, p0/z, [x9, -2, mul vl]
subs    w3, w3, 1
ld1d    {z6.d}, p0/z, [x4, -1, mul vl]
ld1d    {z5.d}, p0/z, [x9, -1, mul vl]
ld1d    {z16.d}, p0/z, [x4, 0, mul vl]
ld1d    {z7.d}, p0/z, [x9, 0, mul vl]
add     x9, x9, 256
add     x4, x4, 256
fmad    z1.d, p0/m, z2.d, z0.d
fmad    z4.d, p0/m, z2.d, z3.d
fmad    z6.d, p0/m, z2.d, z5.d
fmad    z16.d, p0/m, z2.d, z7.d
st1d    {z1.d}, p0, [x1, -3, mul vl]
st1d    {z4.d}, p0, [x1, -2, mul vl]
st1d    {z6.d}, p0, [x1, -1, mul vl]
st1d    {z16.d}, p0, [x1, 0, mul vl]
prfm    16, [x1, 824]
add     x1, x1, 256
bne     .L92
```

High-speed store instruction for the A64FX

Since high-speed store disables hardware prefetch, software prefetch is used as the L1 prefetch.

Since an amount of data equivalent to cache lines (256 B) needs to be stored when high-speed store (zfill) is used, the memory space for the necessary number of rotations is allocated through striping.

* Amount of data equivalent to cache lines (256 B): 4SIMD × 8 rotations for FX100 and 8SIMD × 4 rotations for A64FX

Operating Conditions of High-Speed Store (zfill) (1/2)

- Operating conditions
  - The arrays to be stored must not be dependent between iterations.
  - There must be no reference to any defined array.
  - Memory access must be contiguous.
- Case in which zfill operates

zfill operates because memory access is contiguous.

Triad

```
  real*4 y(n), x1(n), x2(n), c0
  do j = 1, iter
!$omp parallel do
    Do i = 1, n
      y(i)=x1(i) + c0 * x2(i)
    End Do
  enddo
```

iter = 3
n = 3145728

Stream-like operation kernel pattern 1

```
do l = 1, lall
  !$OMP PARALLEL DO
  do k = 1, kall
  do g = 1, gall
    rho(g,k,l) = PROG(g,k,l,1) / metrics(g,k,l)
    vx (g,k,l) = PROG(g,k,l,2) / PROG(g,k,l,1)
    vy (g,k,l) = PROG(g,k,l,3) / PROG(g,k,l,1)
    vz (g,k,l) = PROG(g,k,l,4) / PROG(g,k,l,1)
    ein(g,k,l) = PROG(g,k,l,5) / PROG(g,k,l,1)
  enddo
  enddo
  enddo
```

```
do iq = 1, qall
  do l  = 1, lall
  !$OMP PARALLEL DO
  do k  = 1, kall
  do g  = 1, gall
    q(g,k,l,iq) = PROGq(g,k,l,iq) / PROG(g,k,l,1)
  enddo
  enddo
  enddo
enddo
```

gall=16900, kall=96
lall=1, qall=6

Operating Conditions of High-Speed Store (zfill) (2/2)

Case in which zfill operates

Stream-like operation kernel pattern 2

```
do l = 1, lall
  !$OMP PARALLEL DO
  do k = 1, kall
  do g = 1, gall
    tendency(g,k,l,1) = tendency_0(g,k,l,1) + tendency_11(g,k,l) + tendency_21(g,k,l)
    tendency(g,k,l,2) = tendency_0(g,k,l,2) + tendency_12(g,k,l) + tendency_22(g,k,l)
    tendency(g,k,l,3) = tendency_0(g,k,l,3) + tendency_13(g,k,l) + tendency_23(g,k,l)
    tendency(g,k,l,4) = tendency_0(g,k,l,4) + tendency_14(g,k,l) + tendency_24(g,k,l)
    tendency(g,k,l,5) = tendency_0(g,k,l,5) + tendency_15(g,k,l) + tendency_25(g,k,l)
    tendency(g,k,l,6) = tendency_0(g,k,l,6) + tendency_16(g,k,l) + tendency_26(g,k,l)
  enddo
  enddo
  enddo
```

In this code, zfill can operate. However, the current compiler cannot fully determine whether there is no dependence of the arrays to be stored. There is some room for improvement.

(By dividing the loop, the current compiler can also run zfill. This time, therefore, the evaluation is performed by dividing the loop.)

Case in which zfill does not operate

Stream-like operation kernel pattern 3

```
do l = 1, lall
  !$OMP PARALLEL DO
  do k = 1, kall
  do g = 1, gall
    value(g,k,l,1) = value(g,k,l,1) + tendency(g,k,l,1) * fraction
    value(g,k,l,2) = value(g,k,l,2) + tendency(g,k,l,2) * fraction
    value(g,k,l,3) = value(g,k,l,3) + tendency(g,k,l,3) * fraction
    value(g,k,l,4) = value(g,k,l,4) + tendency(g,k,l,4) * fraction
    value(g,k,l,5) = value(g,k,l,5) + tendency(g,k,l,5) * fraction
    value(g,k,l,6) = value(g,k,l,6) + tendency(g,k,l,6) * fraction
  enddo
  enddo
  enddo
```

Since there is reference to a defined array, zfill does not operate.

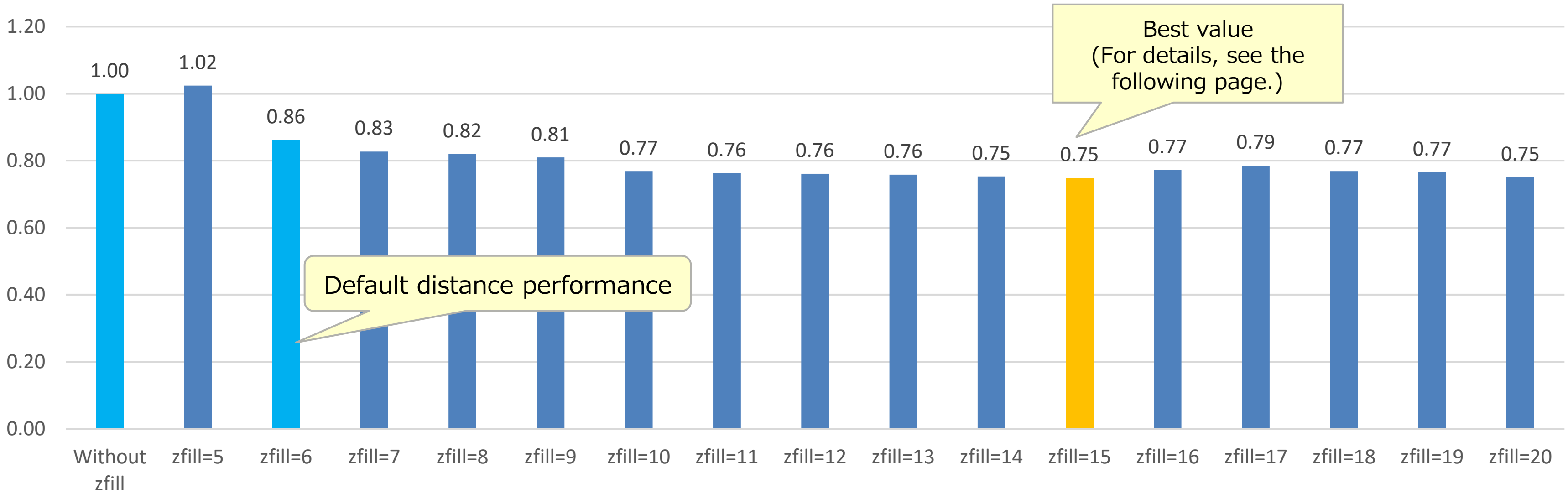
zfill Evaluation: Triad (1/2)

- Like prefetch, high-speed store (zfill) requires that the time for hiding the latency be determined and that instructions be executed in advance.
  The high-speed store (zfill) distance was verified in the case of Triad.
  -> Improvement in performance was observed on 15 cache lines.
  (The following page shows the details.)

STRRAM Triad/1CMG execution

```
!$omp parallel
  Do j = 1, iter   iter=3000
!$omp do
     Do i = 1, n          n=1048512
        y(i)=x1(i) + c0 * x2(i)
     End Do
!$omp end do nowait
  End Do
!$omp end parallel
```

Elapsed time ratio (when the time observed without zfill is 1)

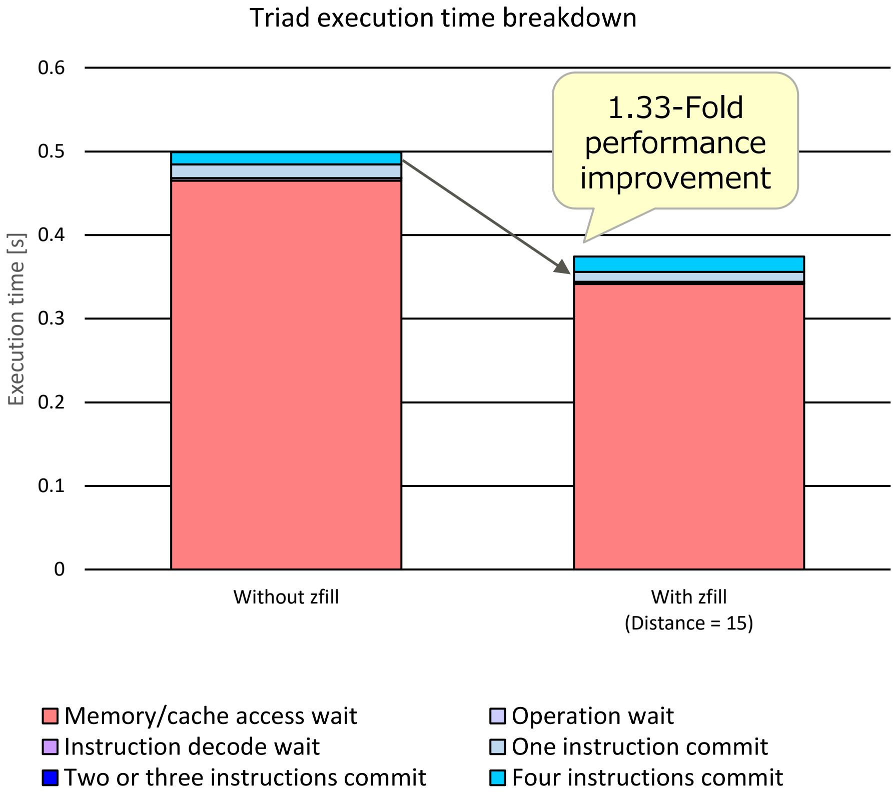


Best value
(For details, see the following page.)

Default distance performance

zfill Evaluation: Triad (2/2)

- Performance improvement by zfill



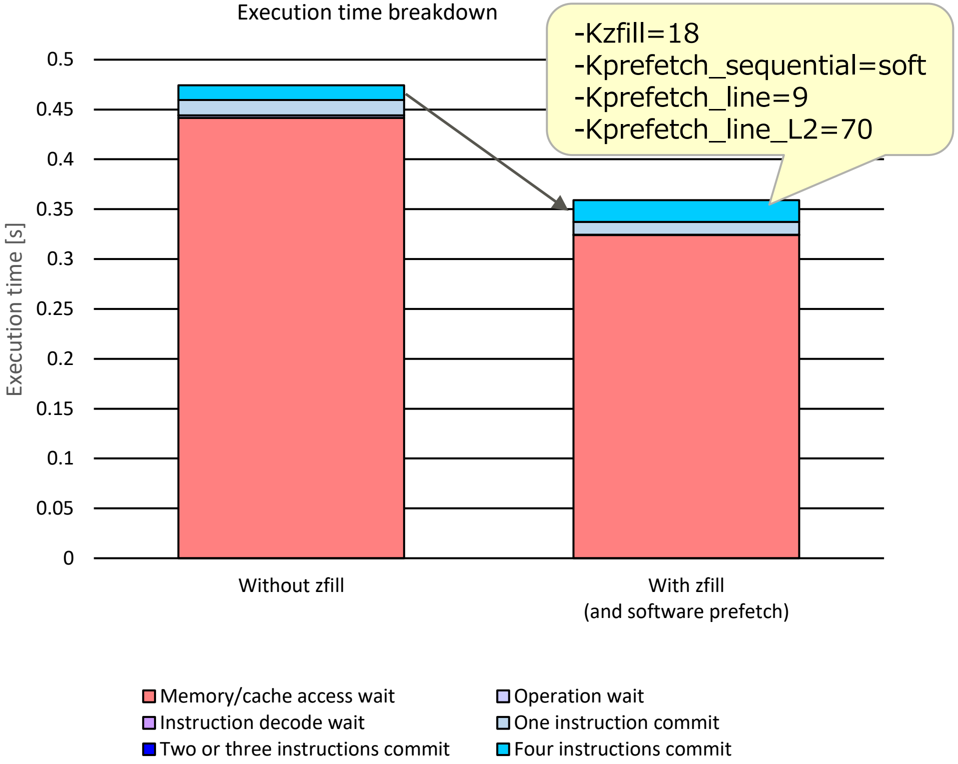
The effect of high-speed store (zfill) reduces the number of memory accesses and improves performance.

PA values

	Without zfill	With zfill (Distance = 15)
Aggregation thread number	0	0
Execution time [s]	0.499	0.374
Total number of effective instructions	2.16E+09	2.18E+09
GFLOPS	12.60	16.83
Memory throughput [GB/s]	206.28	207.88
L1 busy rate/thread	59.97%	56.39%
L2 busy rate	87.65%	93.35%
Memory busy rate	80.58%	81.21%
Floating-point pipeline busy rate/thread	FLA:4.61% FLB:2.70%	FLA:6.44% FLB:3.32%
L1 miss count/thread	2.47E+07	2.47E+07
L1 miss demand rate/thread	3.65%	2.18%
L2 miss count/thread	2.55E+07	1.72E+07
L2 miss demand rate/thread	5.83%	5.27%

zfill Evaluation: Triad [Reference: Number of Innermost Loop Iterations Increased by 10 Times]

Performance of zfill and software prefetch



-Kzfill=18
-Kprefetch_sequential=soft
-Kprefetch_line=9
-Kprefetch_line_L2=70

PA values

	Without zfill	With zfill (and software prefetch)
Aggregation thread number	0	0
Execution time [s]	0.474	0.359
Total number of effective instructions	2.16E+09	2.52E+09
GFLOPS	13.27	17.54
Memory throughput [GB/s]	**222.5**	210.8
L1 busy rate/thread	52.9%	55.09%
L2 busy rate	85.1%	94.39%
Memory busy rate	86.8%	82.32%
Floating-point pipeline busy rate/thread	FLA:4.90% FLB:2.80%	FLA:7.14% FLB:3.01%
L1 miss count/thread	2.46E+07	2.46E+07
L1 miss demand rate/thread	9.40%	0.08%
L2 miss count/thread	2.62E+07	1.64E+07
L2 miss demand rate/thread	12.21%	0.31%

* The evaluation was conducted with the access size (number of innermost loop iterations, array size) increased to 240 MB (by 10 times).

Performing software prefetch and zfill properly improves the performance up to 210.8 GB/s.

zfill Evaluation: Stream-Like Operation Kernel Pattern 1 (1/2)

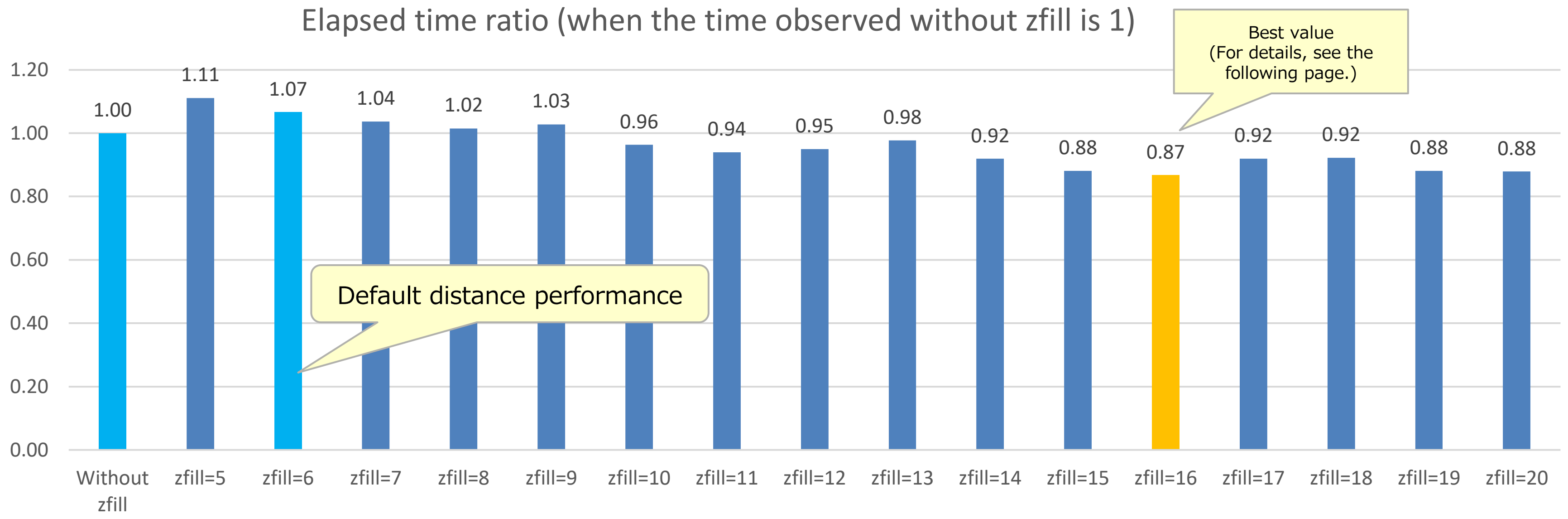
- The high-speed store (zfill) distance was verified in the case of the stream-like operation kernel (pattern 1).
  -> Improvement in performance was observed on 16 cache lines. (The following page shows the details.)

Stream-like operation kernel pattern 1

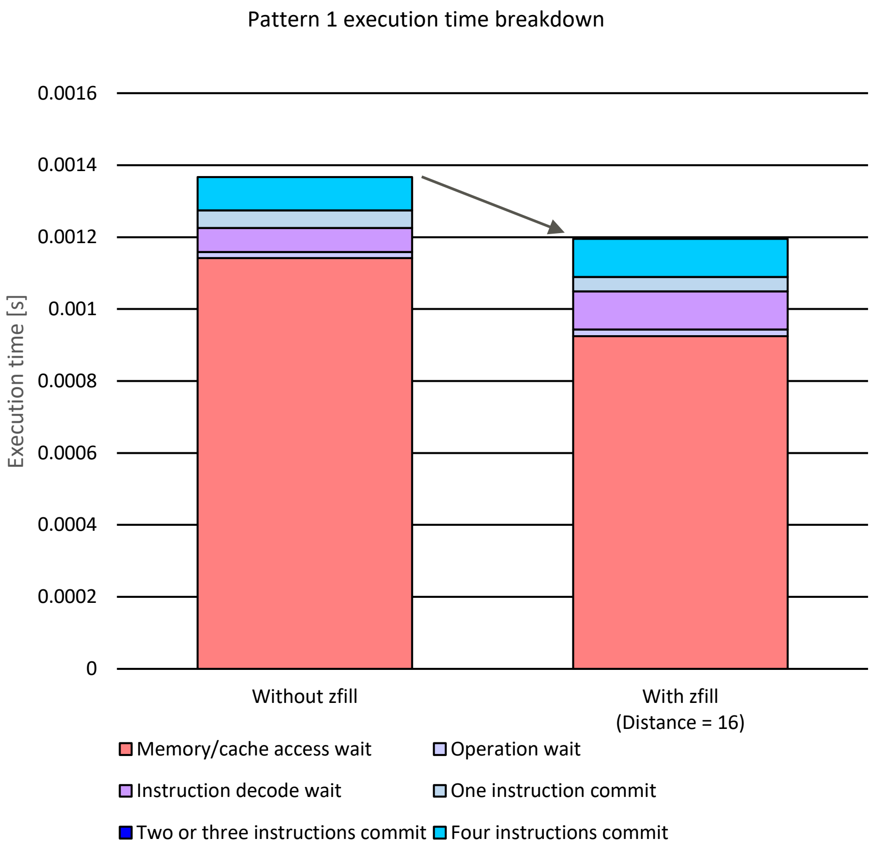
```
do l = 1, lall
  !$OMP PARALLEL DO
  do k = 1, kall
  do g = 1, gall
    rho(g,k,l) = PROG(g,k,l,1) / metrics(g,k,l)
    vx (g,k,l) = PROG(g,k,l,2) / PROG(g,k,l,1)
    vy (g,k,l) = PROG(g,k,l,3) / PROG(g,k,l,1)
    vz (g,k,l) = PROG(g,k,l,4) / PROG(g,k,l,1)
    ein(g,k,l) = PROG(g,k,l,5) / PROG(g,k,l,1)
  enddo
  enddo
  enddo
```

```
do iq = 1, qall
  do l  = 1, lall
  !$OMP PARALLEL DO
  do k  = 1, kall
  do g  = 1, gall
    q(g,k,l,iq) = PROGq(g,k,l,iq) / PROG(g,k,l,1)
  enddo
  enddo
  enddo
  enddo
```

Single precision evaluation
gall=16900,
kall=96
lall=1, qall=6



zfill Evaluation: Stream-Like Operation Kernel Pattern 1 (2/2)



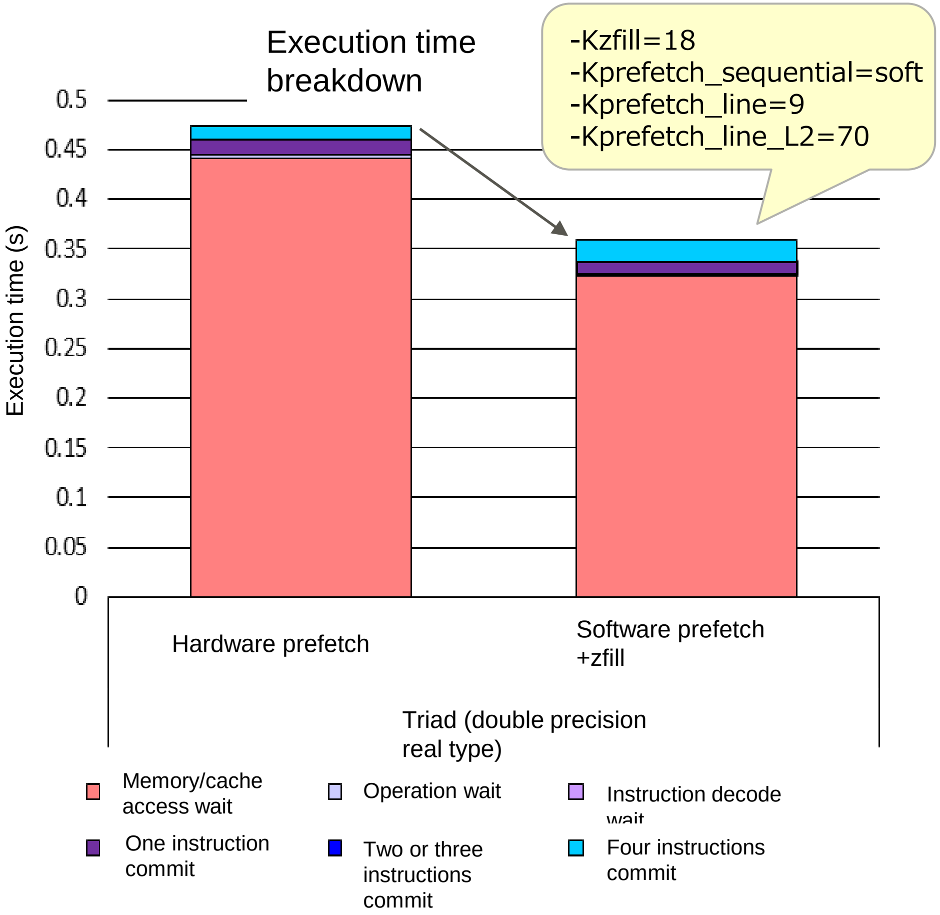
The effect of zfill has been confirmed. Note that the default value of the zfill destination needs to be changed.

PA values

	Without zfill	With zfill (Distance = 16)
Aggregation thread number	0	0
Execution time [s]	0.001372	0.001206
Total number of effective instructions	10775582	11158332
GFLOPS	91.10	103.7
Memory throughput [GB/s]	211.31	194.84
L1 busy rate	47.98%	46.77%
L2 busy rate	83.70%	83.85%
Memory busy rate	83.36%	76.91%
Floating-point pipeline busy rate/thread	FLA:11.64% FLB:10.92%	FLA:12.92% FLB:12.59%
L1 miss count/thread	7.15E+04	7.21E+04
L1 miss demand rate/thread	10.93%	5.37%
L2 miss count/thread	6.96E+04	5.49E+04
L2 miss demand rate/thread	15.31%	13.25%

[Reference] Performance Improvement by zfill

- Performance of software prefetch and zfill



PA values

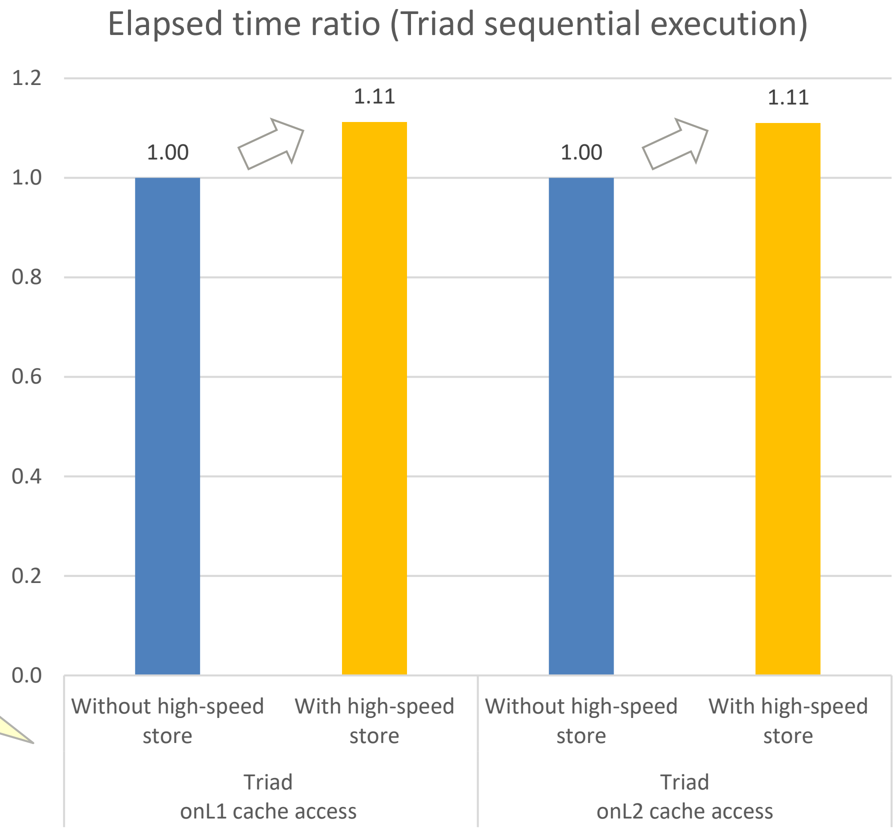
	Hardware prefetch	Software prefetch + zfill
Aggregation thread number	0	0
Execution time [s]	0.474	0.359
Total number of effective instructions	2.16E+09	2.52E+09
GFLOPS	13.27	17.54
Memory throughput [GB/s]	**222.5**	210.8
L1 busy rate/thread	52.9%	55.09%
L2 busy rate	85.1%	94.39%
Memory busy rate	21.7%	20.58%
Floating-point pipeline busy rate/thread	FLA:4.90% FLB:2.80%	FLA:7.14% FLB:3.01%
L1 miss count/thread	2.46E+07	2.46E+07
L1 miss demand rate/thread	9.40%	0.08%
L2 miss count/thread	**2.62E+07**	1.64E+07
L2 miss demand rate/thread	**12.21%**	0.31%

* The evaluation was conducted with the access size (number of innermost loop iterations, array size) increased to 240 MB (by 10 times).

Performing zfill properly improves the performance up to 210.8 GB/s.

[Note] About High-Speed Store of On-Cache Data

- High-speed store during cache access
  Performing high-speed store for on-cache data may lower the performance.
  Before specifying high-speed store, you need to check whether it is actually necessary.



Performance decreased by about 11% when high-speed store was applied to data access to the L1 and L2 caches.

High-speed store (zfill) improves performance when used after access to the target arrays is properly checked.

Data Access Alignment Constraints

- Microarchitecture Affected by Alignment
- Gather Load Instruction Aggregation Function
- High-Speed Operation of the Multiple Structures Instruction
- WB (Write Buffer) Operations

For the performance related to the following alignment changes, see *Basic Kernel Performance*.

- Evaluation Results: Contiguous SIMD Load
- Evaluation Results: Contiguous SIMD Store
- Evaluation Results: Gather Load
- Evaluation Results: Scatter Store
- Evaluation Results: Structure Load (LD2 Instruction)
- Evaluation Results: Structure Store (ST2 Instruction)

Microarchitecture Affected by Alignment

- Alignment changes may cause differences in performance for some of the load / store instructions. Alignment is considered to affect the following.
  - Gather load instruction aggregation function
  - High-speed operation of the Multiple Structures instruction
  - WB (write buffer) operations (WB split and merge)
  - (Store fetch bypass operation)

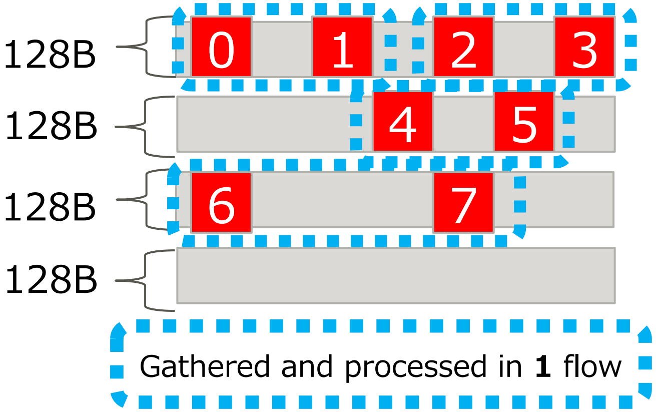
Gather Load Instruction Aggregation Function

What is the gathering function of the Gather instruction?

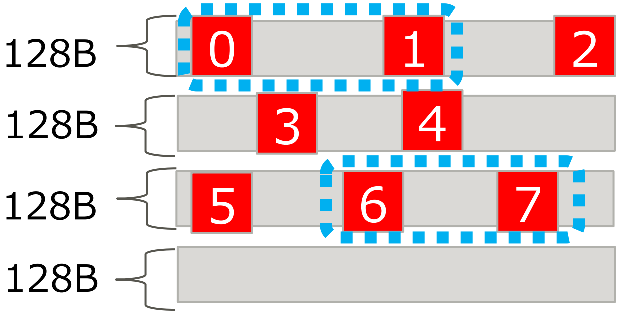
If two elements issued simultaneously **from one FP are adjacent to each other, the** Gather instruction can process them at a high speed. If the addresses of the two adjacent elements match each other within 128 bytes, the instruction can speed up processing by gathering the elements and processing **them** in one flow.

- If two adjacent elements belong to the same 128-byte block, they are gathered and processed in one flow. In the following address pattern examples, ⬚ indicates the gathered parts, and the two elements are processed in one L1D$ pipeline flow.

◆ Address pattern example 1



◆ Address pattern example 2



Pay attention when implementing the starting addresses of arrays to make full use of the gathering function of the Gather instruction.

High-Speed Operation of the Multiple Structures Instruction

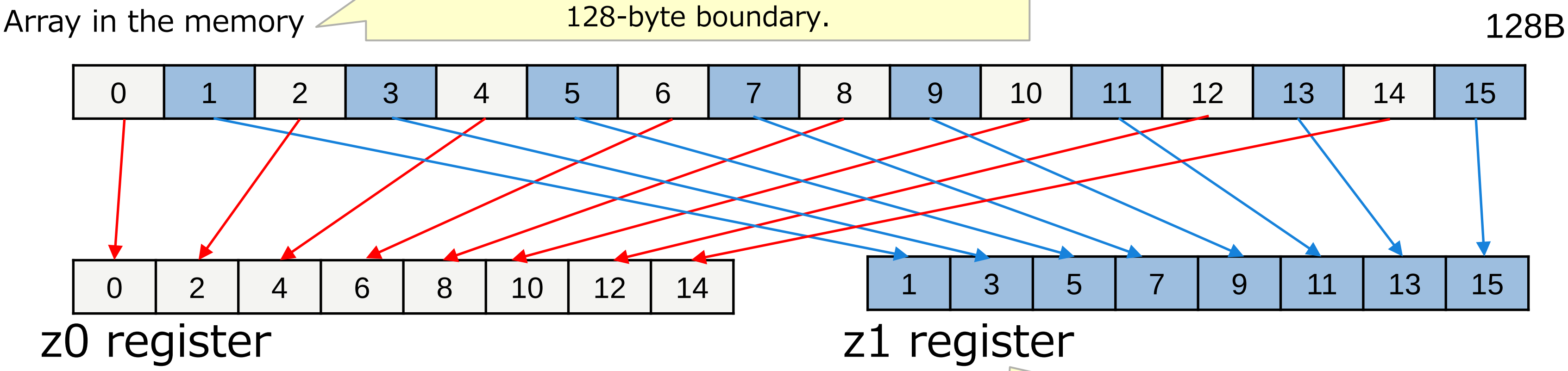
High-speed operation of the Multiple Structures instruction

The Multiple Structures instruction (LD2/ST2) can process data in one flow per register when the address range of the data to be handled fits within a 128-byte boundary, leading to faster processing.

Example) LD2 instruction

```
ld2d  {z0.d, z1.d}, p0/z, [(array address)]
```

The performance changes depending on whether the array in the memory fits within a 128-byte boundary.



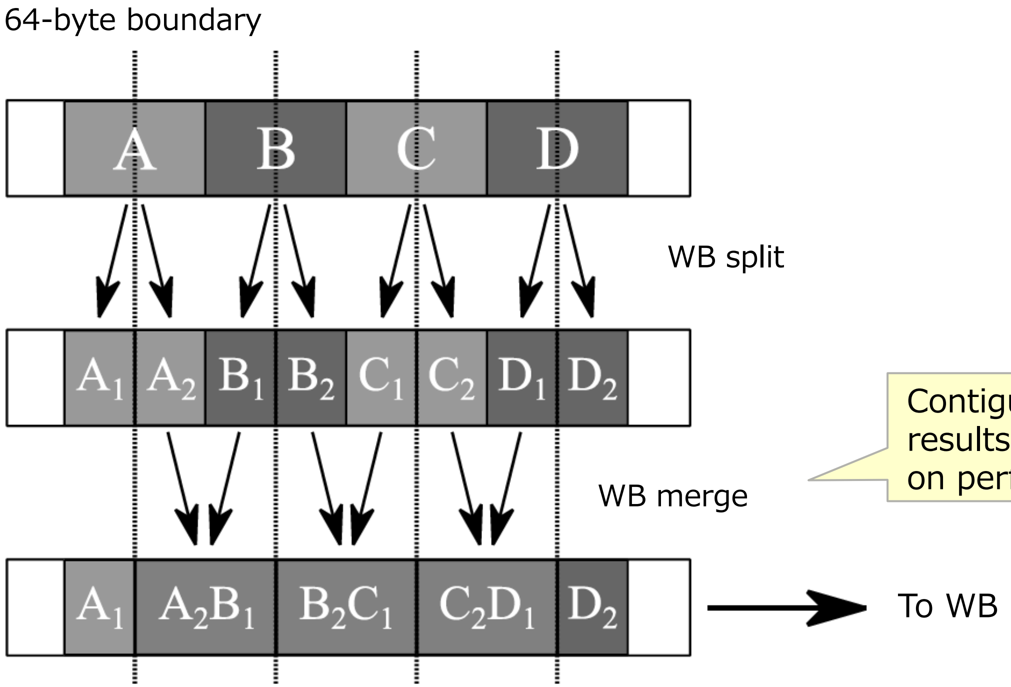
When 0, 2, 4, 6, 8, 10, 12, and 14 fit within a 128-byte boundary, the data can be processed in one flow.
(Otherwise, the data is processed in two flows.)

When 1, 3, 5, 7, 9, 11, 13, and 15 fit within a 128-byte boundary, the data can be processed in one flow.
(Otherwise, the data is processed in two flows.)

WB (Write Buffer) Operations

- WB (write buffer) operations

Data is written from the SP (store port) to the WB (write buffer) in 64-byte units. If blocks of data to be stored each span a 64-byte boundary, storing all the data requires more write operations than the number of data blocks. If blocks of data to be stored each span a 64-byte boundary, each block needs to be split at the 64-byte boundary when written from the SP to the WB (WB split). After that, the split data blocks that fit within the same 64-byte boundary are merged into one block (WB merge) before being written to the WB.



64-byte boundary
A
B
C
D
WB split
A_1 A_2 B_1 B_2 C_1 C_2 D_1 D_2
WB merge
A_1 A_2B_1 B_2C_1 C_2D_1 D_2
To WB

Contiguous access results in little impact on performance.

Verification of Out-of-Order (OoO) Execution

- Out-of-Order Execution-Related Resources
- Purposes of the Verification
- Verification of the Out-of-Order Execution Effect
- Collaboration Between Out-of-Order Execution and Software Scheduling
- Summary

Out-of-Order Execution-Related Resources

- Out-of-order execution-related resources

Item	Unit	A64FX	K computer	FX100
Operation latency FL (FMA)	(Cycle)	9	6	6
Number of inflight instructions (CSE)	(Instruction/core)	128	48	64
Number of inflight load instructions (FP)	(Instruction/core)	40	20	32
Number of inflight store instructions (SP)	(Instruction/core)	24	8	20
RSA (address calculation and integer operation)	(Entry/core)	20	10	16
RSE (integer operation and real number operation)	(Entry/core)	20x2	10+16	16+20
RSBR (branch)	(Entry/core)	19	8	10
Number of renaming registers (GPR)	(Entry/core)	64	32	48
Number of renaming registers (FPR)	(Entry/core)	96	48	64
Number of renaming registers (PDR)	(Entry/core)	32	-	-

Purposes of the Verification

- Verification of the out-of-order execution effect
  - Check whether out-of-order execution is effective in performance improvement, using an actual machine
  - Check changes in the effect of out-of-order execution that occur as the number of operations (chains) changes
  - Check quantitative changes in performance based on the number of out-of-order resources of the A64FX and the number of resources necessary for optimal scheduling

- Collaboration between out-of-order execution and software scheduling
  - Check the effect of scheduling collaboration between software (compiler) and hardware (out-of-order execution)

Verification of the Out-of-Order Execution Effect (1/4)

Evaluation overview

Perform an FMA operation for array y, using constants c0 to c9. Prepare nine cases, each with a different number of FMAs from one to nine.

The expression assumes access to array y. In this code, since m is passed as an argument, the software (compiler) cannot determine m and the loops of i are executed sequentially. Compare the following cases.

(1) OoO enabled/software scheduling disabled
m=0: There is no dependency, and the processing can be overtaken. (-Kfast,noswp,unroll=2)

(2) OoO disabled/software scheduling disabled
m=1: Dependencies emerge, and the loops of i must be executed sequentially. (-Kfast,noswp,unroll=2)

(3) SWPL (OoO enabled/software scheduling enabled)
m=0: Specify NORECURRENCE. (-Kfast)

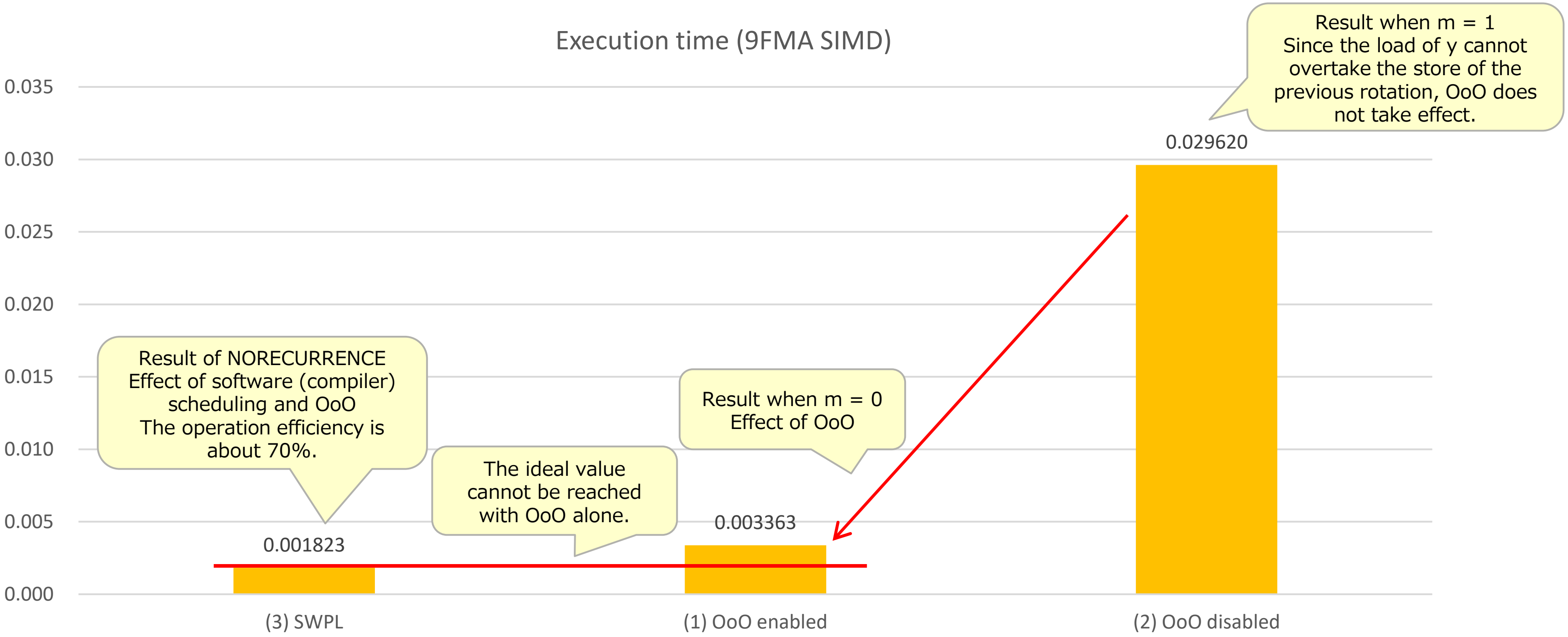
```
[1FMA]
  Do j = 1, n
    Do i = 1, 8
      y(i,j) = c0 + y(i,j-m) * c1
    End Do
  End Do

[2FMA]
  Do j = 1, n
    Do i = 1, 8
      y(i,j) = c0 + y(i,j-m)*(c1 + y(i,j-m) * c2 )
    End Do
  End Do

 : (Omitted)

[9FMA]
  Do j = 1, n
    Do i = 1, 8
      y(i,j) = c0 + y(i,j-m)*(c1 + y(i,j-m)* ¥
               (c2 + y(i,j-m)*(c3 + y(i,j-m)* ¥
               (c4 + y(i,j-m)*(c5 + y(i,j-m)* ¥
               (c6 + y(i,j-m)*(c7 + y(i,j-m)* ¥
               (c8 + y(i,j-m)* c9))))))))
    End Do
  End Do
```

Verification of the Out-of-Order Execution Effect (2/4)

Execution results (9FMA)

While OoO is effective, SWPL is superior to OoO in performance because there are many operations and chains.

```
Do j = 1, n
  Do i = 1, 8
    y(i,j) = c0 + y(i,j-m)*(c1 + y(i,j-m)* ¥
             (c2 + y(i,j-m)*(c3 + y(i,j-m)* ¥
             (c4 + y(i,j-m)*(c5 + y(i,j-m)* ¥
             (c6 + y(i,j-m)*(c7 + y(i,j-m)* ¥
             (c8 + y(i,j-m)* c9))))))))
  End Do
End Do
```



Result when m = 1
Since the load of y cannot overtake the store of the previous rotation, OoO does not take effect.

Result of NORECURRENCE
Effect of software (compiler) scheduling and OoO
The operation efficiency is about 70%.

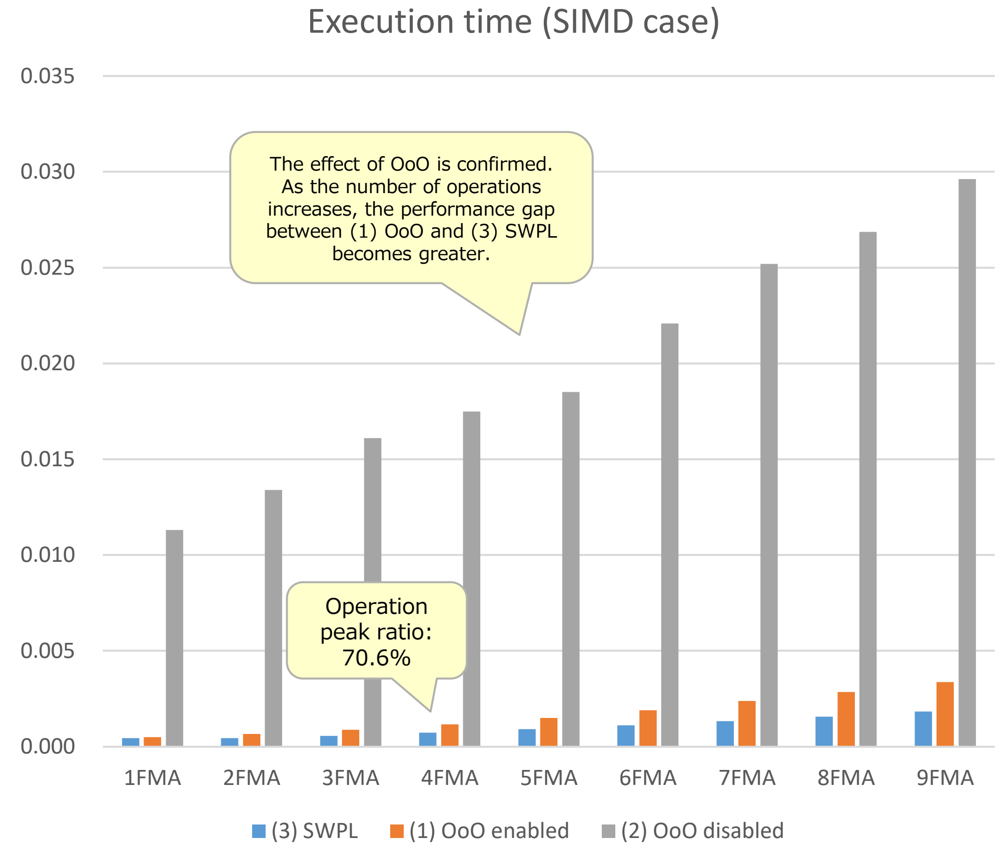
The ideal value cannot be reached with OoO alone.

Result when m = 0
Effect of OoO

Verification of the Out-of-Order Execution Effect (3/4)

- Execution results of the individual cases (execution time)

For the loop of the 2FMA case, the performance of SWPL is almost the same as that of OoO.



The effect of OoO is confirmed. As the number of operations increases, the performance gap between (1) OoO and (3) SWPL becomes greater.

Operation peak ratio: 70.6%

```
[1FMA]
  Do j = 1, n
    Do i = 1, 8
      y(i,j) = c0 + y(i,j-m) * c1
    End Do
  End Do

[2FMA]
  Do j = 1, n
    Do i = 1, 8
      y(i,j) = c0 + y(i,j-m)*(c1 + y(i,j-m) * c2 )
    End Do
  End Do

 : (Omitted)

[9FMA]
  Do j = 1, n
    Do i = 1, 8
      y(i,j) = c0 + y(i,j-m)*(c1 + y(i,j-m)* ¥
                (c2 + y(i,j-m)*(c3 + y(i,j-m)* ¥
                (c4 + y(i,j-m)*(c5 + y(i,j-m)* ¥
                (c6 + y(i,j-m)*(c7 + y(i,j-m)* ¥
                (c8 + y(i,j-m)* c9))))))))
    End Do
  End Do
```

Verification of the Out-of-Order Execution Effect (4/4)

- Number of OoO resources necessary for ideal execution (for hiding operation latency)

	Number of A64FX resources	2FMA	3FMA	9FMA
Number of necessary registers	32+96	39	40	46
Number of RSEs	20x2	36	54	162
Number of RSBRs	16	9	9	9
Number of necessary FPs	40	18	18	18
Number of necessary SPs	24	18	18	18

As the gap from the number of resources increases, adequate scheduling becomes more difficult.

Number of necessary registers = Constant + (2 * latency * number of pipelines)
Number of necessary RSEs = Number of FMAs * latency * number of pipelines
Number of necessary RSBRs = Latency * number of pipelines / number of unrolls
Number of necessary FPs = Number of loads * latency * number of pipelines
Number of necessary SPs = Number of stores * latency * number of pipelines

```
[2FMA]
  Do j = 1, n
    Do i = 1, 8
      y(i,j) = c0 + y(i,j-m)*(c1 + y(i,j-m) * c2 )
    End Do
  End Do

[3FMA]
  Do j = 1, n
    Do i = 1, 8
      y(i,j) = c0 + y(i,j-m)*(c1 + y(i,j-m) * ¥
                 (c2 + y(i,j-m)* c3 ))
    End Do
  End Do

 : (Omitted)

[9FMA]
  Do j = 1, n
    Do i = 1, 8
      y(i,j) = c0 + y(i,j-m)*(c1 + y(i,j-m)* ¥
                 (c2 + y(i,j-m)*(c3 + y(i,j-m)* ¥
                 (c4 + y(i,j-m)*(c5 + y(i,j-m)* ¥
                 (c6 + y(i,j-m)*(c7 + y(i,j-m)* ¥
                 (c8 + y(i,j-m)* c9))))))))
    End Do
  End Do
```

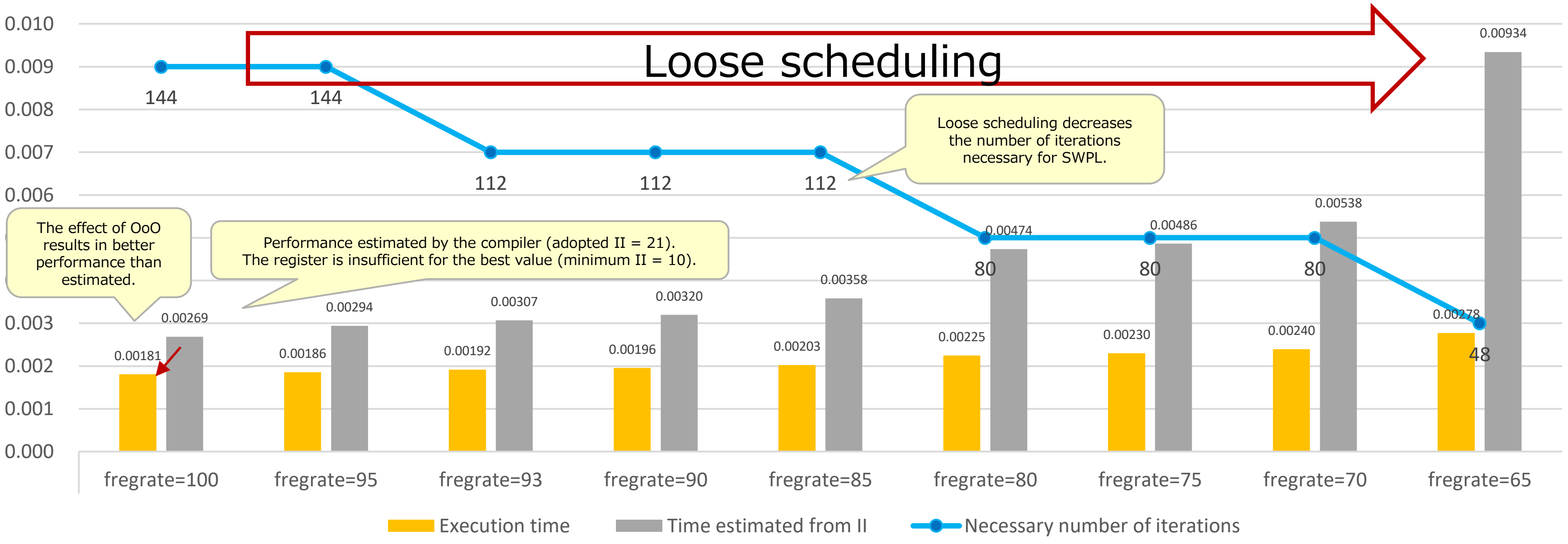
Collaboration Between Out-of-Order Execution and Software Scheduling

- **Out-of-order execution and software scheduling (SWPL)**
  The performance of SWPL improves when used in collaboration with OoO.

- **Loose scheduling and out-of-order execution**
  While loose scheduling through SWPL results in a poorly-ordered instruction sequence, OoO prevents the deterioration width from increasing.
  As scheduling loosens, the number of loop iterations necessary for SWPL decreases, making it possible to use SWPL for the tuning of loops whose iteration counts are known.

```
Do j = 1, n
  Do i = 1, 8
    y(i,j) = c0 + y(i,j)*(c1 + y(i,j)* ¥
             (c2 + y(i,j)*(c3 + y(i,j)* ¥
             (c4 + y(i,j)*(c5 + y(i,j)* ¥
             (c6 + y(i,j)*(c7 + y(i,j)* ¥
             (c8 + y(i,j)* c9))))))))
  End Do
End Do
```

Execution time/necessary number of SWPL iterations (9fma SIMD)



The effect of OoO results in better performance than estimated.

Performance estimated by the compiler (adopted II = 21).
The register is insufficient for the best value (minimum II = 10).

Loose scheduling decreases the number of iterations necessary for SWPL.

Summary

- The effect of out-of-order execution has been confirmed.

- As the link (number) of operations grows larger, software scheduling (SWPL) becomes more important.
  -> When SWPL is not supported for a loop, dividing the loop promotes SWPL.

- Scheduling SWPL loosely on purpose may provide room for tuning for a loop whose iteration count is explicitly decided.

SIMD Width

- About the SIMD Width
- Performance of the Fixed-Length SIMD Width Specification
- Performance Impact of SIMD Width-Conscious Optimization (Tuning)
- Performance of the Variable-Length SIMD Specification
- [Reference] Changes in Power Based on the SIMD Width

About the SIMD Width

- SIMD widths supported by the processor

  The implementor can freely decide the vector length (SIMD width) of the ARM SVE in units of 128 bits from 128 bits to 2048 bits.
  The A64FX is implemented with the vector length being 512 bits.

  - The A64FX supports the following vector lengths.
    - 512 bits
    - 256 bits
    - 128 bits

- Compiler options
  - -Ksimd_reg_size={ 128 | 256 | 512 | agnostic }
    The default is -Ksimd_reg_size=512.
    - simd_reg_size={ 128 | 256 | 512 }
      Specify the vector register size of the SVE. The unit is the bit. At the time of compilation, optimization is performed, assuming that the value specified in this option is the vector register size of the SVE. Note that the generated executable program works normally only in a CPU architecture in which the SVE vector register whose size is equal to that specified in this option is implemented.
    - simd_reg_size=agnostic
      Compilation is performed, without assuming any particular size as the vector register size of the SVE, to create an executable program that decides the SVE vector register size at the time of execution. This executable program can be executed, regardless of the vector register size of the SVE implemented in the CPU architecture.
      Note that the execution performance may become lower than when the -Ksimd_reg_size={128|256|512} option is specified.

Performance of the Fixed-Length SIMD Width Specification

- Changes in performance based on the SIMD width

The change in performance is acceptable as long as the ratio does not exceed twice the original value when the SIMD width is 256 bits or four times when the SIMD width is 128 bits.
If the ratio is larger than that, it is possible that there is another problem.

Program	Floating-point operation peak ratio	Memory throughput peak ratio	Execution time ratio		
			SIMD width = 512 (Default)	SIMD width = 256	SIMD width = 128
Adventure.region1	26.98%	62.38%	1.00	1.67	3.25
Adventure.region2	20.85%	0.02%	1.00	1.49	2.74
FFB.callap_kernel2	39.12%	76.33%	1.00	1.02	1.19
FFB.spmmv_vec8	28.38%	33.94%	1.00	0.99	5.11
GAMERA.TIMER_COMP_MATVEC_IF	20.29%	8.90%	1.00	1.56	2.81
GENESIS.Nonb15F(June)	7.73%	6.41%	1.00	1.78	3.38
GENESIS.Nonb15F(July)	8.04%	4.46%	1.00	1.45	2.07
GENESIS.PairList(June)	6.71%	7.96%	1.00	1.79	1.88
GENESIS.PairList(July)	4.76%	3.04%	1.00	1.08	1.14
NICAM.Horizontal_Adv_flux	2.59%	1.94%	1.00	1.20	1.96
NICAM.Horizontal_Adv_limiter	14.37%	41.61%	1.00	1.21	1.67
NICAM.Radiation_adding	14.48%	47.57%	1.00	1.27	1.84
NICAM.Radiation_dtrn31	9.64%	36.38%	1.00	1.24	1.81
NICAM.Radiation_ptfit2	11.84%	64.94%	1.00	1.05	1.35
NICAM.Vertical_Adv_limiter	6.10%	73.82%	1.00	0.97	1.01
NICAM.diffusion	15.62%	43.44%	1.00	1.78	3.35
NICAM.divdamp	33.16%	36.00%	1.00	1.56	2.81
NICAM.vi_rhow_solver	15.49%	56.23%	1.00	1.34	2.00
NICAM_nsw6.M3_mp_nsw6_OMP9	10.29%	68.25%	1.00	1.01	1.01
NICAM_nsw6.M3_vadv1d_getflux_new	3.06%	54.72%	1.00	1.19	1.54
streamlike_pattern1_check	9.38%	78.69%	1.00	1.00	1.01
streamlike_pattern2_check	2.38%	78.10%	1.00	1.54	2.94
streamlike_pattern3_check	4.15%	78.41%	1.00	0.99	1.00
MD	78.42%	0.02%	1.00	2.70	3.87
N-body	90.36%	0.00%	1.00	1.95	3.32
(Geometric mean)			1.00	1.34	2.01

To be analyzed on the next page

Basically, there is no impact for a kernel with a memory bandwidth bottleneck.

Performance Impact of SIMD Width-Conscious Optimization (Tuning)

For FFB.spmmv_vec8

```
[SIMD width 256 bits]
182              !$omp parallel default(none) private(IP,II,JJ,IP2)
183              !$omp&shared(NP,LIST,AX,A,X)
184              !$omp do
185              !ocl nounroll
186              !ocl swp
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<   SOFTWARE PIPELINING(IPC: 3.01, ITR: 48, MVE: 3, POL: S)
          <<<   PREFETCH(HARD) Expected by compiler :
          <<<     LIST, A, AX
          <<< Loop-information  End >>>
187    1  p         DO IP=1,NP
191    1             !ocl nounroll,loop_nofission
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<   SIMD(VL: 8)
          <<< Loop-information  End >>>
192    2  p   v         DO JJ=1,8
193    2  p   v           AX(JJ,IP  )=AX(JJ,IP  )+A( 1,IP  )*X(JJ,LIST( 1,IP  ))
194    2  p   v           AX(JJ,IP  )=AX(JJ,IP  )+A( 2,IP  )*X(JJ,LIST( 2,IP  ))
  :
226    2  p   v         ENDDO
  :
238    1  p         ENDDO
239              !$omp end do
240              !$omp end parallel
```

Eight innermost rotations are expanded based on the SIMD width, and scheduling is optimized in the external loop.

```
[SIMD width 128 bits]
182              !$omp parallel default(none) private(IP,II,JJ,IP2)
183              !$omp&shared(NP,LIST,AX,A,X)
184              !$omp do
185              !ocl nounroll
186              !ocl swp
187    1  p         DO IP=1,NP
191    1             !ocl nounroll,loop_nofission
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<   SIMD(VL: 4)
          <<<   PREFETCH(HARD) Expected by compiler :
          <<<     X
          <<<   PREFETCH(SOFT) : 30
          <<<    SEQUENTIAL : 30
          <<<     X: 28, AX: 2
          <<<   SPILLS :
          <<<     GENERAL   : SPILL 4  FILL 20
          <<<     SIMD&FP   : SPILL 0  FILL 4
          <<<     SCALABLE  : SPILL 0  FILL 0
          <<<     PREDICATE : SPILL 0  FILL 0
          <<< Loop-information  End >>>
192    2  p   v         DO JJ=1,8
193    2  p   v           AX(JJ,IP  )=AX(JJ,IP  )+A( 1,IP  )*X(JJ,LIST( 1,IP  ))
194    2  p   v           AX(JJ,IP  )=AX(JJ,IP  )+A( 2,IP  )*X(JJ,LIST( 2,IP  ))
  :
226    2  p   v         ENDDO
  :
238    1  p         ENDDO
239              !$omp end do
240              !$omp end parallel
```

As the SIMD width becomes small, the rotation remains even if the innermost loop is expanded with the SIMD, resulting in the optimization of scheduling being applied on the innermost loop.

If optimization (tuning) is performed with the SIMD width in mind, a performance gap greater than expected may arise.

Performance of the Variable-Length SIMD Specification

Performance of the variable-length SIMD (-Ksimd_reg_size=agnostic)

(Ratio when the variable-length SIMD specification + system SIMD width of 512 is 1)

Executed SIMD width

Program	-Ksimd_reg_size=512 (Default)	-Ksimd_reg_size=agnostic		
	SIMD width = 512	SIMD width = 512	SIMD width = 256	SIMD width = 128
Adventure.region1	1.00	1.00	1.69	3.32
Adventure.region2	1.02	1.00	1.62	2.94
FFB.callap_kernel2	**0.18**	1.00	0.87	0.92
FFB.spmmv_vec8	**0.25**	1.00	1.09	1.55
GAMERA.TIMER_COMP_MATVEC_IF	0.85	1.00	1.64	2.84
GENESIS.Nonb15F(June)	0.73	1.00	1.70	3.18
GENESIS.Nonb15F(July)	0.82	1.00	1.31	1.94
GENESIS.PairList(June)	0.59	1.00	1.07	1.15
GENESIS.PairList(July)	1.02	1.00	1.14	1.24
NICAM.Horizontal_Adv_flux	0.68	1.00	1.46	1.63
NICAM.Horizontal_Adv_limiter	0.96	1.00	1.20	1.73
NICAM.Radiation_adding	0.77	1.00	1.39	2.26
NICAM.Radiation_dtrn31	0.74	1.00	1.23	1.64
NICAM.Radiation_ptfit2	1.04	1.00	1.15	1.53
NICAM.Vertical_Adv_limiter	1.04	1.00	1.02	1.04
NICAM.diffusion	**0.10**	1.00	1.07	1.22
NICAM.divdamp	0.81	1.00	1.42	2.15
NICAM.vi_rhow_solver	1.01	1.00	1.28	1.89
NICAM_nsw6.M3_mp_nsw6_OMP9	1.00	1.00	1.01	1.07
NICAM_nsw6.M3_vadv1d_getflux_new	0.81	1.00	1.42	2.19
streamlike_pattern1_check	0.99	1.00	1.00	1.00
streamlike_pattern2_check	0.81	1.00	1.39	2.28
streamlike_pattern3_check	1.01	1.00	1.02	1.01
MD	0.98	1.00	2.71	3.88
N-body	0.96	1.00	1.81	3.46
(Geometric mean)	0.72	1.00	1.31	1.79

A value smaller than 1.0 indicates code whose performance lowers when the variable-length SIMD option is specified.
If SIMD width-conscious optimization is possible, the performance may deteriorate.

[Reference] Changes in Power Based on the SIMD Width

- Power efficiency when the SIMD width changes from 512 bits to 128 bits

Program	Performance improvement ratio (SIMD width changed from 512 to 128) * The larger the value, the better the performance	Power ratio (SIMD width changed from 512 to 128) * The larger the value, the greater the power	Performance improvement ratio/power ratio * The smaller, the more efficient
Adventure.region1	0.80	0.71	0.88
Adventure.region2	0.43	0.88	2.04
FFB.callap_kernel2	0.83	0.93	1.12
FFB.spmmv_vec8	0.20	0.86	4.43
GAMERA.TIMER_COMP_MATVEC_IF	0.36	0.93	2.62
GENESIS.Nonb15F(June)	0.30	0.94	3.13
GENESIS.Nonb15F(July)	0.50	1.02	2.03
GENESIS.PairList(June)	0.91	1.01	1.12
GENESIS.PairList(July)	0.86	0.90	1.05
NICAM.Horizontal_Adv_flux	0.56	0.94	1.68
NICAM.Horizontal_Adv_limiter	0.60	0.93	1.53
NICAM.Radiation_adding	0.55	1.01	1.83
NICAM.Radiation_dtrn31	0.55	0.84	1.52
NICAM.Radiation_ptfit2	0.73	0.82	1.13
NICAM.Vertical_Adv_limiter	0.97	0.96	0.99
NICAM.diffusion	0.28	0.91	3.24
NICAM.divdamp	0.37	0.88	2.38
NICAM.vi_rhow_solver	0.50	1.03	2.05
NICAM_nsw6.M3_mp_nsw6_OMP9	0.98	0.94	0.96
NICAM_nsw6.M3_vadv1d_getflux_new	0.65	0.99	1.53
streamlike_pattern1_check	1.00	1.04	1.04
streamlike_pattern2_check	0.34	0.87	2.56
streamlike_pattern3_check	0.98	1.05	1.07
(Geometric mean)	0.56	0.93	1.64

Less efficient than when the SIMD width is 512

Setting the SIMD width to 128 bits can slightly reduce power consumption. This cannot be said to be efficient, however, considering the performance drop rate.

Power Control

- About the Boost Mode
- About the Eco Mode
- Basic Kernel Power by Mode
- Performance Verification in Boost Mode
- Performance Verification in Eco Mode
- [Reference] Time-Series Power and Performance of an Actual Application (NICAM)

About the Boost Mode

- About the boost mode
  In normal mode, the A64FX operates at 2.0 GHz, taking into consideration the balance between performance and power efficiency. Allowing for cases where the application (job) needs to run as fast as possible, the boost mode is also provided in which the CPU operates at the frequency of 2.2 GHz. In boost mode, the CPU operates at a higher frequency than in normal mode, leading to a rise in its drive voltage.
- In boost mode, the CPU operates at the frequency of 2.2 GHz, while the frequency of the HBM is the same as in normal mode. The following table shows the code characteristics for which improvement in performance can be expected in boost mode.

Code characteristics	Effect of the boost mode on performance
L1 bandwidth bottleneck	Can be expected
L2 bandwidth bottleneck	Can be expected
Memory bandwidth bottleneck	Cannot be expected
Operation bottleneck	Can be expected
Access latency bottleneck	Can be expected
Tofu communication bandwidth bottleneck	Cannot be expected
Tofu communication latency bottleneck	Can be partially expected (CPU operating part only)

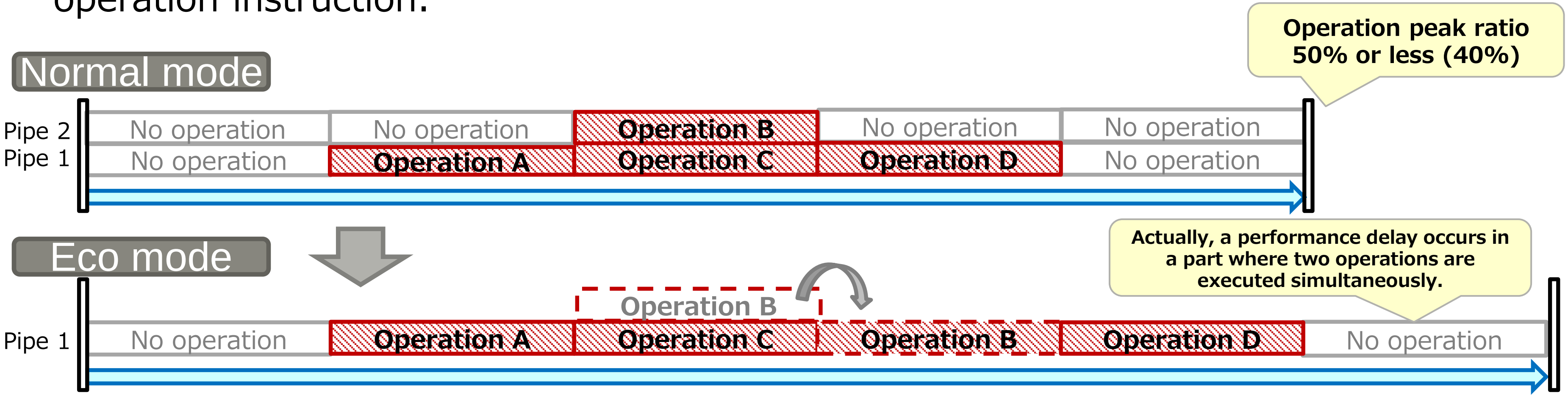
About the Eco Mode (1/2)

- About the eco mode
  In eco mode, Power Knob, which uses only one FPU pipeline, is turned on, while the bottom-up power is reduced by half to only the power for one floating-point operation pipe. The eco mode is effective in reducing power consumption in the running and standby states although the peak floating-point operation performance declines.

- Behavior of the operation pipeline in normal mode and eco mode

  Since the operation pipeline is reduced to only one pipe in eco mode, the performance is certain to drop for code whose floating-point operation peak ratio exceeds 50%.

  Even in cases where the floating-point operation peak ratio does not exceed 50%, the performance may be affected depending on the operation timing of the operation instruction.



About the Eco Mode (2/2)

- The following table shows the code characteristics for which an impact on (drop in) performance can be expected in eco mode.
- Power consumption is certainly reduced more in eco mode than in normal mode.

Code characteristics	Impact of the eco mode on performance
L1 bandwidth bottleneck	None
L2 bandwidth bottleneck	None
Memory bandwidth bottleneck	None
Operation bottleneck	Yes
Access latency bottleneck	None
Intra-node barrier bottleneck	None
Tofu communication bandwidth bottleneck	None
Tofu communication latency bottleneck	None

Basic Kernel Power by Mode (1/2): Verification Conditions

FUJITSU

■ Verification conditions

Item	Details
Measurement pattern	- L1 cache bandwidth bottleneck code - L2 cache bandwidth bottleneck code - Memory bandwidth bottleneck code (three patterns, each with different array values) (1) All 0s (2) Serial numbers (0, 1, 2, 3, 4, ...) (3) 64-byte block of 0s (all bits set to 0) and 64-byte block of the minimum values (all bits set to 1) set alternately - Latency bottleneck (L2 cache) code
Access range * bss area used	- L1 cache bandwidth bottleneck code: 48 KB (3/4 of the L1 size) * Per core - L2 cache bandwidth bottleneck code: 4 MB (1/2 of the L2 size) * Per CMG - Memory bandwidth bottleneck code: 240 MB (30 times the L2 size) * Per CMG - Latency bottleneck code: 1.8 MB * Per CMG
Array type	- Cache bandwidth bottleneck code: Double precision real type - Memory bandwidth bottleneck code/latency bottleneck code: 8-byte integer type
Number of parallel measurements (processes and threads)	4 processes and 12 threads (1 process per CMG and 1 thread per core)
Compilation option	-Kfast,openmp

Code to be measured 1 (Cache/memory bandwidth bottleneck)

```
!$omp parallel
 do j = 1, iter
!$omp do
    Do i = 1, n
       y(i)=x1(i) + c0 * x2(i)
    End Do
!$omp end do nowait
 enddo
!$omp end parallel
```

Code to be measured 2 (latency bottleneck)

```
for (i = 0; i < rep; i++) {
  p = index2[0];
  for (j = 0; j < NL; j++) {
    p = (uint64_t **)*p;
  }
  ans = p;
}
```

Basic Kernel Power by Mode (2/2): Power Values

Verification results

Program	Floating-point operation peak ratio	Memory throughput peak ratio	Busy rate (Normal mode, eco mode off)					Power			
								Normal mode (2.0 GHz)		Boost mode (2.2 GHz)	
			L1 busy rate	L2 busy rate	Memory busy rate * The hardware problem remains unsolved.	Floating-point PL busy rate	Integer PL busy rate	Eco mode off	Eco mode on	Eco mode off	Eco mode on
DGEMM (* Reference)	(94%)	-	(79%)	-	-	(94%)	-	177	125	206	143
STREAM (*Reference)	-	(81%)	-	(92%)	(81%)	-	-	196	164	218	178
L1 bandwidth bottleneck	21.58%	0.00%	93.68%	0.00%	0.00%	54.87%	33.86%	159	132	188	152
L2 bandwidth bottleneck	7.55%	0.00%	98.09%	97.42%	0.00%	19.62%	9.67%	151	122	176	138
Memory bandwidth bottleneck (1)	0.00%	82.50%	76.66%	91.97%	82.50%	6.42%	3.02%	169	143	194	155
Memory bandwidth bottleneck (2)	0.00%	81.80%	75.91%	90.74%	81.80%	6.37%	2.99%	189	157	209	171
Memory bandwidth bottleneck (3)	0.00%	82.50%	76.61%	91.90%	82.50%	6.42%	3.02%	199	171	219	186
Latency bottleneck (L2 cache)	0.00%	0.00%	10.59%	12.67%	0.00%	0.00%	0.54%	121	88	139	99

(1): As the array values, 0s are used.
(2): As the array values, serial numbers (0, 1, 2, 3, 4, ...) are used.
(3): As the array values, 0s (all bits set to 0) and minimum values (all bits set to 1) are used alternately in 64-byte units.
Value in parentheses: Reference value

It has been confirmed that a difference of about 20 to 30 W arises due to the difference in the initial value.

Note) The power values differ for each individual CPU. The values shown above should be considered relative values.

[Reference] Application Kernel Busy Rate and Power Value by Mode

Relationship between busy rate and power

Program	Single/double precision	Floating-point operation peak ratio	Memory throughput peak ratio	Busy rate (Normal mode, eco mode off)					Corrected power			
									Normal mode (2.0 GHz)		Boost mode (2.2 GHz)	
				L1 busy rate	L2 busy rate	Memory busy rate * The hardware problem remains unsolved.	Floating-point PL busy rate	Integer PL busy rate	Eco mode off	Eco mode on	Eco mode off	Eco mode on
Adventure.region1	Double	23.5%	54.7%	66.0%	89.5%	54.7%	54.3%	1.6%	203	158	226	179
NICAM.Radiation_dtrn31	Single	6.3%	77.4%	53.9%	80.5%	77.4%	6.8%	8.5%	198	163	222	180
FFB.callap_kernel2	Single	38.8%	74.8%	57.8%	57.2%	74.8%	80.0%	3.5%	194	140	227	157
streamlike_pattern3_check	Single	4.0%	85.1%	43.3%	78.8%	85.1%	5.4%	4.2%	192	155	213	173
NICAM.Horizontal_Adv_flux	Single	13.8%	46.4%	72.3%	66.6%	46.4%	17.8%	10.5%	191	160	221	178
streamlike_pattern1_check	Single	11.9%	84.5%	46.2%	81.0%	84.5%	10.6%	1.5%	189	158	215	170
NICAM_nsw6.M3_mp_nsw6_OMP9	Single	10.5%	67.2%	41.4%	64.6%	67.2%	9.8%	3.4%	187	153	209	170
NICAM.vi_rhow_solver	Single	14.8%	54.3%	45.8%	53.5%	54.3%	14.6%	[illegible]	[illegible]	[illegible]	[illegible]	172
NICAM.divdamp	Single	13.2%	41.4%	53.3%	75.1%	41.4%	12.	[illegible]	[illegible]	[illegible]	[illegible]	168
Adventure.region0	Double	16.5%	93.7%	45.1%	81.0%	93.7%	19.8%	[illegible]	[illegible]	[illegible]	[illegible]	161
NICAM.Radiation_adding	Single	11.3%	88.2%	48.9%	84.8%	88.2%	9.1%	[illegible]	[illegible]	[illegible]	[illegible]	156
streamlike_pattern2_check	Single	2.3%	86.0%	48.0%	80.9%	86.0%	4.1%	[illegible]	[illegible]	[illegible]	[illegible]	161
NICAM_nsw6.M3_vadv1d_getflux_new	Single	3.0%	79.1%	37.1%	71.0%	79.1%	11.5%	5.6%	171	138	191	150
NICAM.Vertical_Adv_limiter	Single	34.5%	37.0%	26.9%	26.3%	37.0%	49.7%	4.3%	170	127	196	143
FFB.spmmv_vec8	Single	27.0%	32.3%	52.8%	27.0%	32.3%	39.9%	20.1%	166	131	185	150
NICAM.Radiation_ptfit2	Single	16.6%	49.0%	52.6%	46.5%	49.0%	41.3%	4.8%	166	127	189	141
NICAM.Horizontal_Adv_limiter	Single	8.6%	34.7%	37.2%	53.1%	34.7%	14.6%	8.1%	162	127	189	144
Adventure.region2	Double	20.0%	0.0%	78.3%	47.1%	0.0%	50.7%	4.3%	156	119	182	138
NICAM.diffusion	Single	17.4%	10.4%	50.6%	72.3%	10.4%	16.1%	13.1%	156	120	179	137
GAMERA.TIMER_COMP_MATVEC_IF	Single	21.0%	10.4%	57.7%	9.8%	10.4%	36.9%	15.0%	149	113	172	128
QCD.ddd_in_s_	Single	26.3%	0.1%	41.3%	18.5%	0.1%	20.9%	6.9%	149	113	172	129
QCD.jinv_ddd_in_s_	Single	27.4%	0.1%	49.4%	25.8%	0.1%	24.5%	7.0%	148	113	172	129
GENESIS.Nonb15F(June)	Single	7.4%	7.0%	51.3%	12.2%	7.0%	21.7%	1.6%	142	112	165	126
GENESIS.PairList(June)	Single	7.0%	9.1%	27.8%	6.8%	9.1%	18.9%	29.9%	139	106	160	119
GENESIS.Nonb15F(July)	Single	7.9%	5.7%	29.3%	7.2%	5.7%	13.9%	6.2%	133	101	151	114
GENESIS.PairList(July)	Single	5.5%	4.2%	24.0%	6.3%	4.2%	14.9%	31.9%	133	99	152	113

Sort

Programs with higher memory busy rates tend to appear higher in the list.

Performance Verification in Boost Mode

- Boost Mode: Performance Verification Conditions by Code Characteristic
- Boost Mode: Basic Kernel Performance
- [Reference] Application Kernel Measurement Results

Boost Mode: Performance Verification Conditions by Code Characteristic

■ Measurement conditions

		Pattern
Verification code	Basic operation kernel	(1) Triad L1 cache access (L1 bandwidth performance) (2) Triad L2 cache access (L2 bandwidth performance) (3) Triad memory access (memory bandwidth performance) (4) DGEMM (operation performance) (5) L1 access latency code (operation/L1 access latency performance)
	Application kernel	28 kernels
Number of cores to be measured	Basic operation kernel	(1) (5): 1 core execution, (2) (3) (4): 12 core execution (CMG0)
	Application kernel	12 core execution (CMG0)
Compilation option	Basic operation kernel	(1) (5) -Kfast (2) -Kfast,openmp -Kprefetch_sequential=soft ¥ -Kprefetch_cache_level=1 -Kprefetch_line=4 (3) -Kfast,openmp -Kzfill=18 ¥ -Kprefetch_sequential=soft -Kprefetch_line=9 -Kprefetch_line_L2=70 (4) -Kfast,openmp
	Application kernel	The options individually specified for each kernel are used as they are.
Access range	Basic operation kernel	(1) (5) Half the L1 cache size (32 KB) (2) Half the L2 cache size (4 MB) (3) 30 times the L2 cache size (240 MB) (4) TRANSA=N, TRANSB=N, M=23040, N=23040, K=640

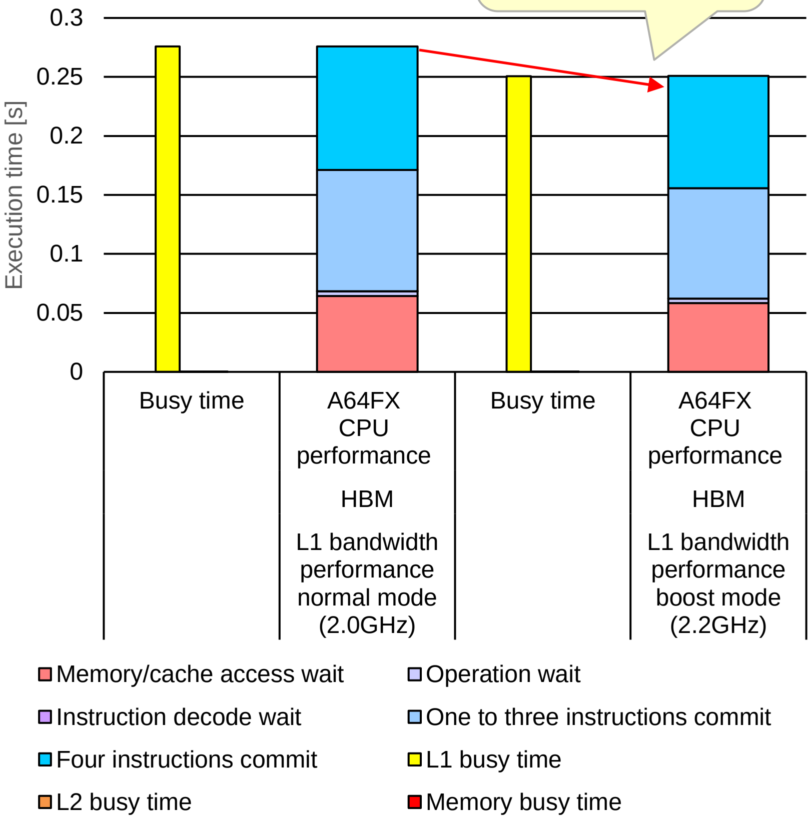
Boost Mode: Basic Kernel Performance (1/5)

FUJITSU

■ L1 bandwidth performance (1 core)

-> A performance improvement of about 10% has been confirmed.
(As expected)

```
Do j = 1, iter
    Do i = 1, n
        y(i)=x1(i) + c0 * x2(i)
    End Do
End Do
```



Performance improvement of about 10%

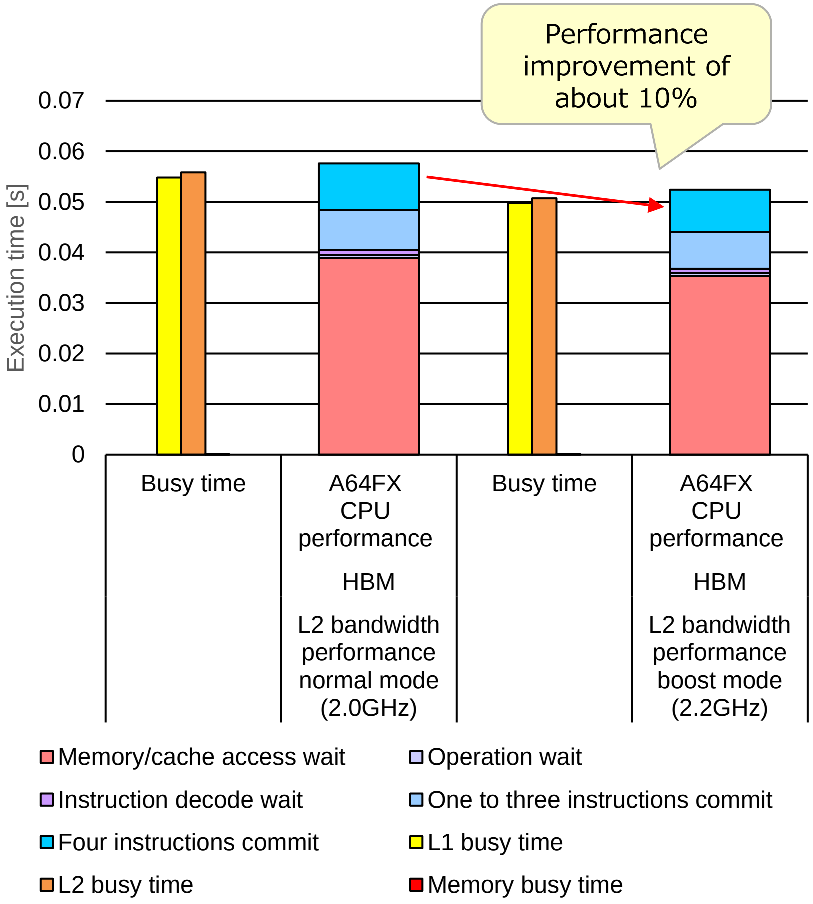
	A64FX CPU performance Normal mode (2.0 GHz)	A64FX CPU performance Boost mode (2.2 GHz)	Ratio (2.2GHz ÷2.0GHz)
Source code version	L1 bandwidth performance	L1 bandwidth performance	
Floating-point precision	Double precision	Double precision	
SIMD width	8	8	
Number of threads	1	1	
Aggregation thread number	0	0	
Execution time [s]	0.276	0.251	0.91
Total number of effective instructions	1.29.E+09	1.29.E+09	
GFLOPS (processes)	14.85	16.33	1.10
Memory throughput [GB/s/process]	0.00	0.00	
L1 busy rate/thread	99.95%	99.93%	
L2 busy rate/thread	0.00%	0.00%	
Memory busy rate/thread	0.00%	0.00%	
Floating-point pipeline busy rate/thread (FLA and FLB)	58.61% 34.38%	58.60% 34.37%	
L1 throughput [GB/s/thread]	178.1	196.0	1.10

Boost Mode: Basic Kernel Performance (2/5)

■ L2 bandwidth performance (1 CMG)

-> A performance improvement of about 10% has been confirmed.
(As expected)

```
!$omp parallel
   Do j = 1, iter
!$omp do
      Do i = 1, n
          y(i)=x1(i) + c0 * x2(i)
      End Do
!$omp end do nowait
   End Do
!$omp end parallel
```



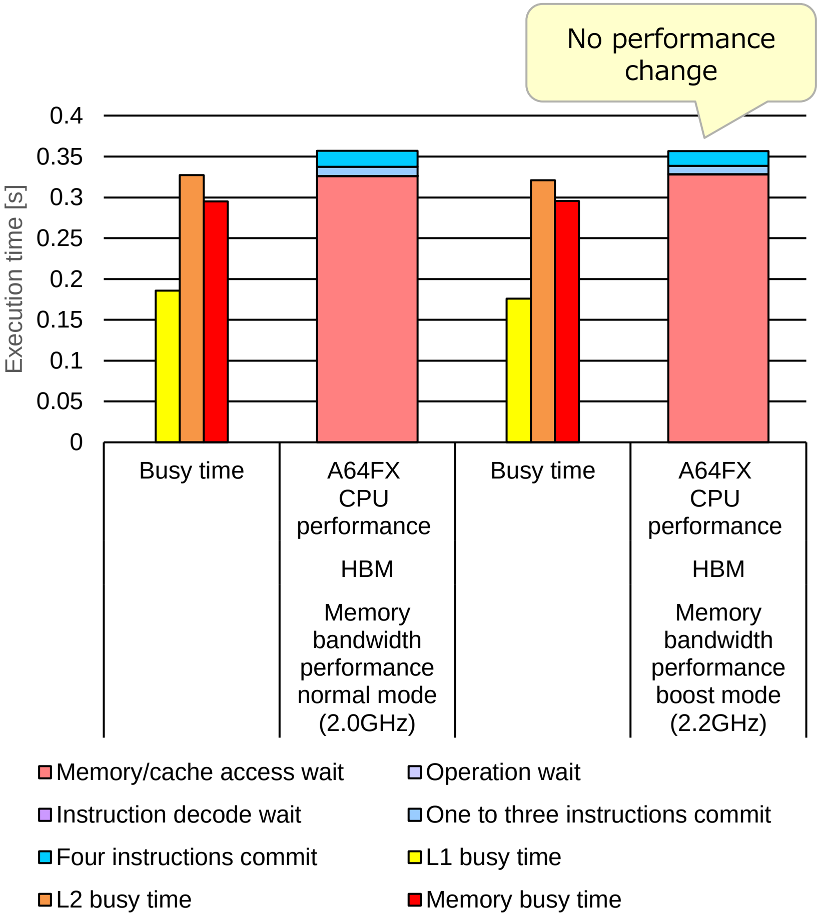
	A64FX CPU performance Normal mode (2.0 GHz)	A64FX CPU performance Boost mode (2.2 GHz)	Ratio (2.2GHz ÷2.0GHz)
Source code version	L2 bandwidth performance	L2 bandwidth performance	
Floating-point precision	Double precision	Double precision	
SIMD width	8	8	
Number of threads	12	12	
Aggregation thread number	0	0	
Execution time [s]	0.058	0.052	0.91
Total number of effective instructions	1.35.E+09	1.35.E+09	
GFLOPS (processes)	60.63	66.67	1.10
Memory throughput [GB/s/process]	0.00	0.01	
L1 busy rate/thread	95.08%	94.97%	
L2 busy rate/thread	96.93%	96.82%	
Memory busy rate/thread	0.00%	0.00%	
Floating-point pipeline busy rate/thread (FLA and FLB)	20.73% 10.88%	20.70% 10.86%	
L1D miss count/thread	1.38E+07	1.38E+07	
L1D miss demand rate/thread	1.00%	1.00%	
L2 throughput [GB/s/process]	727.59	799.99	1.10

Boost Mode: Basic Kernel Performance (3/5)

- Memory bandwidth performance (with zfill/1 CMG)

  -> No change in performance has been confirmed.
  (As expected)

```
!$omp parallel
   Do j = 1, iter
!$omp do
      Do i = 1, n
           y(i)=x1(i) + c0 * x2(i)
      End Do
!$omp end do nowait
   End Do
!$omp end parallel
```



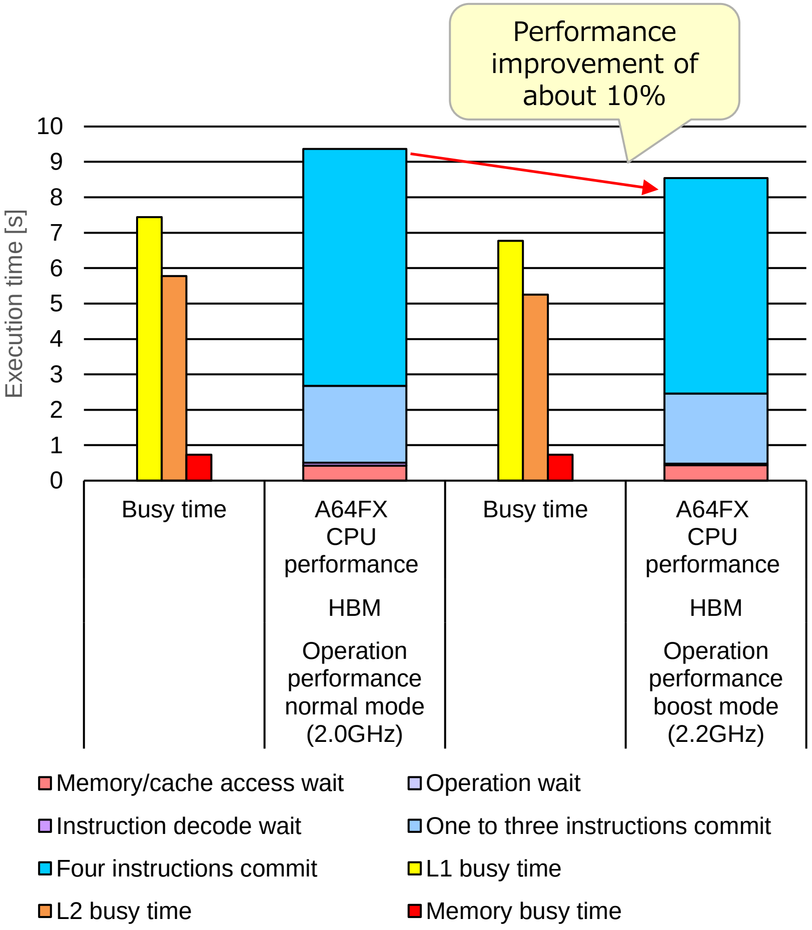
	A64FX CPU performance Normal mode (2.0 GHz)	A64FX CPU performance Boost mode (2.2 GHz)	Ratio (2.2GHz ÷2.0GHz)
Source code version	Memory bandwidth performance	Memory bandwidth performance	
Floating-point precision	Double precision	Double precision	
SIMD width	8	8	
Number of threads	12	12	
Aggregation thread number	0	0	
Execution time [s]	0.357	0.356	1.00
Total number of effective instructions	2.52.E+09	2.52.E+09	
GFLOPS (processes)	17.60	17.66	1.00
Memory throughput [GB/s/process]	211.50	212.18	1.00
L1 busy rate/thread	51.97%	49.32%	
L2 busy rate/thread	91.62%	90.04%	
Memory busy rate/thread	82.62%	82.88%	
Floating-point pipeline busy rate/thread (FLA and FLB)	6.46% 2.73%	5.89% 2.48%	
L1D miss count/thread	2.46E+07	2.46E+07	
L1D miss demand rate/thread	0.07%	0.07%	
L2 miss count/thread	1.64E+07	1.64E+07	
L2 miss demand rate/thread	0.29%	0.29%	

Boost Mode: Basic Kernel Performance (4/5)

- Operation performance (DGEMM/1 CMG)

-> A performance improvement of about 10% has been confirmed.
(As expected)

DGEMM parameters				Number of calls
TRANSA TRANSB	M	N	K	
NN	23040	23040	640	10



Performance improvement of about 10%

	A64FX CPU performance Normal mode (2.0 GHz)	A64FX CPU performance Boost mode (2.2 GHz)	Ratio (2.2GHz ÷2.0GHz)
Source code version	Operation performance	Operation performance	
Floating-point precision	Double precision	Double precision	
SIMD width	8	8	
Number of threads	12	12	
Aggregation thread number	0	0	
Execution time [s]	9.435	8.565	0.91
Total number of effective instructions	7.46.E+11	7.46.E+11	
GFLOPS (processes)	720.20	793.31	1.10
Memory throughput [GB/s/process]	20.02	21.91	
L1 busy rate/thread	79.40%	79.30%	
L2 busy rate/thread	61.62%	61.45%	
Memory busy rate/thread	7.82%	8.56%	
Floating-point pipeline busy rate/thread (FLA and FLB)	94.62% 94.38%	94.51% 94.28%	
L1D miss count/thread	1.80E+09	1.80E+09	
L1D miss demand rate/thread	0.45%	0.46%	
L2 miss count/thread	4.60E+07	4.61E+07	
L2 miss demand rate/thread	52.38%	53.49%	

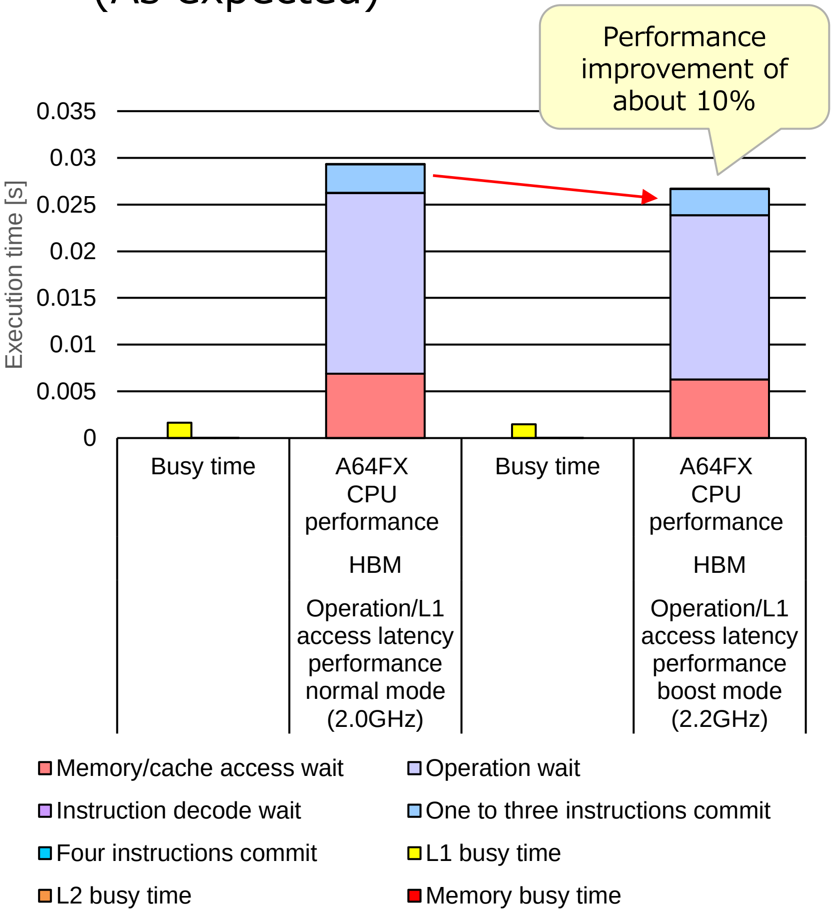
Boost Mode: Basic Kernel Performance (5/5)

Operation/L1 access latency performance

-> A performance improvement of about 10% has been confirmed.
(As expected)

```
Do j = 1, n
  Do i = 1, 8
    y(i,j) = c0 + y(i,j-m)*(c1 + y(i,j-m)* ¥
             (c2 + y(i,j-m)*(c3 + y(i,j-m)* ¥
             (c4 + y(i,j-m)*(c5 + y(i,j-m)* ¥
             (c6 + y(i,j-m)*(c7 + y(i,j-m)* ¥
             (c8 + y(i,j-m)* c9))))))))
  End Do
End Do
```

m=1

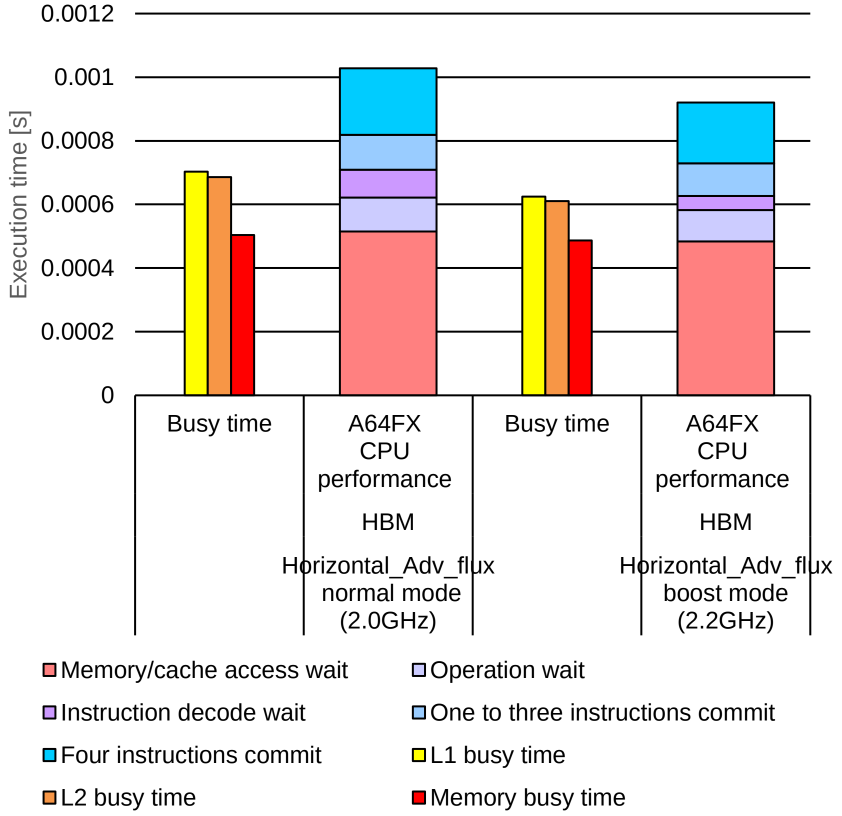


Performance improvement of about 10%

	A64FX CPU performance Normal mode (2.0 GHz)	A64FX CPU performance Boost mode (2.2 GHz)	Ratio (2.2GHz ÷2.0GHz)
Source code version	Operation/L1 access latency performance	Operation/L1 access latency performance	
Floating-point precision	Double precision	Double precision	
SIMD width	8	8	
Number of threads	1	1	
Aggregation thread number	0	0	
Execution time [s]	0.029	0.027	0.91
Total number of effective instructions	7.72.E+06	7.74.E+06	
GFLOPS (processes)	2.52	2.76	1.10
Memory throughput [GB/s/process]	0.00	0.00	
L1 busy rate/thread	5.55%	5.54%	
L2 busy rate/thread	0.01%	0.01%	
Memory busy rate/thread	0.00%	0.00%	
Floating-point pipeline busy rate/thread (FLA and FLB)	4.39% 4.36%	4.39% 4.36%	

[Reference]
Application Kernel Measurement Results (1/2)

■ NICAM Horizontal_Adv_flux

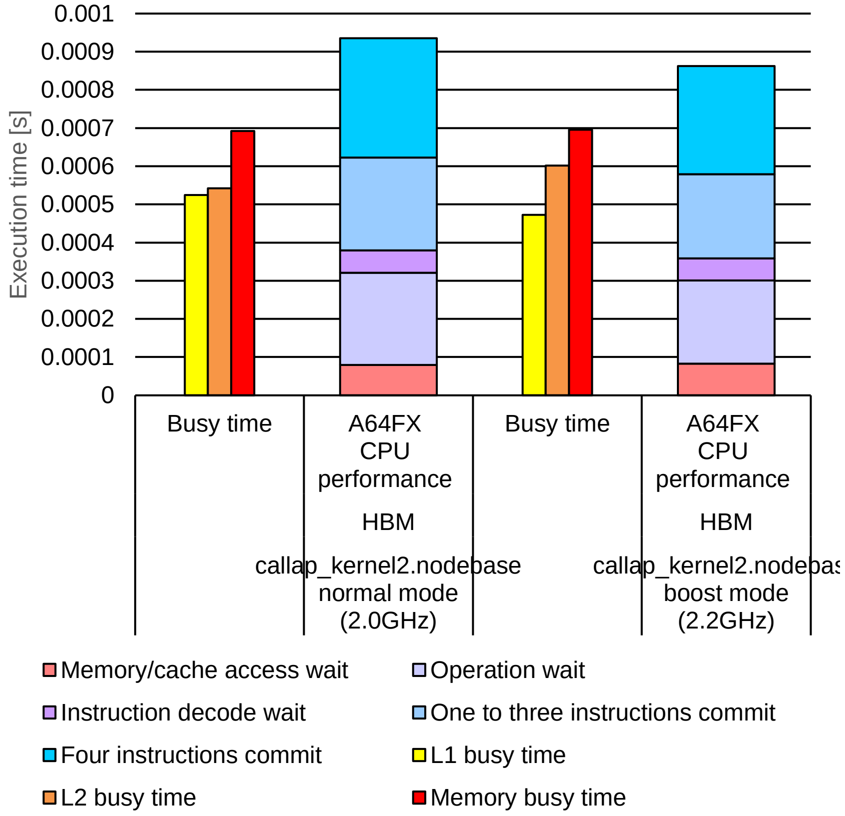


	A64FX CPU performance Normal mode (2.0 GHz)	A64FX CPU performance Boost mode (2.2 GHz)
Source code version	Horizontal_Adv_flux	Horizontal_Adv_flux
Floating-point precision	Single precision	Single precision
SIMD width	16	16
Number of threads	12	12
Aggregation thread number	0	0
Execution time [s]	0.001005	0.000929
Total number of effective instructions	2.43.E+07	2.43.E+07
GFLOPS (processes)	108.75	117.62
Memory throughput [GB/s/process]	123.04	133.74
L1 busy rate/thread	68.40%	67.87%
L2 busy rate/thread	66.77%	66.28%
Memory busy rate/thread	49.04%	52.90%
Floating-point pipeline busy rate/thread (FLA and FLB)	18.06% 14.00%	17.86% 13.85%
L1D miss count/thread	7.99E+04	8.04E+04
L1D miss demand rate/thread	20.81%	20.78%
L2 miss count/thread	2.18E+04	2.61E+04
L2 miss demand rate/thread	6.78%	7.92%

* There is a PA measurement overhead of about 50 to 80 μ.

[Reference]
Application Kernel Measurement Results (2/2)

■ FFB callap_kernel2.nodebase



	A64FX CPU performance Normal mode (2.0 GHz)	A64FX CPU performance Boost mode (2.2 GHz)
Source code version	callap_kernel2. nodebase	callap_kernel2. nodebase
Floating-point precision	Double precision	Double precision
SIMD width	8	8
Number of threads	12	12
Aggregation thread number	0	0
Execution time [s]	0.000939	0.000858
Total number of effective instructions	4.13.E+07	4.13.E+07
GFLOPS (processes)	282.65	309.34
Memory throughput [GB/s/process]	185.36	203.37
L1 busy rate/thread	56.05%	54.81%
L2 busy rate/thread	57.98%	69.72%
Memory busy rate/thread	74.01%	80.58%
Floating-point pipeline busy rate/thread (FLA and FLB)	76.90% 61.84%	74.75% 60.10%
L1D miss count/thread	5.09E+04	5.09E+04
L1D miss demand rate/thread	2.15%	2.13%
L2 miss count/thread	5.05E+04	5.05E+04
L2 miss demand rate/thread	0.89%	0.87%

* There is a PA measurement overhead of about 50 to 80 μ.

Performance Verification in Eco Mode

- Eco Mode: Performance Verification Conditions by Code Characteristic
- Eco Mode: Basic Kernel Performance
- [Reference] Impact of the Eco Mode on Application Kernel Performance
- [Reference] Application Kernel Measurement Results
- [Reference] Time-Series Power and Performance of an Actual Application (NICAM)

Eco Mode: Performance Verification Conditions by Code Characteristic

Measurement conditions

		Pattern
Verification code	Basic operation kernel	(1) Triad L1 cache access (L1 bandwidth performance) (2) Triad L2 cache access (L2 bandwidth performance) (3) Triad memory access (memory bandwidth performance) (4) DGEMM (operation performance) (5) L1 access latency code (operation/L1 access latency performance)
	Application kernel	28 kernels
Number of cores to be measured	Basic operation kernel	(1) (5): 1 core execution, (2) (3) (4): 12 core execution (CMG0)
	Application kernel	12 core execution (CMG0)
Compilation option	Basic operation kernel	(1) (5) -Kfast (2) -Kfast,openmp -Kprefetch_sequential=soft ¥ -Kprefetch_cache_level=1 -Kprefetch_line=4 (3) -Kfast,openmp -Kzfill=18 ¥ -Kprefetch_sequential=soft -Kprefetch_line=9 -Kprefetch_line_L2=70 (4) -Kfast,openmp
	Application kernel	The options individually specified for each kernel are used as they are.
Access range	Basic operation kernel	(1) (5) Half the L1 cache size (32 KB) (2) Half the L2 cache size (4 MB) (3) 30 times the L2 cache size (240 MB) (4) TRANSA=N, TRANSB=N, M=23040, N=23040, K=640

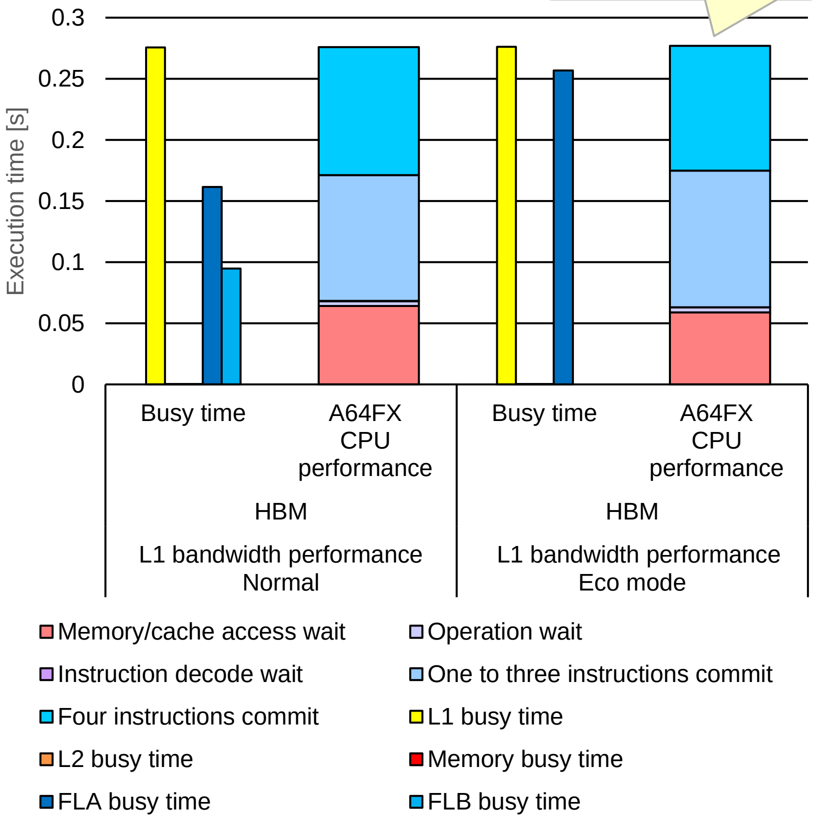
Eco Mode: Basic Kernel Performance (1/5)

L1 bandwidth performance (1 core)

-> No impact on performance has been confirmed.
(As expected)

```
Do j = 1, iter
    Do i = 1, n
        y(i)=x1(i) + c0 * x2(i)
    End Do
End Do
```

No performance change



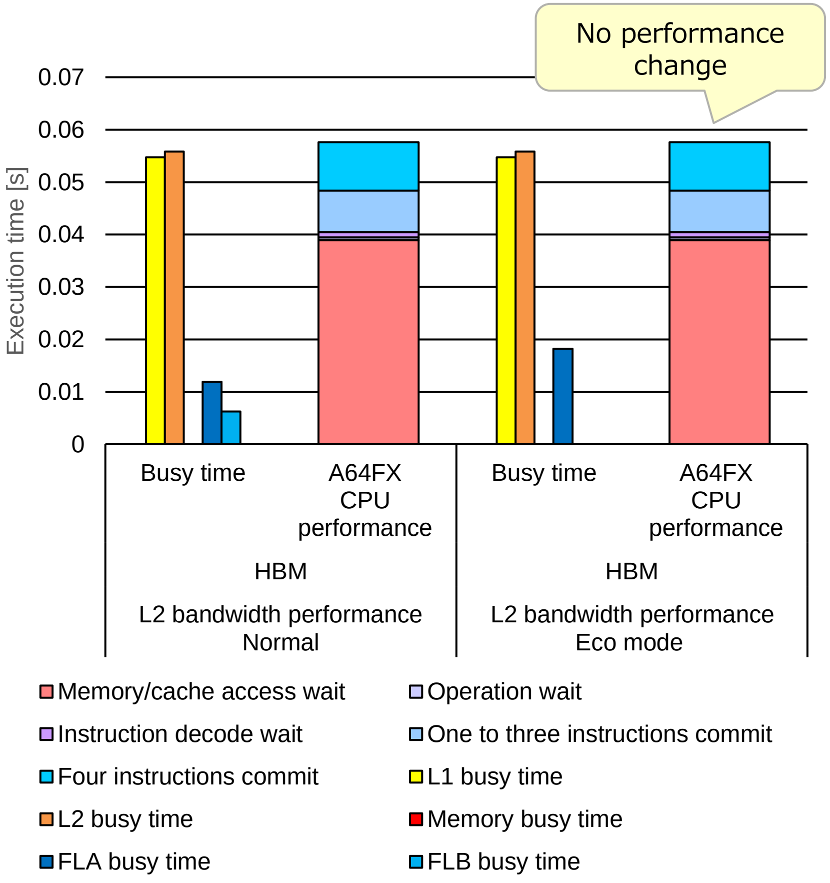
	A64FX CPU performance Normal	A64FX CPU performance Eco mode	Ratio (Eco mode ÷ normal)
Source code version	L1 bandwidth performance	L1 bandwidth performance	
Floating-point precision	Double precision	Double precision	
SIMD width	8	8	
Number of threads	1	1	
Aggregation thread number	0	0	
Execution time [s]	0.276	0.277	1.00
Total number of effective instructions	1.29.E+09	1.29.E+09	
GFLOPS (processes)	14.85	14.78	1.00
Memory throughput [GB/s/process]	0.00	0.00	
L1 busy rate/thread	99.95%	99.75%	
L2 busy rate/thread	0.00%	0.00%	
Memory busy rate/thread	0.00%	0.00%	
Floating-point pipeline busy rate/thread (FLA and FLB)	58.61% 34.38%	92.74% 0.00%	
L1 throughput [GB/s/thread]	178.1	177.4	1.00

Eco Mode: Basic Kernel Performance (2/5)

- L2 bandwidth performance (1 CMG)

-> No impact on performance has been confirmed.
(As expected)

```
!$omp parallel
   Do j = 1, iter
!$omp do
      Do i = 1, n
           y(i)=x1(i) + c0 * x2(i)
      End Do
!$omp end do nowait
   End Do
!$omp end parallel
```



	A64FX CPU performance Normal	A64FX CPU performance Eco mode	Ratio (Eco mode ÷ normal)
Source code version	L2 bandwidth performance	L2 bandwidth performance	
Floating-point precision	Double precision	Double precision	
SIMD width	8	8	
Number of threads	12	12	
Aggregation thread number	0	0	
Execution time [s]	0.058	0.058	1.00
Total number of effective instructions	1.35.E+09	1.35.E+09	
GFLOPS (processes)	60.63	60.68	1.00
Memory throughput [GB/s/process]	0.00	0.00	
L1 busy rate/thread	95.08%	95.00%	
L2 busy rate/thread	96.93%	96.87%	
Memory busy rate/thread	0.00%	0.00%	
Floating-point pipeline busy rate/thread (FLA and FLB)	20.73% 10.88%	31.58% 0.00%	
L1D miss count/thread	1.38E+07	1.38E+07	
L1D miss demand rate/thread	1.00%	1.01%	
L2 throughput [GB/s/process]	727.59	728.15	1.00

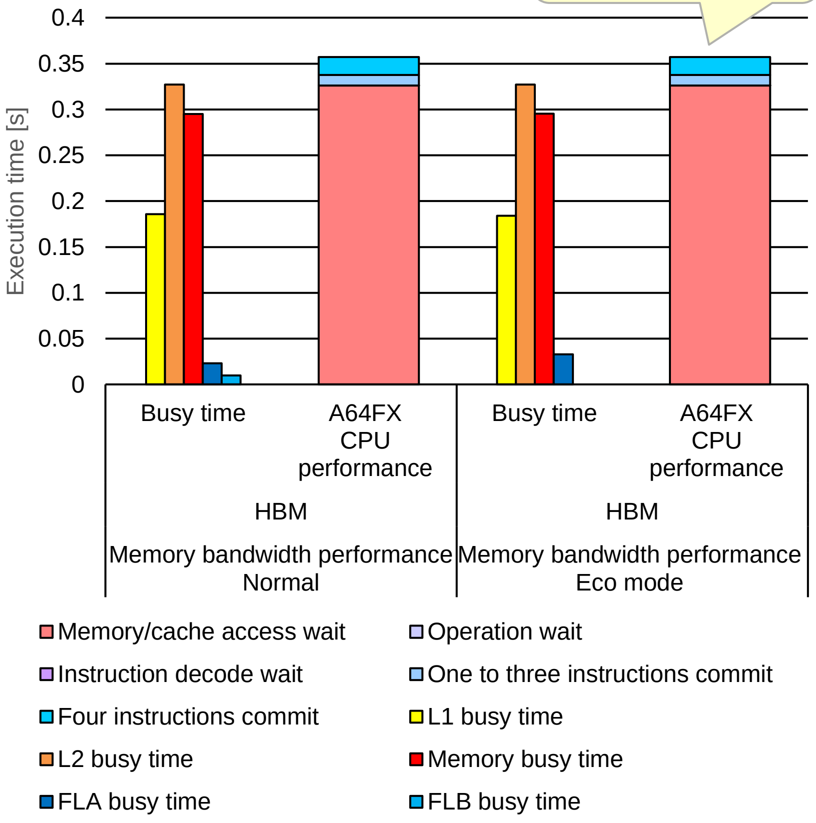
Eco Mode: Basic Kernel Performance (3/5)

- Memory bandwidth performance (with zfill/1 CMG)

  -> No change in performance has been confirmed. (As expected)

```
!$omp parallel
   Do j = 1, iter
!$omp do
      Do i = 1, n
           y(i)=x1(i) + c0 * x2(i)
      End Do
!$omp end do nowait
   End Do
!$omp end parallel
```

No performance change



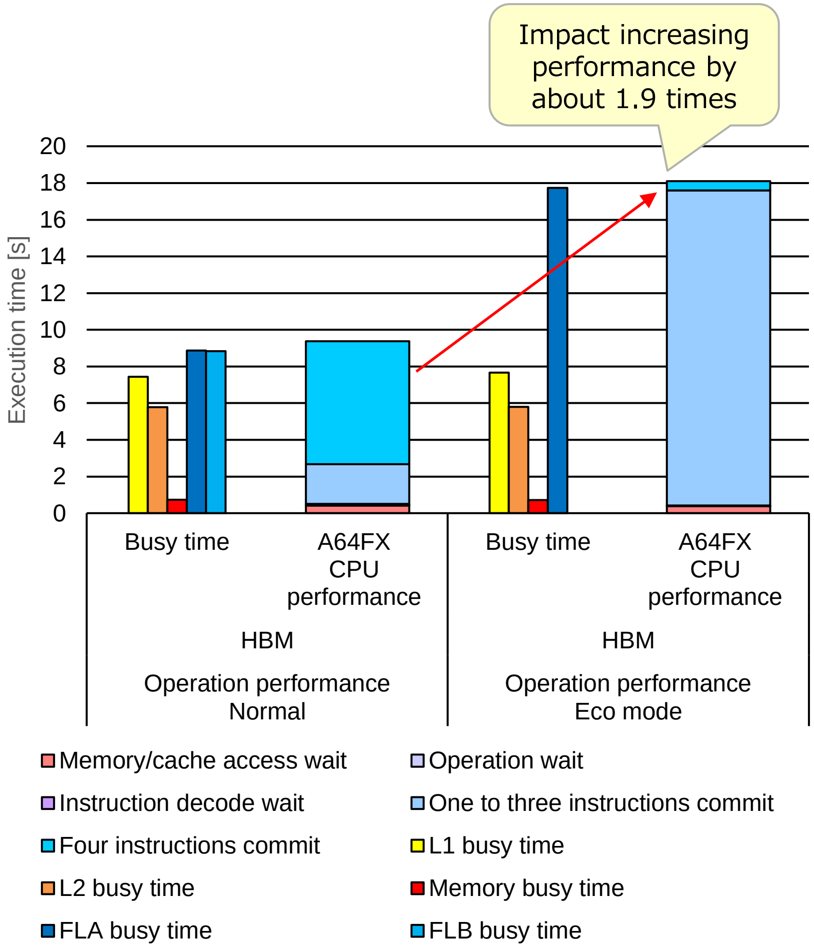
	A64FX CPU performance Normal	A64FX CPU performance Eco mode	Ratio (Eco mode ÷ normal)
Source code version	Memory bandwidth performance	Memory bandwidth performance	
Floating-point precision	Double precision	Double precision	
SIMD width	8	8	
Number of threads	12	12	
Aggregation thread number	0	0	
Execution time [s]	0.357	0.357	1.00
Total number of effective instructions	2.52.E+09	2.52.E+09	
GFLOPS (processes)	17.60	17.62	1.00
Memory throughput [GB/s/process]	211.50	211.78	1.00
L1 busy rate/thread	51.97%	51.50%	
L2 busy rate/thread	91.62%	91.55%	
Memory busy rate/thread	82.62%	82.73%	
Floating-point pipeline busy rate/thread (FLA and FLB)	6.46% 2.73%	9.18% 0.00%	
L1D miss count/thread	2.46E+07	2.46E+07	
L1D miss demand rate/thread	0.07%	0.07%	
L2 miss count/thread	1.64E+07	1.64E+07	
L2 miss demand rate/thread	0.29%	0.29%	

Eco Mode: Basic Kernel Performance (4/5)

Operation performance (DGEMM/1 CMG)

-> An impact increasing performance by about 1.9 times has been confirmed.

DGEMM parameters				Number of calls
TRANSA TRANSB	M	N	K	
NN	23040	23040	640	10



Impact increasing performance by about 1.9 times

	A64FX CPU performance Normal	A64FX CPU performance Eco mode	Ratio (Eco mode ÷ normal)
Source code version	Operation performance	Operation performance	
Floating-point precision	Double precision	Double precision	
SIMD width	8	8	
Number of threads	12	12	
Aggregation thread number	0	0	
Execution time [s]	9.435	18.093	1.92
Total number of effective instructions	7.46.E+11	7.46.E+11	
GFLOPS (processes)	720.20	375.56	0.52
Memory throughput [GB/s/process]	20.02	10.25	0.51
L1 busy rate/thread	79.40%	42.33%	
L2 busy rate/thread	61.62%	32.06%	
Memory busy rate/thread	7.82%	4.00%	
Floating-point pipeline busy rate/thread (FLA and FLB)	94.62% 94.38%	97.99% 0.00%	
L1D miss count/thread	1.80E+09	1.80E+09	
L1D miss demand rate/thread	0.45%	0.45%	
L2 miss count/thread	4.60E+07	4.57E+07	
L2 miss demand rate/thread	52.38%	52.32%	

Eco Mode: Basic Kernel Performance (5/5)

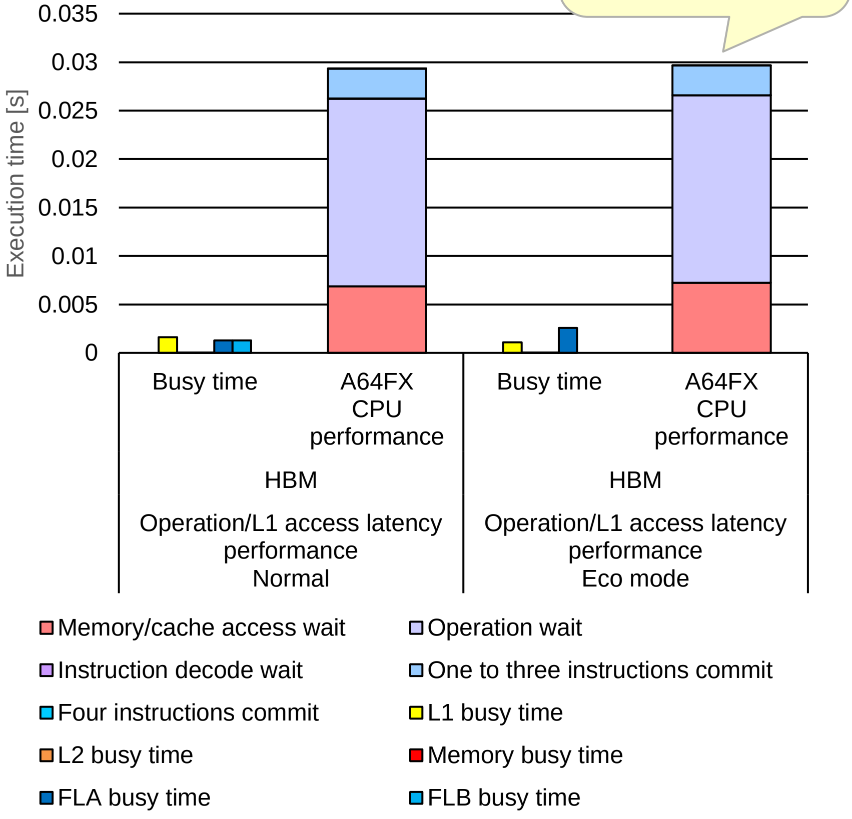
Operation/L1 access latency performance

-> No impact on performance has been confirmed.
(As expected)

m=1

```
Do j = 1, n
 Do i = 1, 8
  y(i,j) = c0 + y(i,j-m)*(c1 + y(i,j-m)* ¥
           (c2 + y(i,j-m)*(c3 + y(i,j-m)* ¥
           (c4 + y(i,j-m)*(c5 + y(i,j-m)* ¥
           (c6 + y(i,j-m)*(c7 + y(i,j-m)* ¥
           (c8 + y(i,j-m)* c9))))))))
 End Do
End Do
```

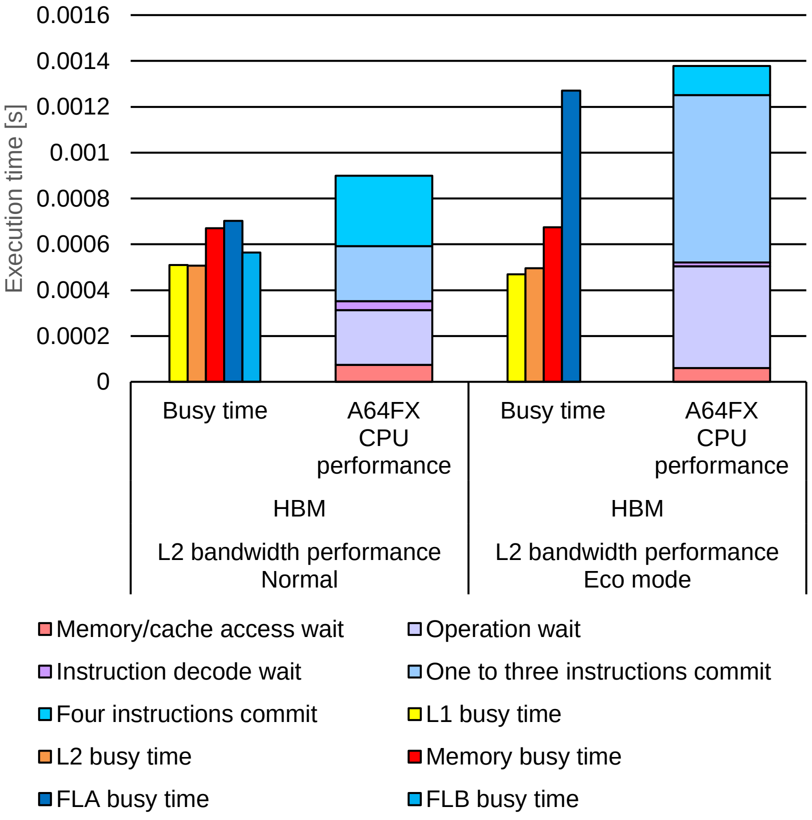
Almost no performance change
(Since operations need to be performed, the processing slightly slows down.)



	A64FX CPU performance Normal	A64FX CPU performance Eco mode	Ratio (Eco mode ÷ normal)
Source code version	Operation/L1 access latency performance	Operation/L1 access latency performance	
Floating-point precision	Double precision	Double precision	
SIMD width	8	8	
Number of threads	1	1	
Aggregation thread number	0	0	
Execution time [s]	0.029	0.030	1.01
Total number of effective instructions	7.72.E+06	7.74.E+06	
GFLOPS (processes)	2.52	2.48	0.99
Memory throughput [GB/s/process]	0.00	0.00	
L1 busy rate/thread	5.55%	3.72%	
L2 busy rate/thread	0.01%	0.01%	
Memory busy rate/thread	0.00%	0.00%	
Floating-point pipeline busy rate/thread (FLA and FLB)	4.39% 4.36%	8.64% 0.00%	

[Reference] Impact of the Eco Mode on Application Kernel Performance

■ FFB callap_kernel2.nodebase Normal mode and eco mode

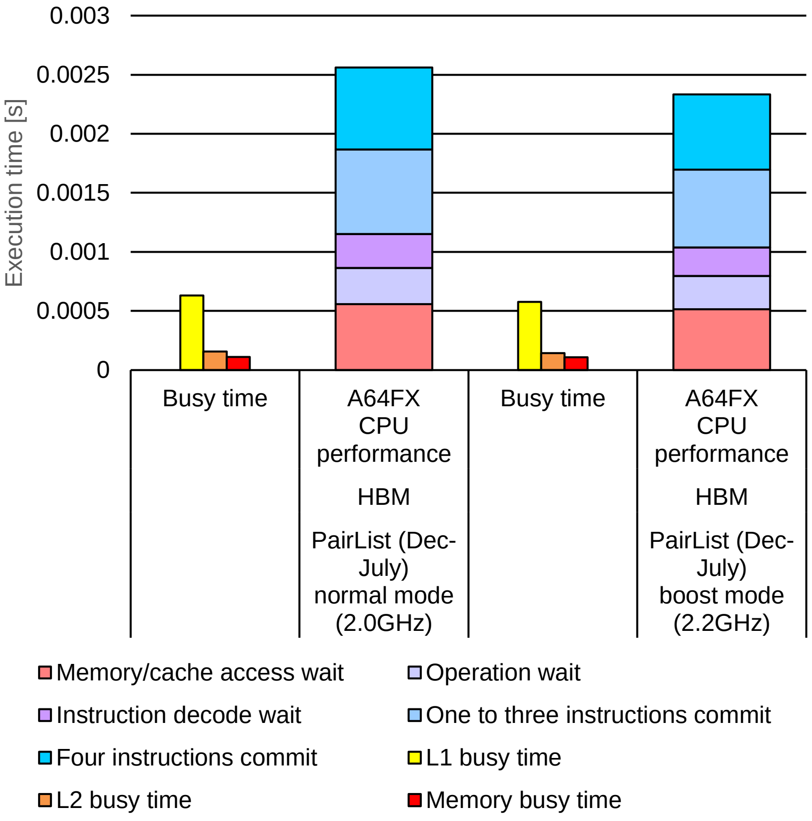


	A64FX CPU performance Normal	A64FX CPU performance Eco mode
Source code version	callap_kernel2. nodebase	callap_kernel2. nodebase
Floating-point precision	Single precision	Single precision
SIMD width	8	8
Number of threads	12	12
Aggregation thread number	0	0
Execution time [s]	0.000912	0.001392
Total number of effective instructions	3.45.E+06	3.45.E+06
GFLOPS (processes)	291.05	190.80
Memory throughput [GB/s/process]	190.72	125.15
L1 busy rate/thread	56.76%	34.02%
L2 busy rate/thread	56.42%	35.99%
Memory busy rate/thread	74.54%	48.90%
Floating-point pipeline busy rate/thread (FLA and FLB)	78.13% 62.85%	92.14% 0.00%
L1D miss count/thread	5.10E+04	5.10E+04
L1D miss demand rate/thread	2.23%	2.30%
L2 miss count/thread	5.07E+04	5.07E+04
L2 miss demand rate/thread	1.05%	1.01%

* Since the PA value measured with -Hmethod set to normal is used, the execution time is slightly longer.

[Reference] Application Kernel Measurement Results

■ GENESIS PairList (Dec-July)



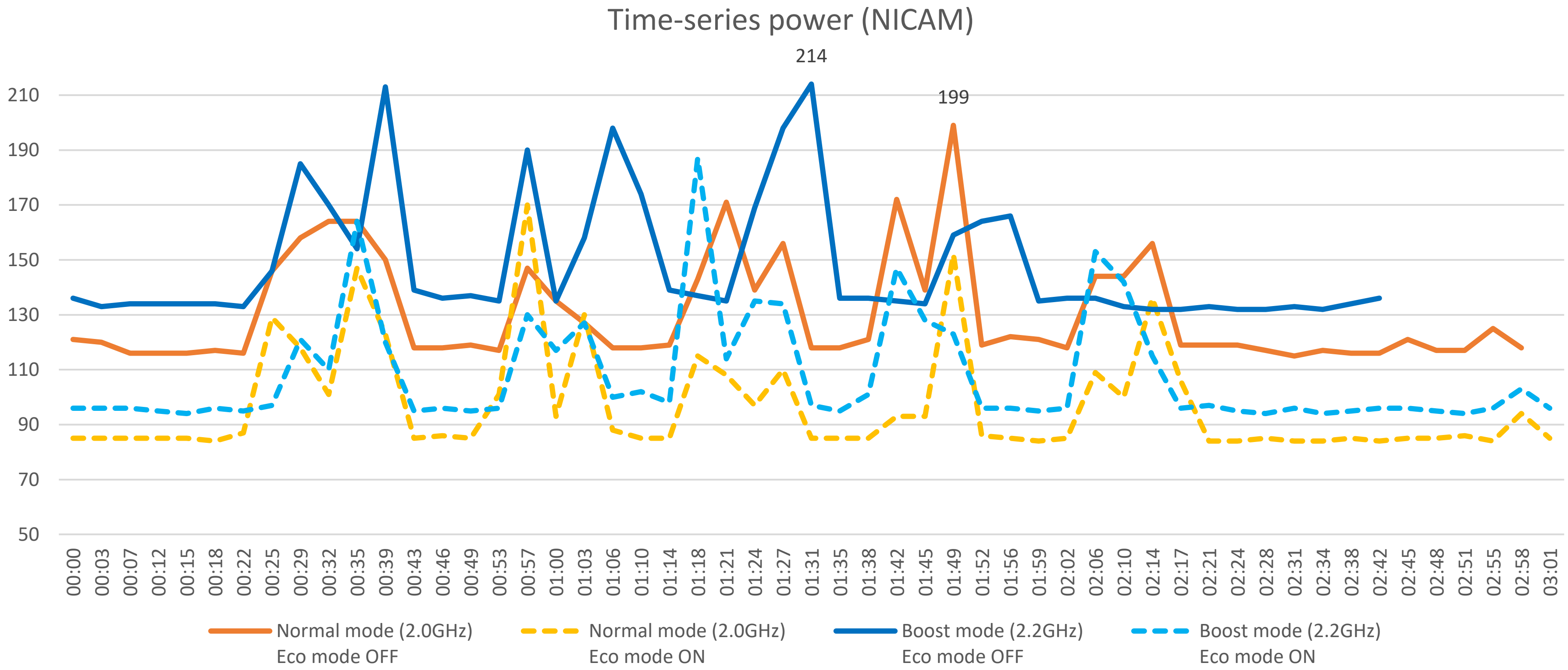
	A64FX CPU performance Normal mode (2.0 GHz)	A64FX CPU performance Boost mode (2.2 GHz)
Source code version	PairList (Dec-July)	PairList (Dec-July)
Floating-point precision	Double precision	Double precision
SIMD width	8	8
Number of threads	12	12
Aggregation thread number	0	0
Execution time [s]	0.002542	0.002347
Total number of effective instructions	1.00.E+08	1.00.E+08
GFLOPS (processes)	42.31	45.82
Memory throughput [GB/s/process]	10.84	11.61
L1 busy rate/thread	24.60%	24.67%
L2 busy rate/thread	6.14%	6.16%
Memory busy rate/thread	4.26%	4.56%
Floating-point pipeline busy rate/thread (FLA and FLB)	15.29% 13.22%	15.36% 13.29%
L1D miss count/thread	1.37E+04	1.36E+04
L1D miss demand rate/thread	73.13%	72.85%
L2 miss count/thread	6.28E+03	6.28E+03
L2 miss demand rate/thread	32.14%	30.22%

* There is a PA measurement overhead of about 50 to 80 μ.

[Reference] Time-Series Power and Performance of an Actual Application (NICAM)

- Time-series power aggregation

The power value was measured at intervals of about three seconds (vertical axis: power value, horizontal axis: elapsed time).
The July 2019 version of NICAM was used.



shaping tomorrow with you

Basic Kernel Performance

Note) The performance values shown herein may vary slightly depending on the measurement software (compiler and library).

- Evaluation Environment and Conditions
- Basic Operation Kernel Performance
  - Arithmetic Operations/Square Root
  - Mathematical Function Performance
- Other Basic Operation Kernel Evaluations
- Access Performance
- Throughput Performance
- Performance Values by Data Type
- Performance Impact by Alignment Changes
- Memory Copy Performance
- Inter-CMG Performance Evaluation
- OpenMP Overhead Evaluation

Evaluation Conditions and Environment

- Evaluation Environment: Measurement System
- Evaluation Conditions: Evaluation Code
- Evaluation Conditions: Compiler Options

Evaluation Environment: Measurement System (1/2)

		K	FX100	Haswell (Xeon E5-2698 v3)	Skylake (Xeon Platinum 8168)	A64FX 1 node
Frequency [GHz]		2.0	1.975	1.9 *1	1.9 *1	2.0
Number of CPUs/node		1	1	2	2	1
Number of cores/node		8	32	32	48	48
Number of CMGs/node			2			4
Memory size/node [GB]		16	32	256	384	32
Cache size	L1 [KiB/core]	32	64	32	32	64
	L2 [KiB/core]			256	1024	
	LL [MiB/CPU]	6	12	40	33	8
Cache latency	L1 [cycle]	4	5	4	4	5 (EX, short) 8 (FL, short) 11 (FL, long)
	L2 [cycle]			11	12	
	LL [cycle]	31	54	Up to 34	45	37 to 47
Cache throughput	L1 [B/cycle]	32	64	96	-	When hit: 128
	L2 [B/cycle]			64		
	LL [B/cycle]	16	32		-	42.7

*1: Set to operate at the fixed frequency of 1.9 GHz.

Evaluation Environment: Measurement System (2/2)

		K	FX100	Haswell	Skylake (Xeon Platinum 8168)	A64FX 1 node
Operation performance per node (Operation performance per core) [GFLOPS]	Double precision	128 (16)	1011.2 (31.6)	972.8 (30.4)	2918.4 (60.8)	3,072.0 (64.0)
	Single precision		2022.4 (63.2)	1945.6 (60.8)	5836.8 (121.6)	6144.0 (128.0)
Main memory latency [ns]		86	160		80	150
Theoretical memory bandwidth per node (Theoretical memory bandwidth per CMG) [GB/s]		64	480 (240)	136	255	1024 (256)

Evaluation Conditions (1/2): Evaluation Code

Evaluation conditions	Pattern
Evaluation code	(1) on L1$ - Arithmetic operations/square root - Mathematical function - Numeric function - Type conversion - Access performance (contiguous access) L1$ access - Access performance (stride access) - Access Performance (indirect access) (2) Access performance (contiguous access) L2$ access (3) Access performance (contiguous access) memory access
Number of cores to be measured	(1) 1 core execution (2) (3) 12 core execution (1 CMG)
Access range	(1) 3/4 of the L1 cache size (48 KB) (2) Half the L2 cache size (4 MB) (3) 3 times the L2 cache size (24 MB)

Evaluation Conditions (2/2): Compiler Options

Evaluation environment	Evaluation options
K (2.0GHz)	-Kfast -V -Nlst=t -Koptmsg=2 [Fortran only] -Cpp -Kautoobjstack,temparraystack [ilfunc evaluation only] -Kilfunc,nomfunc
FX100 (1.975GHz)	-Kfast -V -Nlst=t -Koptmsg=2 [Fortran only] -Cpp -Kautoobjstack,temparraystack [ilfunc evaluation only] -Kilfunc,nomfunc
PRIMERGY RX2530 M1 Haswell (FJ compiler) (1.9 GHz)	-Kfast,CORE_AVX2 -Nlst=t -Koptmsg=2 [Fortran only] -Cpp -Kautoobjstack,temparraystack
PRIMERGY RX2540 M4 Skylake (Intel compiler) (1.9 GHz)	-O3 -no-prec-div -fp-model fast=2 -xCORE-AVX512 -qopt-zmm-usage=high
A64FX (2.0GHz)	-Kfast -V -Nlst=t -Koptmsg=2 [Fortran only] -Cpp -Kautoobjstack,temparraystack

Basic Operation Kernel Performance

- Arithmetic Operations/Square Root
- Mathematical Function Performance

Arithmetic Operations/Square Root Measurement Results (1/2)

Arithmetic operations/square root (real type)

		K		FX100		PRIMERGY RX2530 M1		PRIMERGY RX2540 M4		A64FX			
		Floating-point operation performance (GFLOPS)	Performance improvement ratio	Floating-point operation performance (GFLOPS)	Performance improvement ratio	Floating-point operation performance (GFLOPS)	Performance improvement ratio	Floating-point operation performance (GFLOPS)	Performance improvement ratio	Floating-point operation performance (GFLOPS)	Performance improvement ratio	Operation efficiency (%)	SIMD conversion effect
Double precision	Addition	1.38	1.00	3.37	2.48	4.61	3.52	8.57	6.54	7.42	5.38	11.59	6.55
	Subtraction	1.38	1.00	3.40	2.49	4.62	3.52	8.46	6.46	7.42	5.38	11.59	6.55
	Multiplication	1.38	1.00	3.40	2.50	4.62	3.53	7.51	5.73	7.42	5.38	11.59	6.70
	Product-sum operation	2.87	1.00	6.96	2.46	9.51	3.49	15.79	5.80	14.84	5.18	23.19	6.55
	Division	10.39	1.00	19.06	1.86	3.22	0.33	9.24	0.94	39.57	3.81	61.84	7.94
	Reciprocal	10.79	1.00	19.02	1.79	5.60	0.55	8.38	0.82	39.85	3.69	62.26	8.33
	Square root	11.90	1.00	21.63	1.84	4.84	0.43	8.14	0.72	34.78	2.92	54.34	12.75
Single precision	Addition	1.69	1.00	6.57	3.93	9.11	5.66	13.59	8.45	13.84	8.18	10.81	12.16
	Subtraction	1.68	1.00	6.62	3.99	9.11	5.71	13.78	8.64	13.84	8.24	10.81	12.16
	Multiplication	1.69	1.00	6.62	3.96	9.09	5.65	13.49	8.39	13.84	8.18	10.81	12.16
	Product-sum operation	3.46	1.00	13.52	3.96	19.18	5.84	26.60	8.09	27.68	8.00	21.62	12.20
	Division	9.85	1.00	15.97	1.64	26.98	2.88	39.32	4.20	61.13	6.21	47.76	14.42
	Reciprocal	9.99	1.00	16.84	1.71	28.13	2.97	44.43	4.68	72.00	7.21	56.25	18.64
	Square root	9.61	1.00	17.15	1.81	8.41	0.92	49.29	5.40	52.07	5.42	40.68	23.83

Arithmetic Operations/Square Root Measurement Results (2/2)

■ Arithmetic operations (integer type)

		K		FX100		PRIMERGY RX2530 M1		PRIMERGY RX2540 M4		A64FX			
		Integer operation performance (GOPS)	Performance improvement ratio	Integer operation performance (GOPS)	Performance improvement ratio	Integer operation performance (GOPS)	Performance improvement ratio	Integer operation performance (GOPS)	Performance improvement ratio	Integer operation performance (GOPS)	Performance improvement ratio	Operation efficiency (%)	SIMD conversion effect
8-byte integer	Addition	0.77	1.00	3.43	4.51	4.74	6.47	8.86	12.09	7.42	9.62	11.59	7.91
	Subtraction	0.77	1.00	3.43	4.51	4.74	6.48	8.72	11.90	7.42	9.62	11.59	7.91
	Multiplication	0.33	1.00	3.47	10.64	1.21	3.87	5.86	18.66	7.42	22.44	11.59	7.84
	Product-sum operation	0.66	1.00	7.01	10.69	2.61	4.13	9.21	14.61	14.84	22.36	23.19	13.91
	Division	0.18	1.00	0.18	1.00	0.07	0.42	0.31	1.78	0.09	0.50	0.14	0.58
4-byte integer	Addition	0.84	1.00	3.88	4.71	9.41	11.86	15.53	19.56	13.84	16.56	10.81	14.54
	Subtraction	0.84	1.00	3.88	4.71	9.42	11.86	15.21	19.16	13.84	16.56	10.81	14.69
	Multiplication	0.33	1.00	3.80	11.60	7.28	23.07	11.90	37.72	13.84	41.54	10.77	14.62
	Product-sum operation	0.66	1.00	7.64	11.65	14.27	22.61	25.98	41.17	27.68	41.67	21.62	26.08
	Division	0.18	1.00	0.18	1.00	0.23	1.35	1.57	9.11	0.28	1.54	0.22	1.82

Mathematical Function Measurement Results

Mathematical function

Performance improvement ratio compared with the K (Clock normalized)

		K		FX100		PRIMERGY RX2530 M1		PRIMERGY RX2540 M4		A64FX		
		Floating-point operation performance (GFLOPS)	Performance improvement ratio	Floating-point operation performance (GFLOPS)	Performance improvement ratio	Floating-point operation performance (GFLOPS)	Performance improvement ratio	Floating-point operation performance (GFLOPS)	Performance improvement ratio	Floating-point operation performance (GFLOPS)	Performance improvement ratio	SIMD conversion effect
Double precision	atan	8.71	1.00	13.25	1.54	4.36	0.53	18.80	2.27	15.41	1.77	10.85
	atan2	8.87	1.00	9.69	1.11	2.32	0.28	16.19	1.92	6.88	0.78	15.46
	cos	11.55	1.00	19.43	1.70	4.49	0.41	21.10	1.92	24.44	2.11	8.04
	exp	7.93	1.00	15.22	1.94	4.05	0.54	22.55	2.99	18.04	2.27	10.70
	exp10	7.99	1.00	15.18	1.92	1.56	0.21	22.19	2.92	17.25	2.16	10.29
	log	6.97	1.00	7.87	1.14	3.14	0.47	16.01	2.42	11.80	1.69	8.09
	log10	7.62	1.00	9.45	1.26	2.36	0.33	20.70	2.86	11.84	1.55	8.82
	sin	11.54	1.00	19.47	1.71	5.42	0.49	22.21	2.03	24.17	2.10	7.98
	Exponentiation	7.48	1.00	8.88	1.20	2.66	0.37	13.32	1.88	8.47	1.13	14.97
Single precision	atan	7.29	1.00	9.76	1.36	1.72	0.25	29.51	4.26	36.30	4.98	22.38
	atan2	8.00	1.00	6.76	0.86	1.28	0.17	28.70	3.78	25.67	3.21	22.10
	cos	10.98	1.00	16.37	1.51	1.30	0.13	39.92	3.83	48.93	4.46	15.88
	exp	6.74	1.00	11.44	1.72	1.55	0.24	47.42	7.41	35.75	5.30	21.34
	exp10	7.37	1.00	11.65	1.60	1.52	0.22	43.10	6.15	37.59	5.10	23.35
	log	5.91	1.00	5.49	0.94	0.99	0.18	41.46	7.39	28.96	4.90	17.47
	log10	6.29	1.00	6.67	1.07	1.11	0.19	52.90	8.86	30.70	4.88	17.73
	sin	10.98	1.00	16.36	1.52	1.36	0.13	45.36	4.35	48.98	4.46	17.77
	Exponentiation	7.30	1.00	6.55	0.91	1.02	0.15	25.56	3.69	17.30	2.37	17.95

Mathematical Function Comparison with Other CPUs

		K (GFLOPS)	FX100 (GFLOPS)	Skylake (GFLOPS)	A64FX (GFLOPS)	Compared with K	Compared with FX100	Compared with Skylake
Double precision	atan	8.71	13.25	18.80	15.41	1.77	1.16	0.78
	atan2	8.87	9.69	16.19	6.88	0.78	0.71	0.40
	cos	11.55	19.43	21.10	24.44	2.11	1.26	1.10
	exp	7.93	15.22	22.55	18.04	2.27	1.19	0.76
	exp10	7.99	15.18	22.19	17.25	2.16	1.14	0.74
	log	6.97	7.87	16.01	11.80	1.69	1.50	0.70
	log10	7.62	9.45	20.70	11.84	1.55	1.25	0.54
	sin	11.54	19.47	22.21	24.17	2.10	1.24	1.03
	Exponentiation	7.48	8.88	13.32	8.47	1.13	0.95	0.60
Single precision	atan	7.29	9.76	29.51	36.30	4.98	3.72	1.17
	atan2	8.00	6.76	28.70	25.67	3.21	3.80	0.85
	cos	10.98	16.37	39.92	48.93	4.46	2.99	1.16
	exp	6.74	11.44	47.42	35.75	5.30	3.13	0.72
	exp10	7.37	11.65	43.10	37.59	5.10	3.23	0.83
	log	5.91	5.49	41.46	28.96	4.90	5.28	0.66
	log10	6.29	6.67	52.90	30.70	4.88	4.60	0.55
	sin	10.98	16.36	45.36	48.98	4.46	2.99	1.03
	Exponentiation	7.30	6.55	25.56	17.30	2.37	2.64	0.64
GEOMEAN						2.66	2.00	0.76

There is room for improvement, as of December 2019.

Other Basic Operation Kernel Evaluations

- Numeric Function Measurement Results
- Type Conversion Measurement Results

Numeric Function Measurement Results

Numeric function

		K		FX100		PRIMERGY RX2530 M1		PRIMERGY RX2540 M4		A64FX		
		Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	SIMD conversion effect
Double precision	abs	1.71	1.00	4.82	2.85	6.93	4.26	13.38	8.23	9.78	5.72	7.08
	max	1.42	1.00	3.43	2.45	4.54	3.38	8.08	6.00	7.42	5.24	6.55
	min	1.42	1.00	3.43	2.45	4.57	3.39	8.21	6.10	7.42	5.24	6.55
	mod	1.27	1.00	9.29	7.40	5.37	4.44	18.70	15.47	6.04	4.76	9.38
	sign	1.58	1.00	3.45	3.32	2.46	2.81	7.07	8.07	5.85	6.35	6.37
Single precision	abs	2.02	1.00	5.70	2.86	14.01	7.30	20.59	10.72	17.16	8.49	10.73
	max	1.73	1.00	3.88	2.27	9.00	5.47	12.14	7.39	13.84	7.99	12.16
	min	1.74	1.00	3.83	2.23	9.06	5.48	12.09	7.31	13.84	7.95	12.16
	mod	2.06	1.00	9.63	4.73	13.79	7.05	27.86	14.24	11.11	5.39	18.67
	sign	1.58	1.00	3.90	2.49	4.99	3.32	13.44	8.94	11.64	7.36	12.81

Type Conversion Measurement Results

Type conversion

	K		FX100		PRIMERGY RX2530 M1		PRIMERGY RX2540 M4		A64FX		
	Integer operation performance (GOPS)	Performance improvement ratio	Integer operation performance (GOPS)	Performance improvement ratio	Integer operation performance (GOPS)	Performance improvement ratio	Integer operation performance (GOPS)	Performance improvement ratio	Integer operation performance (GOPS)	Performance improvement ratio	SIMD conversion effect
dble (single precision real number)	1.64	1.00	4.79	2.96	3.79	2.44	9.19	5.91	9.06	5.54	5.78
dble (4-byte integer)	1.29	1.00	4.78	3.76	4.59	3.75	9.74	7.96	9.06	7.04	13.10
real (double precision real number)	1.91	1.00	5.39	2.86	4.64	2.56	8.70	4.80	11.19	5.86	7.59
real (4-byte integer)	1.66	1.00	5.96	3.64	12.07	7.67	18.23	11.57	17.21	10.38	24.87
int (double precision real number)	1.11	1.00	5.79	5.26	4.78	4.53	8.78	8.31	11.19	10.06	27.01
int (single precision real number)	1.13	1.00	5.98	5.34	11.99	11.13	17.09	15.86	17.21	15.18	40.48
aint (double precision real number)	0.83	1.00	4.79	5.82	3.56	4.50	6.37	8.05	9.78	11.75	6.23
aint (single precision real number)	0.87	1.00	5.95	6.92	7.39	8.94	12.60	15.23	17.21	19.77	10.80
nint (double precision real number)	0.80	1.00	4.07	5.15	0.83	1.10	3.92	5.15	9.78	12.19	26.65
nint (single precision real number)	0.82	1.00	4.25	5.22	2.49	3.17	9.10	11.61	17.26	20.93	47.52
anint (double precision real number)	0.69	1.00	4.79	7.04	1.24	1.89	5.25	8.03	9.78	14.21	6.13
anint (single precision real number)	0.71	1.00	5.87	8.37	2.47	3.66	9.17	13.58	17.21	24.21	10.80

Access Performance

- Basic Access Performance

Basic Access Performance (1/2)

Access performance (contiguous access): STREAM Triad case

L1 access

	K		FX100		PRIMERGY RX2530 M1		PRIMERGY RX2540 M4		A64FX			
	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Operation efficiency (%)	Operation efficiency (ideal value) (%)
Contiguous SIMD load × 2/ contiguous SIMD store × 1	2.87	1.00	6.94	2.45	9.02	3.31	11.69	4.29	14.84	5.17	23.19	25.00

L2 access One CMG evaluated for FX100 and A64FX; one CPU evaluated for the others

	K		FX100		PRIMERGY RX2530 M1		PRIMERGY RX2540 M4		A64FX	
	Throughput (GB/s)	Performance improvement ratio	Throughput (GB/s)	Performance improvement ratio	Throughput (GB/s)	Performance improvement ratio	Throughput (GB/s)	Performance improvement ratio	Throughput (GB/s)	Performance improvement ratio
Contiguous SIMD load × 2/ contiguous SIMD store × 1	133.77	1.00	418.62	3.13	301.74	2.26	312.42	2.34	698.88	5.22

Memory access One CMG evaluated for FX100 and A64FX; one CPU evaluated for the others

	K		FX100		PRIMERGY RX2530 M1		PRIMERGY RX2540 M4		A64FX		
	Throughput (GB/s)	Performance improvement ratio	Throughput (GB/s)	Performance improvement ratio	Throughput (GB/s)	Performance improvement ratio	Throughput (GB/s)	Performance improvement ratio	Throughput (GB/s)	Performance improvement ratio	Throughput efficiency (%)
Contiguous SIMD load × 2/ contiguous SIMD store × 1	34.89	1.00	103.52	2.97	43.30	1.24	77.20	2.21	150.68	4.32	78.48

Result without zfill
The result with zfill was 210 GB/s or so.

Basic Access Performance (2/2)

Access performance (stride access) L1 access

		K		FX100		PRIMERGY RX2530 M1		A64FX			
		Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Operation efficiency (%)	Operation efficiency (ideal value) (%)
No load/ indirect SIMD store × 1	Jump width 2	0.88	1.00	1.36	1.56	1.87	2.23	1.60	1.82	2.50	
	Jump width 16	0.82	1.00	1.00	1.23	1.75	2.24	1.41	1.71	2.20	
Indirect SIMD load × 2/ indirect SIMD store × 1	Jump width 2	1.22	1.00	1.58	1.31	2.23	1.93	1.96	1.61	3.07	
	Jump width 16	0.85	1.00	1.24	1.48	2.25	2.79	0.90	1.06	1.40	
Indirect SIMD load × 2/ contiguous SIMD store × 1	Jump width 2	1.82	1.00	2.41	1.34	2.27	1.32	4.72	2.60	7.38	10.00
	Jump width 16	0.91	1.00	2.25	2.49	2.01	2.32	3.05	3.34	4.76	5.56

Access performance (indirect access) L1 access

		K		FX100		PRIMERGY RX2530 M1		PRIMERGY RX2540 M4		A64FX			
		Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Floating-point operation performance (Gflops)	Performance improvement ratio	Operation efficiency (%)	Operation efficiency (ideal value) (%)
No load/ indirect SIMD store × 1	Fixed value	0.98	1.00	1.82	1.88	1.82	1.95	0.15	0.16	1.56	1.59	2.44	
	Contiguous	0.82	1.00	1.58	1.94	1.79	2.30	0.16	0.21	1.53	1.86	2.39	
	Jump width 16	0.75	1.00	0.91	1.23	1.80	2.53	0.39	0.54	1.22	1.63	1.90	
Indirect SIMD load × 2/ indirect SIMD store × 1	Fixed value	1.47	1.00	2.29	1.57	1.51	1.08	0.29	0.21	2.06	1.40	3.22	
	Contiguous	1.05	1.00	1.84	1.78	1.48	1.49	0.34	0.34	2.01	1.92	3.14	
	Jump width 16	0.55	1.00	1.13	2.08	1.50	2.88	0.64	1.23	0.90	1.65	1.41	
Indirect SIMD load × 2/ contiguous SIMD store × 1	Fixed value	1.45	1.00	2.19	1.52	1.74	1.26	1.99	1.44	4.12	2.84	6.44	10.00
	Contiguous	1.12	1.00	2.18	1.97	1.75	1.64	1.95	1.83	4.12	3.67	6.44	10.00
	Jump width 16	0.98	1.00	1.72	1.77	1.38	1.47	1.59	1.70	2.73	2.77	4.26	5.56

Throughput Performance

- Measurement Conditions
- Calculating Measurement Results for Comparison With Other CPUs
- L1 Cache Access
- L2 Cache Access
- Memory Access: Without zfill
- Memory Access: Access Across CMGs

Measurement Conditions

FUJITSU

Measurement conditions

	Pattern
Measurement pattern	Three patterns - L1 cache access throughput measurement - L2 cache access throughput measurement - Memory access throughput measurement
Cores to be measured	- L1 cache access throughput measurement -> 1 core execution (CMG0 computing core) - L2 cache/memory access throughput measurement -> 12 core execution (CMG0)
Compilation option	-Kfast * The prefetch and zfill options are described in the document as necessary.
Type	Double precision real type
Access range	-- bss is used. -- n (number of innermost loop iterations and array size) is as follows. - L1 = 3/4 of the L1 cache size (2048 for the A64FX) - L2 = 1/2 of the L2 cache size (174720 for the A64FX) - Memory = 3 times the L2 cache size (1048512 for the A64FX) * Each array is 256 byte aligned.
Number of outer loop iterations (iter)	- L1 = 1000000 - L2 = 10000 - Memory = 3000

Verification code

```
!$omp parallel
  Do j = 1, iter
!$omp do
     Do i = 1, n
        y(i)=x1(i) + c0 * x2(i)
     End Do
!$omp end do nowait
  End Do
!$omp end parallel
```

Specifications (throughput)
- L1 access: Read: 256 GB/s,
  Write: 128 GB/s
- L2 access: 1024 GB/s
- Memory access: 256 GB/s (1 CMG)

Calculating Measurement Results for Comparison With Other CPUs

Calculating measurement results for comparison with other CPUs

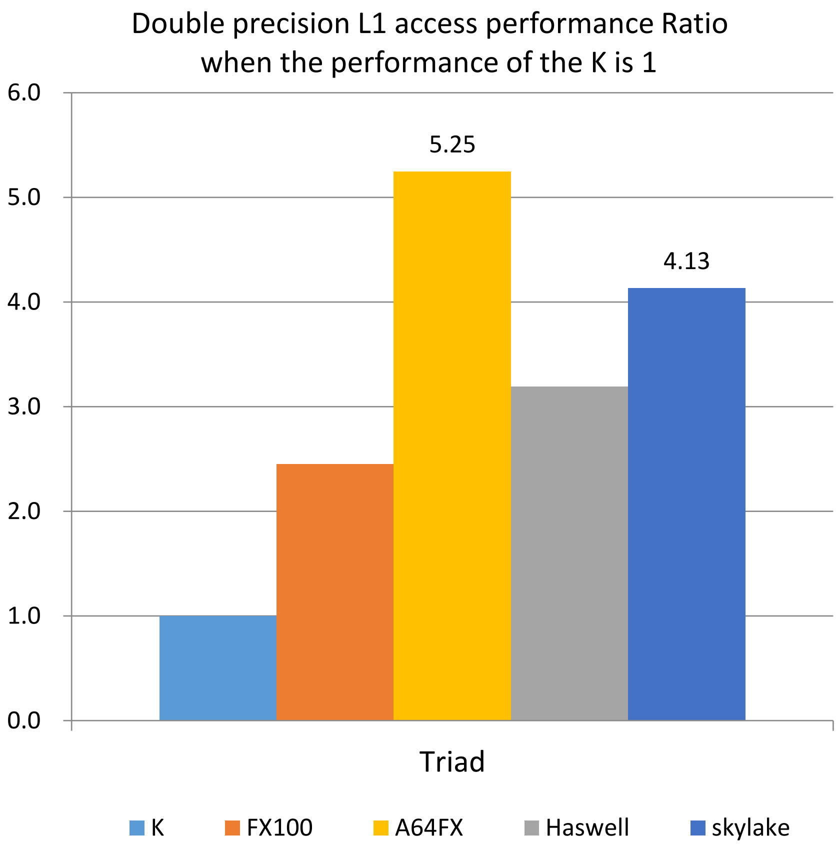
Measurement results	
Floating-point operation performance (Gflops)	Number of floating-point operations ÷ elapsed time ÷ 10^9 The number of floating-point operations for PRIMERGY (Haswell, Skylake) is the same as that for the FX100.
Integer operation performance (GOPS)	Number of integer operations ÷ elapsed time ÷ 10^9 The number of integer operations is the same as the number of floating-point operations.
Throughput (GB/s)	Data transfer volume ÷ elapsed time ÷ 10^9 As with the STREAM benchmark, the data transfer volume does not include that of the data read from those cache lines to which data is written.
Performance improvement ratio (when the performance of the K is 1)	Operation performance: Floating-point operation performance ÷ floating-point operation performance of the K Integer operation performance ÷ integer operation performance of the K Data access performance (throughput): Throughput ÷ throughput of the K The operation performance was calculated, with the CPU operating frequency changed to the same frequency as the K.

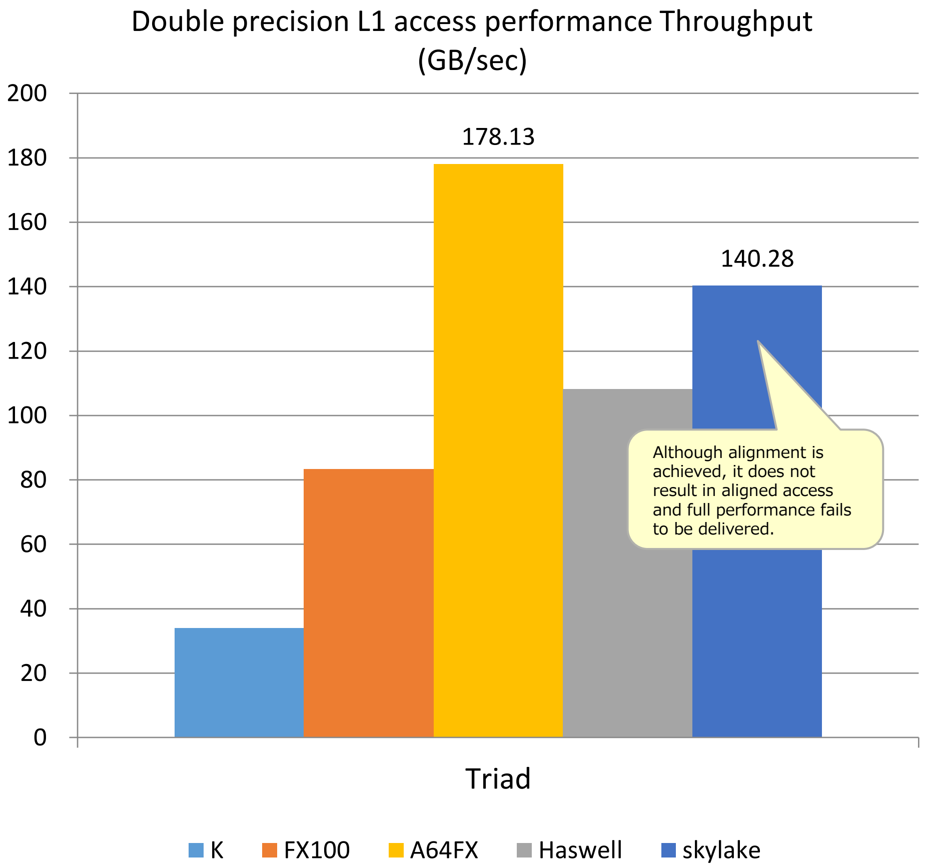
L1 Cache Access

- Ratio when the performance of the K is 1; throughput
  - Triad contiguous access evaluated

	Scatter ST	ST	Gather LD	LD	Expression
Triad contiguous access	0	1	0	2	y(i) = x1(i) + scalar * x2(i)

The specified values of the L1 cache throughput are:
Read: 256 GB/s,
Write: 128 GB/s





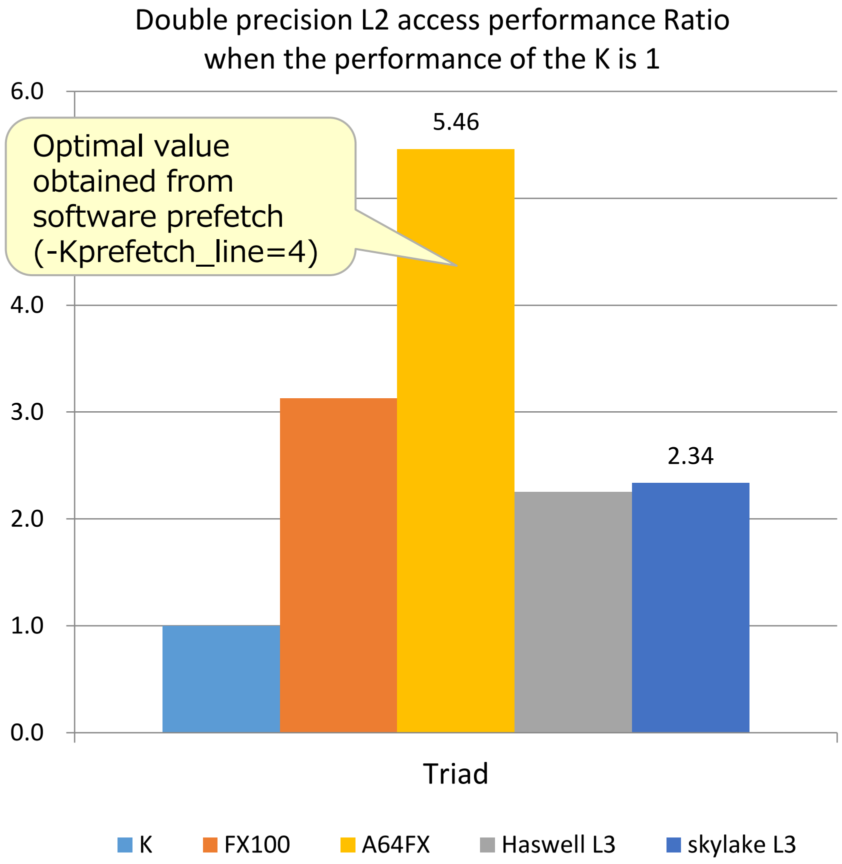
Although alignment is achieved, it does not result in aligned access and full performance fails to be delivered.

L2 Cache Access (1/3)

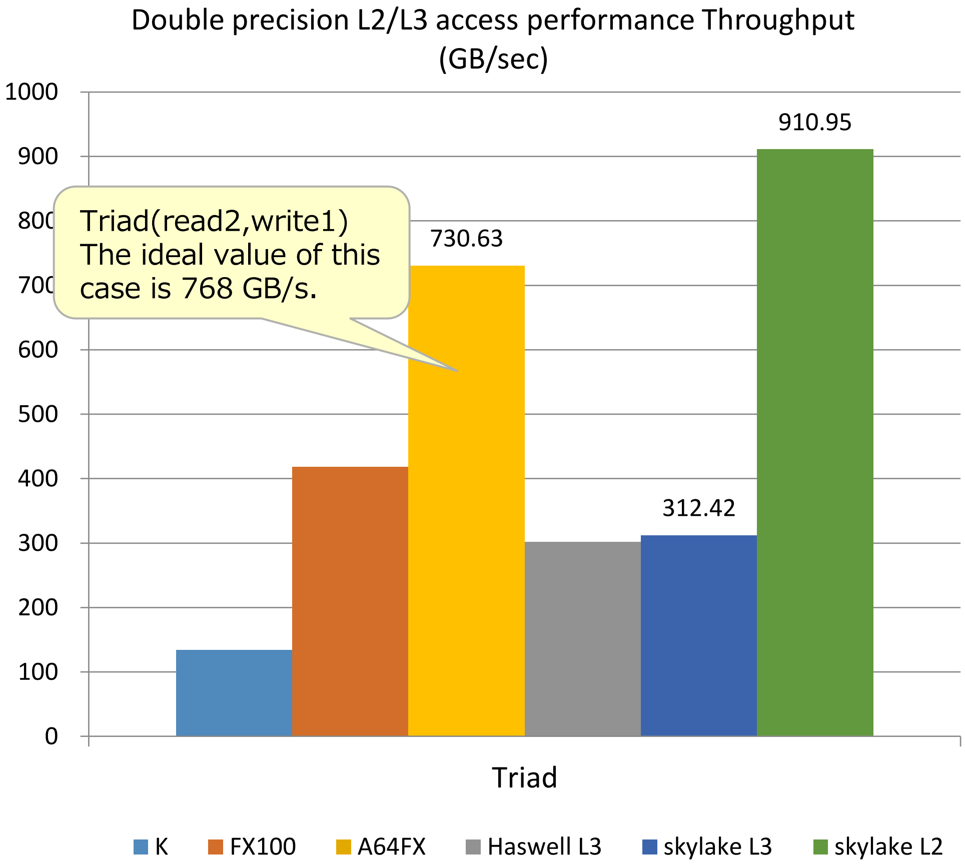
- Ratio when the performance of the K is 1; L2 throughput (per CMG)
  - Triad contiguous access evaluated

	Scatter ST	ST	Gather LD	LD	Expression
Triad contiguous access	0	1	0	2	y(i) = x1(i) + scalar * x2(i)

The specified values of the L2 cache throughput are: 1024 GB/s



Optimal value obtained from software prefetch (-Kprefetch_line=4)

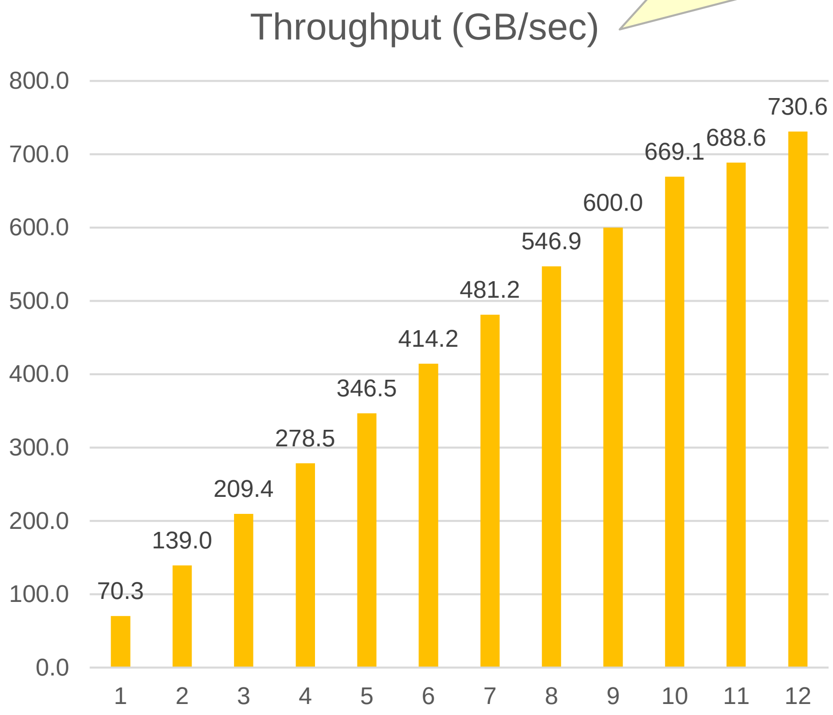


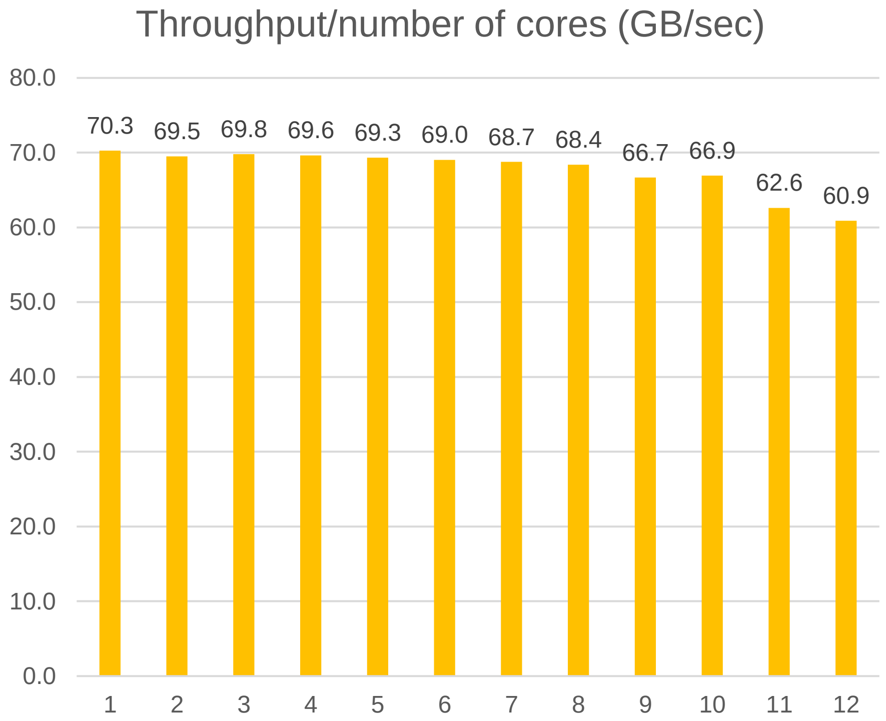
Triad(read2,write1) The ideal value of this case is 768 GB/s.

L2 Cache Access (2/3)

Number of executed threads and throughput

Using CMG0, 1 to 12 threads were executed.

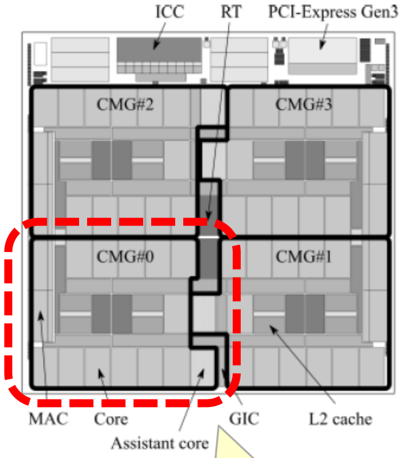




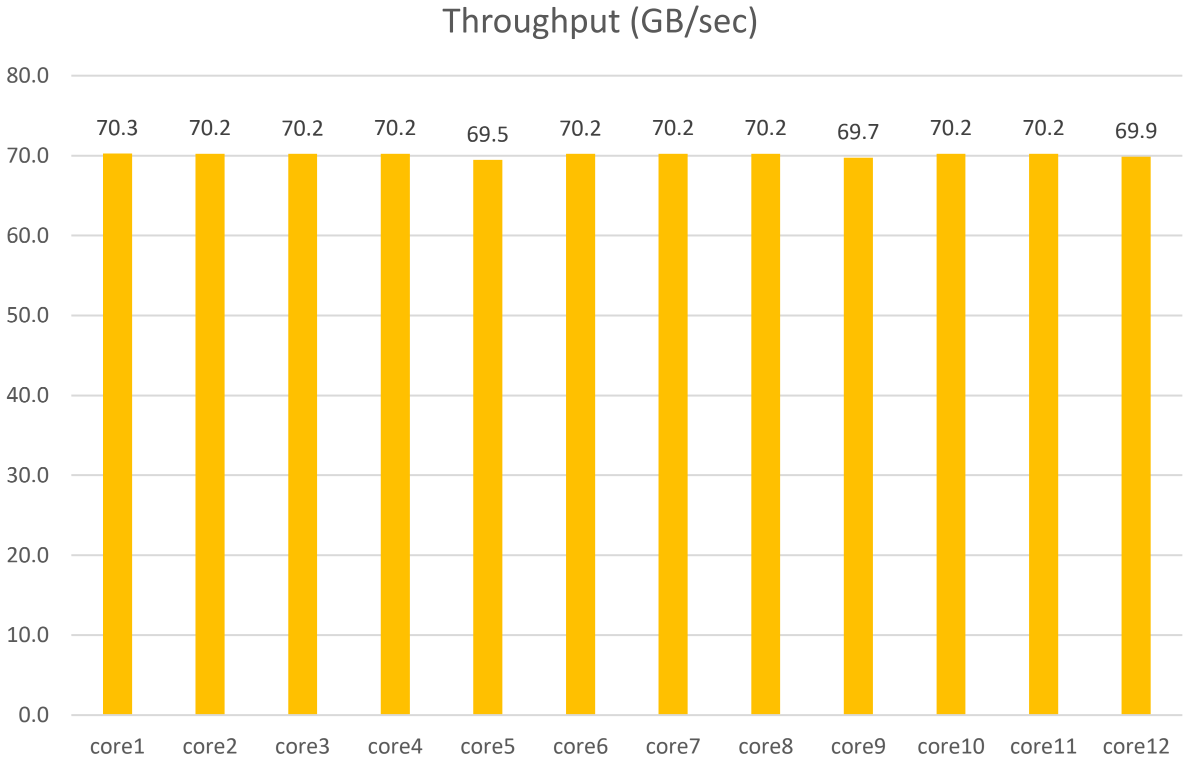
Almost all the performance of the CMG can be delivered through 10 threads (10 core) execution.

L2 Cache Access (3/3)

Difference in throughput due to the executed core



Using cores 1 to 12 of CMG0, a thread was executed per core.



No significant difference in throughput due to the executed core was observed.

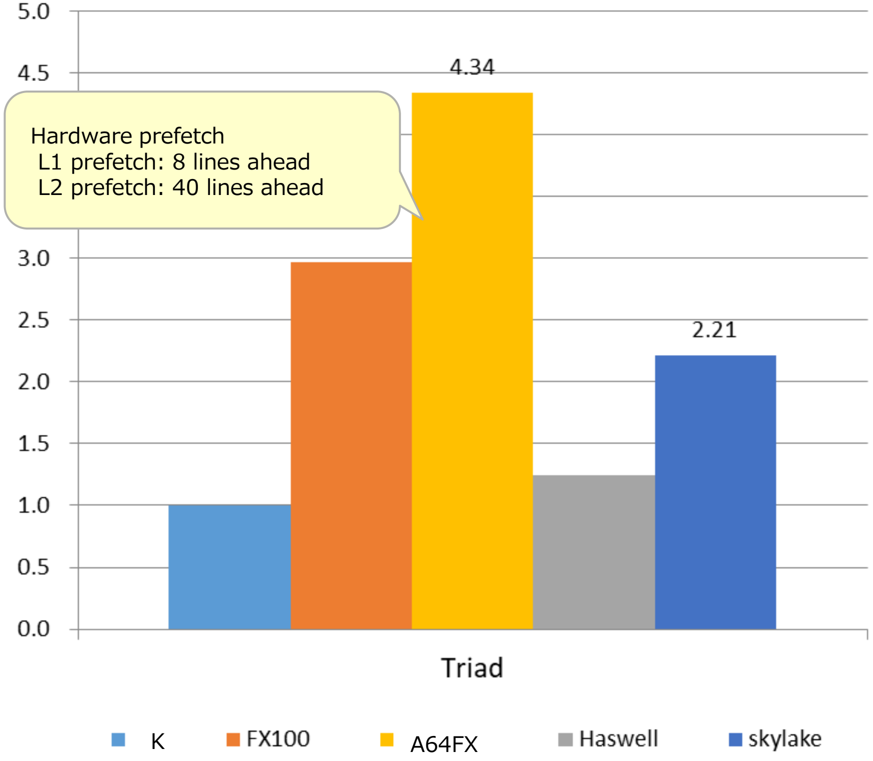
Memory Access: Without zfill (1/3)

- Ratio when the performance of the K is 1; memory throughput (per CMG)
  - Triad contiguous access evaluated

	Scatter ST	ST	Gather LD	LD	Expression
Triad contiguous access	0	1	0	2	y(i) = x1(i) + scalar * x2(i)

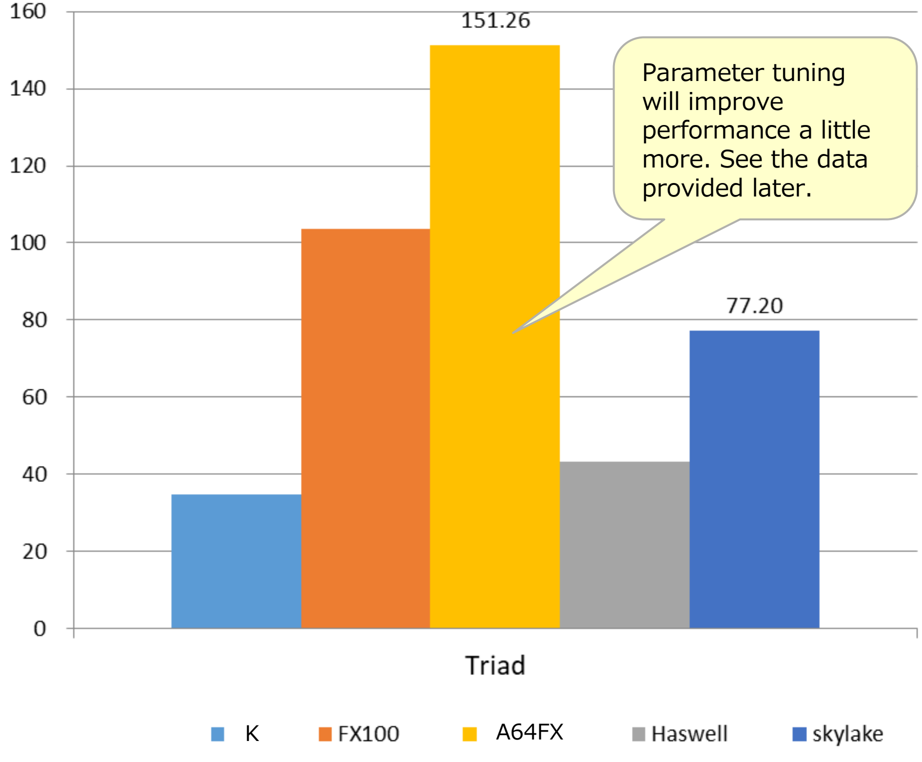
* The value shown here (memory throughput) does not include that of the data read from those cache lines to which data is written.
(Same as the STREAM benchmark)

Double precision memory access performance
Ratio when the performance of the K is 1



Hardware prefetch
L1 prefetch: 8 lines ahead
L2 prefetch: 40 lines ahead

Double precision memory access performance
Throughput (GB/sec)



Parameter tuning will improve performance a little more. See the data provided later.

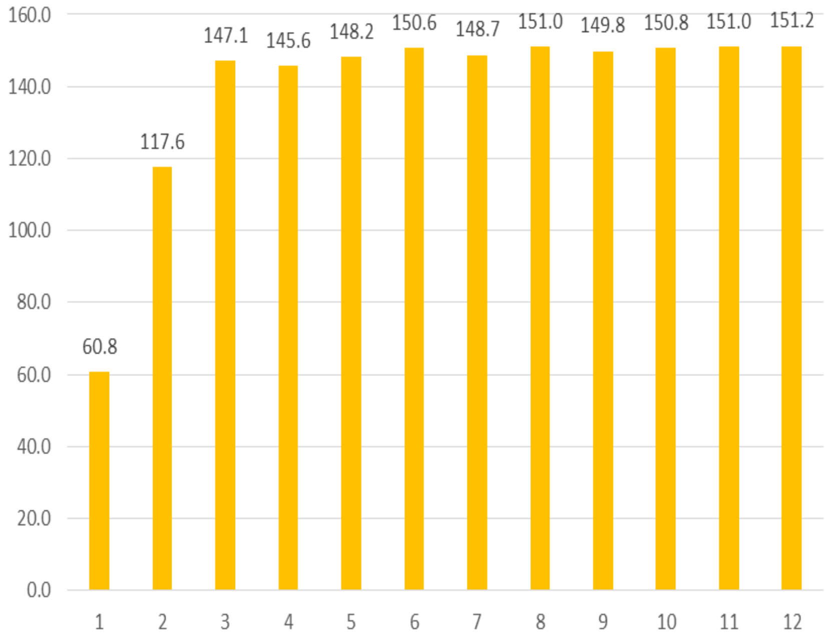
Memory Access: Without zfill (2/3)

- Number of executed cores and throughput (hardware prefetch)

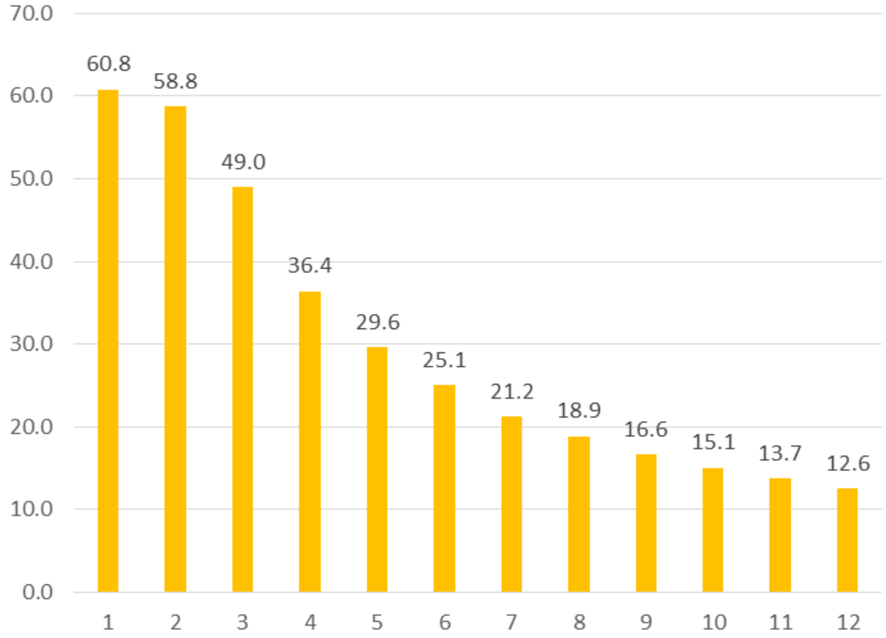
* The value shown here (memory throughput) does not include that of the data read from those cache lines to which data is written.
(Same as the STREAM benchmark)

Using CMG0, 1 to 12 threads were executed.

Throughput (GB/sec)



Throughput/number of cores (GB/sec)

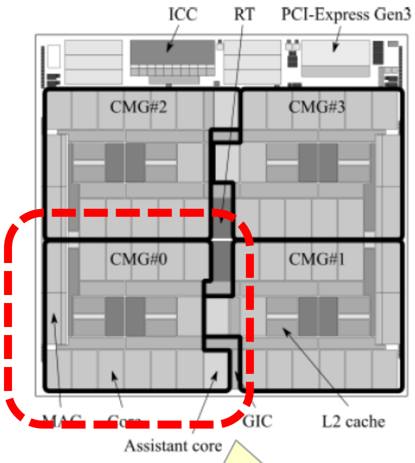


Almost all the performance of the CMG can be delivered through 3 threads (3 core) execution.

Memory Access: Without zfill (3/3)

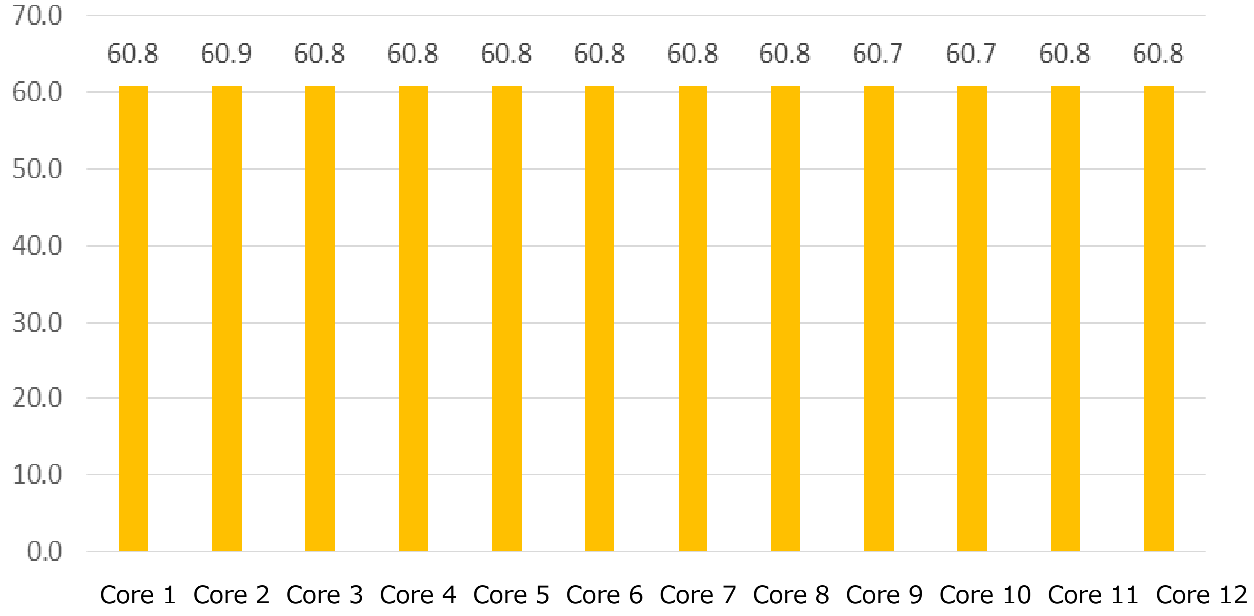
- Difference in throughput due to the executed core (hardware prefetch)

* The value shown here (memory throughput) does not include that of the data read from those cache lines to which data is written.
(Same as the STREAM benchmark)



Using cores 1 to 12 of CMG0, a thread was executed per core.

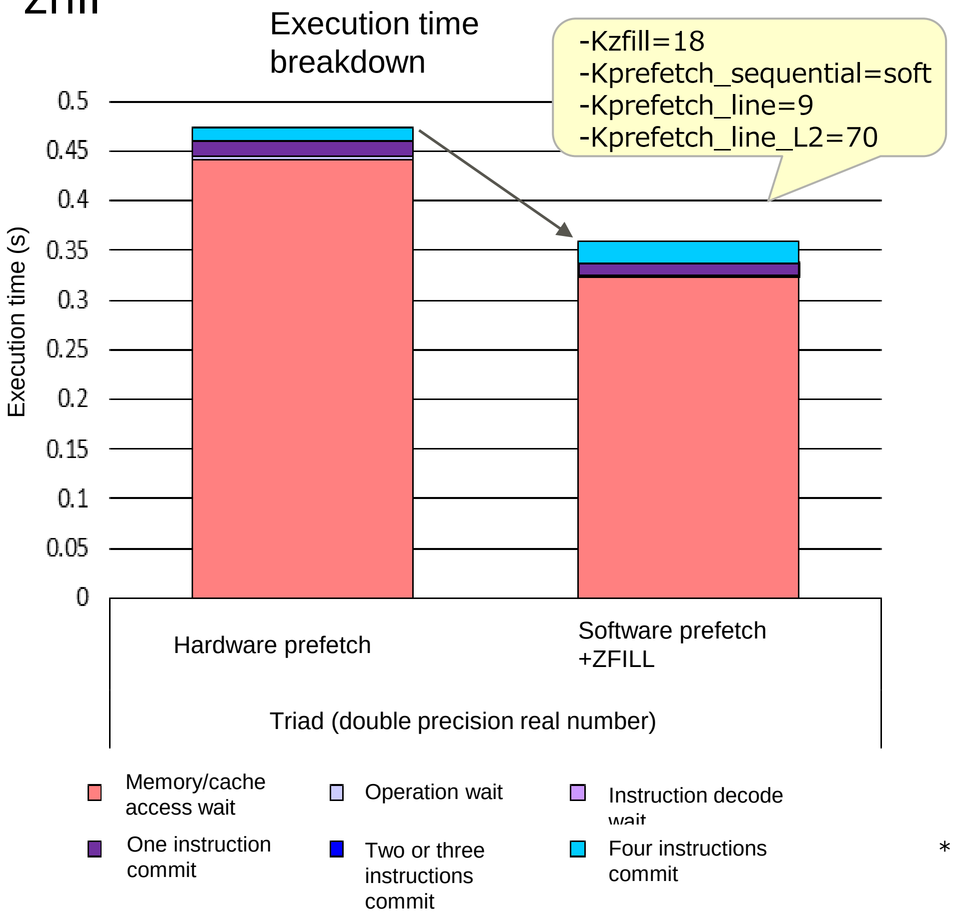
Throughput (GB/sec)



No significant difference in throughput due to the executed core was observed.

[Reference] Performance Improvement by zfill

Performance of software prefetch and zfill



-Kzfill=18
-Kprefetch_sequential=soft
-Kprefetch_line=9
-Kprefetch_line_L2=70

PA values

	Hardware prefetch	Software prefetch + zfill
Aggregation thread number	0	0
Execution time [s]	0.474	0.359
Total number of effective instructions	2.16E+09	2.52E+09
GFLOPS	13.27	17.54
Memory throughput [GB/s]	**222.5**	210.8
L1 busy rate/thread	52.9%	55.09%
L2 busy rate	85.1%	94.39%
Memory busy rate	21.7%	20.58%
Floating-point pipeline busy rate/thread	FLA:4.90% FLB:2.80%	FLA:7.14% FLB:3.01%
L1 miss count/thread	2.46E+07	2.46E+07
L1 miss demand rate/thread	9.40%	0.08%
L2 miss count/thread	**2.62E+07**	1.64E+07
L2 miss demand rate/thread	**12.21%**	0.31%

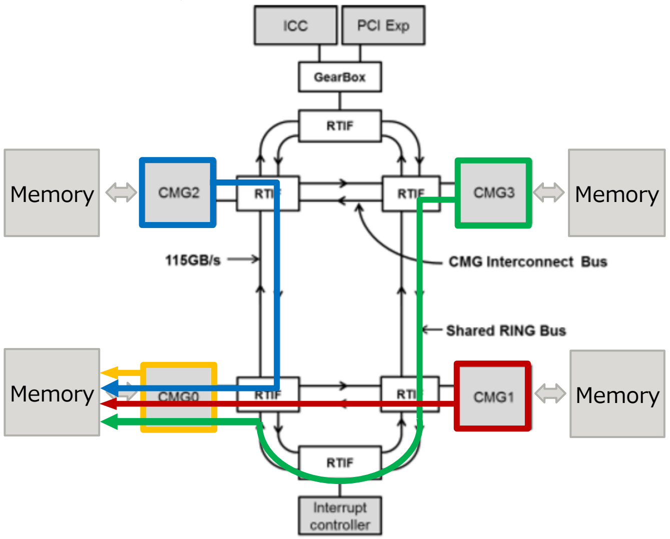
* The evaluation was conducted with the access size (number of innermost loop iterations, array size) increased to 240 MB (by 10 times).

Performing zfill properly improves the performance up to 210.8 GB/s.

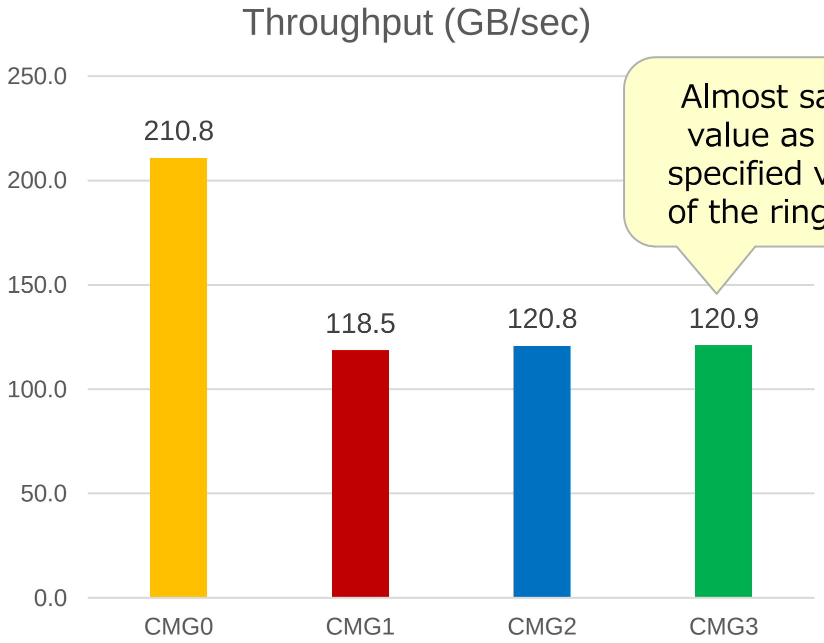
Memory Access: Access Across CMGs

Memory access across CMGs

12 threads executed (1 CMG)
numactl –m4
Access to the CMG0 memory



zfill-tuned program used



Almost same value as the specified value of the ring bus

While the performance almost the same as the specified value is achieved, the transfer rate for memory access across CMGs is about half as high.
Care needs to be exercised when performing parallel processing (OpenMP or MPI).

Performance Values by Data Type

- Contiguous Access
- Gather Load / Scatter Store Access

Contiguous Access

- Evaluation Conditions
- Evaluation Results: L1 Cache Throughput
- Evaluation Results: L2 Cache Throughput
- Evaluation Results: Memory Throughput (With/Without zfill)

Contiguous Access Evaluation Conditions

- Throughput evaluation conditions
  - Evaluated access areas
    L1 cache, L2 cache, memory (with/without zfill)
  - Type patterns
    - Real type: Double precision real number, single precision real number, half precision real number (FP16)
    - Integer type: 8-byte integer, 4-byte integer, 2-byte integer, 1-byte integer
  - Data access pattern
    - Contiguous SIMD load 2 + contiguous SIMD store 1
  - The evaluation code is shown at the right.
  - For other details, see below.

Half precision real numbers were evaluated using an assembler manually modified based on the 2-byte integer pattern.

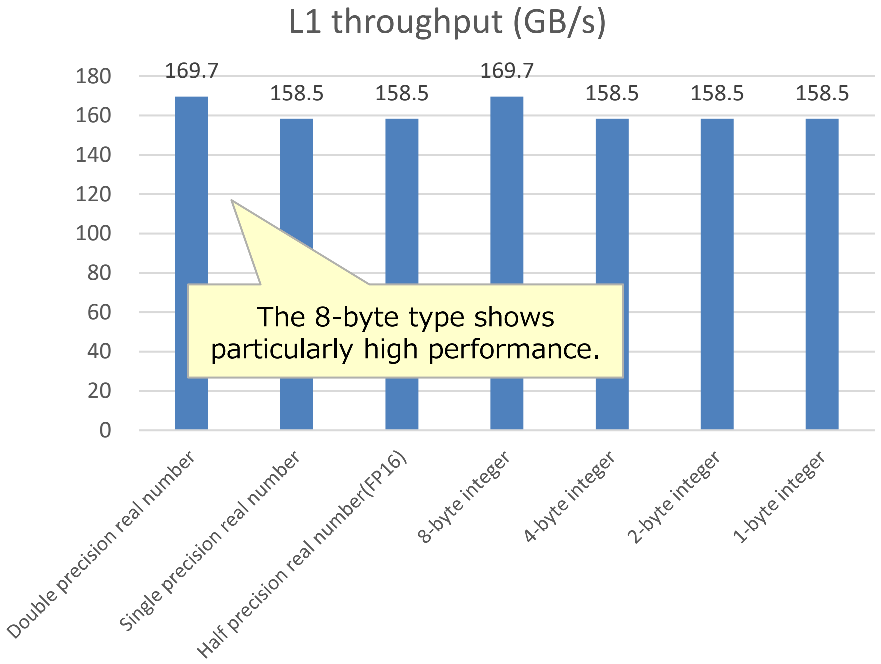
Evaluation code

```
do k  = 1, iter
   do i = 1,n
      y(i) = x1(i) + c * x2(i)
   enddo
enddo
```

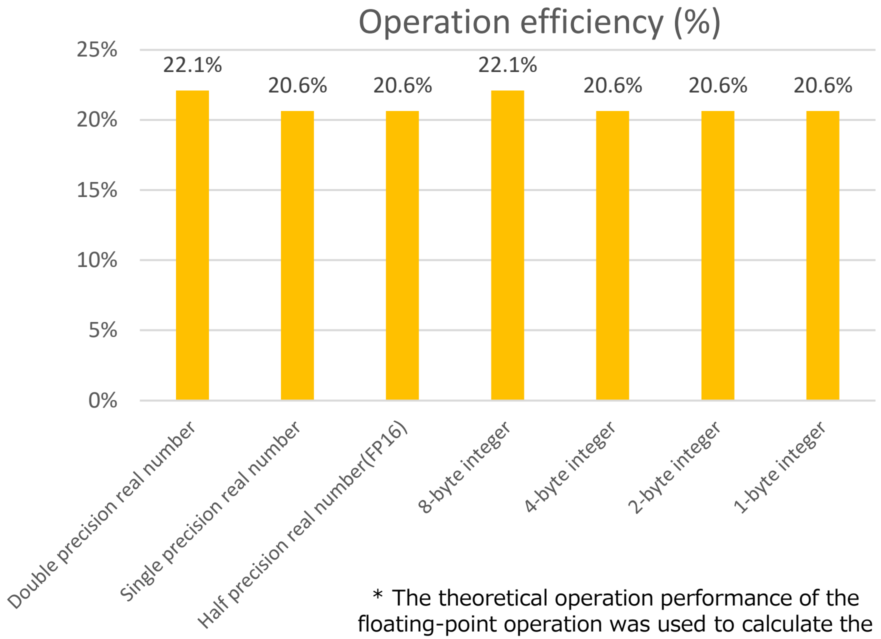
Evaluated access areas	Number of executed cores	n (array size/innermost loop)	iter (outer loop)
L1 cache	1	1/2 of the L1 cache size	1000000
L2 cache	12	1/2 of the L2 cache size	10000
Memory	12	30 times the L2 cache size	300

Contiguous Access Evaluation Results: L1 Cache Throughput

The specified values of the L1 cache throughput are:
Read1: 256 GB/s,
Write1: 128 GB/s,
Read2, Write1: 192 GB/s



The 8-byte type shows particularly high performance.



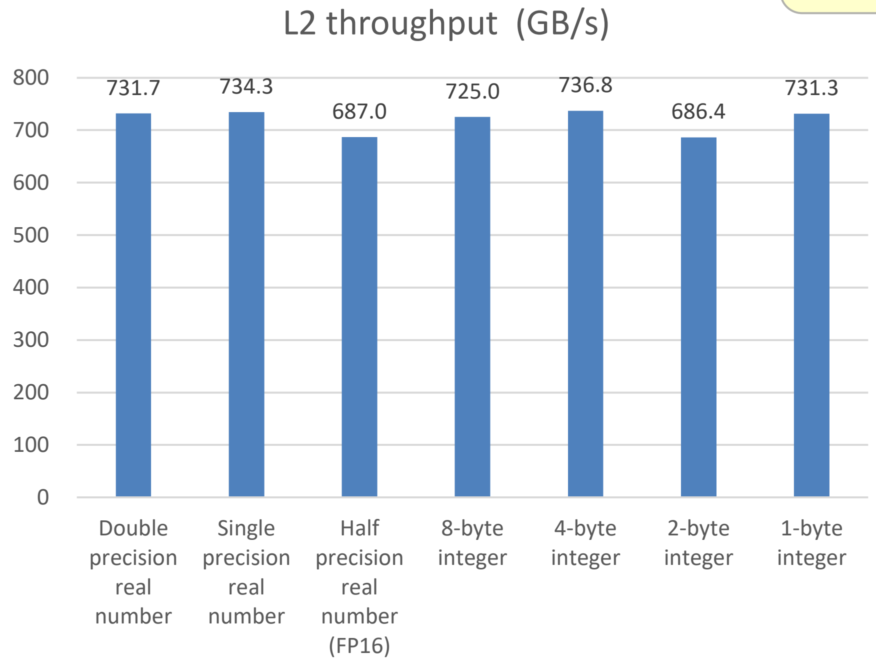
* The theoretical operation performance of the floating-point operation was used to calculate the operation efficiency of the integer operation.

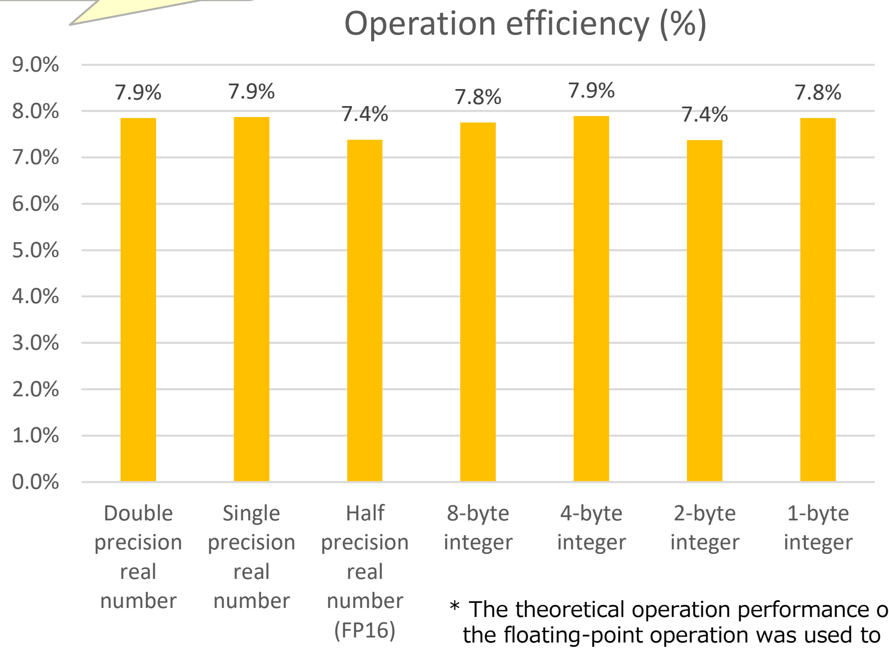
Type	Throughput (GB/s)	GFLOPS /GOPS	Operation performance ratio	L1 busy rate
Double precision real number	169.7	14.1	0.54	96.7%
Single precision real number	158.5	26.4	1.00	96.2%
Half precision real number (FP16)	158.5	52.8	2.00	97.4%
8-byte integer	169.7	14.1	0.54	96.7%
4-byte integer	158.5	26.4	1.00	97.4%
2-byte integer	158.5	52.8	2.00	97.4%
1-byte integer	158.5	105.7	4.00	97.4%

Contiguous Access Evaluation Results: L2 Cache Throughput

-Kprefetch_sequential=soft
-Kprefetch_cache_level=1
-Kprefetch_line=4

The specified values of the L2 cache throughput are:
Read1: 1024 GB/s,
Write1: 512 GB/s,
Read2, Write1: 768 GB/s



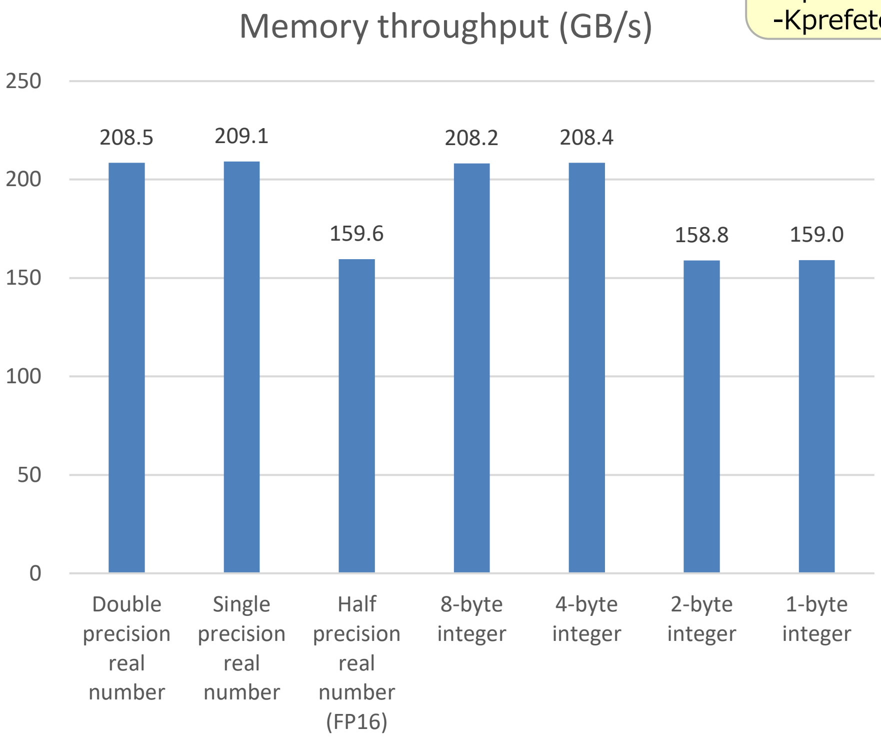


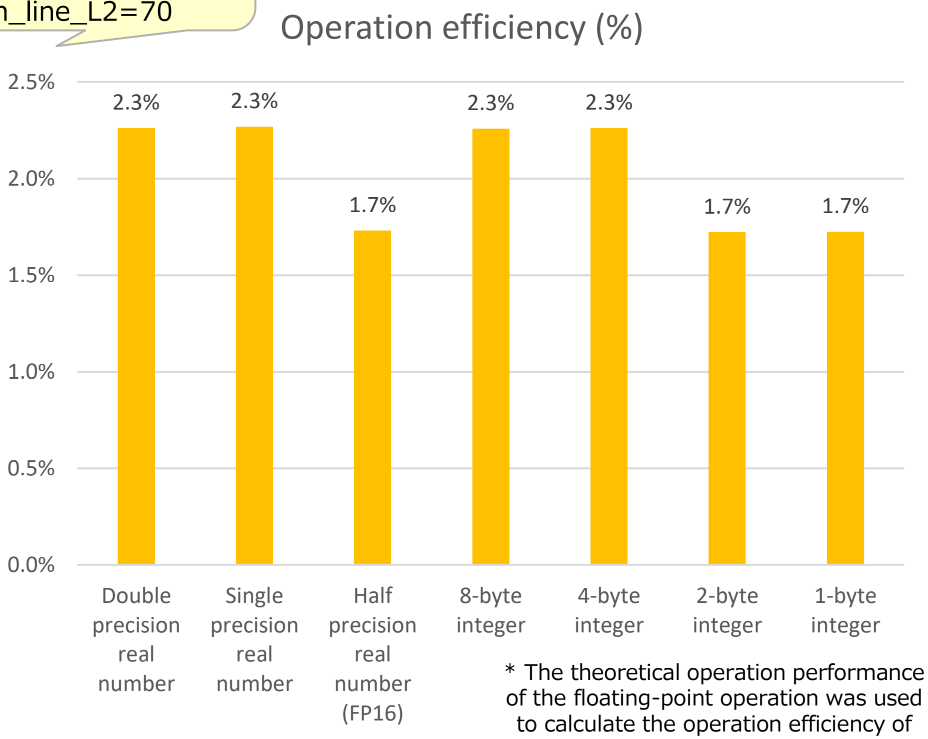
* The theoretical operation performance of the floating-point operation was used to calculate the operation efficiency of the integer operation.

Type	Throughput (GB/s)	GFLOPS /GOPS	L2 busy rate	L1D miss count
Double precision real number	731.7	60.3	96.5%	1.66E+08
Single precision real number	734.3	120.9	97.0%	1.66E+08
Half precision real number (FP16)	687.0	226.7	90.5%	1.65E+08
8-byte integer	725.0	59.6	96.5%	1.66E+08
4-byte integer	736.8	121.2	96.9%	1.66E+08
2-byte integer	686.4	226.5	90.5%	1.65E+08
1-byte integer	731.3	482.2	96.5%	1.66E+08

Contiguous Access Evaluation Results: Memory Throughput (With zfill)

-Kzfill=18
-Kprefetch_sequential=soft
-Kprefetch_line=9
-Kprefetch_line_L2=70



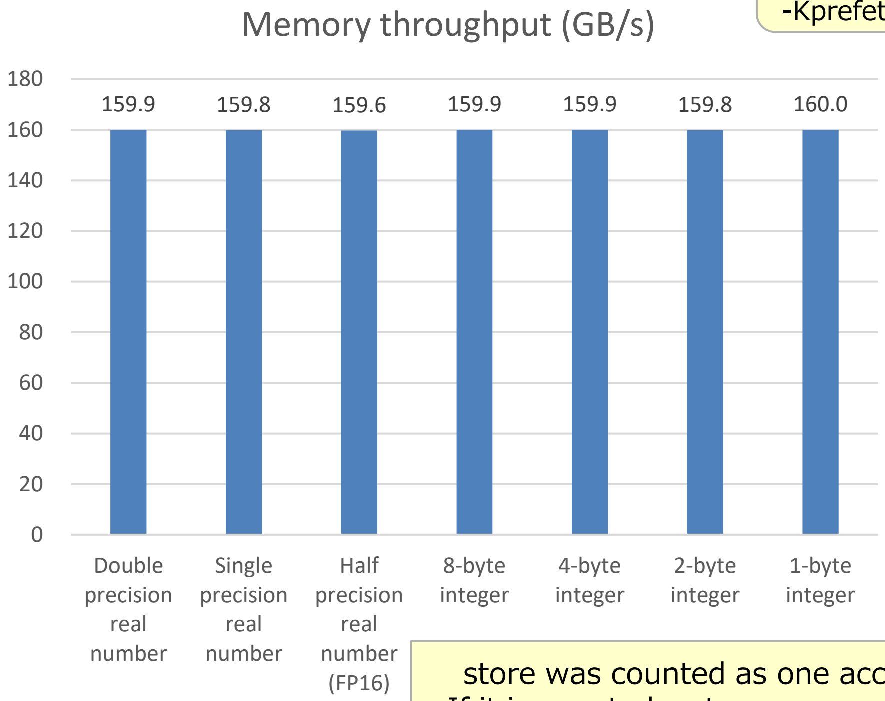


＊ The theoretical operation performance of the floating-point operation was used to calculate the operation efficiency of the integer operation.

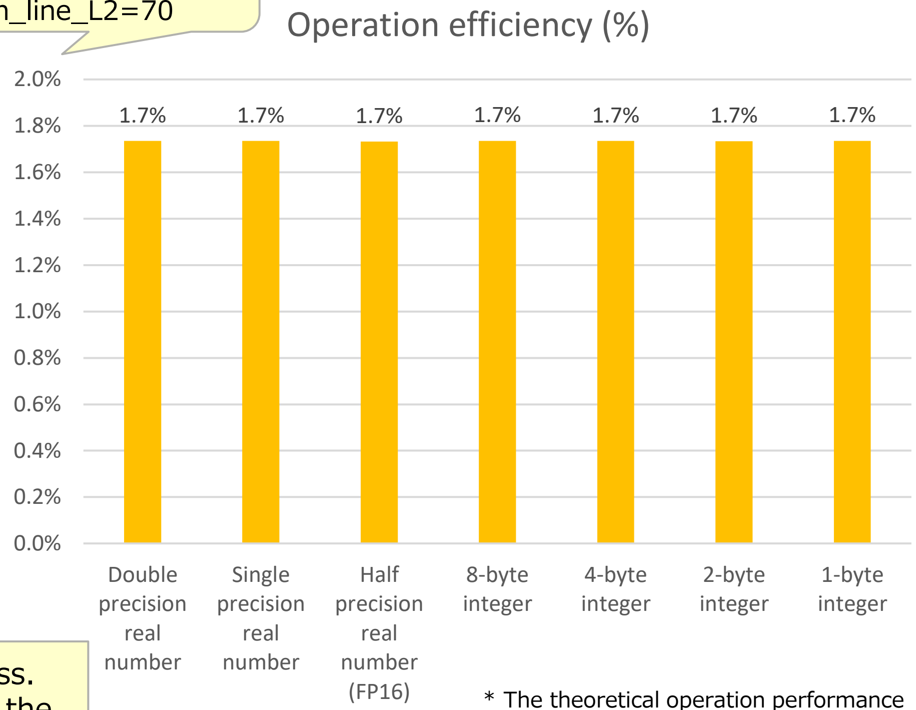
Type	Throughput (GB/s)	GFLOPS /GOPS	L2 busy rate	Memory busy rate	L1D miss count	L2 miss count
Double precision real number	208.5	17.4	93.7%	81.7%	2.95E+08	1.97E+08
Single precision real number	209.1	34.8	93.2%	81.9%	2.95E+08	1.98E+08
Half precision real number (FP16)	159.6	53.2	84.8%	83.3%	2.95E+08	2.96E+08
8-byte integer	208.2	17.3	93.7%	81.6%	2.95E+08	1.98E+08
4-byte integer	208.4	34.7	93.2%	81.7%	2.95E+08	1.97E+08
2-byte integer	158.8	52.9	84.8%	83.0%	2.95E+08	2.96E+08
1-byte integer	159.0	106.0	84.8%	83.0%	2.95E+08	2.96E+08

Contiguous Access Evaluation Results: Memory Throughput (Without zfill)

-Kprefetch_sequential=soft
-Kprefetch_line=9
-Kprefetch_line_L2=70



store was counted as one access.
If it is counted as two accesses, the throughput becomes about 213 GB/s.



* The theoretical operation performance of the floating-point operation was used to calculate the operation efficiency of the integer operation.

Type	Throughput (GB/s)	GFLOPS /GOPS	L2 busy rate	Memory busy rate	L1D miss count	L2 miss count
Double precision real number	159.9	13.3	84.4%	83.5%	2.95E+08	2.96E+08
Single precision real number	159.8	26.6	84.6%	83.5%	2.95E+08	2.96E+08
Half precision real number (FP16)	159.6	53.2	84.7%	83.4%	2.95E+08	2.96E+08
8-byte integer	159.9	13.3	84.5%	83.5%	2.95E+08	2.96E+08
4-byte integer	159.9	26.7	84.5%	83.5%	2.95E+08	2.96E+08
2-byte integer	159.8	53.3	84.4%	83.5%	2.95E+08	2.96E+08
1-byte integer	160.0	106.6	84.6%	83.5%	2.95E+08	2.96E+08

Gather Load / Scatter Store Access

- ISA of Gather Load / Scatter Store
- Evaluation Conditions
- Evaluation Results: L1 Cache Throughput
- Evaluation Results: L2 Cache Throughput
- Evaluation Results: Memory Throughput (With/Without zfill)

ISA of Gather Load / Scatter Store

- 4-byte integer type and 8-byte integer type
  The instructions were excluded from the evaluation because they were the same as those for real numbers (single precision and double precision).
  - 4-byte integer, single precision real number (example: Gather load instruction)

```
ld1w    {z1.s}, p0/z, [x11, z0.s, sxtw]
```

  - 8-byte integer, double precision real number (example: Gather load instruction)

```
ld1d    {z1.d},  p1/z,  [x11,  z0.d]
```

- 1-byte integer, 2-byte integer, half precision real number (FP16)
  SIMD conversion using all of the 512 bytes (1 byte = 64 SIMD/2 bytes = 32 SIMD) cannot be performed due to the instruction specifications of 1-byte integers, 2-byte integers, and half precision real numbers.
  Currently, up to 8 SIMD conversion is supported (16 SIMD to be supported in the future).

```
ld1sh   {z1.d}, p1/z, [x11, z0.d]
```

The index is specified.
This specification cannot
implement 32 SIMD or 64 SIMD.

Gather Load / Scatter Store Access Evaluation Conditions

- Throughput evaluation conditions
  - Evaluated access areas

    L1 cache, L2 cache, memory (with/without zfill)
  - Type patterns
    - Real type: Double precision real number, single precision real number
  - Data access pattern
    - Contiguous SIMD load 2 + contiguous SIMD store 1
    - Gather load 2 + contiguous SIMD store 1
    - Contiguous SIMD load 2 + Scatter store 1
    - Gather load 2 + Scatter store 1
  - The evaluation code is shown at the right.

    (Evaluation code 2 is Gather load + Scatter store.)
  - For other details, see below.

Evaluation code example 1

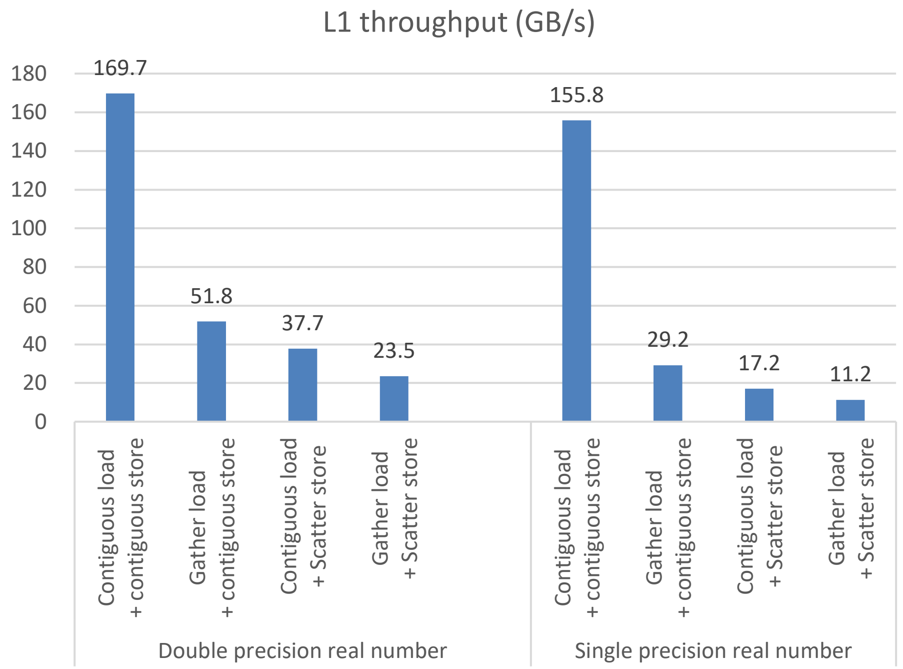
```
do k  = 1, iter
   do i = 1,n
      y(i) = x1(i) + c * x2(i)
   enddo
enddo
```

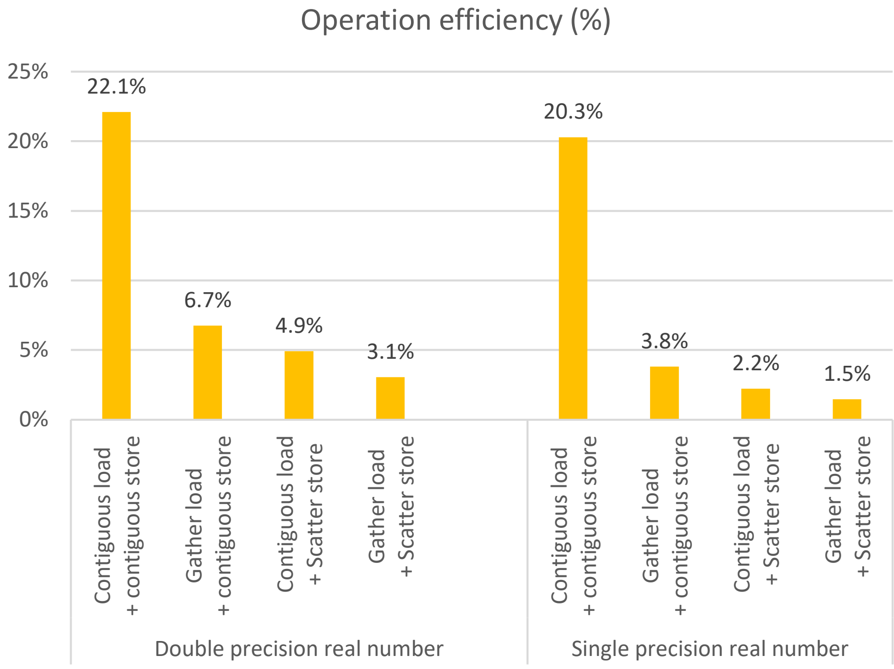
Evaluation code example 2

```
do k  = 1, iter
   do i = 1,n
      y(1,i) = x1(1,i) + c * x2(1,i)
   enddo
enddo
```

Evaluated access areas	Number of executed cores	n (array size/innermost loop)	iter (outer loop)
L1 cache	1	1/2 of the L1 cache size	1000000
L2 cache	12	1/2 of the L2 cache size	10000
Memory	12	30 times the L2 cache size	300

Evaluation Results: L1 Cache Throughput

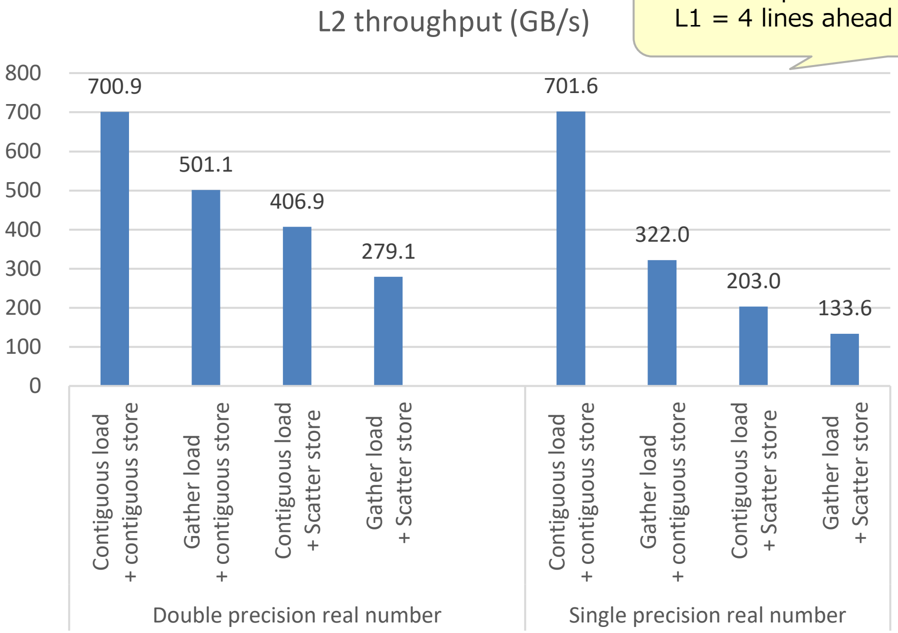




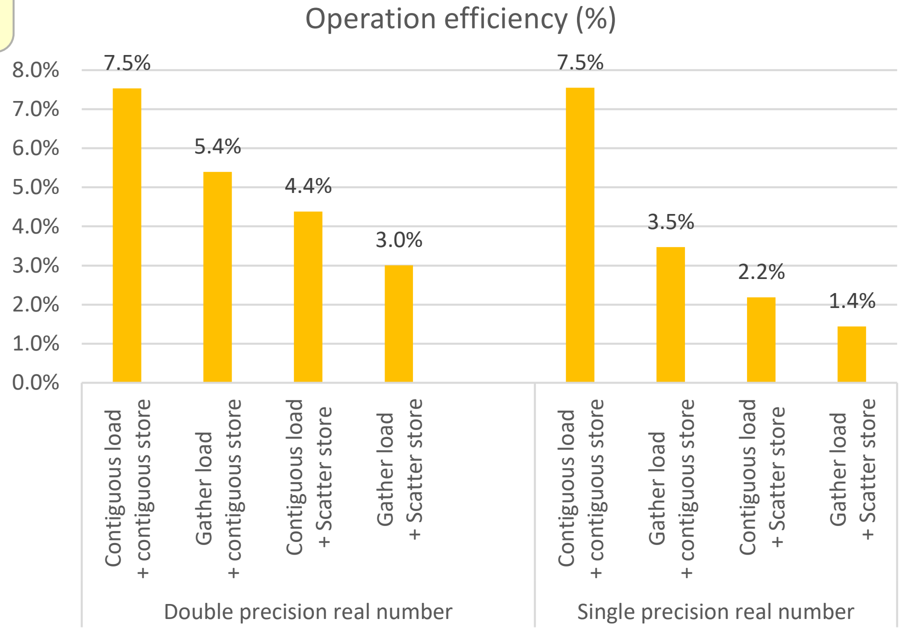
Type	Load / store	Throughput (GB/s)	GFLOPS /GOPS	Operation performance ratio	L1 busy rate
Double precision real number	Contiguous load × 2 + contiguous store × 1	169.7	14.1	1.00	96.7%
	Gather load × 2 + contiguous store × 1	51.8	4.3	0.31	74.4%
	Contiguous load × 2 + Scatter store × 1	37.7	3.1	0.22	59.3%
	Gather load × 2 + Scatter store × 1	23.5	2.0	0.14	55.0%
Single precision real number	Contiguous load × 2 + contiguous store × 1	155.8	26.0	1.00	96.2%
	Gather load × 2 + contiguous store × 1	29.2	4.9	0.19	73.9%
	Contiguous load × 2 + Scatter store × 1	17.2	2.9	0.11	46.0%
	Gather load × 2 + Scatter store × 1	11.2	1.9	0.07	49.8%

In cases including Gather / Scatter, there is no L1 throughput bottleneck.

Evaluation Results: L2 Cache Throughput



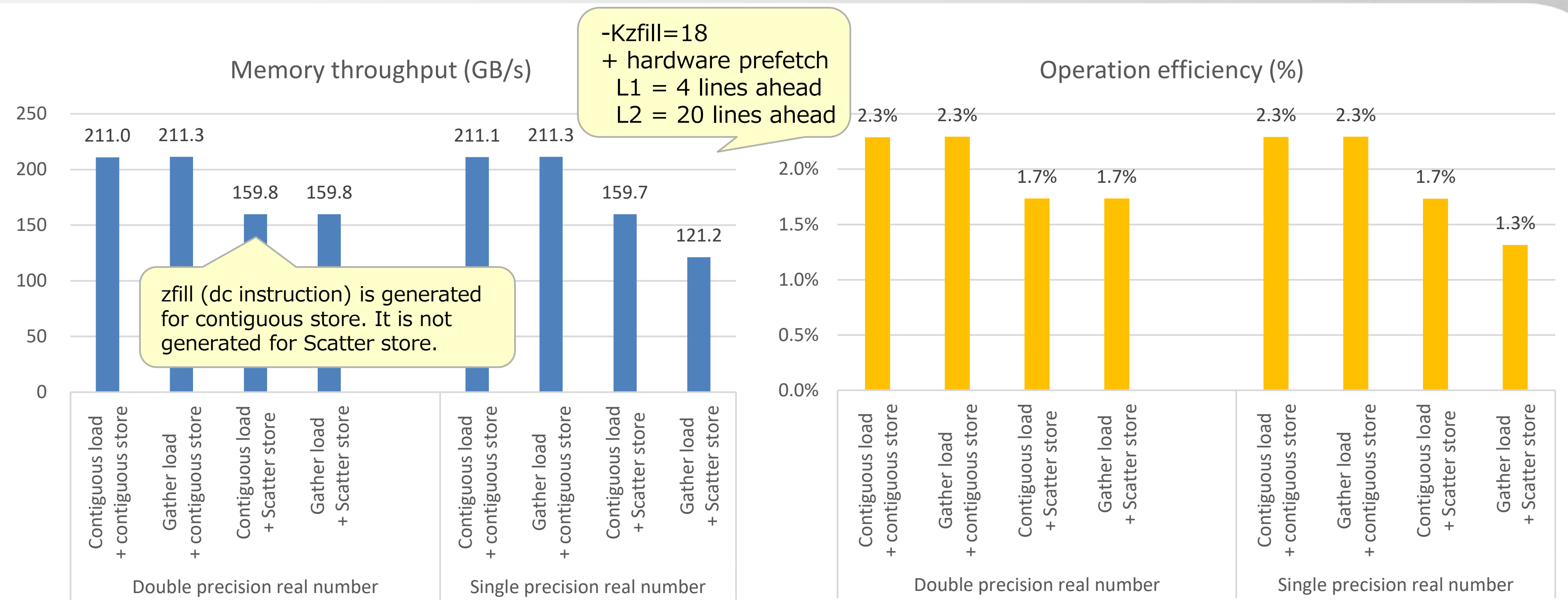
Hardware prefetch
L1 = 4 lines ahead



Type	Load / store	Throughput (GB/s)	GFLOPS /GOPS	L2 busy rate	L1D miss count
Double precision real number	Contiguous load × 2 + contiguous store × 1	700.9	57.9	97.6%	1.65E+08
	Gather load × 2 + contiguous store × 1	501.1	41.5	71.6%	1.65E+08
	Contiguous load × 2 + Scatter store × 1	406.9	33.7	59.0%	1.65E+08
	Gather load × 2 + Scatter store × 1	279.1	23.1	40.4%	1.65E+08
Single precision real number	Contiguous load × 2 + contiguous store × 1	701.6	115.9	97.6%	1.65E+08
	Gather load × 2 + contiguous store × 1	322.0	53.3	45.7%	1.65E+08
	Contiguous load × 2 + Scatter store × 1	203.0	33.6	29.2%	1.65E+08
	Gather load × 2 + Scatter store × 1	133.6	22.1	20.4%	1.65E+08

In cases including Gather / Scatter, there is no L2 throughput bottleneck.

Evaluation Results: Memory Throughput (With zfill)



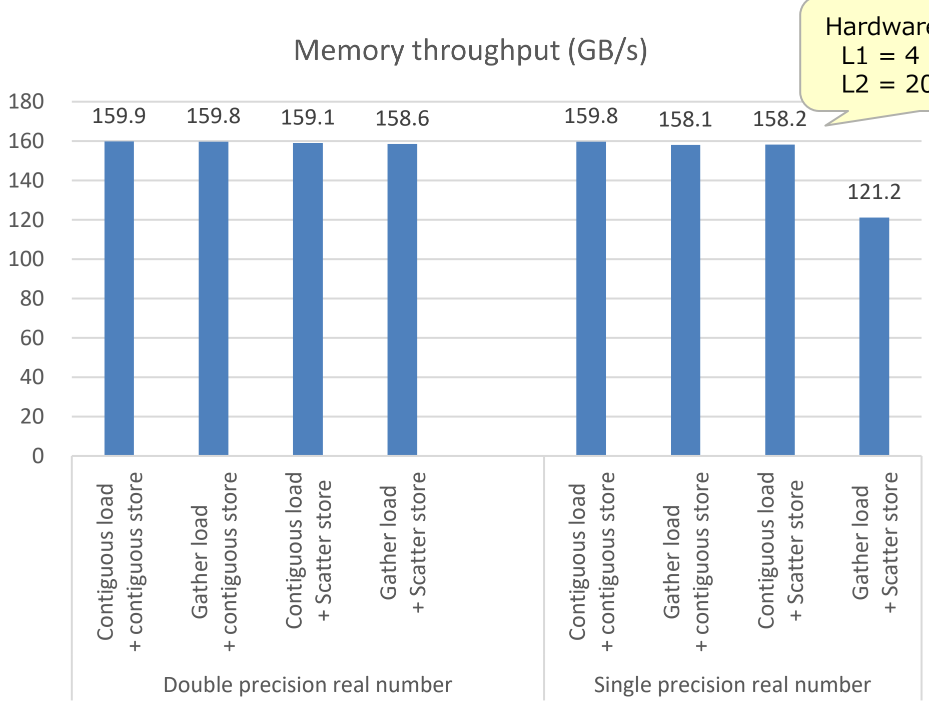
-Kzfill=18
+ hardware prefetch
L1 = 4 lines ahead
L2 = 20 lines ahead

zfill (dc instruction) is generated for contiguous store. It is not generated for Scatter store.

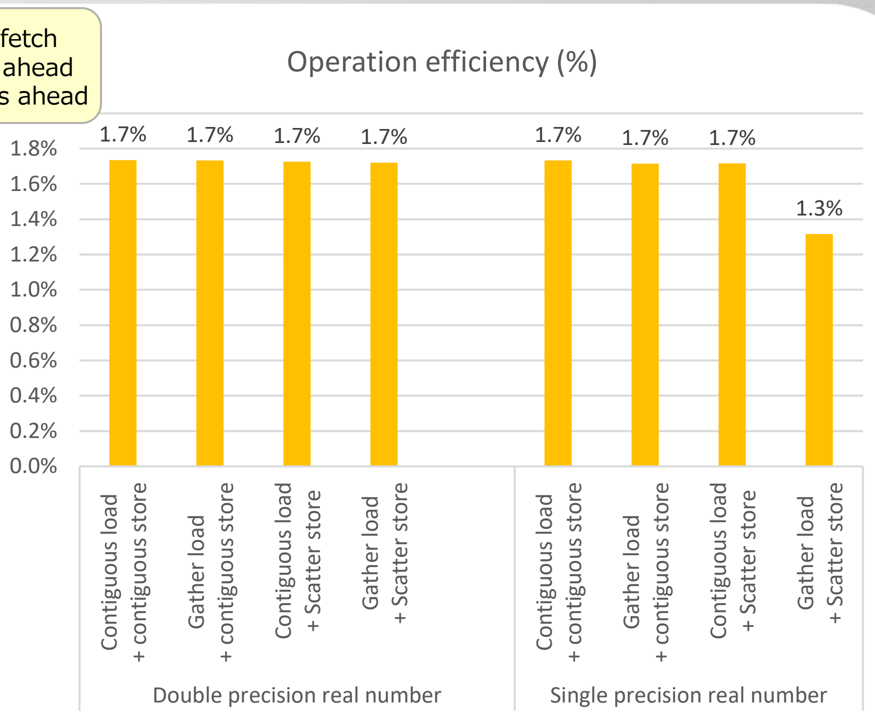
Type	Load / store	Throughput (GB/s)	GFLOPS /GOPS	L2 busy rate	Memory busy rate	L1D miss count	L2 miss count
Double precision Real number	Contiguous load × 2 + contiguous store × 1	211.0	17.6	94.1%	99.9%	2.95E+08	2.60E+08
	Gather load × 2 + contiguous store × 1	211.3	17.6	94.4%	95.6%	2.95E+08	2.43E+08
	Contiguous load × 2 + Scatter store × 1	159.8	13.3	85.1%	87.0%	2.95E+08	3.12E+08
	Gather load × 2 + Scatter store × 1	159.8	13.3	84.5%	86.8%	2.95E+08	3.12E+08
Single precision Real number	Contiguous load × 2 + contiguous store × 1	211.1	35.2	93.9%	99.8%	2.95E+08	2.59E+08
	Gather load × 2 + contiguous store × 1	211.3	35.2	94.0%	92.4%	2.95E+08	2.32E+08
	Contiguous load × 2 + Scatter store × 1	159.7	26.6	84.8%	86.0%	2.95E+08	3.08E+08
	Gather load × 2 + Scatter store × 1	121.2	20.2	48.4%	63.3%	2.95E+08	2.96E+08

Double precision real numbers create a memory throughput bottleneck.

Evaluation Results: Memory Throughput (Without zfill)


Memory throughput (GB/s)

Hardware prefetch
L1 = 4 lines ahead
L2 = 20 lines ahead


Operation efficiency (%)

Type	Load / store	Throughput (GB/s)	GFLOPS /GOPS	L2 busy rate	Memory busy rate	L1D miss count	L2 miss count
Double precision Real number	Contiguous load × 2 + contiguous store × 1	159.9	13.3	86.0%	95.8%	2.95E+08	3.54E+08
	Gather load × 2 + contiguous store × 1	159.8	13.3	85.2%	87.4%	2.95E+08	3.15E+08
	Contiguous load × 2 + Scatter store × 1	159.1	13.3	85.3%	86.5%	2.95E+08	3.12E+08
	Gather load × 2 + Scatter store × 1	158.6	13.2	84.5%	86.1%	2.95E+08	3.12E+08
Single precision Real number	Contiguous load × 2 + contiguous store × 1	159.8	26.6	85.9%	95.7%	2.95E+08	3.54E+08
	Gather load × 2 + contiguous store × 1	158.1	26.4	84.6%	85.7%	2.95E+08	3.11E+08
	Contiguous load × 2 + Scatter store × 1	158.2	26.4	84.5%	85.1%	2.95E+08	3.08E+08
	Gather load × 2 + Scatter store × 1	121.2	20.2	48.4%	63.2%	2.95E+08	2.96E+08

Double precision real numbers create a memory throughput bottleneck in all access patterns.

Performance Impact by Alignment

- Measurement Conditions
- Measurement Results: Contiguous SIMD Load
- Measurement Results: Contiguous SIMD Store
- Measurement Results: Gather Load
- Measurement Results: Scatter Store
- Measurement Results: Structure Load (LD2 Instruction)
- Measurement Results: Structure Store (ST2 Instruction)

Measurement Conditions

- Evaluation overview

  Impact by alignment changes is evaluated using the following data access operations.

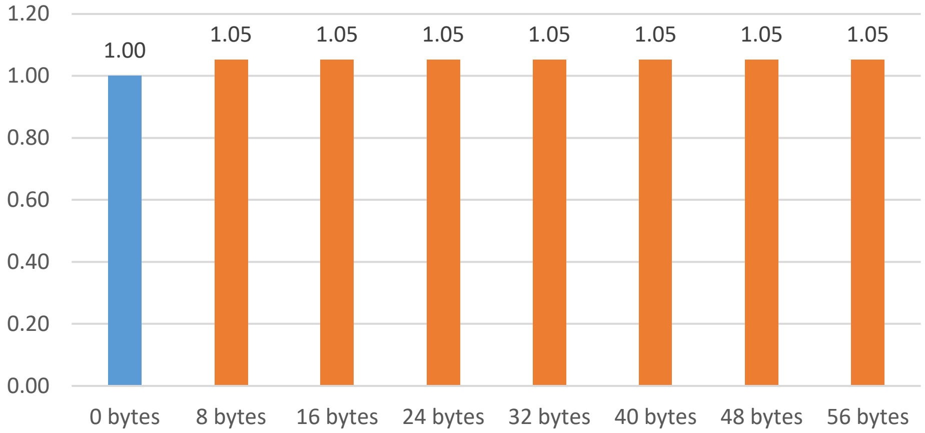
  - Contiguous SIMD load
  - Contiguous SIMD store

- Evaluation conditions
  - The performance is evaluated by accessing a 256-byte aligned array from bytes 0 to 63.
  - The evaluation data types are as follows.
    8-byte type (double precision real number/8-byte integer), 4-byte type (single precision real number/4-byte integer), 2-byte type (half precision real number (FP16)/2-byte integer), 1-byte type (1-byte integer)
  - The evaluation code will be described later together with the evaluation results.
  - The innermost loop (n) is evaluated using the array size and the number of iterations for accessing 1/2 of the L1 cache size. The outer loop (iter) is evaluated using 1000000.
  - The evaluation is performed through 1 core execution (sequential execution).
  - When the Multiple Structures instruction is evaluated, SWPL and loop unrolling are suppressed (out-of-order scheduling only).

Measurement Results: Contiguous SIMD Load (1/2)

8-byte type (double precision real number/8-byte integer)

Contiguous SIMD load (8-byte type, execution time ratio)



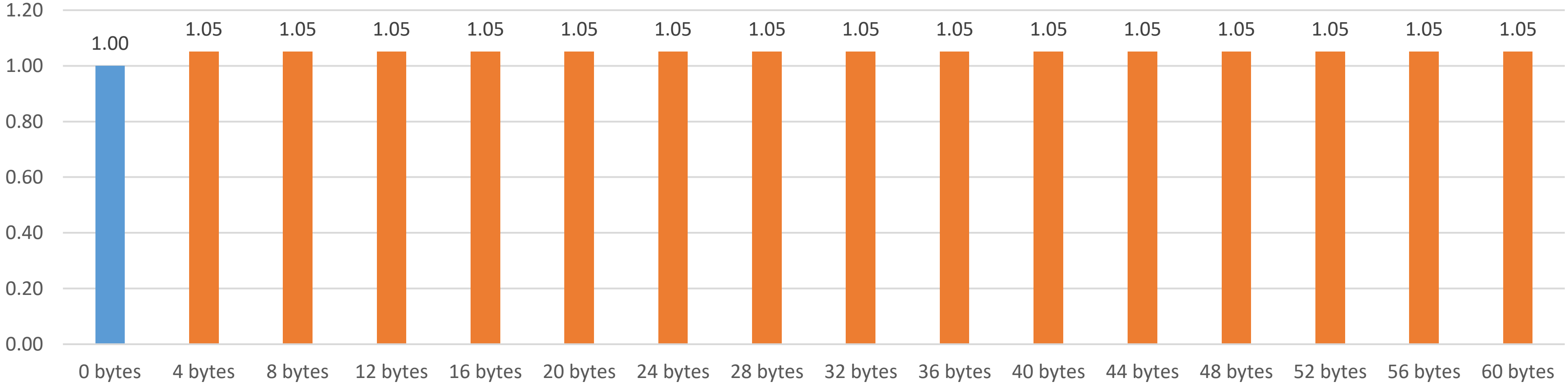
Evaluation code

```
do k  = 1, iter
   do i = 1,n
      y(i) = x1(i)
   enddo
enddo
```

Delete the store instruction from the assembly code.

4-byte type (single precision real number/4-byte integer)

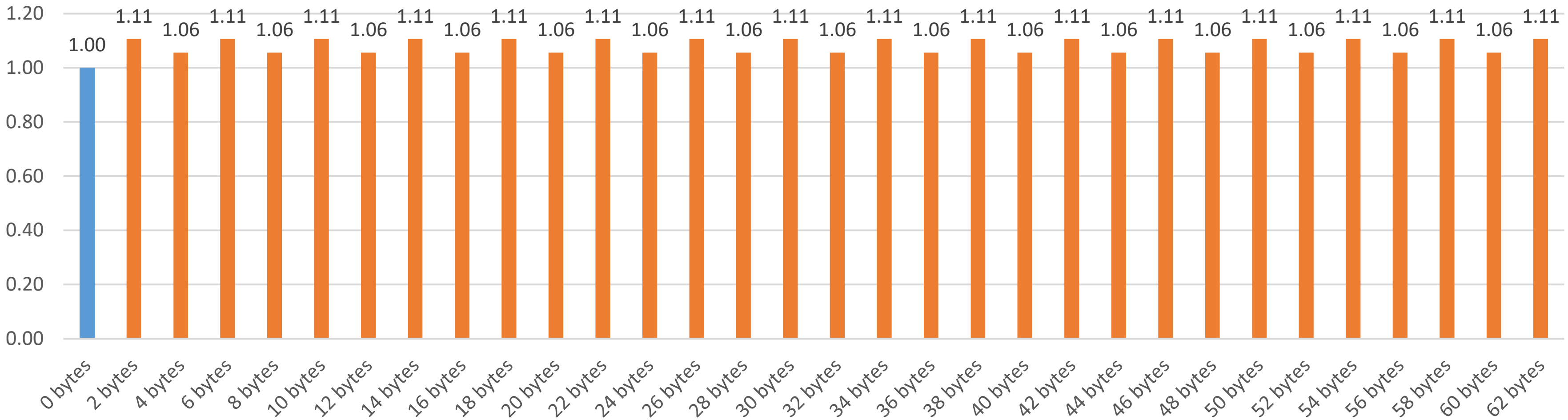
Contiguous SIMD load (4-byte type, execution time ratio)



Measurement Results: Contiguous SIMD Load (2/2)

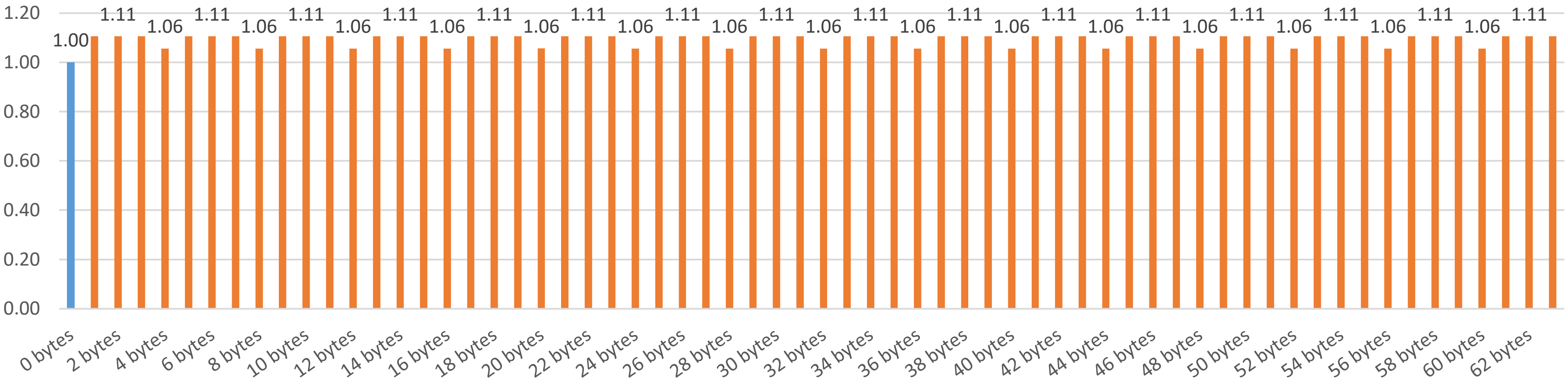
■ 2-byte type (half precision real number (FP16)/2-byte integer)

Contiguous SIMD load (2-byte type, execution time ratio)



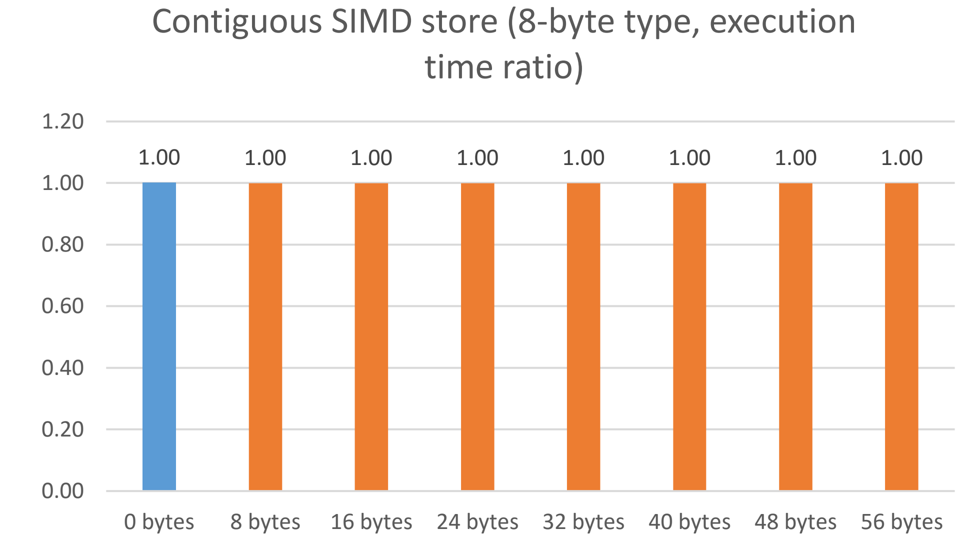
■ 1-byte type (1-byte integer)

Contiguous SIMD load (1-byte type, execution time ratio)



Measurement Results: Contiguous SIMD Store (1/2)

8-byte type (double precision real number/8-byte integer)

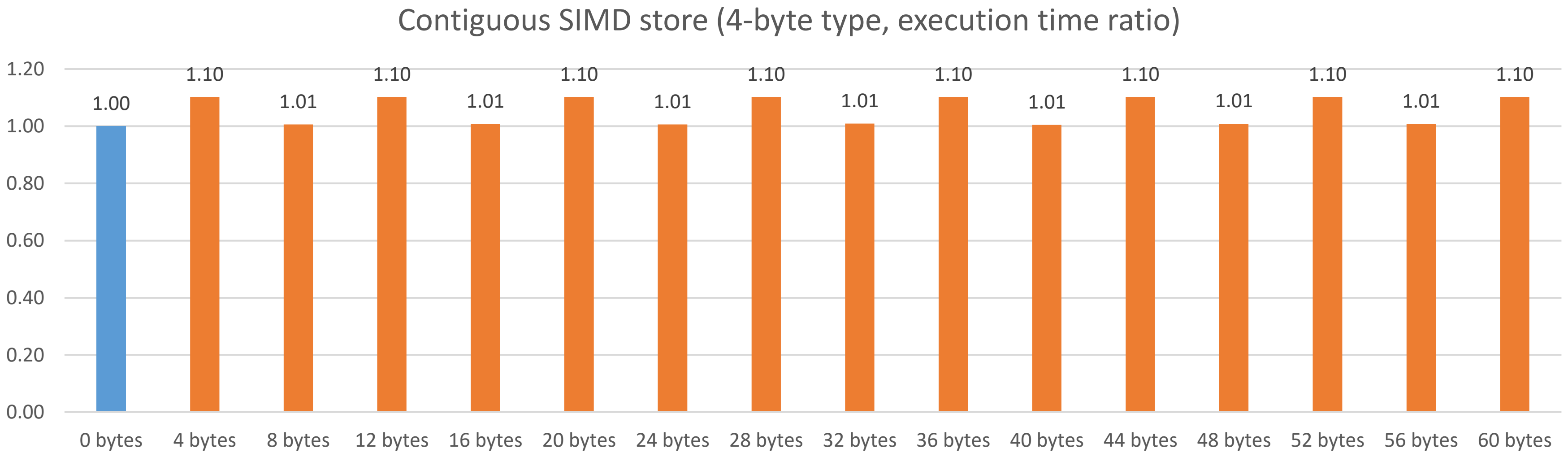


Evaluation code

```
do k  = 1, iter
   do i = 1,n
      y(i) = x1(i)
   enddo
enddo
```

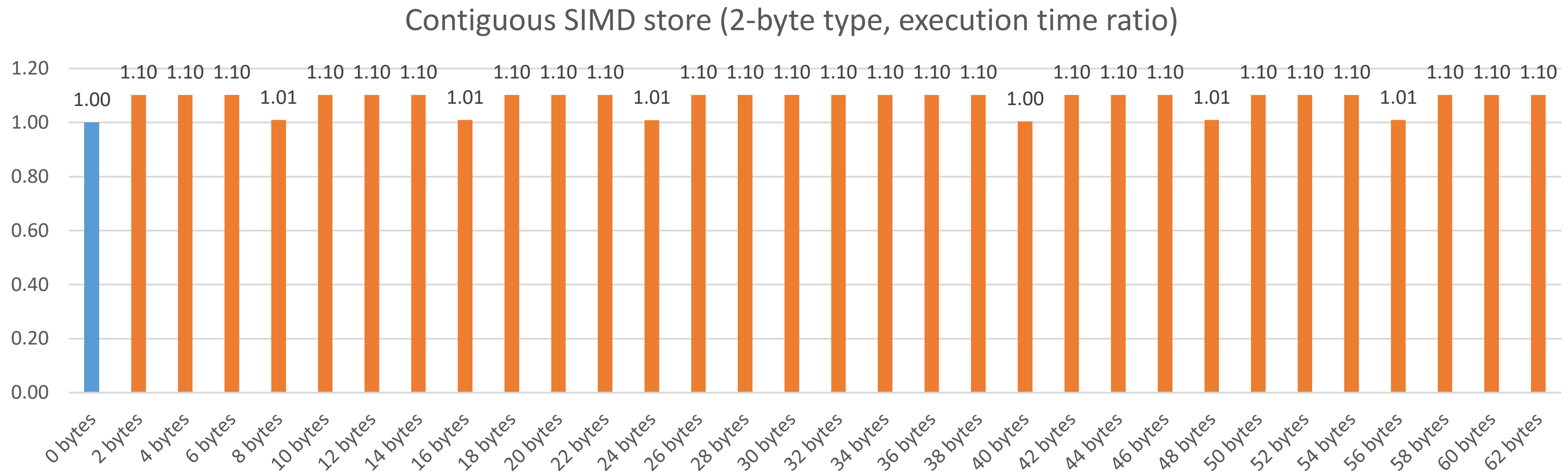
Delete the load instruction from the assembly code.

4-byte type (single precision real number/4-byte integer)

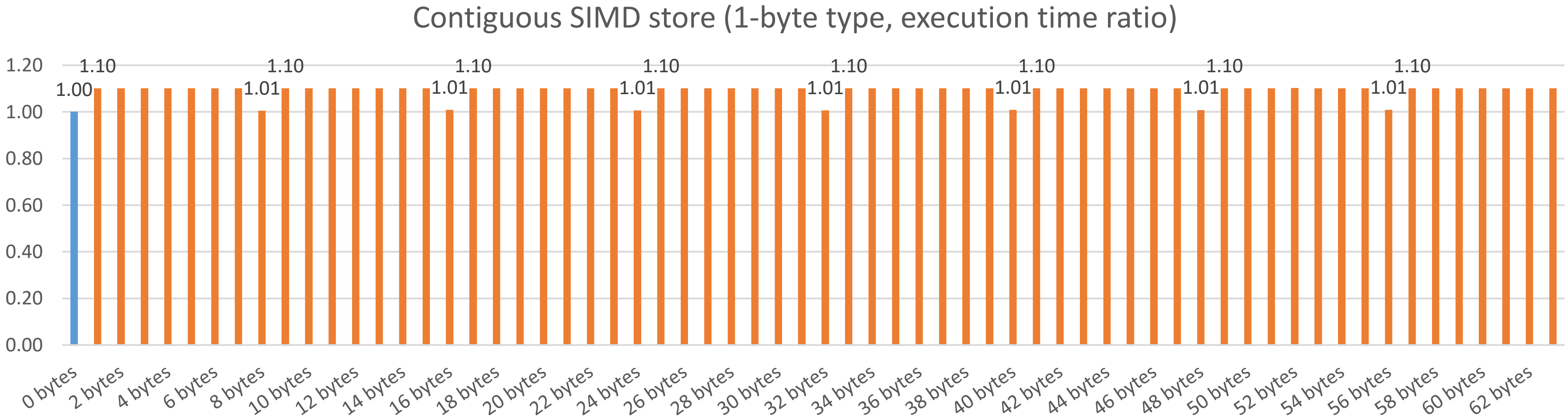


Measurement Results: Contiguous SIMD Store (2/2)

2-byte type (half precision real number)

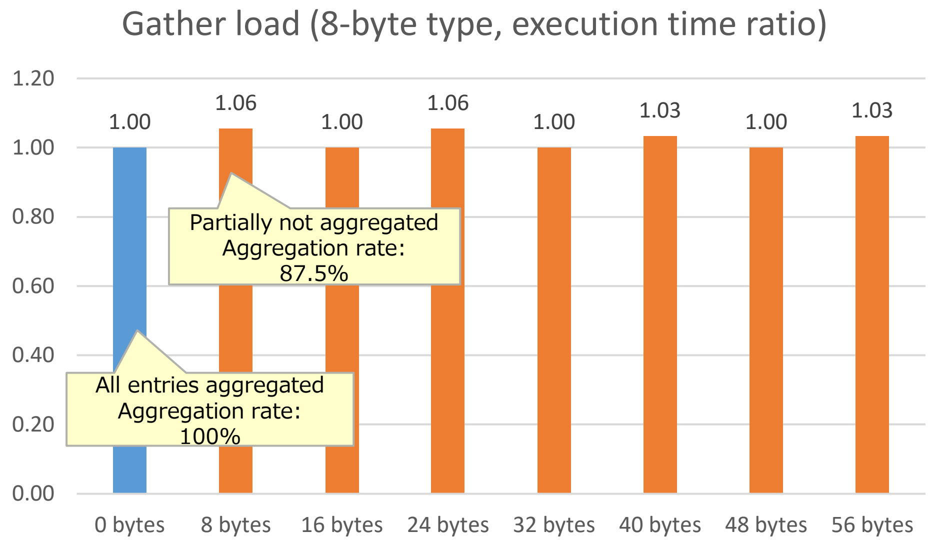


1-byte type (1-byte integer)



Measurement Results: Gather Load

8-byte type (double precision real number/8-byte integer)



Partially not aggregated
Aggregation rate:
87.5%

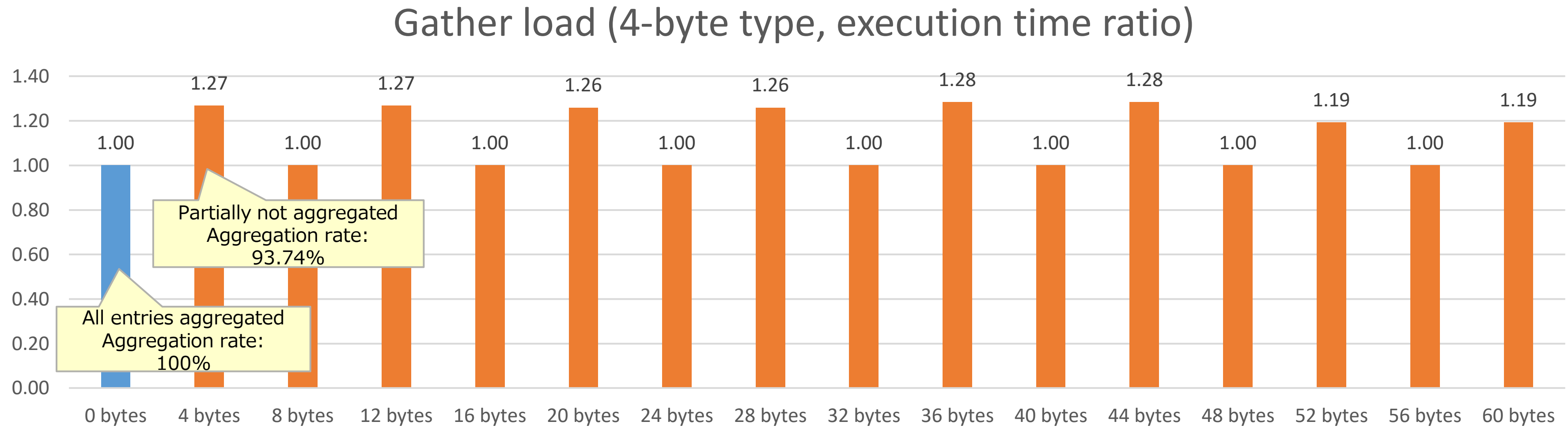
All entries aggregated
Aggregation rate:
100%

Evaluation code

```
do k  = 1, iter
   do i = 1,n
      y(1,i) = x1(1,i)
   enddo
enddo
```

Delete the store instruction from the assembly code.

4-byte type (single precision real number/4-byte integer)



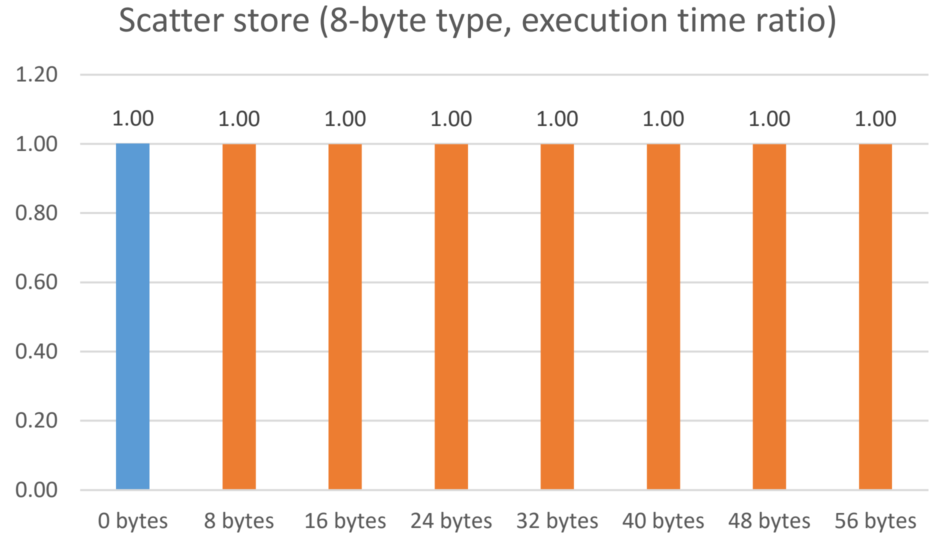
Partially not aggregated
Aggregation rate:
93.74%

All entries aggregated
Aggregation rate:
100%

Measurement Results: Scatter Store

FUJITSU

8-byte type (double precision real number/8-byte integer)

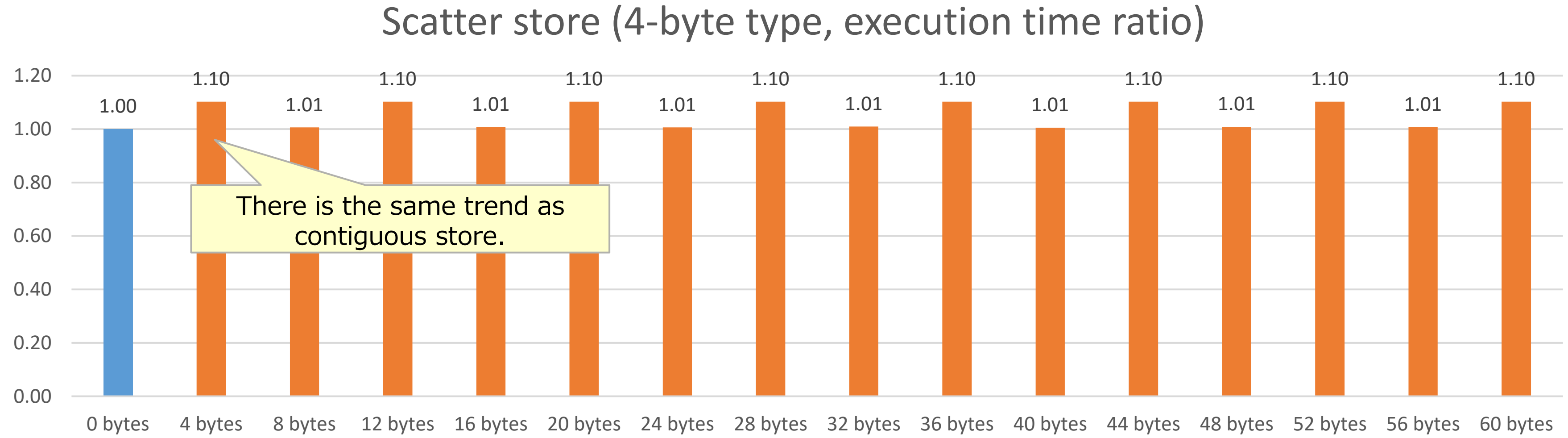


Evaluation code

```
do k  = 1, iter
   do i = 1,n
      y(1,i) = x1(1,i)
   enddo
enddo
```

Delete the load instruction from the assembly code.

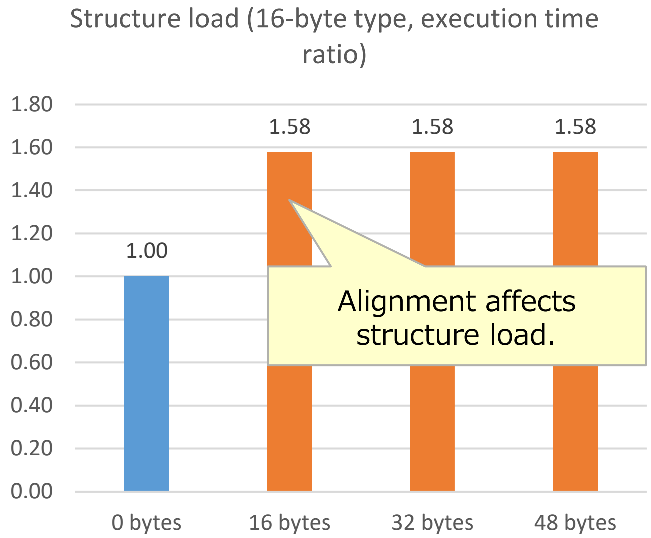
4-byte type (single precision real number/4-byte integer)



There is the same trend as contiguous store.

Measurement Results: Structure Load (LD2 Instruction)

Double precision complex number type (16-byte complex)



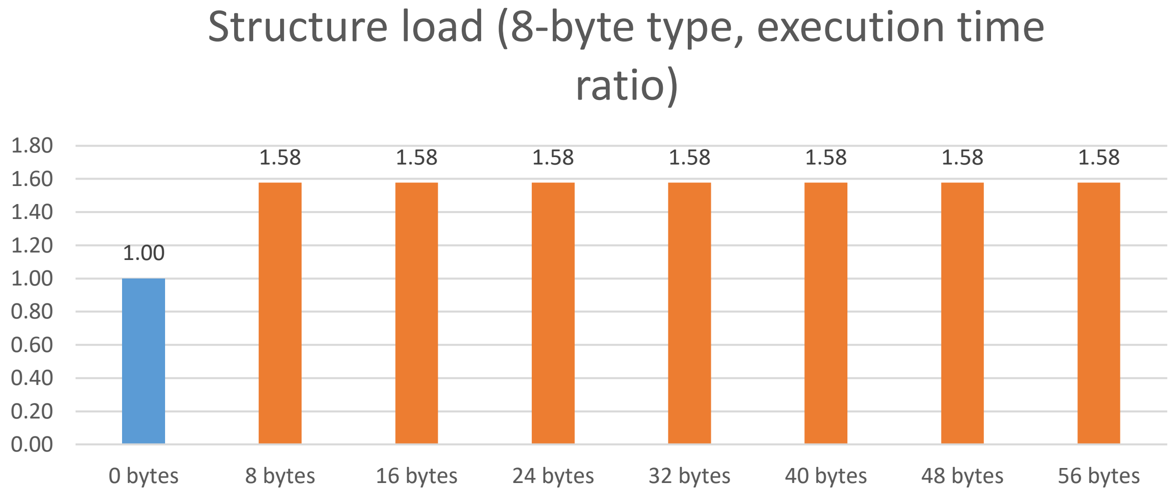
Alignment affects structure load.

Evaluation code

```
do k  = 1, iter
   do i = 1,n
      y(i) =  c1 * x1(i)
   enddo
enddo
```

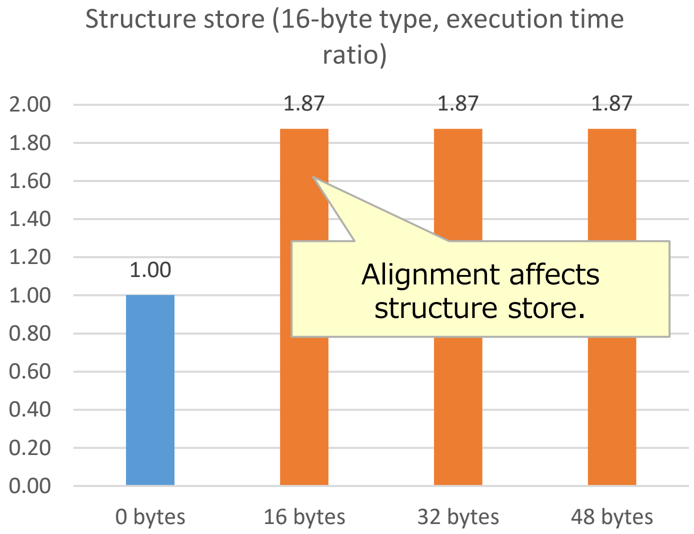
Delete the fadd and store instructions from the assembly code, using the complex number type.

Single precision complex number type (8-byte complex)



Measurement Results: Structure Store (ST2 Instruction)

Double precision complex number type (16-byte complex)


Structure store (16-byte type, execution time ratio)

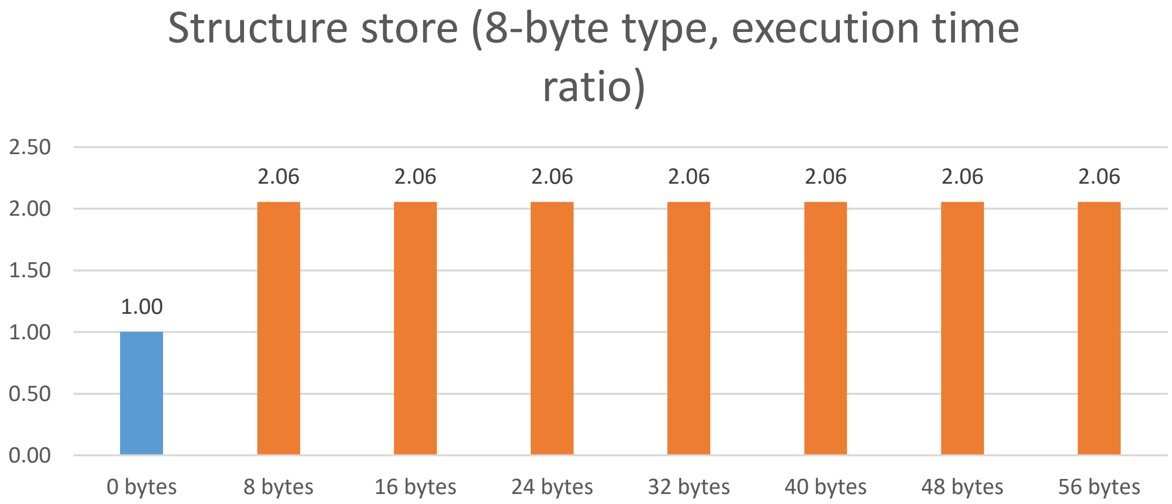
Alignment affects structure store.

Evaluation code

```
do k  = 1, iter
   do i = 1,n
      y(i) = c1 * x1(i)
   enddo
enddo
```

Delete the fadd and load instructions from the assembly code, using the complex number type.

Single precision complex number type (8-byte complex)


Structure store (8-byte type, execution time ratio)

Memory Copy Performance

- Purpose
- Measurement Conditions
- Measurement Results
- Summary

Purpose

- Purpose
  - To evaluate the memory copy performance using Fortran code and memcpy (glibc/high-speed version) and identify use scenes

- About high-speed version memcpy
  - It becomes available when the relevant option is specified. (-Kfast, A64FX, SVE, optlib_string, nolib)
  - High-speed version memcpy is used for the MPI as well.

Measurement Conditions

FUJITSU

Measurement conditions

Condition	Pattern
Verification code	- Fortran code without cache cleaning - Fortran code with cache cleaning - memcpy (C language) without cache cleaning - memcpy (C language) with cache cleaning
Number of threads to be measured	- 1 thread, 12 threads
Compiler	- Compiler for the A64FX (March 20, 2019 version)
Compilation option	- Fortran code : -Kfast ＊ Fortran code with -Kzfill is also evaluated. - memcpy default : -Kfast - memcpy high-speed version : -Kfast,A64FX,SVE,optlib_string,nolib
Access range	- 1 thread: Copy size 256 bytes to 1,073,741,824 bytes - 12 threads: Copy size 288 bytes to 1,073,741,856 bytes ＊ When 12 threads are measured, the copy size per thread is equal to the above copy size divided by 12.

Without cache cleaning
 The cache is not cleared each time memory copy is executed once.
 Depending on the copy size, on cache occurs.
With cache cleaning
 The cache is cleared each time memory copy is executed once.
 Memory access occurs in any copy size.

Code to be measured

```
double y[n], x1[n]  <– 1 thread: 32 to 134217728
for (j = 0; j < iter; j++) {
  Start of timer measurement
  memcpy(y, x1, (copy size));
  End of timer measurement
  Cache cleaning
}
```

Execute the following
 Fortran code.

```
 do i = 1, n
   y(i)=x1(i)
 end do
```

For memcpy, evaluate the following.
 default (glibc)
 High-speed version

＊ default (glibc) was measured of the pre-accelerated version for F64FX.

[Reference]
Memory Copy Performance Measurement Results (1/4)

1 thread, without cache cleaning

Copy size (bytes)	Throughput (GB/sec)			
	Fortran code	Fortran code with zfill	memcpy default	memcpy high-speed version
256	5.69	5.39	3.74	4.53
512	11.25	10.14	6.92	8.98
1,024	19.69	17.66	10.34	16.79
2,048	36.57	29.47	15.06	31.75
4,096	63.02	46.55	18.24	54.25
8,192	88.56	38.64	21.33	79.15
16,384	88.80	40.76	22.90	96.95
32,768	79.63	39.27	20.99	76.03
65,536	54.64	44.04	19.51	55.54
131,072	56.17	45.95	19.57	59.82
262,144	57.28	47.08	19.77	62.10
524,288	57.49	47.72	19.85	63.34
1,048,576	57.78	48.02	19.90	64.12
2,097,152	57.87	47.91	19.90	64.44
4,194,304	44.65	41.41	18.52	35.40
8,388,608	49.66	43.83	18.03	35.79
16,777,216	49.71	43.89	18.03	36.01
33,554,432	49.44	43.70	18.00	35.86
67,108,864	45.24	40.59	17.50	33.71
134,217,728	45.29	40.57	17.50	33.76
268,435,456	45.15	40.56	17.48	33.67
536,870,912	44.45	40.49	17.46	33.56
1,073,741,824	44.42	40.50	17.46	33.59

[Reference]
Memory Copy Performance Measurement Results (2/4)

FUJITSU

■ 1 thread, with cache cleaning

Copy size (bytes)	Throughput (GB/sec)			
	Fortran code	Fortran code with zfill	memcpy default	memcpy high-speed version
256	2.20	1.36	1.88	2.09
512	4.23	2.43	2.47	3.98
1,024	7.56	4.28	2.81	6.87
2,048	9.92	6.85	3.10	9.66
4,096	16.25	11.74	7.13	16.06
8,192	23.11	19.67	10.79	23.21
16,384	30.01	25.13	10.92	30.62
32,768	39.31	33.13	15.37	39.27
65,536	41.39	36.75	16.22	27.84
131,072	44.36	39.71	17.05	27.84
262,144	45.95	41.46	17.53	27.60
524,288	46.62	42.34	17.77	27.64
1,048,576	47.01	42.84	17.90	27.59
2,097,152	47.24	43.00	17.97	27.62
4,194,304	47.34	43.14	18.00	35.73
8,388,608	47.15	43.00	17.99	35.74
16,777,216	44.64	40.60	17.61	34.00
33,554,432	44.07	39.86	17.49	33.65
67,108,864	44.10	39.87	17.49	33.65
134,217,728	44.14	39.96	17.50	33.72
268,435,456	44.03	39.84	17.48	33.67
536,870,912	43.92	39.46	17.46	33.59
1,073,741,824	43.94	39.73	17.46	33.59

[Reference]
Memory Copy Performance Measurement Results (3/4)

■ 12 threads, without cache cleaning

Copy size (bytes)	Throughput (GB/sec)			
	Fortran code	Fortran code with zfill	memcpy default	memcpy high-speed version
288	1.05	1.05	0.77	0.76
576	2.76	2.76	1.57	1.55
1,056	5.82	5.77	2.88	2.87
2,112	11.77	11.42	5.83	5.93
4,128	25.48	25.02	11.20	11.90
8,256	47.86	47.59	22.74	23.76
16,416	88.74	90.45	41.51	41.04
32,832	153.42	160.16	74.70	93.01
65,568	277.83	221.89	120.31	182.64
131,136	467.51	283.54	174.62	315.61
262,176	522.78	298.78	188.62	481.50
524,352	467.34	330.40	200.33	439.71
1,048,608	530.94	362.15	220.41	562.11
2,097,216	570.44	374.27	227.93	601.61
4,194,336	328.48	294.05	189.47	424.89
8,388,672	140.45	206.26	141.31	143.16
16,777,248	140.18	206.39	141.53	143.32
33,554,496	140.23	206.37	141.70	143.61
67,108,896	140.20	206.59	142.16	206.10
134,217,792	140.30	206.83	142.21	206.36
268,435,488	140.21	206.65	142.32	206.51
536,870,976	140.02	206.79	142.38	206.42
1,073,741,856	139.88	206.44	142.45	206.44

[Reference]
Memory Copy Performance Measurement Results (4/4)

FUJITSU

- 12 threads, with cache cleaning

Copy size (bytes)	Throughput (GB/sec)			
	Fortran code	Fortran code with zfill	memcpy default	memcpy high-speed version
288	0.82	0.83	0.97	0.93
576	1.51	1.50	1.91	1.68
1,056	2.82	2.81	3.33	3.10
2,112	5.69	5.58	6.09	5.71
4,128	11.01	11.05	10.41	10.49
8,256	21.44	20.46	17.33	21.50
16,416	36.04	35.88	25.45	36.04
32,832	61.95	55.93	33.42	60.97
65,568	86.67	75.37	44.53	87.19
131,136	109.33	116.36	65.06	101.73
262,176	123.00	144.29	92.06	121.89
524,352	133.54	167.15	102.95	118.93
1,048,608	141.42	185.02	120.94	130.78
2,097,216	137.25	201.39	132.87	139.55
4,194,336	137.96	197.40	133.33	138.75
8,388,672	138.41	204.53	138.73	141.53
16,777,248	138.96	204.34	139.54	142.39
33,554,496	140.32	205.50	141.09	143.06
67,108,896	139.77	206.85	141.68	204.27
134,217,792	140.28	206.23	141.84	205.77
268,435,488	139.95	206.84	141.98	205.82
536,870,976	140.20	205.95	142.16	206.01
1,073,741,856	139.85	205.48	142.27	206.03

Summary

- memcpy high-speed version
  - In every case, this version is faster than memcpy default (glibc version).
    -> The use of the high-speed version of memcpy is recommended.
  - If the copy size (per thread) exceeds 4 MiB, zfill is automatically enabled.
    - When one thread is executed (memory is not busy), care needs to be exercised because the performance drops.
    - When 12 threads are executed (memory is busy), the performance improves.
      -> Depending on the copy size, zfill may fail to be enabled even when the memory is busy.
- Fortran code
  - While the performance is equal to or better than the high-speed version of memcpy, zfill needs to be set manually.
  - Cases when Fortran code is faster than the high-speed version of memcpy
    - When the memory is not busy, Fortran code is faster if the copy size is 4 MiB to 1 GiB.
    - When the memory is busy, Fortran code with zfill is faster. (Compared to when zfill is disabled in the high-speed version of memcpy)

Inter-CMG Performance Evaluation

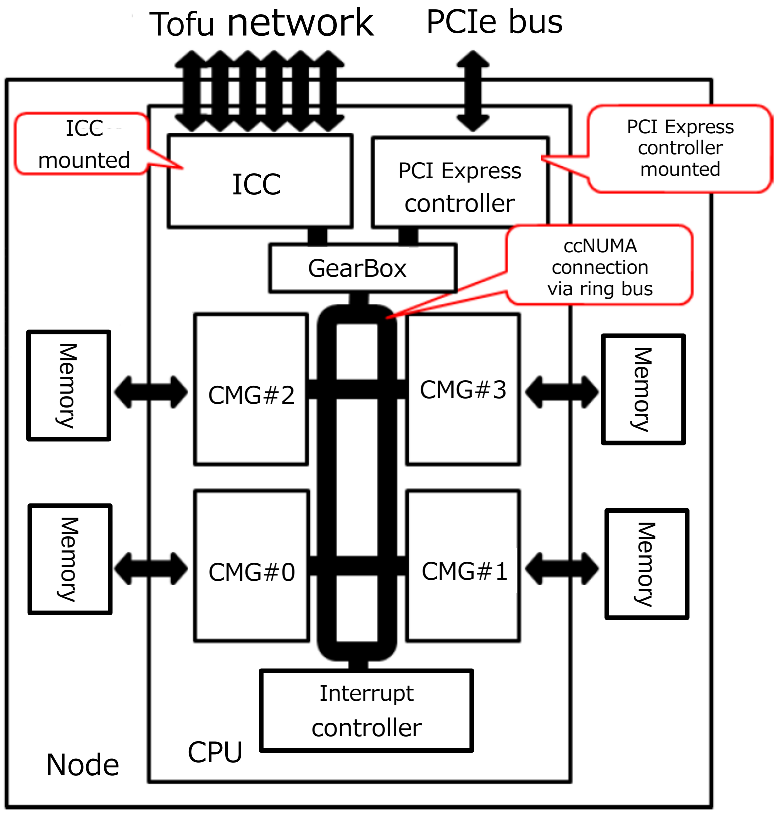
- Hardware Overview
- About the Specification of numactl (Core, Memory, Interleave)
- Throughput Measurement Measurement Conditions
- Throughput Measurement Measurement Results (12 Thread Execution)
- Measurement Results (48 Thread Execution)
- Latency Measurement Measurement Conditions
- Latency Measurement Measurement Results
- Summary
- DGEMM Across CMGs A64FX Implementation
- DGEMM Efficiency in Multiple CMGs

Hardware Overview

- Memory (HBM)
  Four HBM (High Bandwidth Memory) chips are directly connected to the CPU LSI.
  One HBM has a 1024-bit data interface supporting transfer rates of up to 2 Gbps.
  The memory bandwidth is 1024 bits (two-way) × 2 Gbps × 4 (HBM), totaling 1024 GB/s.

- Inter-CMG connection
  The intra-chip network configuration interconnecting CMGs is shown here. This is a two-way ring bus network that connects the following six points.
  - Four CMGs
  - Interconnect controller (ICC)/
    PCI Express controller
  - Interrupt controller

  There are two 64-byte ring buses (two-way), and the transfer capability of these ring buses is 128 GB/s × 2 (two-way).



About the Specification of numactl (Core, Memory, Interleave)

About the specification of numactl

The numactl command uses the following.
- Core specification: -C
- Memory specification: -m
- Interleave specification: –-interleave

CMG	Core	Memory
CMG0	12 to 23	4
CMG1	24 to 35	5
CMG2	36 to 47	6
CMG3	48 to 59	7

Example of numactl)

In this example, 48 threads (48 cores) and all the memories are used. For the specified values, see the table at the right.

```
numactl –C12-59 –m4-7
```

Throughput Measurement Measurement Conditions

Measurement conditions

	Pattern
Code to be measured	- Triad memory access
Number of cores to be measured and memory settings	12 core execution (CMG0) - Intra-CMG memory used - Inter-CMG memory used 48 core execution (CMG0 to CMG3) - Demand paging - Prepaging - Interleave
Compilation option	-Kfast,openmp,zfill
Access range	- Total number of bytes of access arrays: 240 MiB - bss is used. - Double precision operation arrays are used. - Number of innermost loop iterations, array size (n): 10485120 - Number of outer loop iterations (iter): 3
Measurement values	- Memory throughput (GB/s) * Calculated by the memory access volume divided by the measurement time. - Memory throughput peak ratio (%) * The denominator is 256 GB/s when 12 threads are executed and 1024 GB/s when 48 threads are executed.

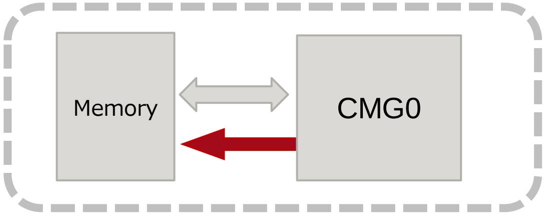
Code to be measured

```
!$omp parallel
 do j = 1, iter
!$omp do
    Do i = 1, n
       y(i)=x1(i) + c0 * x2(i)
    End Do
!$omp end do nowait
 enddo
!$omp end parallel
```

Throughput Measurement Measurement Results (12 Thread Execution)

- Intra-CMG memory used

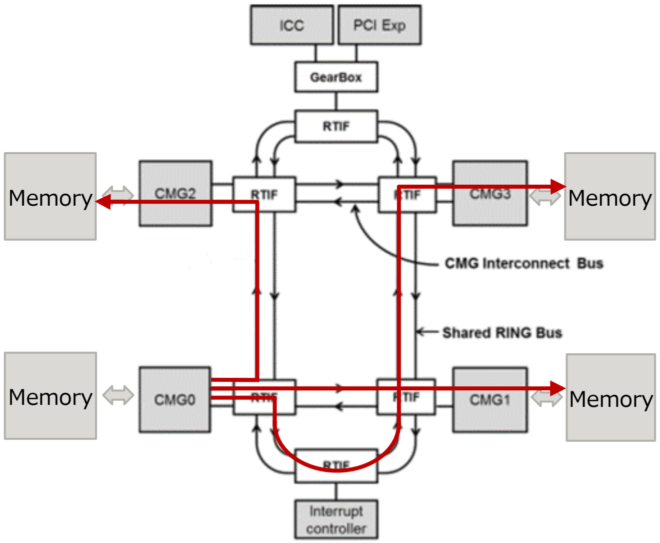
numactl –C12-23 –m4 (Access to the CMG0 memory)



Number of executed threads	Executed cores	Specified memory	Through put (GB/s)	Memory throughput peak ratio (%)
12	12 to 23 (CMG0)	4 (CMG0)	210.8	82.3%

- Inter-CMG memory used

numactl –C12-23 –m5 to 7 (Access to the CMG1 to CMG3 memory)



Number of executed threads	Executed cores	Specified memory	Through put (GB/s)	Memory throughput peak ratio (%)
12	12 to 23 (CMG0)	5 (CMG1)	118.5	46.3%
12	12 to 23 (CMG0)	6 (CMG2)	120.8	47.2%
12	12 to 23 (CMG0)	7 (CMG3)	120.9	47.2%

Measurement Results (48 Thread Execution)

FUJITSU

Four CMG memories used

Demand paging

```
export  XOS_MMM_L_PAGING_POLICY=demand:demand:demand
numactl –C12-59 –m4-7
```

Number of executed threads	Executed cores	Specified memory	Throughput (GB/s)	Memory throughput peak ratio (%)
48	12 to 59 (CMG0 to 3)	4 to 7 (CMG0 to 3)	819.7	80.0%

Prepaging

```
export  XOS_MMM_L_PAGING_POLICY=prepage:prepage:prepage
numactl –C12-59 –m4-7
```

Number of executed threads	Executed cores	Specified memory	Throughput (GB/s)	Memory throughput peak ratio (%)
48	12 to 59 (CMG0 to 3)	4 to 7 (CMG0 to 3)	99.3	9.7%

While 4-7 is set in the memory specification, the declaration is executed with CMG0. This allocates all the arrays to the memory (4) of CMG0, preventing the appropriate performance from being achieved.

Interleave

```
numactl –C12-59 --interleave=4-7
```

Number of executed threads	Executed cores	Specified memory	Throughput (GB/s)	Memory throughput peak ratio (%)
48	12 to 59 (CMG0 to 3)	Interleave (CMG0 to 3)	476.0	46.5%

Use interleave if data cannot be divided on a per-CMG basis due to the characteristics of the application.

One CMG memory used (reference)

```
numactl –C12-59 –m4
```

Number of executed threads	Executed cores	Specified memory	Throughput (GB/s)	Memory throughput peak ratio (%)
48	12 to 59 (CMG0 to 3)	4 (CMG0)	93.2	9.1%

Latency Measurement Measurement Conditions

Measurement conditions

	Pattern
Code to be measured	Memory access latency measurement
Cores to be measured	- 1 core execution × 48 cores (all computing cores of CMG0 to CMG3) -> Access to the memory inside the CMG -> Access to the CMG0 memory (access to the memory outside the CMG for cores of CMG1 to CMG3)
Compilation option	-Kfast
Access range	- bss is used. - Number of inner loop iterations (NL): 1024 - Number of outer loop iterations (rep): 1
Measurement value	Access latency (number of cycles)

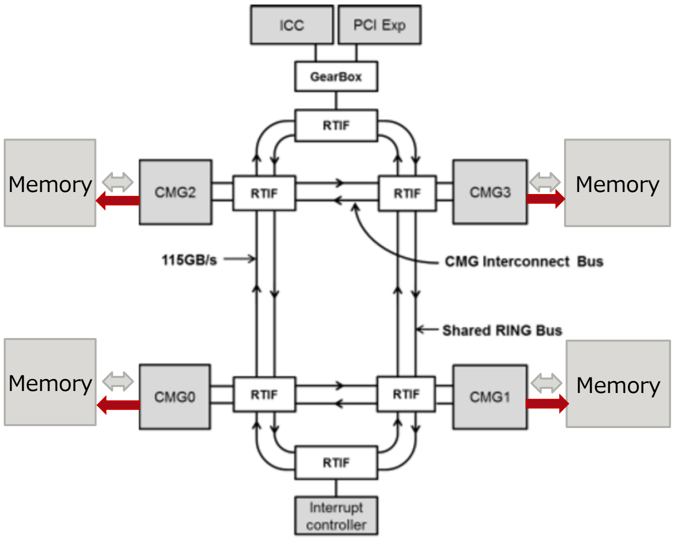
Code to be measured

```
for (i = 0; i < rep; i++) {
    p = 0;
    for (j = 0; j < NL; j++) {
       p = array[p];
    }
    ans = p;
}
```

Latency Measurement Measurement Results (1/2)

- Measurement results (memory access (1))

numactl –m(4 to 7)
Access to each CMG memory



With respect to the specification of 150 ns, there is no problem with the measurement value of 131 ns.

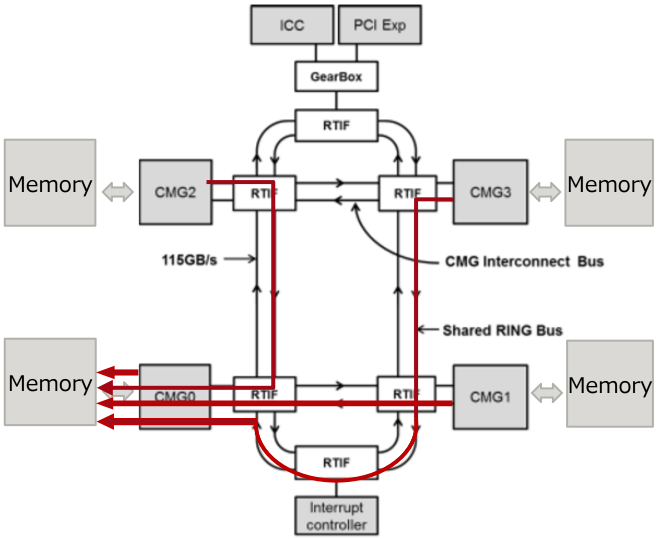
CMG	Core	Latency (cycles)	CMG average (cycles)
0	12	259.86	261.87 131ns
	13	257.17	
	14	259.87	
	15	259.06	
	16	262.60	
	17	259.06	
	18	262.42	
	19	262.22	
	20	263.14	
	21	265.98	
	22	262.56	
	23	268.71	
1	24	256.65	260.75
	25	256.70	
	26	259.86	
	27	256.03	
	28	262.19	
	29	259.02	
	30	262.22	
	31	262.86	
	32	263.14	
	33	262.73	
	34	262.48	
	35	265.35	

CMG	Core	Latency (cycles)	CMG average (cycles)
2	36	262.19	264.54
	37	262.92	
	38	262.89	
	39	262.19	
	40	262.94	
	41	262.94	
	42	263.58	
	43	263.15	
	44	262.88	
	45	265.34	
	46	262.85	
	47	281.17	
3	48	262.19	265.66
	49	259.84	
	50	262.19	
	51	262.95	
	52	262.19	
	53	262.19	
	54	262.28	
	55	263.18	
	56	266.10	
	57	263.58	
	58	281.16	
	59	281.17	

Latency Measurement Measurement Results (2/2)

- Measurement results (memory access (2))

numactl –m4
Access to the CMG0 memory



Reference values for thread parallel processing across CMGs and intra-node MPI communication performance

CMG	Core	Latency (cycles)	CMG average (cycles)
0	12	259.86	261.87 131ns
	13	257.17	
	14	259.87	
	15	259.06	
	16	262.60	
	17	259.06	
	18	262.42	
	19	262.22	
	20	263.14	
	21	265.98	
	22	262.56	
	23	268.71	
1	24	387.24	389.86
	25	387.88	
	26	388.93	
	27	387.22	
	28	388.93	
	29	387.24	
	30	388.98	
	31	388.96	
	32	393.26	
	33	393.26	
	34	393.26	
	35	393.28	

CMG	Core	Latency (cycles)	CMG average (cycles)
2	36	423.55	429.96
	37	423.52	
	38	423.52	
	39	423.54	
	40	424.16	
	41	423.52	
	42	436.25	
	43	436.25	
	44	437.00	
	45	436.25	
	46	436.25	
	47	436.25	
3	48	423.57	429.99 215ns
	49	423.52	
	50	423.54	
	51	424.29	
	52	424.10	
	53	423.52	
	54	436.46	
	55	436.43	
	56	436.25	
	57	436.25	
	58	436.25	
	59	436.25	

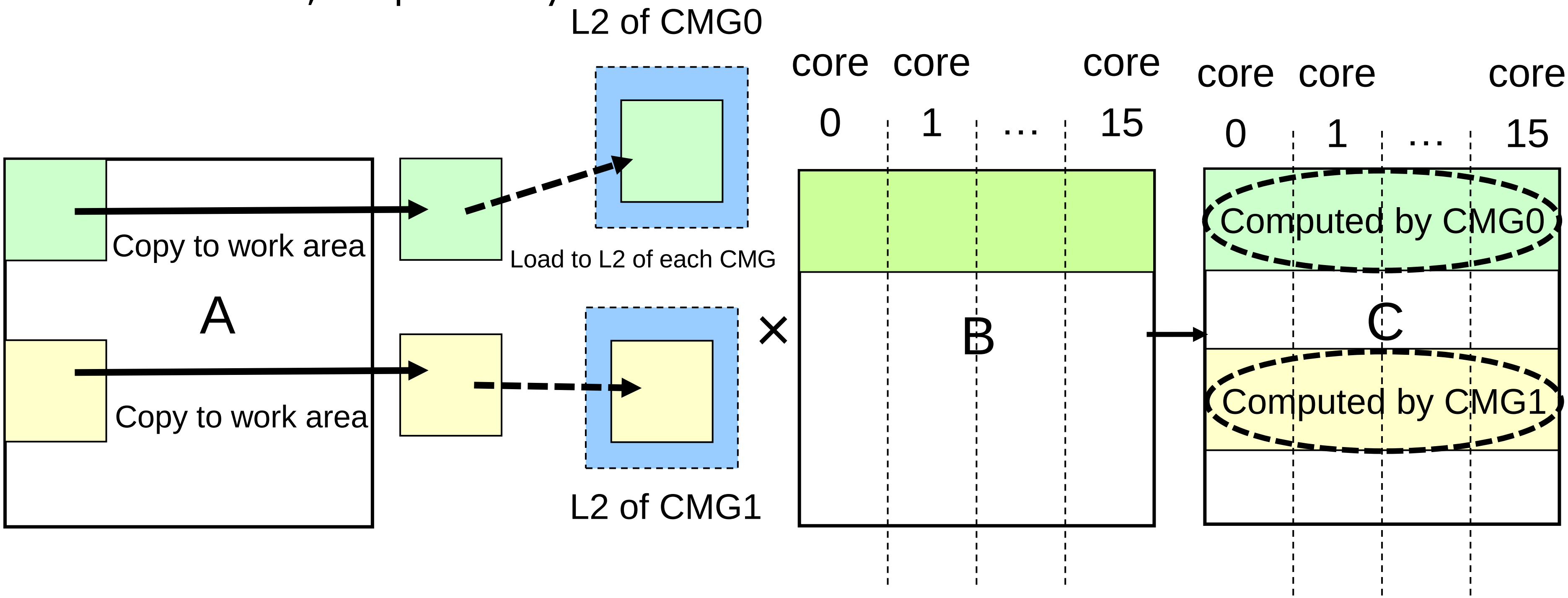
Summary

- In the case of local access (memory access only to the local CMG)
  - There is no performance problem as efficiency of 80% or more is achieved when 12 or 48 threads are executed.
  - There is no problem with the recommended 4-process, 12-thread execution.
- In the case of global access (memory access made to other CMGs as well)
  - There is a bottleneck in the inter-CMG bus performance (peak at 128 GB/s in the case of 2.0 GHz).
  - When only one CMG has memory and the other three CMGs access that memory, the performance may be up to 100 GB/s or so.
- Whether the appropriate performance can be achieved when 48 threads are executed depends on the application.

  -> It is necessary to identify the characteristics of the application and consider executing interleave.

DGEMM Across CMGs A64FX Implementation

- Proposed division for a multitude of cores
  - Use different matrix division dimensions for CMGs and cores, respectively.

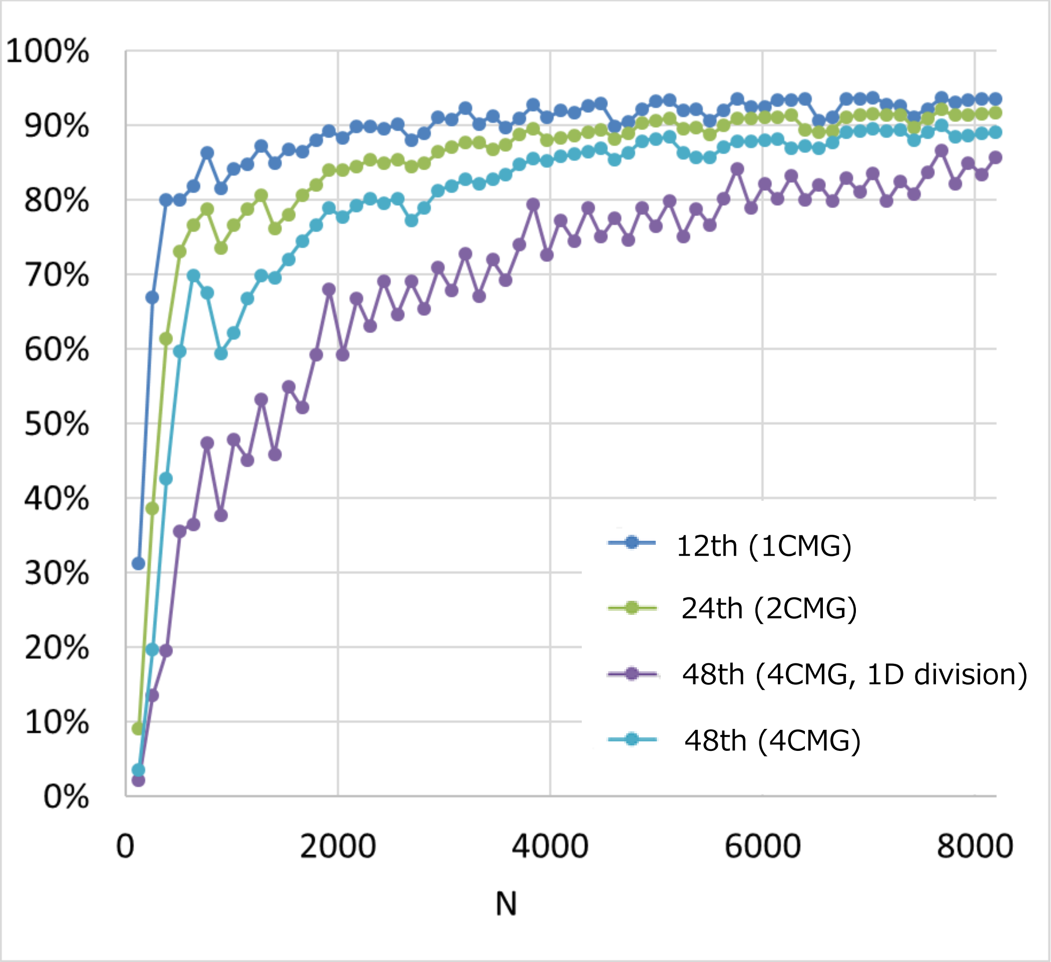


Instead of assigning threads to one dimension, assign them to two dimensions separately in units of a CMG and in units of a core. This reduces the unnecessary movement of entries between cores.

DGEMM Efficiency in Multiple CMGs

- A drop in performance in the range where the value is large (N = 7000 to 8000) is 1.5% when changing from one CMG to two CMGs and 5% or so when changing from one CMG to four CMGs.
- Improvement from one-dimension division
  In the 48th one-dimension division, only the side of N is divided into 48 narrow parts, resulting in a huge drop in performance. This has been improved.
  (In one-dimension division, the lines in the graph are jagged because of register blocking and surplus threads.)
- Why performance declines even after the improvement
  - Performance drops because the size per CMG becomes smaller.
  - Lower performance results from the original matrix data spanning CMGs.

Measured with the A64FX



OpenMP Overhead Evaluation

- Two OpenMP libraries (LLVM version and Fujitsu version)
- Performance of Fujitsu OpenMP Library
- Basic performance of LLVM OpenMP and Fujitsu OpenMP Libraries
- Stream benchmark performance: each compiler x library

Two OpenMP libraries (LLVM version and Fujitsu version) (1/2)

- The LLVM OpenMP library is an OpenMP library based on the LLVM OpenMP Runtime Library extended for the A64FX.

OpenMP Library	Option	Supported Functions
LLVM OpenMP library	-Nlibomp (default)	OpenMP 4.5 and Parts of 5.0 Hardware barrier(Default is Software barrier) Sector cache Bind to core (default)
Fujitsu OpenMP library	-Nfjomplib	OpenMP 3.1 Hardware barrier Sector cache Bind to core (Default when execute on job)

- Compiler combination
  - The object files(.o) are common in Fortran and C/C++ Trad Mode, and libraries used can be specified with the –Nlibomp/-Nfjomplib option.
    (If Clang Mode object files are included, only –Nlibomp option is available)

Option	Fortran	C/C++	
		Trad Mode	Clang Mode
-Nlibomp	Available	Available	Available
-Nfjomplib	Available	Available	Not available

Two OpenMP libraries (LLVM version and Fujitsu version) (2/2)

Selection method

- Specify in compiler option when linking.
  - -Nlibomp (default) : Use LLVM OpenMP Library
  - -Nfjomplib : Use Fujitsu OpenMP Library

Difference in specifications

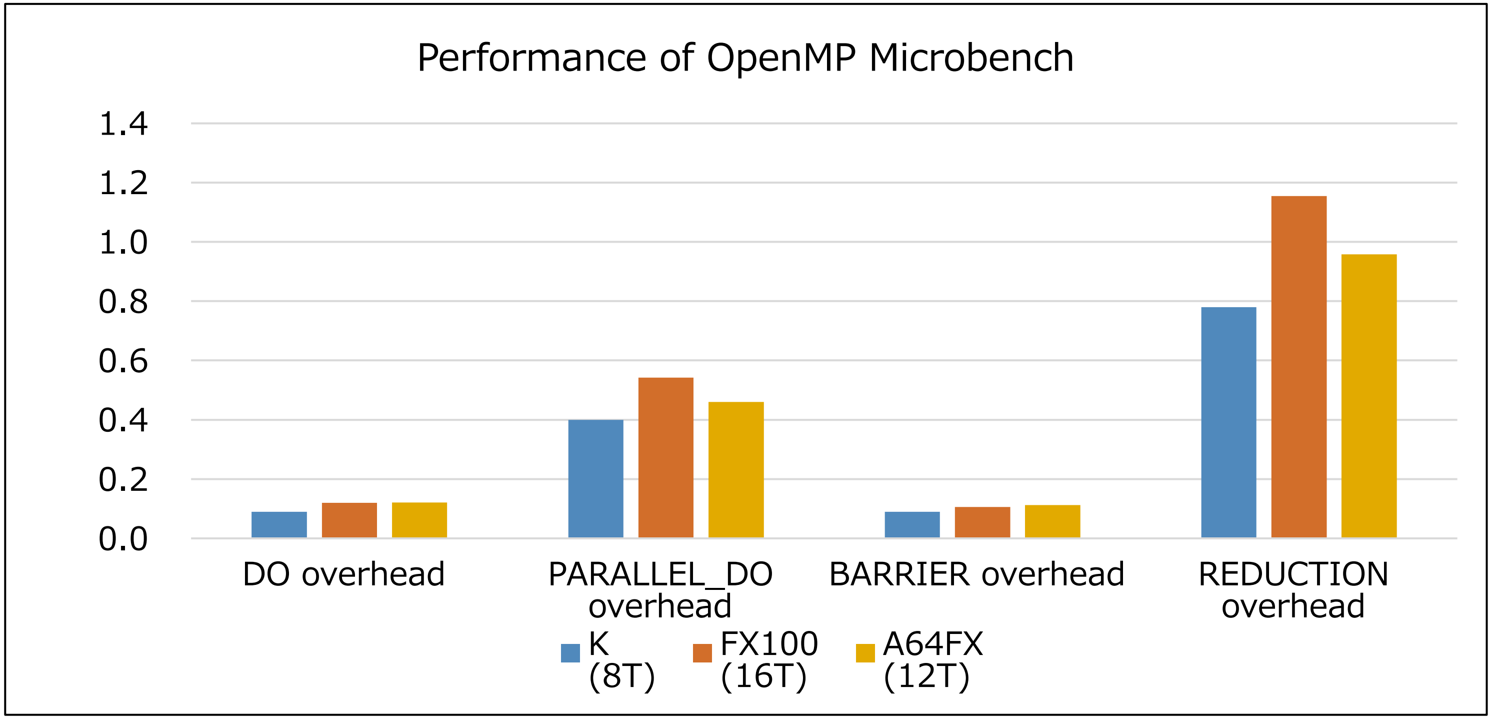
- Thread stack size

Option	Default size	Environment variables for resizing
-Nlibomp	• 8MiB	OMP_STACKSIZE
-Nfjomplib	• Inherit the process stack size. • If the stack size of the process is specified as unlimited. (Memory size / Number of threads) / 5	OMP_STACKSIZE or THREAD_STACK_SIZE

Performance of Fujitsu OpenMP Library (1/3)

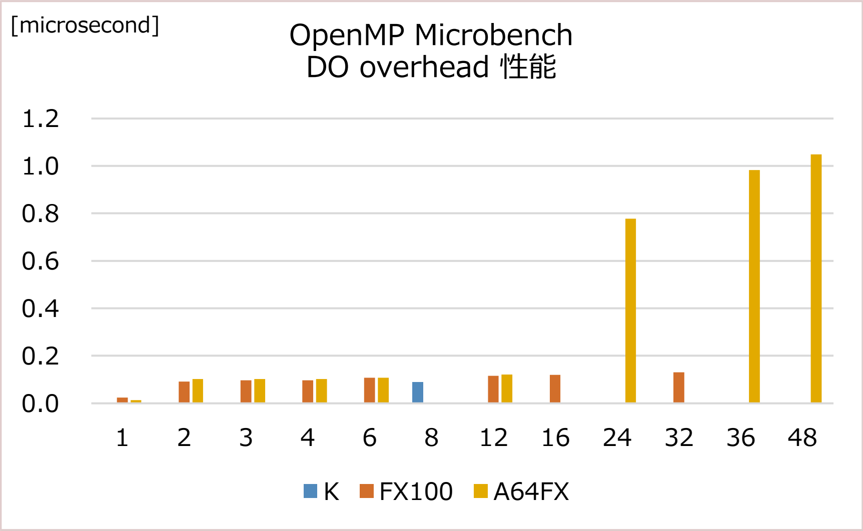
- Typical directive performance comparison

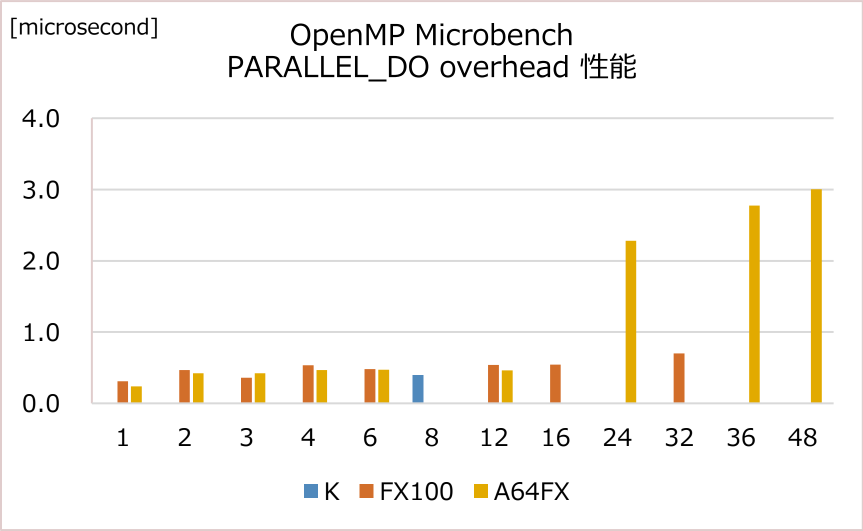
OpenMP Microbench Units:microsecond	K (8T)	FX100 (16T)	A64FX (12T)
DO overhead	0.090	0.120	0.121
PARALLEL_DO overhead	0.400	0.542	0.461
BARRIER overhead	0.090	0.106	0.113
REDUCTION overhead	0.780	1.154	0.957

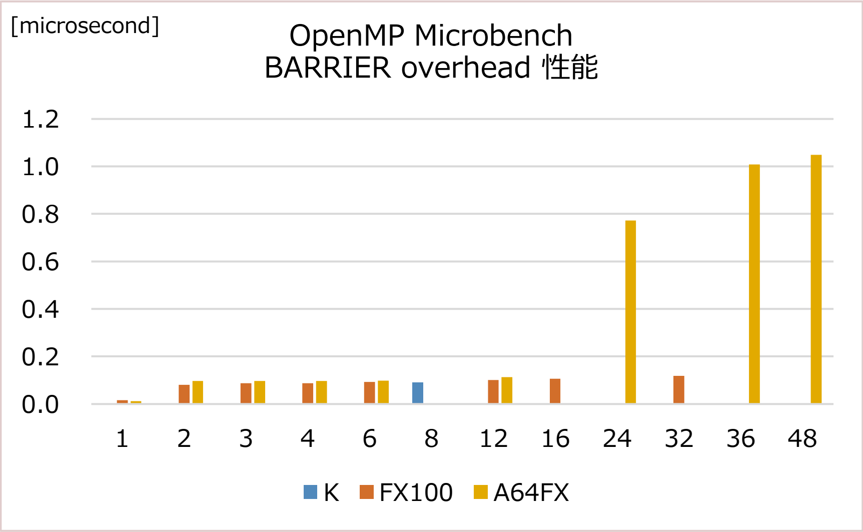


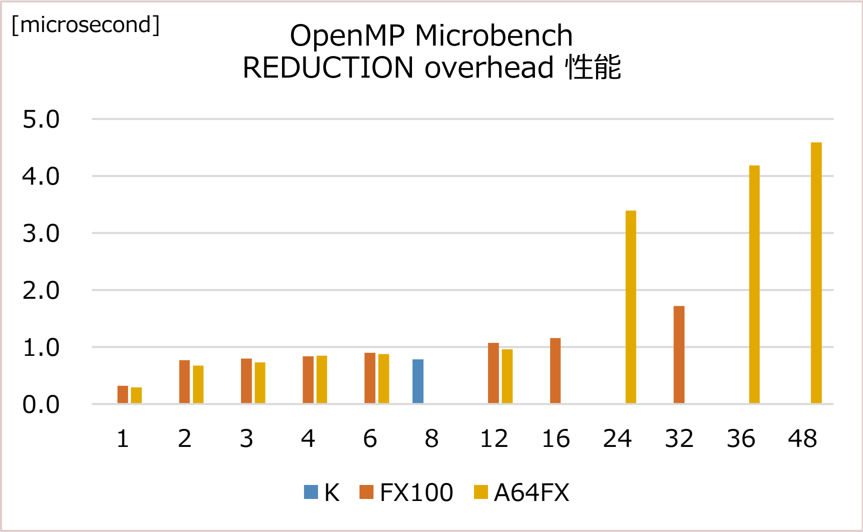
Performance of Fujitsu OpenMP Library (2/3)

Performance comparison (Per Thread)


OpenMP Microbench
DO overhead 性能


OpenMP Microbench
PARALLEL_DO overhead 性能


OpenMP Microbench
BARRIER overhead 性能


OpenMP Microbench
REDUCTION overhead 性能

Performance of Fujitsu OpenMP Library (3/3)

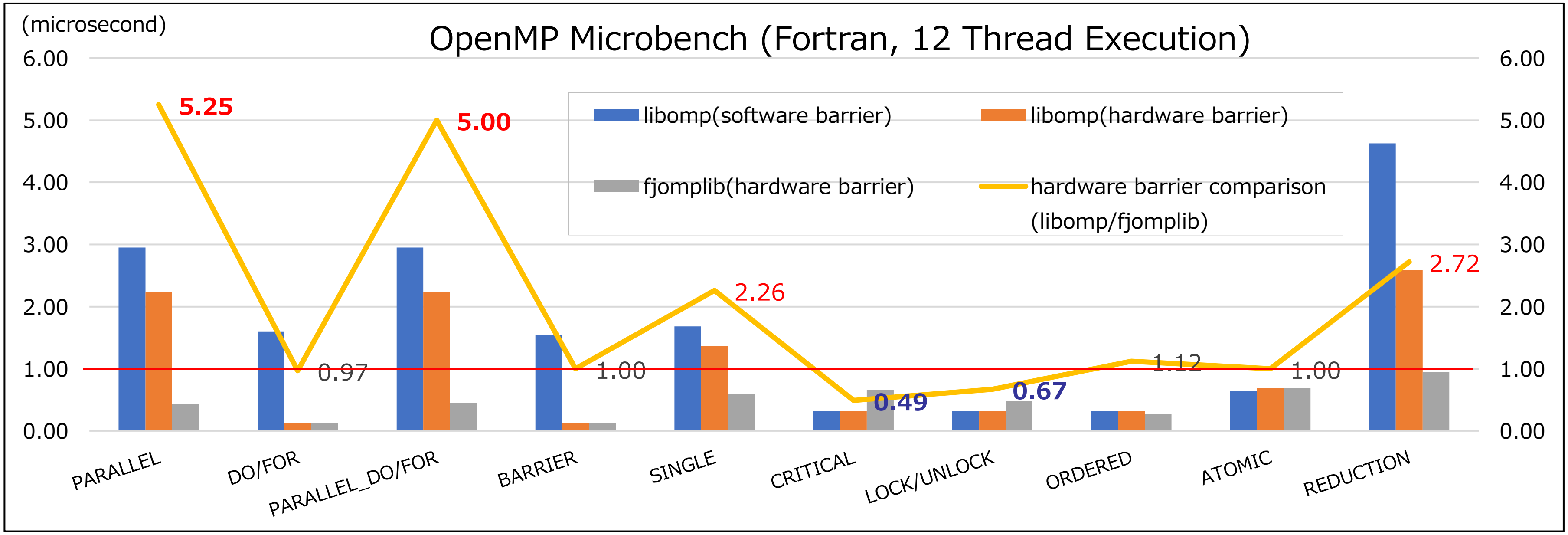
- Performance comparison (Per Thread)

OpenMP Microbench Units: microsecond		Number of threads											
		1	2	3	4	6	8	12	16	24	32	36	48
DO overhead	K						0.090						
	FX100	0.023	0.092	0.097	0.097	0.107		0.115	0.120		0.130		
	A64FX	0.014	0.103	0.103	0.103	0.108		0.121		0.778		0.983	1.049
PARALLEL_DO overhead	K						0.400						
	FX100	0.309	0.466	0.358	0.535	0.478		0.535	0.542		0.701		
	A64FX	0.236	0.422	0.422	0.464	0.471		0.461		2.283		2.774	3.004
BARRIER overhead	K						0.090						
	FX100	0.016	0.081	0.087	0.088	0.092		0.101	0.106		0.119		
	A64FX	0.012	0.097	0.097	0.097	0.098		0.113		0.773		1.008	1.049
REDUCTION overhead	K						0.780						
	FX100	0.317	0.769	0.794	0.835	0.899		1.073	1.154		1.717		
	A64FX	0.292	0.670	0.732	0.850	0.875		0.957		3.390		4.182	4.589

Basic performance of LLVM OpenMP and Fujitsu OpenMP Libraries

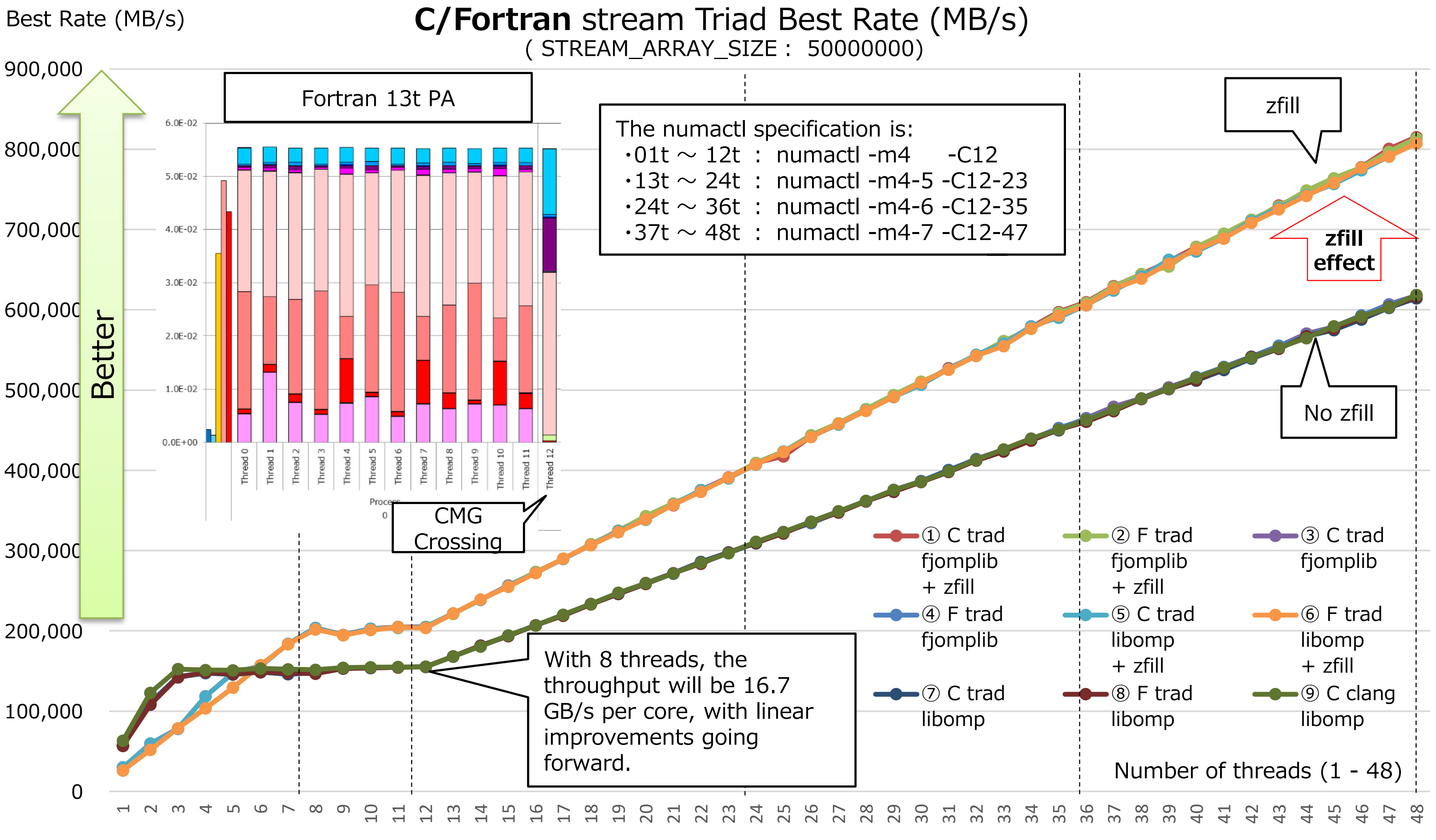
Processing	Run time(microsecond)			hardware barrier comparison (libomp/fjomplib)
	libomp		fjomplib	
	software barrier	hardware barrier	hardware barrier	
PARALLEL	2.95	2.24	0.43	**5.25**
DO/FOR	1.60	0.13	0.13	0.97
PARALLEL_DO/FOR	2.95	2.23	0.45	**5.00**
BARRIER	1.55	0.12	0.12	1.00
SINGLE	1.68	1.37	0.60	2.26
CRITICAL	0.32	0.32	0.66	**0.49**
LOCK/UNLOCK	0.32	0.32	0.48	**0.67**
ORDERED	0.32	0.32	0.28	1.12
ATOMIC	0.65	0.69	0.69	1.00
REDUCTION	4.63	2.59	0.95	2.72

PARALLEL syntax should be used with caution when using LLVM OpenMP libraries(default)



Stream benchmark performance: each compiler x library

- Performance comparison between C(trad/clang) and Fortran
  -> Performance is equivalent



C/Fortran stream Triad Best Rate (MB/s)
(STREAM_ARRAY_SIZE : 50000000)

The numactl specification is:
- ・01t ～ 12t : numactl -m4 -C12
- ・13t ～ 24t : numactl -m4-5 -C12-23
- ・24t ～ 36t : numactl -m4-6 -C12-35
- ・37t ～ 48t : numactl -m4-7 -C12-47

With 8 threads, the throughput will be 16.7 GB/s per core, with linear improvements going forward.

Revision History

Revision History

Version	Date	Details
1.1	May 14, 2020	- First published
1.2	Sep 30, 2020	- Correcting typographical errors and expressions by reviewing articles
1.3	Mar 31, 2021	- Correcting typographical errors and expressions by reviewing articles
1.4	Aug , 2021	- Modified the glibc article for “Memory Copy Performance” pages

FUJITSU

shaping tomorrow with you