Optimization for Arm SVE and Post-K

RIKEN R-CCS
Yuetsu Kodama (yuetsu.kodama@riken.jp)

Outline

- Introduction of RIKEN simulator
  - ✓ Accuracy Improvement of Memory System Simulation for the Post-K gem5 simulator
- Optimization point for SVE
  - ✓ SVE vectorization
  - ✓ Software pipeline
  - ✓ …

RIKEN Simulator (Overview)

- **It is a processor simulator of A64FX.**
  - ✓ It currently enables simulation of 1CMG, where 12 cores with OpenMP execution is available.
- **It is developed based on open source general purpose processor simulator, gem5.**
  - ✓ Initially RIKEN developed O3 (out-of-order) mode for SVE, but currently moved to the version developed by Arm.
  - ✓ RIKEN continues to extend cache and memory system for HPC.
- **It can simulate binaries generated by an Arm SVE compliant compiler (such as Fujitsu prototype compiler or Arm compiler) in an out-of-order execution pipeline.**
  - ✓ You should generate single static binary including the library.
  - ✓ Currently, it supports only Fujitsu and Gcc OpenMP library, not ARM (LLVM) library and any MPI program.
- **It is aimed to estimate the execution time with accuracy that can be used for performance evaluation and tuning.**

gem5

- **Open source general purpose processor simulator**
  - ✓ Refer http://gem5.org for details
  - ✓ Tutorial: ISCA2011, ASPLOS2017, …
- **Multiple ISAs**
  - ✓ Alph, Arm, SPARC, MIPS, Power, x86, GPU, RISC-V, …
- **Multiple System mode**
  - ✓ FS (Full System) mode, SE (System Emulation) mode
- **Several CPU execution model**
  - ✓ Atomic, in-order, out-of-order(o3), …
  - ✓ The O3 pipeline architecture is based on Alpha
  - ✓ Detailed parameters such as out-of-order resource size, the number of function unit, etc. can be set
- **License**
  - ✓ Berkely-style、free to use as long you as leave our copyright on it.

RIKEN Simulator（Details１）

- **Tuning detailed parameters such as o3 resource size according to A64FX**
  - ✓ The size of Reservation station, reorder buffer, rename register, etc.
  - ✓ Latency of each pipeline, number of simultaneously issued instructions, etc.
  - ✓ Number, latency and throughput of computing units, etc.
  - ✓ Latency and throughput for each instruction group
  - ✓ L1 / L2 cache size, number of ways, latency, throughput, cache line size, etc.
- **Differences between gem5 and A64FX**
  - ✓ The number of reservation station (Fused or Distributed)
  - ✓ Memory address calculation (In memory unit or Independent unit)
  - ✓ Function unit for Instruction (single units or multiple units)

RIKEN Simulator（Details 2）

- **Expand the new functions**
  - ✓ L1 cache: 1 port SRAM, access across lines without overhead
  - ✓ L2 cache: high throughput by multiple banks
  - ✓ Memory: support HBM2, interleave and bank schedule for A64FX
  - ✓ Bus: asymmetric throughput at input and output
  - ✓ Software prefetch: store prefetch, target L2 prefetch
  - ✓ Hardware prefetch: K-Computer compliant prefetch, target L2 prefetch (store prefetch)
- **Usability improvement**
  - ✓ Get statistical information of simulation execution (execution time, cache miss, etc.) in a specified region
  - ✓ Get statistical information compatible with Fujitsu detailed profiles.
  - ✓ Counting floating operations taking into account predicates.

We are considering to release these extensions separately from the parameter s of A64FX.

Evaluations

- **It is important to evaluate and confirm the difference between the RIKEN simulator and the actual A64FX.**
- **Target: A64FX test chip**
- **Compiler: Fujitsu compiler prototype in 2019/03**
- **Evaluated program:**
  - Arithmetic pipeline: various kernel loops
  - L2 cache throughput: Stream benchmark for L2 size
  - Memory throughput: Stream benchmark for over L2 size

Evaluation of kernel loops

- **A total of 28 types of kernels that repeat the same operation are evaluated for data that fits on L1 cache.**
- **In the real machine, iter is repeated 1,000,000 times, but in the simulator it is 1000 times. It is because the simulator is very slow, but the timer is enough accurate.**

```
subroutine calc01_add_r8(n, iter, dist, y, x1,
x2)
  real*8 y(n), x1(n), x2(n)
  integer n, iter, i, j, dist

  do j = 1, iter
     do i = 1, n
        y(i) = x1(i) + x2(i)
     end do
  enddo

end subroutine calc01_add_r8
```

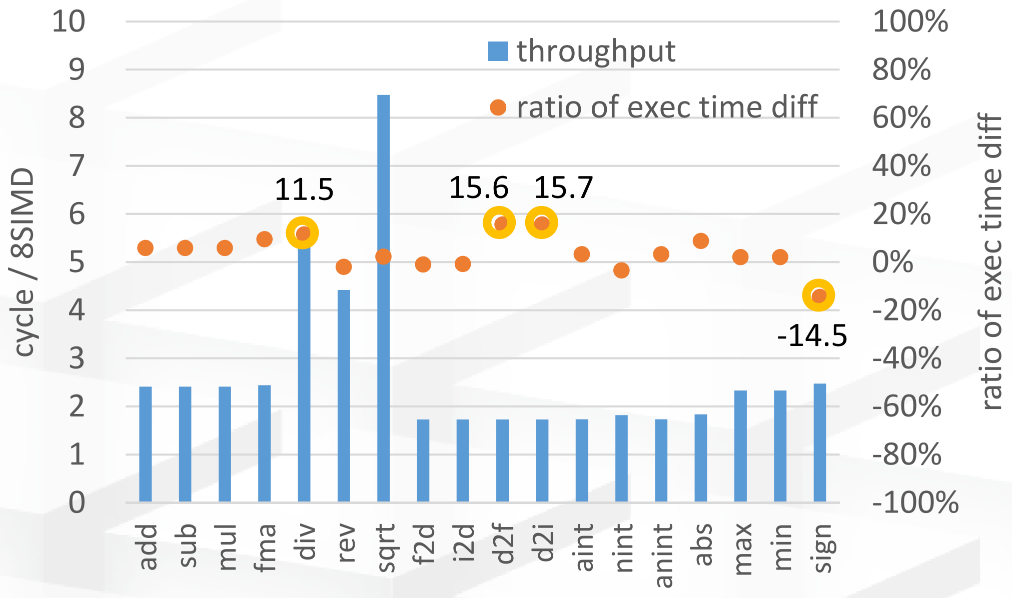
List of 28 kernel loops

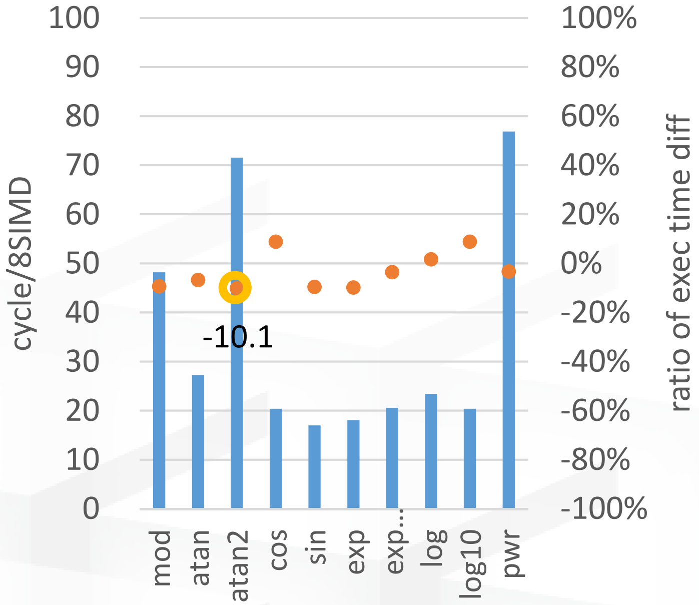
Name	Type	Size	Statement
Basic Arithmetic			
Add	Addition	2048	y(i) = x1(i) + x2(i)
Sub	Subtraction	2048	y(i) = x1(i) - x2(i)
Mul	Multiplication	2048	y(i) = x1(i) * x2(i)
Fma	Sum of products	3072	y(i) = y(i) + c0 * x1(i)
Div	Division	2048	y(i) = x1(i) / x2(i)
Rev	Reciprocal	3072	y(i) = 1 / x1(i)
Sqrt	Square root	3072	y(i) = sqrt(x1(i))
Type Conversion			
F2d	Float to double	4096	y_r8(i) = dble(x1_r4(i))
I2d	Integer to double	4096	y_r8(i) = dble(x1_i4(i))
D2f	Double to float	4096	y_r4(i) = real(x1_r8(i))
D2i	Double to integer	4096	y_i4(i) = int(x1_r8(i))
Aint	Aint conversion	3072	y_r8(i) = aint(x1_r8(i))
Nint	Nint conversion	4096	y_i4(i) = nint(x1_r8(i))
Anint	Anint conversion	3072	y_r8(i)=anint(x1_r8(i))

Name	Type	Size	Statement
Numeric Function			
Abs	Absolute value	3072	y(i) = abs(x1(i))
Max	Maximum value	2048	y(i) = max(x1(i),x2(i))
Min	Minimum value	2048	y(i) = min(x1(i),x2(i))
Mod	Remainder	2048	y(i) = mod(x1(i),x2(i))
Sign	Sign	2048	y(i) = sign(x1(i))
Mathematical Function			
Atan	Atan	3072	y(i) = atan(x1(i))
Atan2	Atan2	2048	y(i) = atan2(x1(i),x2(i))
Cos	Cos	3072	y(i) = cos(x1(i))
Sin	Sin	3072	y(i) = sin(x1(i))
Exp	Exp	3072	y(i) = exp(x1(i))
Exp10	Exp10	3072	y(i) = exp10(x1(i))
Log	Log	3072	y(i) = log(x1(i))
Log10	Log10	3072	y(i) = log10(x1(i))
Pwr	Power	2048	y(i) = x1(i)**x2(i)

Results of kernel loop

- In 23/28 kernels (80%), the difference in execution time between the RIKEN simulator and the A64FX is 10% or less.
- The average of absolute difference is 6.6%, and the largest is 15.7%, which is considered to be enough for evaluation.
- The difference in d2f and d2i seems to be because merge in write buffer is not implemented in RIKEN simulator.

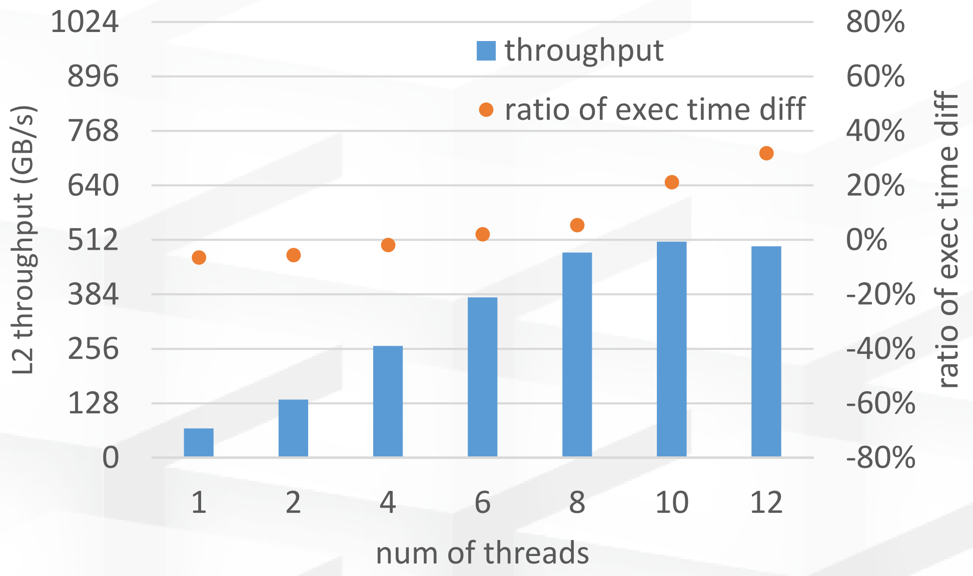




http://arxiv.org/abs/1904.06451 16 Apr 2019 cs.DC
Evaluation of the RIKEN Post-K Processor Simulator

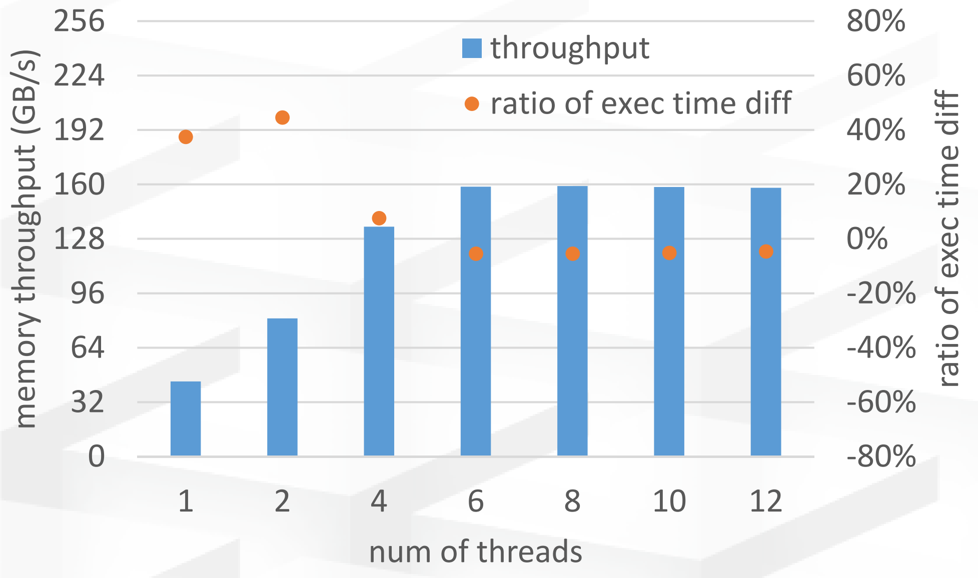
Results of L2 Stream benchmark

- **Measure the total L2 throughput by changing the number of threads from 1 to 12 using software prefetch.**
- **In the RIKEN simulator, the throughput is saturated with 8 threads, while in the A64FX test chip, the throughput has been improved to 12 threads.**
- **The difference is large with 10 threads or more.**
  - ✓ We plan to service requests from each core fairly in L2.
  - ✓ We consider improving bus performance between L1/L2.



Results of Memory Stream benchmark

- **Measure the total memory throughput by changing the number of threads from 1 to 12 using hardware prefetch.**
- **Up to 6 threads achieve scalable throughput, and above that achieves maximum performance.**
- **There is a large difference in execution time between the RIKEN simulator and the A64FX test chip in 1-2 threads.**
  - ✓ Since hardware prefetch for store access is not implemented in the RIKEN simulator, it will be reevaluated after implementation.



Memory access in Stream benchmark

- **Stream triad**

```
for (i=0; i<N; i++) {
    y[i] = x1[i] + c * x2[i];                    2 read 1 store
}
```

- **Memory access**

  Since the store instruction writes to part of the cache line, it is necessary to read the cache line from memory before writing to maintain cache line consistency. Therefore, the actual memory access is 3 read 1.

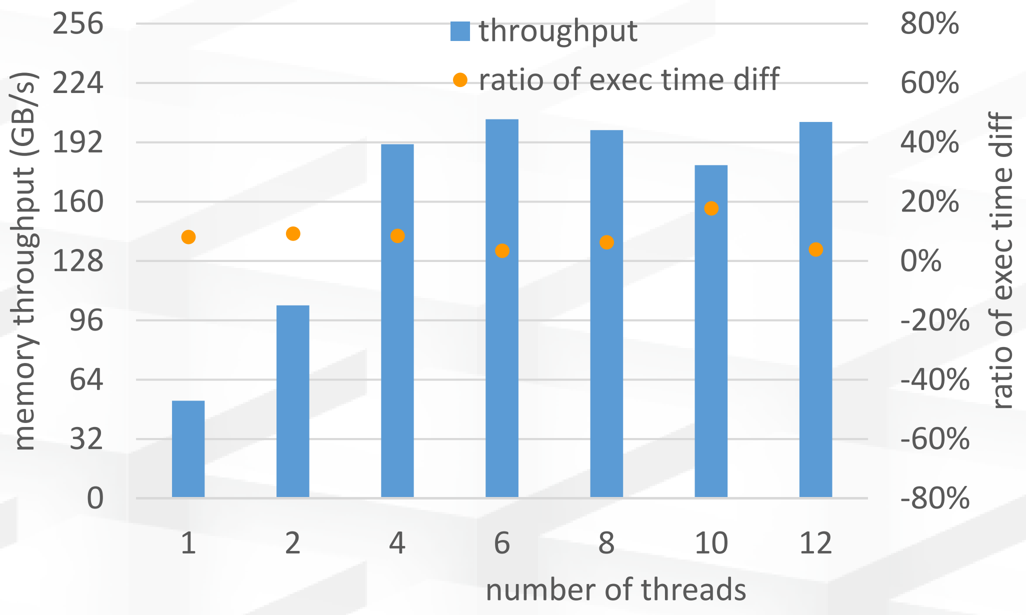
  Therefore, when Stream throughput is 150 GB/s, the memory throughput achieves 200 GB/s.

- **ZFILL Optimization**

  However, when writing to all cache lines as in Stream, it is not necessary to preload from memory. Therefore, the Fujitsu compiler provides the ZFILL option. ZFILL uses a 'DC ZVA' instruction that zero-fills a cache line. As a result, the A64FX test chip achieves 200 GB/s in Stream throughput.

Results of ZFILL optimization

- **The RIKEN simulator also implemented the DC ZVA instruction, and evaluated the ZFILL optimization using software prefetch with 10 times larger data than previous evaluation.**
  - ✓ The RIKEN simulator also confirmed Stream throughput of 200 GB/s, which is almost same as the throughput of the A64FX test chip. (currently continue to test the effect of prefetch distance)
  - ✓ When the number of threads is large, the values are somewhat different. We plan re-evaluation after implementing fairness access in L2 cache.



Conclusion

- We developed a RIKEN simulator that can perform cycle-level processor simulation as an evaluation environment until the Post-K system can be used.
- From the evaluation with Kernel loop, L2 Stream, Memory Stream, many results showed that the differences between the RIKEN simulator and the A64FX test chip are about 10% or less.
- There were some cases where the difference was large, but we will continue to develop RIKEN simulator.
- In the future, we plan to evaluate application kernels those are closer to the actual application.

Outline

- Introduction of RIKEN simulator
  - ✓ Accuracy Improvement of Memory System Simulation for the Post-K gem5 simulator
- Optimization point for SVE
  - ✓ SVE vectorization
  - ✓ Software pipeline
  - ✓ …

SVE vectorization

- Vectorization by SVE is the key to get high performance.
- Not yet summarize how to vectorization if the software cannot be vectorization, but the key is to keep compiler friendly description (it may be required rewriting programs).
  - ✓ Simple consecutive loop
  - ✓ Not element wise description, but loop description
- Notes: Following example is only checked by Fujitsu compiler, and how to optimization is heavily depend on the compiler and the version.

A Pitfall

```
void mycopy(double *x, double *y, int n)
{
  int i;

  for (i = 0; i < n; i++)
    y[i] = x[i];
}
```

It is very simple loop, but it cannot be vectorized
Because *x and *y may be overlap.

→

```
void mycopy_with_restrict(double * restrict x, double * restrict y, int n)
{
  int i;

  for (i = 0; i < n; i++)
    y[i] = x[i];
}
```

Can be vectorized with restrict attribute.

Simple Linpack

```
m = n % 4;
for ( i = 0; i < m; i++ ) {
    dy[i] = dy[i] + da * dx[i];
}
for ( i = m; i < n; i = i + 4 ) {
    dy[i  ] = dy[i  ] + da * dx[i  ];
    dy[i+1] = dy[i+1] + da * dx[i+1];
    dy[i+2] = dy[i+2] + da * dx[i+2];
    dy[i+3] = dy[i+3] + da * dx[i+3];
}
```

In daxpy code, there is a hand unrolled code !!
And it cannot be vectorized.

→

```
for ( i = 0; i < n; i++ )  {
    dy[i] = dy[i] + da * dx[i];
}
```

Simple loop will be vectorized automatically.

Simple Linpack(2)

```
for ( j = k+1; j <= n; j++ ) {
    t = a[l-1+(j-1)*lda];
    if ( l != k ) {
      a[l-1+(j-1)*lda] = a[k-1+(j-1)*lda];
      a[k-1+(j-1)*lda] = t;
    }
    daxpy ( n-k, t, a+k+(k-1)*lda, a+k+(j-1)*lda);
}
…
void daxpy ( int n, double da, double dx[], double dy[restrict])


→

void daxpy ( int n, double da, double dx[restrict], double dy[restrict])
```

It uses 1d vector not 2d array, so compiler cannot detect independence of two vector.

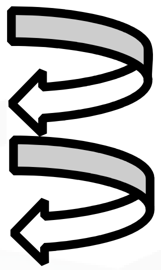
Compiler message for optimize

- **In Fujitsu compiler, you can check whether the loop is vectorized or not by compiler messages.**

```
<<< Loop-information Start >>>
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING
<<<    PREFETCH(HARD) Expected by compiler :
<<<      x2, x1, y
<<< Loop-information  End >>>
```

software pipeline

- Software pipeline was the key optimization for in-order processor, but even for out-of-order processor, since o3 resources are limited and for the loops of large (or not small) body, instructions of next loop cannot be entered to instruction queue, software pipeline is effective.
- In fujitsu compiler –Kfast includes swp,unroll,SVE
  - ✓ -Kfast,noswp,nounroll,NOSVE
  - ✓ -Kfast,noswp,nounroll
  - ✓ -Kfast



Effect of SVE

Effect of software pipeline

Loop split

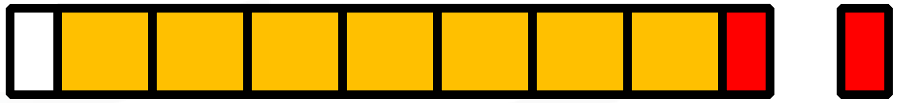
- **Software pipeline requires many registers to keep values between loops, but it caused register spill in large loop body or many loop overlapping.**
  - ✓ In K and FX100, 128 vector registers support it, while arm SVE has 32 vector registers.
  - ✓ Loop splitting is one of optimizations for loop with large body.
  - ✓ Fujitsu compiler will support auto loop splitting, but currently user should manually specify the loop split point.
    - ✓ #pragma statement fission_point
    - ✓ -Kocl

fdiv and fsqrt

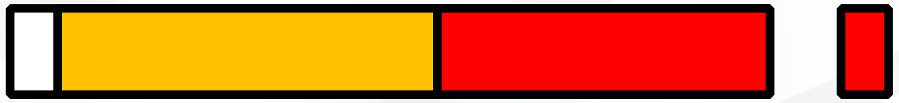
- fdiv is an non-pipeline instruction with long latency (depends on vector length and element size, for example, more than 100 cycles for 8 double SIMD).
- frecpe/frecps are pipeline instruction for reciprocal operaiotns with same latency of fma.
- In fujitsu compiler with –Kfast, fdiv is inlined with frecpe/frecps and equivalent to 12 flop for double, and 7 flop for float.
- fsqrt is inlined with frsqrte/frsqrts and equivalent to 18 flop for double, and 8 flop for float.

Data alignment

- **Using wide SIMD, the alignment of cache access becomes more important in general.**
- For example, cache line size = 64 bytes, and consecutive access by 8byte access. Even if the vector is unaligned, unaligned access is only 1/8.

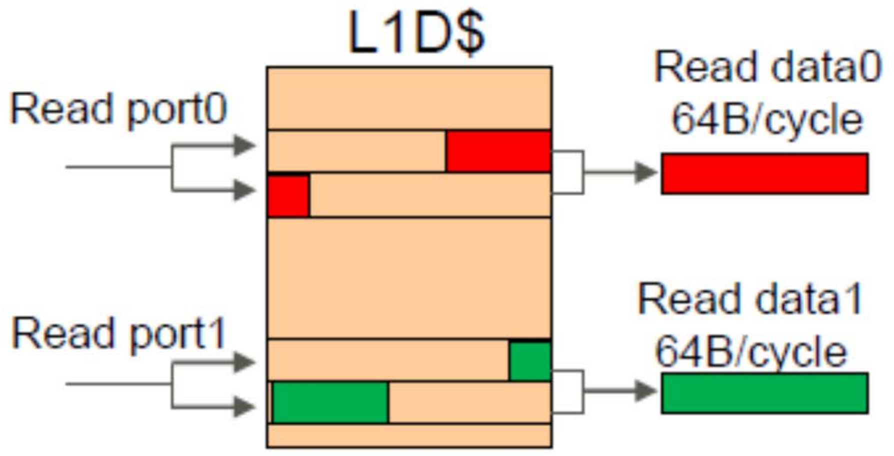


- For the 256bit SIMD case, ½ access becomes unalined.



A64FX L1 cache throughput

- L1 cache throughput maximizes core performance
  - Sustained throughput for 512-bit wide SIMD load
    - An unaligned SIMD load crossing cache line keeps the same throughput



From slides of hotchips 2018

- In gem5, unaligned access divided to two cache access, but gm5-riken supports the behavior of A64FX.

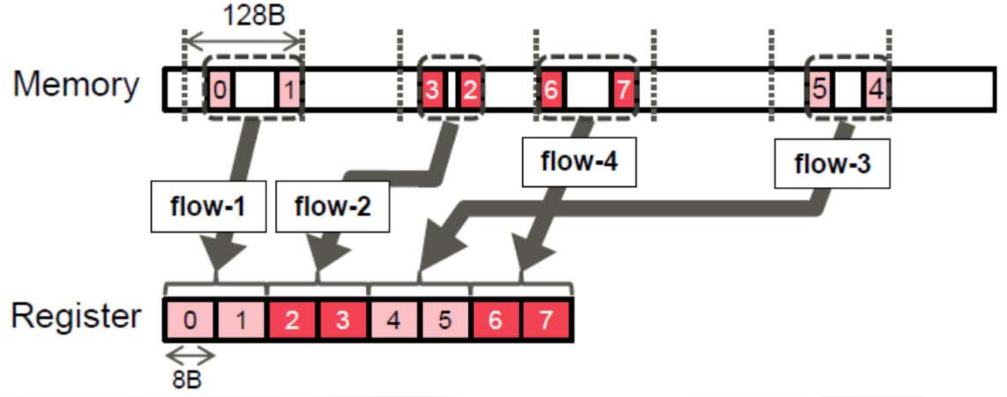
Gather load

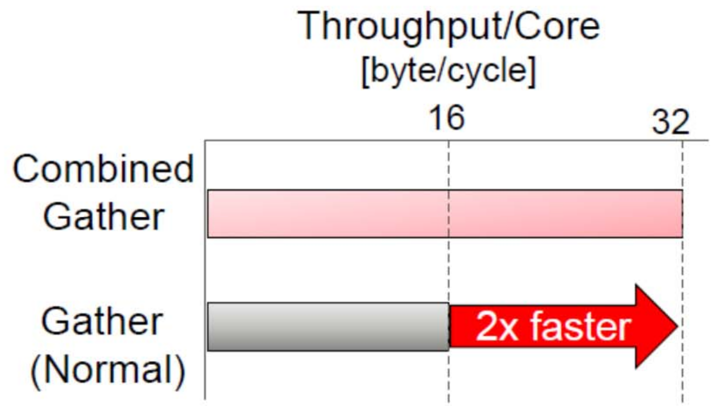
- The throughput of gather load is depend on each processor's microarchitecture.
- In gem5, gather load is implemented by micro operations that access each element.
  - ✓ For example, 512bit SIMD with 8 byte element, gather load takes 8 memory access, and the throughput is limited in 8bytes/cycle, while that of contiguous load is 64bytes/cycle.

A64FX Gather load

- "Combined Gather" mechanism increasing gather throughput
  - Gather processing is important for real HPC applications
  - A64FX introduces "Combined Gather" mechanism enabling to return up to two consecutive elements in a "128-byte aligned block" simultaneously

< Combined Gather >





From slides of hotchips 2018

Gather load vs predicate

- For following program that loads selectively, gather load or predicate load are available, and which is used is compiler-dependent.

```
for (i=0; i<SIZE; i+=2) {
     c[i] = a[i] + b[i];
   }
```

In Fujitsu compiler generates gather load

```
for (i=0; i<SIZE; i++) {
     if (i%2 == 0)
       c[i] = a[i] + b[i];
   }
```

In Fujitsu compiler generates predicate

Vector length

- **In fujitsu compiler –Ksimd_regsize=512 by default**
  - ✓ -Kfast
  - ✓ -Kfast,-Ksimd_regsize=agnostic

Gem5 can change vector length by --arm-sve-vl, and gem5-o3 also can change by –v option.

-v 1024 (same as --arm-sve-vl=8): 1024bit

-v 512 (same as --arm-sve-vl=4): 512bit

-v 256 (same as --arm-sve-vl=2) : 256bit

-v 128 (same as --arm-sve-vl=1) : 128bit