ABC of modern HPC for LQCD

1. Modern CPU Architecture for LQCD

Issaku Kanamori (R-CCS)

Dec. 10, 2021 @ R-CCS

1. Motivation
2. Data access: memory hierarchy, cache, prefetch
3. SIMD: data layout
4. Pipeline
5. Summary, or tips for performance tunings

Motivation

- Lattice QCD simulation uses large computational resources
  performance is also important
  - ambitious targets   large lattice, fine lattice, high stats., light mass, etc.
  - competition   cross check is also important, but...
  - finite time, money, power   efficient use of computer
- Optimization of the simulation code is needed
  - not everybody's job: we also need people to analyze the data
  - but it's better to know what is behind the optimization
    - which machine? is the performance reasonable? how much it can be improved? what's in the discussion for the next machine?

References

- A64FX: https://github.com/fujitsu/A64FX

  especally, A64FX_Microarchitecture_Manual_en_1.6.pdf there

- textbook of computer architecture:

  David A. Patterson and John L. Hennessy, “Computer Organization and Design”

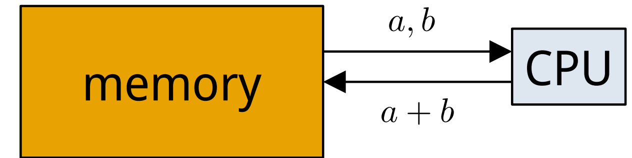
  my version is a Japanese translation of the 5th edition (2014, Elsevier)

  コンピュータの構成と設計第 5 版 上・下 日経 BP 社 （ 2014 ）

  the latest is the 6th edition (2020, Morgan Kaufmann) [Japanese translation: 2021]

Data access

- time scale of the CPU core: cycle ~ 10^{-9} sec.
  Fugaku (2GHz): $1/(2\text{x}10^9) = 0.5\text{x}10^{-9}$ sec.



- main memory: O(100) cycles to access
  arithmetic is O(1) cycle, memory is too slow (high latency)
  Fugaku: 271-289 cycles
  theoretical peak of single core of Fugaku: 8640 floating point operations in 270 cycles
  8 (SIMD) x 2 (FMA) x 2 (pipeline) x 270 (cycle) = 8640
  Spending ~1/300 of computation time without computing? It is a big loss of opportunity!
- cache: "faster" (but smaller) memory between the CPU and the main memory
  CPU, L1 cache, L2 cache, L3 cache, main memory
  last level cache
  Fugaku has no L3 cache so L2 is the last level cache

Cache memory

- L2 cache (and L3 cache): lower latency, recently used data is left
- L1 cache: much lower latency (but size is much smaller)

Fugaku

	main memory	L2 cache	L1 cache
latency (cycle)	271-289	45-56	5-11
size	32 GiB/cpu	8 MiB/CMG (32 MiB/cpu)	16KiB/core (768 KiB/cpu)

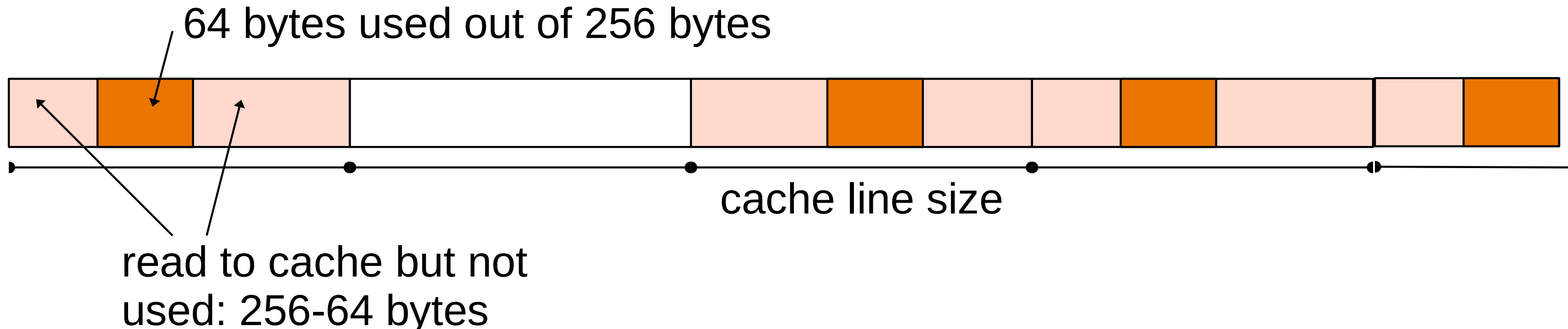
- to hide the latency, data can be prefetched
  hardware prefetch,
  software prefetch (by compiler), manual prefetch

```
// sketch of L2 prefetch
for(i=0; i<N; i++){
  a[i]+=b[i];
  load (a[i+10], b[i+10]) to L2 (prefetch)
}
```

- (this is to see the typical time scale; prefetch tunings should be the last step of the optimization)

Cache line size

- size of data is read to cache simultaneously: cache line size
  - Fugaku: 256 bytes (= 4 x 512 bits)
    continuous 256 bytes data is simultaneously read to the cache
  - Discontinuous memory access wastes the memory band width
    - 4 x 64 bytes data from single cache line: 256 bytes of read
    - random access of 4 x 64 bytes date: 4 x 256 bytes of read



Memory band width and B/F

- Flop: count multiplication and summation (does not count $\times(1,-1,i,-i)$)

  ex: $(a+ib)(x+iy) = ax - by + i(ay+bx)$ 6 Flop
  $ax - by : 2 \text{ mult and } 1 \text{ add}$
  $ay + bx : 2 \text{ mult and } 1 \text{ add}$

  $(c+id) += (a+ib)(x+iy)$ 8 Flop
  $c += ay + by : 2 \text{ mult and } 2 \text{ add}$
  $d += ay + bx : 2 \text{ mult and } 2 \text{ add}$

  Memory access: 1 double precision number is 8 byte

  ex: $(c+id) += (a+ib)(x+iy)$ load: 48 bytes (a,b,c,d,x,y), store 16 bytes (c,d)

  64 bytes of memory access for 8 Flop

  byte-per-flop (B/F) is 8 cf. inner product in CG solver: load of (a,b,x,y), B/F=4

- B/F of recent machines is <=0.5 (sometimes <<0.5)

  memory bandwidth can be a bottleneck

  Fugaku: 0.33, Intel KNL: ~0.1

cf. FLOPS:
Flop per second

B/F in LQCD QCD is memory bandwidth determining

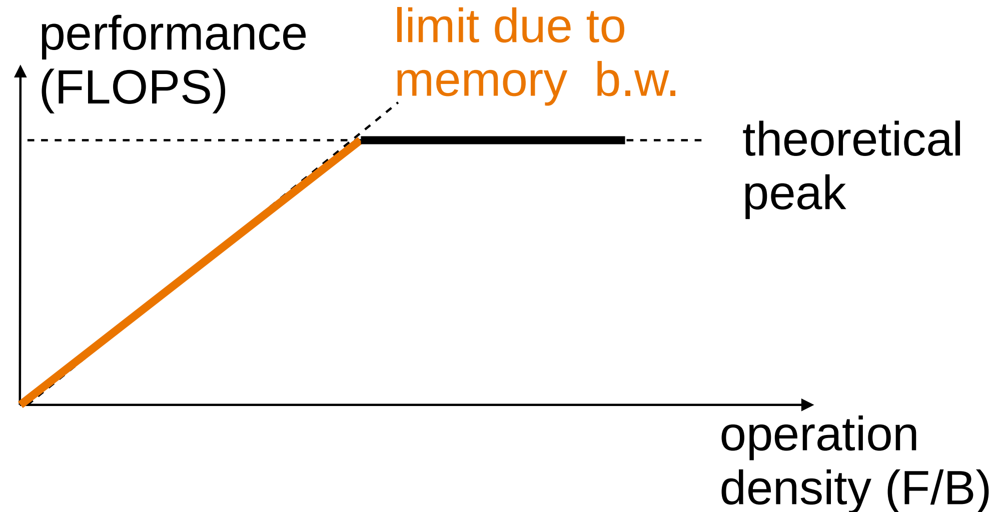
It depends on the details of the implementation

- Wilson Dirac operator: B/F = 1.12 (Dirac rep.)
  - 1368 Flop / site
  - 1536 Byte / site
- Clover Dirac operator: B/F = 0.94 (Dirac rep.)
  - 1944 Flop / site
  - 1824 Byte / site
- Domainwall Dirac operator: B/F = 0.72 (Shamir, Ls=8)
  - 11520 Flop / site
  - 8256 Byte / site

Fugaku: B/F=0.33
KNL: B/F ~ 0.15
significantly smaller than multiplication of the Dirac operatoar.
since the peak performance /node is roughly the same, a naive estimate of performace/node of LQCD is
Fugaku ~ 2 x KNL

Roofline model

- from the B/F ratio, one can predict the performance (if the memory access is the bottle neck)
- ex.
  Wilson mult [Fugaku]
  (0.33/1.12) x 3072
  = 900 GFlops /node



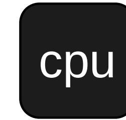
SIMD

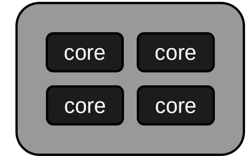
- To increase the performance of one processor
  - increase frequency : stopped at 2-3 GHz heat
  - increase number of cores (~ bundle several small CPUs into 1 processor):

    typical laptop: 2 cores, Fugaku: 48 (+2 or 4) cores, KNL: 64-68 cores

    memory/cache coherence to keep the consistency of the data in memory

    thread paralelization





  - increase the data size processed simultaneously

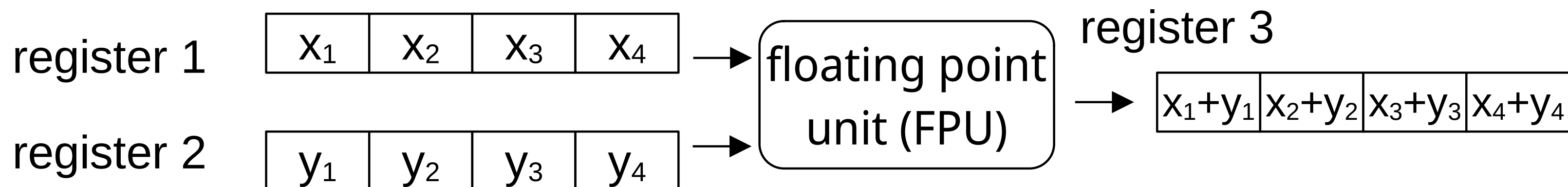
    Single Instruction Multiple Data (SIMD)

	x
+	y
	$x+y$

	x_1	x_2	x_3	x_4
+	y_1	y_2	y_3	y_4
	x_1+y_1	x_2+y_2	x_3+y_3	x_4+y_4

applies the same instruction (+) to several numbers at once
Fugaku: (512bit)
8 double / 16 float / 32 half prec.

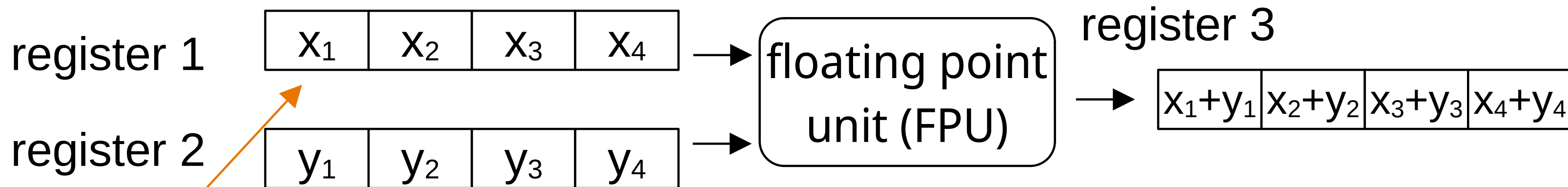
SIMD register



register:

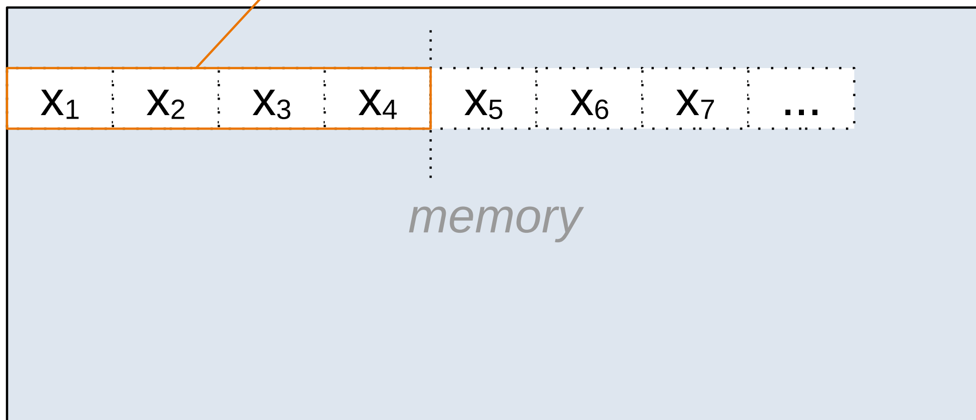
- a piece of small memory in the core
- input/output of the FPU must be data in register (some can be in memory with Intel AVX-512)
    1. load data from memory to register
    2. arithmetics in FPU
    3. store data from register to memory
- usually holds only 1 variable
- SIMD register: can holds several variables
- Fugaku has 32 registers for floating point operation for each core

Data layout for SIMD

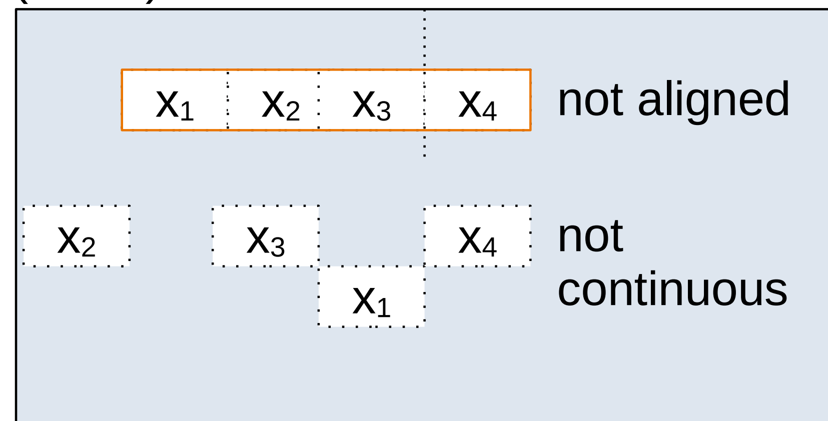


1. **load data from memory to register**
2. arithmetics in FPU
3. **store data from register to memory**

efficient
- continuous in the memory
- address is multiple of the register size (aligned)



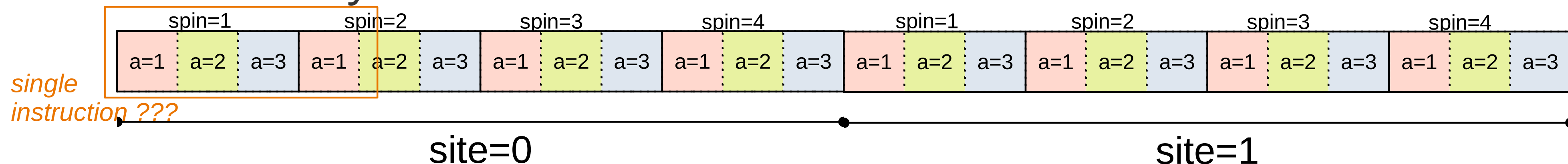
(much) less efficient



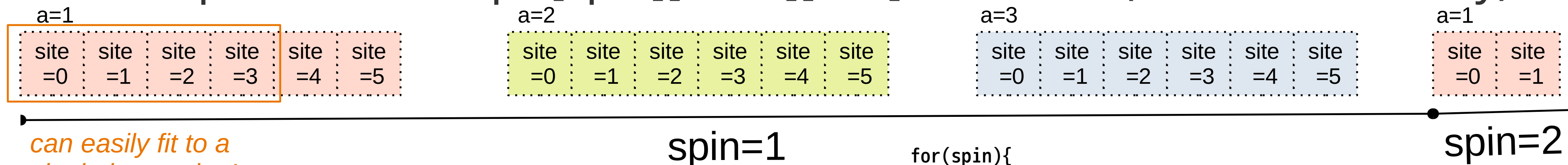
Data layout for QCD

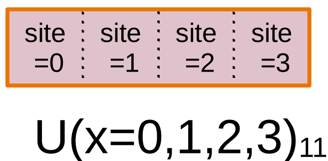
$$\psi'(x)^\alpha_a = U_\mu(x)_{ab}\psi(x)^\alpha_b$$

- std::complex<double> psi[site][spin][color]; // not good (array of structure)
  in the meory:

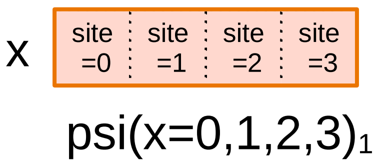


- std::complex<double> psi[spin][color][site]; // better (structure of array)





x



$U(x=0,1,2,3)_{11}$ $psi(x=0,1,2,3)_1$

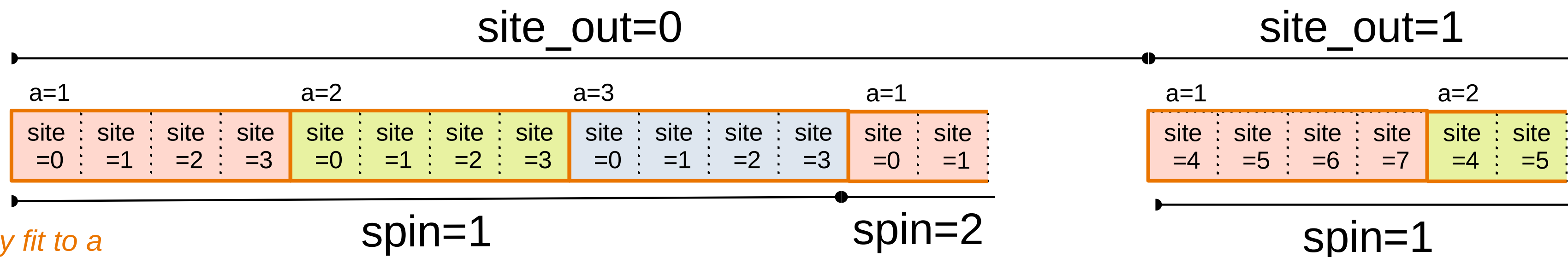
```
for(spin){
  for(b=1,2,3){
    for(x){ psi'[spin][b][x] = 0; // SIMD vectorized}
    for(a=1,2,3){
      for(x){  // SIMD vectorized
         psi'[spin][b][x]+=U[b][a]*psi[spin][a][x];
}}}
```

Data layout for QCD

$$\psi'(x)^\alpha_a = U_\mu(x)_{ab}\psi(x)^\alpha_b$$

- std::complex<double> psi[site_out][spin][color][site_in];   // best

site = (SIMD length) * site_out + site_in   (array of structure of array)



can easily fit to a single instruction!

site =0	site =1	site =2	site =3

x

site =0	site =1	site =2	site =3

$U(x=0,1,2,3)_{11}$ $psi(x=0,1,2,3)_{1}$

```
for(site_out){
  for(spin){
    for(b=1,2,3){
      for(site_in){ psi'[spin][b][x] = 0; // SIMD vectorized  }
      for(a=1,2,3){
         for(site_in){  // SIMD vectorized
            psi'[spin][b][x]+=U[b][a]*psi[spin][a][x];
}}}}
```

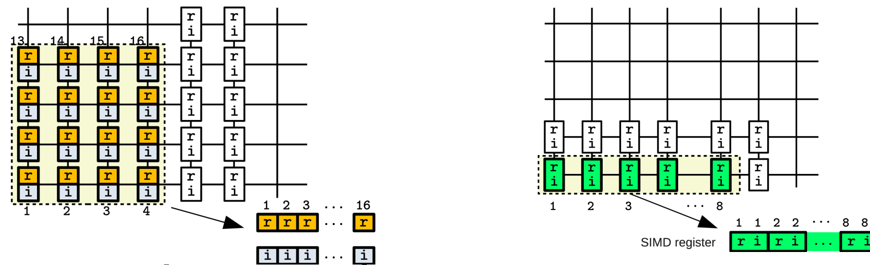
- local volume must be multiple of the SIMD length
- above assumes (re,im) of complex number is the most inner

SIMD width

- Intel
  - SSE, SSE2, SSE3 : 128 bits (= double x 2)
  - AVX, AVX2 : 256 bits (= double x 4)
  - AVX-512 : 512 bits (= double x8)
  - different instructions for each SIMD width
- A64FX (Fugaku)
  - Scalable Vector Extension with 512 bits
  - SVE: the same instruction work for different SVE length (up to 2048 bits)
    mask operations for each lane in the SIMD registor
  - (using instructions for fixed SVE length can be faster... cf. Grid with gcc)

Miscellaneous about SIMD

- Fused Multiple and Add (FMA) is available: ax+y in 1 instruction
- Optimized QCD code usually uses intrinsics for SIMD variables
- Compiler may automatically generate SIMDized binary, if the data structure is proper (& simple) and the compiler clever enough
- on Fugaku, putting the real part and imaginary part of complex numbers to different SIMD registers (left) is faster

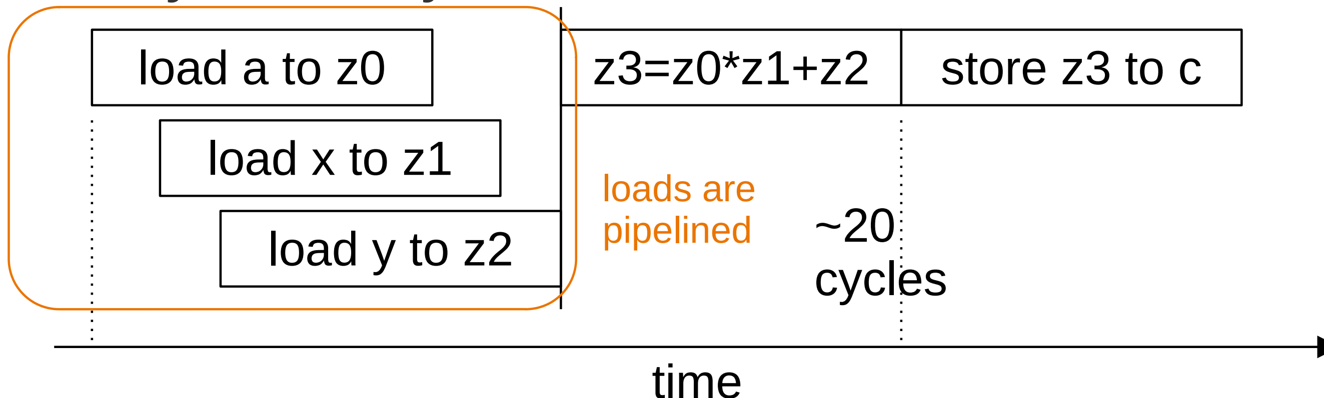


- Way of packing site differs code to code

  QWS: 1-dim, Bridge++: 2-dim tiling, Grid: to sub domains

Latency and pipeline

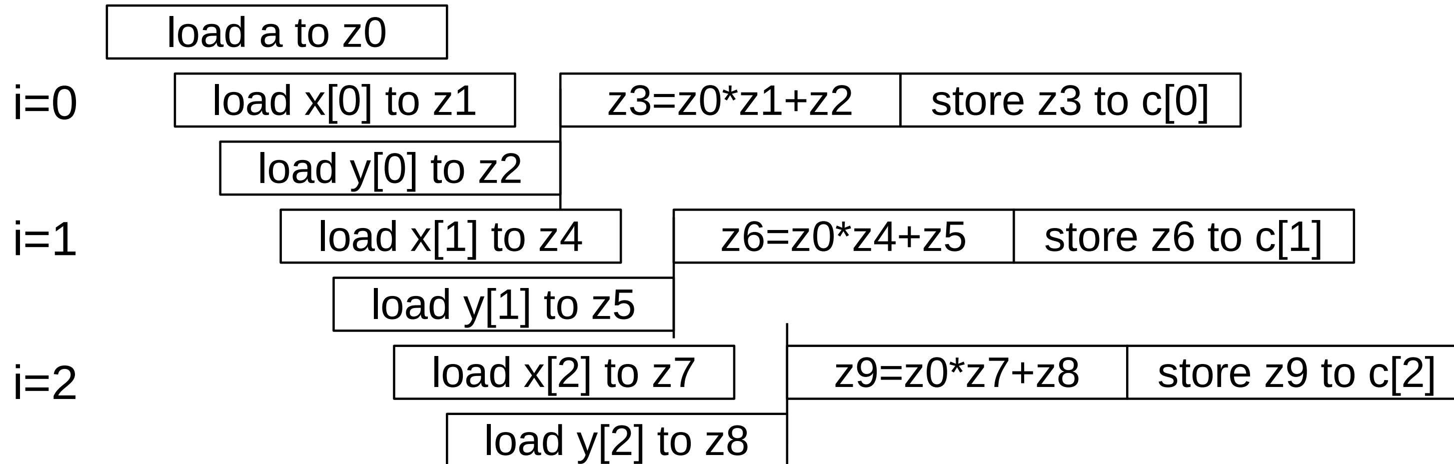
- each instruction takes several cycles (latency) [Fugaku]
  1. load data from memory to register [8-11 cycles from L1 cache] intel 8
  2. do some arithmetics [9 cycles for FMADD (ax+y)] intel 4-6
  3. store data to memory [NA]
- c=ax+y >~ 20 cycles



2(FMA) x 8(SIMD) FLOP in 19 cycles
(not 8x3+9 =33)

Pipeline

- for(i) { c[i]=a*x[i]+y[i]; }



* the figure assumes load can be issued simultaneously with FMA or store so 1 FMA in 2 cycles

1 set: load x 2, FMA x 1, store x 1
if there are enough registers, 1 FMA in 4 cycles*

Enough Registers?

- each register is occupied ~20 cycles (registers for c[i] are occupied less, though)
- each set uses 3 registers (x[i], y[i], c[i])
- If FMA is issued every 2 cycles, we need 10 sets (= 30 registers)
- If FMA is issued every cycle, we need 20 sets (= 60 registers)
- Fugaku: 32 registers so every 2 cycles at most for this loop
- More complicated loop body: more register hungry

  the complicated loop body itself may cause making pipeline difficult

  (loop fission technique)

```
for(i){
  f(i);
  g(i);
}
```

```
// loop fission
for(i){ f(i);  }
for(i){ g(i);  }
```

Register spill

- limited number of registers:

```
x=0.0;  // register is assigned for x
a1=b1*a0;  a2=b2*a1;   // several registers for these varialbes
....   // Oh, no register is left;
```

(store x to memory (L1 cache)) register spill !

```
....
x+=a10; // !!! x is not in the register,
```

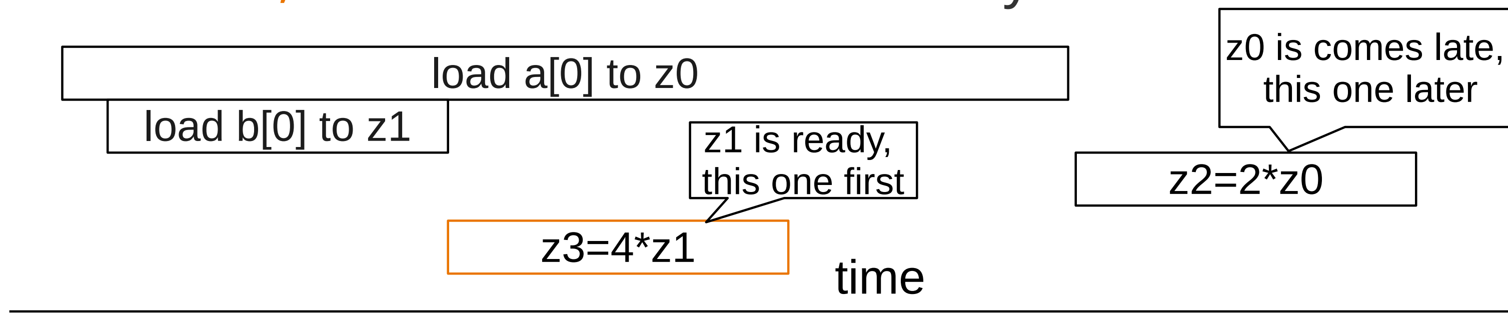
need to load x from the memory

- even loading from L1 cache, it causes a significant penalty in the performance

Out of order

- order of execution can be reordered

```
z0=a[0];
z1=b[0];
z2=2*z0;  // need to wait till z0 is ready
z3=4*z1;  // need to wait till z1 is ready
```



Summary, or Tips for tuning: 1

- Data access: bandwidth
  - LQCD is memory bandwidth limited application
    B/F of Dirac op. mult ~ 1 >> typical CPUs
    (B/F of Fugaku ~ 0.3 is rather large nowadays)
  - Once data is loaded to the cache, reuse it as much as possible
    loop tiling, cache blocking
- Data access: latency
  - Main memory is far away: O(100) cycles
  - data prefetch

Summary, or Tips for tuning: 2

- SIMD
  - Without SIMD, the theoretical peak performance would be 1/8 (double prec.) or 1/16 (single prec.)
  - use SIMD friendly data layout: array of structure of array
  - single perc. floating operation is x2 faster (half prec. x4 faster) than double precision. Use mixed prec. algorithm

Summary, or Tips for tuning: 3

- Out-of-Order, Pipeline etc.
  - Each instruction has latency, ~10 cycles on Fugaku (larger than intel)
  - Execution order can be reordered to reduce waiting time of data
  - Loop fission (or fusion), loop unrolling for pipeline execution