How to get peak FLOPS (CPU) — What I wish I knew when I was twenty about CPU —

Kenjiro Taura

Contents

Contents

What you need to know to get a nearly peak FLOPS

- so you now know how to use multicores and SIMD instructions
- they are two key elements to get a nearly peak FLOPS
- the last key element: Instruction Level Parallelism (ILP) of superscalar processors

Contents

An endeavor to nearly peak FLOPS

- let's run the simplest code you can think of

```
#if __AVX512F__
const int vwidth = 64;
#elif __AVX__
const int vwidth = 32;
#else
#error "you'd better have a better machine"
#endif

const int valign = sizeof(float);
typedef float floatv __attribute__((vector_size(vwidth),aligned(valign)));
/* SIMD lanes */
const int L = sizeof(floatv) / sizeof(float);
```

```
  floatv a, x, c;
  for (i = 0; i < n; i++) {
    x = a * x + c;
  }
```

- the code performs L × n FMAs and almost nothing else

Notes on experiments

- the source code for the following experiments is in `06axpb` directory
- the computation is trivial but the measurement part is Linux-specific
  - `perf_event_open` to get *CPU clocks* (not *reference clocks*)
  - `clock_gettime` to get time in nano second resolution
- it will compile on MacOS too, but the results are incomplete and less accurate
  - CPU clocks not available
  - gettimeofday (micro second granularity) substitutes for `clock_gettime` (nano second granularity)

Notes on experiments

- on Linux, you need to allow user processes to get performance events by

```
$ sudo sysctl -w kernel.perf_event_paranoid=-1
```

- exact results depend on the CPU microarchitecture and ISA

```
$ cat /proc/cpuinfo
```

  and google the model name (e.g., "Xeon Gold 6126")
- the following experiments show results on an Skylake X CPU
  - Skylake X is a variant of Skylake supporting AVX-512
  - taulec, login node of the IST cluster, and the `big` partition of IST cluster
- there is a Skylake, which has the same microarchitecture but does not support AVX-512 (laptop CPUs)

Let's run it!

- compile

```
1 $ g++ -o axpb.g++ -O3 -fopenmp -march=native axpb.cc
```

Let's run it!

- compile

```
1 $ g++ -o axpb.g++ -O3 -fopenmp -march=native axpb.cc
```

- and run!

```
1  login000:06axpb$ ./axpb.g++ -a simd
2   algo = simd
3      bs = 1 (cuda block size)
4      c = 1 (the number of variables to update in the inner loop)
5      m = 16 (the number of FP numbers to update)
6      n = 1000000 (the number of times to update each variable)
7      L = 16 (SIMD lanes on the CPU)
8  1403468 nsec
9  3641400 ref clocks
10 4022127 cpu clocks
11
12 1.403468 nsec       for performing x=ax+b for 16 variables once
13 3.641400 ref clocks for performing x=ax+b for 16 variables once
14 4.022127 cpu clocks for performing x=ax+b for 16 variables once
15
16 22.800662 flops/nsec
17 8.787829 flops/ref clock
18 7.955989 flops/cpu clock
19 X[7] = 0.005949
```

Let's run it!

- compile

```
1 $ g++ -o axpb.g++ -O3 -fopenmp -march=native axpb.cc
```

- and run!

```
1  login000:06axpb$ ./axpb.g++ -a simd
2   algo = simd
3      bs = 1 (cuda block size)
4      c = 1 (the number of variables to update in the inner loop)
5      m = 16 (the number of FP numbers to update)
6      n = 1000000 (the number of times to update each variable)
7      L = 16 (SIMD lanes on the CPU)
8  1403468 nsec
9  3641400 ref clocks
10 4022127 cpu clocks
11
12 1.403468 nsec       for performing x=ax+b for 16 variables once
13 3.641400 ref clocks for performing x=ax+b for 16 variables once
14 4.022127 cpu clocks for performing x=ax+b for 16 variables once
15
16 22.800662 flops/nsec
17 8.787829 flops/ref clock
18 7.955989 flops/cpu clock
19 X[7] = 0.005949
```

Let's run it!

- compile

```
1 $ g++ -o axpb.g++ -O3 -fopenmp -march=native axpb.cc
```

- and run!

```
1  login000:06axpb$ ./axpb.g++ -a simd
2   algo = simd
3      bs = 1 (cuda block size)
4      c = 1 (the number of variables to update in the inner loop)
5      m = 16 (the number of FP numbers to update)
6      n = 1000000 (the number of times to update each variable)
7      L = 16 (SIMD lanes on the CPU)
8  1403468 nsec
9  3641400 ref clocks
10 4022127 cpu clocks
11
12 1.403468 nsec       for performing x=ax+b for 16 variables once
13 3.641400 ref clocks for performing x=ax+b for 16 variables once
14 4.022127 cpu clocks for performing x=ax+b for 16 variables once
15
16 22.800662 flops/nsec
17 8.787829 flops/ref clock
18 7.955989 flops/cpu clock
19 X[7] = 0.005949
```

How to investigate

- put a landmark in the assembly code

```
1  asm volatile ("# axpb_simd: ax+b loop begin");
2  for (long i = 0; i < n; i++) {
3    x = a * x + b;
4  }
5  asm volatile ("# axpb_simd: ax+b loop end");
```

How to investigate

- put a landmark in the assembly code

```
1  asm volatile ("# axpb_simd: ax+b loop begin");
2  for (long i = 0; i < n; i++) {
3    x = a * x + b;
4  }
5  asm volatile ("# axpb_simd: ax+b loop end");
```

- compile into assembly

```
1  $ g++ -o axpb.S -S  -O3  -fopenmp -march=native axpb.cc
```

- and see `axpy.S` in your editor

Assembly

```
  # axpy_simd: ax+b loop begin
# 0 "" 2
#NO_APP
  testq   %rdi, %rdi
  jle     .L659
  xorl    %edx, %edx
  .p2align 4,,10
  .p2align 3
.L660:
  addq $1,%rdx
  vfmadd132ps %zmm0,%zmm1,%zmm2
  cmpq %rdx,%rdi
  jne .L660
.L659:
#APP
# 63 "axpy.cc" 1
  # axpy_simd: ax+b loop end
```

- why?

Suspect looping overhead?

- if you suspect the overhead of other instructions, here is an unrolled version that has much fewer overhead instructions
- its performance is identical

```
#pragma GCC optimize("unroll-loops", 8)
long axpy_simd(long n, floatv a,
               floatv* X, floatv c) {
  ...
  for (i = 0; i < n; i++) {
    x = a * x + c;
  }
}
```

⇒

```
1  .L1662:
2    addq $8, %rdx
3    vfmadd132ps %zmm0,%zmm1,%zmm2
4    vfmadd132ps %zmm0,%zmm1,%zmm2
5    cmpq %rdx,%rdi
6    vfmadd132ps %zmm0,%zmm1,%zmm2
7    vfmadd132ps %zmm0,%zmm1,%zmm2
8    vfmadd132ps %zmm0,%zmm1,%zmm2
9    vfmadd132ps %zmm0,%zmm1,%zmm2
10   vfmadd132ps %zmm0,%zmm1,%zmm2
11   vfmadd132ps %zmm0,%zmm1,%zmm2
12   jne .L1662
```

Contents

Latency and throughput

- a (Skylake-X) core can execute *two* `vfmaddps` *instructions every cycle*
- yet, it does *not* mean the result of `vfmaddps` at line 3 below is available in the next cycle for `vfmaddps` at the next line

```
1  .L1662:
2    addq $8, %rdx
3    vfmadd132ps %zmm0,%zmm1,%zmm2
4    vfmadd132ps %zmm0,%zmm1,%zmm2
5    cmpq %rdx,%rdi
6    vfmadd132ps %zmm0,%zmm1,%zmm2
7      ...
8    vfmadd132ps %zmm0,%zmm1,%zmm2
9    jne .L1662
```

Latency and throughput

- a (Skylake-X) core can execute *two* `vfmaddps` *instructions every cycle*
- yet, it does *not* mean the result of `vfmaddps` at line 3 below is available in the next cycle for `vfmaddps` at the next line

```
1  .L1662:
2    addq $8, %rdx
3    vfmadd132ps %zmm0,%zmm1,%zmm2
4    vfmadd132ps %zmm0,%zmm1,%zmm2
5    cmpq %rdx,%rdi
6    vfmadd132ps %zmm0,%zmm1,%zmm2
7      ...
8    vfmadd132ps %zmm0,%zmm1,%zmm2
9    jne .L1662
```

- *what you need to know:*
  - "two `vfmadd132ps` instructions every cycle" refers to the *throughput*
  - each instruction has a specific *latency* (> 1 cycle)

Latencies

instruction	Haswell	Broadwell	Skylake
fp add	3	3	4
fp mul	5	3	4
fp fmadd	5	5	4
typical integer ops	1	1	1
...	...	...	...

Valuable resources for detailed analyses

- Software optimization resources by Agner
  - *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*
  - *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*
- Intel Intrinsics Guide
- Intel Architecture Code Analyzer (later)

Our code in light of latencies

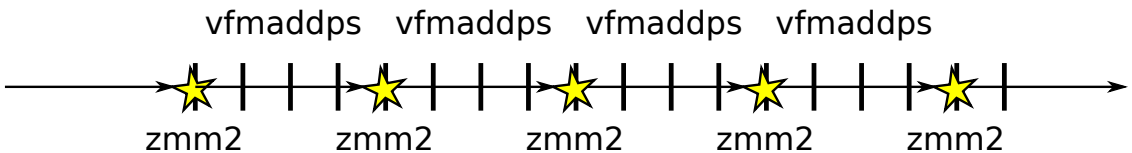
- in our code, a `vfmadd` uses the result of the immediately preceding `vfmadd`
- *that was obvious from the source code too*

```
1  .L1662:
2    addq $8, %rdx
3    vfmadd132ps %zmm0,%zmm1,%zmm2
4    vfmadd132ps %zmm0,%zmm1,%zmm2
5    cmpq %rdx,%rdi
6    vfmadd132ps %zmm0,%zmm1,%zmm2
7      ...
8    vfmadd132ps %zmm0,%zmm1,%zmm2
9    jne .L1662
```

```
1  for (i = 0; i < n; i++) {
2    x = a * x + c;
3  }
```

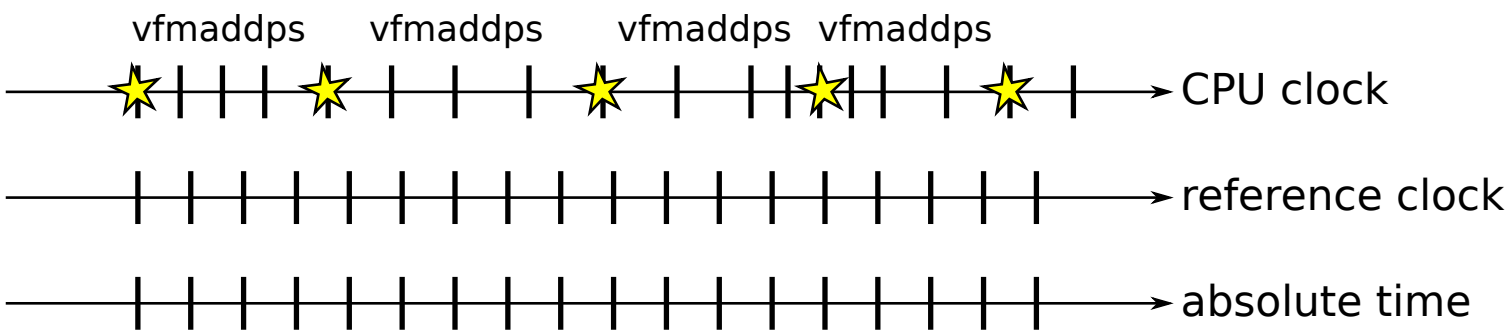
Conclusion:

the loop can't run faster than 4 cycles/iteration



CPU clocks vs. reference clocks

- CPU changes clock frequency depending on the load (DVFS)
- reference clock runs at the same frequency (it is always proportional to the absolute time)
- an instruction takes a specified number of *CPU clocks*, not reference clocks
- the CPU clock is more predictable and thus more convenient for a precise reasoning of the code



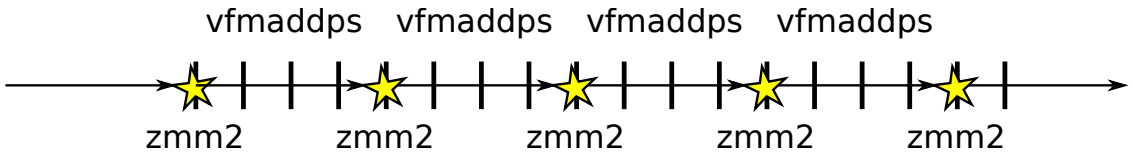
Contents

How to overcome latencies?

- increase parallelism (no other ways)!

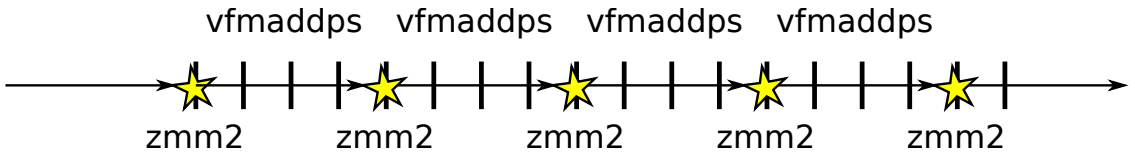
How to overcome latencies?

- increase parallelism (no other ways)!
- you *can't* make a serial chain of computation run faster (change the algorithm if you want to)

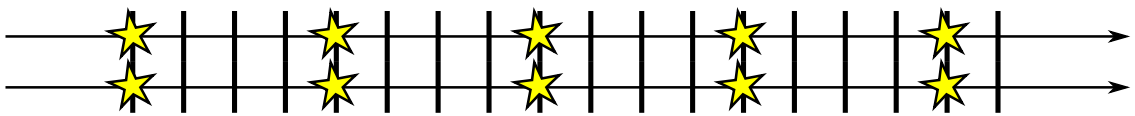


How to overcome latencies?

- increase parallelism (no other ways)!
- you *can't* make a serial chain of computation run faster (change the algorithm if you want to)

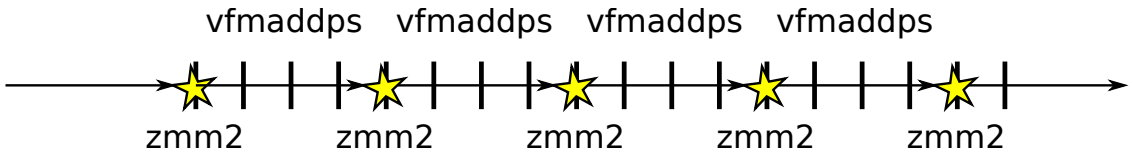


- you *can* only increase *throughput*, by running multiple *independent* chains

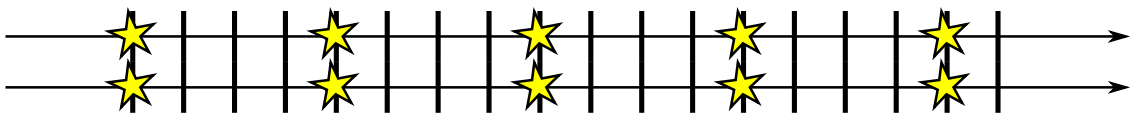


How to overcome latencies?

- increase parallelism (no other ways)!
- you *can't* make a serial chain of computation run faster (change the algorithm if you want to)



- you *can* only increase *throughput*, by running multiple *independent* chains



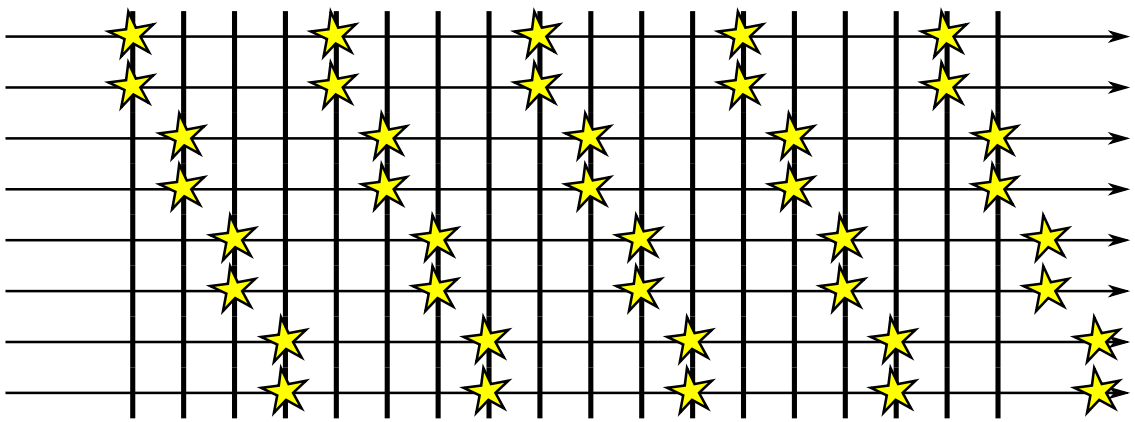
- we expect the following to finish in the same number of cycles as the original one, despite it performs twice as many flops

```
1  for (i = 0; i < n; i++) {
2    x0 = a * x0 + c;
3    x1 = a * x1 + c;
4  }
```

Increase the number of chains further . . .

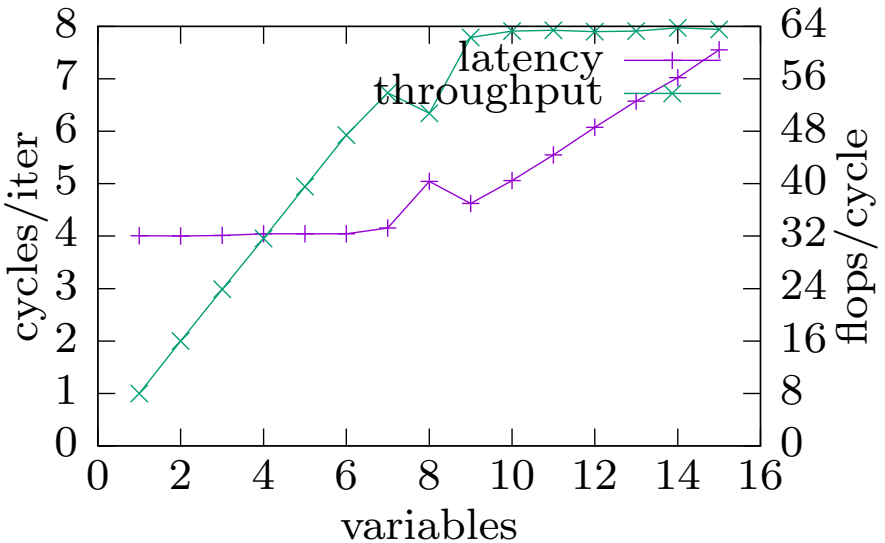
- we expect to reach peak FLOPS with $\geq 2/(1/4) = 8$ chains

```
1  template<int nv>
2  long axpy_simd_c( ... ) {
3    for (long i = 0; i < n; i++) {
4      for (long j = 0; j < nv; j++) {
5        X[j] = a * X[j] + c;
6      } } }
```



Results

a compile-time constant number of variables




```
1  for (i = 0; i < n; i++) {
2    x0 = a * x0 + b;
3    x1 = a * x1 + b;
4      ...
5  }
```

chains	clocks/iter	flops/clock
1	4.010	7.979
2	4.003	15.987
3	4.013	23.916
4	4.043	31.653
5	4.043	39.568
6	4.047	47.439
7	4.157	53.878
8	5.044	50.751
9	4.621	62.314
10	5.057	63.270
11	5.549	63.427
12	6.076	63.194
13	6.573	63.283
14	7.022	63.794
15	7.552	63.558

Contents

Superscalar processors

What you need to know:

Superscalar processors

What you need to know:

- instructions are decoded in the program order,

Superscalar processors

What you need to know:

- instructions are decoded in the program order,
- but the execution orders are not constrained by it *(out of order execution)*

Superscalar processors

What you need to know:

- instructions are decoded in the program order,
- but the execution orders are not constrained by it *(out of order execution)*
- $\Rightarrow$ as a crude approximation, performance is constrained only by

Superscalar processors

What you need to know:

- instructions are decoded in the program order,
- but the execution orders are not constrained by it *(out of order execution)*
- $\Rightarrow$ as a crude approximation, performance is constrained only by
  - *latency:* imposed by *dependencies* between instructions

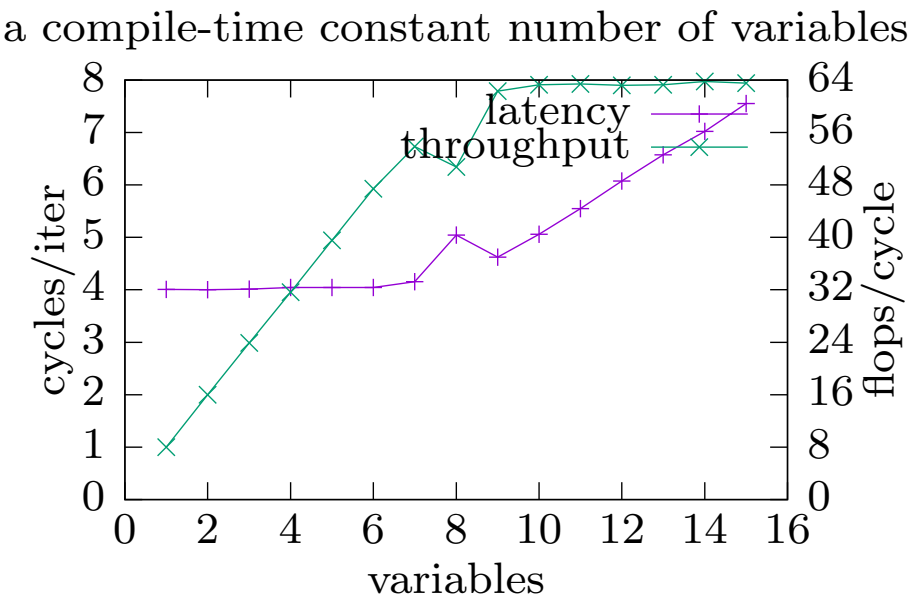
Superscalar processors

What you need to know:

- instructions are decoded in the program order,
- but the execution orders are not constrained by it *(out of order execution)*
- $\Rightarrow$ as a crude approximation, performance is constrained only by
  - *latency:* imposed by *dependencies* between instructions
  - *throughput:* imposed by execution resources of the processor (e.g., two fmadds/cycle)

A general theory of workload performance on aggressive superscalar machines

- *dependency* constrains how fast a computation can proceed, even if there are infinite number of execution resources
- increase the number of independent computations and you increase *throughput*, until it hits the limit of execution resources



Contents

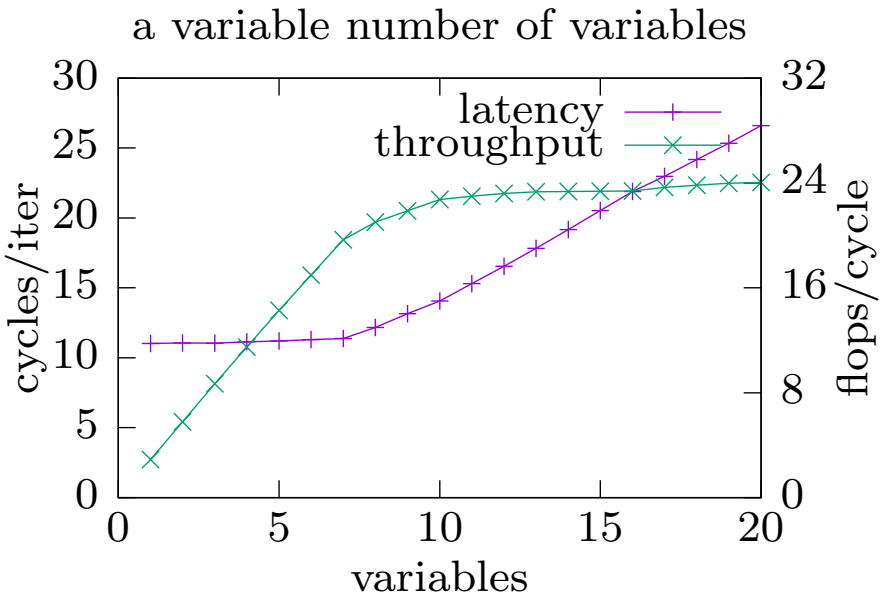
What if the number of chains is a variable?

- purpose: illustrate the same concept with a slightly more complex/common case
- let's try the following code, identical to the one we successfully achieved nearly peak performance for, except that *the number of variables (m) is now a variable* (not a compile-time constant)

```
1  void axpy_simd_m(..., long m) {
2    for (long i = 0; i < n; i++) {
3      for (long j = 0; j < m; j++) {
4        X[j] = a * X[j] + c;
5      } } }
```

When we experiment . . .

chains	clocks/iter	flops/clock
1	11.034	2.899
2	11.062	5.785
3	11.047	8.689
4	11.150	11.479
5	11.199	14.286
6	11.308	16.951
7	11.389	19.666
8	12.174	21.027
9	13.159	21.885
10	14.071	22.741
. . .	. . .	. . .
17	22.977	23.675
18	24.171	23.829
19	25.345	23.988
20	26.605	24.055



- the pattern is similar, but there are two differences:
  - the latency of a single update became ≈ 11 cycles
  - the throughput hits the plateau at ≈ 24 flops/cycles (≈ 0.75 vfmaddps/cycle)

Take a look at assembly

- it looks like:

```
1 .L1811:
2   vmovaps %zmm0, %zmm2
3   addq    $64, %rcx
4   vfmadd132ps -64(%rcx), %zmm1, %zmm2
5   vmovups %zmm2, -64(%rcx)
6   cmpq    %rcx, %r8
7   jne     .L1811
```

- what's the difference from the code we have seen before (whose latency = 4 cycles)?

```
1 .L1800:
2   addq    $1, %rdx
3   vfmadd132ps %zmm0, %zmm1, %zmm2
4   cmpq    %rdx, %rdi
5   jne     .L1800
```

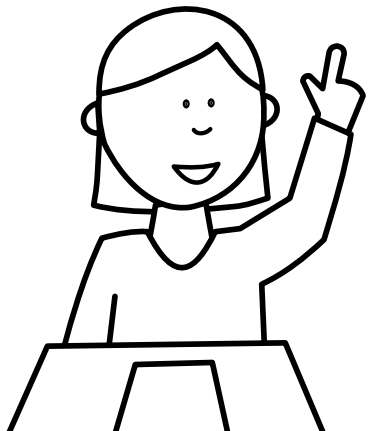
The reason of the *latency* (11 cycles)

- both stem from the fact that the code now involves load/stores
- *what you need to know:*
  just like FMA, each instruction has its own latency

Latencies of various instructions

instruction	Haswell	Broadwell	Skylake
fp add	3	3	4
fp mul	5	3	4
fp fmadd	5	5	4
typical integer ops	1	1	1
load	3	3	3(*)
store	4	4	3(*)
...	...	...	...

- $3 + 4 + 3 = 10 \neq 11$; I could not get information that confirms the extra cycle, but a simpler experiment shows the same result [1]may be 512 bit store takes 4 cycles?)



The reason of the *throughput*

- *what you need to know:*

  Just like FMA, all instructions have their own throughput limits, due to execution resources (dispatch ports and execution units)

- "two `vfmadd`s per cycle" is just an example of it

Some representative throughput limits

- Throughput = the number of that instruction that can be executed by cycle

instruction	Haswell	Broadwell	Skylake
fp add/mul/fmadd	2	2	2
load	2	2	2
store	1	1	1
typical integer ops	4	4	4
...	...	...	...

- Note: I couldn't get the reason for $\approx 0.75 < 1$ (iterations/cycle)

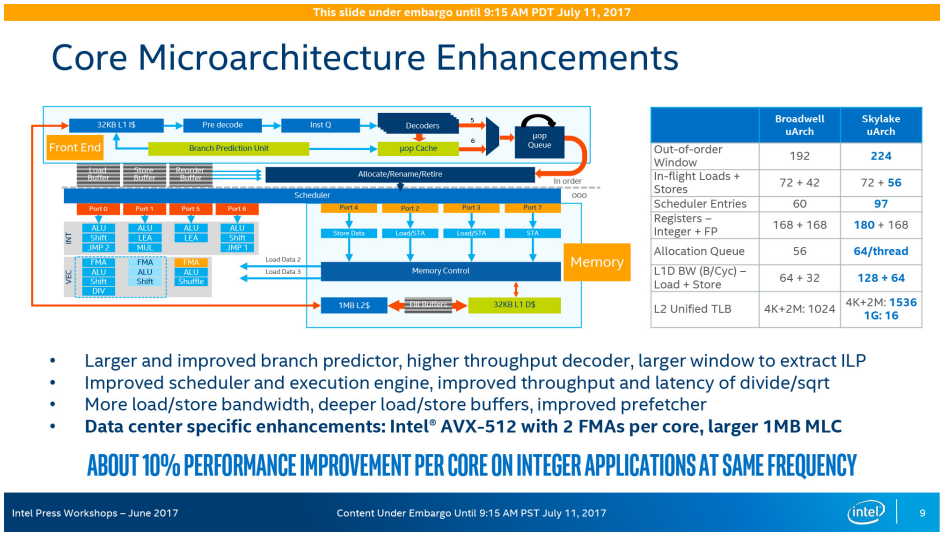
A back of envelope calculation

instruction	type	1/throughput
`vmovaps %zmm0,%zmm2`	register move	0.33
`addq $64,%rcx`	int op	0.25
`vfmadd132ps -64(%rcx),%zmm1,%zmm2`	load + FMA	0.5, 0.5
`vmovups %zmm2,-64(%rcx)`	store	1.0
`cmpq %rcx,%r8`	compare	0.25
`jne .L1811`	jump	1-2

- I don't know what `1-2` means
- we can conclude that the throughput ≤ 1 due to the store
- more general cases require understanding *dispatch ports*

Dispatch ports

- each instruction (μ-operation) is dispatched to a specific *port*
- Skylake X ports
  - fmadd → port 0 or 1
  - load → port 2 or 3
  - store → port 4
  - load/store address generation → port 7
  - int → port 0, 1, 5, or 6
  - etc.



source: `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)`

- each port can take only a single operation per cycle
  - this determines the aggregate throughput of all instructions that go to that port
- *with destination ports of instructions, one can calculate the throughput limit of a given loop*

Intel Architecture Code Analyzer

- a great tool to analyze the throughput (and latency to some extent) limit
- `https://software.intel.com/en-us/articles/intel-architecture-code-analyzer`

Num Of Uops	Ports pressure in cycles 0 - DV	1	2 - D	3 - D	4	5	6	7		
0*										vmovaps ymm10, ymm0
0*										vmovaps ymm11, ymm0
2^	1.0		1.0 1.0							vfmadd132ps ymm10, ymm1, ymmword ptr [rax]
0*										vmovaps ymm12, ymm0
2^				0.2	1.0			0.8	CP	vmovaps ymmword ptr [rax], ymm10
0*										vmovaps ymm13, ymm0
0*										vmovaps ymm14, ymm0
0*										vmovaps ymm15, ymm0
2^		1.0		1.0 1.0						vfmadd132ps ymm11, ymm1, ymmword ptr [rax+0x20]
2^	1.0		1.0 1.0							vfmadd132ps ymm12, ymm1, ymmword ptr [rax+0x40]
2^		1.0		1.0 1.0						vfmadd132ps ymm13, ymm1, ymmword ptr [rax+0x60]
2^	1.0		1.0 1.0							vfmadd132ps ymm14, ymm1, ymmword ptr [rax+0x80]
2^					1.0			1.0	CP	vmovaps ymmword ptr [rax+0x20], ymm11
0*										vmovaps ymm2, ymm0
0*										vmovaps ymm3, ymm0
2^		1.0		1.0 1.0						vfmadd132ps ymm15, ymm1, ymmword ptr [rax+0xa0]
2^	1.0		1.0 1.0							vfmadd132ps ymm2, ymm1, ymmword ptr [rax+0xc0]
2^					1.0			1.0	CP	vmovaps ymmword ptr [rax+0x40], ymm12
2^		1.0		1.0 1.0						vfmadd132ps ymm3, ymm1, ymmword ptr [rax+0xe0]
1						1.0				add rax, 0x100
2^					1.0			1.0	CP	vmovaps ymmword ptr [rax-0xa0], ymm13
2^			0.3	0.2	1.0			0.5	CP	vmovaps ymmword ptr [rax-0x80], ymm14
2^			0.2	0.3	1.0			0.5	CP	vmovaps ymmword ptr [rax-0x60], ymm15
2^			0.5	0.5	1.0				CP	vmovaps ymmword ptr [rax-0x40], ymm2
2^			0.3	0.2	1.0			0.5	CP	vmovaps ymmword ptr [rax-0x20], ymm3
1							1.0			cmp r8, rax
0F										jnz 0xffffffffffffff68

Total Num Of Uops: 34

- checking the web site, I realized Intel now says it reached End Of Life and suggests use of LLVM-MCA (I will show it

How to overcome the throughput limit?

- the goal is two iterations/cycle (throughput limit of FMA)
- the bottleneck is a store instruction (1/cycle)
- we obviously need to quit loading/storing data for every single fmadd

```
1 for (i = 0; i < n; i++) {
2   for (j = 0; j < nv; j++) {
3     x[j] = a * x[j] + c; // load; fmadd; store
4   }
5 }
```

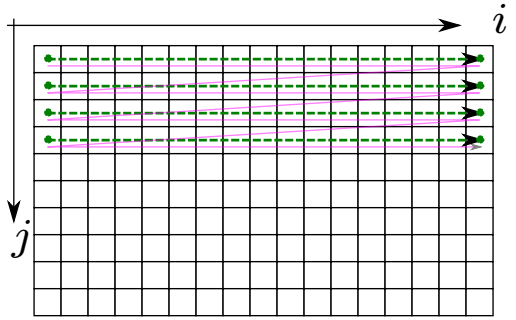
- the minimum "unit" of a computation should look like:

```
1 load x[j] to a register;
2 do "a * x + c" several times on the register;
3 store the result to x[j];
```

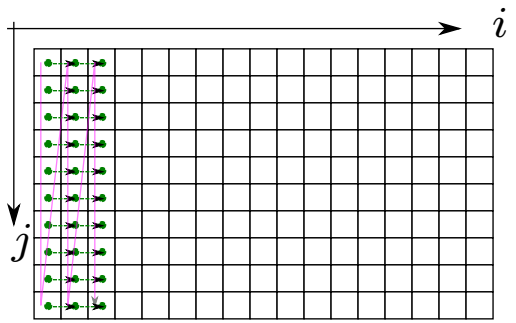
- and run multiple independent units

Several ways to arrange computation

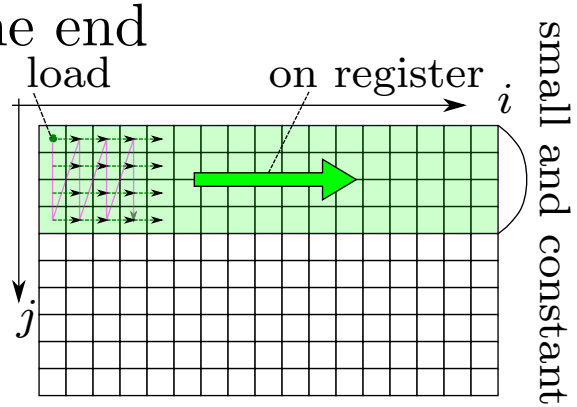
- take a variable at a time and run it until the end (suffer from latency)



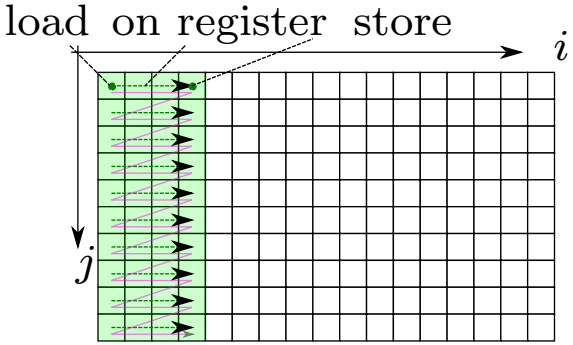
- advance all variables one step at a time (suffer from the store throughput)



- strategy 1: take a few variables and run them until the end



- strategy 2: advance all variables, a few steps at a time



Implementing strategy 1

- say we advance ten elements at a time

```
1  for (j = 0; j < nv; j += b) {
2    /* run b variables until the end */
3    for (i = 0; i < n; i++) {
4      for (jj = j; jj < j + b; jj++) {
5        xx[jj] = a * xx[jj] + c;
6      }
7    }
8  }
```

- we hope it loads/stores each variable only once through the i loop (line 2)!
- this coding *depends on the compiler's smartness* we have witnessed
  - promote fixed-sized arrays into registers

Implementing strategy 2

- advance all variables, say three, steps at a time

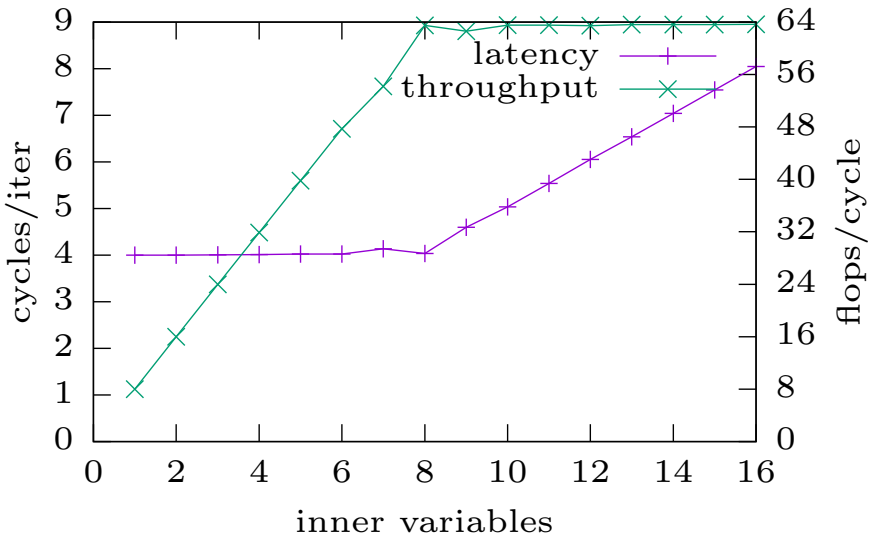
```
1  for (i = 0; i < n; i += 3) {
2    /* run all variables 3 steps */
3    for (j = 0; j < m; j++) {
4      for (ii = 0; ii < 3; ii++) {
5        x[j] = a * x[j] + c;
6      }
7    }
8  }
```

- again, we hope the compiler's smartness to eliminate intermediate load/stores (purple parts)
- the latency of a single j iteration increases, but we hope the j loop exposes lots of independent computations

Results (strategy I)

chains	clocks/iter	flops/clock
1	4.002	7.994
2	4.001	15.992
3	4.005	23.968
4	4.014	31.886
5	4.021	39.788
6	4.022	47.734
7	4.133	54.196
8	4.032	63.478
9	4.599	62.613
10	5.034	63.558
11	5.539	63.544
12	6.051	63.452
13	6.538	63.628
14	7.043	63.608
15	7.544	63.626
16	8.044	63.646

a compile-time constant number of variables in the innermost loop

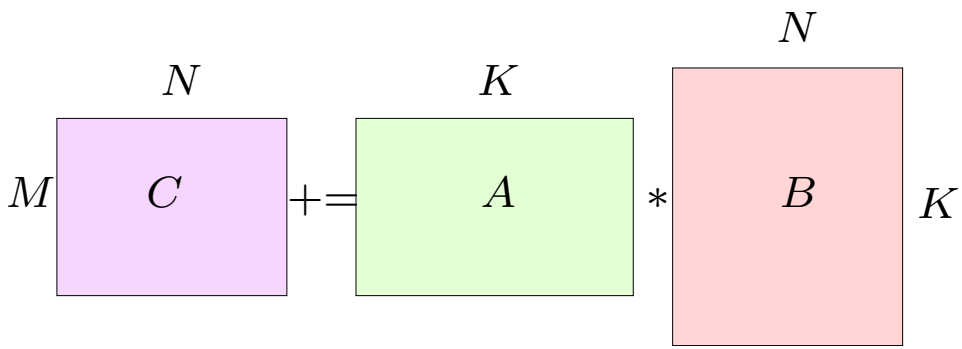


Contents

Developing near peak FLOPS matrix multiply

- let's develop a (single core) matrix multiply that runs at fairly good FLOPS on Skylake-X
- it is a great application of the concept you have just learned

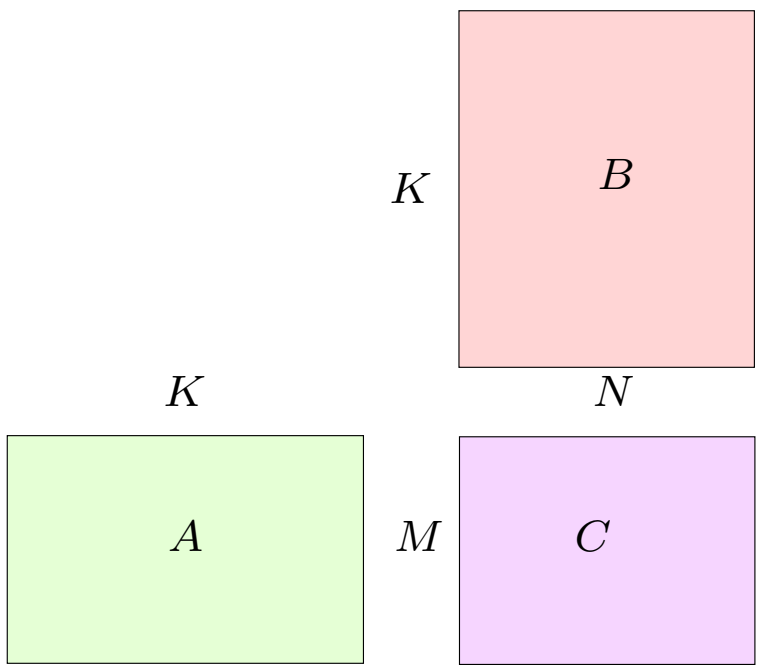
$$C = A * B + C$$



Developing near peak FLOPS matrix multiply

- let's develop a (single core) matrix multiply that runs at fairly good FLOPS on Skylake-X
- it is a great application of the concept you have just learned

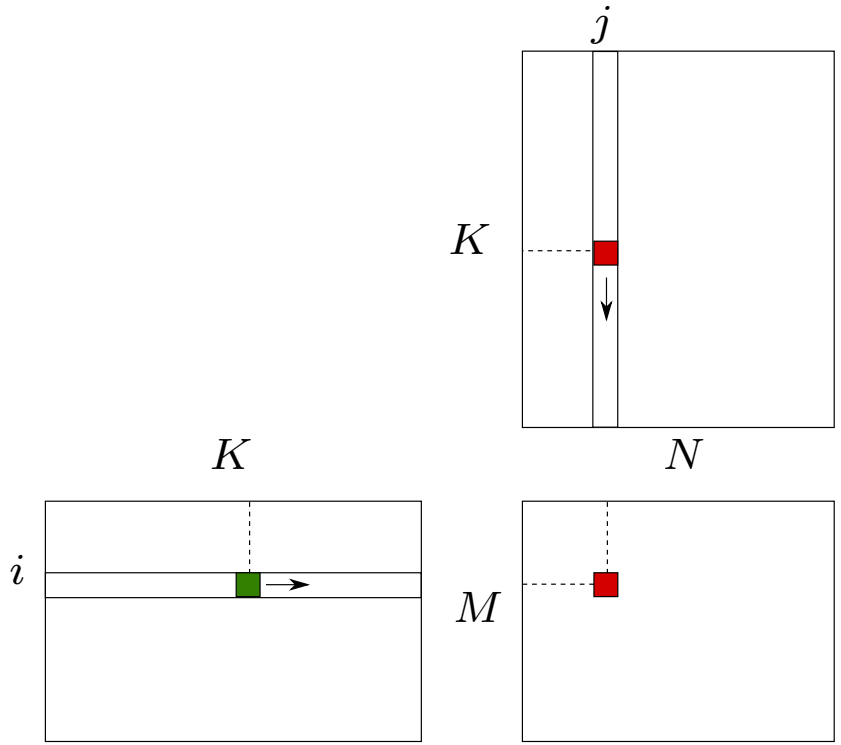
$$C = A * B + C$$



A few simplifying assumptions

- we add assumptions that M, N, and K are multiple of certain numbers along the way, (forget how to process "remainder" rows/columns in this slide)
- we assume matrix sizes are known at compile time and are "convenient" (e.g., they are small)
- multiplication of larger (and unknown size) matrices can be built on top of this

Step 1: Baseline code




```
1  $ ./mmc00
2  M = 12, N = 32, K = 192
3  A : 12 x 192 (ld=192) 9216 bytes
4  B : 192 x 32 (ld=32) 24576 bytes
5  C : 12 x 32 (ld=32) 1536 bytes
6  total = 35328 bytes
7    ...
8  3.456 CPU clocks/iter
9  2.520 REF clocks/iter
10 0.579 flops/CPU clock
11 0.794 flops/REF clock
12 2.058 GFLOPS
```

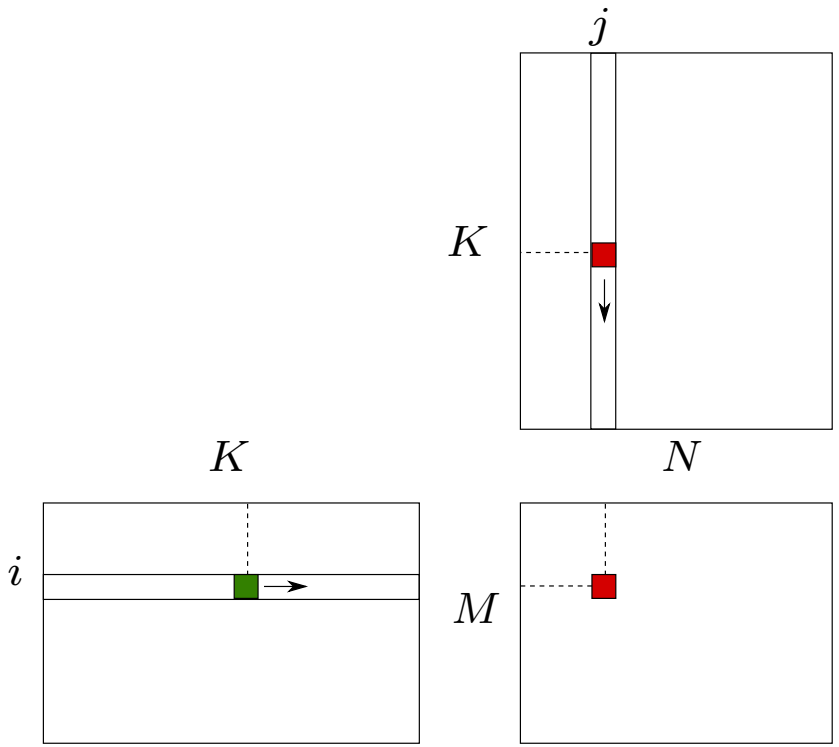
```
1  for (i = 0; i < M; i++)
2    for (j = 0; j < N; j++)
3      for (k = 0; k < K; k++)
4        C(i,j) += A(i,k) * B(k,j);
```

- it runs at ≈ 3.5 clocks / innermost loop
- latency of fmadd on `C(i,j)` – fraction

Step 1: analysis

- latency limit : latency of FMA
  - I don't know why it's slightly *smaller* than 4, but note that the true dependence occurs only for additions
- throughput limit : not important
- $\rightarrow \approx$ 2 flops / 4 cycles = 0.5 flops/cycle

Step 2: Vectorization

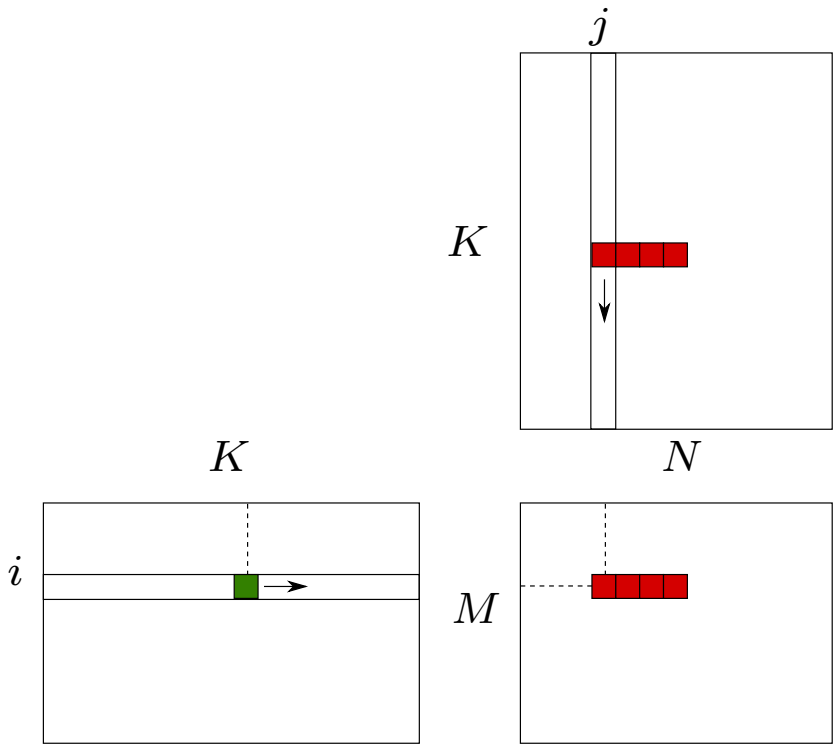



```
1  M = 12, N = 32, K = 192
2  A : 12 x 192 (ld=192) 9216 bytes
3  B : 192 x 32 (ld=32) 24576 bytes
4  C : 12 x 32 (ld=32) 1536 bytes
5  total = 35328 bytes
6  repeat : 100000 times
7   ...
8  3.555 CPU clocks/iter
9  2.681 REF clocks/iter
10 9.002 flops/CPU clock
11 11.936 flops/REF clock
12 30.960 GFLOPS
```

```
1  for (i = 0; i < M; i++)
2    for (j = 0; j < N; j += L)
3      for (k = 0; k < K; k++)
4        C(i,j:j+L) += A(i,k) * B(k,j:j+L);
```

- assumption: N is a multiple of SIMD lanes (L)

Step 2: Vectorization




```
1  M = 12, N = 32, K = 192
2  A : 12 x 192 (ld=192) 9216 bytes
3  B : 192 x 32 (ld=32) 24576 bytes
4  C : 12 x 32 (ld=32) 1536 bytes
5  total = 35328 bytes
6  repeat : 100000 times
7   ...
8  3.555 CPU clocks/iter
9  2.681 REF clocks/iter
10 9.002 flops/CPU clock
11 11.936 flops/REF clock
12 30.960 GFLOPS
```

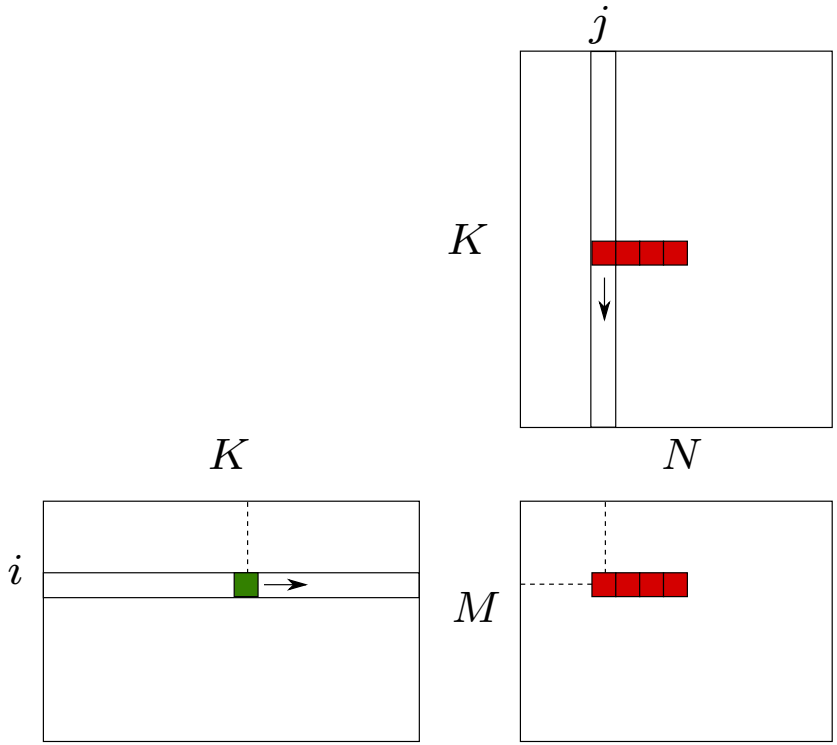
```
1  for (i = 0; i < M; i++)
2    for (j = 0; j < N; j += L)
3      for (k = 0; k < K; k++)
4        C(i,j:j+L) += A(i,k) * B(k,j:j+L);
```

- assumption: N is a multiple of SIMD lanes (L)

Step 2: analysis

- almost the same as step 1, except that each iteration now does 16 FMAs (as opposed to an FMA)
- $\rightarrow \approx$ 32 flops / 4 cycles = 8 flops/cycle

Step 3: increase parallelism!



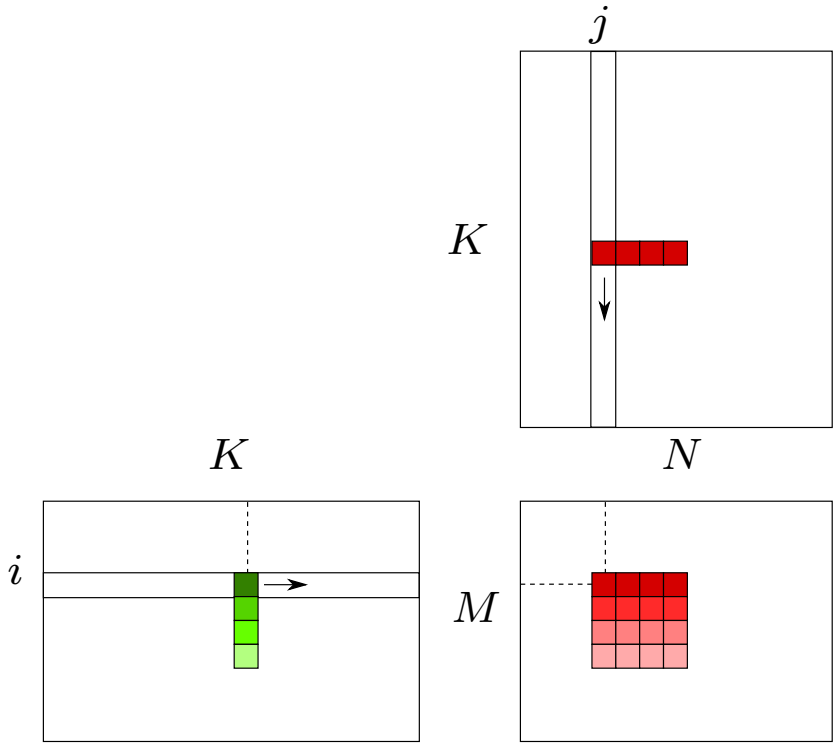

```
1  login000:07mm$ ./mmc02
2  M = 8, N = 32, K = 192
3  A : 8 x 192 (ld=192) 6144 bytes
4  B : 192 x 32 (ld=32) 24576 bytes
5  C : 8 x 32 (ld=32) 1024 bytes
6     ...
7  5.451 CPU clocks/iter
8  4.387 REF clocks/iter
9  46.966 flops/CPU clock
10 58.349 flops/REF clock
11 151.341 GFLOPS
```

- update bM vector elements of C concurrently

```
1 for (i = 0; i < M; i += bM)
2   for (j = 0; j < N; j += L)
3     for (k = 0; k < K; k++)
4       for (di = 0; di < bM; di++)
5         C(i+di,j:j+L) += A(i+di,k) * B(k,j:j+L);
```

- Skylake requires $bM \geq 8$ to reach peak FLOPS

Step 3: increase parallelism!



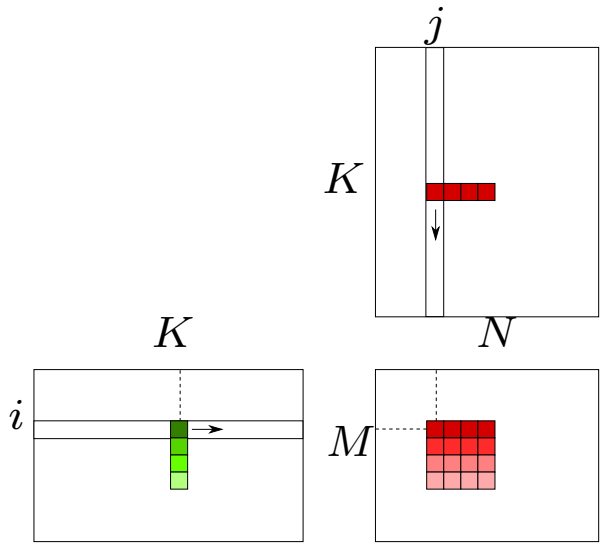

```
1  login000:07mm$ ./mmc02
2  M = 8, N = 32, K = 192
3  A : 8 x 192 (ld=192) 6144 bytes
4  B : 192 x 32 (ld=32) 24576 bytes
5  C : 8 x 32 (ld=32) 1024 bytes
6     ...
7  5.451 CPU clocks/iter
8  4.387 REF clocks/iter
9  46.966 flops/CPU clock
10 58.349 flops/REF clock
11 151.341 GFLOPS
```

- update bM vector elements of C concurrently

```
1 for (i = 0; i < M; i += bM)
2   for (j = 0; j < N; j += L)
3     for (k = 0; k < K; k++)
4       for (di = 0; di < bM; di++)
5         C(i+di,j:j+L) += A(i+di,k) * B(k,j:j+L);
```

- Skylake requires $bM \geq 8$ to reach peak FLOPS

Step 3: analysis



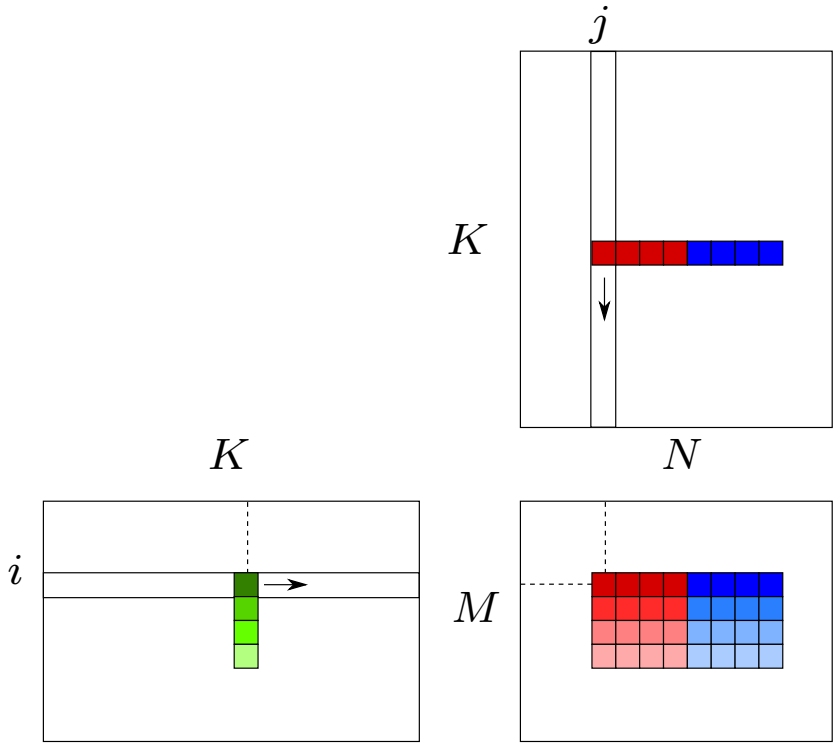

```
1  for (i = 0; i < M; i += bM)
2   for (j = 0; j < N; j += L)
3    for (k = 0; k < K; k++)
4     for (di = 0; di < bM; di++)
5      C(i+di,j:j+L) += A(i+di,k) * B(k,j:j+L);
```

- the for loop at line 4 performs
  - bM loads (broadcasts) for `A(i+di,k)`
  - 1 load for `B(k,j:j+L)`
  - bM FMAs
- remember the load throughput = 2 loads/cycle
- to achieve 2 FMAs/cycle, we must have

$$\text{the number of loads} \leq \text{the number of FMAs}$$

- *we need to remove an extra load instruction*

Step 4: Reuse an element of A




```
1  $ ./mmc03
2  M = 8, N = 32, K = 192
3  A : 8 x 192 (ld=192) 6144 bytes
4  B : 192 x 32 (ld=32) 24576 bytes
5  C : 8 x 32 (ld=32) 1024 bytes
6    ...
7  4.926 CPU clocks/iter
8  4.938 REF clocks/iter
9  51.969 flops/CPU clock
10 51.846 flops/REF clock
11 134.474 GFLOPS
12
```

- update $bM' \times bN$ block rather than $bM \times 1$

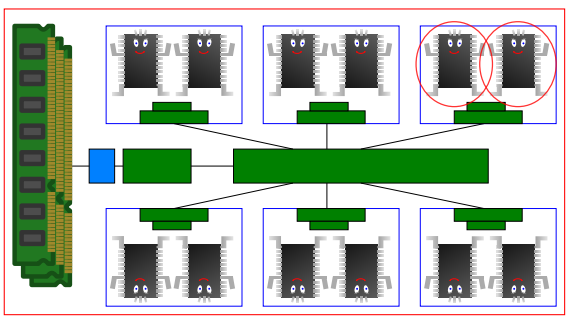
```
1 for (i = 0; i < M; i += bM')
2   for (j = 0; j < N; j += bN * L)
3     for (k = 0; k < K; k++)
4       for (di = 0; di < bM'; di++)
5         for (dj = 0; dj < bN * L; dj += L)
6           C(i+di,j+dj:j+dj+L) += A(i+di,k) * B(k,j+dj:j+L);
```

Step 4: Analysis

- the for loop at line 4 performs
  - *bM'* loads (broadcast) for `A(i+di,k)`
  - *bN* loads for `B(k,j:j+L)`
  - *bM'* $\times$ bN FMAs
- the minimum requirement for it to achieve the peak FLOPS is *bM'* $\times$ *bN* ≥ 8
- in the experiments, when we set *bM'* $= 6$ and *bN* $= 2$, it gets 59 flops/cycle (92% of the peak)
- we need to note that this happens only when the matrix is small ($M = 12, N = 32, K = 160$) and we repeat it many times
- the issue for large matrices will be the next topic

Simultaneous Multithreading (SMT)

- each physical core has several *hardware threads* or *virtual cores*
  - recent Xeon processors (including Skylake-X) : 2
  - Knights Landing/Mill : 4
  - IBM Power : 8
- a.k.a. Hyperthreading®
- virtual cores on a single physical core
  - are concurrently executed by the hardware
  - have their own registers (switching between them have almost no overhead)
  - *share most execution resources* (dispatch ports, floating point units, L1/L2 caches, etc.)



Performance implications of virtual cores

- having as many threads as virtual cores on a physical core
  - helps improve throughput of *latency-bound* applications, but
  - does not help *throughput-bound* applications
- note: if you have more threads than virtual cores, operating systems get involved to switch between them (in a much coarser granularity, say 1 ms (10^{-3} sec) rather than every cycle $\sim$ 1 ns (10^{-9} sec))
  - it never helps mitigate the latency of arithmetic operations
  - it helps mitigate the latency of much bigger I/O latencies (say when accessing HDD)

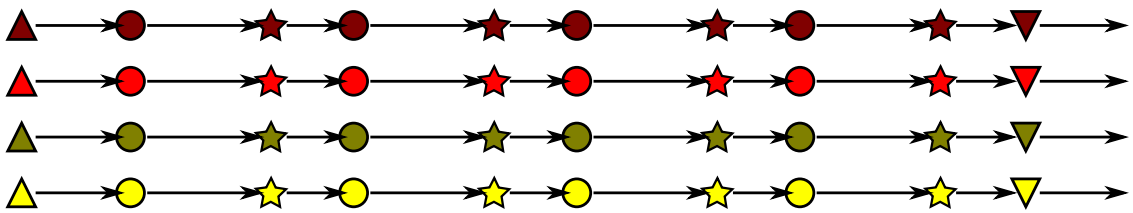
Takeaways (1)

- peak FLOPS of recent Intel CPUs = "execute two `fmadds` every cycle" *(no other combinations)*
  - other processors have different limits, but the basic is the same
- no, single-core performance is not about reducing the number of instructions
- it's about how to increase parallelism
  - SIMD
  - ILP

Takeaways (2)

- dependent instructions incur latencies and hinder parallelism

- independent instructions are executed in parallel, up to throughput limits



- throughput limits are determined by dispatch ports