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

- measure how fast we can iterate the following loop (a similar experiment we did on GPU)

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

- the code performs `L` $\times$ `n` FMAs and almost nothing else (`L` $=$ the number of lanes in a single SIMD variable)

Assembly

```
.LBB3_8:
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  addq $-8, %rax
  jne .LBB3_8
```

- the loop is unrolled eight times
- why does it take > 3 cycles to do a single fmadd?

Contents

Latency and throughput

- our core (Ice Lake) can execute *two `fmadd` instructions every cycle*
- but it does *not* mean the result of `vfmadd` at a line below is available in the next cycle for `vfmadd` at the next line

```
.LBB3_8:
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  addq $-8, %rax
  jne .LBB3_8
```

Latency and throughput

- our core (Ice Lake) can execute *two `fmadd` instructions every cycle*
- but it does *not* mean the result of `vfmadd` at a line below is available in the next cycle for `vfmadd` at the next line

```
.LBB3_8:
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  addq $-8, %rax
  jne .LBB3_8
```

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

Latencies/throughput

instruction	Haswell	Broadwell	Skylake
fp add	3	3	4/2
fp mul	5	3	4/2
fp fmadd	5	5	4/2
typical integer ops	1	1	$1/>2$
...	...	...	...

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

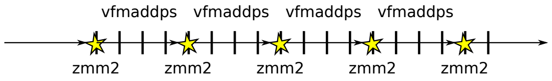
- in our code, a `vfmadd` uses the result of the immediately preceding `vfmadd`
- there are *dependencies* between them
- *that was obvious from the source code too*

```
.LBB3_8:
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
     ...
  vfmadd213pd %zmm1, %zmm0, %zmm2
  vfmadd213pd %zmm1, %zmm0, %zmm2
  addq $-8, %rax
  jne .LBB3_8
```

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

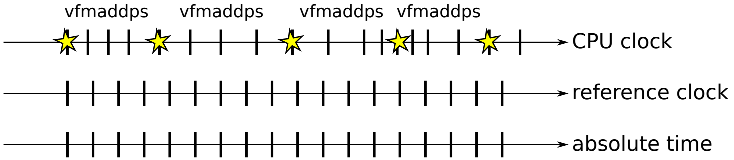
Conclusion:

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



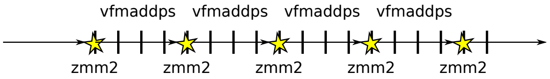
Contents

How to overcome latencies?

- increase parallelism (no other ways)!

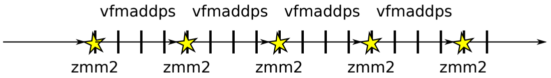
How to overcome latencies?

- increase parallelism (no other ways)!
- you *can't* make a serial chain of dependent computation run faster than determined by latencies

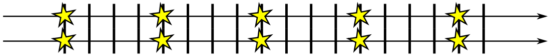


How to overcome latencies?

- increase parallelism (no other ways)!
- you *can't* make a serial chain of dependent computation run faster than determined by latencies

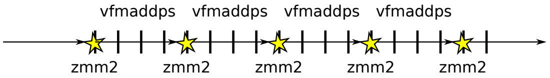


- you *can* only increase *throughput*, by running multiple *independent* chains

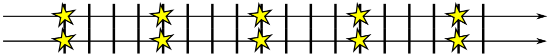


How to overcome latencies?

- increase parallelism (no other ways)!
- you *can't* make a serial chain of dependent computation run faster than determined by latencies



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



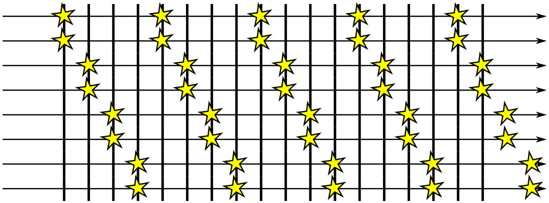
- we expect the following to finish in the same number of cycles as the original one, despite it performs twice as many flops

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

Increase the number of chains further ...

- we expect to reach peak FLOPS with $\geq 2/(1/4) = 8$ chains (i.e., `nv` ≥ 8)

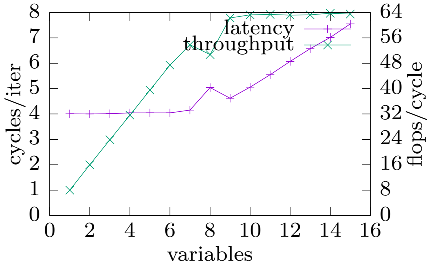
```
long axpy_simd_c( ... ) {
  for (long i = 0; i < n; i++) {
    for (long j = 0; j < nv; j++) {
      X[j] = a * X[j] + c;
    } } }
```



- note: the above reasoning assumes a compiler's smartness
- in particular, `X[j] = a * X[j] + c` is compiled into an FMA instruction on registers without load/store instructions (i.e., each of `X[0], ..., X[7]` gets assigned a register)

Results

a compile-time constant number of variables



```
for (i = 0; i < n; i++) {
  x0 = a * x0 + b;
  x1 = a * x1 + b;
     ...
}
```

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

how modern aggressive superscalar processors work:

Superscalar processors

how modern aggressive superscalar processors work:

- instruction decoding goes much ahead of actual executions

Superscalar processors

how modern aggressive superscalar processors work:

- instruction decoding goes much ahead of actual executions
- the actual execution of an instruction does not happen until, and happens as soon as, *its operands and execution resources are ready (out of order execution)*

Superscalar processors

how modern aggressive superscalar processors work:

- instruction decoding goes much ahead of actual executions
- the actual execution of an instruction does not happen until, and happens as soon as, *its operands and execution resources are ready (out of order execution)*
- $\Rightarrow$ as a crude approximation, performance is constrained by

Superscalar processors

how modern aggressive superscalar processors work:

- instruction decoding goes much ahead of actual executions
- the actual execution of an instruction does not happen until, and happens as soon as, *its operands and execution resources are ready (out of order execution)*
- ⇒ as a crude approximation, performance is constrained by
  - *latency:* imposed by *dependencies* between instructions

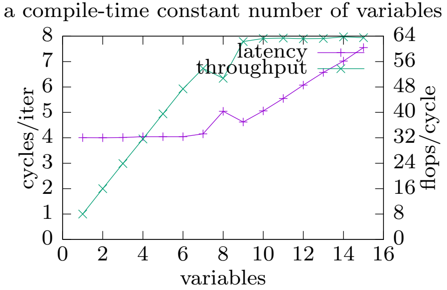
Superscalar processors

how modern aggressive superscalar processors work:

- instruction decoding goes much ahead of actual executions
- the actual execution of an instruction does not happen until, and happens as soon as, *its operands and execution resources are ready (out of order execution)*
- $\Rightarrow$ as a crude approximation, performance is constrained 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



A more general understanding about *throughput* limits

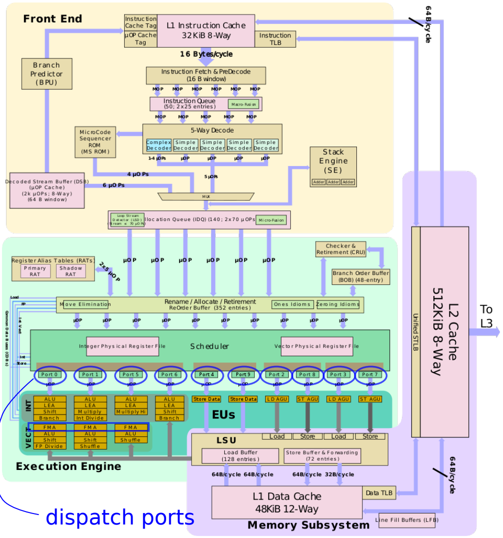
- *what you need to know:*
  all instructions have their own throughput limits (just like FMA), due to execution resources
- some examples of recent Intel CPUs

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

- e.g., a loop containing 10 load instructions takes $\geq 10/2 = 5$ cycless/iteration
- different but similar instructions may use the same execution resource so may be subject of the same limitation
- a more general reasoning $\Rightarrow$ *dispatch ports*

Dispatch ports

- each instruction (μ-operation) is dispatched to a specific execution unit through a *dispatch port*
- each port can take only a single operation per cycle
- this determines the throughput of all instructions that go to that port
- *with destination ports of instructions, one can calculate the throughput limit of a given loop*



Chipwikia - Sunny cove architecture, CC BY-SA 4.0,

`https://commons.wikimedia.org/w/index.php?curid=122557706`

LLVM Machine Code Analyzer (`llvm-mca`)

- a great tool to analyze the throughput (and latency to some extent) limit
- given a code sequence, it shows
  - latency and
  - dispatch port

  of each instruction and, based on them calculates the number of cycles per iteration,
- under some simplifying assumptions
  - the given sequence repeats many times
  - no cache misses (!)
  - no dependencies through memory (load does not depend on earlier stores)
  - no branch misprediction
- $\Rightarrow$ a great tool to analyze the innermost, straight sequence of instructions without branches (basic blocks)

How to use `llvm-mca`

1. generate assembly (get `program.s`) by, e.g.,

```
clang -O3 -mavx512f -mfma ... program.c -S
```

2. find the loop you want to analyze in the assembly
3. sandwich it by `# LLVM-MCA-BEGIN` and `# LLVM-MCA-END`

```
# LLVM-MCA-BEGIN
.L123
   ...
   ...
   jne .L123
# LLVM-MCA-END
```

4. run `llvm-mca` tool on the assembly code

```
llvm-mca program.s
```

How to use `llvm-mca`

- it shows
    - latency of each instruction
    - dispatch port used by each instruction

  and how many instructions use each of the dispatch ports (therefore the throughput limit of the loop)
- with `--timeline` option,

```
llvm-mca --timeline program.s
```

  it also shows when each instruction gets decoded, dispatched, and finished (particularly instructive)

Example

- input (assembly)

```
# LLVM-MCA-BEGIN
.LBB3_8:
 # xmm0 = (xmm1 * xmm0) + xmm2
 vfmadd213sd %xmm2, %xmm1, %xmm0
 vfmadd213sd %xmm2, %xmm1, %xmm0
 vfmadd213sd %xmm2, %xmm1, %xmm0
 vfmadd213sd %xmm2, %xmm1, %xmm0
 vfmadd213sd %xmm2, %xmm1, %xmm0
 vfmadd213sd %xmm2, %xmm1, %xmm0
 vfmadd213sd %xmm2, %xmm1, %xmm0
 vfmadd213sd %xmm2, %xmm1, %xmm0
 addq $-8, %rax
 jne .LBB3_8
# LLVM-MCA-END
```

Example

- output (dispatch port used by each instruction)

```
Resource pressure by instruction:
[0]   [1]   [2]   [3]   [4] .. [11] Instructions:
 -     -    0.99 0.01  -       -    vfmadd213sd %xmm2, %xmm1, %xmm0
 -     -     -   1.00  -       -    vfmadd213sd %xmm2, %xmm1, %xmm0
 -     -    0.99 0.01  -       -    vfmadd213sd %xmm2, %xmm1, %xmm0
 -     -     -   1.00  -       -    vfmadd213sd %xmm2, %xmm1, %xmm0
 -     -    1.00  -    -       -    vfmadd213sd %xmm2, %xmm1, %xmm0
 -     -     -   1.00  -       -    vfmadd213sd %xmm2, %xmm1, %xmm0
 -     -    1.00  -    -       -    vfmadd213sd %xmm2, %xmm1, %xmm0
 -     -     -   1.00  -       -    vfmadd213sd %xmm2, %xmm1, %xmm0
 -     -     -   0.01  -       -    addq $-8, %rax
 -     -    0.04  -    -       -    jne .LBB3_8
```

Example

- output (timeline)

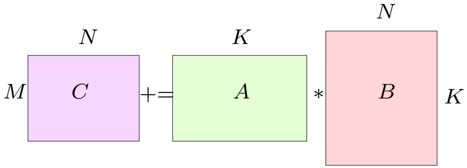
```
             ...
D=====...==eeeeER   .     .     .     .  vfmadd213sd %xmm2, %xmm1, %xmm0
.D====...======eeeeER     .     .     .  vfmadd213sd %xmm2, %xmm1, %xmm0
.D====...==========eeeeER.     .     .  vfmadd213sd %xmm2, %xmm1, %xmm0
.DeE--...---------------R.     .     .  addq $-8, %rax
.D=eE-...---------------R.     .     .  jne .LBB3_8
.D====...==============eeeeER .     .  vfmadd213sd %xmm2, %xmm1, %xmm0
.D====...==================eeeeER   .  vfmadd213sd %xmm2, %xmm1, %xmm0
. D===...======================eeeeER vfmadd213sd %xmm2, %xmm1, %xmm0
             ...
```

Contents

Developing near peak FLOPS matrix multiply

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

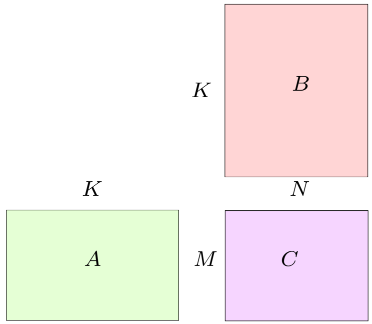
$$C = A * B + C$$



Developing near peak FLOPS matrix multiply

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

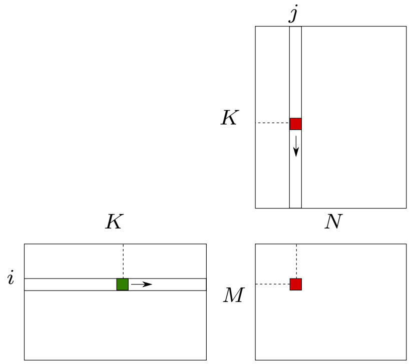
$$C = A * B + C$$



A few convenient assumptions

- we add assumptions that M, N, and K are multiple of certain numbers along the way, (don't worry about "remainder" rows/columns)
- we assume matrix sizes are conveniently small (don't worry about memory access cost, which is actually a significant factor to design matrix multiply for larger matrices)
- multiplication of larger (and unknown size) matrices can be built on top of this

Step 1: Baseline code



```
$ mm_base 8 32 192
M = 8, N = 32, K = 192
L : 16
A : 8 x 192 (ld=192) 6144 bytes
B : 192 x 32 (ld=32) 24576 bytes
C : 8 x 32 (ld=32) 1024 bytes
total = 31744 bytes
repeat : 20346 times
perform 1000046592 fmas ... done
2844287815 clocks
0.351598 fmas/cycle
```

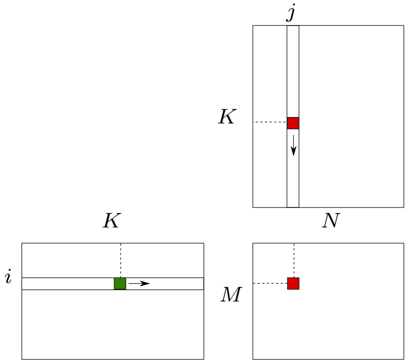
```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < K; k++)
      C(i,j) += A(i,k) * B(k,j);
```

- it runs at $\approx$ 2.8 clocks / innermost loop

Step 1: analysis

- latency limit : latency of FMA
  - the reason why it's slightly *smaller* than 4 is there are some overlaps between different elements of C
  - if you set $M = N = 1$ and K large, it's almost exactly 4
- throughput limit : not important
- achieved performance : 1000046592 fmas / 2844287815 cycles $\approx$ 0.4 fmas/cycle

Step 2: Vectorization

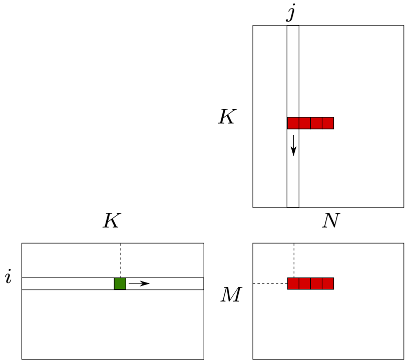


```
$ mm_simd 8 32 192
M = 8, N = 32, K = 192
L : 16
A : 8 x 192 (ld=192) 6144 bytes
B : 192 x 32 (ld=32) 24576 bytes
C : 8 x 32 (ld=32) 1024 bytes
total = 31744 bytes
repeat : 20346 times
perform 1000046592 fmas ... done
180175475 clocks
5.550404 fmas/cycle
```

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

- assumption: N is a multiple of SIMD lanes (L)
- it still runs at $\approx$ 2.8 clocks / innermost iteration

Step 2: Vectorization



```
$ mm_simd 8 32 192
M = 8, N = 32, K = 192
L : 16
A : 8 x 192 (ld=192) 6144 bytes
B : 192 x 32 (ld=32) 24576 bytes
C : 8 x 32 (ld=32) 1024 bytes
total = 31744 bytes
repeat : 20346 times
perform 1000046592 fmas ... done
180175475 clocks
5.550404 fmas/cycle
```

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

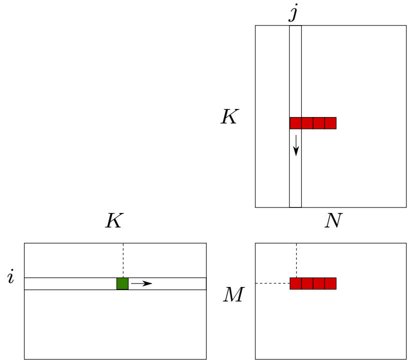
- assumption: N is a multiple of SIMD lanes (L)
- it still runs at $\approx$ 2.8 clocks / innermost iteration

Step 2: analysis

- the speed is still limited by latency
- the only difference is that each iteration now performs 16 fmas (as opposed to an fma)
- achieved throughput :

$$1000046592 \text{ fmas}/180175475 \text{ cycles} \approx 5.5 \text{ fmas/cycle}$$

Step 3: increase parallelism!



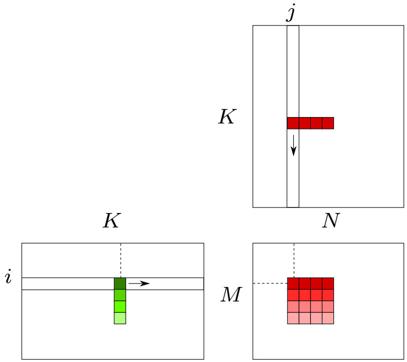
```
$ ./mm_simd_ilp 8 32 192
M = 8, N = 32, K = 192
L : 16
A : 8 x 192 (ld=192) 6144 bytes
B : 192 x 32 (ld=32) 24576 bytes
C : 8 x 32 (ld=32) 1024 bytes
total = 31744 bytes
repeat : 20346 times
perform 1000046592 fmas ... done
64836630 clocks
15.424099 fmas/cycle
```

- update bM vector elements of C concurrently

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

- Ice Lake requires $bM \geq 8$ to reach peak FLOPS

Step 3: increase parallelism!



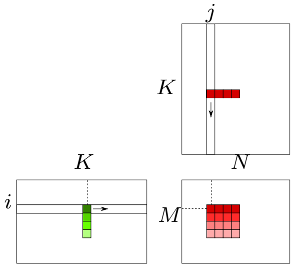
```
$ ./mm_simd_ilp 8 32 192
M = 8, N = 32, K = 192
L : 16
A : 8 x 192 (ld=192) 6144 bytes
B : 192 x 32 (ld=32) 24576 bytes
C : 8 x 32 (ld=32) 1024 bytes
total = 31744 bytes
repeat : 20346 times
perform 1000046592 fmas ... done
64836630 clocks
15.424099 fmas/cycle
```

- update bM vector elements of C concurrently

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

- Ice Lake requires $bM \geq 8$ to reach peak FLOPS

Step 3: analysis

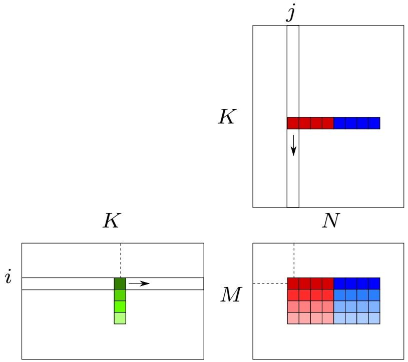


```
1  for (i = 0; i < M; i += bM)
2   for (j = 0; j < N; j += L)
3    for (k = 0; k < K; k++)
4     for (di = 0; di < bM; di++)
5      C(i+di,j:j+L) += A(i+di,k) * B(k,j:j+L);
```

- 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
- the load/broadcast throughput = 2 per cycle
- to achieve 2 FMAs/cycle, we must have

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

Step 4: Reuse an element of A



```
$ mm_simd_lip_4x2 8 32 192
M = 8, N = 32, K = 192
L : 16
A : 8 x 192 (ld=192) 6144 bytes
B : 192 x 32 (ld=32) 24576 bytes
C : 8 x 32 (ld=32) 1024 bytes
total = 31744 bytes
repeat : 20346 times
perform 1000046592 fmas ... done
38635137 clocks
25.884381 fmas/cycle
```

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

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

Step 4: Analysis

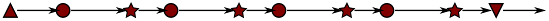
- the for loop at line 4 performs
  - *bM'* loads (broadcast) for `A(i+di,k)`
  - *bN* loads for `B(k,j:j+L)`
  - *bM'* $\times$ *bN* SIMD FMAs
- the minimum requirement for it to achieve the peak FLOPS is *bM'* $\times$ *bN* ≥ 8
- in the experiments, when we set *bM'* $= 8$ and *bN* $= 2$, it gets 25 fmas/cycle ($\approx$ 80% of the peak)
- we need to note that this happens only when the matrix is small ($M = 8, N = 32, K = 192$) and we repeat it many times
- the issue for large matrices will be the next topic

Takeaways (1)

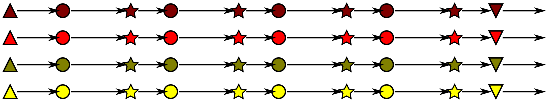
- peak FLOPS of many recent Intel CPUs = "execute two `fmadds` every cycle" *(no other combinations)*
  - other processors have different limits, but the basics is the same
  - cf. NVIDIA GPUs = "execute two warps (each doing `fmadd`) every cycle"
- single-core performance is not about reducing the number of instructions
- it's about how to increase parallelism
  - CPU : SIMD $\times$ ILP
  - GPU : threads, threads, threads, ...
  - but the internal machinery is similar (warp $\approx$ SIMD, ILP $\sim$ warps in an SM)
  - how they expose parallelism to the programmer is different

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