Exploring a nontrivial upper bound for pipelined double-precision FMA instructions on Knight’s Landing

Damon McDougall (TACC)
John McCalpin (TACC)
Conclusion

You cannot exceed 85.71% of theoretical peak performance on KNL

6/7 is approximately 85.71%

This will affect literally nobody but the curious. Why?

KNL core can only issue 2 instructions per cycle

KNL peak is 2 AVX-512 VFMADD instructions per cycle

Other instructions displace VFMADD and reduce attainable peak

Intel’s DGEMM requires just under 20% of ‘work’ that isn’t a VFMADD

Useful codes don’t *just* do FMAs. They do other stuff too.
Conclusion

You cannot exceed 85.71% of theoretical peak performance on KNL. 6/7 is approximately 85.71%

This will affect literally nobody but the curious. Why?

KNL core can only issue 2 instructions per cycle

KNL peak is 2 AVX-512 VFMADD instructions per cycle

Other instructions displace VFMADD and reduce attainable peak

Intel’s DGEMM requires just under 20% of ‘work’ that isn’t a VFMADD

Useful codes don’t *just* do FMAs. They do other stuff too.

We started with VFMADDs, but the first point applies to all VPU instructions.
The story

I read a Colfax Research article benchmarking SKX
The article is here: https://colfaxresearch.com/skl-avx512
The article compares peak performance of BDW and SKX
They run a pipelined double-precision FMA code to get their results
The code doesn’t do anything useful
It merely satiates my juvenile desire to see peak FLOPs in action
I cleaned up a few things in Colfax code and ran it on KNL
I got less than the numbers I expected
I asked John about it…
…the rest, as they say, is history.
Some background

KNL (7250) has 34 tiles
Each tile has 2 cores
Cores on a tile share an L2 cache
Each core has 2 VPUs
A picture helps…

Notes:
- Reservation stations aren’t shared
- VPUUs mostly symmetrical
- Retirement stage?
Two VPU’s are connected to the core. They are tightly integrated into the pipeline, with the allocation unit dispatching instructions directly into the VPUs. The VPUs are mostly symmetrical, and each can provide a steady-state throughput of one AVX-512 instruction per cycle, providing a peak of 64 single-precision or 32 double-precision floating-point operations per cycle from both VPUs.


What this doesn’t say:
• Each VPU can provide a steady-state throughput of one AVX-512 instruction per cycle *at the same time*
Our list of experiments ->
There are so many, I bet you didn’t notice this is upside-down ->
Our experiments

John did at least 80% of this work – my contribution was to pester him with questions

We (John) ran 258 experiments. Too many to talk about all of them
I’ll only mention a couple of the important/interesting ones

We pipelined AVX-512:

- VFMADD – executes in VPU[01], 6 cycle latency
- VPXORQ – executes in VPU[01], 2 cycle latency
- VPBROADCASTQ – executes in VPU0, 3 cycle latency
- VPABSQ – executes in VPU[01], 2 cycle latency
- VPLZCNTQ – executes in VPU0, 2 cycle latency

These come from Agner’s instruction tables
VFMADD

Executes in either VPU
We know how many FMAs we issue
   No. of loop iterations * number of FMAS per iteration
We know how many cycles the application takes
   Fixed function counter read operation
Instructions are aligned
Loop overhead is subl and jnz (this should take one cycle)
We see 12 fully pipelined FMAs execute in 14 cycles (6/7)
Why?
Are we rename register limited?

Little’s law: concurrency = latency * throughput

\[
\text{latency} \times \text{throughput} = \text{num\_rename\_regs} \times \text{num\_cycles} \times 2
\]

KNL core is based off of Silvermont core which has 14 cycle pipeline

So: num\_cycles is at least 6 because of VFMADD latency

But could be as high as 14 -- or higher?

num\_rename\_regs is at least 3 and at most 4

3 when the FMA overwrites its input register and 4 when it doesn't

Guessing: latency \times throughput = 2 \times 3 \times 14 = 84

There are 72 rename registers and 72/84 = 6/7

It is possible we are rename register limited
Allocation unit. The allocation unit reads two micro-ops per cycle from the instruction queue. It assigns the necessary pipeline resources required by the micro-ops, such as reorder buffer (ROB) entries (72), rename buffer entries (72), store data buffers (16), gather-scatter table entries (4), and reservation station entries. It also renames the register sources and destinations in the micro-ops.

Not clear how rename buffer entries are distributed:
• 72 entries total on the core?
• 72 entries each for IEU and VPU?

So let’s use fewer rename registers

VPABSQ has 2 zmm operands
So now: latency * throughput is much smaller
We still see only 12 VPABSQ instructions retire every 14 cycles
And the extra cycle manifests itself as a rat stall (same in the VFMADD case)
So we’re not (just) rename register limited
We’ve been using both VPUs. What if we only use one of them?
Ok, so what if we only use one VPU?

We have experiments that pipeline VPLZCNTQ
We have experiments that pipeline VPBROADCASTQ
Both experiments show throughput of 1 instruction per cycle
Both VPLZCNTQ and VPBROADCASTQ only execute in VPU0
No rat stalls
KORTESTW only executes in VPU1. No rat stalls here either
So it’s something to do with both VPUs together
Is it just the VPUs? What about the integer or memory EUs?
More on the one-cycle stall

All of our integer (ADDQ) and memory instructions experiments show a sustained throughput of 2 instructions per cycle
So the behaviour is *only* when using VPU instructions *together*
  Regardless of datatype and SIMD width
For pipelined VFMADD experiment
We introduce or own one-cycle stall: nop + jmp to next instruction
Our experiments show no more rat stalls (still retire 6 instructions every 7 cycles)
  I.e., nop + jmp overlaps with the existing throughput limitation
We also introduced bubbles by creating register name dependencies
  These are really hard to control (re-using input registers)
  Sometimes they overlap perfectly with the existing bubble
  Sometimes they don’t and we see both bubbles
Mixing instructions

We even tried mixing instruction pairs. E.g.,
  VFMADD/VPABSQ
  VFMADD/VPERMD

There’s not really much to add to the story here
The instructions with the shorter latency retire earlier
Until such a time when the ROB fills up
When this happens
  What were RAT stalls are now ROB stalls
  This is unsurprising: the retirement unit has to wait for the longer latency instructions to finish executing
Wrap up

We checked the obvious things:
  Instructions are not too long
  No instruction prefixes
Perhaps retirement unit is causing a 1 cycle bubble so it can copy stuff out of the ROB?
Perhaps it’s done to guarantee forward progress?
Does anyone in the audience know?
  If so, maybe they can’t say
Thanks for listening. Questions?

Damon McDougall – dmcddougall@tacc.utexas.edu
John McCalpin – mccalpin@tacc.utexas.edu