intel
experience what's inside™

WHY VECTORIZE?

Jim Cownie

Overview

Fundamental trends

- Feature sizes
- Performance, memory bandwidth
- Energy

What they mean for architecture

Vectors

- History
- Now

Software

- What do I have to do to use them?

“It's tough to make predictions, especially about the future” So we look at the past…

Performance

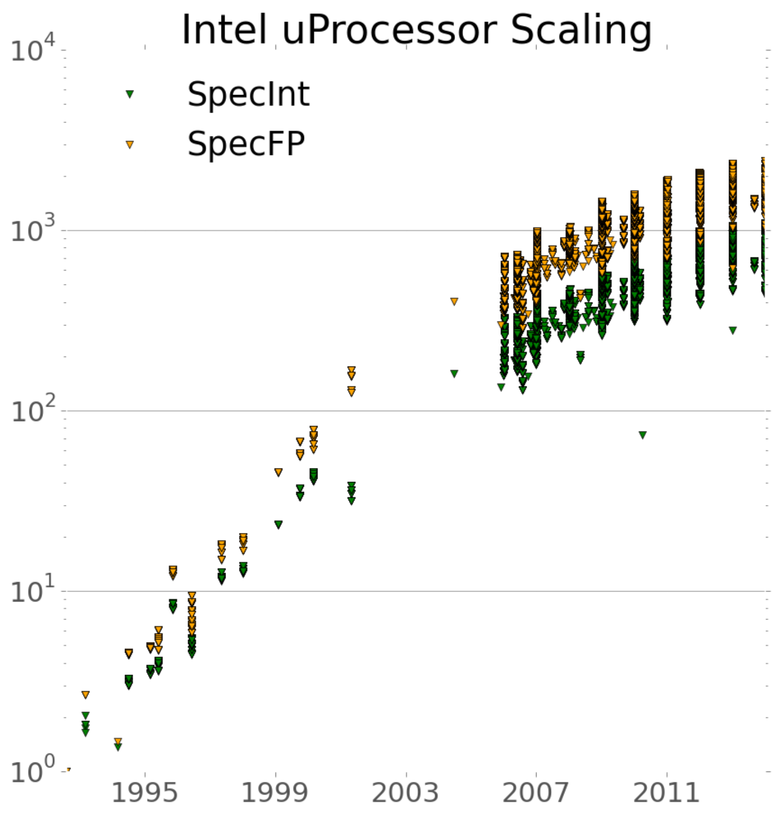
FP improvement higher than integer

Still rising, but not on trend

Why not?

Notes:

- Data from Stanford CPUdb
- This starts later than the next few graphs (no SPEC numbers before this!)
- Y-axis for these “Scaling” graphs is
  - Logarithmic
  - Ratio of metric/min(metric)

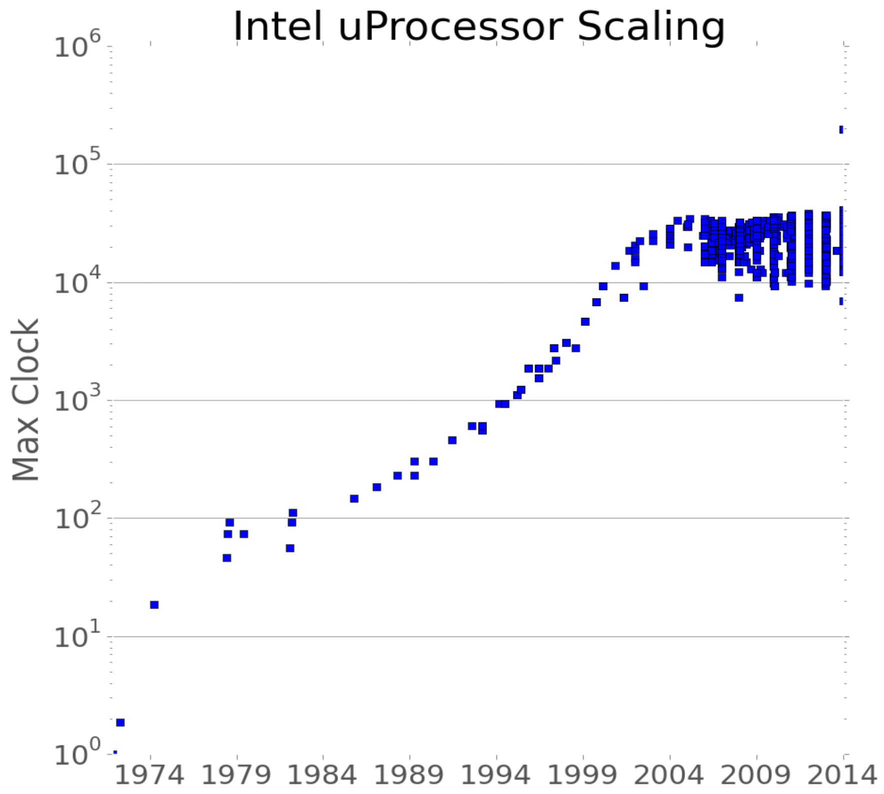


What about other metrics? Clock speed

Ouch!

Hardly any change since 2004

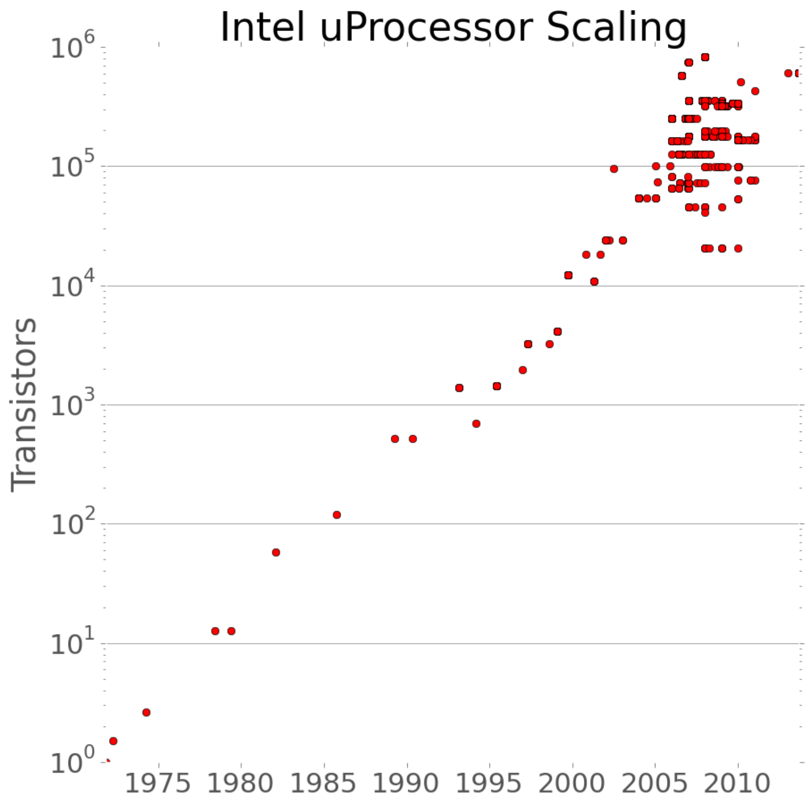
Performance improvements have to come from somewhere else



What about other metrics?
Number of Transistors

Moore's Law:

Available transistors double every (1½years, 2years, 2½years)

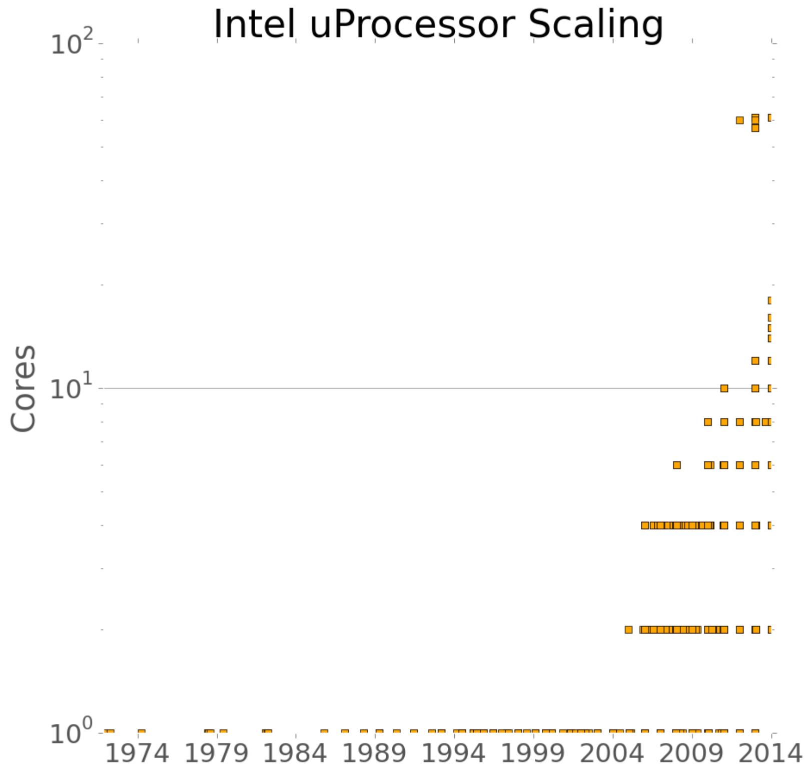


What about other metrics? Number of Cores

Didn't increase until after 2004

Has increased rapidly since

That's where lots of the transistors are going



What about other metrics?

Power

Increasing until ~2004

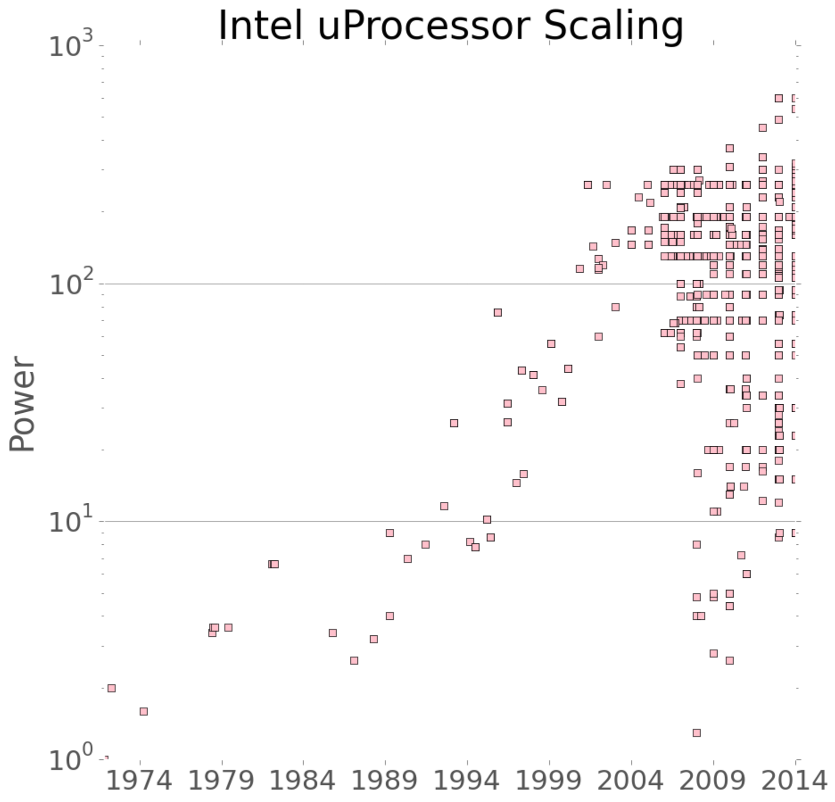
~Static or decreasing since

In some ways that's good

- Don't want to burn a hole in your pocket
- Energy efficiency has to improve

Power = $CV^2f + VI_{leakage}$

Power limit bounds frequency once Dennard scaling stops

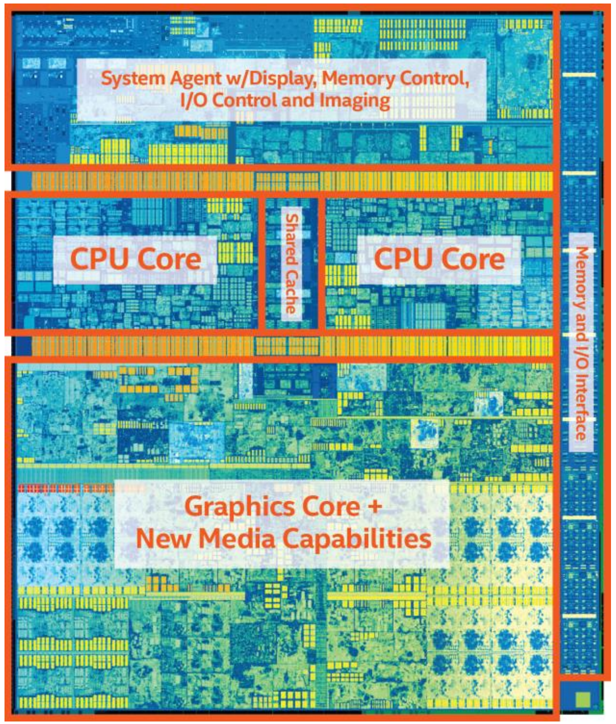


How do we use all the transistors?

Eat other system components
- Graphics
- Memory i/f
- PCI i/f

Add cache
Replicate cores
This is a laptop part.
- Two cores x two HW threads
- **256 bit (8 single or 4 double) SIMD FP unit.**
- **FMA (fused multiply add)**

Intel® 7th generation Core processor ("Kaby Lake") released January 2017

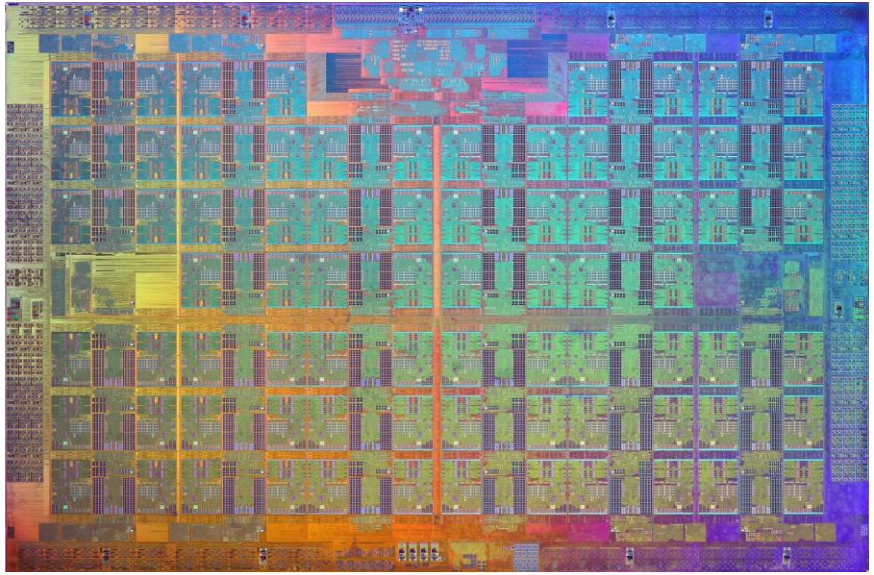

System Agent w/Display, Memory Control, I/O Control and Imaging
CPU Core
Shared Cache
CPU Core
Memory and I/O Interface
Graphics Core + New Media Capabilities

Data and thread parallelism are mandatory to extract all available performance.

How do we use all the transistors for HPC?

Intel® Xeon Phi™ 7200 series processor

- Up to 72 cores
- **Two 512 bit SIMD FPU** (16SP, 8DP, with FMA) per core
- **Vector Peak Perf:** 3+TF DP and 6+TF SP Flops
- 14nm process
- Boots normal OSes runs X86_64 code
- Fortran, C, C++, OpenMP*, MPI, Python,...
- Slower clock than mainstream Xeons
- 16GiB in package high BW MCDRAM



Data and thread parallelism are even more important here!

Fundamentals: Energy

ALU ops themselves are cheap

Locality (even on die) is important, and becomes critical in the future

- Register files and caches matter (must re-use data)
- Instruction decode matters (so need to execute fewer)

Nvidia* energy numbers	**2010 40nm**	**2017 10nm high freq**	**2017/2010**
DP Fused Multiply Add	50 pJ	8.7 pJ	17%
3x64bit read, 1x64bit write in 8K SRAM	56 pJ	9.6 pJ	17%
On die wire energy (256b moving 10mm)	310 pJ	200 pJ	**65%**
DRAM Access	10,000 pJ	1,700 pJ	17%

Vectors and vector registers help both issues

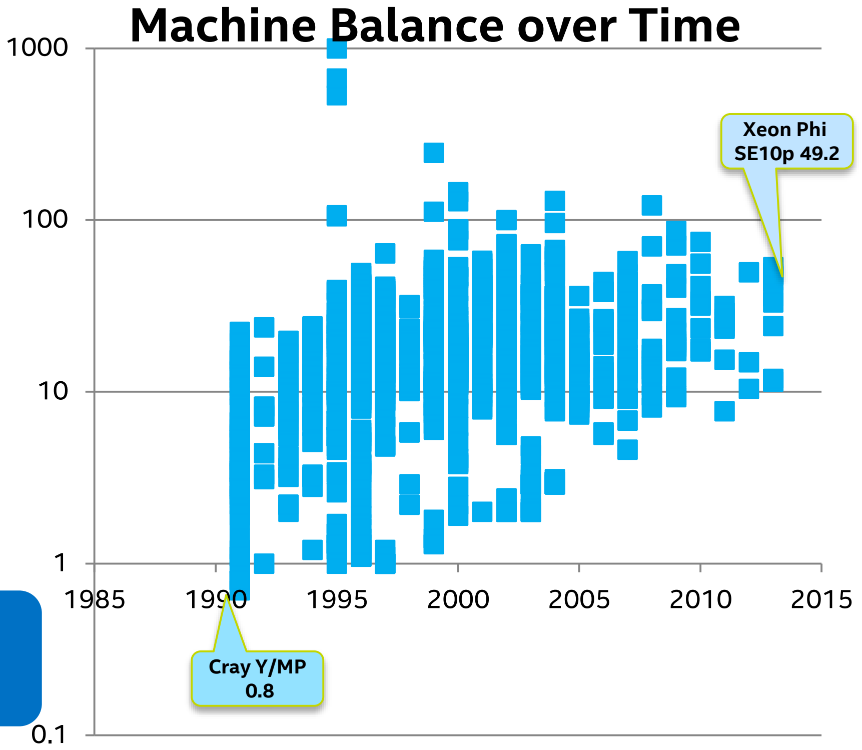
Memory Bandwidth keeps getting (relatively) worse

Smaller balance is better (it's flops/bw)

Lots of variation within a year

But the **best** machine now is >10x worse than in 1997

Now need > 10 ops per load or store to achieve peak FLOPS



History

Ancient history

- Illiac IV (1972)
- CDC Star (1974) – Memory to memory architecture
- Cray 1 – Vector registers

Lots of memory bandwidth/flop !

Main benefits come from good pipelining of operations

More recent history

X86

- SSE(1999), SSE2(2001), SSE3(2004), SSE4(2007) 8x128b XMM registers
- AVX1(2011), AVX2(2013) 16x256b YMM registers
- Intel® Xeon Phi™ coprocessor (2012) 16x512b ZMM registers, 2017: AVX512

ARM : Neon, Power: Altivec, …

Achieving Performance

Every code has a performance limiting resource

- Otherwise it would run faster until it did!

We need to understand what the limiting resource is

- Architect: so we can provide more of it
- Programmer: so we can rewrite our code to avoid it

The commonest limits for HPC codes are

- Memory bandwidth
- Floating point resources

Vectors help

- Efficient way to drive lots of ALUs
- Encourage efficient use of memory bandwidth

The Fundamental Idea

Expose data parallelism to the hardware

If it knows, it can exploit it, but the compiler has to tell it…

```
void xpy (float *x, float const * y, int len) {
    for (int i=0; i<len; i++)
        x[i] = x[i] + y[i];
}
```

We seem to be operating on a whole vector, but it's not obvious…

Can the compiler see it with this code?

No: src and dest can alias; need restrict qualifiers

Why can't that code just vectorize?

Consider this invocation

```
float x[10];
for (int i=0; i<10; i++)
    x[i] = (float)i;
xpy(&x[1],&x[0], 9);
```

```
void xpy(float *x,float const *y,int len)
{
    for (int i=0; i<len; i++)
        x[i] = x[i] + y[i];
}
```

Correct result is 0,1,3,6,10,15,21,28,36,45

If you force vectorization (with SIMD width of 4) you get 0,1,3,6,10,**9,11,13,15,17**

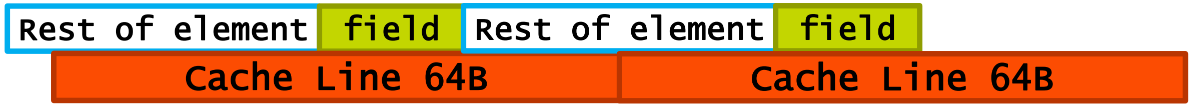
The compiler cannot vectorize unless you give it more information.

`restrict` on a pointer says "it does not alias" => no order between accesses via it and other pointers

Array of Structures and Structure of Arrays

C/C++ make it natural to write code in array of structures (AOS) form like this :-

```
void xpy (element *x, element const * y, int len) {
    for (int i=0; i<len; i++)
        x[i].field = x[i].field + y[i].field;
}
```



Inefficient; requires scatter/gather and wastes memory bandwidth

Better to rework to structure of arrays form (but a hard change in a large codebase ☹)

What does the compiler need to know?

Aliasing information

- Use `restrict` (or Fortran which has always specified no overlap of arguments)
- Use `omp simd` or other pragmas (there are way too many!)

Alignment information `omp simd aligned`

Expected iteration counts

- Pragmas for these too

Look at vectorization reports `–vec-report`*n*

When the hints fail… force the issue

- Use OpenMP 4.5 `simd` directives

  `#pragma omp simd` (`!$omp simd` in Fortran)
- Can annotate vector callable functions
- Can be used without also using OpenMP parallelism (to mix with Intel® Threading Building Blocks, for instance) `–fopenmp-simd`

Low level interface: Intrinsics

Another way of writing something very close to assembler code.

Like this

```
register __m128d sigma2;
sigma2  = _mm_setzero_pd()
v_8_9   = _mm_load_pd(&Matrix[0][8]);
xsigma1 = _mm_add_pd(_mm_mul_pd(v_8_9,v_8_9),xsigma1);
```

Explicitly operating on SIMD-width vectors

Slightly higher than machine code, but not much! (note the explicit 128b width!)

Low level interface

Plus

- Highest performance is possible
- You control the horizontals and the verticals

Minus

- Like all ASM code it's hard work writing this stuff
- Not portable
  - Certainly not to non-X86
  - May not be highest performance on future cores (e.g. SSE to AVX2 or AVX2 to KNC or AVX512)

Use only if you really need to (I try to avoid them)
Read the compiler's asm output before you start.

Why Vectorize rather than Parallelize?

Parallelization redistributes work to get you an answer sooner

- Using four cores I may be able to execute 4 CPU seconds of work in ~1 elapsed second (but I still consumed 4 CPU seconds of resource)

Vectorization reduces the total amount of work

- Using vectorization I may be able to compute 4 times faster on a single CPU, reducing that 4 CPU seconds to 1 CPU second

If you’re charged for machine time vectorization should save you money, whereas parallelization won’t!

Memory Bandwidth

Often the limiting factor in HPC codes, rather than raw FLOPS

Intel® Advisor can help

- Roofline model
- Vectorization reports integrated back into code

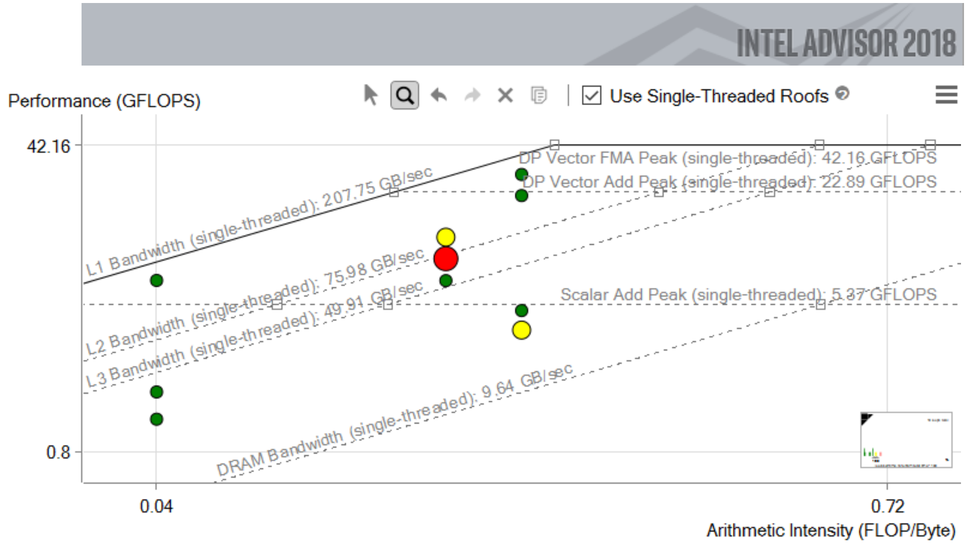
New!

Find Effective Optimization Strategies

Intel Advisor: Cache-aware roofline analysis

Roofline Performance Insights

- Highlights poor performing loops
- Shows performance “headroom” for each loop
  - Which can be improved
  - Which are worth improving
- Shows likely causes of bottlenecks
- Suggests next optimization steps



New!

Find Effective Optimization Strategies

Intel Advisor: Cache-aware roofline analysis

Roofs Show Platform Limits

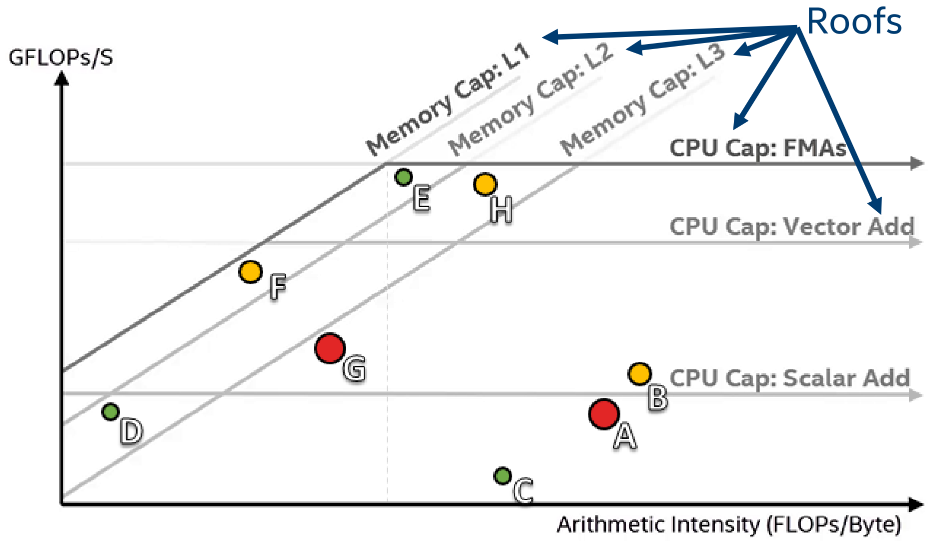
- Memory, cache & compute limits

Dots Are Loops

- Bigger, red dots take more time so optimization has a bigger impact
- Dots farther from a roof have more room for improvement

Higher Dot = Higher GFLOPs/sec

- Optimization moves dots up
- Algorithmic changes move dots horizontally



Which loops should we optimize?

- A and G are the best candidates
- B has room to improve, but will have less impact
- E, C, D, and H are poor candidates

Roofline tutorial video

Conclusions

Vectors are here to stay

Code needs to be designed with that in mind

- SOA instead of AOS
- Give the compiler all the information you can

On Intel® Xeon Phi™ processor and AVX-512 enabled Xeons you're only using 1/16th of the SP float capability if you haven't vectorized

Even on a laptop Intel® processors you're down by 8x today!

Good Luck!

References

Stanford CPUDB http://cpudb.stanford.edu/
Energy numbers from Nvidia
http://research.nvidia.com/publication/gpus-and-future-parallel-computing
page 9
Stream http://www.cs.virginia.edu/stream/

Free Software

As students at a degree awarding institution you qualify:-

https://software.intel.com/en-us/qualify-for-free-software/student

Free download of Intel® Parallel Studio XE

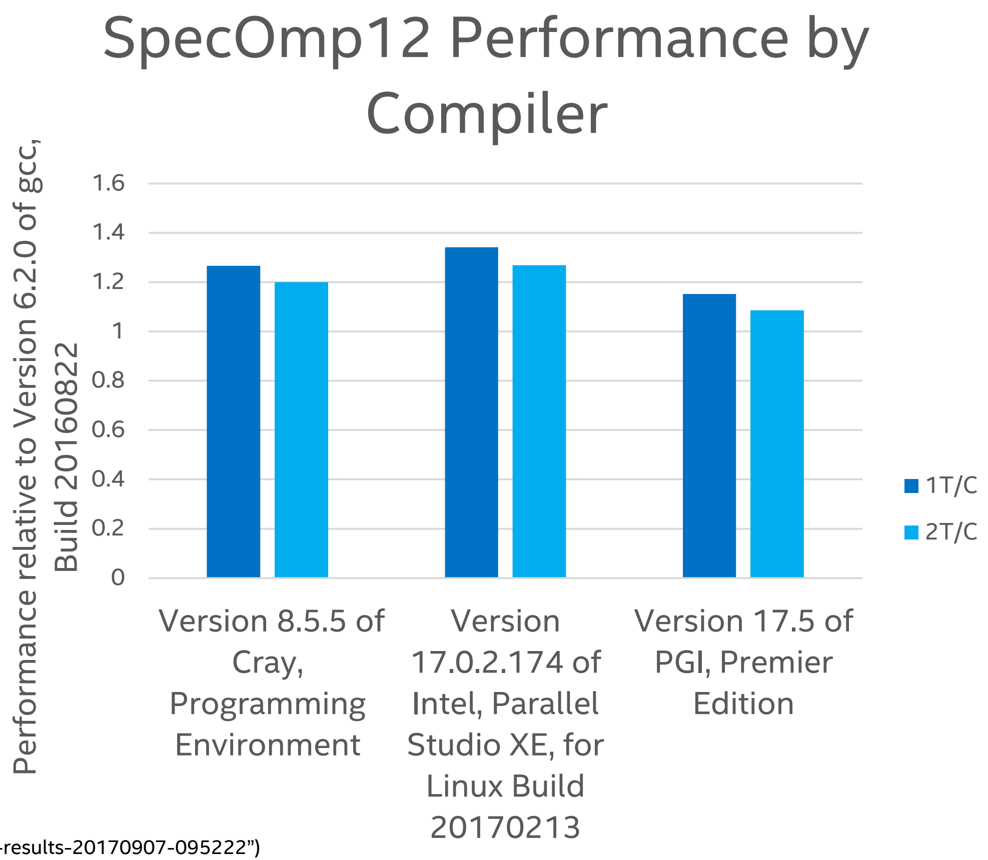
BACKUP

Legal Disclaimer & Optimization Notice

INTEL® COMPILER PERFORMANCE

- Compare SpecOMP12 on identical hardware
- Single node in a Cray* XC30: 2xIntel® Xeon E5-2697 v2, Aug 2017 data collected by Indiana University
- It is hard to find such results; most vendor runs just use Intel® compilers
  - I wonder why? ☺
  - I couldn't find SpecINT/FP runs to compare!

>20% performance improvement over GNU compilers!



Data from http://spec.org downloaded on 8 September 2017 ("omp20212-results-20170907-095222")