Savitzky-Golay Filter Algorithm for Large One-Dimensional Data Sets

Prof. Jeffrey S. Dunham
William R. Kenan Jr. Professor of Natural Sciences
Middlebury College
Middlebury, Vermont
dunham@middlebury.edu

July 11, 2017
Colfax Research MC² 004: Signal Processing
https://colfaxresearch.com/mc2-series/

Middlebury College

Department of Physics

Outline

1. Chaos experiment - physics

2. Savitzky-Golay (SG) filter for chaos data set – "embarrassingly parallel" stencil calculation

3. Compute-only ("speed of light") code for KNL

“Big” data in a “small” lab

National Instruments NI-USB-6251

1.25 MS/s (MS = MegaSamples)
16-bit resolution A/D
counters, D/A, etc.

USB with laptop or desktop

NI LabVIEW data acquisition software

108 GS/24 hr
216 GB file/24 hr

“If you build it, they will come.” *Field of Dreams* [altered slightly]

Videos

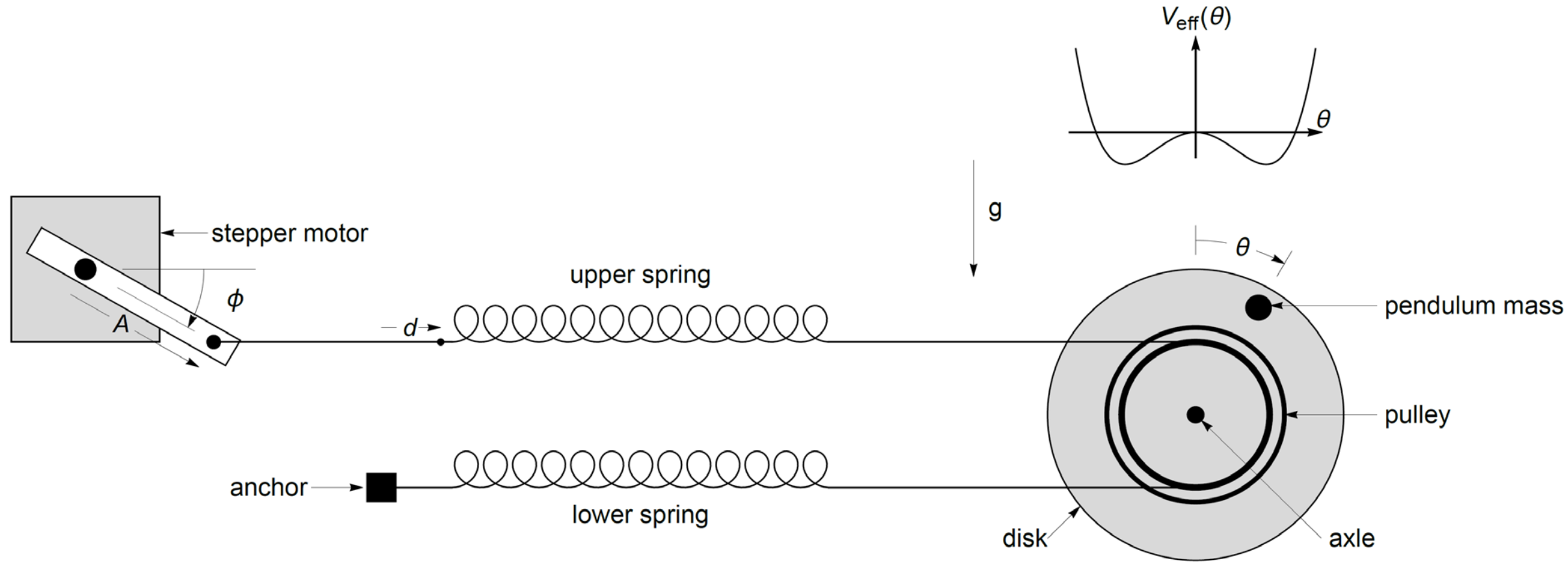
The driven chaotic pendulum in action
http://go.middlebury.edu/dunham-lab-pendulum-motion.mp4

4096 Poincaré sections per drive motor revolution
http://go.middlebury.edu/dunham-lab-pendulum-motion-plot.mp4

Kinematics and mechanics reminder:

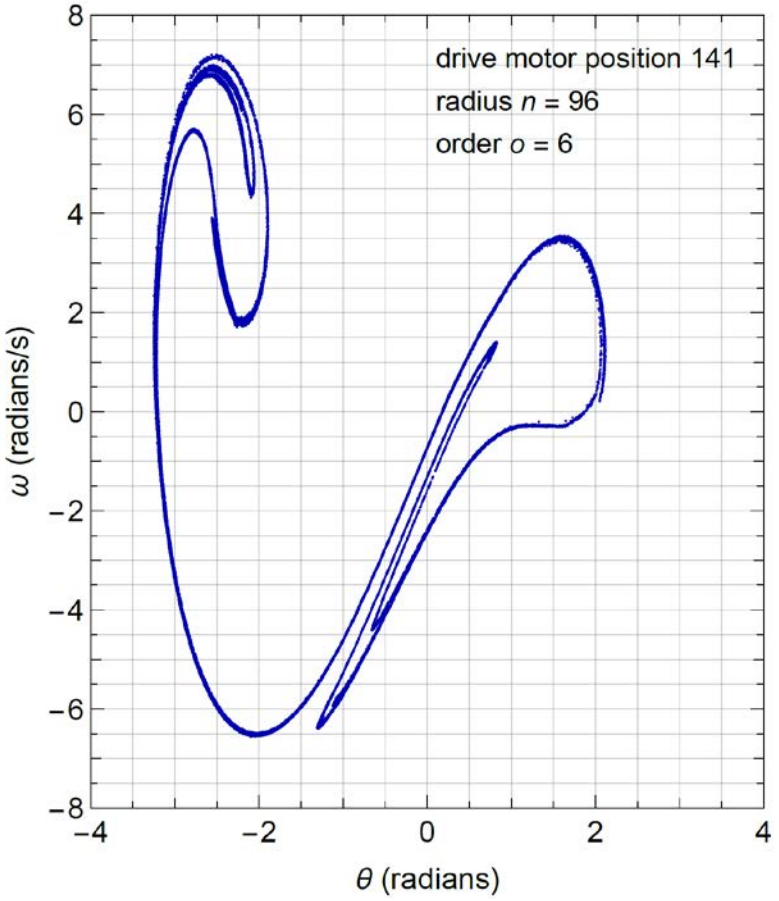
angular position: θ	linear position: x
angular velocity: $\omega \equiv \frac{d\theta}{dt}$	linear velocity: $v \equiv \frac{dx}{dt}$
angular acceleration: $\alpha \equiv \frac{d\omega}{dt}$	linear acceleration: $a \equiv \frac{dv}{dt}$
angular Newton's second law: $\tau_{\text{net ext}} = I\alpha$	linear Newton's second law: $F_{\text{net ext}} = ma$

Apparatus

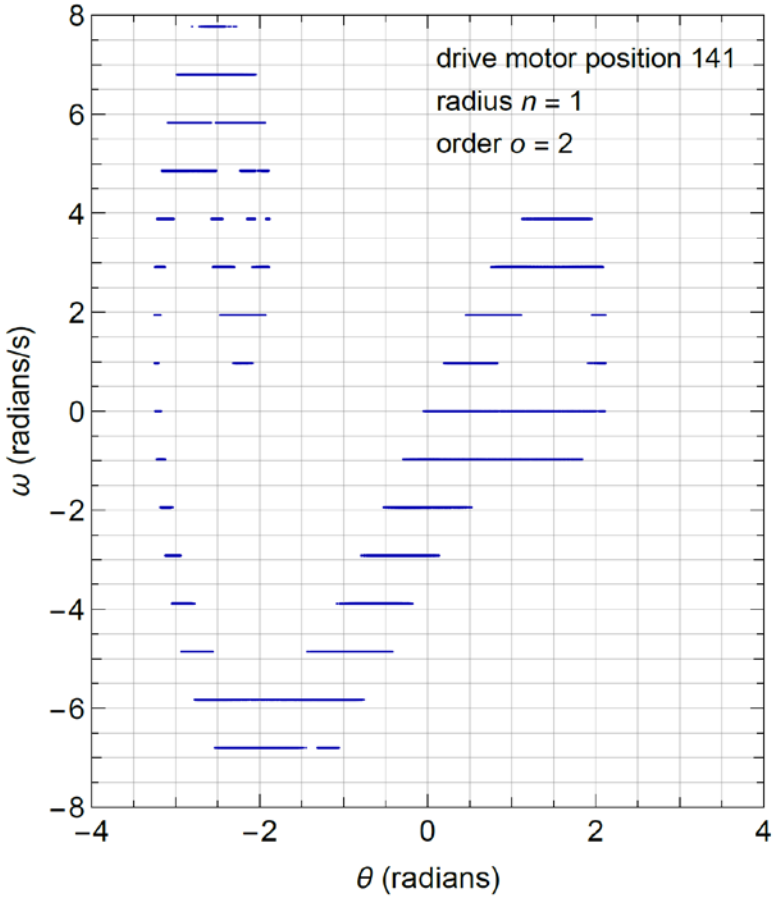

$V_{eff}(\theta)$
θ
g
θ
stepper motor
ϕ
A
upper spring
d
pendulum mass
pulley
anchor
lower spring
disk
axle
4,096 ticks/rev
10,000 ticks/rev

“Right” SG filter parameters *radius* (= *n*) and *order* (= *o*)

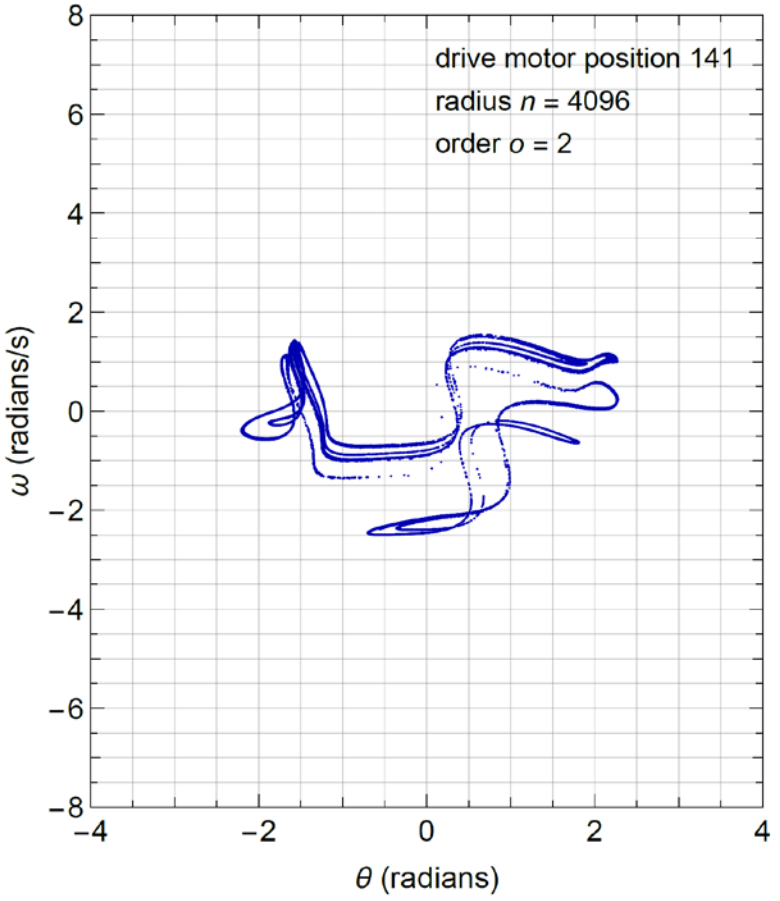
“just right”



“too little”



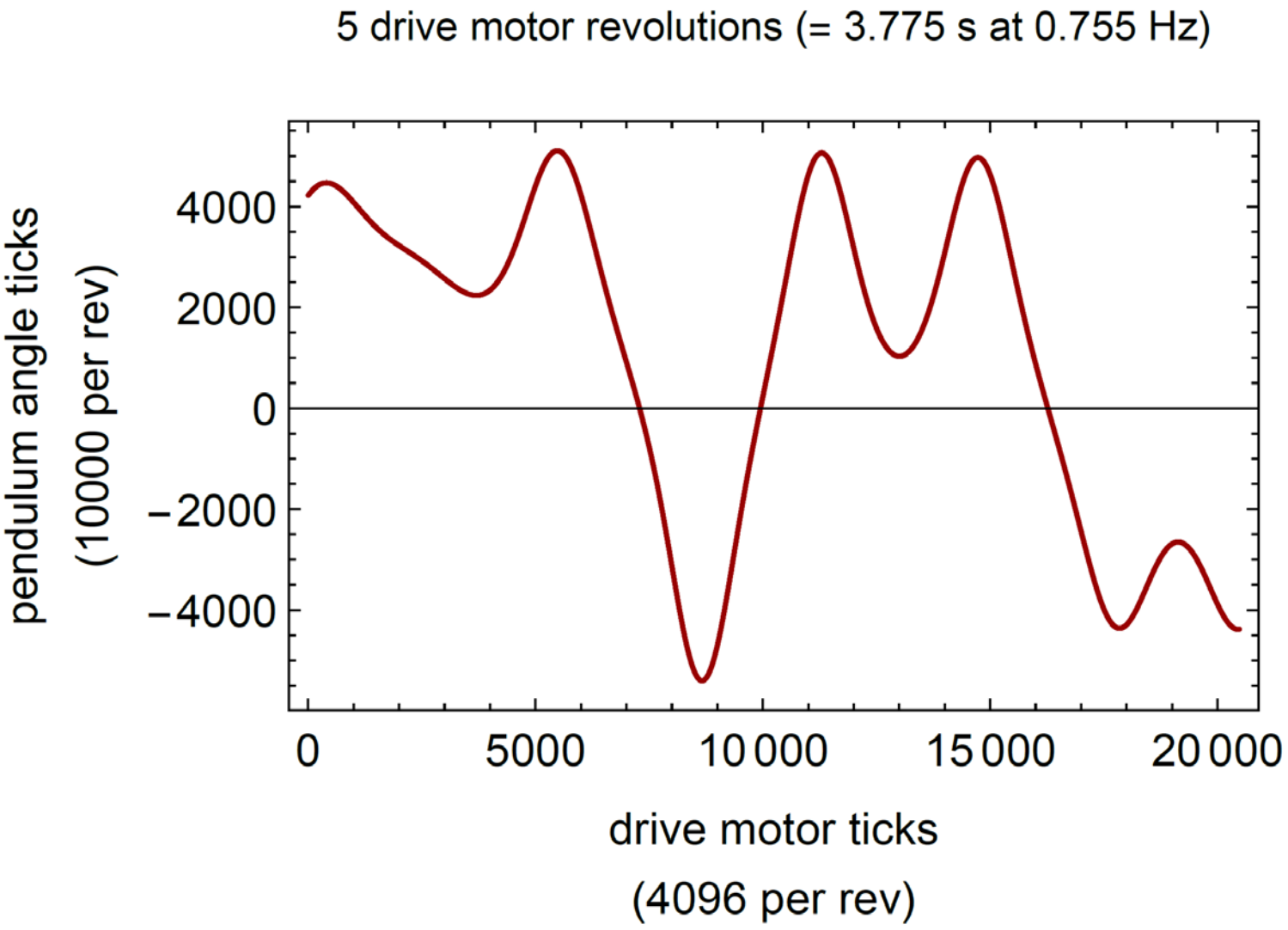
“too much”



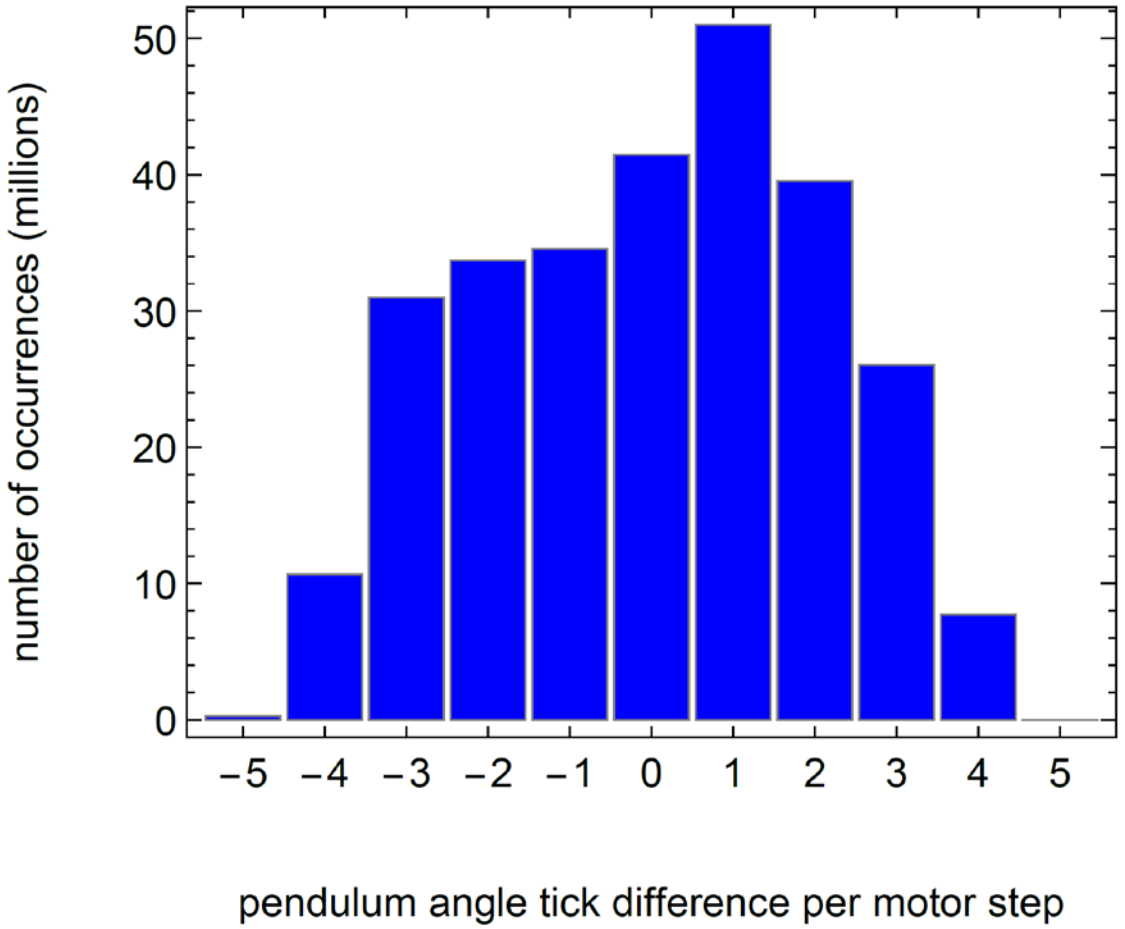
Raw Data

32-bit integers:

4219
4220
4221
4222
. . .
-4379
-4379
-4379
-4379
-4379



Finding derivatives from small integer differences!



24 hours of data in histogram.

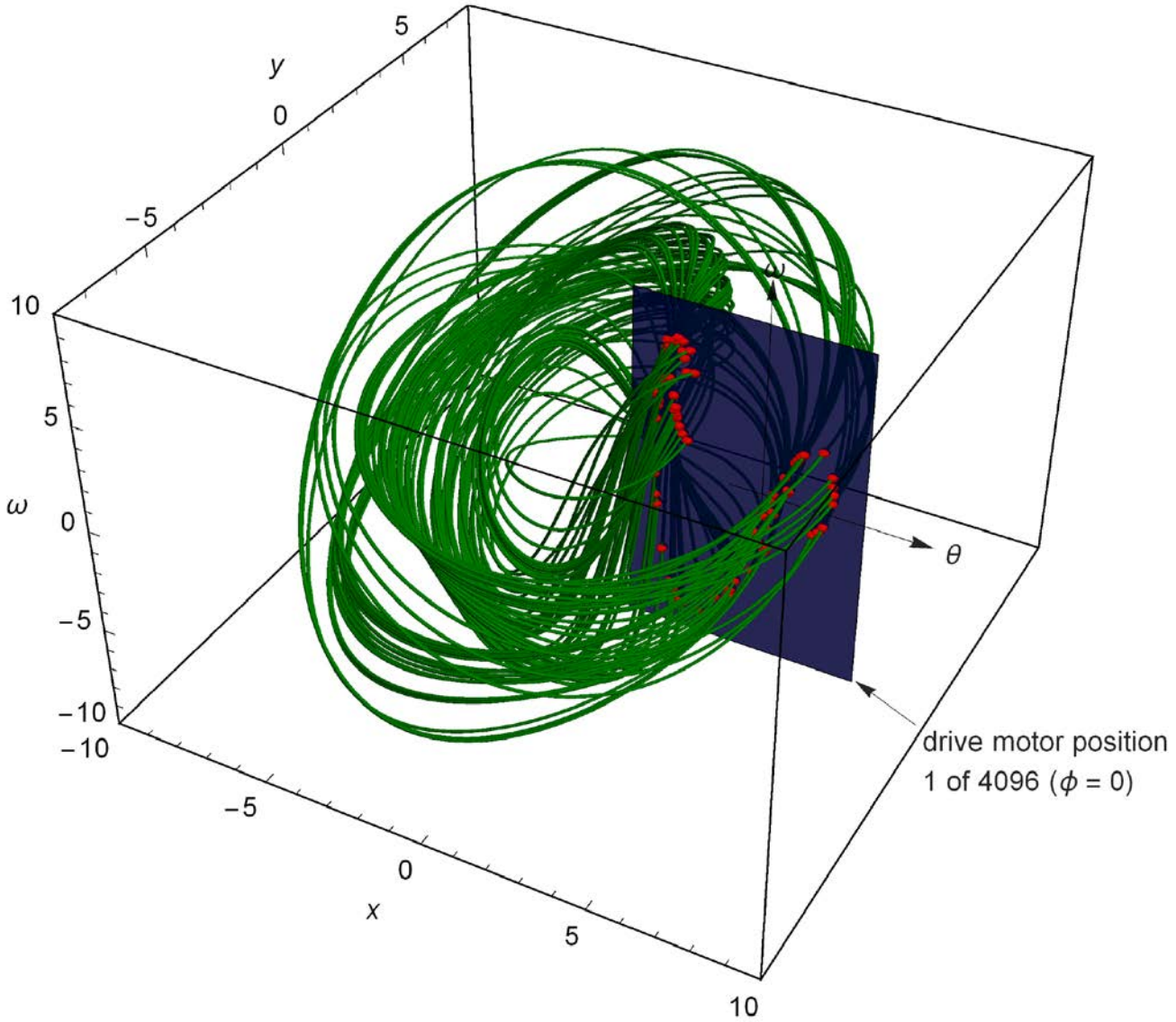
Pendulum moves 5 or fewer ticks (CW or CCW) per drive motor step!

Tick counts are integers.

Derivatives obtained from small integer differences at adjacent motor steps, $\omega = \Delta\theta / \Delta t$ and $\alpha = \Delta\omega / \Delta t$, will form a small set of discrete values.

Need to find derivatives from large neighborhood (stencil) around a given point.

Chaotic trajectory in phase space



100 drive motor rotations

major angle around torus is drive motor angle ϕ

red spheres indicate phase (θ, ω) of pendulum when drive motor in position 1 of 4096 (ϕ)

4096 Poincaré planes around torus

phase-space trajectory is non-intersecting

Phenomena we see

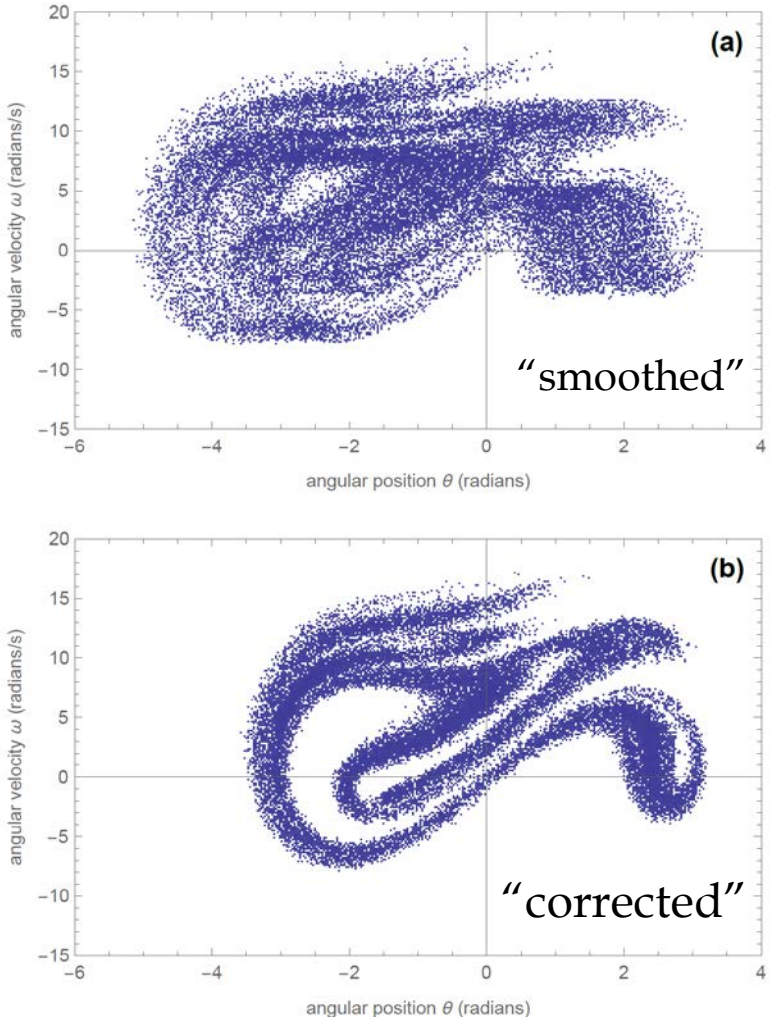
1. Periodic
   period-1, period-2, period-3, period-5, period-7, etc.
   (data-taking times: minutes)

2. Chaos
   data-taking times: 24 hours minimum
   10-50 days for excellent quality Poincaré sections
   (stability of apparatus components?)

Experiment history

Original paper:

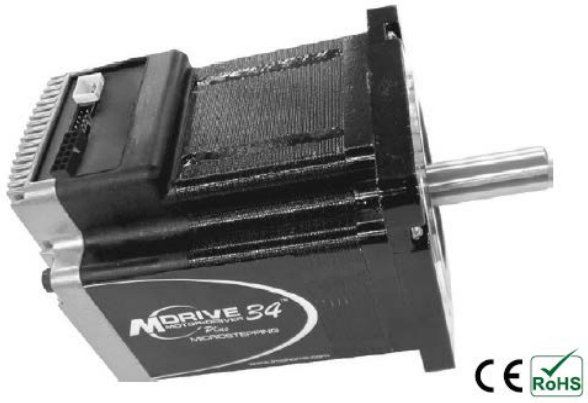
Robert DeSerio, “Chaotic pendulum: The complete attractor”, *American Journal of Physics* **71**, March 2003, pp. 250-257.



Junda (2009) our first attempts →

Experiment improvements

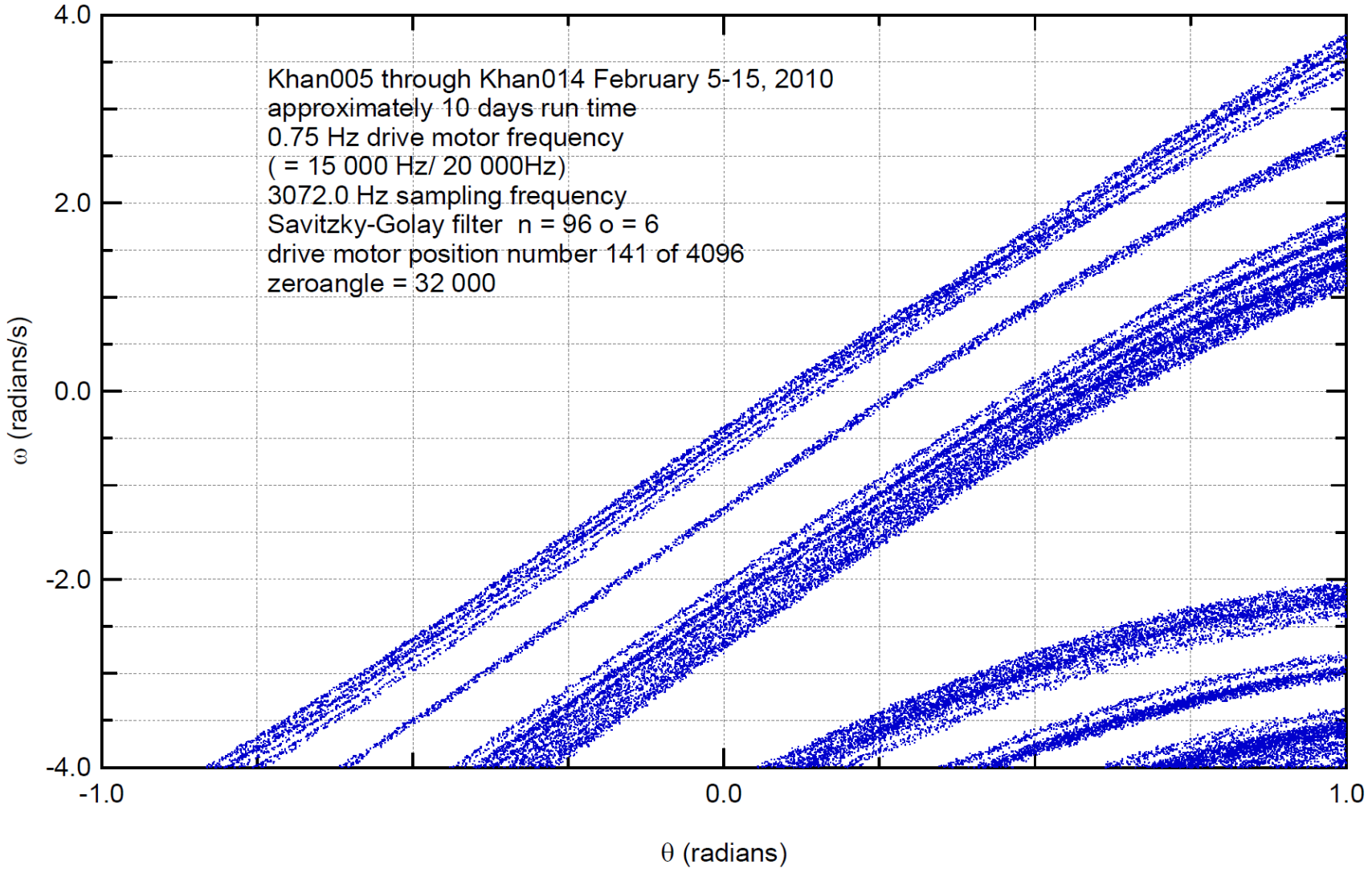
Intelligent Motion Systems MDrive34Plus microstepping drive motor with integrated 4096 ticks/revolution optical rotary encoder



US Digital optical rotary encoder 10,000 ticks/revolution

(coming soon: 40,000 ticks/revolution)

Improved resolution of fractal



Understanding the data

- Correctness!
- Fractal dimension
- Lyapunov exponents
- Optimal SG filter parameters
- Figure of merit for a fractal extracted from experimental data
- Quantify discretization and environmental “noise” contributions

None of these possible without filtered (“smoothed”) data!

Typical 24-hour run

0.755 Hz drive motor frequency

Raw Input:
276,030,000 32-bit integers pendulum angle ticks (1.1 GB file)

SG Filter Output:
275,988,480 (= 67,380 × 4,096) DP (or SP) smoothed values of θ, ω, and α
67,380 full motor rotations (67,380 points per Poincaré section)

Why use Savitzky-Golay (SG) Filter?

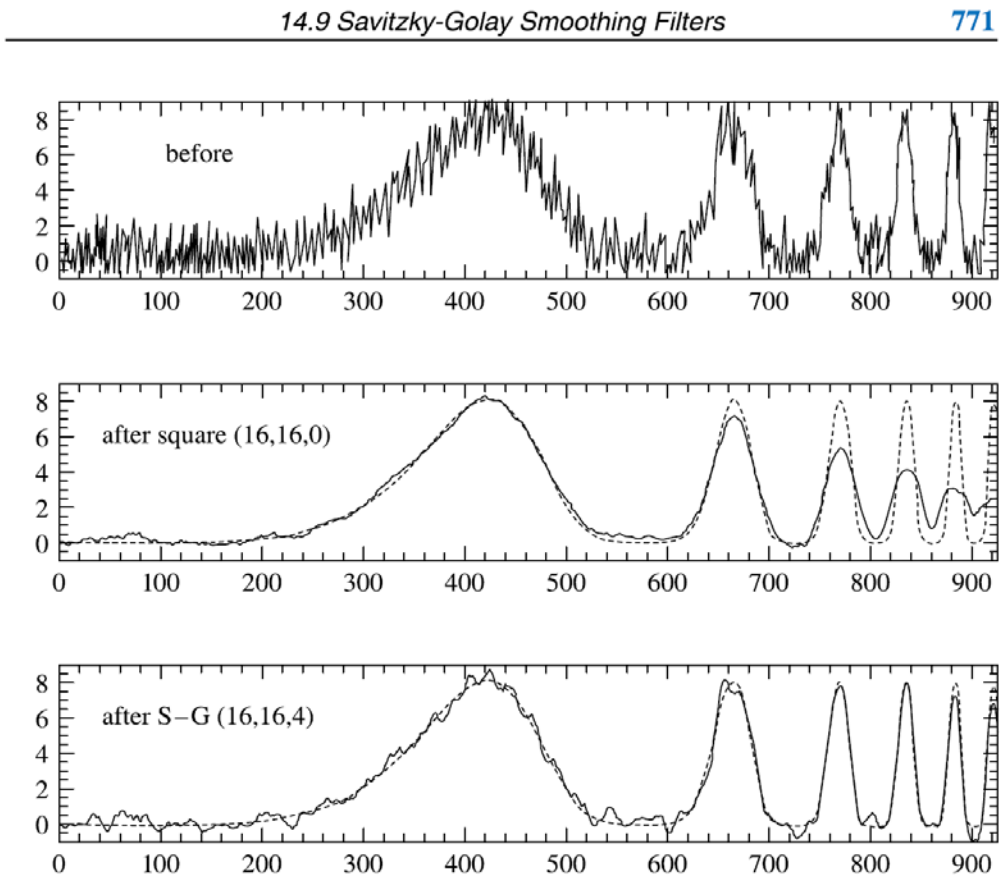


Figure 14.9.1. Top: Synthetic noisy data consisting of a sequence of progressively narrower bumps and additive Gaussian white noise. Center: Result of smoothing the data by a simple moving window average. The window extends 16 points leftward and rightward, for a total of 33 points. Note that narrow features are broadened and suffer corresponding loss of amplitude. The dotted curve is the underlying function used to generate the synthetic data. Bottom: Result of smoothing the data by a Savitzky-Golay smoothing filter (of degree 4) using the same 33 points. While there is less smoothing of the broadest feature, narrower features have their heights and widths preserved.

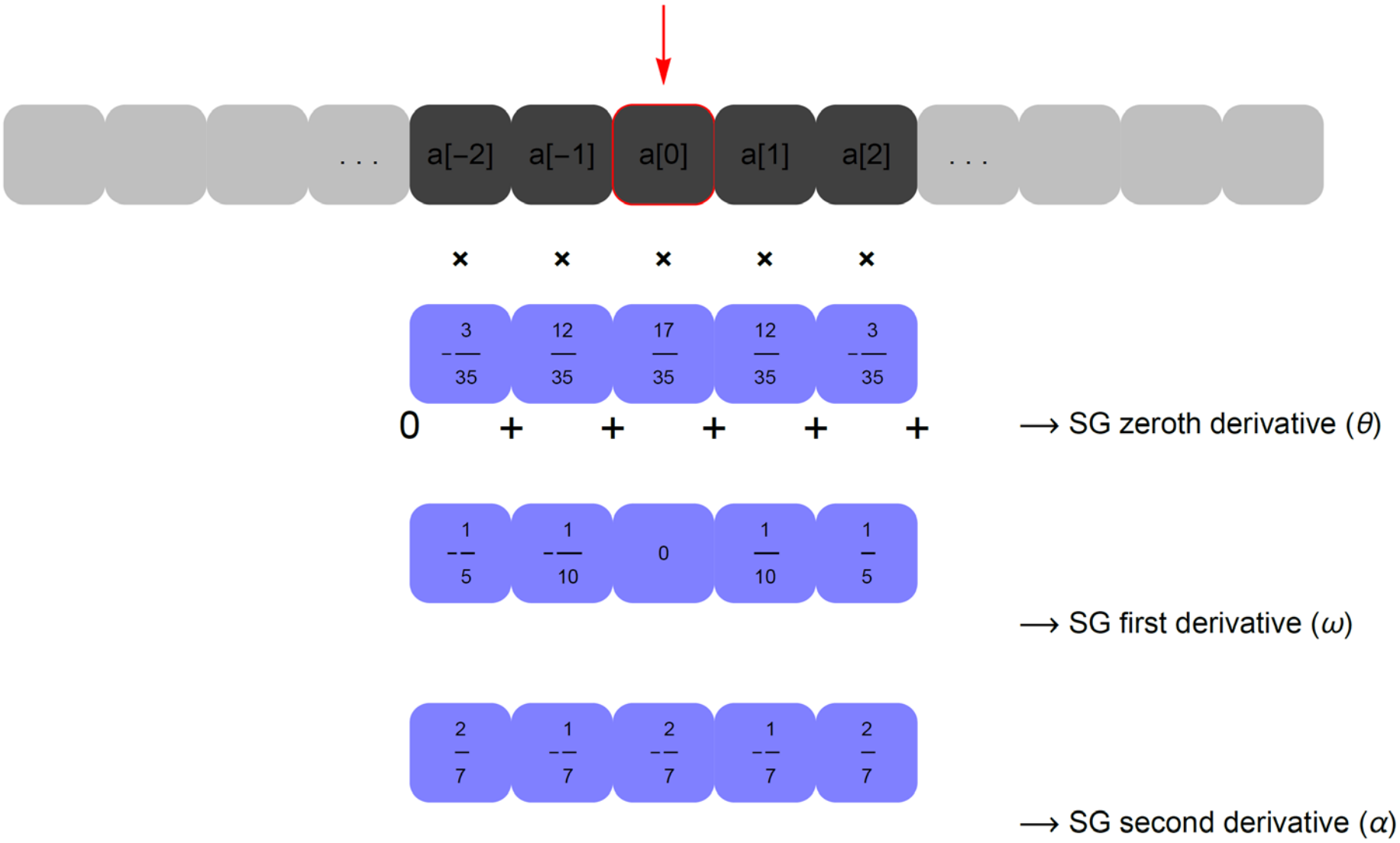
Preserves heights and widths of features in data.

We need smoothed 0^{th}, 1^{st}, and 2^{nd} derivatives of data: θ, ω, and α.

But, computationally expensive.

Numerical Recipes: The Art of Scientific Computing, 3rd edition, W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, (Cambridge University Press, 2007) pp. 766-772

“Small” SG filter example: $n = 2$, $order = 2$ (quadratic)



How many FLOPs for a one-day data set?

320 billion ≈ 275,988,480 (data set values)

× 193 (= 96 + 1 + 96, for n = 96 SG-filter radius)

× 2 (FMA = fused multiply-add)

× 3 (smoothed derivatives: θ, ω, and α)

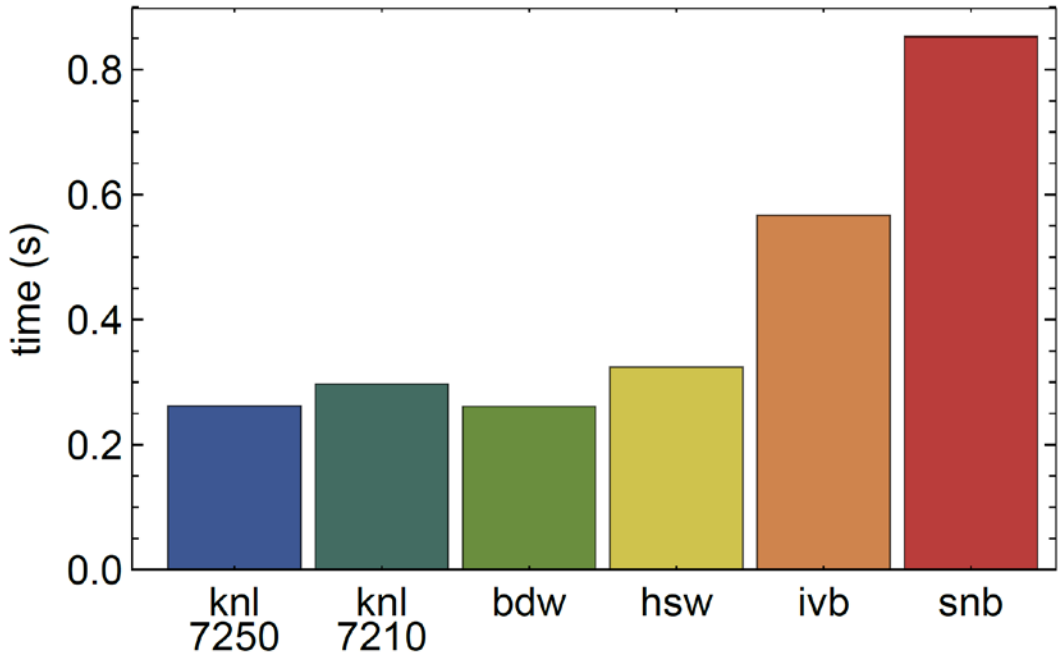
(This calculation took more than 3 hours seven years ago!
Less than ½ second now on KNL.)

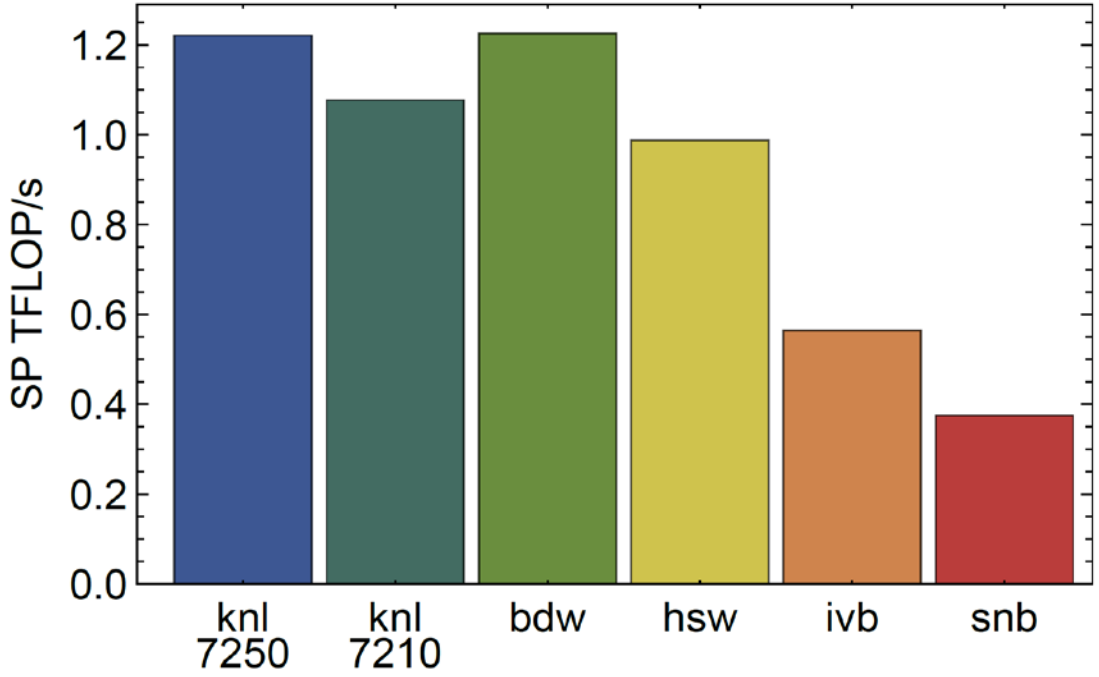
Code snippet for SG filter

```
181     omp_set_num_threads(128);
182
183 #pragma omp parallel for schedule(guided)
199 for(int j = 0 ; j < nSelectedDataElements; j++) //FOLDED SG loop version
200 {
201
202         register realtype temp0 = realtypebuffer[j + offset + (nSGElements-1)/2]\
203          * realtypebufferSG0[(nSGElements-1)/2];
204         register realtype temp1 = realtypebuffer[j + offset + (nSGElements-1)/2]\
205          * realtypebufferSG1[(nSGElements-1)/2];
206         register realtype temp2 = realtypebuffer[j + offset + (nSGElements-1)/2]\
207          * realtypebufferSG2[(nSGElements-1)/2];
208
209
210 #pragma unroll(2)
211 #pragma omp simd
212         for (int k = 0; k < (nSGElements-1)/2; k++) {
213
214         temp0 += (realtypebuffer[j + offset + k] + realtypebuffer[j + \
215                 offset + (nSGElements-1) - k])  * realtypebufferSG0[k];
216         temp1 += (realtypebuffer[j + offset + k] - realtypebuffer[j + \
217                 offset + (nSGElements-1) - k])  * realtypebufferSG1[k];
218         temp2 += (realtypebuffer[j + offset + k] + realtypebuffer[j + \
219                 offset + (nSGElements-1) - k])  * realtypebufferSG2[k];
220
221         }
222
223         derivative0[j] = temp0;
224         derivative1[j] = temp1;
225         derivative2[j] = temp2;
226 }
```

VITO ≡
vectorize inner,
thread outer

Single-Precision SG Filter (320 billion SP FLOPs)

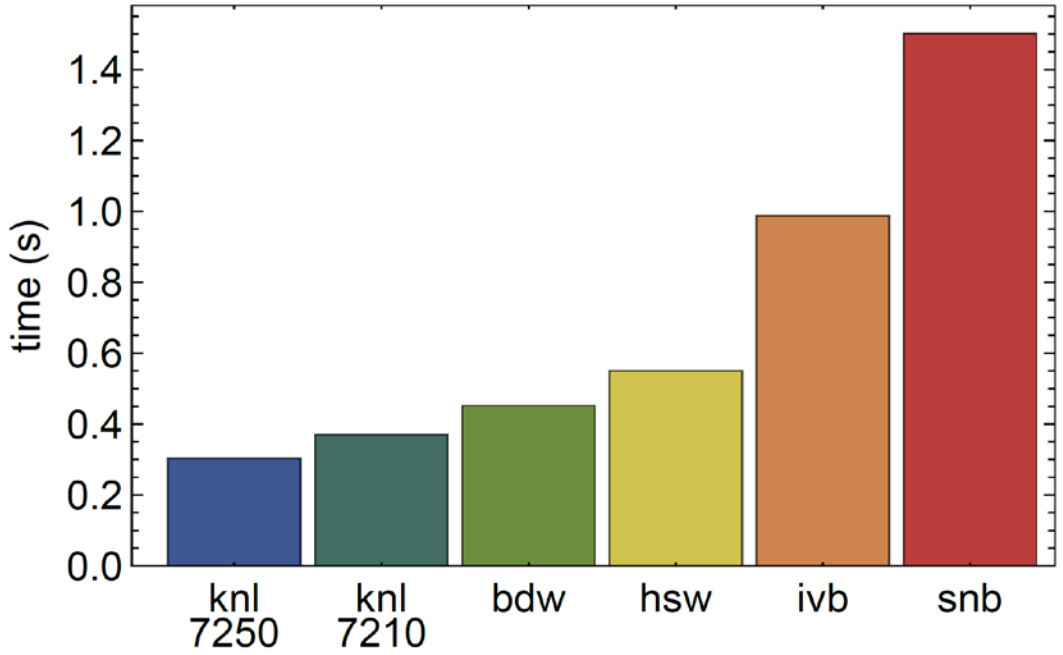


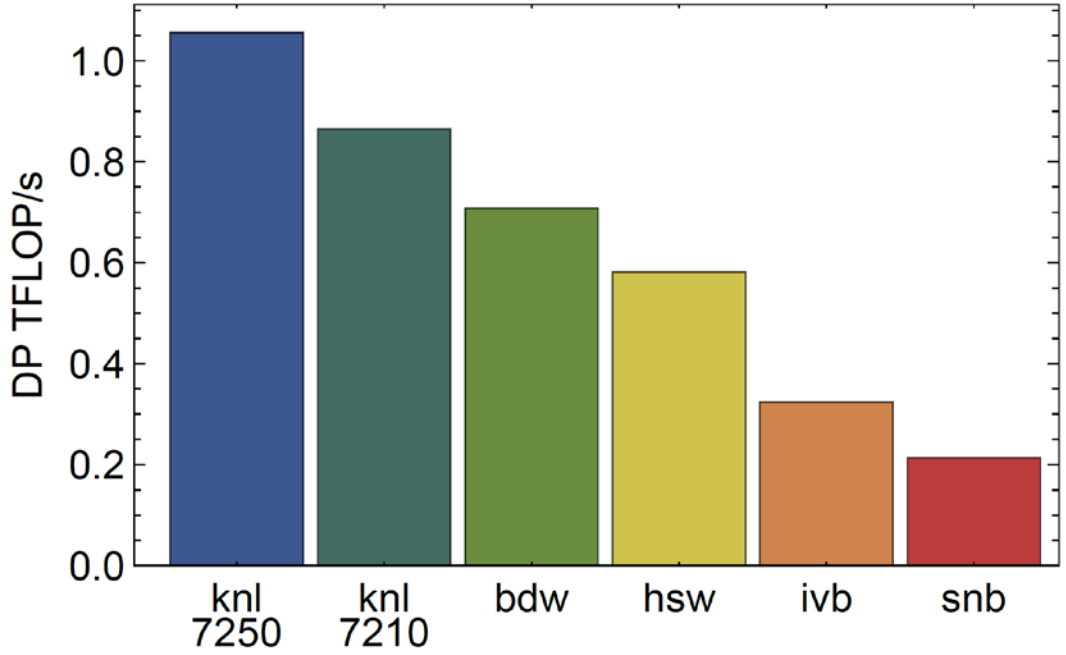


snb	"Sandy Bridge"	Xeon e5-2690	AVX	32
ivb	"Ivy Bridge"	Xeon e5-2697v2	AVX	48
hsw	"Haswell"	Xeon e5-2697v3	AVX2	56

bdw	"Broadwell"	Xeon e5-2699v4	AVX2	88
knl7210	DAP workstation	Xeon Phi 7210	AVX512	256
knl7250	Colfax Cluster	Xeon Phi 7250	AVX512	272

Double-precision SG filter (320 billion DP FLOPs)

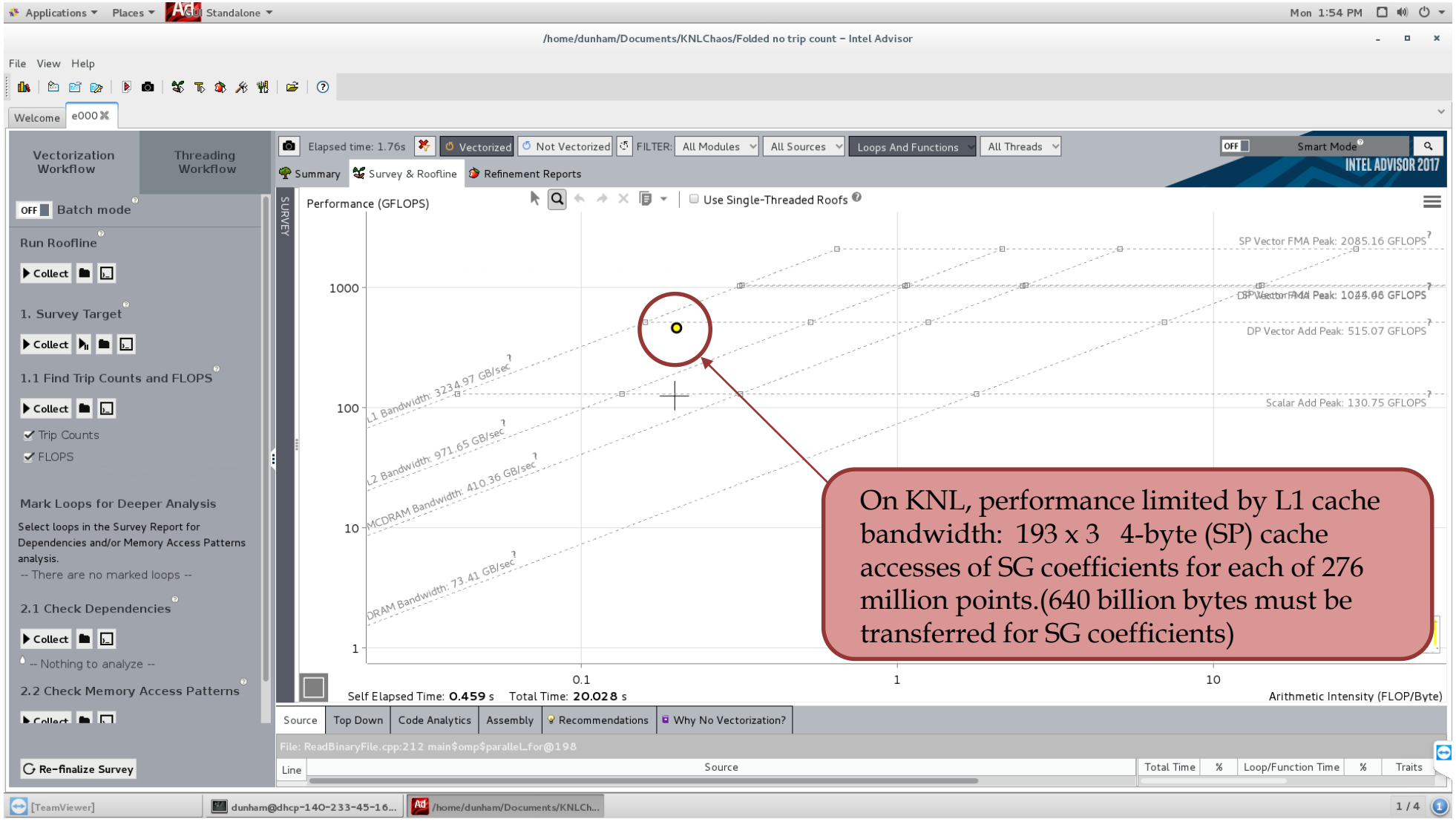




snb	"Sandy Bridge"	Xeon e5-2690	AVX	32
ivb	"Ivy Bridge"	Xeon e5-2697v2	AVX	48
hsw	"Haswell"	Xeon e5-2697v3	AVX2	56

bdw	"Broadwell"	Xeon e5-2699v4	AVX2	88
knl7210	DAP workstation	Xeon Phi 7210	AVX512	256
knl7250	Colfax Cluster	Xeon Phi 7250	AVX512	272

Intel Advisor Roofline Analysis



Welcome back to 2001?

1 threads	O0	O1	O2	O3
SSE2	2431.54	238.64	129.73	117.15
SSE3	2426.30	238.57	138.53	116.10
SSSE3	2425.41	238.47	138.68	116.04
SSE4.1	2425.94	238.67	138.44	116.17
SSE4.2	2426.56	238.62	131.06	117.14
AVX	2426.34	197.19	97.20	68.79
CORE-AVX2	2426.26	174.92	62.42	43.80
MIC-AVX512	2426.30	152.45	55.81	30.55

4000 times slower

64 threads	O0	O1	O2	O3
SSE2	37.83	3.80	2.28	1.89
SSE3	38.06	3.82	2.43	1.88
SSSE3	37.95	3.80	2.40	1.87
SSE4.1	37.99	3.79	2.40	1.87
SSE4.2	37.99	3.80	2.28	1.87
AVX	38.00	3.15	1.78	1.14
CORE-AVX2	37.99	2.79	1.27	0.76
MIC-AVX512	38.19	2.45	1.17	0.59

128 threads	O0	O1	O2	O3
SSE2	24.86	3.76	2.19	1.80
SSE3	25.02	3.77	2.35	1.81
SSSE3	24.92	3.74	2.36	1.80
SSE4.1	24.92	3.73	2.34	1.78
SSE4.2	24.95	3.76	2.17	1.78
AVX	24.87	3.11	1.68	1.13
CORE-AVX2	24.89	2.77	1.20	0.77
MIC-AVX512	24.93	2.46	1.09	0.64

Time, in seconds, on Xeon Phi 7210 for SG filter, using double-precision (DP), to analyze 24-hour data set for different instruction sets and compiler option settings, using Intel compiler.

Compute-only code ("speed of light" code)

No memory accesses whatsoever.

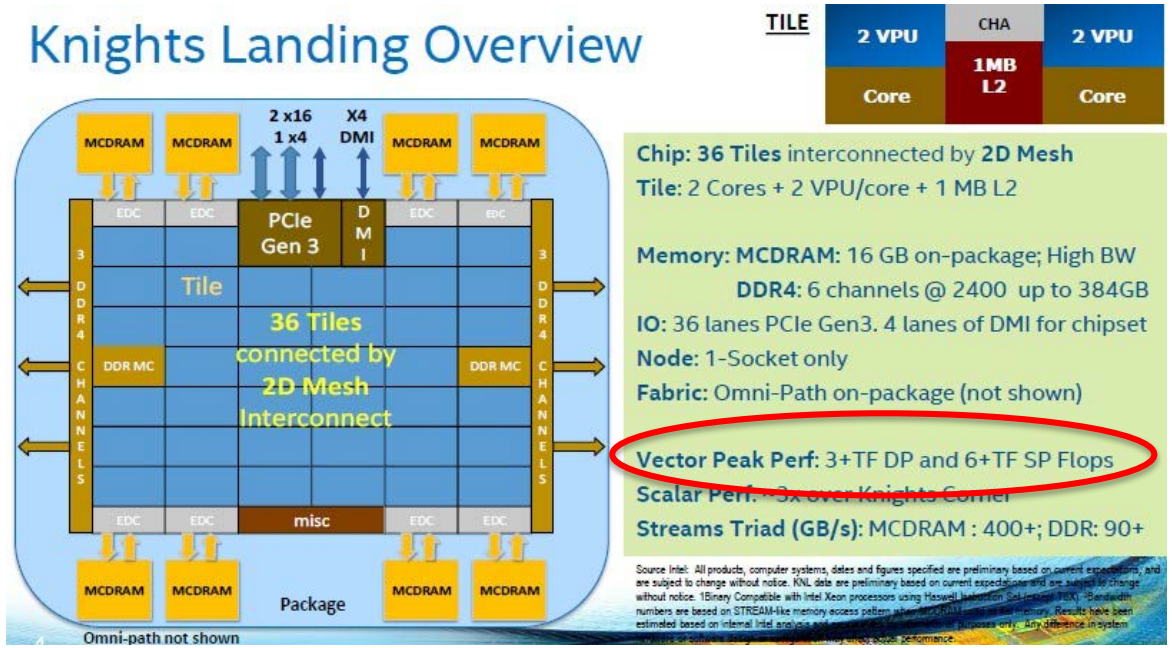
Registers and VPU only. Fused multiply-add (FMA) instructions only.

How to verify

3+ TFLOP/s DP

6+ TFLOP/s SP

stated in various places?



Compute-only ("speed of light") code expectations

(128 threads on 64-core Xeon Phi 7210 DAP workstation, nominal 1.3 GHz* clock)

SP: 4.51 TFLOP/s = (64 core)×(2 VPU/core)×(2 FLOP/FMA)
×(16 SP operands/vector)×(~~1.3~~ 1.1×10^9 clock/s)

DP: 2.25 TFLOP/s = (64 core)×(2 VPU/core)×(2 FLOP/FMA)
×(8 DP operands/vector)×(~~1.3~~ 1.1×10^9 clock/s)

[*"Frequency listed is nominal (non-AVX) TDP frequency. For all-tile turbo frequency, add 100 MHz. For single-tile turbo frequency, add 200 MHz. For high-AVX instruction frequency, subtract 200 MHz."
Footnote 3, p. 4 Product Brief: Intel Xeon Phi Processor. PDF online.]

Assembly for compute-only code: 10 FMA's in loop body

ps ≡ packed single precision

zmm ≡ 512-bit register

*fma = fma * b + c*

```
..B1.260: # Preds ..B1.260 ..B1.259
# Execution count [1.00e+06]
addl $1,    %eax                          #251.3 c1
vfmadd213ps %zmm0, %zmm12, %zmm11         #256.15 c1
vfmadd213ps %zmm0, %zmm12, %zmm10         #257.15 c1
vfmadd213ps %zmm0, %zmm12, %zmm9          #258.15 c7 stall 2
vfmadd213ps %zmm0, %zmm12, %zmm8          #259.15 c7
vfmadd213ps %zmm0, %zmm12, %zmm7          #260.15 c13 stall 2
vfmadd213ps %zmm0, %zmm12, %zmm6          #261.15 c13
vfmadd213ps %zmm0, %zmm12, %zmm5          #262.15 c19 stall 2
vfmadd213ps %zmm0, %zmm12, %zmm4          #263.15 c19
vfmadd213ps %zmm0, %zmm12, %zmm3          #264.15 c25 stall 2
vfmadd213ps %zmm0, %zmm12, %zmm2          #265.16 c25
cmpl $50000000, %eax                      #251.3 c25
jb ..B1.260   # Prob 99%                  #251.3 c27
```

= *c* + *b* * *fma*

VITO ≡ vectorize inner, thread outer

vfmadd213ps: latency 6 throughput 0.5

Code for compute only ("speed of light")

Use intrinsics for definiteness and speed: $fma = fma * b + c$

Example "skeleton" code: 10 vector FMA instructions in main iteration loop

```
1  #include <immintrin.h>  //needed for AVX512 intrinsics
2
3  #define NUM_ITERATIONS  50000000   //50M
4  #define VEC_LENGTH  16  // = 16 for single precision
5  #define NUM_FMA_INSTRUCTIONS 10  //number of fma instructions in main iteration loop
6  . . .
7  for (int nThreads = 1; nThreads <= 256; nThreads += 1)
8      {
9          omp_set_num_threads(nThreads);
10
11         for (int trial = 0; trial < 5; j++)
12             {
13                 register __m512 fma1_avx;
14                 . . .
15                 register __m512 fma10_avx;
16
17                 register __m512 b_avx;
18                 register __m512 c_avx;
19
20                 register __m512 result_avx;
21
```

Initialization details

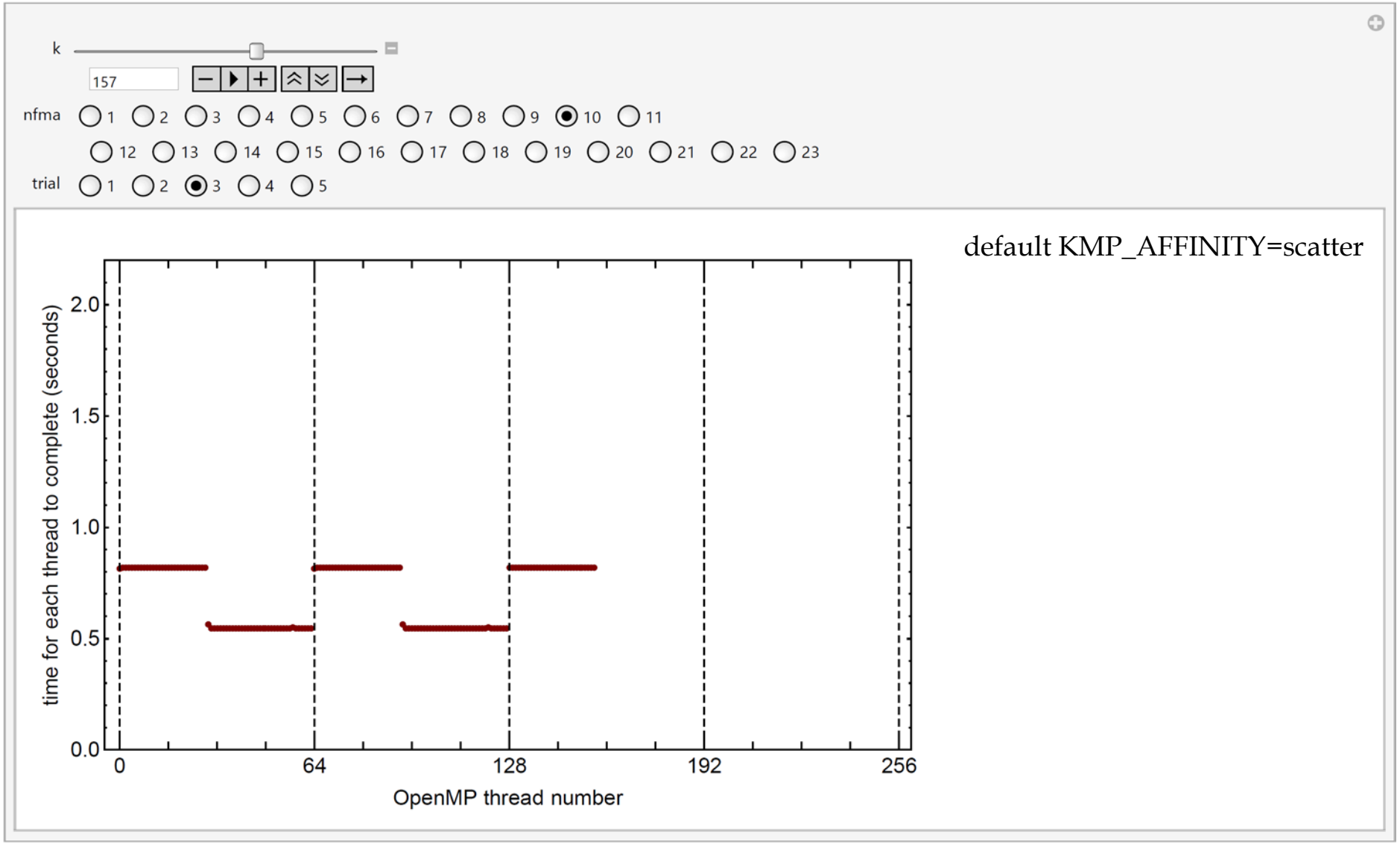
```
22          const float fma_init[VEC_LENGTH] \
23          __attribute__((aligned(64))) = \
24          { 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
25            1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f };
26
27          const float b_init[VEC_LENGTH] \
28          __attribute__((aligned(64))) = \
29          { 0.51f, 0.52f, 0.53f, 0.54f, 0.55f, 0.56f, 0.57f, 0.58f,
30            0.59f, 0.60f, 0.61f, 0.62f, 0.63f, 0.64f, 0.65f, 0.66f };
31
32          const float c_init[VEC_LENGTH] \
33          __attribute__((aligned(64))) = \
34        { 0.49f, 0.48f, 0.47f, 0.46f, 0.45f, 0.44f, 0.43f, 0.42f,
35          0.41f, 0.40f, 0.39f, 0.38f, 0.37f, 0.36f, 0.35f, 0.34f };
36
37        float result1[VEC_LENGTH] __attribute__((aligned(64))) = \
38        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
39          0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 };
40        . . .
41        float result10[VEC_LENGTH] __attribute__((aligned(64))) = \
42        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
43          0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 };
44
45        fma1_avx = _mm512_load_ps(&fma_init[0]);
46        . . .
47        fma10_avx = _mm512_load_ps(&fma_init[0]);
48
49        b_avx = _mm512_load_ps(&b_init[0]);
50        c_avx = _mm512_load_ps(&c_init[0]);
51
52
```

Main timed loop

```
53                  double begin_priv = omp_get_wtime();  //begin timing
54
55  //***Beginning of main iterated-fma loop executed by each thread in the team****
56
57              for (long index = 0; index < NUM_ITERATIONS; index++)
58                  {
59                  fma1_avx = _mm512_fmadd_ps(fma1_avx, b_avx, c_avx);
60                  . . .
61                  fma10_avx = _mm512_fmadd_ps(fma10_avx, b_avx, c_avx);
62                  }
63
64  //*************************End of main iterated-fma loop***********************
65
66              double end_priv = omp_get_wtime();  //end timing
67
68              _mm512_store_ps(&result1[0], fma1_avx);
69              . . .
70              _mm512_store_ps(&result10[0], fma10_avx);
71  }
72  }
```

VITO ≡ vectorize inner, thread outer

Animation



Out[19]=

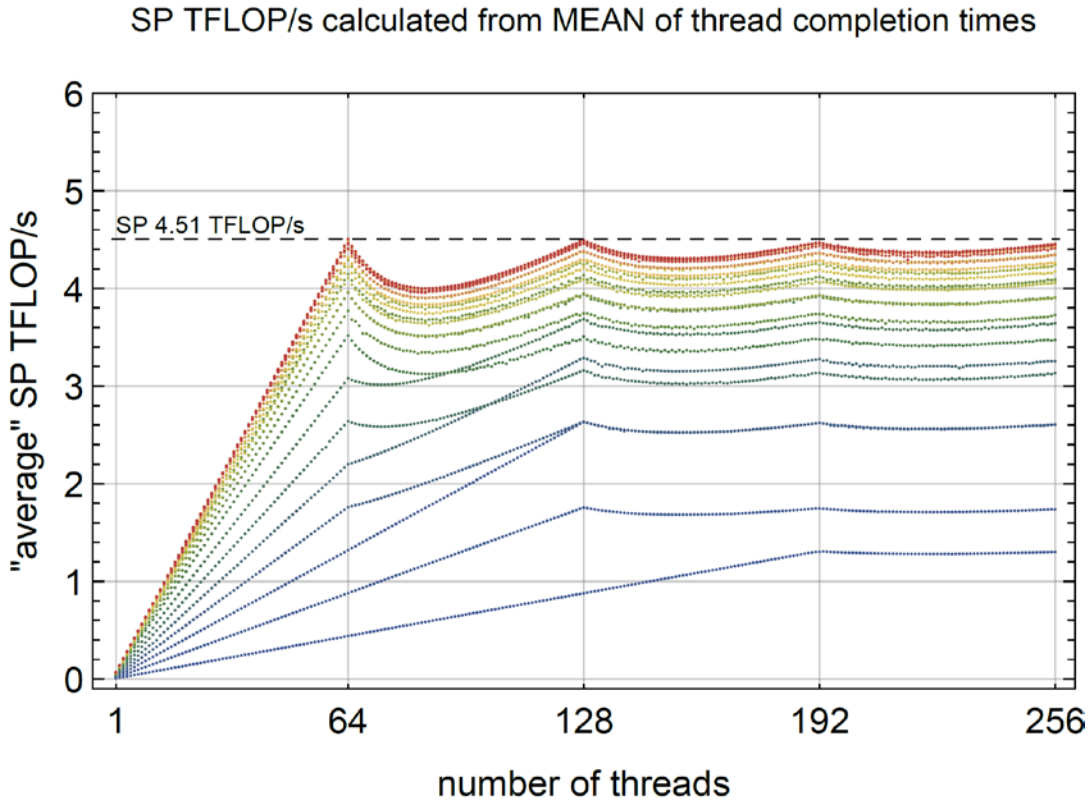
For 1, 2, 3, . . . , 256 OpenMP threads, have each thread compute 50 million iterations of a loop body containing *nfma* = 10 single-precision FMA instructions.

How much time does it take each individual OpenMP thread to complete its loop?

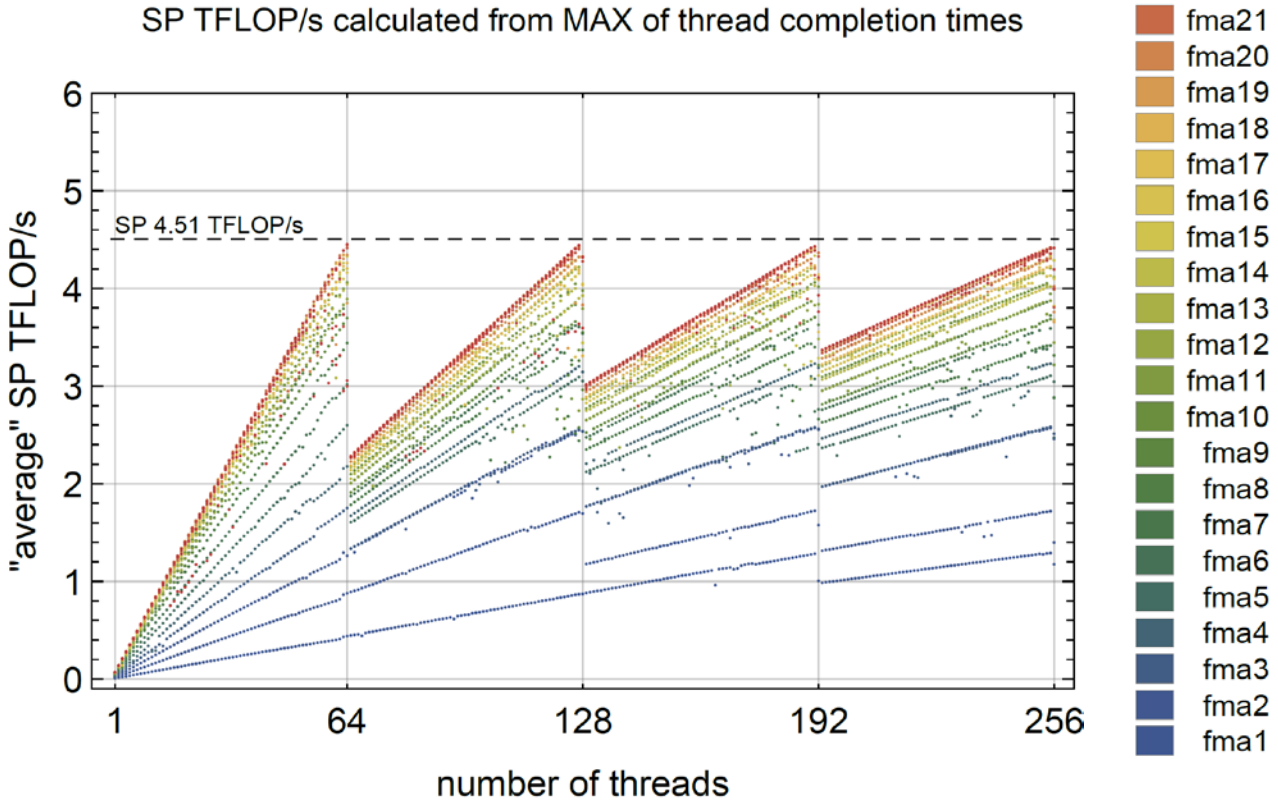
Single precision (SP) results (Xeon Phi 7210, 64 cores)

default KMP_AFFINITY=scatter

"reasonable, fair" ✓



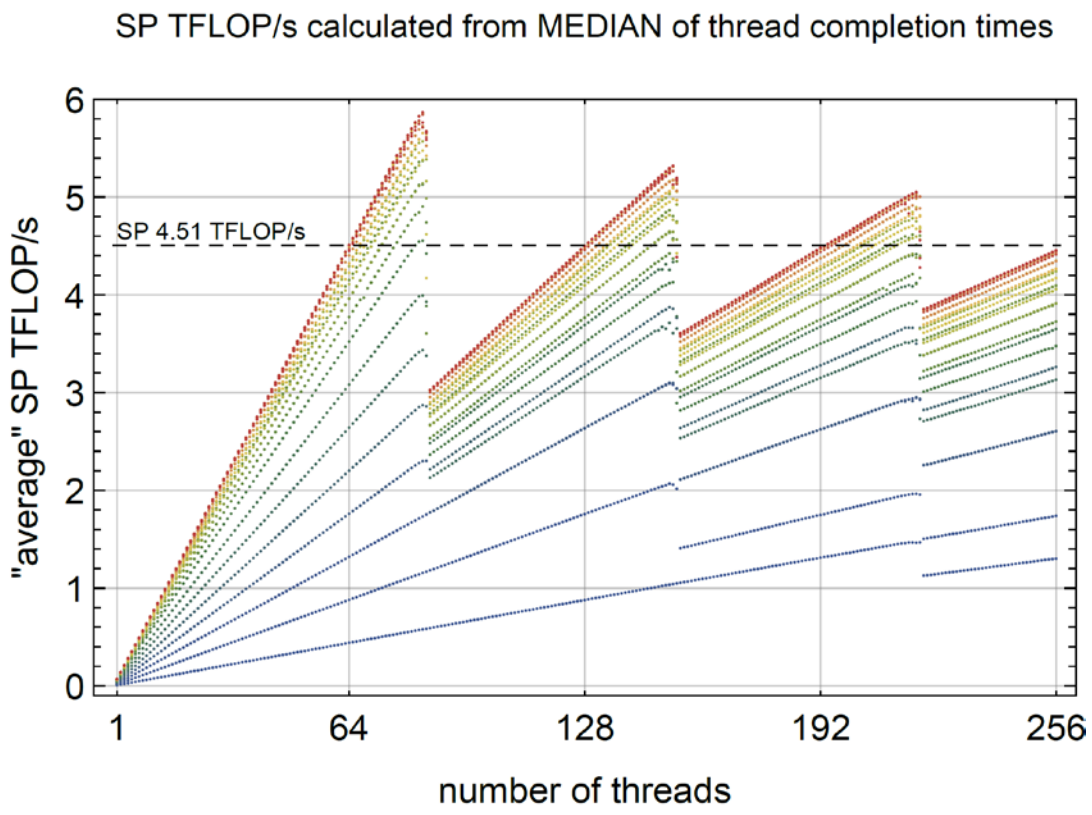
"conservative, pessimistic" ✓



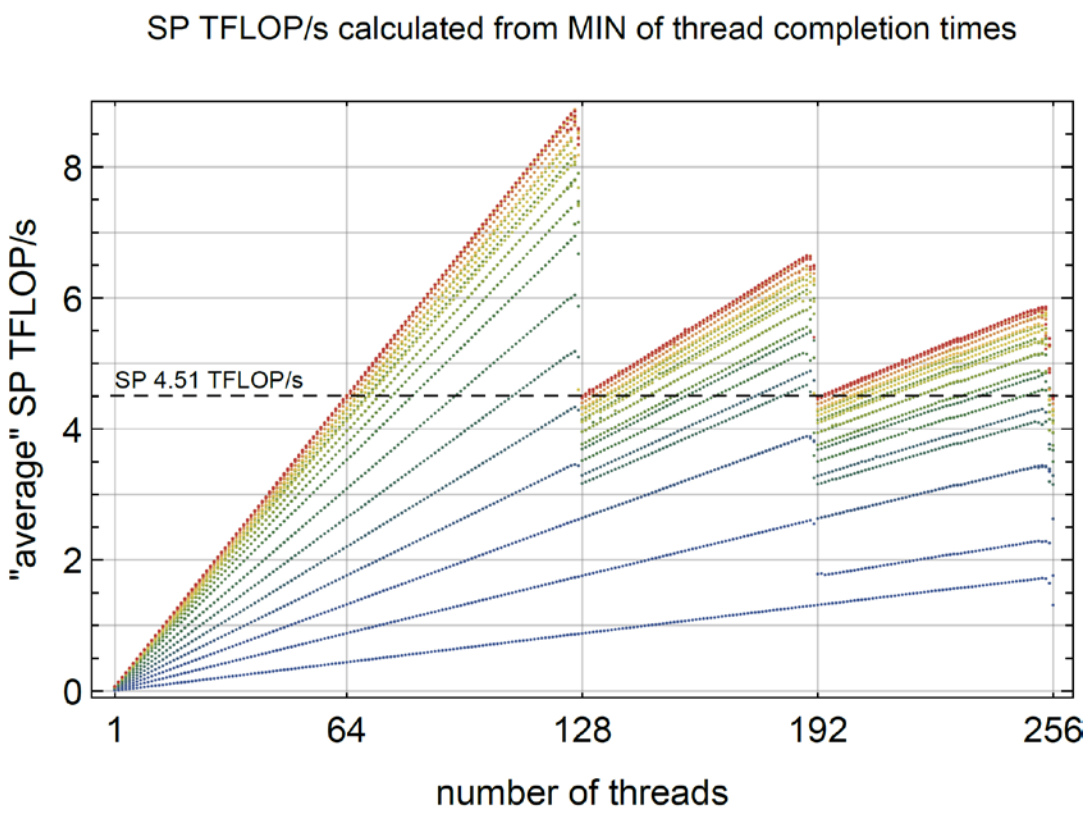
Single precision (SP) results (Xeon Phi 7210, 64 cores)

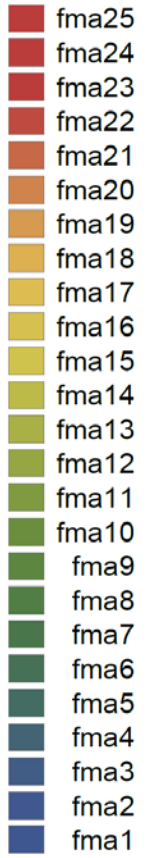
default KMP_AFFINITY=scatter

"too optimistic" ✘



"much too optimistic" ✘

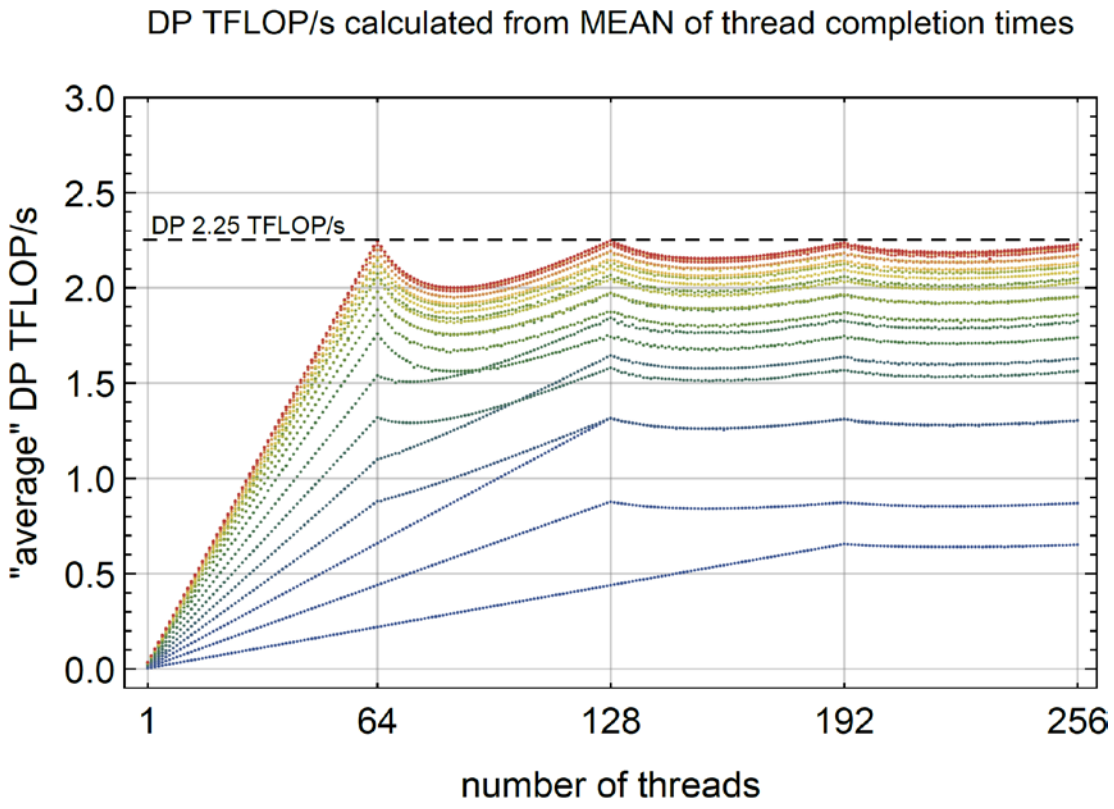




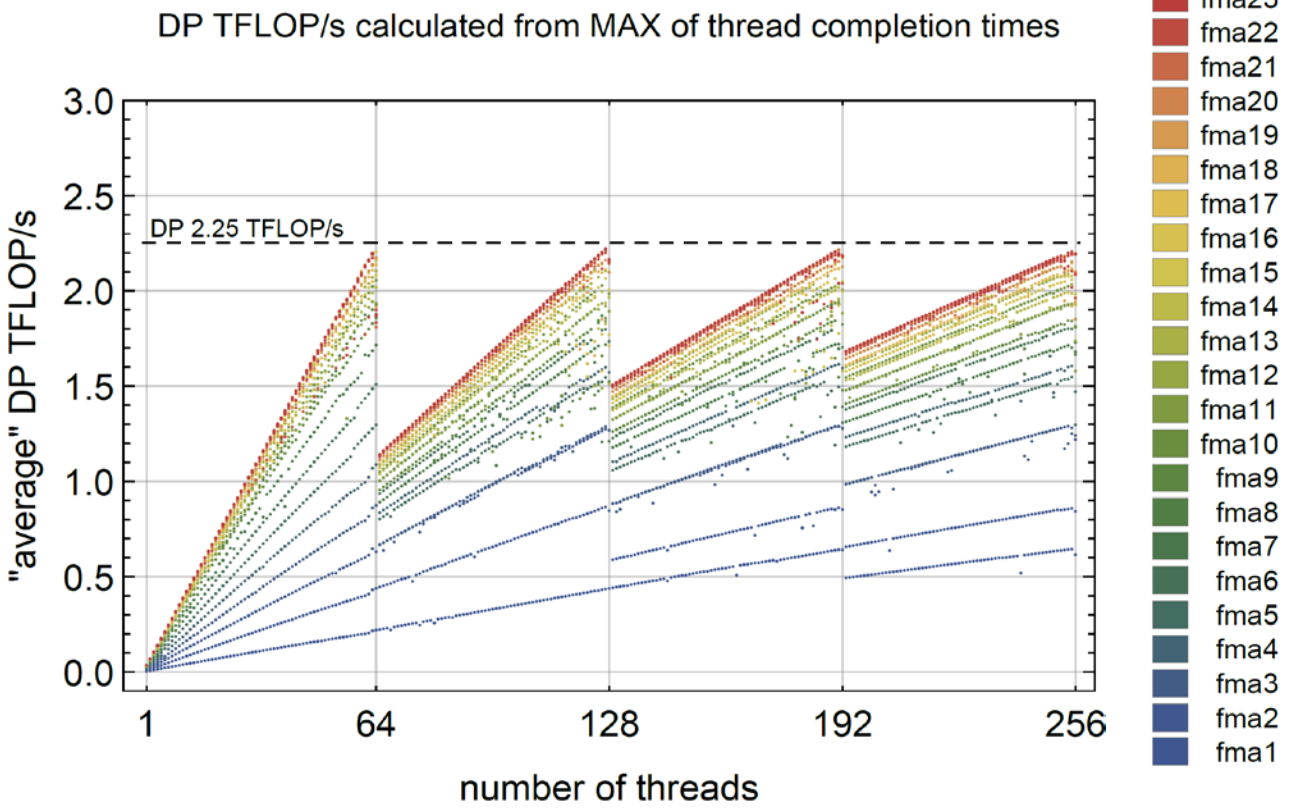
Double precision (DP) results (Xeon Phi 7210, 64 cores)

default KMP_AFFINITY=scatter

"reasonable, fair" ✓



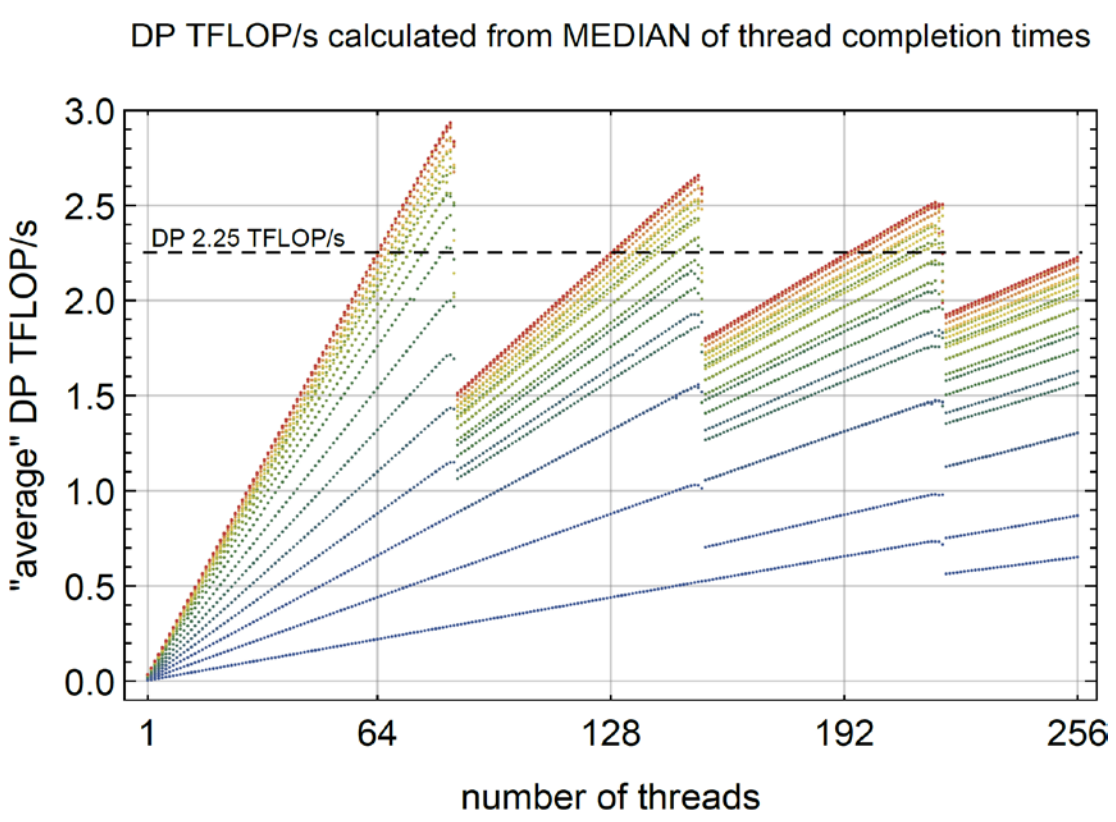
"conservative, pessimistic" ✓



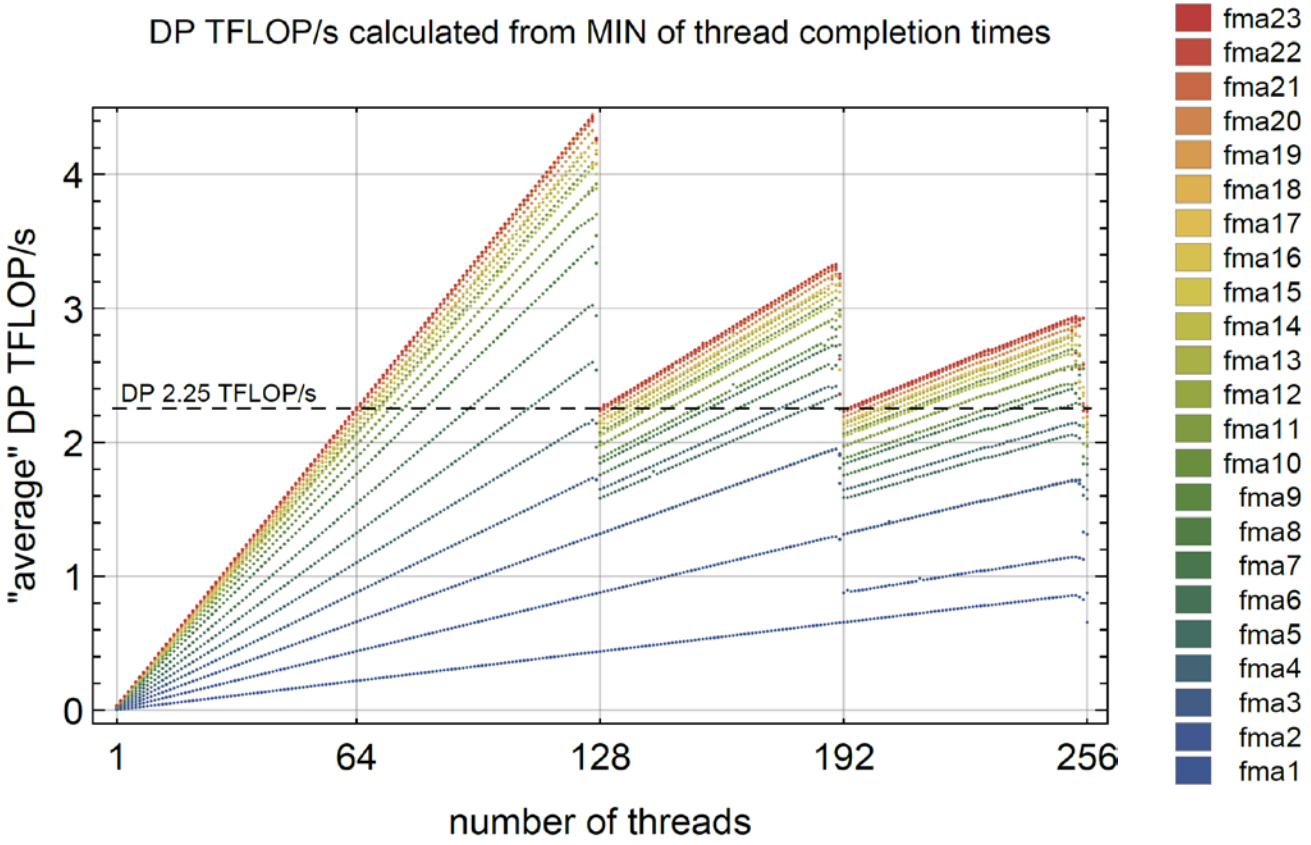
Double precision (DP) results (Xeon Phi 7210, 64 cores)

default KMP_AFFINITY=scatter

"too optimistic" ✘



"much too optimistic" ✘



Lesson learned

Beyond multithreading and vectorization, must optimize instruction-level parallelism (ILP). (same story on GPUs)

"There should always be FMA instructions ready to execute in the VFU [VPU] . . . ."

Intel Xeon Phi Processor High Performance Programming (Knight's Landing Edition), James Jeffers, James Reinders, and Avinash Sodani, (Morgan Kaufmann, 2016) p. 119.

Summary

- "Small" experiments can have "big" data analysis problems
- Code modernization is for everyone
- "Trickle down" from supercomputing to everyday computing
- Good to develop definite, but reasonable, expectations for code modernization and code optimization

Acknowledgments

Asa Phillips ’17
Jovita Ho ’16
Shujaat Khan ’10
Jonathan Spencer ’10
Max Junda ’09
Prof. Noah Graham
Cris Butler
Lance Ritchie
Jonathan Kemp

Special thanks to
Colfax Research:

Andrey Vladimirov
Gautam Shah

Who needs to learn about modern code?

Near Disaster Threatens Schools If Levy Fails

WORKING ON MATH honors projects involving use of a computer are two Roosevelt High students Jeff Dunham (foreground) son of Mr. and Mrs. Deane Dunham, and Stuart Liddle, son of Mr. and Mrs. W. M. Liddle. Dunham is sorting computer cards indicating the positions of the planets so that he can figure any planet's position in relation to any other planet at any given date. This honors program, one of several at Roosevelt, would, along with several others for gifted students, be seriously cut back or eliminated in the event of failure of the May 18 school levy.

UNIVERSITY DISTRICT
HERALD
SEATTLE'S 1st COMMUNITY NEWSPAPER
Vol 54, No. 50 4719 Brooklyn Ave. N.E. Seattle, Washington 98105 525-3700 41 10¢ per Copy Wednesday, May 5, 1971

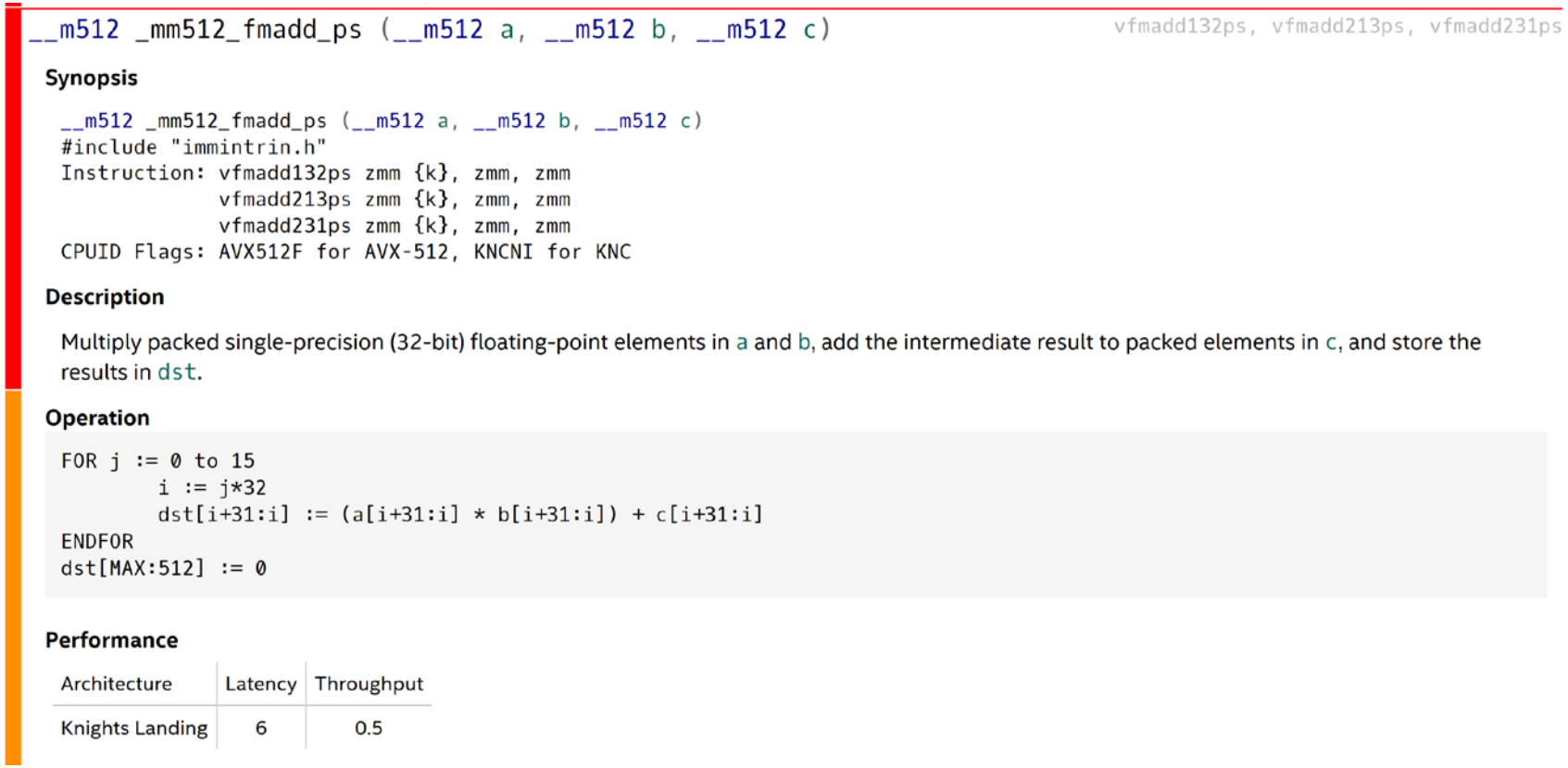
Me!

You?

Attend Colfax HOW series course.

Intrinsics reference

https://software.intel.com/sites/landingpage/IntrinsicsGuide/#



__m512 _mm512_fmadd_ps (__m512 a, __m512 b, __m512 c) vfmadd132ps, vfmadd213ps, vfmadd231ps

Synopsis

```
__m512 _mm512_fmadd_ps (__m512 a, __m512 b, __m512 c)
#include "immintrin.h"
Instruction: vfmadd132ps zmm {k}, zmm, zmm
             vfmadd213ps zmm {k}, zmm, zmm
             vfmadd231ps zmm {k}, zmm, zmm
CPUID Flags: AVX512F for AVX-512, KNCNI for KNC
```

Description

Multiply packed single-precision (32-bit) floating-point elements in a and b, add the intermediate result to packed elements in c, and store the results in dst.

Operation

```
FOR j := 0 to 15
        i := j*32
        dst[i+31:i] := (a[i+31:i] * b[i+31:i]) + c[i+31:i]
ENDFOR
dst[MAX:512] := 0
```

Performance

Architecture	Latency	Throughput
Knights Landing	6	0.5