Vector Parallelism on Multi-Core Processors

Steve Lantz, Cornell University

CoDaS-HEP Summer School, July 25, 2019

Cornell University
Center for Advanced Computing

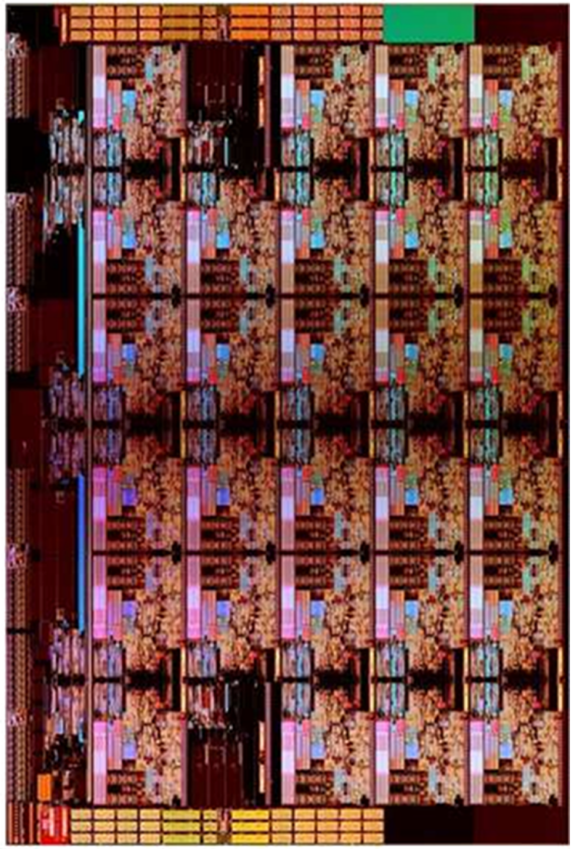
PART I:

Vectorization Basics

(author: Steve Lantz)

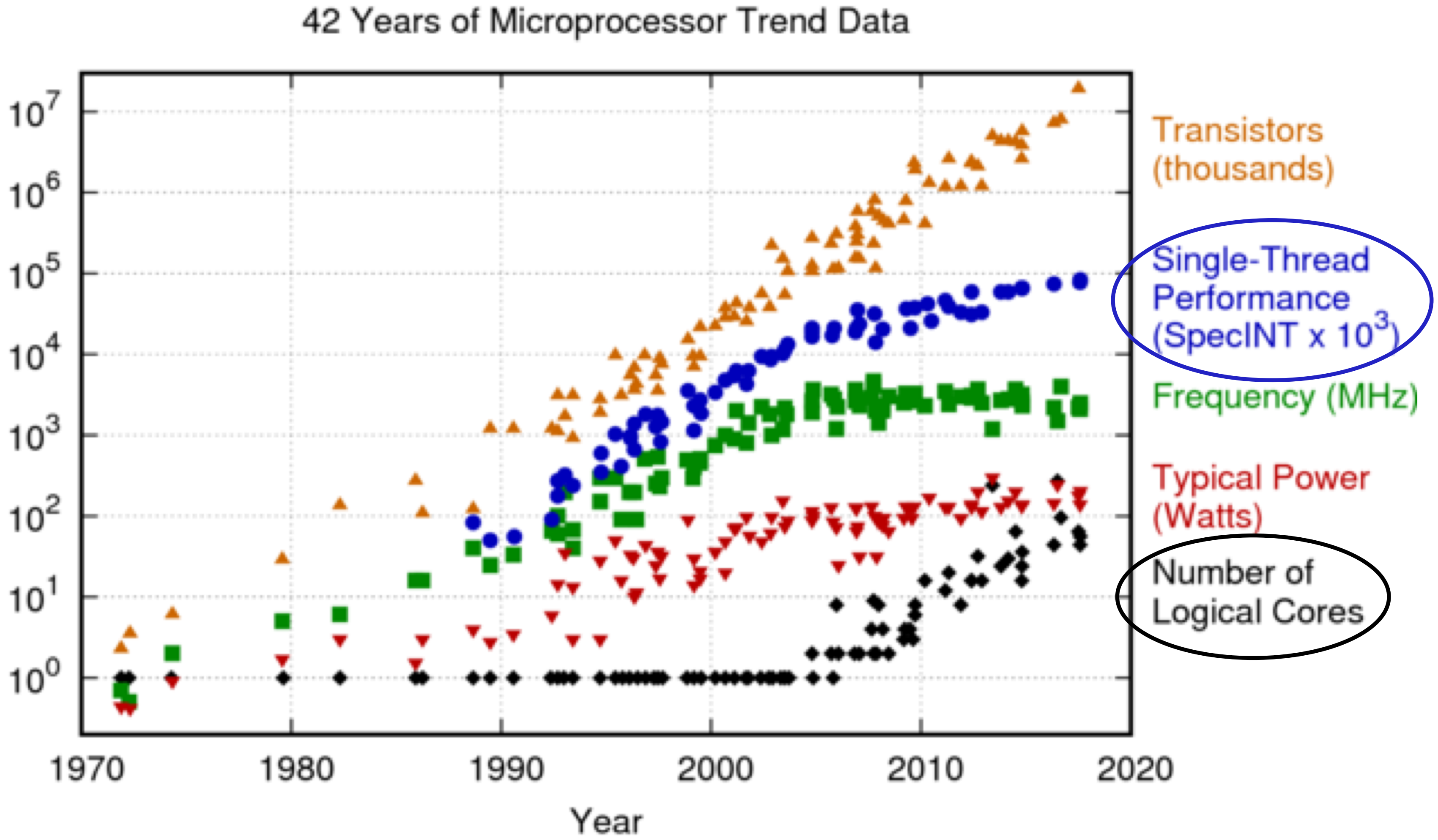
Vector Parallelism: Motivation

- CPUs are no faster than they were a decade ago
  - Power limits! “Slow” transistors are more efficient, cooler
- Yet process improvements have made CPUs denser
  - Moore’s Law! Add 2x more “stuff” every 18–24 months
- One way to use extra transistors: **more cores**
  - Dual-core Intel chips arrived in 2005; counts keep growing
  - 6–28 in Skylake-SP (doubled in 2-chip Cascade Lake module)
- Another solution: **SIMD or vector operations**
  - First appeared on Pentium with MMX in 1996
  - Vectors have ballooned: 512 bits (16 floats) in Intel Xeon
  - Can *vectorization* increase speed by an order of magnitude?



Die shot of 28-core Skylake-SP
Source: wikichip.org

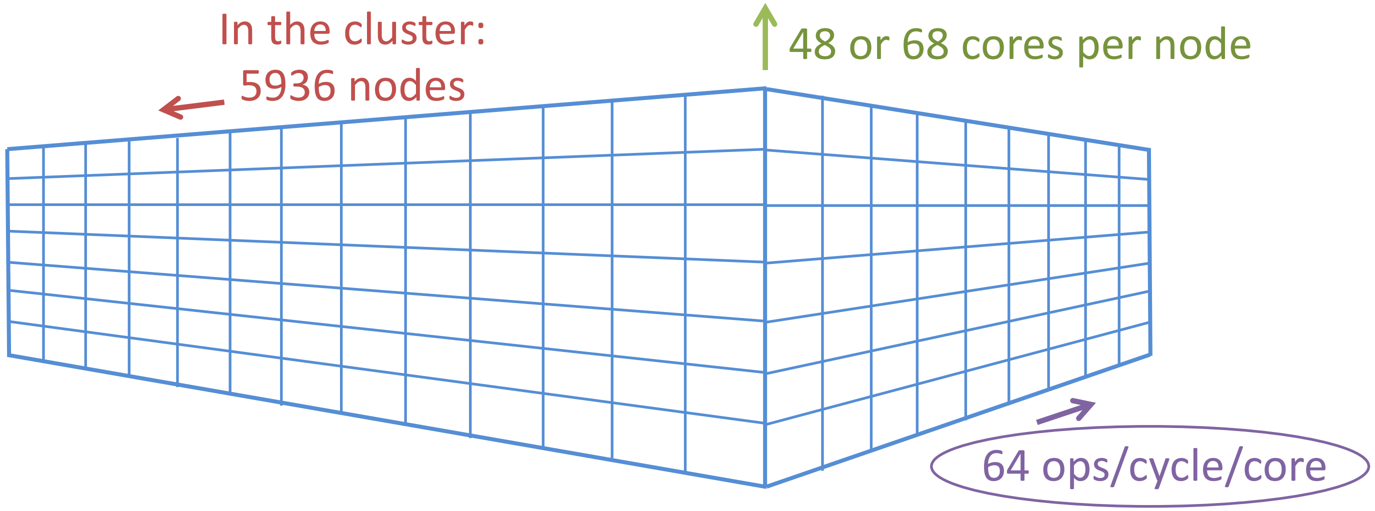
What Moore's Law Buys Us, These Days…



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

A Third Dimension of Scaling

- Along with scaling *out* and *up*, you can *"scale deep"*
  - Arguably, vectorization can be as important as multithreading
- Example: Intel processors in TACC Stampede2 cluster



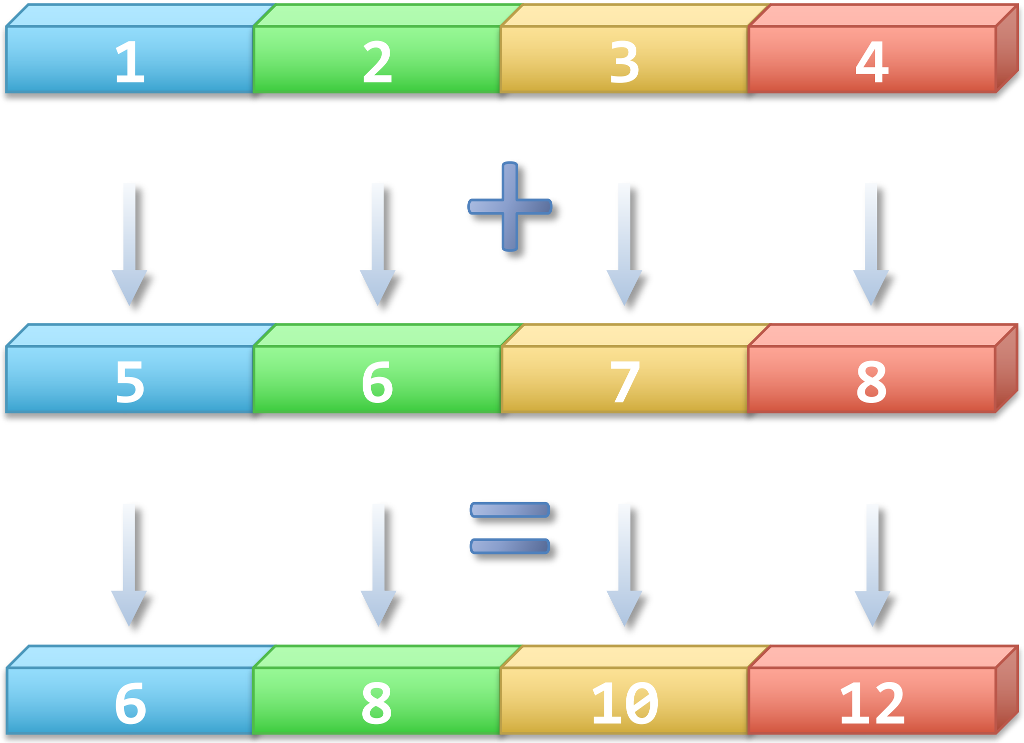
In the cluster:
5936 nodes

48 or 68 cores per node

64 ops/cycle/core

  - 1736 Skylake-SP + 4200 Xeon Phi (KNL) nodes; 48 or 68 cores each
  - Each core can do up to **64 operations/cycle** on vectors of 16 floats

How It Works, Conceptually



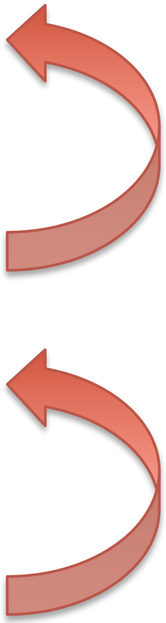
SIMD: Single Instruction, Multiple Data

Three Ways to Look at Vectorization

1. **<u>Hardware Perspective</u>:** Run vector instructions involving special registers and functional units that allow in-core parallelism for operations on arrays (vectors) of data.

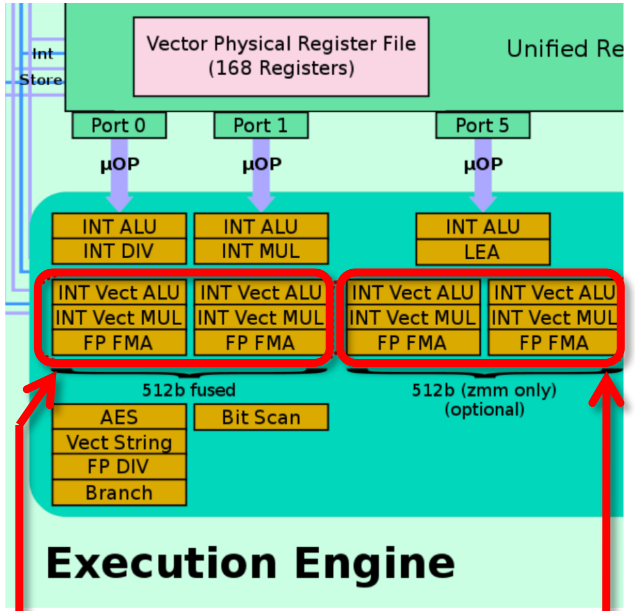
2. **<u>Compiler Perspective</u>:** Determine how and when it is possible to express computations in terms of vector instructions.

3. **<u>User Perspective</u>:** Determine how to write code with SIMD in mind; e.g., in a way that allows the compiler to deduce that vectorization is possible.



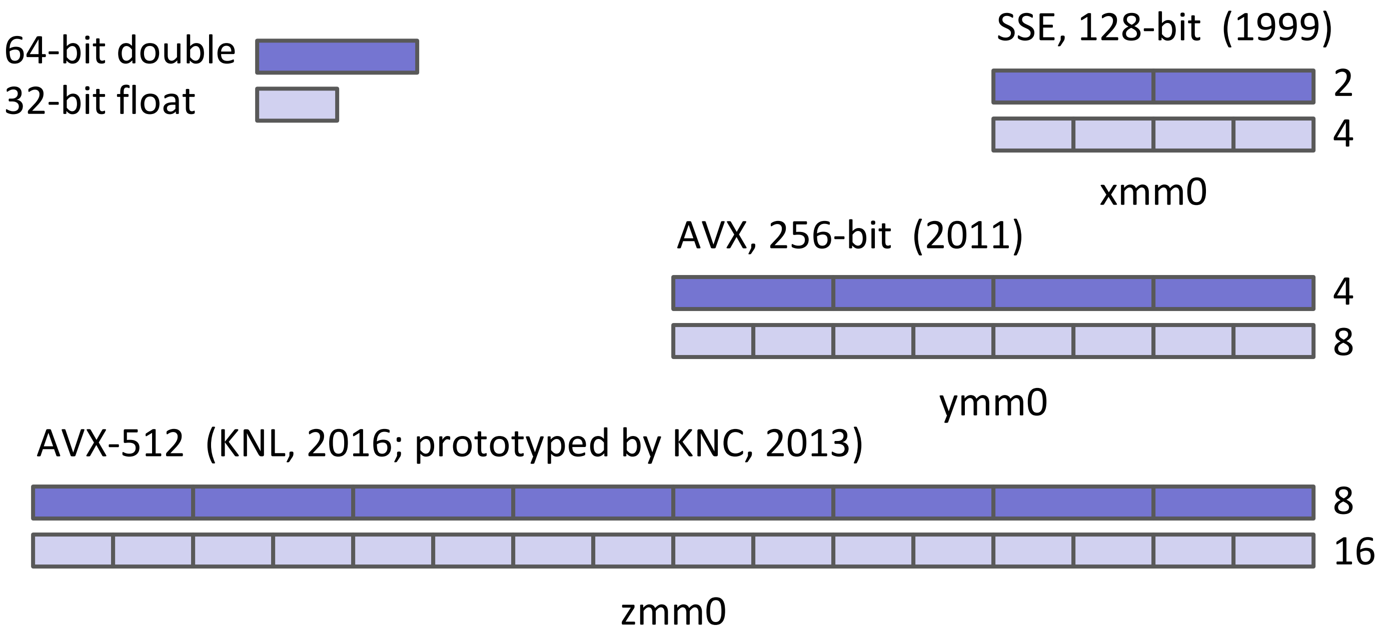
Hardware Perspective

- SIMD = Single Instruction, Multiple Data
  - Part of commodity CPUs (x86, x64, PowerPC) since late ’90s
- Goal: parallelize computations on vector arrays
  - Line up operands, execute one op on all simultaneously
- SIMD instructions have gotten quicker over time
  - Initially, several cycles for execution on small vectors
  - Intel AVX introduced pipelining of some SIMD instructions
  - Now: multiply-and-add large vectors on every cycle
- Intel’s latest: Knights Landing, Skylake-SP, Cascade Lake
  - 2 VPUs (vector processing units) per core, in most models
  - 2 ops/VPU if they do FMAs (Fused Multiply-Add) every cycle



Partial block diagram of SKL-SP core
Source: wikichip.org

Evolution of Vector Registers, Instructions



- A core has 16 (SSE, AVX) or 32 (AVX-512) vector registers
- In each cycle, VPUs can access registers, do FMAs (e.g.)

Peak Flop/s, and Why It's (Almost) a Fiction

- Peak flop/s (FLoating-point OPs per second) is amplified by vector FMAs
- Example: Intel Xeon Gold 6130 "Skylake-SP" @ 2.1 GHz
  - (2 x 16 flop/VPU) x (2 VPUs/core) x (16 cores) x 2.1 GHz = 2150 Gflop/s (?)
- *Dubious assumption #1:* data are loaded and stored with no delay
  - Implies heavy reuse of data in vector registers, perfect prefetching into L1 cache
- *Dubious assumption #2:* code is perfectly vectorized
  - Otherwise the scalar fraction of work *S* limits the first factor to 1/*S* (Amdahl's Law)
- *Dubious assumption #3:* no slow operations like division, square root
- *Dubious assumption #4:* clock rate is fixed
  - If all cores are active, clock is actually throttled to 1.9 GHz to prevent overheating

Instructions Do More Than Just Flops…

- *Data Access:* Load/Store, Pack/Unpack, Gather/Scatter
- *Data Prefetch:* Fetch, but don’t load into a register
- *Vector Rearrangement:* Shuffle, Bcast, Shift, Convert
- *Vector Initialization:* Random, Set
- *Logic:* Compare, AND, OR, etc.
- *Math:* Arithmetic, Trigonometry, Cryptography, etc.
- *Variants of the Above…* Mask, Swizzle, Implicit Load…
  - Combine an operation with data selection or movement
- This is why AVX-512 comprises over 4000 instructions

Extension	CSL	SKL	KNL
AVX512F *Foundation*	X	X	X
AVX512CD *Conflict Det.*	X	X	X
AVX512BW *Byte & Word*	X	X	
AVX512DQ *Dble. & Quad.*	X	X	
AVX512VL *Vector Length*	X	X	
AVX512PF *Prefetch*			X
AVX512ER *Exp. & Recip.*			X
AVX512VNNI *Neural Net.*	X		

How Do You Get Vector Speedup?

- Program the key routines in assembly?
  - Ultimate performance potential, but only for the brave
- Program the key routines using intrinsics?
  - Step up from assembly; useful in spots, but risky

✓ Link to an optimized library that does the heavy lifting
  - Intel MKL, e.g., written by people who know all the tricks
  - BLAS is the portable interface for doing fast linear algebra

✓ Let the compiler figure it out
  - Relatively “easy” for user, “challenging” for compiler
  - Compiler may need some guidance through directives
  - Programmer can help by using simple loops and arrays

Compiler Perspective

- Think of vectorization in terms of loop unrolling
  - Unroll by 4 iterations, if 4 elements fit into a vector register

```
for (i=0; i<N; i++) {
    c[i]=a[i]+b[i];
}
```

```
for (i=0; i<N; i+=4) {
    c[i+0]=a[i+0]+b[i+0];
    c[i+1]=a[i+1]+b[i+1];
    c[i+2]=a[i+2]+b[i+2];
    c[i+3]=a[i+3]+b[i+3];
}
```

```
Load a(i..i+3)
Load b(i..i+3)
Do 4-wide a+b->c
Store c(i..i+3)
```

Loops That the Compiler Can Vectorize

Basic requirements of vectorizable loops:

- Number of iterations is known on entry
  - No conditional termination (“break” statements, while-loops)
- Single control flow; no “if” or “switch” statements
  - Note, the compiler may convert “if” to a masked assignment!
- Must be the innermost loop, if nested
  - Note, the compiler may reorder loops as an optimization!
- No function calls but basic math: pow(), sqrt(), sin(), etc.
  - Note, the compiler may inline functions as an optimization!
- All loop iterations must be independent of each other

Compiler Options and Optimization

- Intel compilers start vectorizing at optimization level `-O2`
  - Default is SSE instructions, 128-bit vector width
  - To tune vectors to the host machine: `-xHost`
  - To optimize across objects (e.g., to inline functions): `-ipo`
  - To disable vectorization: `-no-vec`
- GCC compilers start vectorizing at optimization level `-O3`
  - Default for x86_64 is SSE (see output from gcc -v, no other flags)
  - To tune vectors to the host machine: `-march=native`
  - To optimize across objects (e.g., to inline): `-fwhole-program`
  - To disable vectorization: `-fno-tree-vectorize` (after `-O3`)
- Why disable or downsize vectors? To gauge their benefit!

Machine-Specific Compiler Options

- Intel compilers
  - Use `-xCORE-AVX2` to compile for AVX2, 256-bit vector width
  - Use `-xCOMMON-AVX512` to compile with AVX-512F + AVX-512CD
  - For SKL-SP: `-xCORE-AVX512 -qopt-zmm-usage=high`
  - For KNL (MIC architecture): `-xMIC-AVX512`
- GCC compilers
  - Use `-mavx2` to compile for AVX2
  - GCC 4.9+ has separate options for most AVX-512 extensions
  - GCC 5.3+ has `-march=skylake-avx512`
  - GCC 6.1+ has `-march=knl`
  - GCC 9.1+ has `-march=cascadelake`

Exercise 1

- Copy the code at right and paste it into a local file named abc.c
- We will see how different compiler flags affect the vectorization of simple loops

```
#include <stdio.h>
#define ARRAY_SIZE 1024
#define NUMBER_OF_TRIALS 1000000

void main(int argc, char *argv[]) {

    /* Declare arrays small enough to stay in L1 cache.
       Assume the compiler aligns them correctly. */
    double a[ARRAY_SIZE], b[ARRAY_SIZE], c[ARRAY_SIZE];
    int i, t;
    double m = 1.5, d;

    /* Initialize a, b and c arrays */
    for (i=0; i < ARRAY_SIZE; i++) {
        a[i] = 0.0; b[i] = i*1.0e-9; c[i] = i*0.5e-9;
    }
    /* Perform operations with arrays many, many times */
    for (t=0; t < NUMBER_OF_TRIALS; t++) {
        for (i=0; i < ARRAY_SIZE; i++) {
            a[i] += m*b[i] + c[i];
        }
    }
    /* Print a result so the loops won't be optimized away */
    for (i=0; i < ARRAY_SIZE; i++) d += a[i];
    printf("%f\n",d/ARRAY_SIZE);
}
```

Exercise 1 (cont'd.)

1. Invoke your compiler with no special flags and time a run:

```
gcc-9 abc.c -o abc
/usr/bin/time ./abc
```

2. Repeat this process for the following sets of options:

```
gcc-9 -O2 abc.c -o abc
gcc-9 -O3 -fno-tree-vectorize abc.c -o abc
gcc-9 -O3 abc.c -o abc
gcc-9 -O3 -msse3 abc.c -o abc
gcc-9 -O3 -march=native abc.c -o abc
gcc-9 -O3 -march=??? abc.c -o abc  #take a guess
```

(Refer to a previous slide if you have the Intel compiler)

Exercise 1 (still cont'd.)

3. Your best result should be from `-march=native` . Why?
   - You can try `-mavx2` on a laptop, but maybe it isn't as good
   - Here is the current list of architectures that gcc knows about
4. Other things to note:
   - Optimization `-O3` is degraded by `-fno-tree-vectorize`
   - Not specifying an architecture at `-O3` is equivalent to `-msse3`
5. Do you get the expected speedup factors?
   - SSE registers hold 2 doubles; AVX registers hold 4 doubles
   - Recent laptops should be able to do AVX (but not AVX-512)

Why Not Use an Optimized Library?

- Optimized libraries like OpenBLAS may not have the exact function you need
- The kernel of abc.c looks like a DAXPY, or double-precision (aX + Y)… but it isn’t quite…
- It turns out the inner loop must be replaced by two DAXPY calls, not one, and the resulting code runs several times slower

```
for (t=0; t < NUMBER_OF_TRIALS; t++) {
    for (i=0; i < ARRAY_SIZE; i++) {
        a[i] += m*b[i] + c[i];
    }
}
```

```
for (t=0; t < NUMBER_OF_TRIALS; t++) {
    cblas_daxpy(ARRAY_SIZE, m, b, 1, a, 1);
    cblas_daxpy(ARRAY_SIZE, 1.0, c, 1, a, 1);
}
```

Optimization Reports

Use optimization report options for info on vectorization:

```
icc -c -O3 -qopt-report=2 -qopt-report-phase=vec myvec.c
```

The `=n` controls the amount of detail in `myvec.optrpt`

n	Description of information presented
0	No vector report
1	Lists the loops that were vectorized
2	(default level) Adds the loops that were not vectorized, plus a short reason
3	Adds summary information from the vectorizer about all loops
4	Adds verbose information from the vectorizer about all loops
5	Adds details about any data dependencies encountered (proven or assumed)

“Level 2” for GCC: `-fopt-info-vec` and `-fopt-info-vec-missed`

Exercise 2

Let’s examine optimization reports for the abc.c code.

1. Recompile the code with `-O3`, along with optimization reporting (`-fopt-info-vec`) from the vectorizer.
   - Confirm that the inner loops were vectorized as expected.
2. Repeat (1), but this time with vectorization turned off (i.e., `-fno-tree-vectorize`) . Do you get any output?
3. Repeat (1), but now add `-fopt-info-vec-missed` (loops that missed out on vectorization) to see what else the compiler tried to do with this code.
   - Considering that the main loops ultimately vectorized, you may find that gcc gives way too much information here.

User Perspective

- User’s goal is to supply code that runs well on hardware
- Thus, you need to know the hardware perspective
  - Think about how instructions will run on vector hardware
  - Try also to combine additions with multiplications
  - Furthermore, try to reuse everything you bring into cache!
- And you need to know the compiler perspective
  - Look at the code like the compiler looks at it
  - At a minimum, set the right compiler options!

Vector-Aware Coding

- Know what makes codes vectorizable at all
  - The “for” loops (C) or “do” loops (Fortran) that meet constraints
- Know where vectorization ought to occur
- Arrange vector-friendly data access patterns (unit stride)
- Study compiler reports: do loops vectorize as expected?
- Evaluate execution performance: is it near the roofline?
- Implement fixes: directives, compiler flags, code changes
  - Remove constructs that hinder vectorization
  - Encourage/force vectorization when compiler fails to do it
  - Engineer better memory access patterns

Challenge: Loop Dependencies

- Vectorization changes the order of computation compared to sequential case
  - Groups of computations now happen simultaneously
- Compiler must be able to prove that vectorization will produce correct results
- Key criterion: “unrolled” loop iterations must be independent of each other
  - Wider vectors means that more iterations must be independent
  - Note, not all kinds of dependencies are detrimental
- Compiler performs dependency analysis and vectorizes accordingly
  - It will make conservative assumptions about dependencies, unless guided by directives

Loop Dependencies: Read After Write

Consider adding the following vectors in a loop, N=5:

a = {0,1,2,3,4}

b = {5,6,7,8,9}

```
for(i=1; i<N; i++)
  a[i] = a[i-1] + b[i];
```

Applying each operation sequentially:

a[1] = a[0] + b[1] → a[1] = 0 + 6 → a[1] = 6
a[2] = a[1] + b[2] → a[2] = 6 + 7 → a[2] = 13
a[3] = a[2] + b[3] → a[3] = 13 + 8 → a[3] = 21
a[4] = a[3] + b[4] → a[4] = 21 + 9 → a[4] = 30

a = {0, 6, 13, 21, 30}

Loop Dependencies: Read After Write

Consider adding the following vectors in a loop, N=5:

a = {0,1,2,3,4}

b = {5,6,7,8,9}

```
for(i=1; i<N; i++)
  a[i] = a[i-1] + b[i];
```

Applying each operation sequentially:

a[1] = a[0] + b[1] → a[1] = 0 + 6 → a[1] = 6

a[2] = a[1] + b[2] → a[2] = 6 + 7 → a[2] = 13

a[3] = a[2] + b[3] → a[3] = 13 + 8 → a[3] = 21

a[4] = a[3] + b[4] → a[4] = 21 + 9 → a[4] = 30

a = {0, 6, 13, 21, 30}

Loop Dependencies: Read After Write

Now let's try vector operations:

a = {0,1,2,3,4}

b = {5,6,7,8,9}

```
for(i=1; i<N; i++)
  a[i] = a[i-1] + b[i];
```

Applying vector operations, i={1,2,3,4}:

a[i-1] = {0,1,2,3} (load)

b[i] = {6,7,8,9} (load)

{0,1,2,3} + {6,7,8,9} = {6, 8, 10, 12} (operate)

a[i] = {6, 8, 10, 12} (store)

a = {0, 6, 8, 10, 12} ≠ {0, 6, 13, 21, 30} NOT VECTORIZABLE

Loop Dependencies: Synopsis

- Read After Write
  - Also called “flow” dependency
  - Variable written first, then read
  - Not vectorizable

```
for(i=1; i<N; i++)
  a[i] = a[i-1] + b[i];
```

- Write After Read
  - Also called “anti” dependency
  - Variable read first, then written
  - Vectorizable

```
for(i=0; i<N-1; i++)
  a[i] = a[i+1] + b[i];
```

Loop Dependencies: Synopsis

- Read After Read
  - Not really a dependency
  - Vectorizable

```
for(i=0; i<N; i++)
  a[i] = b[i%2] + c[i];
```

- Write After Write
  - a.k.a “output” dependency
  - Variable written, then re-written
  - Not vectorizable
  - Exception: array sums and products (+=, *=) are vectorizable

```
for(i=0; i<N; i++)
  a[i%2] = b[i] + c[i];
```

Loop Dependencies: Aliasing

- In C, pointers can hide data dependencies!
  - The memory regions that they point to may overlap
- Is this vectorizable?

```
void compute(double *a, double *b, double *c) {
    for (i=1; i<N; i++) {
        a[i] = b[i] + c[i];
    }
}
```

  - …Not if we give it the arguments `compute(a,a-1,c)`
  - In effect, `b[i]` is really `a[i-1]` → Read After Write dependency
- Compilers can usually cope, at some cost to performance

Dependencies and Optimization Reports

- Loop-carried dependencies are a common reason for vectorization failure
- Optimization reports can tell you where the compiler detected apparent dependencies
  - Select a report level that gives info about the loops where vectorization was missed
- Remember, the compiler is conservative: you need to dig into the details of the report and see if dependencies really exist in the code
  - The Intel compiler is generally better than gcc for doing this because it is more concise

Exercise 3

1. Make a copy of abc.c called dep.c. Edit it and change the innermost of the nested loops to look like this:

```
for (i=1; i < ARRAY_SIZE; i++) {
    a[i] += m*b[i] + a[i-1];
}
```

2. Compile the code with vectorization enabled, and request a report with info on loops that missed out:

```
gcc-9 -O3 dep.c -o dep -fopt-info-vec-missed
```

3. Look for notes about dependencies that were detected in the loop starting on line 20 – or grep for “depend”

Loop Dependencies: Vectorization Hints

- Sometimes, it is impossible for the compiler to prove that there is no data dependency that will affect correctness
  - e.g., unknown index offset, complicated use of pointers
- To stop the compiler from worrying, you can give it the IVDEP (Ignore Vector DEPendencies) hint
  - It assures the compiler, “It’s safe to assume no dependencies”
  - Compiler may still choose not to vectorize based on cost
  - Example: assume we know M > vector width in doubles...

```
void vec1(double s1, int M,
   int N, double *x) {
#pragma GCC ivdep      // for Intel, omit GCC
for(i=M; i<N; i++) x[i] = x[i-M] + s1;
```

Intel Pragmas Affecting Vectorization

- #pragma ivdep
  - Compiler ignores apparent dependencies, still considers cost
- #pragma vector always
  - Vectorizes the loop if it is correct to do so (no dependencies)
  - Overrides a decision not to vectorize based upon cost
- #pragma simd
  - Vectorizes the loop regardless of apparent dependencies or cost
  - Combines “vector always” and “ivdep”
  - Being phased out in favor of OpenMP 4.0 “#pragma omp simd”
- #pragma novector
  - Prevents vectorization of a particular loop

OpenMP 4.0 and Vectorization

- #pragma omp simd
  - Can be combined with other OpenMP constructs
  - Has its own special clauses
  - May not be required for all compilers: n order to vectorize the example at right, GCC needs “simd”, Intel doesn’t

```
#pragma omp for simd private(x) reduction(+:sum)
for (j=1; j<=num_steps; j++) {
   x = (j-0.5)*step;
   sum = sum + 4.0/(1.0+x*x);
}
```

DEMO:
How to vectorize pi_loop.c
with the gcc compiler

Loop Dependencies: Language Constructs

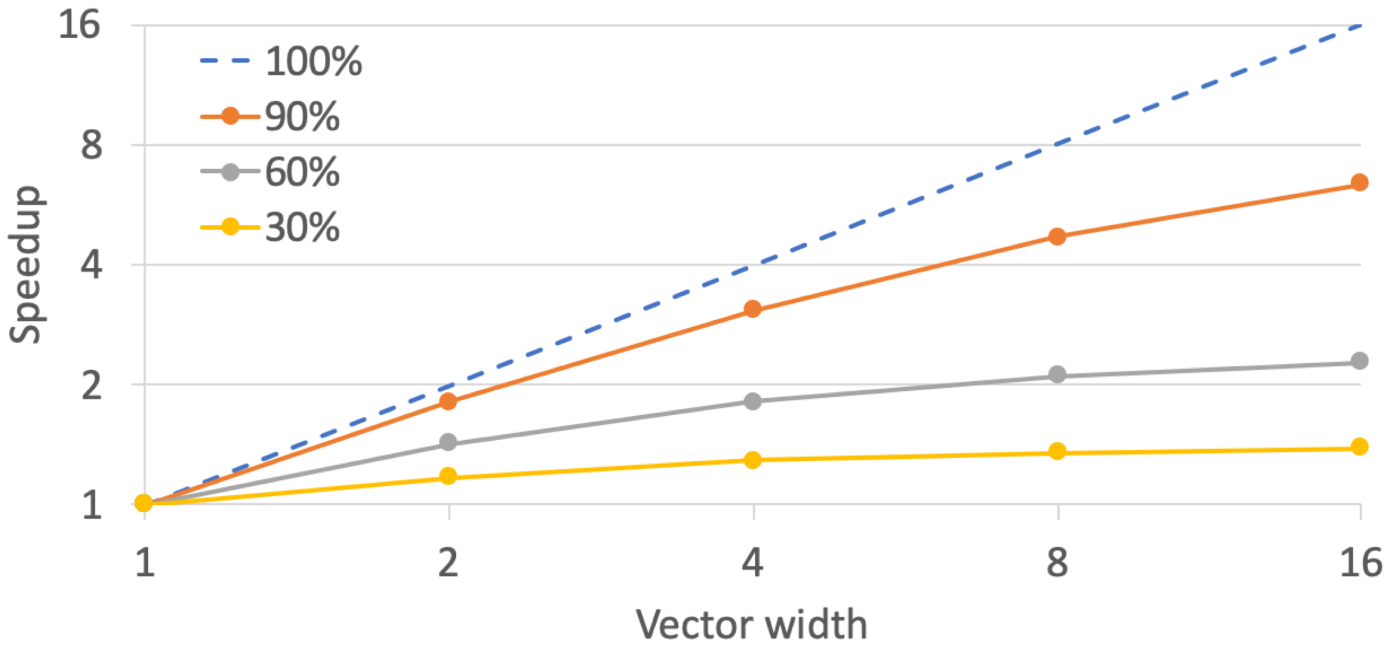
- C99 introduced ‘restrict’ keyword to language
  - Instructs compiler to assume addresses will not overlap, ever

```
void compute(double * restrict a, double * restrict b, double * restrict c) {
    for (i=0; i<N; i++) {
        a[i] = b[i] + c[i];
    }
}
```

- Intel compiler may need extra flags: `-restrict -std=c99`

A Quick Word on Amdahl's Law

- SIMD is parallel, so Amdahl's Law is in effect!
  - Linear speedup is possible only for *perfectly* parallel code
  - Serial/scalar portions of workload will limit performance
  - Asymptote of the speedup curve is 1/(unvectorized fraction)

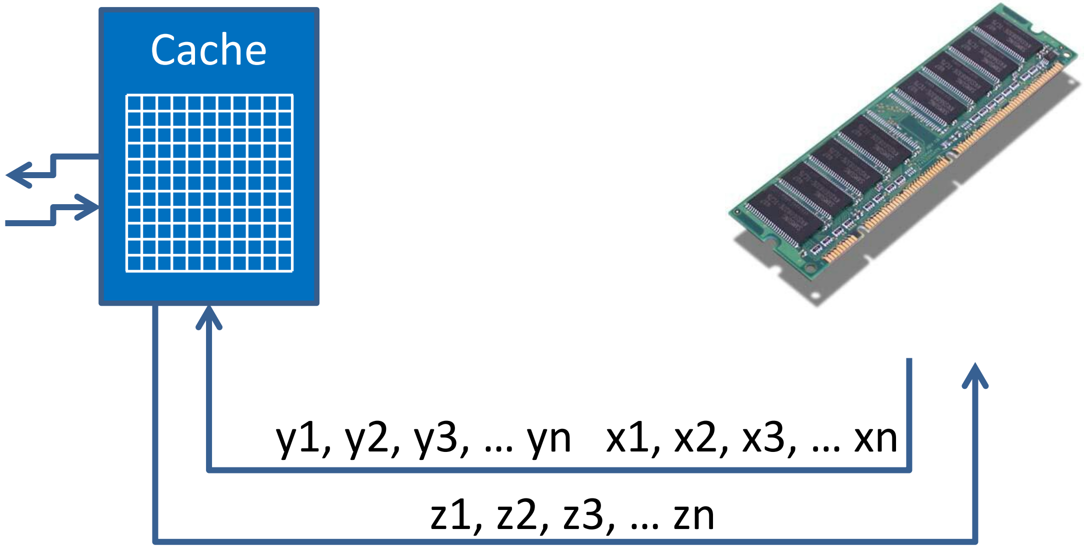


Memory Performance and Vectorization

- We have mostly been focusing on faster flop/s, but flop/s don’t happen unless data are present
  - Moving data from memory is often the rate-limiting step!
- Data (including scalar data + neighbors) travel between RAM and caches in groups called “cache lines” that are the exact same size as vectors
- But wait… if data movement is “vectorized”, just like adds and multiplies are vectorized, then everything is getting the same speedup, right?
  - Um, no. The data rate for RAM is slow, even if it is always “vectorized” in a sense
  - Well… loads from L1 cache to registers, and stores from registers to L1, do get vectorized. But that’s just the final short step if the data start way out in RAM

Cache and Alignment

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_n \end{bmatrix} = a * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$



- Optimal vectorization takes you beyond the SIMD unit!
  - Cache lines start on 16-, 32-, or 64-byte boundaries in memory
  - Sequential, aligned access is much faster than random/strided

Strided Access

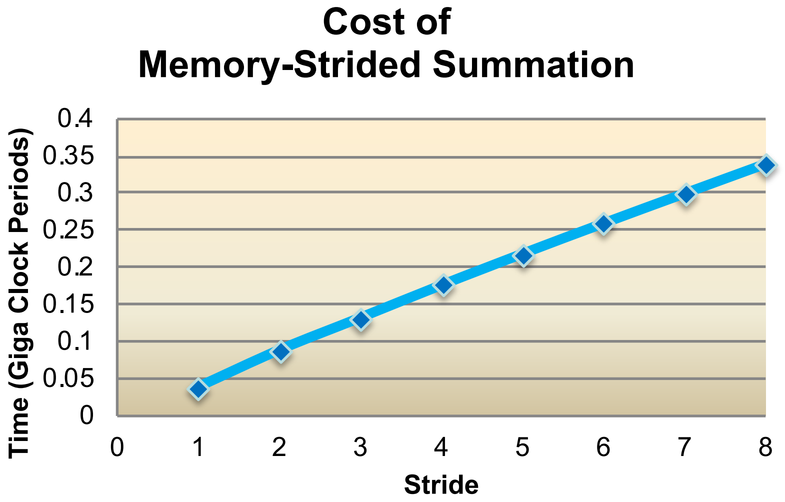
- Fastest usage pattern is “stride 1”: perfectly sequential
  - Cache lines arrive in L1d as full, ready-to-load vectors
- Stride-1 constructs:
  - Storing data in structs of arrays vs. arrays of structs
  - Looping through arrays so their “fast” dimension is innermost
    - C/C++: stride 1 on last index (columns)
    - Fortran: stride 1 on first index (rows)

```
for(j=0;j<n;j++) {
    for(i=0;i<n;i++) {
        a[j][i]=b[j][i]*s;
    }
}
```

```
do j=1,n
    do i=1,n
        a(i,j)=b(i,j)*s
    end do
end do
```

Penalty for Strided Access

- Striding through memory reduces effective memory bandwidth!
  - Roughly by 1/(stride)
- Why? For some stride *s*, data must be "gathered" from *s* cache lines to fill a vector register
- It's worse than non-aligned access



```
for (i=0; i<4000000*istride;
          i+=istride) {
      a[i] = b[i] + c[i]*sfactor;
}
```

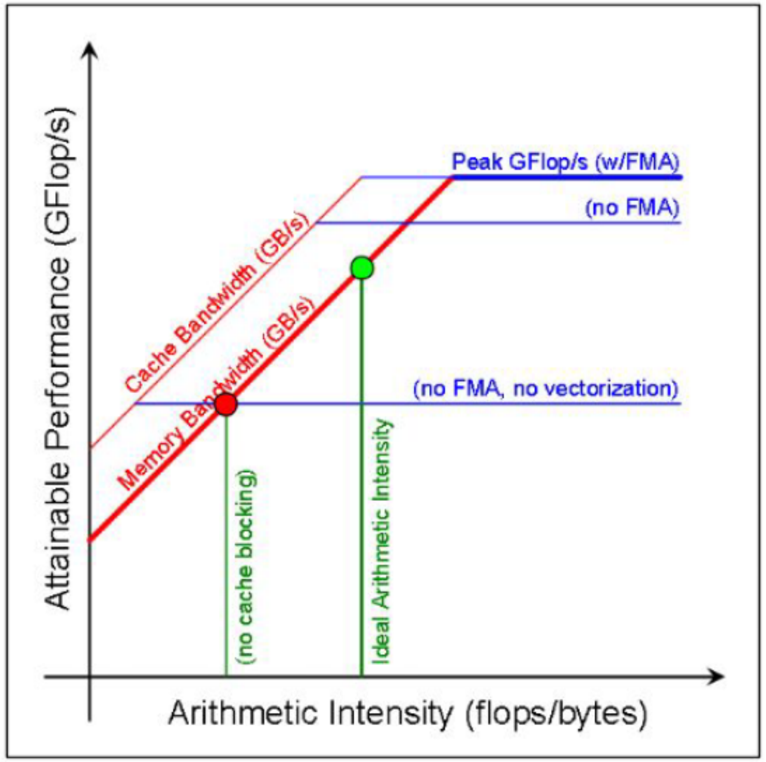
Diagnosing Cache & Memory Deficiencies

- Really bad stride patterns may prevent vectorization
  - The Intel vector report might say: “vectorization possible but seems inefficient”
- Bad stride and other problems may be difficult to detect
  - The result is merely poorer performance than might be expected
- Profiling tools like Intel VTune can help
- Intel Advisor makes recommendations based on source

Towards Peak Flop/s: Arithmetic Intensity

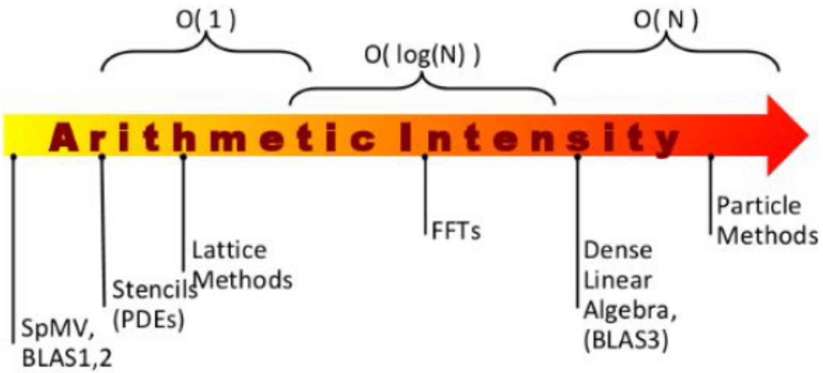
- Even a simple stride-1 loop may not get peak flop/s rate!
- The most common reason is that the *arithmetic intensity* (AI = flop/byte) of the code is too low
  - VPU becomes stalled waiting for loads and stores to complete
  - Delays become longer as the memory request goes further out in the hierarchy from L1 to L2 (to L3?) to RAM
  - Even if the right vectors are in L1 cache, there is limited bandwidth to the registers
- If the goal is to maximize flop/s, you’ll want to try to improve AI
- Also want threads to work on independent, cache-size chunks of data
  - Watch out for false sharing, where 2 threads fight needlessly over a cache line

Roofline Analysis



$$\begin{array}{c}\mathit{Attainable}\\ \mathit{FLOPs/sec}\end{array} = \min\begin{cases}\mathit{Peak\,FLOPs/sec}, \\ \begin{array}{c}\mathit{Peak\,Memory}\\ \mathit{Bandwidth}\end{array} \; x \; \begin{array}{c}\mathit{Arithmetic}\\ \mathit{Intensity}\end{array}\end{cases}$$

$$\mathit{Arithmetic\ Intensity} = \frac{\mathit{Total\ FLOPs}}{\mathit{Total\ Bytes}}$$



Deslippe et al., “Guiding Optimization Using the Roofline Model,” tutorial presentation at IXPUG2016, Argonne, IL, Sept. 21, 2016. https://anl.app.box.com/v/IXPUG2016-presentation-29

What Does Roofline Analysis Tell You?

- Roofline analysis is a way of telling whether a code is compute bound or *memory bound*
- The “roofline” is actually a performance ceiling which is determined by *hardware characteristics*
- The key parameter is the *arithmetic intensity* or AI (flop/byte) of the code: it tells you whether data can be loaded [stored] fast enough from [to] memory
- Appropriate to use with codes that are looking to achieve the highest flop/s rate possible

What AI is Required for Peak Flop/s?

- Again, a typical processor core from Intel can do 2 vector flops on every cycle (FMA = 1 add + 1 multiply)
- A typical processor can also do 1 vector load on every cycle, and 1 vector store every other cycle, which implies:
  - Main vectorized loop must do at least 2 flops *per load*
  - Main vectorized loop must do at least 4 flops *per store*
- Implications: 50% of operands must be either *vectors of constants*, or *variables that aren't reloaded* on every iteration; and, every stored result must take 2 or more FMAs
  - For the latest server-class processors, all the above per-core rates are doubled; implications are the same

Conclusions: Vectorization Basics

- The compiler “automatically” vectorizes tight loops
- Write code that is vector-friendly
  - Innermost loop accesses arrays with stride one
  - Loop bodies consist of simple multiplications and additions
  - Data in cache are reused; loads are stores are minimized
- Write code that avoids the potential issues
  - No loop-carried dependencies, branching, aliasing, etc.
- This means you know where vectorization should occur
- Optimization reports will tell you if expectations are met
  - See whether the compiler’s failures are legitimate
  - Fix code if the compiler is right; use #pragma if it is not

PART II:

Performance Problems in Vectorized Track Finding

(original author: Matevž Tadel, UCSD)

History of the Project

- NSF grant: *“Particle Tracking at High Luminosity on Heterogeneous, Parallel Processor Architectures”*
  - Cornell, Princeton, UCSD ➛ all CMS
  - HL-LHC: high pile-up, 200 interactions per bunch crossing
  - New computer architectures: MIC / AVX-512, GPUs, ARM-64
  - Goal: make tracking software more general and faster!
- Proposal: explore how far we can get by enhancing the parallelism of existing, production tracking algorithms:
  - Keep well-known physics performance – efficiencies, fake rates
  - Make code amenable to vectorization and multi-threading, through new data structures and generalized algorithms

Complexity of Tracking

- Number of hits grows linearly with N of tracks
  - Combinatorial explosion: L layers gives N^L combinations of hits
- Have to use cuts and cleverness to limit the search space
  - Traditional tracking: add hits from every layer, limit explosion by limiting total number of track candidates considered per seed (i.e., starter track with 3 or 4 hits)
  - Tracklets & cellular automata ➛ divide and conquer; e.g., with three layers for tracklets, there is only N^3 growth
- In the end we want both speed and physics performance
  - Can be different for different applications / stages of processing

Main Parts of Track Finding & Fitting

- Propagation to next hit / sensor / layer
  - Costliest part is calculating derivatives for error propagation
  - Can rely on automatic vectorization by compiler:

```
#pragma simd
for t in [ tracks ]
    » about 80 lines of calculations …
```

- Hit selection
  - This is hard, depends on space-partitioning data structures
  - Will not be covered here
- Kalman update
  - Can’t rely on automatic vectorization for lots of small matrices
  - The rest of the talk is mostly about this

Objects in Track Finding & Fitting

- Hit: 3-vec of position, 3x3 symm. covariance matrix, label
  - 40 bytes – a bit less than cache line
- Track: 6-vec of pos+mom, 6x6 sym cov matrix, hit indices
  - Keep just indices of assigned hits – 256 bytes – 4 cache lines
  - More traditional representation is 5 + 5x5 sym
  - In 6x6 the covariance matrix is notably simpler (block diagonal!), but one needs a way to exploit this
- Kalman Filter: a set of operations using the above objects
  - Mostly multiplications; intermediate results are also 6x3 matrices; product of symmetric matrices is not symmetric
  - Similarity operation
  - 3x3 matrix inversion

Vectorization – Factors

- Architecture
  - Number and width of vector registers
  - Memory hierarchy, esp. L1 size and latency
- Algorithms
  - HEP algorithms seldom allow automatic compiler vectorization
  - Need to add an implicit or explicit inner loop in key sections, sometimes generated by compiler (with -O3 or specific options)
  - Can vectorize by hand in some cases (next section); need to deal with edge effects and imbalance
- Data structures
  - Size, alignment, reuse of data in registers, L1, L2,…

Outline

- First encounter with vectorization on a new architecture and compiler
  - Exercise – absolute floating point performance estimation
- Vectorizing small matrix operations – Matriplex
  - Semi-automatic vectorization of matrix operations
- Conclusion
  - Hardware limitations

FIRST ENCOUNTER WITH VECTORIZATION ON A NEW ARCHITECTURE AND COMPILER

How to Know *The Right Thing to Do*™

- Usually we tune existing code:
  - Run profilers → fix hotspots
  - Measure relative speed-up & declare victory
  - But: how much performance was left on the table?
- When trying something really new you want to know absolute performance
  - How many floating point operations am I doing compared to the peak of the architecture?
  - We were in this situation with vectorization / Xeon Phi / Intel compiler back in 2013

mtorture – Machine Torture

- Torture machine (and yourself) until performance and compiler behavior is understood: https://github.com/osschar/mtorture
- All tests have basic dependence on problem size, i.e., number of elements processed in the same inner loop
  - Loop many times over the same data – sum up flops, measure time
  - For smaller problem sizes the data will fit into L1, then L2, L3
  - No manual prefetching – we let compiler / CPU do whatever it does
  - Maybe with manual prefetching better results could be obtained for larger N
- Absolute efficiency metric requires absolute normalization:
  - CPU clock speed
  - Width of the vector unit

Test t1 – ArrayTest

Objective: Find out performance of your laptop CPU / compiler.

- **common.h:** definitions of aligned operator new, compiler hints, global / environment variables
- **ArrayTest.h /.cxx:** do requested operation over n elements in float arrays:
    - sum2: c = a + b → 1 * n ops
    - sum2_sqr: c = a*a +2*a*b + b*b → 6 * n ops
    - …see others in ArrayTest.h /.cxx
  - All return number of floating point operations performed
- **Timing.h /.cxx:** takes a function object, runs it repeatedly until approximately TEST_DURATION
  - Sums up the operation count, knows the execution time and assumed clock frequency => calculates flops
  - From assumed vector unit width can estimate effective vector unit usage
- **t1.cxx:** takes a test (e.g. sum2) and runs it for n : N_VEC_MIN to N_VEC_MAX, n = 2 * n
  - print results in format accepted by TTree::ReadFile()

Example of a Low Level Test: sum2

```
long64 ArrayTest::sum2(int n)
{
  float *Z = fA[0];
  float *A = fA[1];
  float *B = fA[2];

  ASSUME_ALIGNED(Z, 64); // Vector loads/stores can be done directly
  ASSUME_ALIGNED(A, 64);
  ASSUME_ALIGNED(B, 64);
  ASSUME(n%16, 0);  // No trailing elements, always full vector width

#pragma simd    // Force compiler to vectorize, no array dependencies
  for (int i = 0; i < n; ++i)
  {
    Z[i] = A[i] + B[i];
  }

  return n;
}
```

Exercise: Do the Following

```
git clone git@github.com:osschar/mtorture.git
cd mtorture

# Check max frequency of your CPU, see the top of README.md
# Edit Timing.cxx to set your frequency.
# Can also set vector unit width - default is 8 (assume AVX)

# For macOS, modify Makefile, set CXX to your gcc,
# this should be enough for MacPorts: CXX := c++-mp-5
# this should be enough for Homebrew: CXX := c++-9
make t1
TEST_DURATION=0.1 ./t1
# should print out about 30 lines of numbers

# Now, run a bunch of different tests, packed in the script:
./codas.sh
# This will store the output into independent files, e.g.,
# arr_sum2_O3.rt. Should take about 2 minutes ...
```

Output Explanation

This output data can be read by TTree::ReadTree(), e.g.:

```
matevz@glut mtorture> ./t1
NVec/I:Time/D:Gops:Gflops:OpT:VecUt ⟵ branch names and types (I-int, D-double)
#   NVec        Time        Gops      Gflops      OpT    VecUt
       1    0.963609    1.563768    1.622824   0.6242   0.0780
...
      32    1.052645   16.436191   15.614183   6.0055   0.7507
      64    1.001637   17.609308   17.580529   6.7617   0.8452
     128    0.774350   14.021096   18.106919   6.9642   0.8705
     256    1.016159   18.213868   17.924229   6.8939   0.8617
     512    1.036856   19.447122   18.755853   7.2138   0.9017
    1024    1.043660   19.826760   18.997337   7.3067   0.9133
...
```

Time: actual runtime, in principle only tells you if Timing calibration was ok

Gops: giga operations during the test (as reported by the test function)

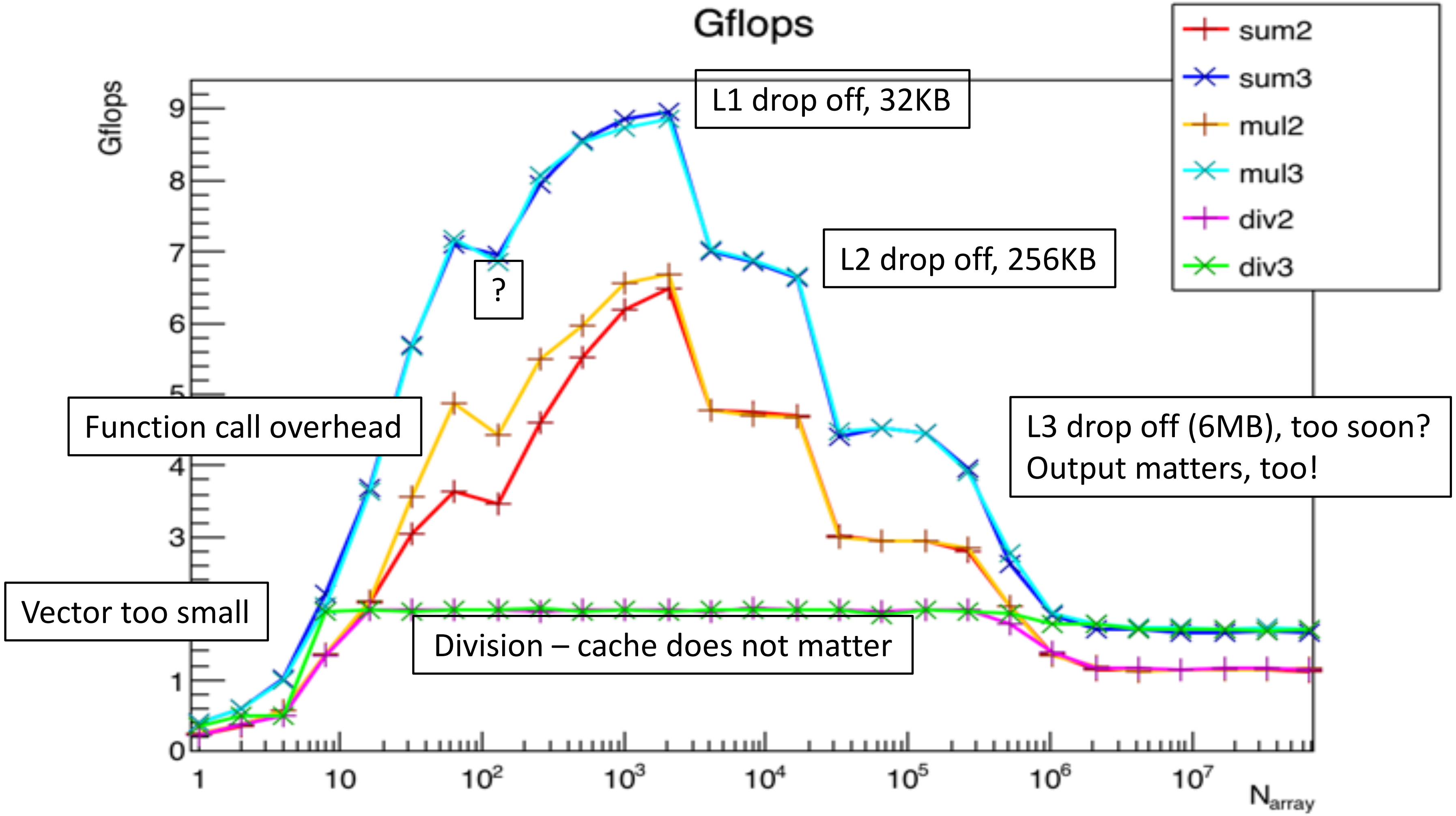
Gflops: giga operations per second – Gops / Time

OpT: operations per clock tick – Gflops / *given CPU frequency*

VecUt: vector utilization – OpT / *given vector unit width*

Example Plot, Gflops

Plotter.C – ROOT macro / class for plotting a set of such outputs on the same plot. E.g. plot_min() for a laptop (2.6 GHz, V_w = 8):



L1 drop off, 32KB

?

L2 drop off, 256KB

Function call overhead

L3 drop off (6MB), too soon? Output matters, too!

Vector too small

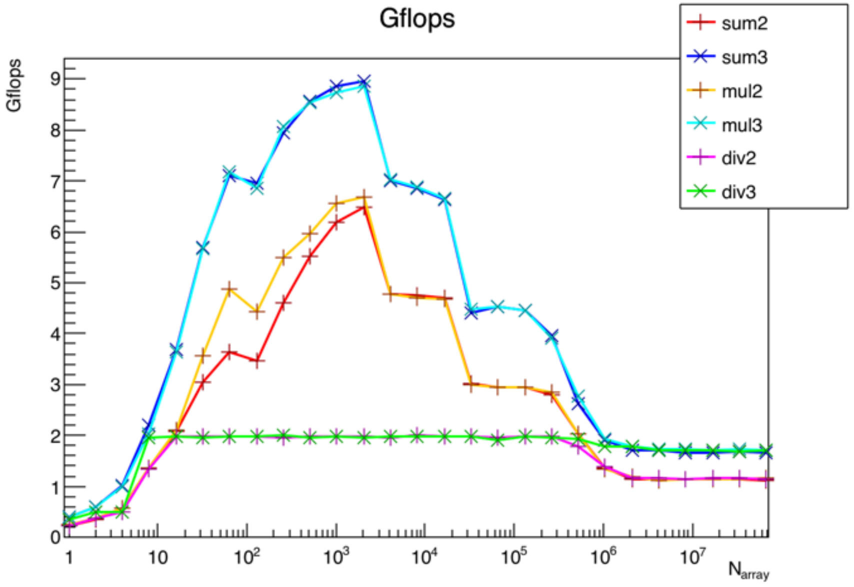
Division – cache does not matter

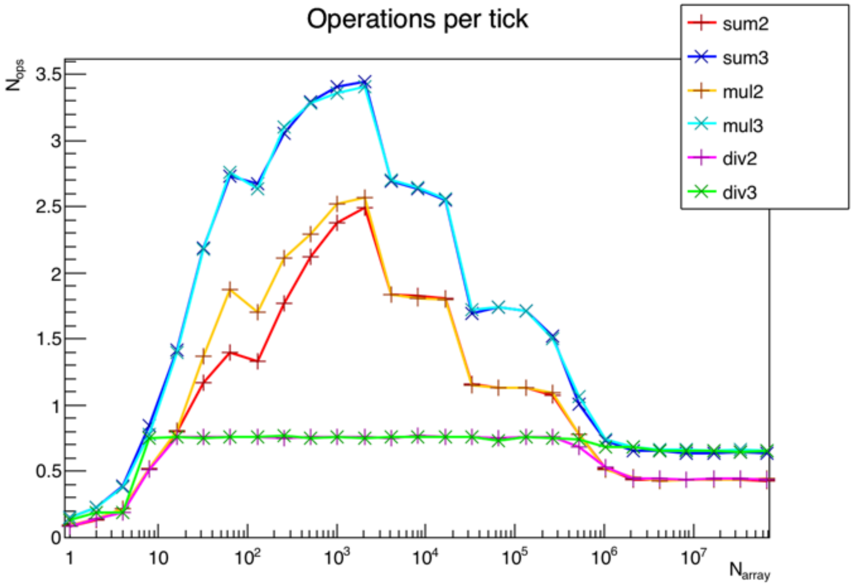
Your Turn to Plot Some Graphs

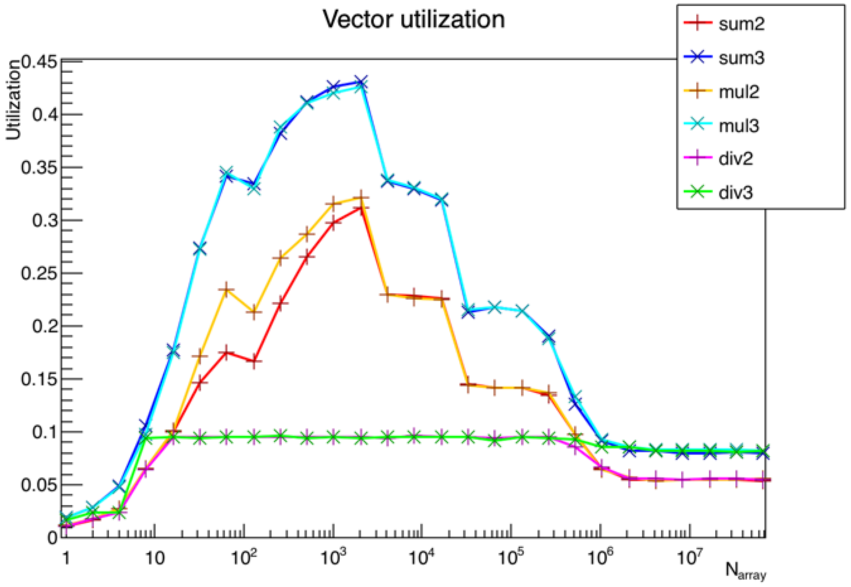
```
# assuming you successfully ran codas.sh
# setup ROOT environment
source <path-to-root/bin>/thisroot.sh

# codas.C has a set of predefined multigraphs:
root codas.C
>> plot_min()

# Compare to results for Matevz’s laptop.
# Poorer? Try setting OPT:= -O3 -mavx
# Or try setting OPT := -O3 -march=native
```

Normalization with Clock Speed, Vector Width

- It’s really just normalization.
  - Helps you see different plateaus
  - On next slides will mostly show Vector Utilization
- Why is vector utilization low?
  - B/W limited!



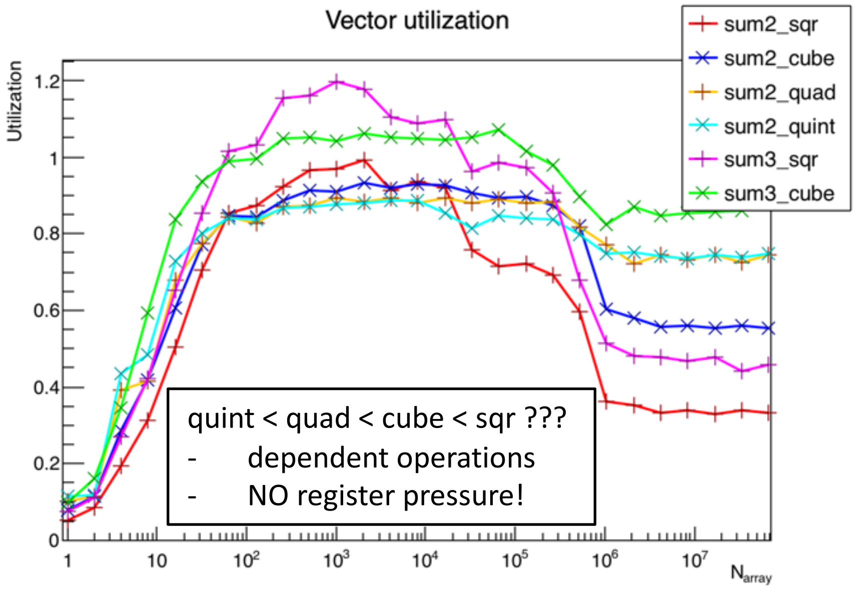




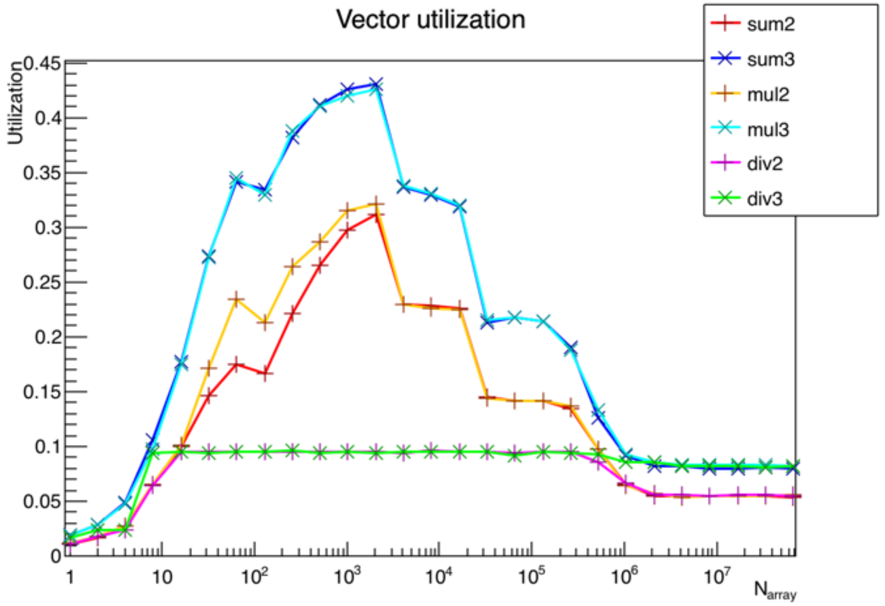
plot_sig() – Significant Computation / Load

- All cache effects less pronounced.
  - High op tests don’t mind L1 at all.
- Vector Utilization > 1?
  - Multiple vector units! FMA operations!
    - Can also be too low clock in Timing.cxx / turbo!
  - Gets even more pronounced on desktop / server CPUs.

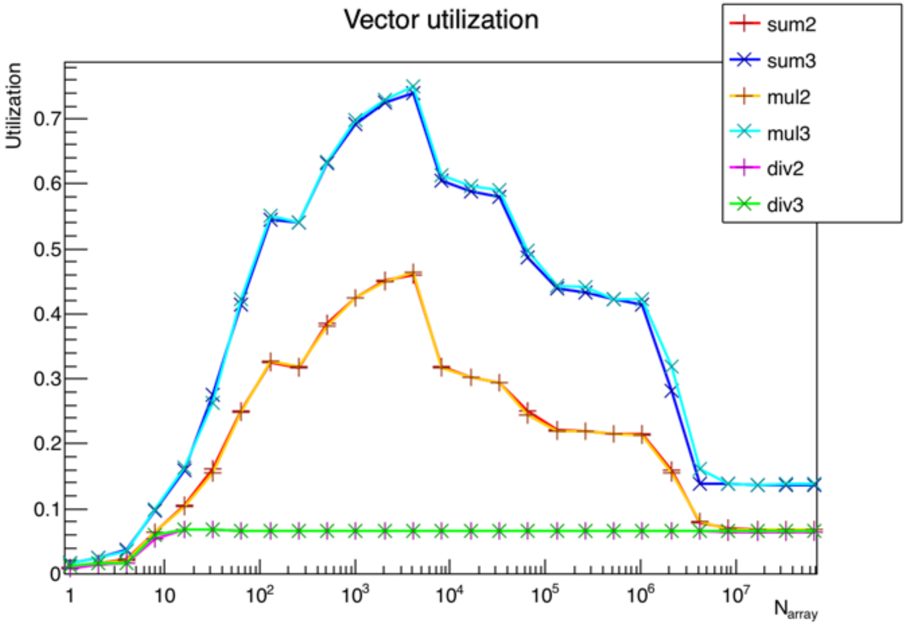
test (2+1)	N_{ops}	test (3+1)	N_{ops}
sum2	1	sum3	2
sum2_sqr	6	sum3_sqr	12
sum2_cube	10	sum3_cube	22
sum2_quad	15		
sum2_quint	19		



quint < quad < cube < sqr ???
- dependent operations
- NO register pressure!

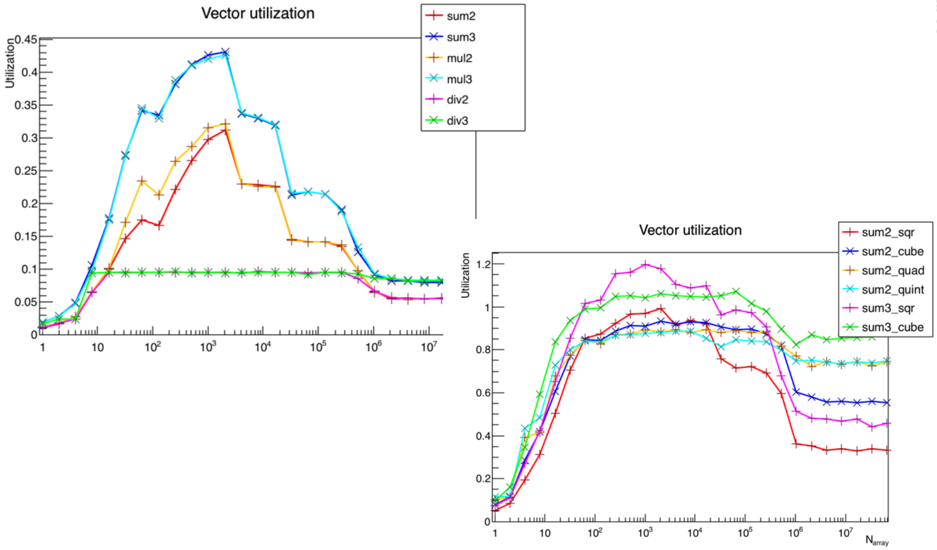


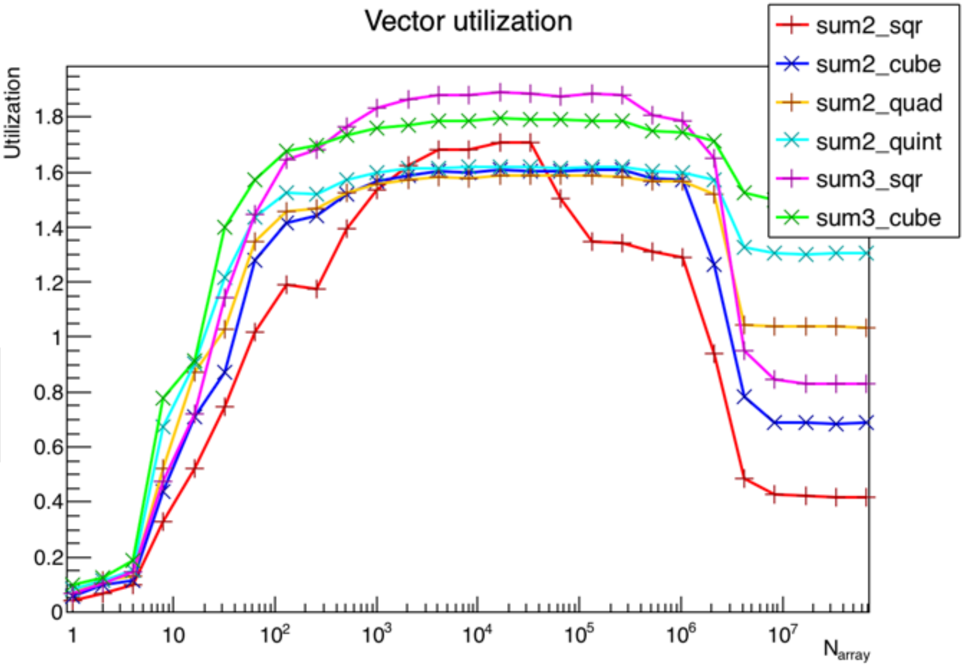
Compare Laptop to Desktop CPU



Better, right?

- more vector processing units
- FMAs
- deeper pipelines
- larger L3 (e.g, 8 vs. 6 MB)

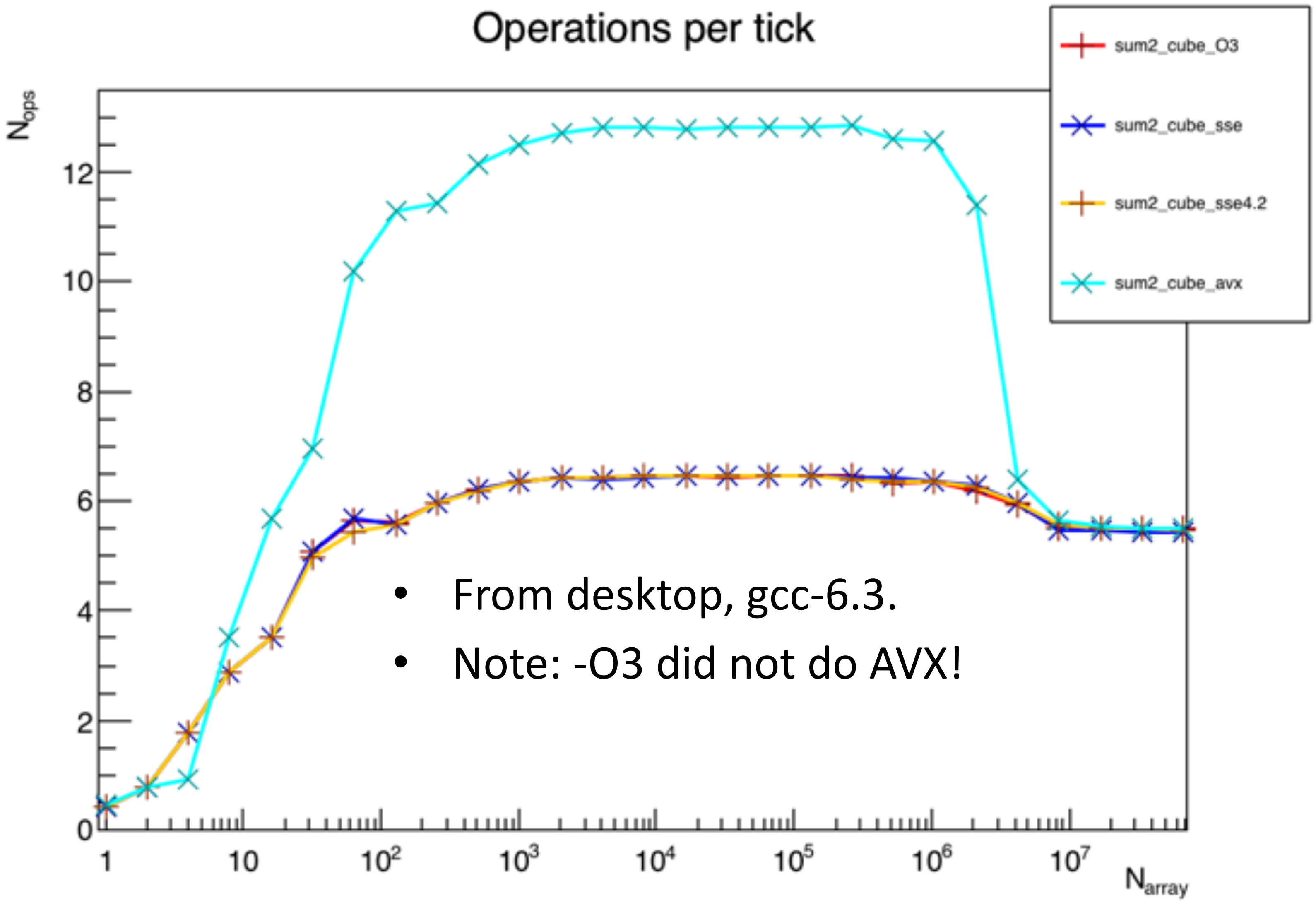




Advanced Usage

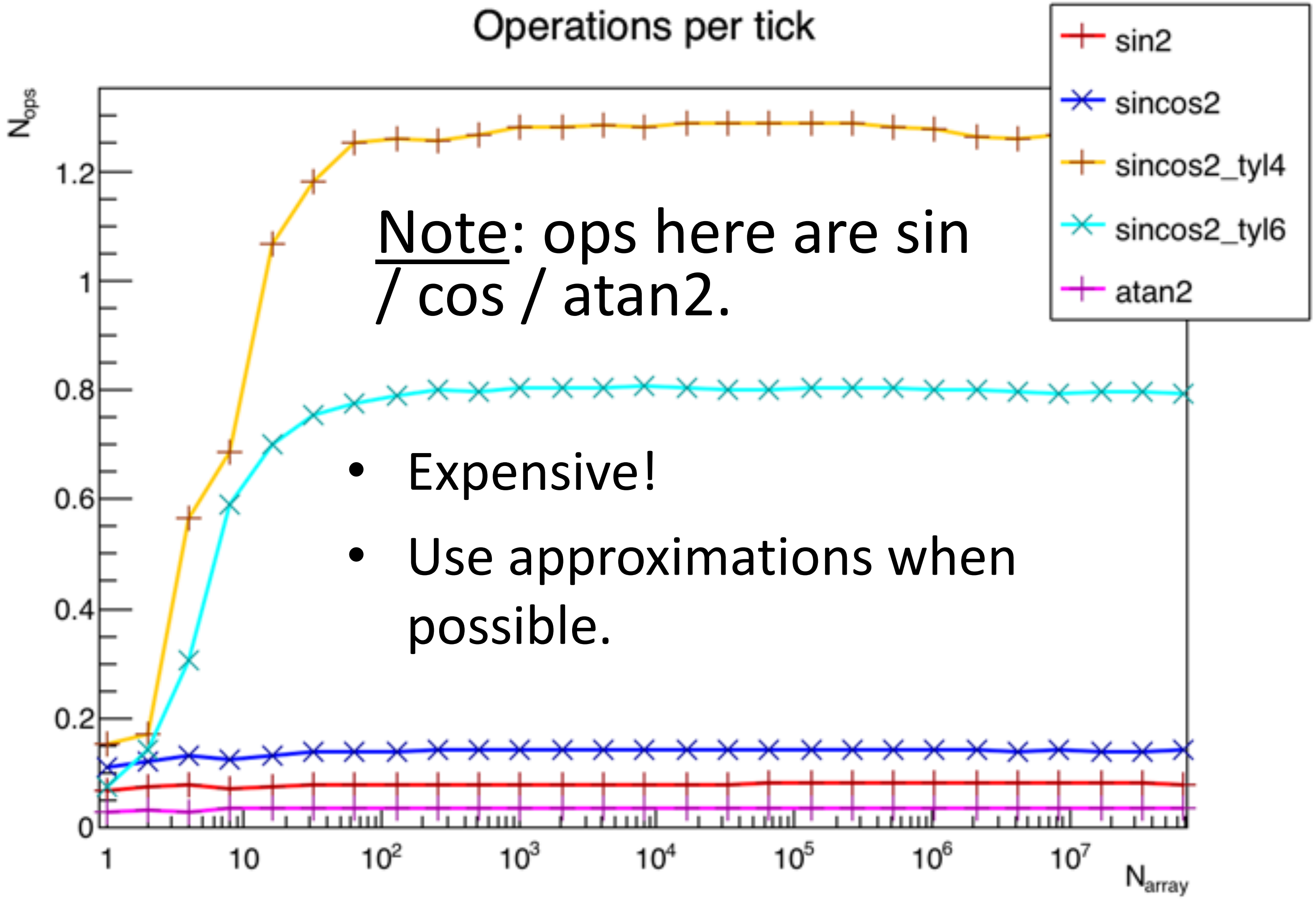
- When changing compiler options or compile time constants the test needs to be recompiled for every invocation.
  - That’s what the perl library Test.pm and perl scripts t1.pl are doing there.
- See codas.sh, parts that are commented out:
  - run_vset() – use different vectorization instruction sets:
    - -msse4.2, -mavx
    - Note: some might not be available on your machine.
  - Use plot_vecopt() in codas.C to compare results.
  - run_trig() – trigonometric functions / plot_trig()

plot_vecopt() – Effect of Vectorization Opts



- From desktop, gcc-6.3.
- Note: -O3 did not do AVX!

plot_trig() – Trigonometric Functions



Note: ops here are sin / cos / atan2.

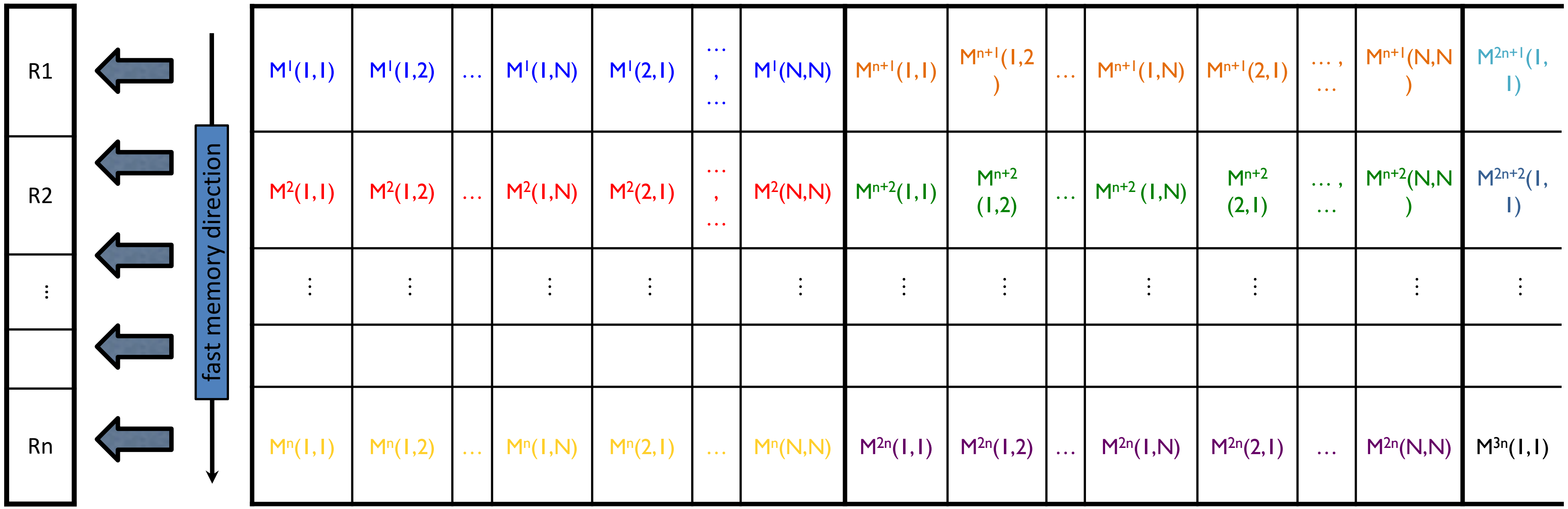
- Expensive!
- Use approximations when possible.

VECTORIZING MATRIX OPERATIONS – MATRIPLEX

Matriplex – Introduction

- Nearly impossible to vectorize small matrix/vector ops
  - Many multiplications and additions, but pattern keeps changing
  - Lots of shuffling and replication as a consequence
- Expand the ops by doing V_W (8 or 16) matrices in parallel!
  - Matriplex is a library that helps you do it in optimal fashion
  - Effectively, #pragma simd over N matrix multiplications
  - Requires all the matrices to be present in L1 at the same time
  - Pressure on cache and registers
    - » 6x6 floats * 4 Bytes * 3 operands * 8 = 3456 Bytes
    - » 6x6 floats * 4 Bytes * 3 operands * 16 = 6912 Bytes

Matriplex – The Idea



$M^1(1,1)$	$M^1(1,2)$	…	$M^1(1,N)$	$M^1(2,1)$	…, …	$M^1(N,N)$	$M^{n+1}(1,1)$	$M^{n+1}(1,2)$	…	$M^{n+1}(1,N)$	$M^{n+1}(2,1)$	…, …	$M^{n+1}(N,N)$	$M^{2n+1}(1,1)$
$M^2(1,1)$	$M^2(1,2)$	…	$M^2(1,N)$	$M^2(2,1)$	…, …	$M^2(N,N)$	$M^{n+2}(1,1)$	$M^{n+2}(1,2)$	…	$M^{n+2}(1,N)$	$M^{n+2}(2,1)$	…, …	$M^{n+2}(N,N)$	$M^{2n+2}(1,1)$
⋮	⋮		⋮	⋮		⋮	⋮	⋮		⋮	⋮		⋮	⋮
$M^n(1,1)$	$M^n(1,2)$	…	$M^n(1,N)$	$M^n(2,1)$	…	$M^n(N,N)$	$M^{2n}(1,1)$	$M^{2n}(1,2)$	…	$M^{2n}(1,N)$	$M^{2n}(2,1)$	…	$M^{2n}(N,N)$	$M^{3n}(1,1)$

- "Matrix-major" memory representation
  - Operate on a number of NxN matrices in parallel
    - n = vector unit width (or some multiple of it, as long as things fit in L1)
  - Trivial loading of vector registers
  - Requires repacking of input data

N-way SIMD with 3x3 Matrices

```
static void Multiply(const MPlex<T, 3, 3, N>& A,
                     const MPlex<T, 3, 3, N>& B,
                     MPlex<T, 3, 3, N>& C)
{
   const T *a = A.fArray; ASSUME_ALIGNED(a, 64);
   const T *b = B.fArray; ASSUME_ALIGNED(b, 64);
         T *c = C.fArray; ASSUME_ALIGNED(c, 64);
#pragma simd
   for (int n = 0; n < N; ++n)
   {
      c[ 0*N+n] = a[ 0*N+n]*b[ 0*N+n] + a[ 1*N+n]*b[ 3*N+n] + a[ 2*N+n]*b[ 6*N+n];
      c[ 1*N+n] = a[ 0*N+n]*b[ 1*N+n] + a[ 1*N+n]*b[ 4*N+n] + a[ 2*N+n]*b[ 7*N+n];
      c[ 2*N+n] = a[ 0*N+n]*b[ 2*N+n] + a[ 1*N+n]*b[ 5*N+n] + a[ 2*N+n]*b[ 8*N+n];
      c[ 3*N+n] = a[ 3*N+n]*b[ 0*N+n] + a[ 4*N+n]*b[ 3*N+n] + a[ 5*N+n]*b[ 6*N+n];
      c[ 4*N+n] = a[ 3*N+n]*b[ 1*N+n] + a[ 4*N+n]*b[ 4*N+n] + a[ 5*N+n]*b[ 7*N+n];
      c[ 5*N+n] = a[ 3*N+n]*b[ 2*N+n] + a[ 4*N+n]*b[ 5*N+n] + a[ 5*N+n]*b[ 8*N+n];
      c[ 6*N+n] = a[ 6*N+n]*b[ 0*N+n] + a[ 7*N+n]*b[ 3*N+n] + a[ 8*N+n]*b[ 6*N+n];
      c[ 7*N+n] = a[ 6*N+n]*b[ 1*N+n] + a[ 7*N+n]*b[ 4*N+n] + a[ 8*N+n]*b[ 7*N+n];
      c[ 8*N+n] = a[ 6*N+n]*b[ 2*N+n] + a[ 7*N+n]*b[ 5*N+n] + a[ 8*N+n]*b[ 8*N+n];
   }
}
```

Matriplex – Features

- Only needed operations were implemented (+ test stuff)
- GenMul.pm – generates matrix multiplications using Perl
  - Standard and symmetric matrices supported
  - In-code transpose (for similarity transformation)
  - Takes advantage of known 0 and 1 elements
  - Accounts for operation latencies in accumulating dot products
  - Generates standard C++ (unrolled loops) or intrinsics
    - » intrinsics done for MIC, AVX, and AVX512 (no FMA on AVX, came with AVX2)
    - » Improvements in icc have largely done away with the need for intrinsics
  - Initial tests were done with icc in 2013/4
    - » loop unrolling brought ~x2 speedup
    - » and intrinsics another x2 (at least on MIC)

Example Script Using GenMul.pm

```
use GenMul;

my $DIM = 6;

### Propagate Helix To R -- final similarity, two ops.
# outErr = errProp * outErr * errPropT
# outErr is symmetric

### MATRIX DEFINITIONS

$errProp = new GenMul::Matrix
  ('name'=>'a', 'M'=>$DIM, 'N'=>$DIM);
$errProp->set_pattern(<<"FNORD");
x x 0 x x 0
x x 0 x x 0
x x 1 x x x
x x 0 x x 0
x x 0 x x 0
0 0 0 0 0 1
FNORD

$outErr = new GenMul::MatrixSym
    ('name'=>'b', 'M'=>$DIM, 'N'=>$DIM);
```

```
$temp    = new GenMul::Matrix('name'=>'c', 'M'=>$DIM,
'N'=>$DIM);

$errPropT = new GenMul::MatrixTranspose($errProp);

### OUTPUT

$m = new GenMul::Multiply;

# outErr and c are just templates ...

$m->dump_multiply_std_and_intrinsic
  ("MultHelixProp.ah”, $errProp, $outErr, $temp);

$temp   ->{name} = 'b';
$outErr->{name} = 'c';

$m->dump_multiply_std_and_intrinsic
  ("MultHelixPropTransp.ah”, $temp, $errPropT, $outErr);
```

Example of Generated Code

```
#define LD(a, i)          _mm512_load_ps(&a[i*N+n])
#define ADD(a, b)         _mm512_add_ps(a, b)
#define MUL(a, b)         _mm512_mul_ps(a, b)
#define FMA(a, b, v)      _mm512_fmadd_ps(a, b, v)
#define ST(a, i, r)       _mm512_store_ps(&a[i*N+n], r)
```

```
#ifdef MIC_INTRINSICS

   for (int n = 0; n < N; n += 64 / sizeof(T))
   {
      __m512 a_0 = LD(a, 0);
      __m512 b_0 = LD(b, 0);
      __m512 c_0 = MUL(a_0, b_0);
      __m512 b_1 = LD(b, 1);
      __m512 c_1 = MUL(a_0, b_1);
…. . .
      __m512 a_12 = LD(a, 12);
      __m512 c_12 = MUL(a_12, b_0);
      __m512 c_13 = MUL(a_12, b_1);
      __m512 c_14 = MUL(a_12, b_3);
      ST(c, 6, c_6);
      ST(c, 7, c_7);
. . . . . .
      ST(c, 33, c_33);
      __m512 c_34 = b_19;
      __m512 c_35 = b_20;
      ST(c, 34, c_34);
      ST(c, 35, c_35);
   }
```

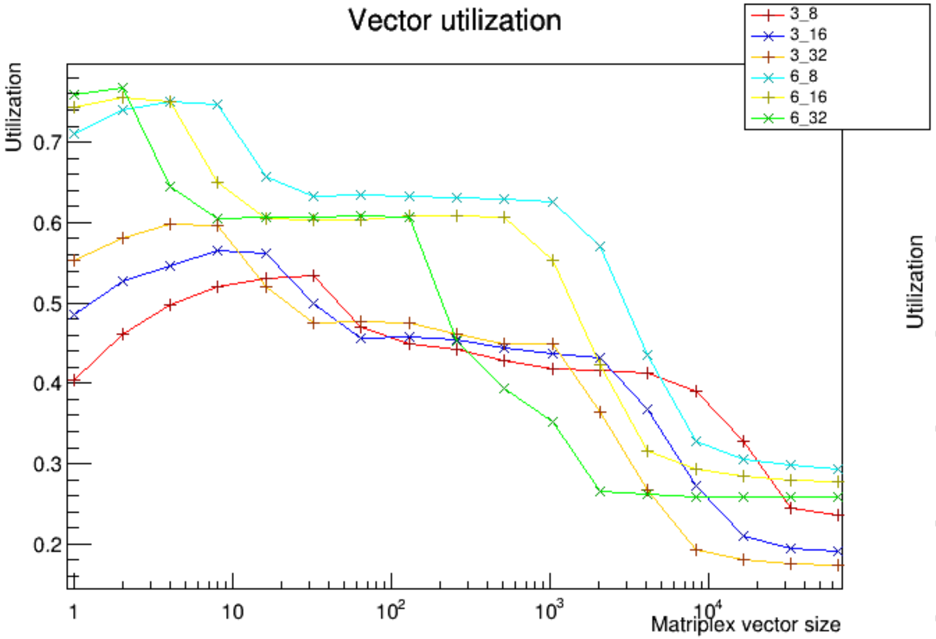
```
#else

#pragma simd
   for (int n = 0; n < N; ++n)
   {
      c[ 0*N+n] =   a[ 0*N+n]*b[ 0*N+n] +
                    a[ 1*N+n]*b[ 1*N+n] +
                    a[ 3*N+n]*b[ 6*N+n] +
                    a[ 4*N+n]*b[10*N+n];
      c[ 1*N+n] =   a[ 0*N+n]*b[ 1*N+n] +
                    a[ 1*N+n]*b[ 2*N+n] +
                    a[ 3*N+n]*b[ 7*N+n] +
                    a[ 4*N+n]*b[11*N+n];
      c[ 2*N+n] =   a[ 0*N+n]*b[ 3*N+n] +
                    a[ 1*N+n]*b[ 4*N+n] +
                    a[ 3*N+n]*b[ 8*N+n] +
                    a[ 4*N+n]*b[12*N+n];
……
      c[33*N+n] =   b[18*N+n];
      c[34*N+n] =   b[19*N+n];
      c[35*N+n] =   b[20*N+n];
   }
#endif
```

Matriplex Templates

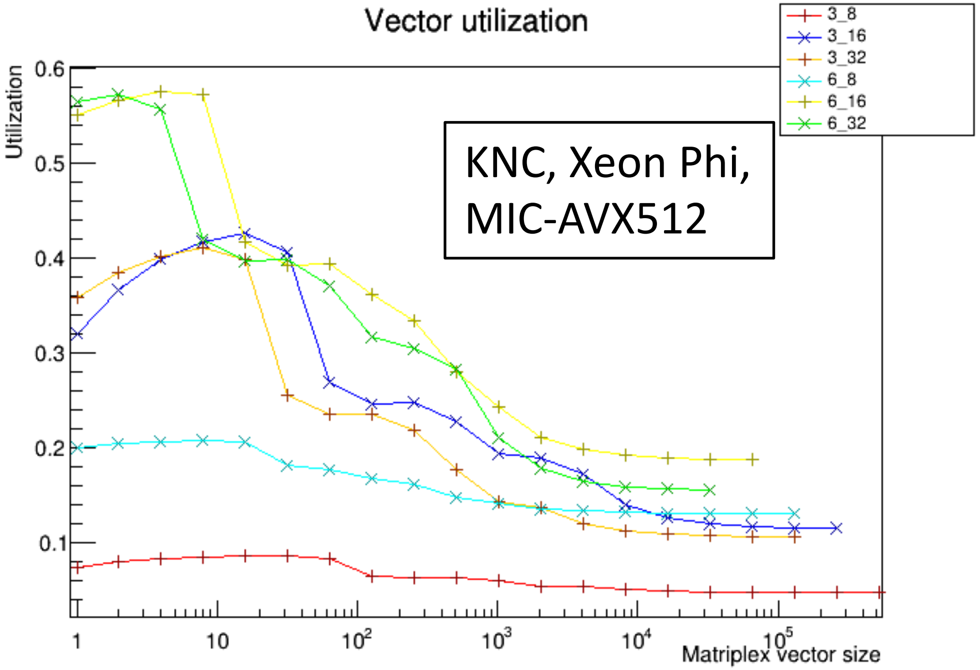
```
template<typename T, idx_t D1, idx_t D2, idx_t N>
class Matriplex
{ enum { kRows = D1, kCols = D2,
         kSize = D1 * D2, kTotSize = N * kSize };

   T fArray[kTotSize] __attribute__((aligned(64)));
   . . .
};
// Covers also vectors with D2 = 1 and scalars with D1 = D2 = 1.

template<typename T, idx_t D, idx_t N>
class MatriplexSym
{  enum { kRows = D, kCols = D,
          kSize = (D + 1) * D / 2, kTotSize = N * kSize };

   T fArray[kTotSize] __attribute__((aligned(64)));
   . . .
};
```

Matrix Multiplication

- Note: x-axis is now Matriplex size
  - First number is dimension
  - Second is Size of the Matriplex (parameter N)



Sandy Bridge, Xeon, AVX



KNC, Xeon Phi, MIC-AVX512

Test t3 – MPlexTest

OUT OF ORDER

- t3.cxx
  - Like t1, just uses MPlexTest
- MPlexTest.h /.cxx
  - Allocates MatriplexVectors, defines test functions
  - Note the MPT_DIM / MPT_SIZE #defines (requires recompiling!)
- Matriplex/MatriplexVector.h
  - Thought this will be useful … now just a testing construct
- Matriplex/Matriplex.h & MatriplexSym.h
  - The real thing™
- codas-mplex.sh, codas.C
  - Runs a bunch of tests with 3x3 and 6x6 matrices, does plots

Getting Data into and out of Matriplexes

- CopyIn
  - Take one std Matrix and distribute it into the plex.
- SlurpIn
  - Build plexes by taking element (i,j) of each matrix.
  - MIC and AVX512 have a special *gather* instruction for input matrices that are addressable from a common address base.
- CopyOut – populate output matrix
  - Jumps over 8 or 16 floats (16 floats is a cache line) – yikes.
  - Copy out is done infrequently and often only for selected parts.
  - It hasn’t shown up on the radar of things to fix yet.
  - CopyIn did and that’s why we have SlurpIn ☺

CONCLUSION

Conclusions: Vectorized Track Finding

- Vectorization can give a significant boost depending on:
  - The ability to express the problem in vector form
  - The problem size and problem complexity
  - The ability to move data through cache hierarchy to VPUs
- Understanding performance at the simplest level is vital
- Memory issues can blur vectorization effects:
  - Multithreading – cache sharing, locking
  - Swapping algorithms or data blocks – cache thrashing
- Use of code-line level profiling is crucial!
  - Don't be afraid of the assembler view, it is often revealing
- Experiment – being frustrated is part of the game ☺