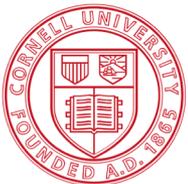


Vector Parallelism on Multi- and Many-Core Processors

Steve Lantz, Cornell University

Matevž Tadel, UC San Diego

CoDaS-HEP Summer School, July 26, 2018



Cornell University
Center for Advanced Computing

UC San Diego

PART I:

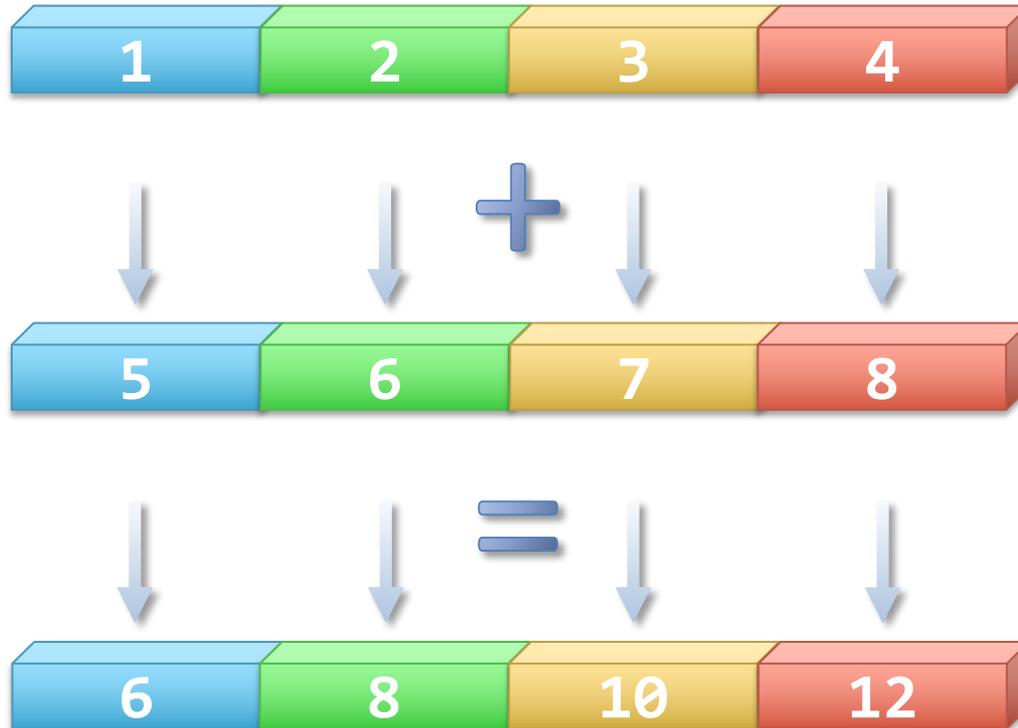
Vectorization Basics

(author: Steve Lantz)

Vector Parallelism: Motivation

- CPU speeds hit a plateau a decade ago
 - Power limitations! “Slow” transistors are more efficient, cooler
- But process improvements keep making space cheaper
 - Moore’s Law! Easy to add 2x more “stuff” every 18–24 months
- One solution: **more cores**
 - First dual-core Intel CPUs appeared in 2005
 - Counts keep growing: 6–28 in Skylake-SP, 61–72 in Xeon Phi
- Another solution: **SIMD or vector operations**
 - First appeared on Pentium with MMX in 1996
 - Vectors keep ballooning: 512 bits (16 floats) in Xeon, Xeon Phi
 - *Vectorization* may increase speed by an order of magnitude

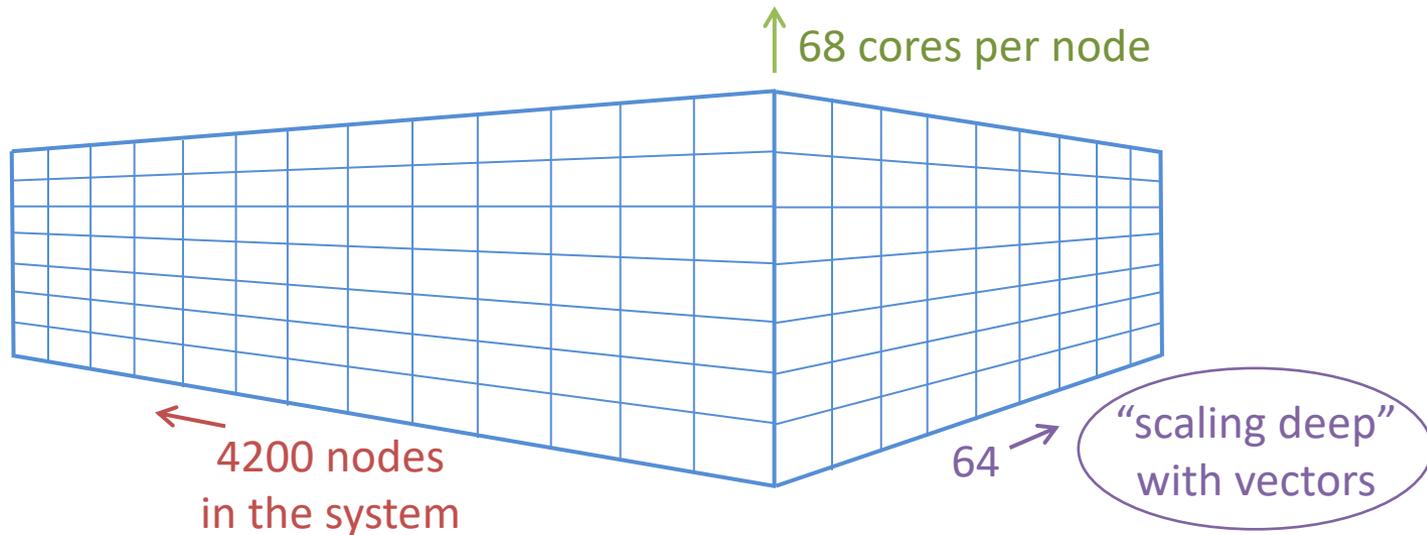
How It Works, Conceptually



SIMD: Single Instruction, Multiple Data

A Third Dimension of Scaling

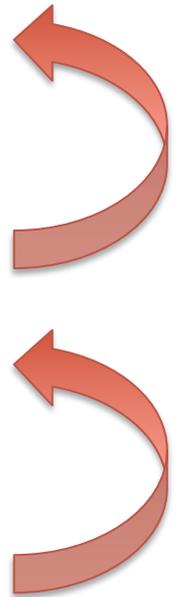
- Along with scaling *out* and *up*, you can “*scale deep*”
 - Arguably, vectorization can be as important as multithreading
- Example: Xeon Phi Knights Landing (KNL) in Stampede2



- 4200 KNL nodes; each KNL has 68 physical cores
- Each core can do **64 operations/cycle** with vectors of 16 floats

Three Ways to Look at Vectorization

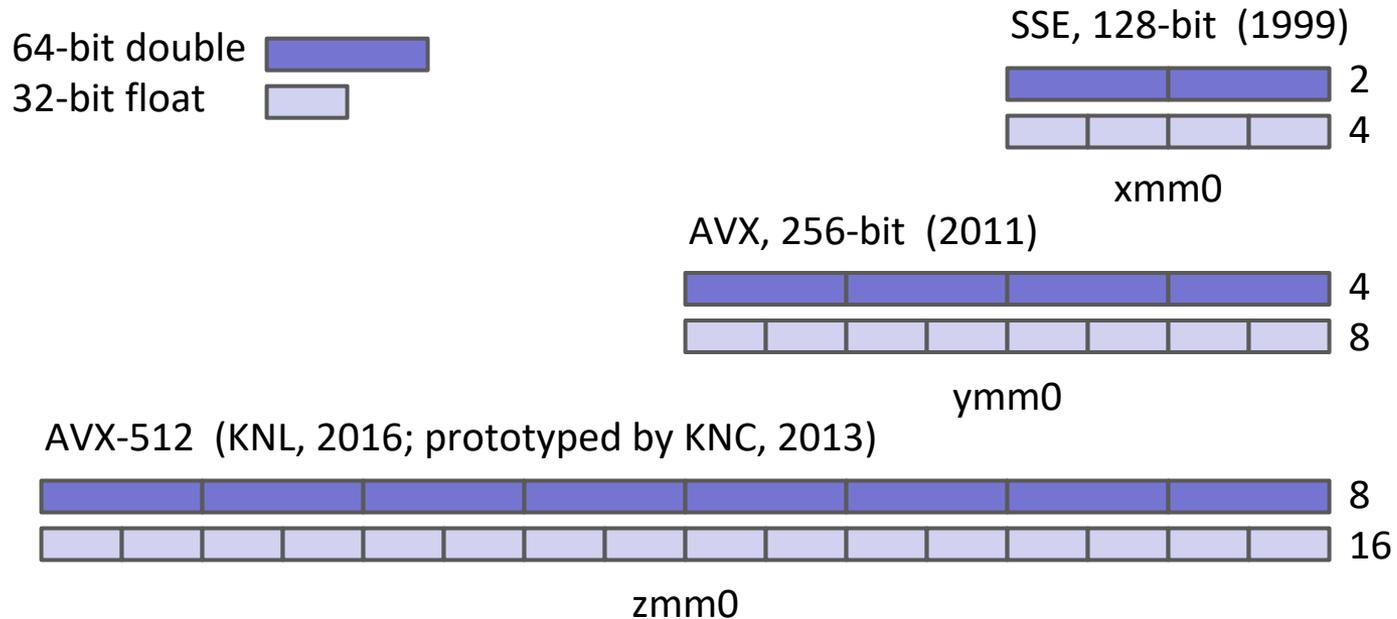
1. **Hardware Perspective**: Run vector instructions involving special registers and functional units that allow in-core parallelism for operations on arrays (vectors) of data.
2. **Compiler Perspective**: Determine how and when it is possible to express computations in terms of vector instructions
3. **User Perspective**: Determine how to write code in a manner that allows the compiler to deduce that vectorization is possible.



Hardware Perspective

- SIMD = Single Instruction Multiple Data
 - Part of commodity CPUs (x86, x64, PowerPC, etc.) since late '90s
- Goal: parallelize computations on vector arrays
 - Line up operands, execute one operation on all simultaneously
- SIMD instructions have gotten quicker over time
 - Initially they took several cycles for execution on small vectors
 - With Intel AVX came pipelining of some SIMD instructions
 - Now one can multiply-and-add large vectors on every cycle
- Intel's latest: Knights Landing (KNL) and Skylake-SP
 - Two VPUs (vector processing units) per core, in most models
 - Each VPU can execute an FMA (Fused Multiply-Add) every cycle

Evolution of Vector Registers, Instructions



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

Peak Flop/s

- Vectorization is essential for attaining peak flop/s
 - Flop/s = floating point operations per second
- Ideal speedup (vs. scalar) is proportional to vector width
- Extra factor of 2 from Fused Multiply-Add (FMA):
 - 128-bit SSE – 4x double, 8x single (pipelined, not truly fused)
 - 256-bit AVX – 8x double, 16x single (pipelined; FMA in AVX2)
 - 512-bit AVX – 16x double, 32x single (FMA)
- Example: Intel Xeon Gold 6130 “Skylake-SP” @ 2.1 GHz
 - **Two** AVX-512 units/core * 1 FMA/cycle = 64 flop/cycle SP
 - 64 flop/cycle/core * 16 cores * 1.9 GHz = 1950 Gflop/s peak SP
 - Note, clock is throttled to 1.9 GHz due to heating at peak load

Why Peak Flop/s Is (Almost) a Fiction

- Assumes all code is perfectly vectorized
 - SIMD is parallel, so Amdahl's Law is in effect, which means...
 - Any serial/scalar portions of code limit the speedup
- Assumes no slow operations like division, square root
- Assumes data are loaded and stored with no delay
 - Only true for data in L1 cache and vector registers
 - Implies heavy reuse of data and perfect prefetching
- Assumes there are no issues with access to RAM
 - Bandwidth is sufficient; latency is hidden by other operations
 - All data are aligned properly, e.g., on 64-byte boundaries
- Assumes clock rate is fixed! (*But a hot chip runs slower*)

Hardware Does More Than Just Flops

- Data Access: Load/Store, Pack/Unpack, Gather/Scatter
- Data Prefetch (i.e., fetch, but don't load into a register)
- Vector Rearrangement: Shuffle, Broadcast, Shift, Convert
- Vector Initialization: Random, Set
- Logic: Compare, AND, OR, etc.
- Math: Arithmetic, Trigonometry, Cryptography, etc., etc.
- Operations may include a Swizzle, Mask, or Implicit Load
 - Swizzle: rearrange vector elements “in flight”
 - Mask: apply operation only to selected vector elements
 - Implicit Load: argument is an address, rather than a register

The AVX-512 Instruction Set(s)

- Adds ~4000 instructions to do math, move data, or both

Extension	SKX-SP	KNL	Functionality
AVX-512F	X	X	<i>Foundation</i> : expands upon AVX to support 512-bit registers; adds masked operations and other new features.
AVX-512CD	X	X	<i>Conflict Detection</i> : permits the vectorization of loops with certain kinds of write conflicts.
AVX-512BW	X		<i>Byte and Word</i> : adds support for vectors comprised of bytes, or of 8- or 16-bit integers; allows masked operations.
AVX-512DQ	X		<i>Doubleword and Quadword</i> : adds instructions for vectors of 32- or 64-bit integers; allows masked operations.
AVX-512VL	X		<i>Vector Length</i> : enables AVX-512 to work with up to 32 of the smaller-size SSE or AVX registers; allows masked operations.
AVX-512PF		X	<i>Prefetch</i> : adds prefetch operations for the gather and scatter functionality introduced in AVX2 and AVX-512.
AVX-512ER		X	<i>Exponential and Reciprocal</i> : includes new operations for 2^x exponentials, reciprocals, and reciprocal square roots.

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
 - For SKL-SP/KNL common subset: `-mavx512f -mavx512cd`
 - GCC 4.9+ has separate options for each AVX-512 extension
 - GCC 5.3+ has `-march=skylake-avx512`
 - GCC 6.1+ has `-march=knl`

Exercise 1

- Copy the code at right and paste it into a local file
- Name it 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

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

void main(int argc, char *argv[]) {
    int i, t;
    double m = 1.5;

    /* Initialize b and c arrays */
    for (i=0; i < ARRAY_SIZE; i++) {
        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. Compile, run, and time the code with no special flags:

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

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

```
gcc -O2 abc.c -o abc
gcc -O3 -fno-tree-vectorize abc.c -o abc
gcc -O3 abc.c -o abc
gcc -O3 -msse3 abc.c -o abc
gcc -O3 -march=native abc.c -o abc
gcc -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 probably 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 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)

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.opttrpt`

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 vectorization enabled, along with “level 1” optimization reporting (`-fopt-info-vec`) from the vectorizer. Confirm that the inner loops in the code 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 “level 2” (loops that missed out on vectorization) to see what else the compiler tried to do with this code (gcc gives way too much information, considering that the main loops ultimately vectorized).

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 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
- Need to consider independence of “unrolled” loop iterations
 - Wider vectors means that more iterations must be independent
- Compiler performs dependency analysis and makes conservative assumptions, unless helped 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:

$$a[1] = a[0] + b[1] \rightarrow a[1] = 0 + 6 \rightarrow a[1] = 6$$

$$a[2] = a[1] + b[2] \rightarrow a[2] = 6 + 7 \rightarrow a[2] = 13$$

$$a[3] = a[2] + b[3] \rightarrow a[3] = 13 + 8 \rightarrow a[3] = 21$$

$$a[4] = a[3] + b[4] \rightarrow a[4] = 21 + 9 \rightarrow a[4] = 30$$

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

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:

$$a[1] = a[0] + b[1] \rightarrow a[1] = 0 + 6 \rightarrow a[1] = 6$$

$$a[2] = a[1] + b[2] \rightarrow a[2] = 6 + 7 \rightarrow a[2] = 13$$

$$a[3] = a[2] + b[3] \rightarrow a[3] = 13 + 8 \rightarrow a[3] = 21$$

$$a[4] = a[3] + b[4] \rightarrow a[4] = 21 + 9 \rightarrow a[4] = 30$$

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

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)

$a = \{0, 6, 8, 10, 12\} \neq \{0, 6, 13, 21, 30\}$ **NOT VECTORIZABLE**

Loop Dependencies: Write After Read

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

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

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

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

Applying each operation sequentially:

$$a[0] = a[1] + b[0] \rightarrow a[0] = 1 + 5 \rightarrow a[0] = 6$$

$$a[1] = a[2] + b[1] \rightarrow a[1] = 2 + 6 \rightarrow a[1] = 8$$

$$a[2] = a[3] + b[2] \rightarrow a[2] = 3 + 7 \rightarrow a[2] = 10$$

$$a[3] = a[4] + b[3] \rightarrow a[3] = 4 + 8 \rightarrow a[3] = 12$$

$a = \{6, 8, 10, 12, 4\}$

Loop Dependencies: Write After Read

Now let's try vector operations:

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

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

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

Applying vector operations, $i=\{0,1,2,3\}$:

$a[i+1] = \{1,2,3,4\}$ (load)

$b[i] = \{5,6,7,8\}$ (load)

$\{1,2,3,4\} + \{5,6,7,8\} = \{6, 8, 10, 12\}$ (operate)

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

$a = \{6, 8, 10, 12, 4\} = \{6, 8, 10, 12, 4\}$ **VECTORIZABLE**

Loop Dependencies: Summary

- 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: Summary

- 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

```
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 this sort of analysis 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 -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;
```

Other Pragmas Affecting Vectorization

- `#pragma vector always`
 - Vectorizes the loop if it is correct to do so (no dependencies)
 - Overrides any decision not to vectorize based upon cost
 - Intel only
- `#pragma omp simd`
 - Part of OpenMP (so that compiler option needs to be set)
 - Vectorizes the loop regardless of cost or safety
 - Same as Intel's non-portable "`#pragma simd`"
- `#pragma novector`
 - Prevents vectorization
 - Intel only

Loop Dependencies: Language Constructs

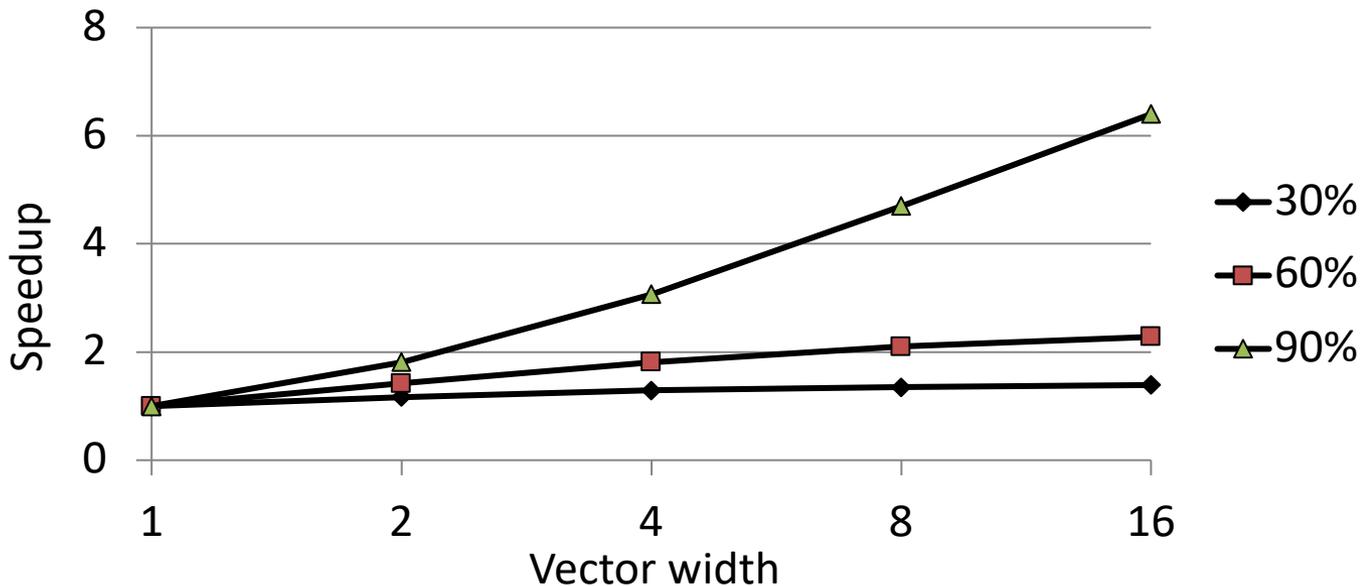
- C99 introduced 'restrict' keyword to language
 - Instructs compiler to assume addresses will not overlap, ever

```
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

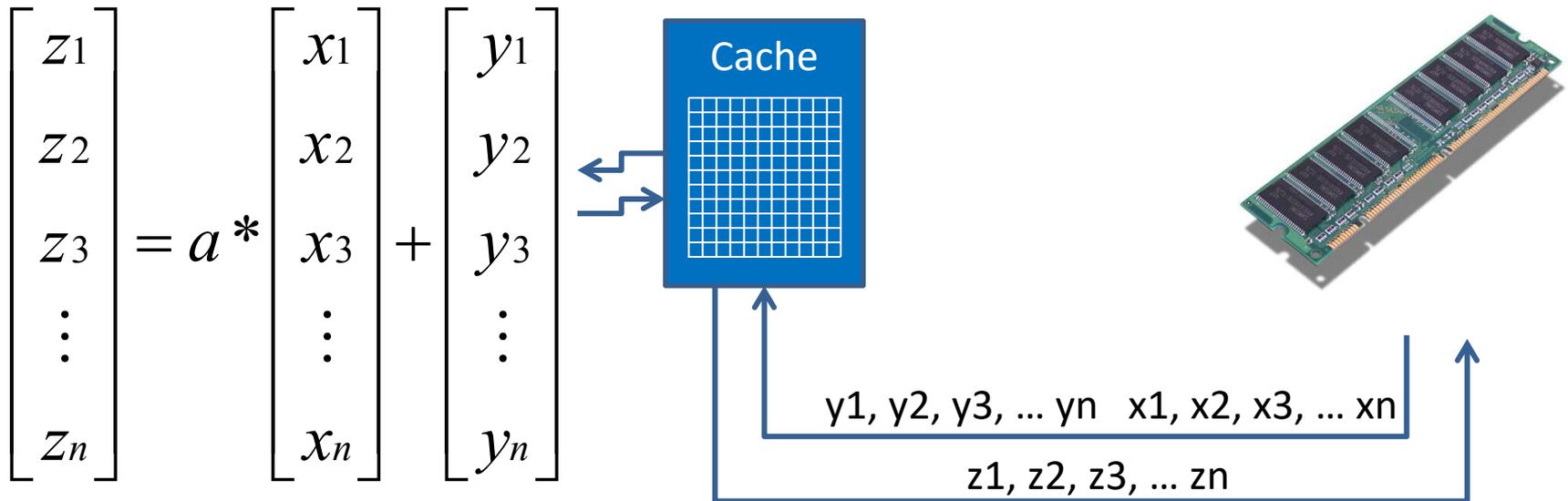
- SIMD is parallel, so Amdahl's law is in effect!
 - Linear speedup is possible only for *perfectly* parallel code
 - Serial/scalar portions of code (or CPU) will limit performance
 - Asymptote of the speedup curve is $1/(\text{unvectorized fraction})$



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
 - Wait... if loads and stores are vectorized along with adds and multiplies, then everything gets the same speedup, right?
 - No. Loads and stores just take the data one short step between L1 cache and registers
- Key fact to know: all data travel between RAM and caches as vectors called “cache lines” – even scalar data
 - This is not really good news. The rate for RAM is slow to begin with, and vectorization won't speed it up further

Cache and Alignment



- 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

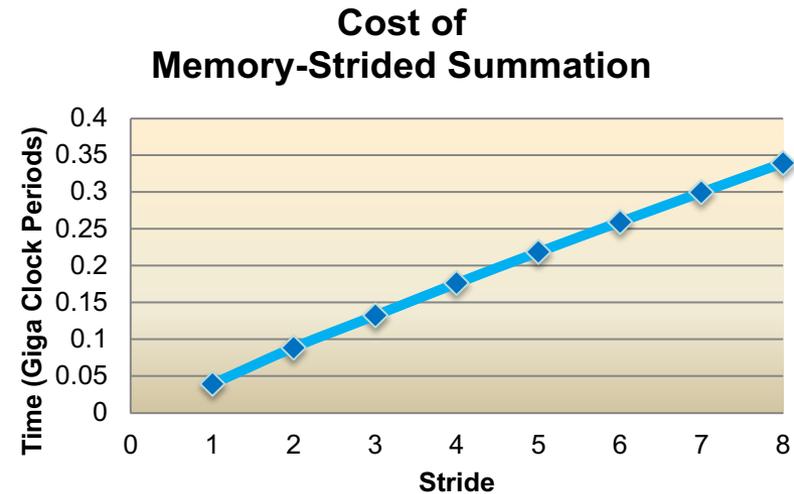
- Fastest usage pattern is “stride 1”: perfectly sequential
 - Cache lines arrive in L1d as full, ready-to-load vectors
- Stride-1 constructs:
 - Storing data in structs of arrays vs. arrays of structs
 - Looping through arrays so their “fast” dimension is innermost
 - C/C++: stride 1 on last index (columns)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

Strided Access

- Striding through memory reduces effective memory bandwidth!
 - Roughly by $1/\text{stride}$
- Why? Data must be “gathered” from 2+ 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;
}
```

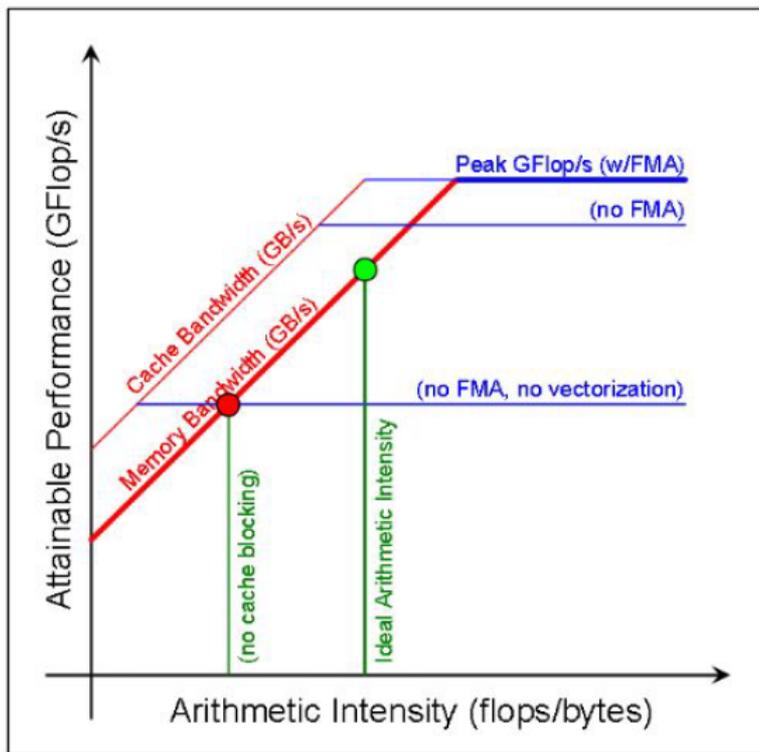
Diagnosing Cache & Memory Deficiencies

- Really bad stride patterns may prevent vectorization
 - One reason for the Intel vector report saying: “vectorization possible but seems inefficient”
- Bad stride and other problems may be difficult to detect
 - Merely result in poorer performance than expected
- Profiling tools like Intel VTune can help
- Intel Advisor makes recommendations based on source

Towards Peak Flop/s: Arithmetic Intensity

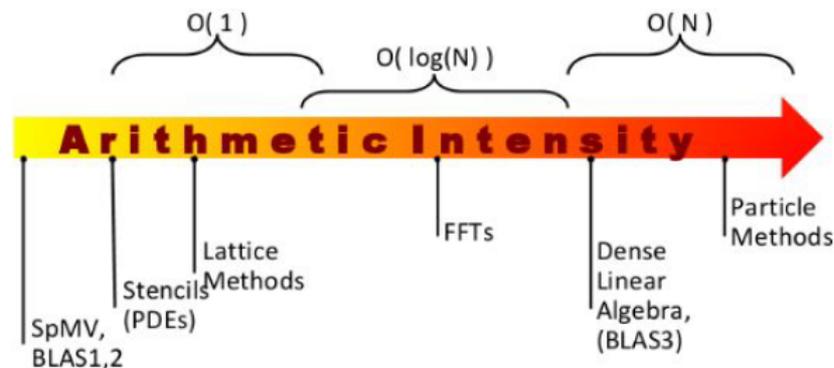
- Even a simple stride-1 loop may not get peak flop/s rate!
- The most common reason is that the *arithmetic intensity* (AI = flop/byte) of the code is too low
 - VPU becomes stalled waiting for loads and stores to complete
 - Delays become longer as the memory request goes further out in the hierarchy from L1 to L2 (to L3?) to RAM
 - Even if the right vectors are in L1 cache, low AI can be an issue due to the limited bandwidth to the registers
- One way to maximize flop/s is to improve AI
- The other way is to reuse the data that are brought into caches and registers (don't have to go as far to reload)

Roofline Analysis



$$\text{Attainable FLOPs/sec} = \min \left\{ \begin{array}{l} \text{Peak FLOPs/sec,} \\ \text{Peak Memory Bandwidth} \times \text{Arithmetic Intensity} \end{array} \right.$$

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



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?

- A typical processor from Intel can do 2 vector flops on every cycle (FMA = 1 add + 1 multiply)
- A typical processor can also do 1 load on every cycle and 1 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*; every stored result must take 2 or more FMAs
 - For the latest server-class chips, all the above rates are doubled, but the 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, 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

(author: Matevž Tadel)

Introduction

- 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!
- Explore how far can we get with (more or less) adiabatic changes to traditional / current 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
 - 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 pos; 3x3 sym cov matrix; label
 - 40 bytes – a bit less than cache line
- Track: 6-vec of pos+mom; 6x6 sym cov matrix
 - Plus 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, 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*TM

- 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
 - 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

Example 1: 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: $c = a + b \rightarrow 1 * n \text{ ops}$
 - sum2_sqr: $c = a*a + 2*a*b + b*b \rightarrow 6 * n \text{ ops}$
 - ...
 - see others in ArrayTest.h / .cxx
 - All return number of floating point operations performed
- **Timing.h / .cxx:** takes a function object and runs it enough times to get total runtime to approximately specified 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 name of the 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 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;
}
```

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++-8
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_03.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)

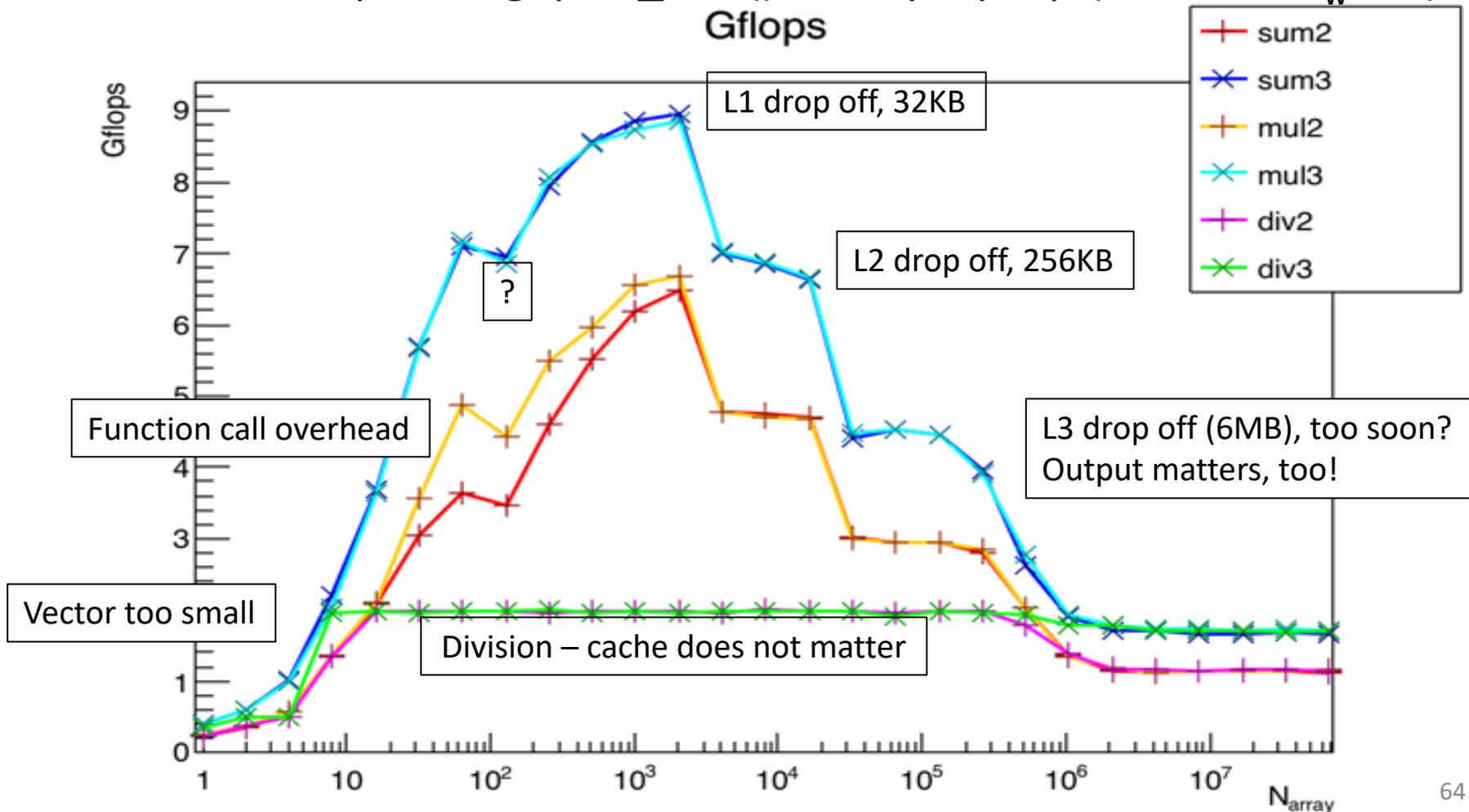
Gflops: giga operations per second – Gops / Time

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

VecUt: vector utilization – OpT / given vector unit width

E.g. Plot, Gflops

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



Your Turn to Plot Some Graphs

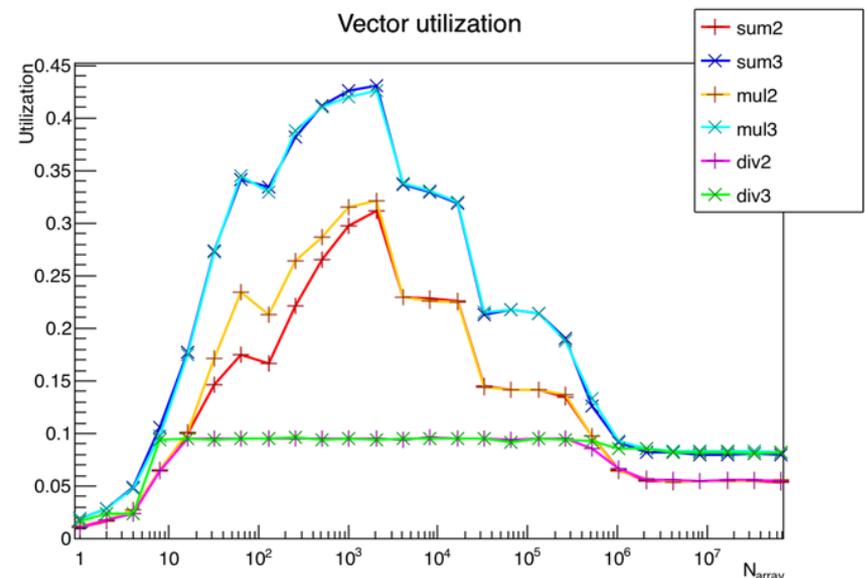
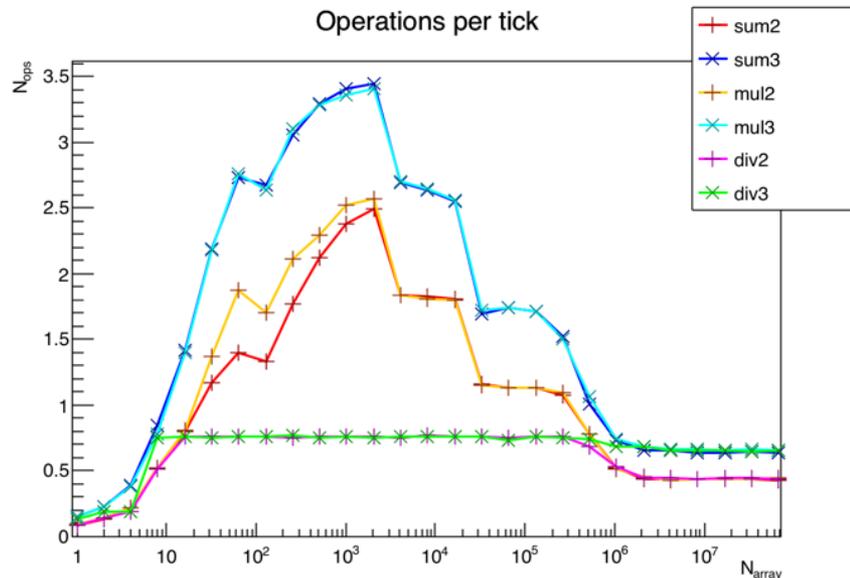
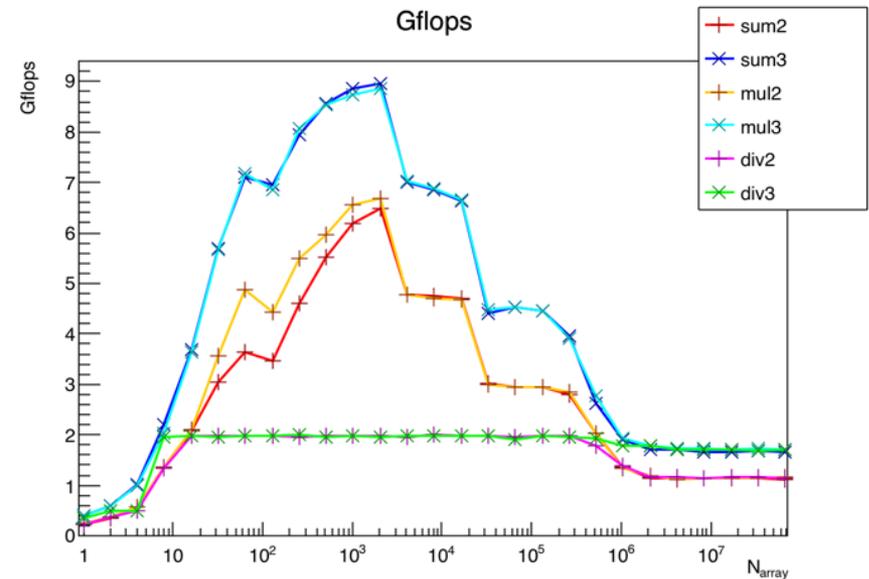
```
# assuming you successfully ran codas.sh
# setup ROOT environment
source <path-to-root/bin>/thisroot.sh

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

# Compare to results for my laptop.
# Poorer? Try setting OPTS:= -O3 -mavx
# Or try setting OPTS := -O3 -march=native
```

Normalization with Clock Speed, Vec Width

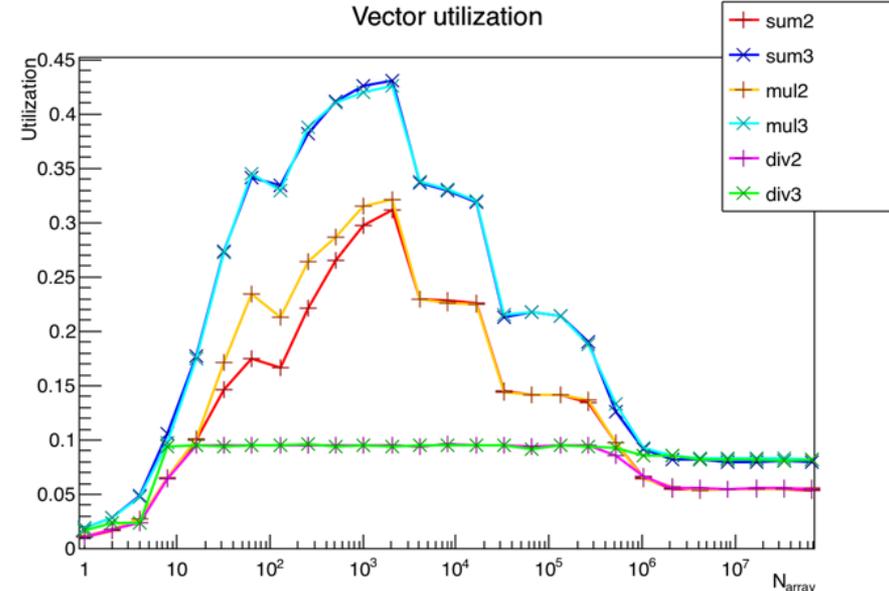
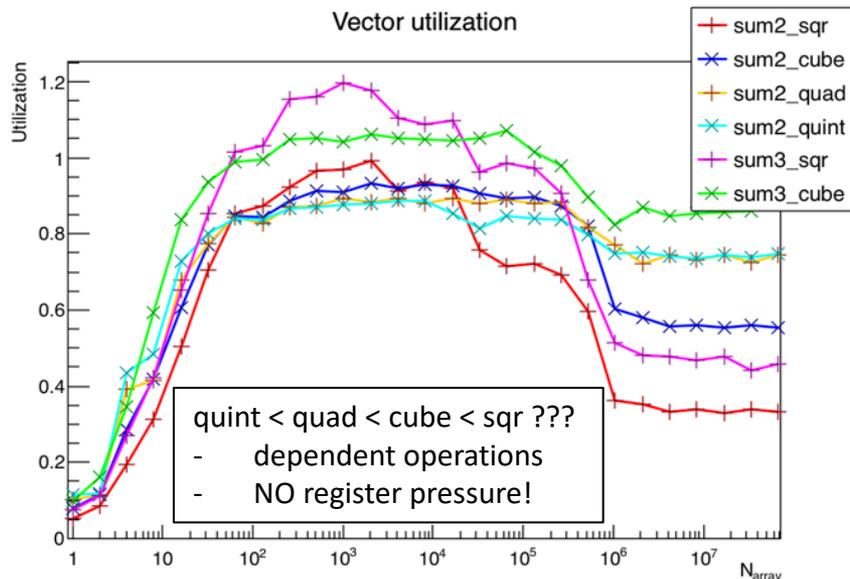
- It's really just normalization.
 - Helps you see different plateaus
 - On next slides will mostly show Vector Utilization
- Why is vector utilization low?
 - B/W limited!



plot_sig() – Significant Computation / Load

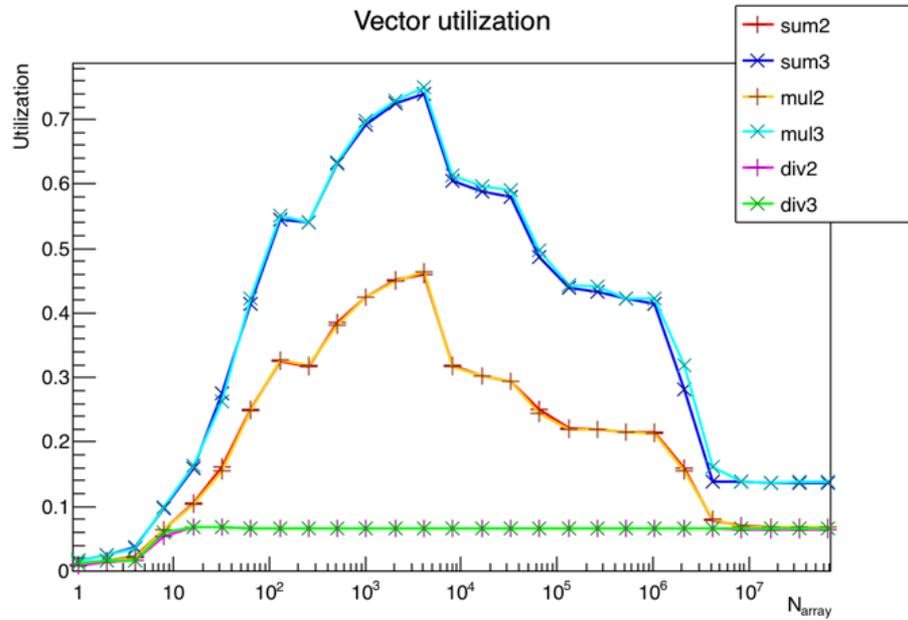
- All cache effects less pronounced.
 - High op tests don't mind L1 at all.
- Vector Utilization > 1?
 - Multiple vector units!
 - Can also be too low clock in Timing.cxx / turbo!
 - Gets even more pronounced on desktop / server CPUs.

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		



Compare Laptop to Desktop CPU

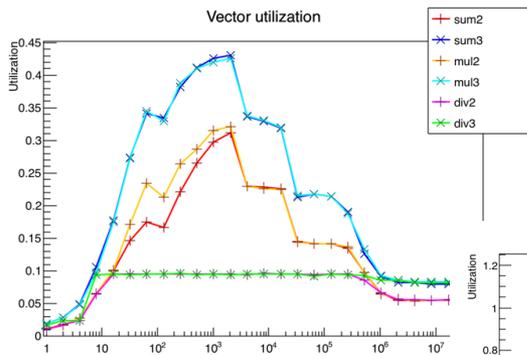
Vector utilization



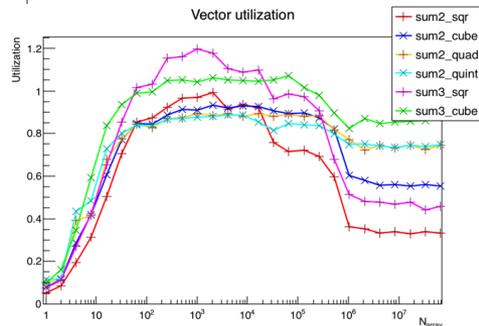
Better, right?

- more vector processing units
- ? deeper pipelines ?
- larger L3 (8 vs. 6 MB)

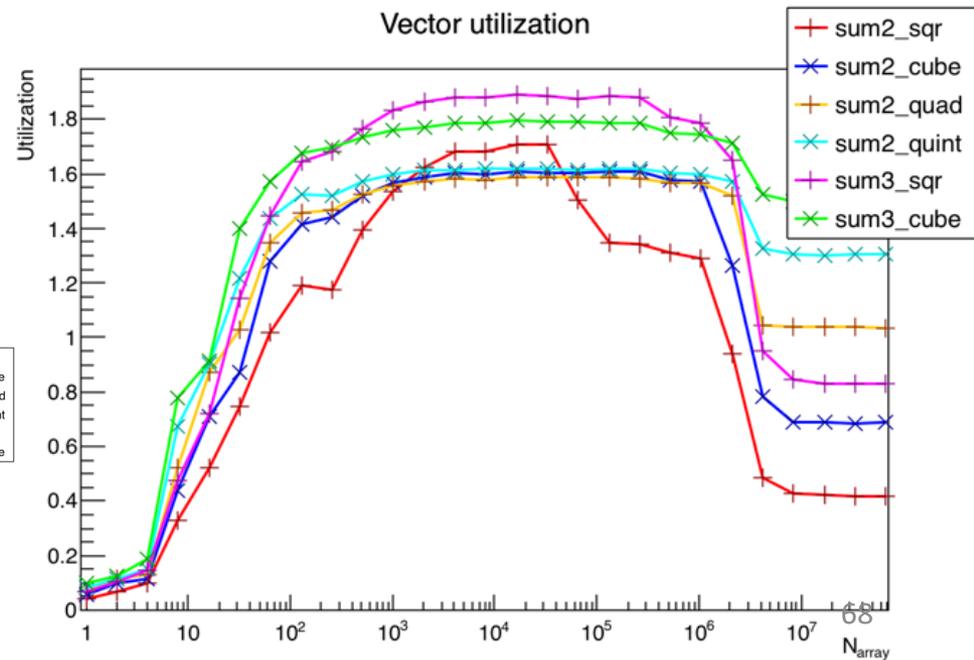
Vector utilization



Vector utilization



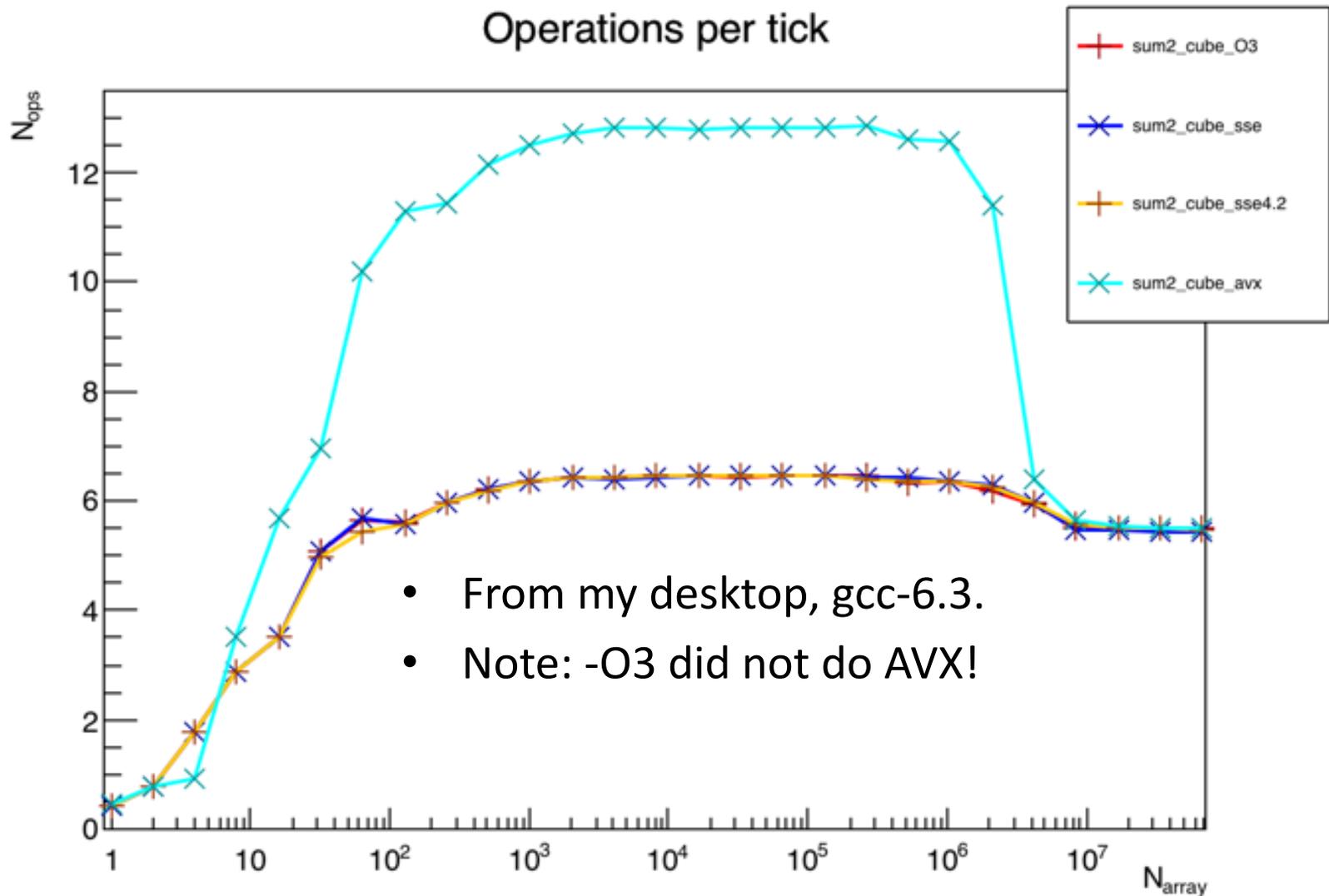
Vector utilization



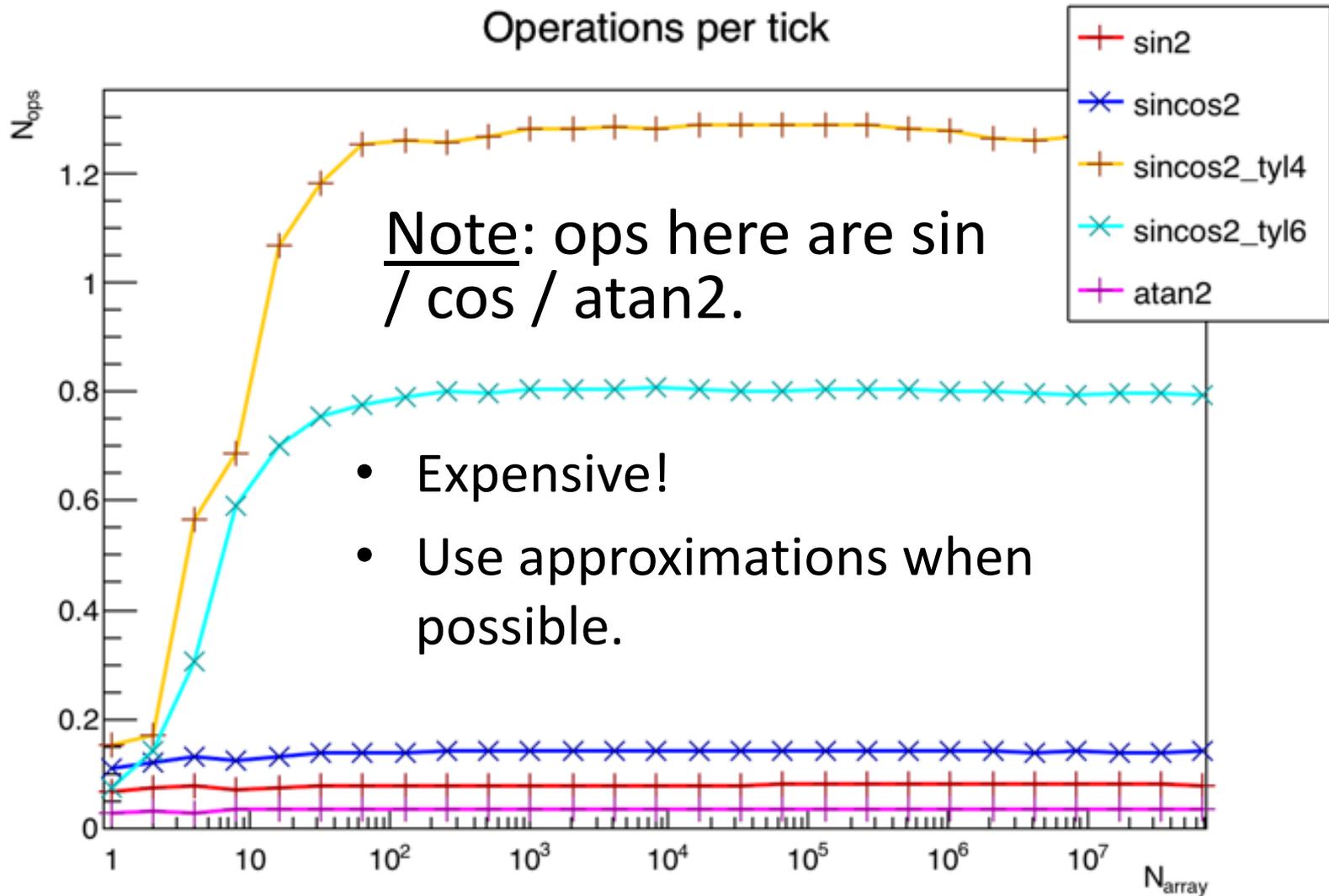
Advanced Usage

- 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



plot_trig() – Trigonometric Functions

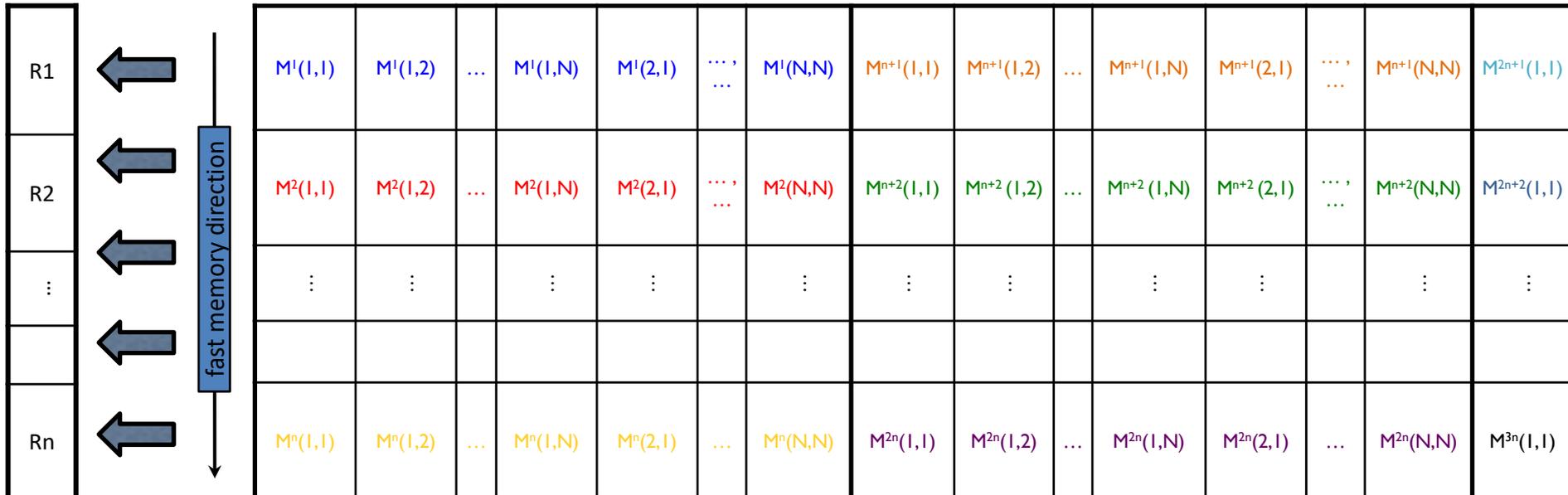


VECTORIZING MATRIX OPERATIONS – MATRIPLEX

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
 - » $6 \times 6 \text{ floats} * 4 \text{ Bytes} * 3 \text{ operands} * 8 = 3456 \text{ Bytes}$
 - » $6 \times 6 \text{ floats} * 4 \text{ Bytes} * 3 \text{ operands} * 16 = 6912 \text{ Bytes}$

Matrplex – The Idea



- “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];
    }
}
```

General Matrixplex Multiplication SIMD

```
template<typename T, idx_t D1, idx_t D2, idx_t D3, idx_t N>
void MultiplyGeneral(const MPlex<T, D1, D2, N>& A,
                    const MPlex<T, D2, D3, N>& B,
                    MPlex<T, D1, D3, N>& C)
{
    for (idx_t i = 0; i < D1; ++i)
    {
        for (idx_t j = 0; j < D3; ++j)
        {
            const idx_t ijo = N * (i * D3 + j);
            for (idx_t n = 0; n < N; ++n) {
                C.fArray[ijo + n] = 0;
            }
            //#pragma omp simd collapse(2)
            for (idx_t k = 0; k < D2; ++k)
            {
                const idx_t iko = N * (i * D2 + k);
                const idx_t kjo = N * (k * D3 + j);
#pragma simd
                for (idx_t n = 0; n < N; ++n)
                {
                    C.fArray[ijo + n] += A.fArray[iko + n] * B.fArray[kjo + n];
                }
            }
        }
    }
}
```

Matrplex – 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)
 - » need for intrinsics was finally fixed a couple of years ago in icc
 - Initial tests were done with icc in 2013/4
 - » loop unrolling brought ~x2 speedup
 - » and intrinsics another x2 (at least on MIC)

E.g. 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);
```

E.g. Generated Code

```
#ifndef 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);
```

```
}
```

```
#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)
```

```
#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)));

    Matriplex() {}
    . . .
};
```

// Covers also vectors with $D2 = 1$ and scalars with $D1 = D2 = 1$.

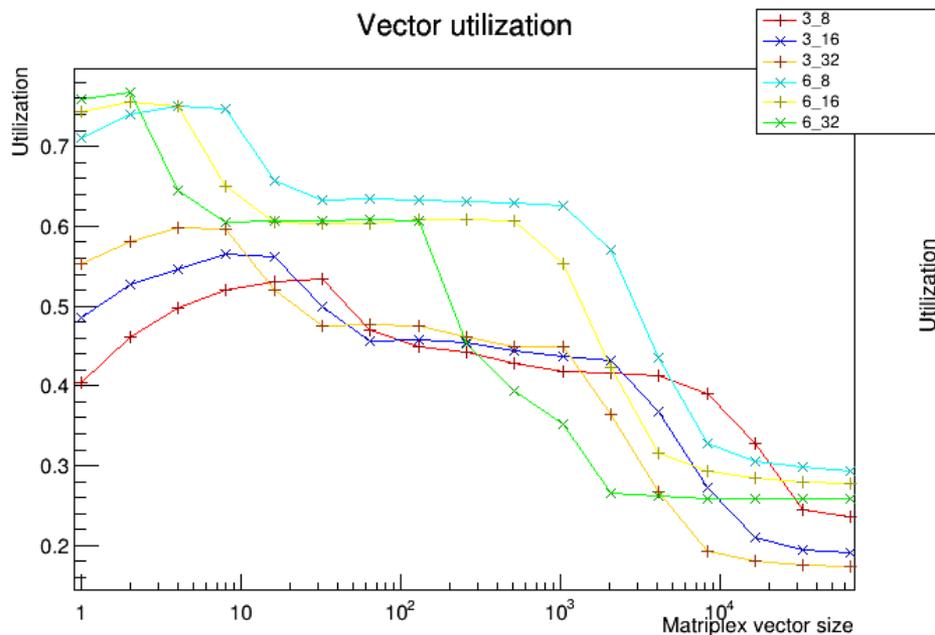
```
template<typename T, idx_t D, idx_t N>
class MatriplexSym
{ enum { kRows = D, kCols = D,
        kSize = (D + 1) * D / 2, kTotSize = N * kSize };

    T fArray[kTotSize] __attribute__((aligned(64)));

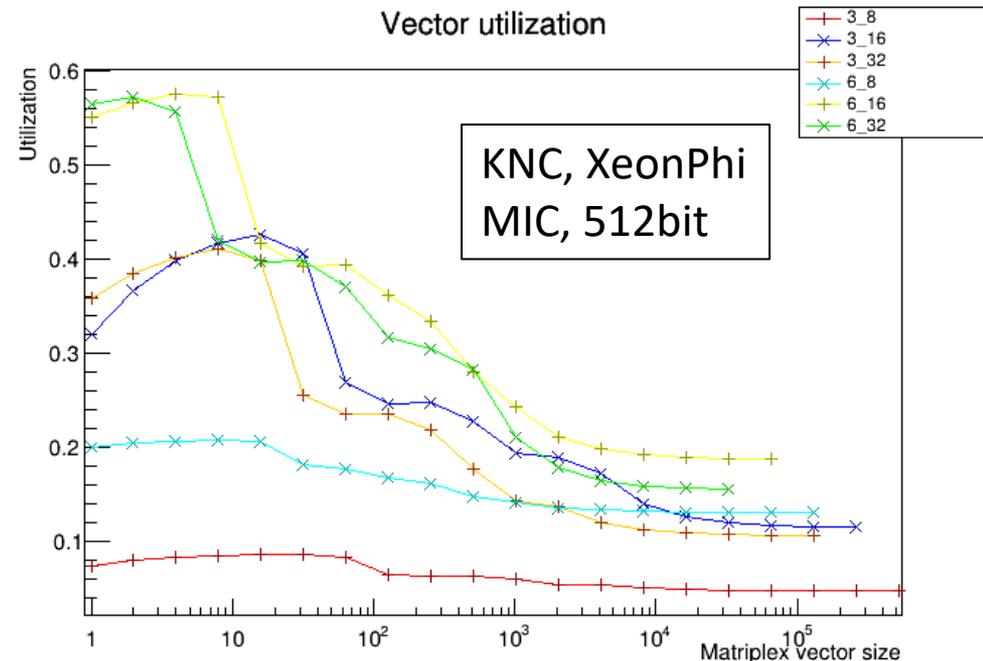
    . . .
};
```

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, XeonPhi
MIC, 512bit

Example 2: t3, MPlexTest

- 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

OUT OF ORDER

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

CPU Grievances

- Chips seem optimized for large PDE problems
 - We have a lot of small objects that we have to combine in different ways ... but we know this a bit in advance!
- Things that will probably not change too much
 - # registers, cache size
- Things that could be more flexible
 - Cache / prefetching control
 - HT / extra execution unit control for repacking
- We have natural allies:
 - Game development & financial industry (& machine learning)

Conclusions: Vectorized Track Finding

- Vectorization can give a significant boost depending on:
 - The ability to express the problem in vector form
 - The problem size & problem complexity
 - The ability to pipe data through cache hierarchy & exec. units
- Understanding performance at the simplest level is vital
- Additional code features can blur vectorization effects:
 - Multi-threading – 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 😊