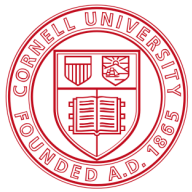
Vector Parallelism for Kalman-Filter-Based Particle Tracking on Multi- and Many-Core Processors

Steve Lantz, Cornell University

Matevž Tadel, UC San Diego

CoDaS-HEP Summer School, July 13, 2017



Cornell University
Center for Advanced Computing

UC San Diego

PART II:

Performance Problems in Vectorized Track Finding

Introduction

- NSF Grant: *"Particle Tracking at High Luminosity on Heterogeneous, Parallel Processor Architectures"*
  - Cornell, Princeton, UCSD → all CMS
  - SLHC, high pile-up environment, 200 interactions per bunch crossing
  - Learn about new architectures and try to use them:
    - MIC / AVX-512 512-bit vector operations, GPUs, ARM-64
  - Explore how far can we get with (more or less) adiabatic changes to traditional / current tracking algorithms:
    - Understood physics performance − efficiencies, fake rates
    - Vectorize, make amenable to multi-threading
    - Invent new data structures and generalized algorithms
      - » Can we make a general tracking software, sacrificing specificity ... and make it run faster!

Complexity of Tracking

- Number of hits grows linearly with N of tracks
  - Considering L layers gives N^L combinatorials
    - Traditional tracking
      - Search new hits on every layer.
      - Limit explosion by limiting total number of candidates considered per seed.
    - Newer techniques
      - tracklets & cellular automata ➛ divide and conquer
      - E.g. using 3 layers for tracklets: N^3 growth of combinatorials
  - In both cases use cuts and other cleverness to limit the search space.
  - What matters at the end is 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
  - The costliest thing is calculating derivates needed for propagation of errors.
  - Can rely on compiler vectorization:

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

- Hit selection
  - This is hard, goes in hand with space partitioning data structures.
  - Will not be covered here.
- Kalman Update
  - Here we deal with the small matrices.
  - Automatic vectorization does not work.
  - The rest of the talk is mostly about this.

Objects in track finding & fitting

- Hit: 3 vec − pos; 3x3 sym cov matrix; label
  - 40 bytes − a bit less than cache line
- Track: 6 vec - pos+mom; 6x6 sym cov matrix
  - + 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 equations yield a set of operations between these objects.
  - Mostly multiplications
    - intermediate results are also 6x3 matrices;
    - product of symmetric matrices is not a symmetric matrix.
  - Similarity operation and 3x3 matrix inversion.

Vectorization – Factors

- Architecture
  - Number, width of vector registers
  - Memory hierarchy, esp. L1 size and latency
- Algorithms
  - HEP algorithms rarely allow automatic compiler vectorization.
  - Restructure code by adding an implicit or explicit additional inner loop in critical sections:
    - » Can be performed by compiler (at −O3 or specific options).
    - » Or can vectorize by hand (next section).
  - – 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 / 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 / COMPILER

How to know we're doing *The Right Thing ™*

- Usually we tune existing code:
  - Run profilers → fix hotspots
  - Measure relative speed-up & declare victory.
  - How much performance was left under 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

- Turture 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 measurement 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 → 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 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 dependecies
  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 osx, modify Makefile, set CXX to your gcc,
# this should be enough: CXX := c++-mp-5
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)

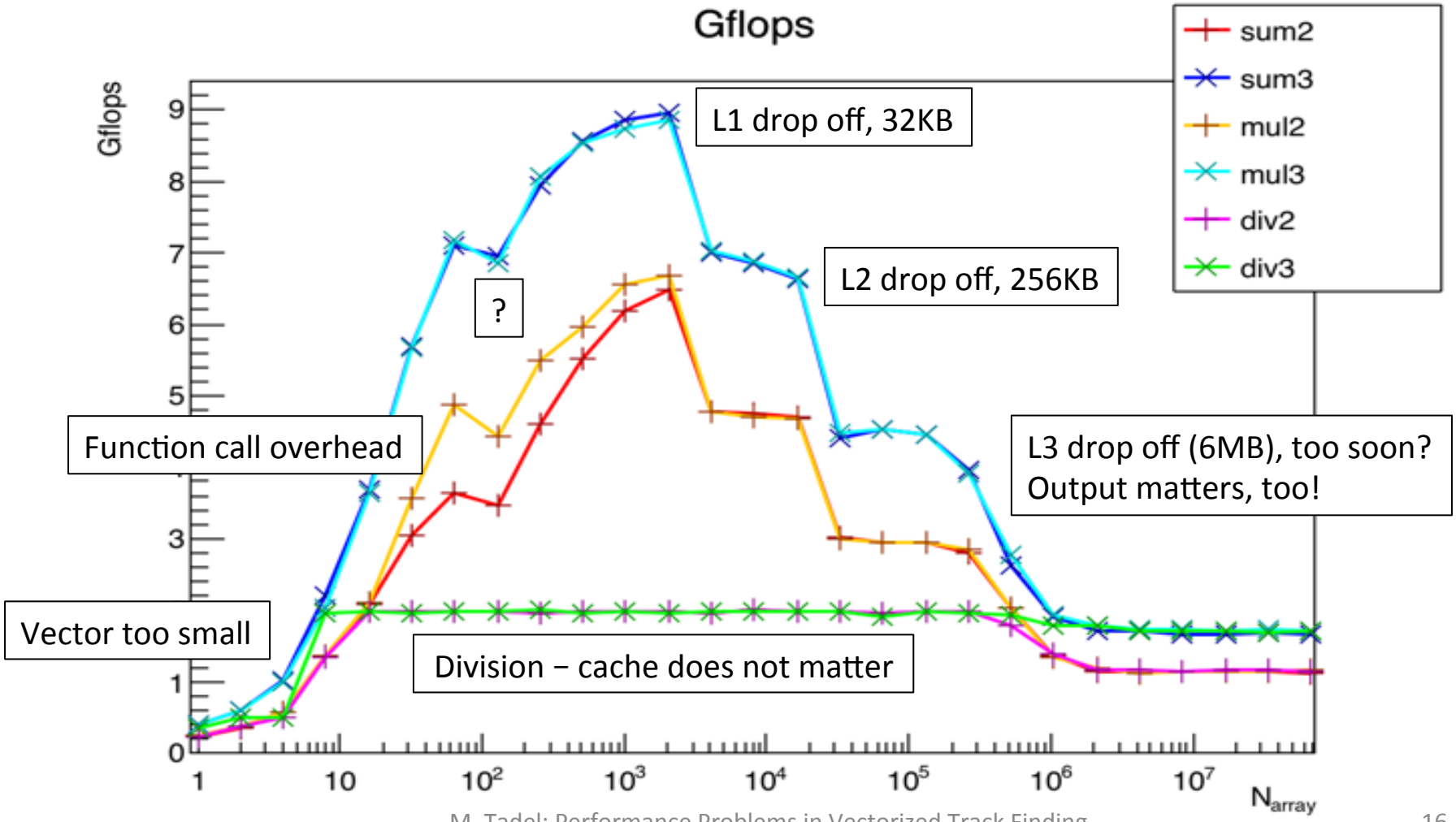
Gflops: giga operations per second – Gops / Time

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

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

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):

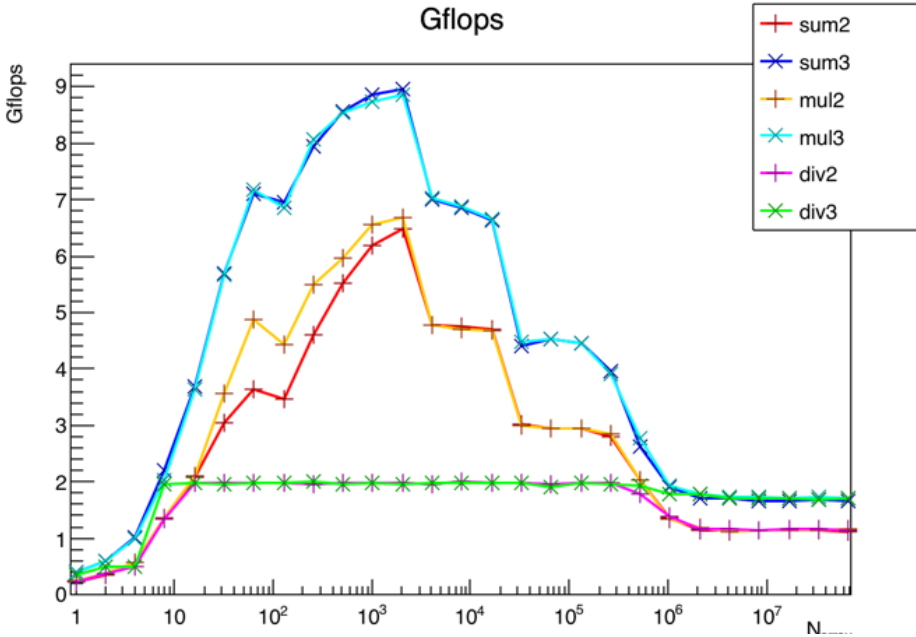


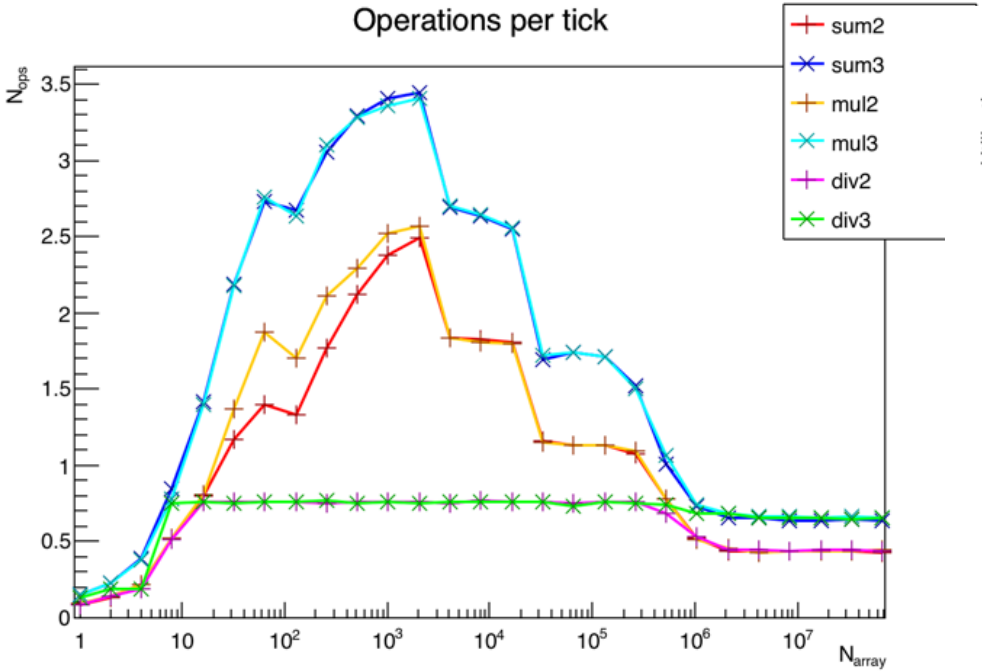
Your turn to plot some graphs

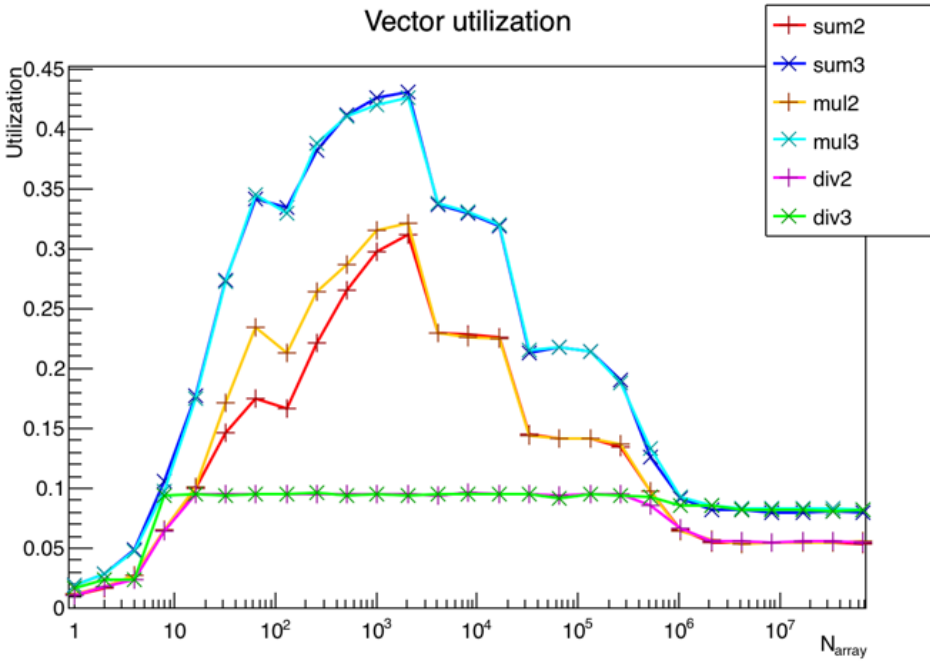
```
# assuming you successfully ran codas.sh
# setup ROOT environment
source <path-to-root/bin>/thisroot.sh

# 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
```

Normalization with clock speed / vec unit 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!



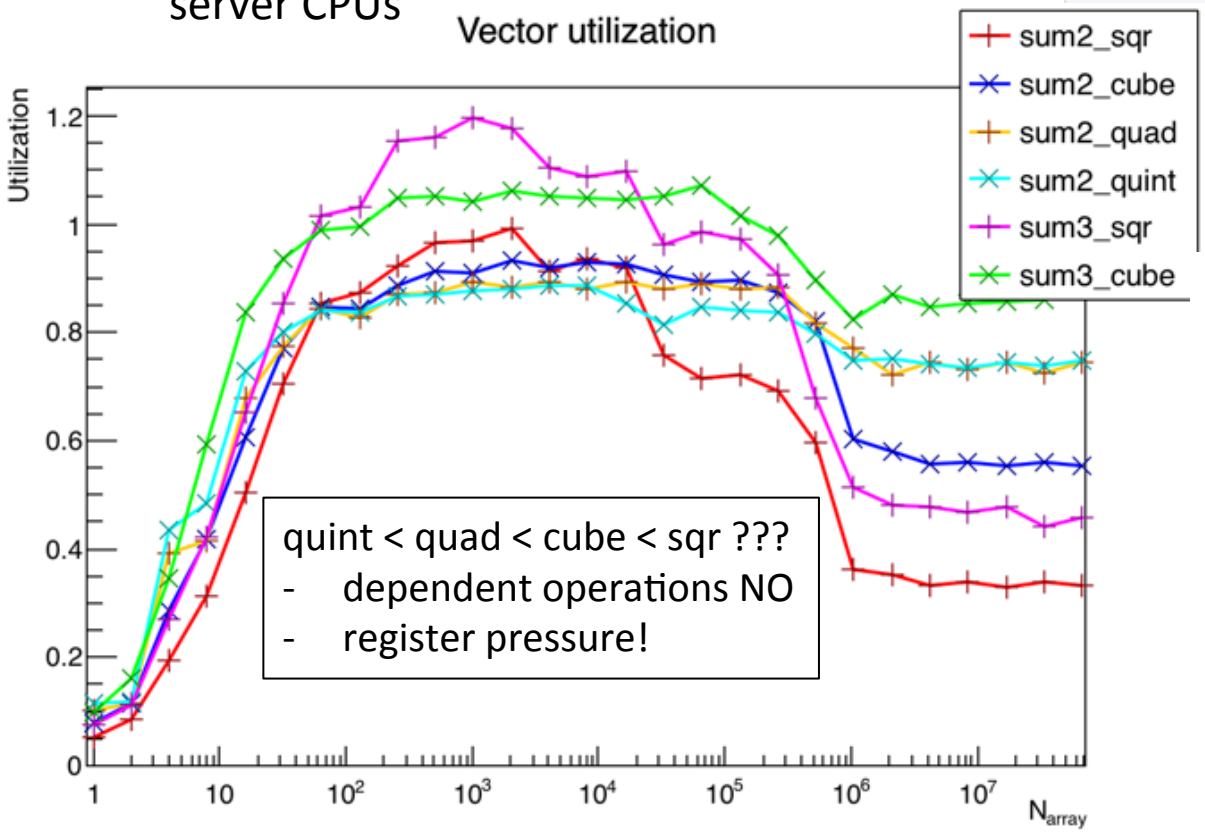


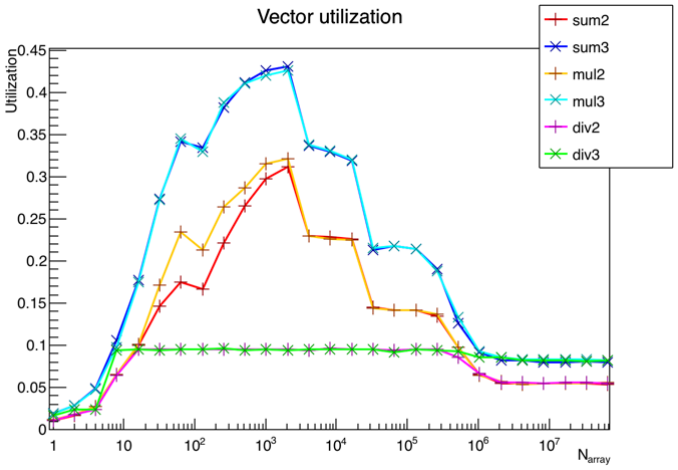


plot_sig() – significant computation per load

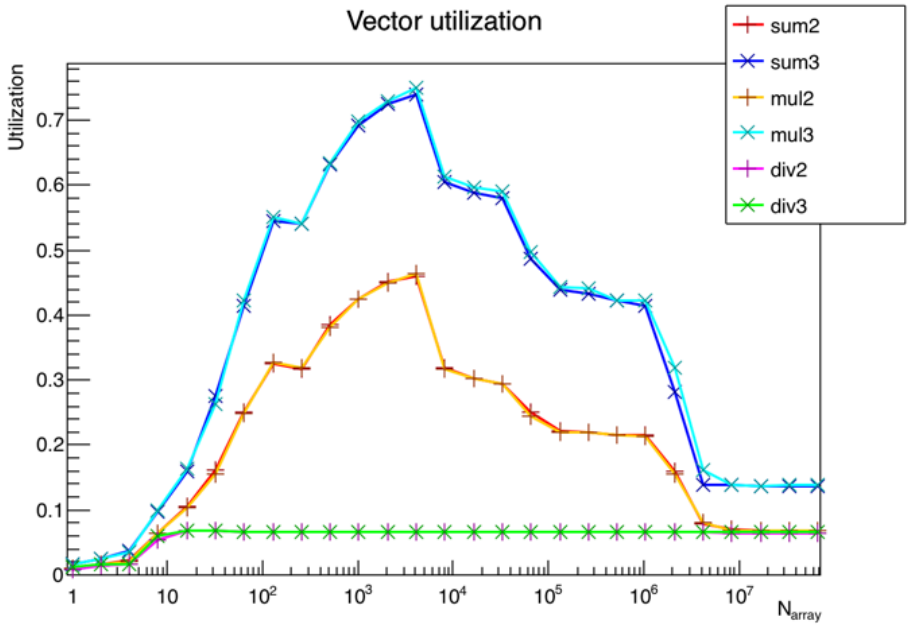
- All cache effects less pronounced.
  - High op tests don't mind L1 at all.
- Vector Utilization > 1?
  - Multiple vector units!
    - Can also be too low clock in Timing.cxx / turbo!
  - Gets even more pronounced on desktop / server CPUs

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		
quint	19		



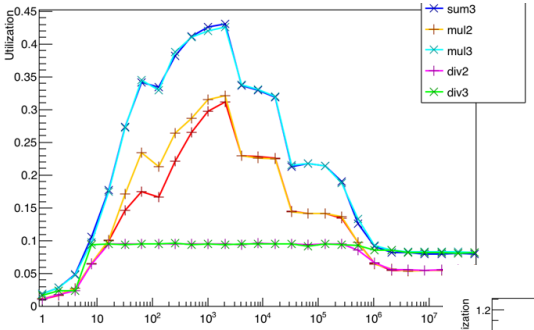


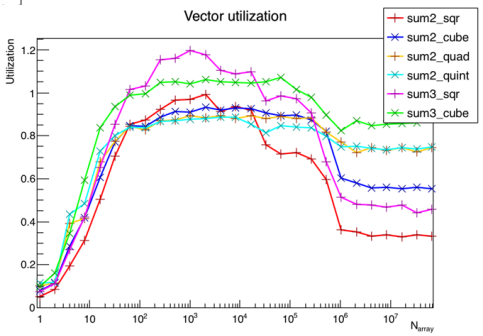
Compare laptop to desktop CPU

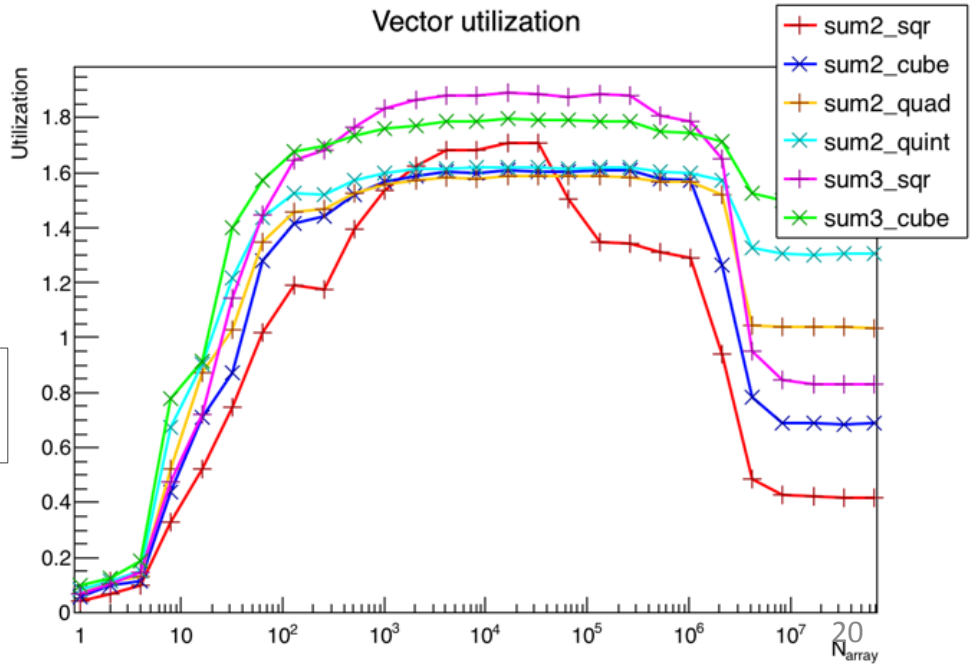


Better, right?

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



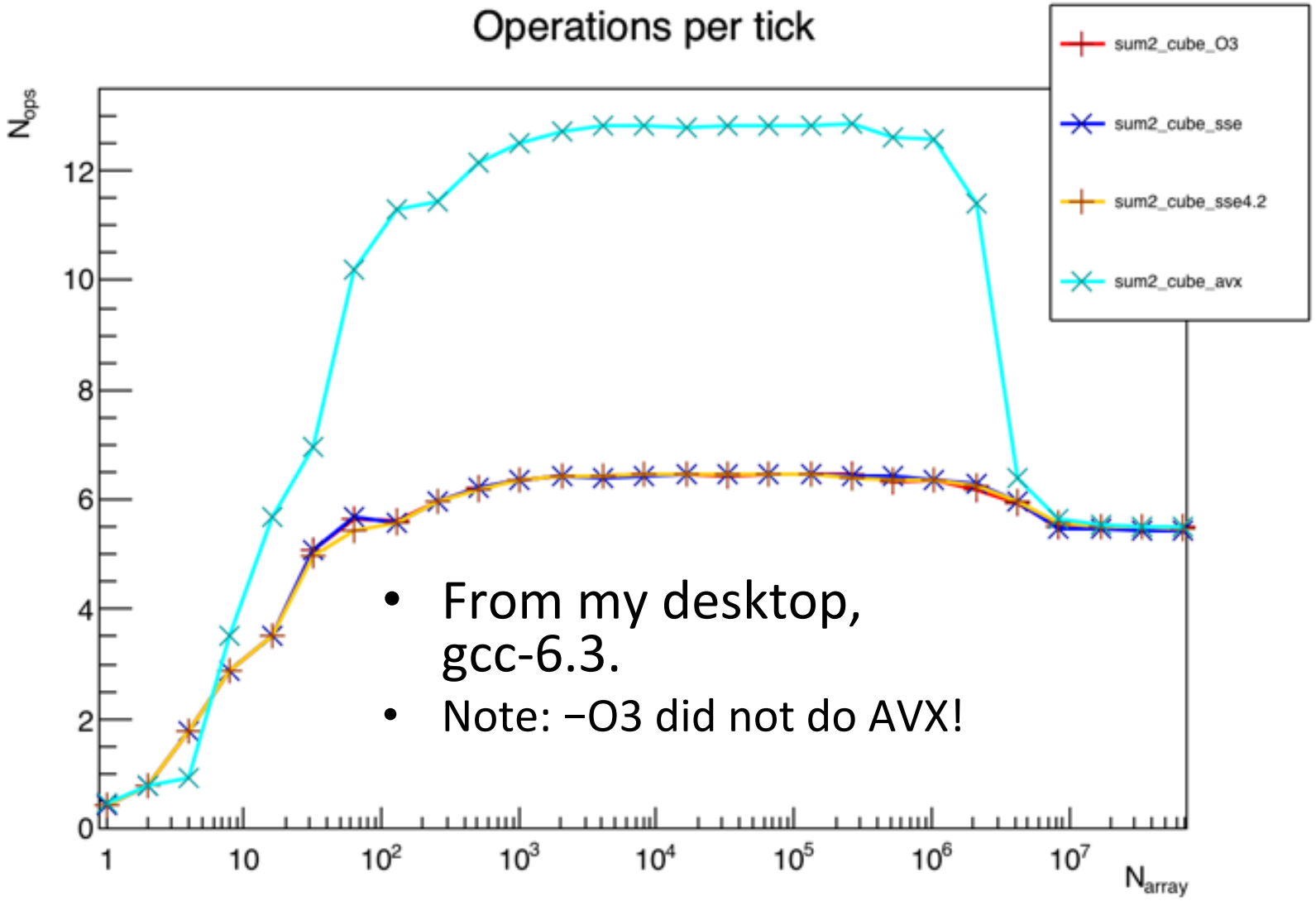




Advanced usage

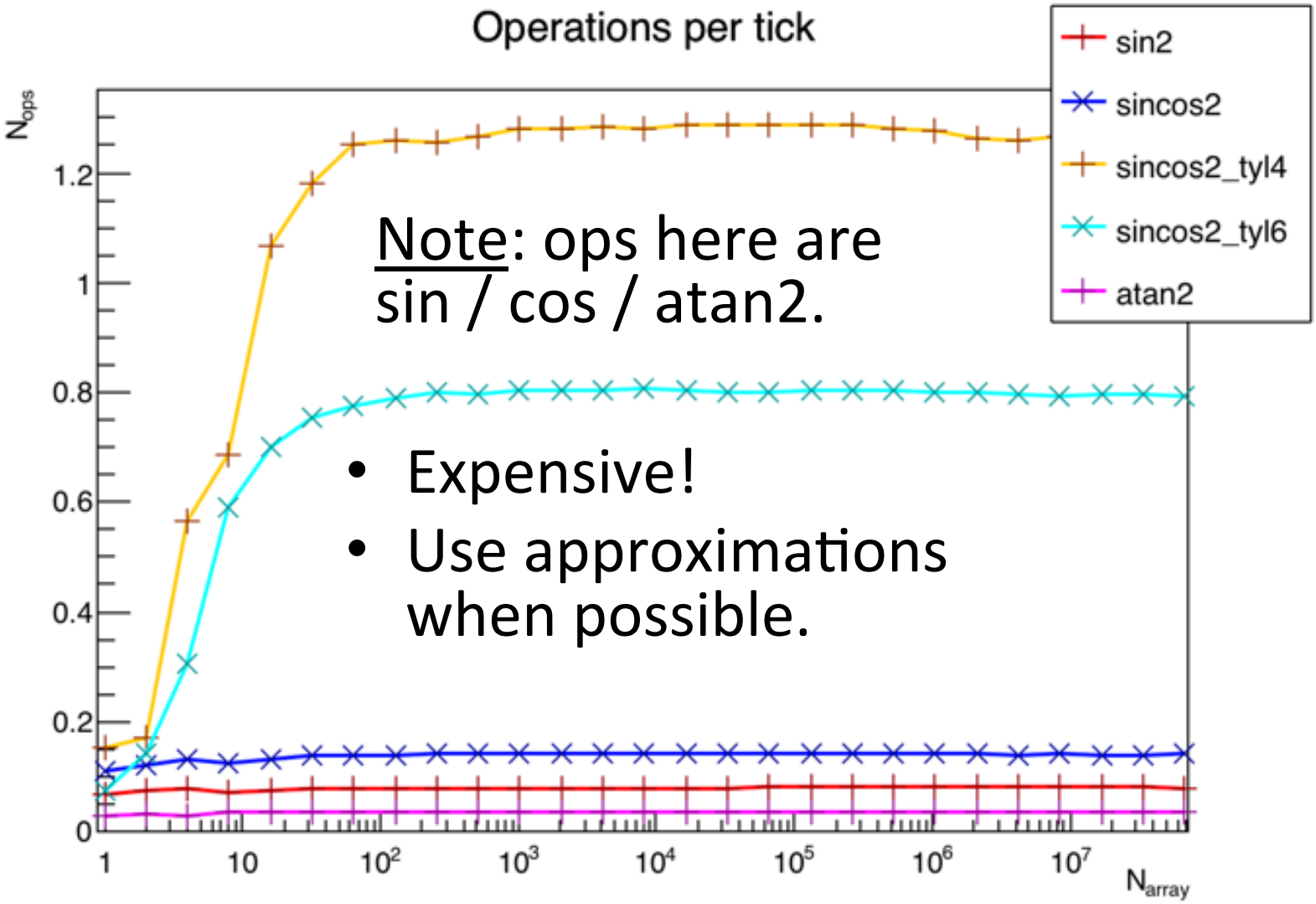
- When changing compiler options or compile time constants the test needs to be recompiled for every invocation.
  - That's what the perl library Test.pm and perl sctipts t1.pl are doing there.
- See codas.sh, parts that are commented out
  - run_vset() − use different vectorization instruction sets:
    - » -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 options



- From my 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

- Trouble with small matrix/vector vectorization
  - Near impossible to do it for individual operations
    - even though there are many multiplications and additions
    - but the pattern / ordering changes all the time so one would have to shuffle / enter the same value into the vectors several times
  - Do it for V_W (8 or 16) matrices in parallel!
    - Matriplex is a library that should help you do it in an optimal fashion.
    - Effectively, #pragma simd over expanded matrix multiplication code:
      - E.g. for 3x3 * 3*3, see next page
      - Compare to Matriplex::MultiplyGeneral(), see page after
    - 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

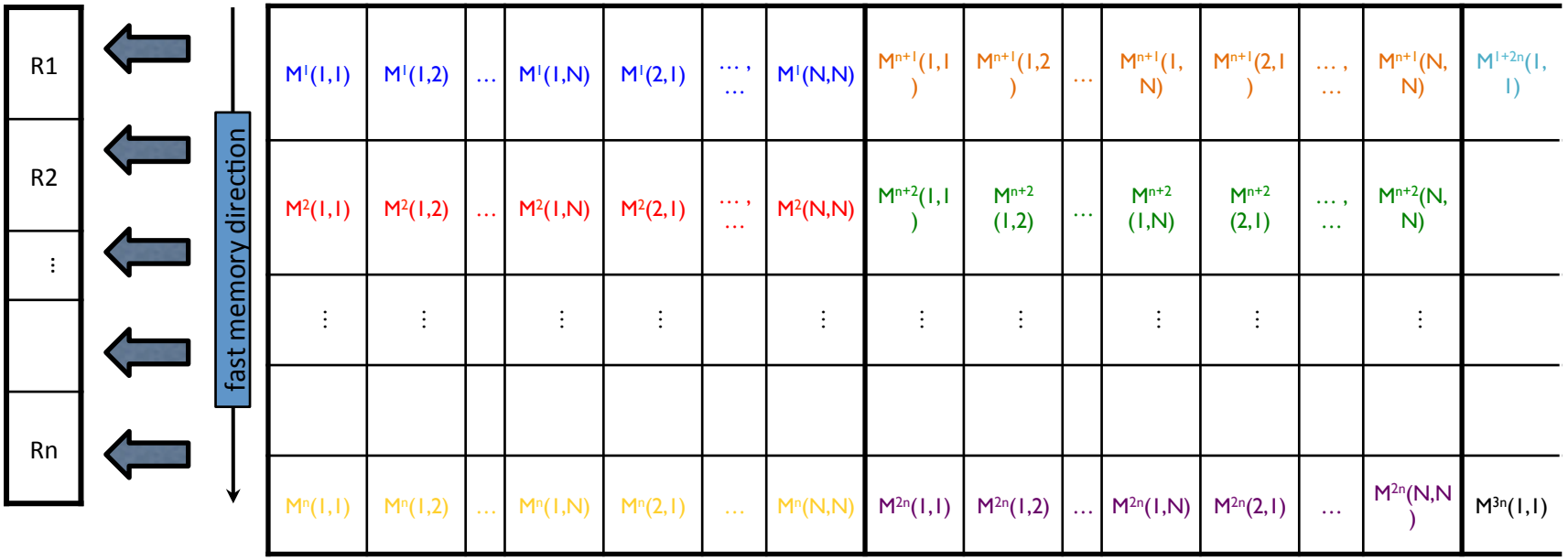
3x3 Matrix SIMD

```
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];
            }
   }}}}
```

Matriplex – the idea



- "Matrix-major" memory representation
  - Operate on a number of 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

Matriplex – features

- Only operations that were needed were implemented (+ some stuff for performance tests)
- GenMul.pm – a PERL module for generation of various matrix multiplications
  - standard and symmetric matrices supported
  - in-code transpose (for similarity transformation)
  - takes advantage of known 0 and 1 elements
  - generates standard C++ (unrolled loops) or intrinsics
    - initial tests were done with icc in 2013/4
      - » loop unrolling brought ~x2 speedup
      - » and intrinsics another x2 (at least on MIC)
    - this has been fixed last year in icc.
    - intrinsics done for MIC, AVX, and AVX512 (no FMA on AVX)
    - takes into account operation latencies when accumulating dot product sums

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

```
#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);
  }
```

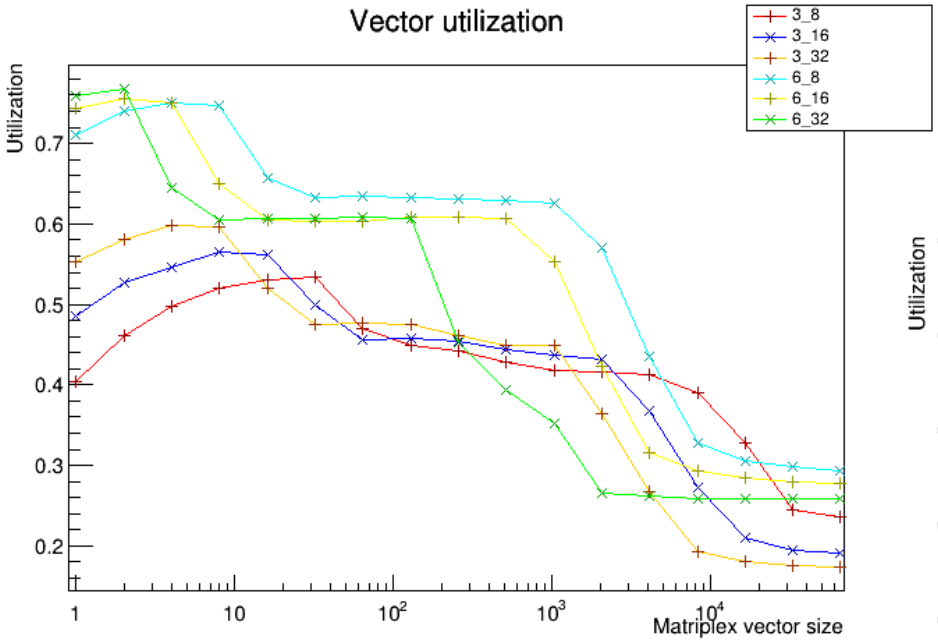
```
#else

#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

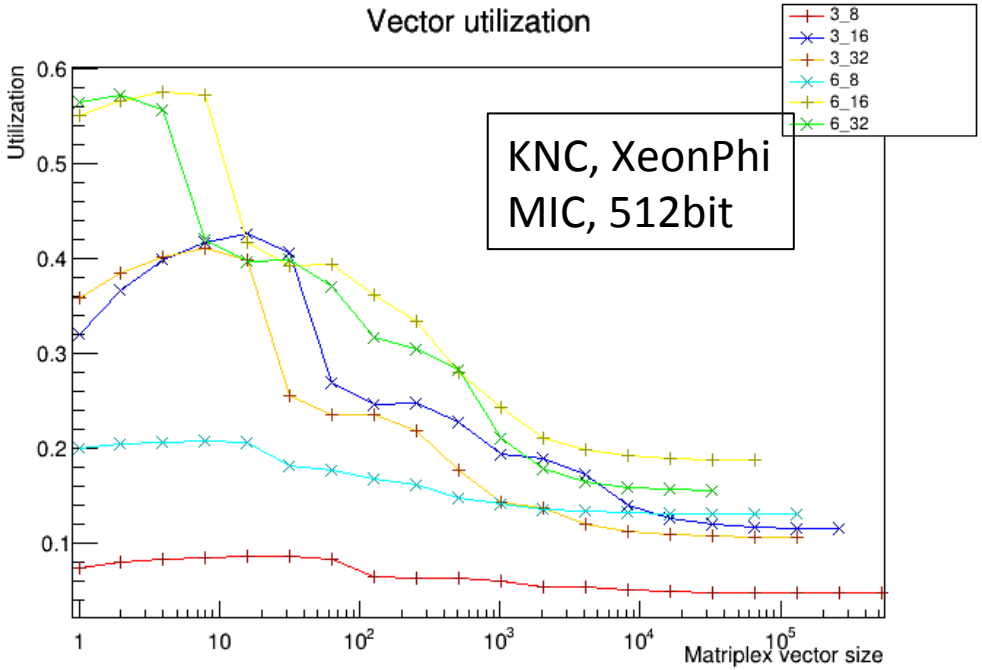
```
template<typename T, idx_t D1, idx_t D2, idx_t N>
class Matriplex
{ enum { kRows = D1, kCols = D2,
         kSize = D1 * D2, kTotSize = N * kSize };

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

   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)



SandyBridge, Xeon AVX



KNC, XeonPhi MIC, 512bit

Example 2: t3, MPlexTest

OUT OF ORDER

- t3.cxx
  - like t1, just uses MPlexTest
- MPlexTest.h/.cxx
  - Note the MPT_DIM / MPT_SIZE defines
    - » Requires recompilation!
  - allocates MatriplexVectors
  - test functions to perform requested operations
- Matriplex/MatriplexVector.h
  - Thought this will be useful … now just a testing construct.
- Matriplex/Matriplex.h & MatriplexSym.h
  - The real thing™
- codas-mplex.sh
  - runs a bunch of tests with 3x3 and 6x6 matrices
- codas.C:
  - functions plot_mplex_3_8(), plot_mplex_6_8()

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.
    - Requires all input matrices to be 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 much less frequently and often only selected parts / matrices are copied out
    - 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:
  - # 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

Conclusion

- Vectorization can bring significant flops boost
- But, it depends on:
  - the ability of expressing the problem in vector form;
  - problem size & problem complexity;
  - the ability to pipe data through the cache hierarchy & exec. units.
- Understanding performance at the simplest level is crucial.
  - Additional code features can blur out vectorization effects:
    - Multi-threading − cache sharing, locking.
    - Swapping between different algorithms or data blocks can lead to cache repopulation / 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 ☺