Parallelized Track Reconstruction for the LHC: the mkFit Project

Steve Lantz, Cornell University

CoDaS-HEP Summer School, July 21, 2023
1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions
Outline

1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions
High Performance Computing in High Energy Physics

Collaborators
K. McDermott, G. Niendorf, M. Reid, D. Riley, P. Wittich (Cornell);
S. Berkman, G. Cerati, P. Gartung, M. Kortelainen (Fermilab);
B. Wang (NVIDIA);
P. Elmer (Princeton);
L. Giannini, S. Krutelyov, M. Masciovecchio, M. Tadel, E. Vourliotis, F. Würthwein, A. Yagil (UCSD);
B. Gravelle, B. Norris (U. Oregon);
A. R. Hall (USNA).

Photo: CMS detector, LHC, CERN

Key reference: S. Lantz et al., *J. Inst.* 15 P09030 (2020)
The **Compact Muon Solenoid** (CMS) is one of the detectors in the LHC (actual photo).

The **Large Hadron Collider** repeatedly smashes beams of protons into each other as they go around a ring 17 miles in circumference at nearly the speed of light.
Collision Energy Becomes Particle Masses: $E=mc^2$
Higgs Discovery @ LHC

Big news on July 4, 2012!

Physicists Find Elusive Particle Seen as Key to Universe

Physicists have announced today that they have seen a clear signal of a particle that looks like the Higgs boson — a key part of the mechanism that gives all particles their masses.
Big Data Challenge

• 40 million collisions a second
• Most are boring
  – Dropped within 3 μs
• 0.5% are interesting
  – Worthy of reconstruction...
• Higgs events: super rare
  – $10^{16}$ collisions $\rightarrow 10^6$ Higgs
  – Maybe 1% of these are found
• Ultimate “needle in a haystack”
• “Big Data” since before it was cool

http://www.hep.ph.ic.ac.uk/~wstirlin/plots/plots.html
CMS: Like a Fast Camera for Identifying Particles

Particles interact differently, so CMS is a detector with different layers to identify the decay remnants of Higgs bosons and other unstable particles.
CMS Is About to Get Busier

Simulation of pile-up = 140 at CMS in r-z plane

- By 2025-2029, the instantaneous luminosity of the LHC will increase by a factor of 2.5, transitioning to the High Luminosity LHC (HL-LHC)
- Significant increase in number of interactions per bunch crossing, i.e., “pile-up”, on the order of 140–200 interactions per event
Reconstruction Will Soon Run Into Trouble

- Higher detector occupancy puts a strain on read-out, selection, and event reconstruction.
- A slow step in reconstruction is combining ~$10^6$ energy deposits (“hits”) in the tracker to form charged-particle trajectories – tracking.
- Tracking is typically the biggest contributor to reconstruction time per event in CMS, and for high pile-up, it diverges.
- We can no longer rely on Moore’s Law scaling of CPU frequency to keep up with growth in reconstruction time – we need a new solution.
- Can we make the tracking algorithm concurrent to gain speed?
Overview of CPU Speed and Complexity Trends

48 Years of Microprocessor Trend Data

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 2010-2019 by K. Rupp

GitHub link
Two Types of Intra-Processor Parallelism

- **Vectorization (data parallelism)**
  - “Lock step” Instruction Level Parallelization: SIMD = Single Instruction, Multiple Data
  - Requires minimization of branching and efficient memory utilization
  - It’s all about finding simultaneous operations, on well-aligned data

- **Multithreading (task parallelism)**
  - OpenMP, Threading Building Blocks, Pthreads, etc., to use multiple cores
  - It’s all about sharing work and balancing the load, with minimal overhead

- To occupy a processor fully, both types need to be identified and addressed
  - Vectorized loops (not the whole code) gain 8x or 16x performance on CPUs
  - Multithreading offers a further Mx speedup on M cores

- Prior tracking algorithms did not do this at the event level—can we? (How?)
Outline

1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions
History of the Trackreco/mkFit Project

• 2015 NSF PIF (Physics at the Information Frontier) 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 (at the time) computer architectures: MIC / AVX-512, GPUs, ARM-64
  – Goal: make tracking software more general and faster!

• Proposal: enhance the parallelism of existing, production tracking algorithms based on Kalman Filter:
  – Keep well-known physics performance – efficiencies, fake rates
  – Make code amenable to vectorization and multithreading, through new data structures and generalized algorithms
Why Kalman Filter for Particle Tracking?

• Naively, each particle’s trajectory is described by a single helix
• Forget it
  – Non-uniform B field
  – Scattering
  – Energy loss
  – ...
• Trajectory is only *locally helical*
• Kalman Filter allows us to take these effects into account, while preserving a locally smooth trajectory
• Goal is to reconstruct the trajectory (track) of each charged particle
• Solenoidal B field bends the trajectory in one plane (“transverse”)
• Trajectory is a helix described by 5 parameters, \( p_T, \eta, \varphi, z_0, d_0 \)
• We are most interested in high-momentum (high-\( p_T \)), low-curvature tracks
• But trajectories may change due to interaction with materials...
• Ultimately we care mainly about:
  – *Initial track parameters*
  – *Exit position to the calorimeters*

• *Kalman Filter is well suited for this job*
Kalman Filter

- Method for obtaining best estimate of the parameters of a trajectory
- For particle tracking: a natural way of including interactions in the material (process noise) and hit position uncertainty (measurement error)
- Used both in pattern recognition (i.e., determining which hits to group together as coming from one particle) and in fitting (i.e., determining the ultimate track parameters)

• Use Kalman procedure to estimate slope and y-intercept of a straight-line fit to noisy data
• Parameter values improve as data points are added
• 30-line script in MATLAB
Track reconstruction has 3 main steps: **seeding, building, and fitting**

- Building and fitting repeat the basic logic unit of the Kalman Filter...
  
  
  - From current *track state* (parameters and uncertainties), track is *propagated* to next layer
  
  - Using hit measurement data, track state is *updated (filtered)*
  
  - Amount of correction is inversely weighted by hit uncertainty
  
  - Procedure is repeated until last layer is reached
Track Fitting as Kalman Filter

• The track fit consists of the simple repetition of the basic logic unit for hits that are already determined to belong to the same track
• Divided into two stages
  – Forward fit: best estimate at collision point
  – Backward smoothing: best estimate at face of calorimeter
• Computationally, the Kalman Filter is a sequence of matrix operations with small matrices (dimension 6 or less)
• But, every single track can be fit in parallel
• Building is harder than fitting!
• After propagating a track candidate to the next layer, hits are searched for within a compatibility window
• Track candidate needs to \textit{branch} in case of multiple compatible hits
  – The algorithm needs to be robust against missing/outlier hits
• Due to branching, track building has typically been the \textit{most time consuming step} in event reconstruction, by far
Parallelization Plan for CPUs

1. Partition the tracks (or track candidates) into SIMD-size bunches
   - Assign bunches to different CPU threads
   - Try to vectorize operations within each bunch

2. Propagate bunches to next detector layer
   - Rely on automatic vectorization by compiler, here
   - Costliest part: computing derivatives for error propagation

3. Select one or more compatible hits in the layer (building only)
   - This is hard! Depends on space-partitioning the data structures containing hits
   - Combinatorial explosion! Need to cap the number of track candidates per seed

4. Perform Kalman updates on track parameters and errors
   - But auto-vectorization doesn’t work well for small matrices... **must focus efforts here**

# multithread this loop...
For b in [ bunches ]
#pragma omp simd
for t in [ track bunch b ]
# ~80 lines of calculations
Outline

1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. **Vectorization of the basic Kalman Filter operations**
4. Tuning Matriplex methods to improve vectorization
5. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions
How Do We Get Vector Speedup?

• Program the key routines in assembly...
  – Ultimate performance potential, but only for the brave
• Program the key routines using SIMD 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

All these were tried!
Objects in Track Finding and Fitting

- **Hit**: 3-vector of position, 3x3 symmetric covariance matrix, label
  - 40 bytes, a bit less than a 64-byte cache line

- **Track**: 6-vector of position and momentum, 6x6 symm. cov. matrix, hit indices
  - Not the most compact representation: helix has 5 parameters, 5x5 symm. cov. matrix
  - But with 6x6, the covariance matrix is block diagonal, one can do sparse matrix tricks
  - Keep just the indices of assigned hits – 256 bytes – 4 cache lines

- **Kalman Filter**: a set of operations using the above objects
  - Mostly multiplications; intermediate results are 6x3 matrices
  - Similarity operations that transform between measurement basis, parameter basis
  - 3x3 matrix inversion
  - Be careful, the product of symmetric matrices is not symmetric
**Matriplex – The Key Idea**

- Nearly impossible to vectorize small matrix/vector ops individually
  - Many multiplications and additions, but pattern of access and operations is inconsistent
- Expand identical operations by doing $V_W$ (8 or 16) matrices simultaneously!
  - **Matriplex** is a library that helps you do it in optimal fashion
  - Effectively, creates $V_W$-way SIMD operations from $V_W$ matrix multiplications
  - Input data are repacked so that loading vector registers is trivial
- But vectorization hardly matters if the data aren’t in cache memory...
  - Best if all matrices are present in L1 data cache together (L1d size: 32-64 kB)
  - Can be done, but puts pressure on both cache and registers
    - $6 \times 6$ floats * 4 Bytes * 3 operands * 8 = 3456 Bytes
    - $6 \times 6$ floats * 4 Bytes * 3 operands * 16 = 6912 Bytes
Matriplex Structure for Kalman Filter Operations

- Store in “matrix-major” order so **16 matrices work in sync (SIMD)**
  - Potential for 60 vector units in Intel Xeon SP to work on 960 tracks at once!
  - Each individual matrix is small: 3x3 or 6x6, and may be symmetric

Matrix size **NxN**, vector unit size **n = 16 for AVX-512 → data parallelism**
template <typename T, idx_t D1, idx_t D2, idx_t N>
class Matriplex { // Covers also vectors with D2 = 1 and scalars with D1 = D2 = 1.
public:
    typedef T value_type;
    static constexpr int kRows = D1;
    static constexpr int kCols = D2;
    static constexpr int kSize = D1 * D2;
    static constexpr int kTotSize = N * kSize;

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

template <typename T, idx_t D, idx_t N>
class MatriplexSym {
public:
    typedef T value_type;
    static constexpr int kRows = D;
    static constexpr int kCols = D;
    static constexpr int kSize = (D + 1) * D / 2;
    static constexpr int kTotSize = N * kSize;

    T fArray[kTotSize] __attribute__((aligned(64)));
};
N-way SIMD with 3x3 Matrices

```c
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 omp 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[ 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[ 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];
    }
}
```

Compiler should convert each line in the loop into a single vector instruction.
What About SIMD Intrinsics?

• Initial versions of the fitting code relied heavily on C++ intrinsic functions
• Improvements in compilers have largely removed the need for them
  – They are still used for packing Matriplexes from input matrices
• Intrinsics for multiplying symmetric matrices are still generated using Perl
  – Vectorization is otherwise tricky because only lower triangular parts are held in memory
  – To account for FMA latencies, elements are not written immediately after computation
  – Macros enable switching among SIMD intrinsics for AVX, AVX2, AVX512
  – The FMA instruction must be emulated for AVX, as it came in with AVX2

```c
#if defined(__AVX512F__)
#define LD(a, i) _mm512_load_ps(&a[i * N + n])
#define ST(a, i, r) _mm512_store_ps(&a[i * N + n], r)
#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)
#endif
```
Outline

1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions
Vector-Aware Coding and Performance Tuning

• 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?
• 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
• Turn to performance tools, if further speedup is desired
• Fit benchmark: average of 10 events, $10^6$ tracks each, single thread
• Matriplex width varies from 1 (quasi-unvectorized) to 16 (full)
• Maximum speedup is only ~4.4x. What’s wrong?
### Clues from Intel Advisor

**General Exploration viewpoint**

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>Clockticks</th>
<th>Instructions Retired</th>
<th>CPI Rate</th>
<th>Start Address</th>
<th>Vectorization Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>helixAtRFromIterative</td>
<td>5,320,000,000</td>
<td>2,240,000,000</td>
<td>2.375</td>
<td>0x4376b0</td>
<td>9.826  25.393</td>
</tr>
<tr>
<td>Matriplex::MatriplexSym&lt; float, (int)6, (int)16&gt;::Subtract</td>
<td>1,330,000,000</td>
<td>630,000,000</td>
<td>2.111</td>
<td>0x40e24a</td>
<td>0.889  0.964</td>
</tr>
<tr>
<td>_int64_t rb_memcpy</td>
<td>840,000,000</td>
<td>490,000,000</td>
<td>1.714</td>
<td>0x48ac40</td>
<td>6.000  7.500</td>
</tr>
<tr>
<td>Matriplex::MatriplexSym&lt; float, (int)3, (int)16&gt;::CopyIn</td>
<td>700,000,000</td>
<td>630,000,000</td>
<td>1.111</td>
<td>0x423b46</td>
<td>0.000  0.000  0.000</td>
</tr>
<tr>
<td>updateParametersMplex</td>
<td>630,000,000</td>
<td>490,000,000</td>
<td>1.286</td>
<td>0x40d550</td>
<td>10.000 5.882</td>
</tr>
<tr>
<td>(anonymous namespace)::MultHelixProp</td>
<td>630,000,000</td>
<td>350,000,000</td>
<td>1.800</td>
<td>0x43de40</td>
<td>7.000 14.737</td>
</tr>
<tr>
<td>Matriplex::MatriplexSym&lt; float, (int)3, (int)1&gt;::CopyIn</td>
<td>560,000,000</td>
<td>140,000,000</td>
<td>4.000</td>
<td>0x423b4c</td>
<td>0.000 0.000 0.000</td>
</tr>
<tr>
<td>(anonymous namespace)::PolarErr</td>
<td>560,000,000</td>
<td>0</td>
<td>0</td>
<td>0x40720</td>
<td>6.500 21.667</td>
</tr>
<tr>
<td>MkFilter::InputTracksAndHits</td>
<td>490,000,000</td>
<td>140,000,000</td>
<td>3.500</td>
<td>0x423830</td>
<td>0.000 0.000 0.000</td>
</tr>
<tr>
<td>Matriplex::MatriplexSym&lt; float, (int)6, (int)16&gt;::CopyIn</td>
<td>420,000,000</td>
<td>490,000,000</td>
<td>0.857</td>
<td>0x4238db</td>
<td>0.000 0.000 0.000</td>
</tr>
<tr>
<td>MkFilter::FilterTracks</td>
<td>420,000,000</td>
<td>70,000,000</td>
<td>6.000</td>
<td>0x424c70</td>
<td>6.667</td>
</tr>
</tbody>
</table>

- Taking lots of time in routines that are unvectorized (or nearly so)
- Ideal vectorization intensity should be 16
- **Subtract** and **CopyIn** appear to be the top offenders
More Clues From Optimization Reports

• Intel compilers have an option to generate vectorization reports
• One report showed a problem in a call to a Matripex method...

remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below...

remark #15346: vector dependence: assumed FLOW dependence between outErr line 183 and outErr line 183

```c
outErr.Subtract(propErr, outErr);
```

• OK! – so outErr (a reference) is both input and output. But we know that is totally safe, because Subtract just runs element-wise through fArray
• Compiler must often make conservative assumptions by default
Fixing the False Loop-Carried Dependence

• Just add a pragma to ignore vector dependence
  – Later this was changed to the even stronger `#pragma omp simd`
• Single change gave ~10% performance gain! (at full vector width)

```c
MatriplexSym& Subtract(const MatriplexSym& a, const MatriplexSym& b)
{
  #pragma ivdep
  for (idx_t i = 0; i < kTotSize; ++i)
  {
    fArray[i] = a.fArray[i] - b.fArray[i];
  }
}
```
CopyIn: Initialization of Matriplex from Track Data

- Load into register: simple vector copy
- Store from register: messy stride-N write?
SlurpIn: Faster, One-Pass Initialization of Matriplex

- Load into register: vector gather op (hardware)
- Store from register: simple vector copy
Getting Data into and out of Matrplexes

• **CopyIn**
  – Take one data array and distribute it into the Matrplex.

• **SlurpIn**
  – Build the Matrplex by taking elements \((i,j)\) of all data arrays.
  – AVX-512 includes 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.
  – CopyOut 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 😊
Retest of Track Fitting in a Simplified Detector

- After fixing Subtract and switching to SlurpIn, test runs 25\% faster at full vector width, maximum speedup goes from \(\sim 4.4\times\) to \(\sim 5.6\times\)
- Amdahl’s Law: can’t get full speedup until *everything* is vectorized
A Quick Word on Amdahl’s Law

• SIMD means parallel, so Amdahl’s Law is in effect!
  – Linear speedup is possible only for \textit{perfectly} parallel code
  – Amdahl’s asymptote of the speedup curve is $1/(\text{serial fraction})$
  – Speedup of 16x is unattainable even if 99\% of work is vector

![Graph showing speedup vs. vector width for different percentages of vector work.](image-url)
1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions
Laptop Vector Utilization as a Function of Array Size

Vector utilization

- L1 drop off, 32KB
- L2 drop off, 256KB
- L3 drop off (6MB), too soon?
- Output matters, too!
- Loop overhead
- Vector too small
- Division – cache does not matter

“mtorture” code by Matevž Tadel, UCSD
HPC Vector Utilization as a Function of Matriplex Array Size

• Square matrix multiplications
• First number is dimension
• Second is Matriplex width

Sandy Bridge, Xeon, AVX

“mtorture” code by Matevž Tadel, UCSD
1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions
Intel Advisor’s Vectorization Report: gcc vs. icc

Vectorization and Code Insights

Vectorization and Code insights perspective lets you identify loops that will benefit most from vector parallelism, discover performance issues preventing from effective vectorization.

**Program Metrics**

<table>
<thead>
<tr>
<th>Program Elapsed Time</th>
<th>10.01s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPU Threads</td>
<td>1</td>
</tr>
<tr>
<td>Vector Instruction Set</td>
<td>AVX2, AVX</td>
</tr>
</tbody>
</table>

**Performance Characteristics**

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>CPU Time</th>
<th>9.99s</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in 1 Vectorized Loop</td>
<td>0.01s</td>
<td>0.1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time in scalar code</td>
<td>9.98s</td>
<td>99.9%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Vectorization Gain/Efficiency (Not Available)**

Consider rebuilding your application with Intel Compiler 16.0 and higher.

---

**Program Metrics**

<table>
<thead>
<tr>
<th>Program Elapsed Time</th>
<th>6.34s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPU Threads</td>
<td>1</td>
</tr>
<tr>
<td>Vector Instruction Set</td>
<td>AVX2, AVX, ...</td>
</tr>
</tbody>
</table>

**Performance Characteristics**

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>CPU Time</th>
<th>6.33s</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in 12 Vectorized Loops</td>
<td>0.47s</td>
<td>7.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time in scalar code including time in 21 unrolled loops</td>
<td>5.86s</td>
<td>92.6%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Vectorization Gain/Efficiency**

- Vectorized Loops Gain/Efficiency: 2.39x
- Program Approximate Gain: 1.06x

---

work with Patrick Gartung, Fermilab

Center for Advanced Computing
Recent Resolution of a Long-Term Mystery!

• The Intel C/C++ Compiler Classic always produced much faster code than GCC
• The reason could be traced to sin/cos functions needed during propagation
  – icc vectorized these from its SVML, enabling vectorization of a larger loop
  – gcc has an equivalent vector math library, libmvec, but it did not come until glibc 2.22
  – Thus, older operating systems such as CentOS 7 did not include libmvec
• The full solution did not arrive until last year…
  – AlmaLinux 8 (and similar CentOS 8 replacements) shipped with libmvec
  – For gcc to link to it, \texttt{-ffast-math} (or at least a subset of it) must also be specified
  – But still, gcc found the propagation loop too complicated to vectorize
  – The main loop had to be broken into many subloops that were obviously vectorizable

work with Patrick Gartung, Fermilab
1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions
Strategy for Track Building with “mkFit”

• Keep the same goal of vectorizing and multithreading all operations
  – Vectorize by continuing to use Matriplex, just as in fitting
  – Multithread by binning tracks in eta (related to angle from axis)
• Add two big complications
  – Hit selection: hit(s) on next layer must be selected from ~10k hits
  – Branching: track candidate must be cloned for >1 selected hit
• Speed up hit selection by binning hits in both eta and phi (azimuth)
  – Faster lookup: compatible hits for a given track are found in a few bins
• Limit branching by putting a cap on the number of candidate tracks
  – Sort the candidate tracks at the completion of each layer
  – Keep only the best candidates; discard excess above the cap
Simplifying the Geometry

- Don’t propagate to one of the tiled, overlapping modules in CMS; instead, SIMD-propagate bunches of tracks to an average $r$ (barrel) or $z$ (disk/endcap)
- Search for nearby hits in a global coordinate space
- Pay one-time, up-front cost (per event) to transform all hits into global coordinates
• Eta binning is natural for both track candidates and hits
  – Tracks don’t curve in eta
• Form overlapping bins of hits, 2x wider than bins of track candidates
  – Track candidates never need to search beyond one extra-wide bin
• Associate threads with distinct eta bins of track candidates
  – Assign 1 thread to j bins of track candidates, or vice versa (j can be 1)
  – Threads work entirely independently $\rightarrow$ task parallelism
Profiling showed the busiest functions were memory operations!

Cloning of candidates and loading of hits were major bottlenecks

- This was alleviated by reducing sizes of Track by 20%, Hit by 40%
- Track now references Hits by index, instead of carrying full copies
Related Scaling Problems?

- Test parallelization by assigning threads to 21 eta bins
  - For $n_{\text{EtaBin}}/n_{\text{Threads}} = j > 1$, assign $j$ eta bins to each thread
  - For $n_{\text{Threads}}/n_{\text{EtaBin}} = j > 1$, assign $j$ threads to each eta bin
- Observe poor scaling and saturation of speedup
Amdahl’s Law Again

• Possible explanation: some fraction $B$ of work is a serial bottleneck
• If so, the minimum time for $n$ threads is set by Amdahl’s Law:

$$T(n) = T(1) \left[ (1-B)/n + B \right]$$

parallelizable... not!

• Note, asymptote as $n \to \infty$ is not zero, but $T(1)B$
• Idea: plot the scaling data to see if it fits the above functional form
  – If it does, start looking for the source of $B$
  – **Progressively exclude any code not in an OpenMP parallel section**
  – Trivial-looking code may actually be a serial bottleneck...
Busted!

• Huge improvement from excluding *one code line* creating eta bins
  
  `EventOfCombCandidates event_of_comb_cands;
  // constructor triggers a new `std::vector<EtaBinOfCandidates>`

• Accounts for 0.145s of serial time (0.155s)... scaling is still not ideal
Intel VTune Shows Another Issue

• VTune reveals non-uniformity of occupancy within OpenMP threads
  – Some threads take far longer than others: *load imbalance*
  – Worsens as threads increase: test below uses 42 threads on Xeon Phi

• Need dynamic reallocation of thread resources, e.g., task queues
Improvement with Intel Threading Building Blocks

- TBB allows eta bins to be processed by varying numbers of threads
- Allows idle threads to steal work from busy ones
- Much better load balance
Summary: Building Tracks in Parallel with mkFit

- Nested levels of parallel tasks for track building:
  1. Loop over different events;
  2. Loop over different \( \eta \)-regions;
  3. Loop over \( z-/r- \) and \( \varphi \)-sorted groups of seeds.

- Parallel tasks scheduled through Intel TBB
  - Dynamic task stealing to balance workloads

- Basic parallel task includes simplified two-step propagation
  - Propagate to average \( r \) or \( z \) of detector layer, compute compatibility window
  - Propagate to each hit in window, select which hit(s) to add to track based on \( \chi^2 \)
  - Kalman calculations include multiple scattering and energy loss in detector layer
Outline

1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matirplex methods to improve vectorization
5. Checking the cache performance of Matirplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions
**mkFit Code Performance**

- Estimates of parallelization based on Amdahl’s Law
  - ~70% vectorized
  - 95%+ multithreaded
- Up to 6.7x faster building time where mkFit is used
  - Reduction of 25% in total tracking time
  - Event throughput increase of 10–15% in LHC Run 3

CMS is now using mkFit by default for computing most tracks

---

“KNL” — 64 cores: Intel Xeon Phi 7210 @ 1.30 GHz
“SKL-SP” — 2-socket x 16 cores: Intel Xeon Gold 6130 @ 2.10 GHz
Future Directions

• Extend the mkFit paradigm to more applications
  – Example: extend to more complex track building steps for further speed-up

• Apply to track fitting
  – Time for fitting is now comparable to track building

• Build tracks for the High Level Trigger
  – The HLT computes on the raw data in real time and decides which events to keep

• Modify for CMS Phase-2 geometry and configuration
  – Optimize and tune for the new detector
  – Look for synergies with other algorithms