SIMD studies in the LHCb reconstruction software

> DANIEL HUGO CÁMPORA PÉREZ
BEN COUTURIER
On the menu

Vectorisation is good

Get creative
Vectorisation is a hot topic

- Tuning algorithms in an orthogonal dimension

- It is not new but it is evolving, and it is going to stay
  - SSE (1999), SSE2 (2001)
  - AVX512 (2013) – 512-bit registers

- Time to expand our toolbox!
Can we do this?

- Code dependencies to architecture
  - SSE2 is safe in x86_64, templated versions to other architectures
- Auto-vectorisation doesn’t always do the job
  - Reordering code is sometimes worth trying, and quite cheap
- Our CPUs have more to offer, this is a free meal
- Code readability doesn’t need to suffer
  - Good practices still exist

```c
// const float det = (z1*z1)*z2 + z1*(z3*z3) + (z2*z2)*z3 - z2*(z3*z3) - z1*(z2*z2) - z3*(z1*z1);
// const float det1 = (x1)*z2 + z1*(x3) + (x2)*z3 - z2*(x3) - z1*(x2) - z3*(x1);
// const float det2 = (z1*z1)*x2 + x1*(z3*z3) + (z2*z2)*x3 - x2*(z3*z3) - x1*(z2*z2) - x3*(z1*z1);
// const float det3 = (z1*z1)*z2*x3 + z1*(z3*z3)*x2 + (z2*z2)*z3*x1 - z2*(z3*z3)*x1 - z1*(z2*z2)*x3 - z3*(z1*z1)*x2;
const float det = ((z1*z1)*(z2-z3))+ ((z2*z2)*(z3-z1)) + ((z3*z3)*(z1-z2));
const float det1 = (x1)*(z2-z3) + (x2)*(z3-z1) + (x3)*(z1-z2);
const float det2 = (z1*z1)*(x2-x3) + (z2*z2)*(x3-x1) + (z3*z3)*(x1-x2);
const float det3 = (z1*z1)*(z2*x3 -z3*x2) + (z2*z2)*(z3*x1 -z1*x3) + (z3*z3)*(z1*x2 -z2*x1);
```

TrackFit solveParabola
gcc 4.8 reports a 1.18x speedup
Intel Xeon E5-2670 v2 @ 2.50GHz
Horizontal vectorisation

- Parallelisable loops

```c++
// for (int i = 0; i < ArraySize; ++i) {
// const float x = input[i].x;
// const float y = input[i].y;
// output[i].r = std::sqrt(x * x + y * y);
// output[i].phi = std::atan2(y, x) * 57.295780181884765625f; // 180/pi
// if (output[i].phi < 0.f) {
// output[i].phi += 360.f;
// }
// }

for (size_t i = 0; i < x_mem.vectorsCount(); ++i) {
    const float_v x = x_mem.vector(i);
    const float_v y = y_mem.vector(i);
    r_mem.vector(i) = Vc::sqrt(x * x + y * y);
    float_v phi = Vc::atan2(y, x) * 57.295780181884765625f;
    phi(phi < 0.f) += 360.f;
    phi_mem.vector(i) = phi;
}
```

- Scalable to future architectures
- Data required to be an SoA

Example from M. Kretz vc library
Vertical vectorisation

- Portion of code with many EX operations, few loads and stores

```c
// tx = (sxz * s0 - sx * sz) / den;
// ty = (uyz * u0 - uy * uz) / den2;
// nx0 = (sx * sz2 - sxz * sz) / den;
// ny0 = (uy * uz2 - uyz * uz) / den2;

v1 = _mm_shuffle_ps(v_sxz, v_sx, _MM_SHUFFLE(2,0,2,0));
v2 = _mm_shuffle_ps(v_s0, v_sxz, _MM_SHUFFLE(3,1,2,0));
v3 = _mm_mul_ps(v1, v2);
v4 = _mm_shuffle_ps(v_sx, v_sxz, _MM_SHUFFLE(2,0,2,0));
v5 = _mm_shuffle_ps(v_sx, v_sx, _MM_SHUFFLE(3,1,3,1));
v6 = _mm_mul_ps(v4, v5);
v1 = _mm_sub_ps(v3, v6);
v2 = _mm_div_ps(v1, v7);
```

Intel Intrinsics – PrPixel fit

- Vector width is fixed – Need to think in vector width when writing code
- Not scalable – Bigger registers won’t make your code faster
- Data structure is not free either – Requires an AoS
Our weapons of choice

higher level

vc
Fog’s Vectorclass

gcc intrinsics
Intel intrinsics
ASM

lower level

vc – Vector Class library
- Geared towards horizontal vectorisation
- Good code scales “out of the box”

Fog’s Vectorclass
- Exposes the vector ISA in a human readable way
- Good for vertical vectorisation

Intrinsics
- Gives a taste of how the architecture works
- Well documented – throughput, latency considerations per instruction
Some LHCb specific results

Application-specific implementations will vary the result of your vectorisation code.

<table>
<thead>
<tr>
<th>library</th>
<th>PrPixel addHits</th>
<th>RICH EigenGeom</th>
<th>arch portable</th>
<th>comp portable</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
</tr>
<tr>
<td>intrinsics</td>
<td>1.94x</td>
<td>2.57x</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>gcc intrinsics</td>
<td>1.45x</td>
<td>2.09x</td>
<td>yes</td>
<td>no (icc)</td>
</tr>
<tr>
<td>vc</td>
<td>1.66x</td>
<td>2.26x</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Vectorclass</td>
<td>1.60x</td>
<td>2.23x</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Vertical vectorisation examples – All SSE
Same program, compiled with gcc 4.8.1
HLT2010 - Intel Xeon CPU X5650 @ 2.67GHz
HLT2015 - Intel Xeon CPU E5-2630 v3 @ 2.40GHz
Cross compiler results

![Bar chart showing cross compiler results for different architectures and compilers.](chart.png)

- **HLT2010 - X5650 @ 2.67GHz**
  - gcc 4.8.1
  - gcc 4.9.2
  - clang 3.7.0
  - icc 15.0

- **HLT2015 - E5-2630 v3 @ 2.40GHz**
  - gcc 4.8.1
  - gcc 4.9.2
  - clang 3.7.0
  - icc 15.0

- **E5-2687W v3 @ 3.10GHz**
  - gcc 4.8.1
  - gcc 4.9.2
  - clang 3.7.0
  - icc 15.0
On the menu

Vectorisation is good

Get creative
Rethinking our software

- How about making tracking reconstruction parallel *by design*?
To the cost of...

<table>
<thead>
<tr>
<th>vanilla - brunel</th>
<th>1249264 tracks including</th>
<th>30365 ghosts [ 2.4 %], Event average 1.9 % ****</th>
</tr>
</thead>
<tbody>
<tr>
<td>velo</td>
<td>1080519 from 1169018 [ 92.4 %]</td>
<td>21110 clones [ 1.9 %], purity: 99.83 %, hitEff: 95.92 %</td>
</tr>
<tr>
<td>long</td>
<td>367708 from 370217 [ 99.3 %]</td>
<td>3109 clones [ 0.8 %], purity: 99.83 %, hitEff: 98.34 %</td>
</tr>
<tr>
<td>long&gt;5GeV</td>
<td>244699 from 245655 [ 99.6 %]</td>
<td>1373 clones [ 0.6 %], purity: 99.84 %, hitEff: 98.79 %</td>
</tr>
<tr>
<td>long_strange&gt;5GeV</td>
<td>14788 from 15034 [ 98.4 %]</td>
<td>119 clones [ 0.8 %], purity: 99.92 %, hitEff: 98.30 %</td>
</tr>
<tr>
<td>long_fromB&gt;5GeV</td>
<td>7245 from 7359 [ 98.5 %]</td>
<td>27 clones [ 0.4 %], purity: 99.18 %, hitEff: 99.00 %</td>
</tr>
<tr>
<td>long_fromB&gt;5GeV</td>
<td>20967 from 21087 [ 99.4 %]</td>
<td>95 clones [ 0.5 %], purity: 99.79 %, hitEff: 98.72 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gpu</th>
<th>1184118 tracks including</th>
<th>11428 ghosts [ 1.0 %], Event average 0.8 % ****</th>
</tr>
</thead>
<tbody>
<tr>
<td>velo</td>
<td>1036991 from 1169018 [ 88.7 %]</td>
<td>20273 clones [ 1.9 %], purity: 99.79 %, hitEff: 95.76 %</td>
</tr>
<tr>
<td>long</td>
<td>366453 from 370217 [ 99.0 %]</td>
<td>4269 clones [ 1.2 %], purity: 99.78 %, hitEff: 98.04 %</td>
</tr>
<tr>
<td>long&gt;5GeV</td>
<td>244348 from 245655 [ 99.5 %]</td>
<td>2263 clones [ 0.9 %], purity: 99.82 %, hitEff: 98.50 %</td>
</tr>
<tr>
<td>long_strange&gt;5GeV</td>
<td>14689 from 15034 [ 97.7 %]</td>
<td>139 clones [ 0.9 %], purity: 99.24 %, hitEff: 98.11 %</td>
</tr>
<tr>
<td>long_fromB&gt;5GeV</td>
<td>7226 from 7359 [ 99.2 %]</td>
<td>52 clones [ 0.7 %], purity: 99.15 %, hitEff: 98.72 %</td>
</tr>
<tr>
<td>long_fromB&gt;5GeV</td>
<td>20844 from 21087 [ 99.8 %]</td>
<td>161 clones [ 0.8 %], purity: 99.74 %, hitEff: 98.37 %</td>
</tr>
<tr>
<td>long_fromB&gt;5GeV</td>
<td>17110 from 17242 [ 99.2 %]</td>
<td>113 clones [ 0.7 %], purity: 99.79 %, hitEff: 98.58 %</td>
</tr>
</tbody>
</table>

Courtesy of C. Potterat
What *had* to change

Algorithm makes assumptions from the framework, and data design
- *Many* events on flight
- Data issues
  - Data format – AoS vs SoA – Hiding under templates
  - Data alignment – Cache line splits
  - Locality

Optimisation is a tricky process
- Profiling tools are a must for quality software
  - Not always integrable with framework
LHCb PR

Performance and Regression tests
- Brunel timing with different configs
- Moore/Brunel Valgrind/Callgrind profiling

Periodic testing
- Cron like system
- Integration with Jenkins
- Automatic building / testing

Configuration
- Consolidated XML file
- Nightlies SVN repo
All aboard the vectorisation train

- Vectors have been there for a while and will stay – **Time to use them**

- Need to evaluate our software cross machine, library, compiler – LHCb PR solves that

- Some issues to rethink
  - float vs double – more precision doesn’t mean better code
  - loads and stores are main show stopper – Sections of code with many EX ops are more likely to benefit from vectorisation
  - Data issues – format, alignment, locality – for the most part unsolved and hard with an existing big framework
  - manycore is next stop, redesigning for SIMD is a good idea

- Evaluated vector libraries
  - Putting readability over other factors –
    - Fog’s **Vectorclass** for vertical vectorisation
    - **vc** for horizontal one
    - Awaiting for vc’s simdarray branch
Handy materials

- [http://www.agner.org/optimize/](http://www.agner.org/optimize/)
- Intel intrinsics guide
- Microsoft guide to intrinsics
- [http://d3f8ykwia686p.cloudfront.net/1live/intel/An_Introduction_to_Vectorization_with_Intel_Compiler_021712.pdf](http://d3f8ykwia686p.cloudfront.net/1live/intel/An_Introduction_to_Vectorization_with_Intel_Compiler_021712.pdf)
- [http://code.compeng.uni-frankfurt.de/projects/vc](http://code.compeng.uni-frankfurt.de/projects/vc)
- [http://fias.uni-frankfurt.de/de/cs/kisel/lectures/](http://fias.uni-frankfurt.de/de/cs/kisel/lectures/)
Fire in the hole
Backup! Backup!
Simple yet expressive!

\[
\begin{align*}
v0 &= \_\text{mm\_set\_1\_ps}(-0.f); \\
v1 &= \_\text{mm\_xor\_ps}(v0, v1);
\end{align*}
\]

\[
\begin{align*}
v\_\text{compValue} &= \_\text{mm\_set\_1\_ps}(10e-10); \\
v\_\text{sign\_mask} &= \_\text{mm\_set\_1\_ps}(-0.f); \\
v\_1s &= \_\text{mm\_set\_1\_ps}(1.f); \\
v7 &= \ldots \\
v6 &= \_\text{mm\_and\_not\_ps}(v\_\text{sign\_mask}, v7); \\
v1 &= \_\text{mm\_cmplt\_ps}(v6, v\_\text{compValue}); \\
v2 &= \_\text{mm\_and\_not\_ps}(v1, v7); \\
v3 &= \_\text{mm\_and\_ps}(v1, v\_1s); \\
v7 &= \_\text{mm\_add\_ps}(v2, v3);
\end{align*}
\]

\[
v1 = -v1;
\]
v0 = _mm_set1_ps(-0.f);
v1 = _mm_xor_ps(v0, v1);

v_compValue = _mm_set1_ps(10e-10);
v_sign_mask = _mm_set1_ps(-0.f);
v_1s = _mm_set1_ps(1.f);
v7 = ...

v6 = _mm_andnot_ps(v_sign_mask, v7);
v1 = _mm_cmplt_ps(v6, v_compValue);
v2 = _mm_andnot_ps(v1, v7);
v3 = _mm_and_ps(v1, v_1s);
v7 = _mm_add_ps(v2, v3);

v1 = -v1;

v_compValue = 10e-10f;
v_1s = 1.0f;
v7 = ...

// Conditional assignment
v7(abs(v7) < v_compValue) = v_1s;
# Cross compiler solveParabola

## Scifi solveParabola

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Reordered</th>
<th>Reordered 2</th>
<th>Vector</th>
<th>Vector aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc 4.8</td>
<td>1.11x</td>
<td>1.18x</td>
<td>1.22x</td>
<td>1.29x</td>
</tr>
<tr>
<td>icc 14.0</td>
<td>1.05x</td>
<td>1.04x</td>
<td>1.15x</td>
<td>1.21x</td>
</tr>
<tr>
<td>clang 3.4</td>
<td>1.14x</td>
<td>1.16x</td>
<td>1.36x</td>
<td>1.43x</td>
</tr>
</tbody>
</table>

Intel Xeon E5-2670 v2 @ 2.50GHz
All results compared against vanilla version with same compiler, -O2

speedup is measured against vanilla Brunel implementation.
The percentage is how much time is saved, in the whole algorithm (Pr::PrAlgorithms PrSeedingXLayers in this case), tried against hundreds of MC events.
More cross compiler results (speedup)