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)
    - SSE3 (2004), SSSE3 (2006), SSE4.1, SSE4.2 (2008) – 128-bit registers
  - AVX (2011), AVX2 (2012) – 256-bit registers
  - 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

```cpp
// 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
Vertical vectorisation

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

```c
// tx = (sx * s0 - sx * sz) / den;
// ty = (uyz * u0 - uy * uz) / den2;
// x0 = (sx * sz2 - sxz * sz) / den;
// y0 = (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

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

higher level
vc
Fog’s Vectorclass

gcc intrinsics

lower level
Intel intrinsics
ASM
Some LHCb specific results

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

| library       | PrPixel addHits | RICH EigenGeom | arch portable | comp portable |
|---------------|----------------|----------------|---------------|---------------|
|               | X5650 @ 2.67GHz | ES-2630 v3 @ 2.40GHz | X5650 @ 2.67GHz | ES-2630 v3 @ 2.40GHz |
| sequential    | 1.00x          | 1.00x          | 1.00x         | 1.00x         |
| intrinsics    | 1.94x          | 2.57x          |               |               |
| gcc intrinsics| 1.45x          | 2.09x          | yes           | no (icc)      |
| vc            | 1.66x          | 2.26x          | 1.35x         | 1.49x         |
| Vectorclass   | 1.60x          | 2.23x          | 1.35x         | 1.50x         |

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

- HLT2010 - X5650 @ 2.67GHz
- HLT2015 - E5-2630 v3 @ 2.40GHz
- 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...

| vanilla - brunel       |                  |              |                  |                  |
|------------------------|------------------|--------------|------------------|------------------|
| 1249264 tracks including | 30365 ghosts [ 2.4 %], Event average 1.9 % **** |
| velo : 1080519 from 1169018 [ 92.4 %] | 21110 clones [ 1.9 %], purity: 99.83 %, hitEff: 95.92 % |
| long : 367708 from 370217 [ 99.3 %] | 3109 clones [ 0.8 %], purity: 99.83 %, hitEff: 98.34 % |
| long>5GeV : 244699 from 245655 [ 99.6 %] | 1373 clones [ 0.6 %], purity: 99.84 %, hitEff: 98.79 % |
| longStrange>5GeV : 14788 from 15034 [ 98.4 %] | 119 clones [ 0.8 %], purity: 99.32 %, hitEff: 98.30 % |
| long_fromB : 7245 from 7359 [ 98.5 %] | 27 clones [ 0.4 %], purity: 99.18 %, hitEff: 99.00 % |
| long_fromB>5GeV : 17179 from 17242 [ 99.6 %] | 55 clones [ 0.3 %], purity: 99.84 %, hitEff: 98.94 % |

| gpu                     |                  |              |                  |                  |
|-------------------------|------------------|--------------|------------------|------------------|
| 1184118 tracks including | 11428 ghosts [ 1.0 %], Event average 0.8 % **** |
| velo : 1036991 from 1169018 [ 88.7 %] | 20273 clones [ 1.9 %], purity: 99.79 %, hitEff: 95.76 % |
| long : 366453 from 370217 [ 99.0 %] | 4269 clones [ 1.2 %], purity: 99.78 %, hitEff: 98.04 % |
| long>5GeV : 244348 from 245655 [ 99.5 %] | 2263 clones [ 0.9 %], purity: 99.82 %, hitEff: 98.50 % |
| longStrange>5GeV : 14689 from 15034 [ 97.7 %] | 139 clones [ 0.9 %], purity: 99.24 %, hitEff: 98.11 % |
| long_fromB : 7226 from 7359 [ 99.2 %] | 52 clones [ 0.7 %], purity: 99.15 %, hitEff: 98.72 % |
| long_fromB>5GeV : 17110 from 17242 [ 99.2 %] | 113 clones [ 0.7 %], purity: 99.79 %, hitEff: 98.58 % |

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 sindarray branch
Handy materials

- [http://www.agner.org/optimize/](http://www.agner.org/optimize/)
- Intel intrinsics guide
- Microsoft guide to intrinsics
- [http://d3f8ykwhia686p.cloudfront.net/1live/intel/An_Introduction_to_Vectorization_with_Intel_Compiler_021712.pdf](http://d3f8ykwhia686p.cloudfront.net/1live/intel/An_Introduction_to_Vectorization_with_Intel_Compiler_021712.pdf)
- [http://d3f8ykwhia686p.cloudfront.net/1live/intel/CompilerAutovectorizationGuide.pdf](http://d3f8ykwhia686p.cloudfront.net/1live/intel/CompilerAutovectorizationGuide.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!

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

v1 = -v1;

v_compValue = _mm_set1_ps(1e-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);
Simple yet expressive!

\[
\begin{align*}
v_0 &= \text{_mm_set1_ps}(-0.f); \\
v_1 &= \text{_mm_xor_ps}(v_0, v_1); \quad v_1 = -v_1; \\
\text{v_compValue} &= \text{_mm_set1_ps}(10e-10); \\
\text{v_sign_mask} &= \text{_mm_set1_ps}(-0.f); \\
\text{v_1s} &= \text{_mm_set1_ps}(1.f); \\
v_7 &= \text{...} \\
v_6 &= \text{_mm_andnot_ps}(v\_sign\_mask, v_7); \\
v_1 &= \text{_mm_cmplt_ps}(v_6, \text{v_compValue}); \\
v_2 &= \text{_mm_andnot_ps}(v_1, v_7); \\
v_3 &= \text{_mm_and_ps}(v_1, v_1s); \\
v_7 &= \text{_mm_add_ps}(v_2, v_3); \\
\text{v_compValue} &= 10e-10f; \\
\text{v_1s} &= 1.0f; \\
v_7 &= \text{...} \\
// \text{Conditional assignment} \\
v_7(\text{abs}(v_7) < \text{v_compValue}) = \text{v_1s}; 
\end{align*}
\]
Cross compiler solveParabola

Scifi solveParabola

| Compiler | Reordered | Reordered 2 | Vector | Vector aligned |
|----------|-----------|------------|--------|----------------|
| gcc 4.8  | 1.11x     | 1.18x      | 1.22x  | 1.29x          |
| icc 14.0 | 1.05x     | 1.04x      | 1.15x  | 1.21x          |
| clang 3.4| 1.14x     | 1.16x      | 1.36x  | 1.43x          |

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)

| Compiler | HLT2010 - X5650 @ 2.67GHz | HLT2015 - E5-2630 v3 @ 2.40GHz | E5-2687W v3 @ 3.10GHz |
|----------|---------------------------|-------------------------------|-----------------------|
| gcc 4.8.1 | 1.5                       | 2.0                           | 2.5                   |
| gcc 4.9.2 | 1.7                       | 2.2                           | 2.7                   |
| clang 3.7.0 | 2.0                       | 2.3                           | 2.8                   |
| icc 15.0  | 2.3                       | 2.5                           | 3.0                   |