Vector processing, Boost.SIMD and compiler auto-vectorization

André Tupinambá

Vector processing

Processor instructions over multiple data

Scalar (SISD)
- 1 instruction
- 1 data op.

Vetorial (SIMD)
- 1 instruction
- *n* data operation

Processor instructions

- x86 – Intel
  - MMX, SSE(1 to 4.2, SSSE3); AVX(2); FMA3
- x86 – AMD
  - 3D Now!; 3D Now!+; SSE4a; FMA4
- Xeon Phi – Intel
  - MIC, AVX-512
- ARM
  - Neon
- PowerPC
  - VMX, VSX

x86 Vector instructions

Pentium (1993) – Scalar

64 bits

MMX (1997) – 2 ints

128 bits

SSE (1999) to SSE 4.2 (2008) – 4 floats

256 bits

AVX (2011) and AVX2 (2013) – 8 floats

Example: Gaussian Elimination

From wikipedia:

“Is an algorithm for solving systems of linear equations. (...) uses a sequence of elementary row operations”

$$\left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 1 & 1 & -1 & 1 \\ 3 & 11 & 5 & 35 \end{array}\right] \to \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 0 & -2 & -2 & -8 \\ 0 & 2 & 2 & 8 \end{array}\right] \to \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 0 & -2 & -2 & -8 \\ 0 & 0 & 0 & 0 \end{array}\right] \to \left[\begin{array}{ccc|c} 1 & 0 & -2 & -3 \\ 0 & 1 & 1 & 4 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

Gaussian Elimination code

```
using t_vec = std::vector< float >;
void gauss( t_vec& mat, t_vec& fac ) {
  size_t wd = fac.size();
  for( size_t ln = 0; ln < wd - 1; ++ln ) {
    for( size_t y = ln + 1; y < wd; ++y ) {
      float sc = mat[  y * wd + ln ] /
                 mat[ ln * wd + ln ];
      fac[ y ] -= sc * fac[ ln ];
      for( size_t x = ln; x < wd; ++x ) {
        mat[ y * wd + x ] -= sc * mat[ ln * wd + x ];
}}}}
```

Algorithm at a glance

1. Set first line as baseline
2. For each line below baseline
3. Choose a constant value
4. Subtract each number of line from baseline * constant
5. Move baseline down 1 line
6. Repeat 2-5 to the end

SIMD intrinsics

- ***Intrinsics*** are functions whose implementation are handled specially by the compiler

- SIMD intrinsics are generally translated into assembly instructions
  - There are specific intrinsics for MMX, SSE, SSE2, SSE3, etc.

https://software.intel.com/sites/landingpage/IntrinsicsGuide

Intel® 64 and IA-32 architectures software developer's manual volumes 2A, 2B, 2C, and 2D: Instruction set reference, A-Z

Gaussian elimination - SSE2 Code

```
using t_vec = std::vector< float >;
void gaussIntrinsics( t_vec& mat, t_vec& fac ) {
  size_t wd = fac.size();
  for( size_t ln = 0; ln < wd - 1; ++ln ) {
    for( size_t y = ln + 1; y < wd; ++y ) {
      float sc = mat[ y * wd + ln ] / mat[ ln * wd + ln ];
      fac[ y ] -= sc * fac[ ln ];
      __m128 xSc = _mm_set1_ps( sc );
      size_t norm = ln & ~(3);
      for( size_t x = norm; x < wd; x += 4 ) {
        __m128 b = _mm_load_ps( mat.data() + ln * wd + x );
        __m128 val = _mm_load_ps( mat.data() + y * wd + x );
        val = _mm_sub_ps( val, _mm_mul_ps( xSc, b ) );
        _mm_store_ps( mat.data() + y * wd + x, val );
}}}}
```

Performance analysis

- Environment
  - Core i7-4870HQ
  - MacOSX El Captain 10.11.6
  - Clang 800.0.42.1 (XCode 8.2.1)
- Two compiler setups
  - With SSE2 (-msse2 -O3)
  - With AVX2 (-mavx2 -O3)
- Gaussian elimination
  - 768 x 768 matrix
  - Run 200 times

Results


Base
Intrinsics
30.00
22.50
15.00
7.50
0.00
seconds
25.59
7.54
26.09
7.67
SSE2
AVX2

Problems with intrinsics

- Too low level
  - Specific code for each processor and data type
  - The example code is for SSE2 and float only
- Makes code harder to read
  - Intrinsics are mnemonic
    - _mm_mul_ss, _mm_mul_ps, _mm_mul_sd, _mm_mul_pd, _mm_mulhi_epi16, _mm_mulhi_epu16, _mm_mullo_epi16, _mm_mul_su32, _mm_mul_epu32
- Manually care about memory alignment
  - Older processors don’t allow load/store from unaligned addresses
  - Newer ones have faster loads/stores on aligned memory
    - Different intrinsics for aligned or unaligned loads/stores

Boost.SIMD

- Boost candidate library
- Subset of NumScale's bSIMD library

- Supports x86 instruction sets only
  - SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, FMA3, AVX2

https://github.com/NumScale/boost.simd

Code with Boost.SIMD

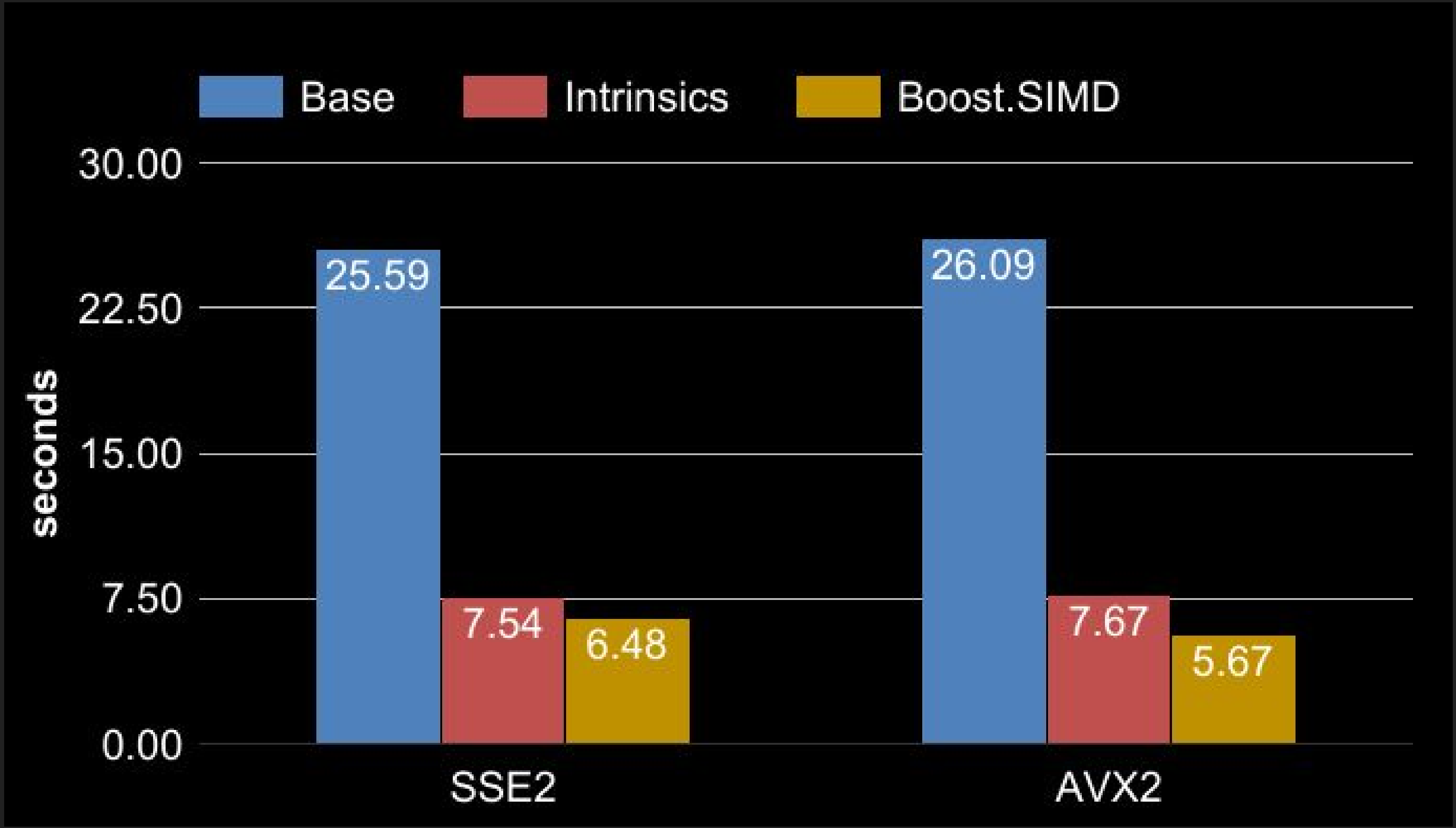
```
namespace bs = boost::simd;
using t_vec = std::vector< float, bs::allocator<float> >;
using t_pack = bs::pack< float >;
void gaussBoost( t_vec& mat, t_vec& fac ) {
  size_t wd = fac.size();
  t_pack* pm = reinterpret_cast< t_pack* >( mat.data() );
  size_t ps = t_pack::static_size;
  for( size_t ln = 0; ln < wd - 1; ++ln ) {
    for( size_t y = ln + 1; y < wd; ++y ) {
      float sc = mat[ y * wd + ln ] / mat[ ln * wd + ln ];
      fac[ y ] -= sc * fac[ ln ];
      size_t norm = ln & ~(ps - 1);
      for( size_t x = norm; x < wd; x += ps ) {
        pm[ ( y * wd + x ) / ps ] -= sc * pm[ ( ln * wd + x ) / ps ];

}}}}
```

Code with Boost.SIMD

```
namespace bs = boost::simd;
using t_vec = std::vector< float, bs::allocator<float> >;
using t_pack = bs::pack< float >;
void gaussBoost( t_vec& mat, t_vec& fac ) {
  size_t wd = fac.size();
  t_pack* pm = reinterpret_cast< t_pack* >( mat.data() );
  size_t ps = t_pack::static_size;
  for( size_t ln = 0; ln < wd - 1; ++ln ) {
    for( size_t y = ln + 1; y < wd; ++y ) {
      float sc = mat[ y * wd + ln ] / mat[ ln * wd + ln ];
      fac[ y ] -= sc * fac[ ln ];
      size_t norm = ln & ~(ps - 1);
      for( size_t x = norm; x < wd; x += ps ) {
        pm[ ( y * wd + x ) / ps ] = bs::fma( -sc, pm[ ( ln * wd + x ) / ps ],
                                            pm[ ( y * wd + x ) / ps ] );
}}}}
```

Results (update)



Boost.SIMD

- Easy to change type
  - The change from float to double:

```
using t_vec = std::vector< double, bs::allocator< double > >;
using t_pack = bs::pack< double >;
```

- Easy to change instruction set
  - Just recompile
- Not so low level
  - FMA, dot product, reduce, ranges, transform
  - Arithmetic operator overload
  - STL allocator

Compiler auto-vectorization

- Compiler optimization feature
  - Usually enabled in -O3
- Two methods
  - Loop vectorize
  - Similar operation vectorize
- Data dependency problems
  - Compiler need to be sure that data are independent

Gaussian Elimination code

```
using t_vec = std::vector< float >;
void gauss( t_vec& mat, t_vec& fac ) {
  size_t wd = fac.size();
  for( size_t ln = 0; ln < wd - 1; ++ln ) {
    for( size_t y = ln + 1; y < wd; ++y ) {
      float sc = mat[  y * wd + ln ] /
                 mat[ ln * wd + ln ];
      fac[ y ] -= sc * fac[ ln ];
      for( size_t x = ln; x < wd; ++x ) {
        mat[ y * wd + x ] -= sc * mat[ ln * wd + x ];
}}}}
```

How help compiler to vectorize?

- Loop vectorize cares about data dependencies
  - Wait! The data are independent here!

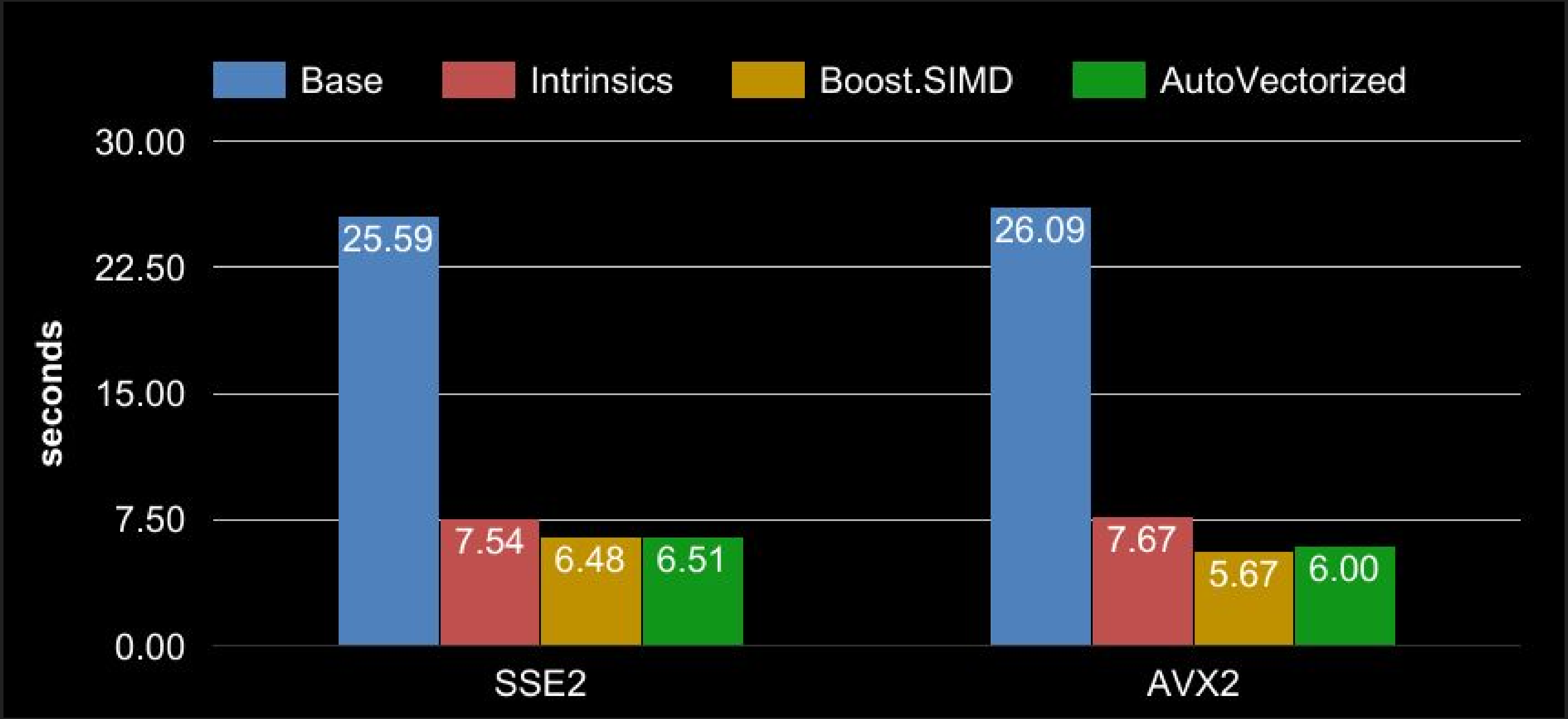
- We know, but the compiler not
  - Compiler can give you some hints
  - In GCC: -ftree-vectorizer-verbose=2

- Let’s change to pointers...

Gaussian Elimination code

```
using t_vec = std::vector< float >;
void gauss( t_vec& mat, t_vec& fac ) {
  size_t wd = fac.size();
  for( size_t ln = 0; ln < wd - 1; ++ln ) {
    for( size_t y = ln + 1; y < wd; ++y ) {
      float sc = mat[  y * wd + ln ] /
                 mat[ ln * wd + ln ];
      fac[ y ] -= sc * fac[ ln ];
      float* pBase = mat.data() + ln * wd + ln;
      float* pLine = mat.data() + y * wd + ln;
      for( size_t x = ln; x < wd; ++x ) {
        *pLine++ -= sc * *pBase++;
}}}}
```

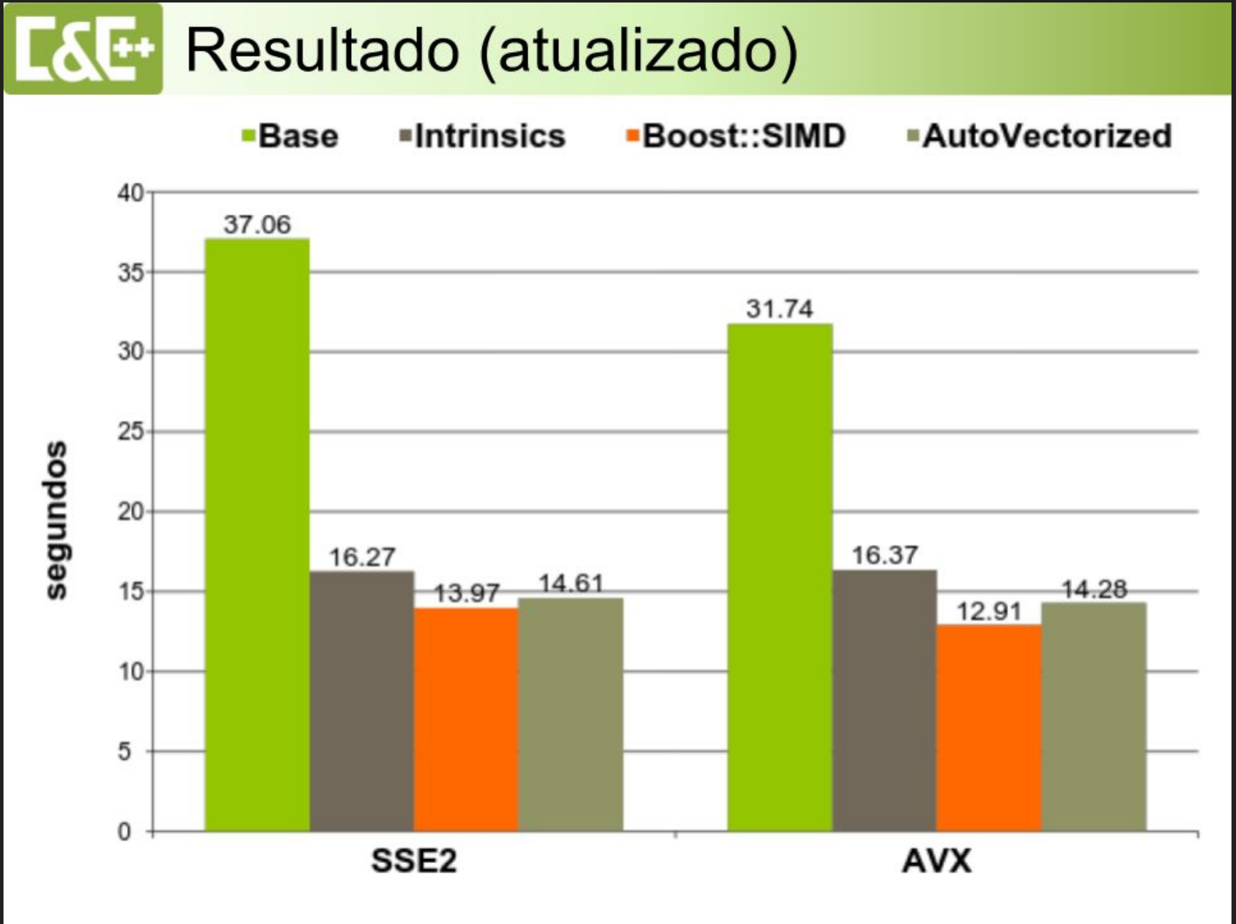
Results (update)



Conclusion

- Vector processing was able to get better performance in this scenario
    - ~3.95x in SSE2, ~4.60x AVX2
- Intrinsics, Boost.SIMD and auto-vectorization achieve similar results
- Boost.SIMD is simpler than intrinsics and you have more control than auto-vectorization
- Auto-vectorization is simpler than Boost.SIMD, but need compiler help and some try and error

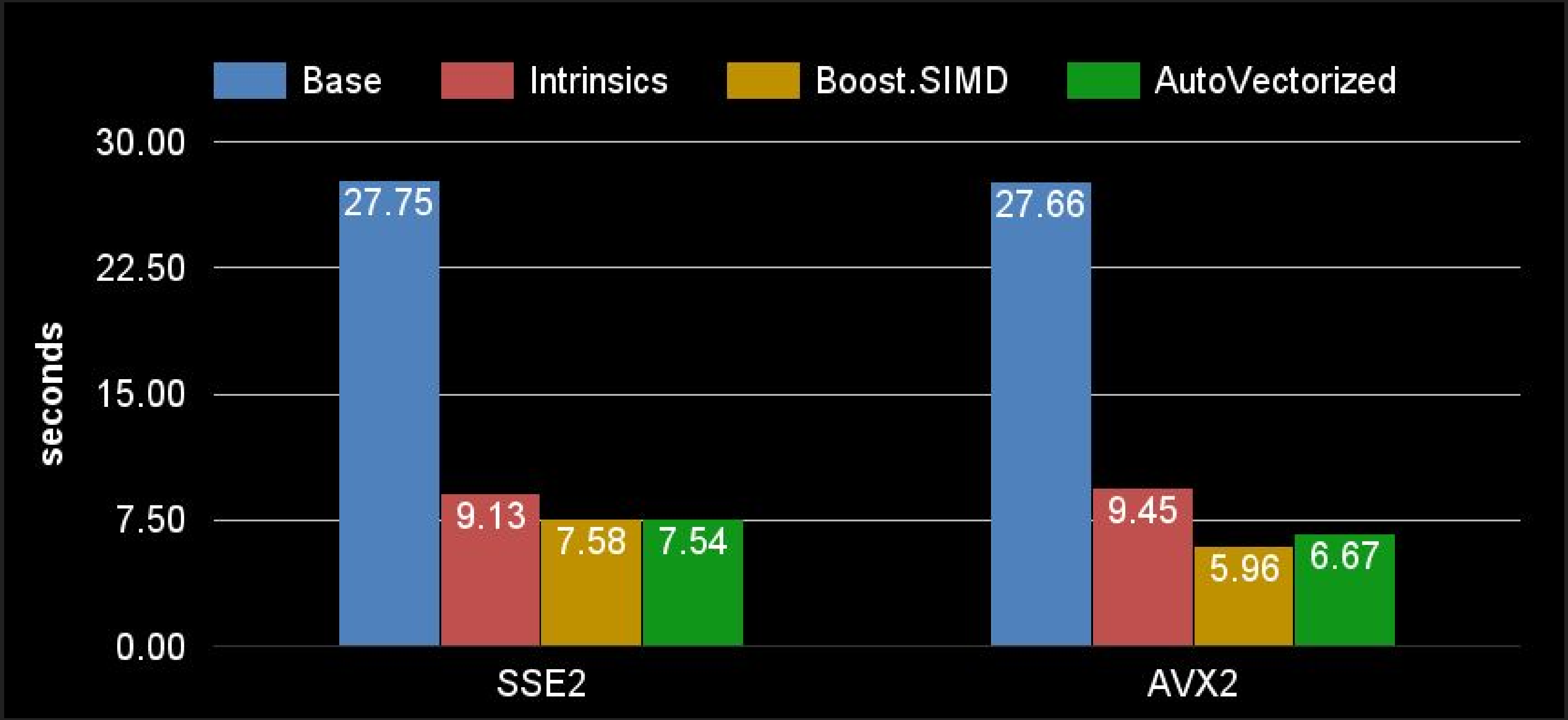
Old results



- Core i3-2310M
- Windows 7 64bits
- Visual Studio 2013

- ~2.6x SSE2
- ~2.4x AVX

Arista server results



GCC 4.9, Xeon E5-2640, ~3.66x SSE2, ~4.64x AVX2

Get the code!

- Code and presentation are in github
    - Also with bs::ranges, bs::transform, OpenMP and loop-unroll

https://github.com/andrelrt/boostSimdTest

- Keep in touch
    - andrelrt@gmail.com

Vector processing, Boost.SIMD and compiler auto-vectorization

André Tupinambá