



Processamento vetorial, Boost::SIMD e autovetorização

11º Encontro do grupo C/C++ Brasil

André Tupinambá



O que é processamento vetorial?

Instruções do processador capazes de executar sobre um conjunto de dados

Escalar (SISD)

- 1 instrução
- 1 operação



+



=

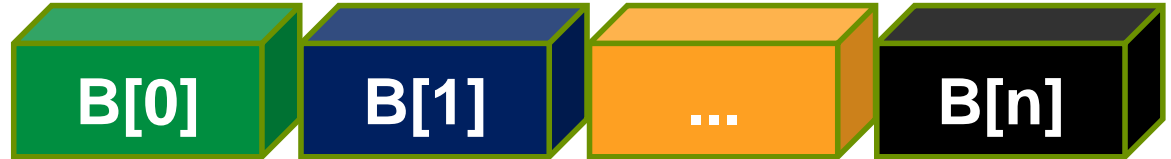


Vetorial (SIMD)

- 1 instrução
- n operações



+



=



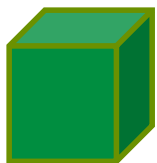


Conjunto de instruções específicas

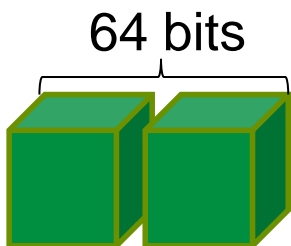
- x86 – Intel
 - MMX, SSE(1 a 4.2, SSSE3); AVX(2); FMA3
- x86 – AMD
 - 3D Now!; 3D Now!+; SSE4a; FMA4
- Xeon Phi – Intel
 - MIC
- ARM
 - Neon
- PowerPC
 - VMX, VSX



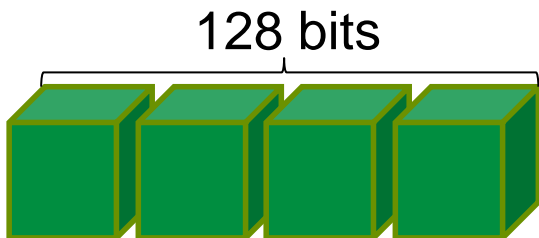
Processamento vetorial no x86



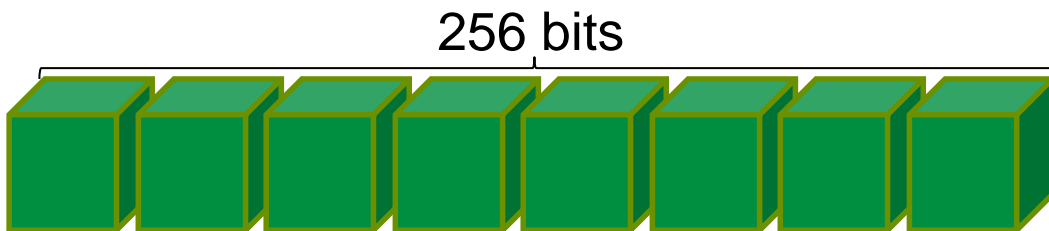
Pentium (1993) – Escalar



MMX (1997) – 2 inteiros 32 bits



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



AVX (2011) e AVX2 (2013) – 8 floats

- Algoritmo para resolver sistemas de equações lineares
 - Consiste em aplicar sucessivas operações elementares entre as linhas de uma matriz
 - Também usado para inversão de matrizes

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

Fonte: Wikipédia



Código da Eliminação de Gauss

```
typedef std::vector< float > t_vec;
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];
            }
        }
    }
}
```



Característica do algoritmo

- Cada laço depende somente da linha base
 - Para mudar a linha base precisa calcular toda matriz
- Independência entre as linhas
 - Cada linha pode ser tratada isoladamente
- Independência entre os números na linha
 - Cada número pode ser tratado isoladamente
- Oportunidade para usar processamento vetorial



SIMD *intrinsics*

- *Intrinsics* são funções tratadas especialmente pelo compilador
 - Similar a funções *inline*, mas possui um tratamento específico
- Os *intrinsics* SIMD são traduzidos diretamente para instruções em assembly
 - Existem *intrinsics* específicos para as instruções MMX, SSE, SSE2, SSE3 ...
- Intel 64 and IA-32 Architectures Software Developer's Manual Volumes 2A, 2B, and 2C: Instruction Set Reference, A-Z



Código usando *intrinsics* SSE2

```
typedef std::vector< float > t_vec;
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);
                }
            }
        }
    }
}
```

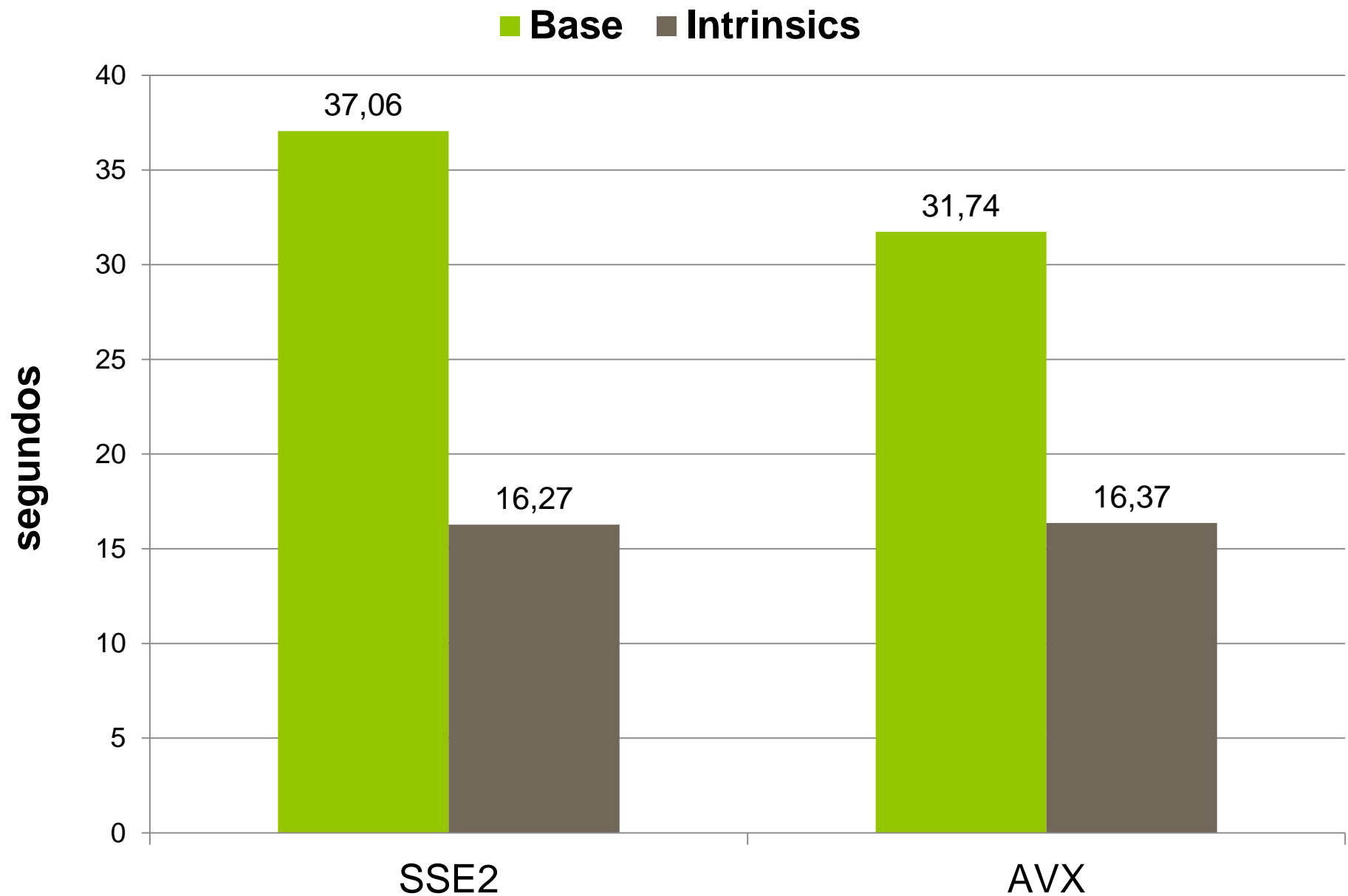


Avaliação de desempenho

- Ambiente
 - Processador Core i3 - 2310M (possui AVX)
 - Windows 7 64-bits
 - Visual Studio Community 2013
- Duas compilações
 - Com suporte a SSE2
 - Com suporte a AVX
- Eliminação de Gauss
 - Matrizes 768 x 768
 - Executado 200 vezes



Resultado





Problemas ao usar *intrinsics*

- “Baixo nível”
 - Código ligado ao tipo e às instruções
 - Exemplo somente com SSE2 para float
- Leitura difícil
 - *Intrinsics* são mnemônicos
 - `_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`
- Gerência de memória
 - O vector precisa estar alinhado



Boost::SIMD

- Biblioteca “boost candidate”
 - Ainda não é oficialmente aceita
- Base da biblioteca NT² da NumScale
- Suporta várias instruções e processadores
 - Intel/AMD x86 (SSE, SSE2, SSE3...)
 - Intel XeonPhi (MIC)
 - IBM POWER/PowerPC (VMX, VSX)
 - IBM BlueGene-Q (QPX)
 - ARM processors (NEON)
 - Texas Instruments (C6x)

<http://www.numscale.com/products/boost-simd>



Código usando Boost::SIMD

```
namespace bs = boost::simd;
```

```
→ typedef std::vector< float, bs::allocator<float> > t_vec;
```

```
→ typedef bs::pack< float > t_pack;
```

```
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];
```

```
}}}}
```



Código usando Boost::SIMD

```
namespace bs = boost::simd;
```

```
→ typedef std::vector< float, bs::allocator<float> > t_vec;
```

```
→ typedef bs::pack< float > t_pack;
```

```
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] );
```

```
}}}}
```



Facilidades do Boost::SIMD

- Liberdade de tipo

- Mudança para trabalhar com **double**:

```
typedef vector< double, bs::allocator<double> > t_vec;  
typedef bs::pack< double > t_pack;
```

- Permite usar diferentes instruções SIMD

- Mas precisa recompilar o código

- Funções um pouco mais alto nível

- E.g., possui FMA e Dot product mesmo se o processador não possua essas instruções

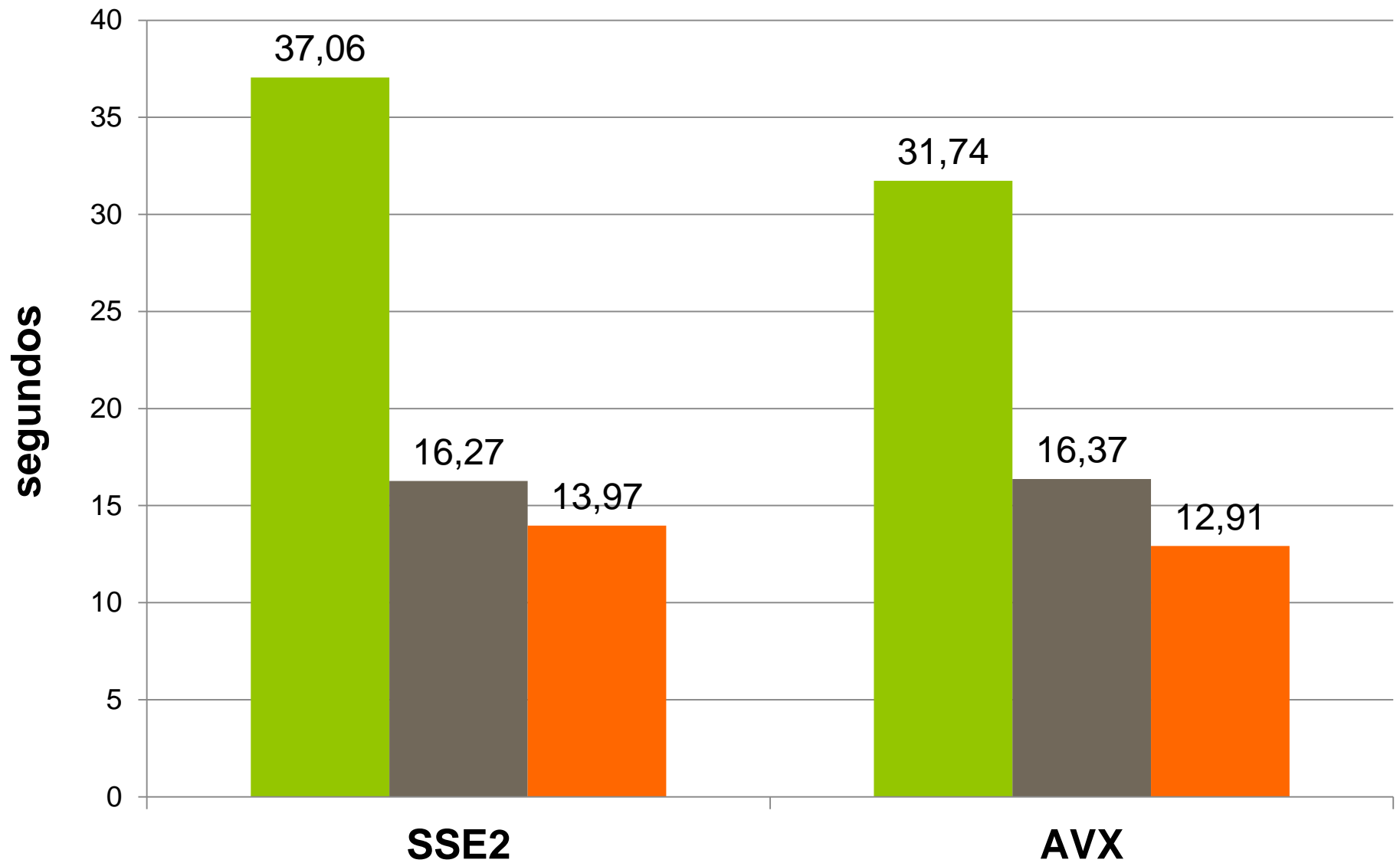
- Sobrecarga dos operadores C++

- Gerenciamento de memória



Resultado (atualizado)

■ Base ■ Intrinsics ■ Boost::SIMD






Autovetorização

- Capacidade do compilador em encontrar oportunidades para usar processamento vetorial
- Precisa da “ajuda” do desenvolvedor
- Vamos voltar ao nosso exemplo...



Código da Eliminação de Gauss

```
typedef std::vector< float > t_vec;
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];
            }
        }
    }
}
```



Como ajudar o compilador

- O compilador precisa “ter certeza” que os dados:
 - Sejam independentes entre os loops
 - São tratados igualmente e de forma sequencial
- O uso de ponteiros pode ajudar neste caso



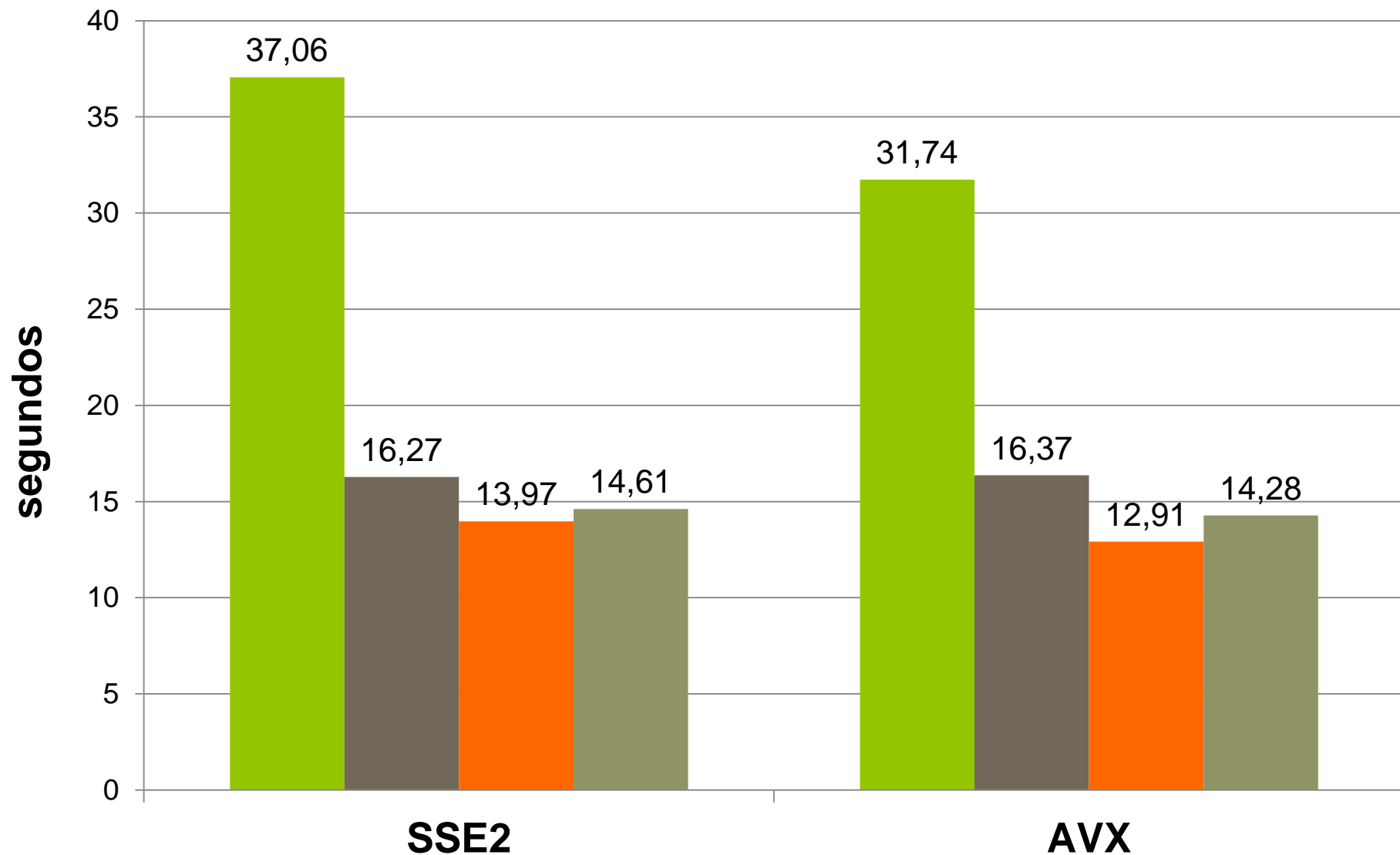
Código da Eliminação de Gauss

```
typedef std::vector< float > t_vec;
void gaussAutovec(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++;
            }
        }
    }
}
```



Resultado (atualizado)

■ Base ■ Intrinsics ■ Boost::SIMD ■ AutoVectorized





Melhoria no desempenho

- Speedup conseguido através do uso de vetorização

	SSE2	AVX
Intrinsics	2,28x	1,94x
Boost::SIMD	2,65x	2,46x
AutoVectorized	2,54x	2,22x



Conclusões

- A vetorização conseguiu reduzir o tempo do processamento em $\sim 2x$
- A Boost::SIMD, *intrinsics* e autovetorização possuem ganhos de desempenho similares
- A Boost::SIMD possibilita usar o mesmo código para conjunto de instruções diferentes
- A autovetorização é mais simples, mas requer conhecimento do compilador e alguma tentativa e erro



Entre em contato

- Código da apresentação disponível
 - Também com loop unroll e OpenMP
 - <https://github.com/andreirt/boostSimdTest>
- C & C++ Brasil
 - <http://ccppbrasil.org>
- Telegram
 - <http://telegram.me/andreirt>
- Email
 - andreirt@gmail.com



Processamento vetorial, Boost::SIMD e autovetorização

11º Encontro do grupo C/C++ Brasil

André Tupinambá