
LNCC
Laboratório Nacional de Computação Científica

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO

GOVERNO DO BRASIL
DO LADO DO POVO BRASILEIRO

# *Profiling* e otimização em códigos C/C++

## Parte 1

**Professor:** Hiago Mayk G. de A. Rocha
**E-mail:** mayk@lncc.br

# Professor

**Hiago Mayk Gomes de Araújo Rocha**

Endereço para acessar este CV: http://lattes.cnpq.br/8662578168161845

ID Lattes: **8662578168161845**

Última atualização do currículo em 29/12/2025

Possui doutorado em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS), concluído em 2024, mestrado em Sistemas e Computação pela Universidade Federal do Rio Grande do Norte (UFRN), concluído em 2019, e bacharelado em Tecnologia da Informação e em Ciência da Computação, ambos pela UFRN, em 2017. Atualmente, atua como pesquisador adjunto no Laboratório Nacional de Computação Científica (LNCC). Suas áreas de interesse concentram-se em arquitetura de computadores e computação de alto desempenho. **(Texto informado pelo autor)**

- **Comunicação/Dúvidas:** mayk@lncc.br

# Sobre o curso

- Parte 1:
  - **Mais teórica:** por que pequenas mudanças no código/*flags* tem grande impacto na performance?
- Parte 2:
  - **Mais prática:** como eu consigo identificar os gargalos no meu código?

- Repositório: https://github.com/PPG-LNCC-HiagoRocha/PV2026_SD02_II_Profiling_e_otimizacao_em_codigo_C

# Agenda

- Motivação
- Otimização:
  - Estudo de Caso e Ambiente Experimental
  - **Versões 1-3:** *Loops* Aninhados em Python, Java e C
  - **Versão 4:** Ordem dos *Loops*
  - **Versão 5:** *Flags* de Otimização
  - **Versão 6:** *Loops* Paralelos
  - **Versão 7:** *Tiling*
  - **Versão 8:** Vetorização do Compilador
  - **Versão 9:** Intel MKL
- Mais otimizações…

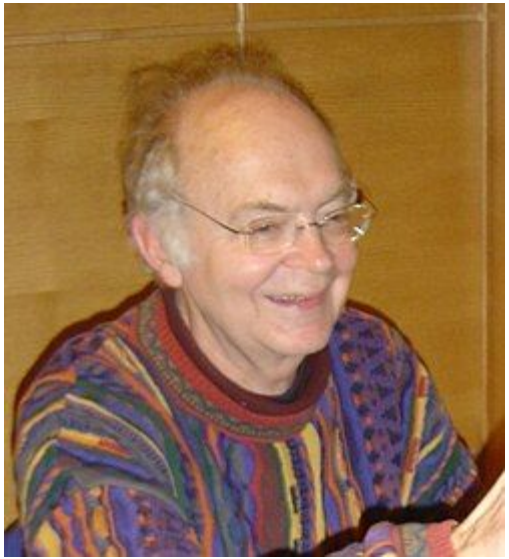
# Agenda

- **Motivação**
- Otimização:
  - Estudo de Caso e Ambiente Experimental
  - **Versões 1-3:** *Loops* Aninhados em Python, Java e C
  - **Versão 4:** Ordem dos *Loops*
  - **Versão 5:** *Flags* de Otimização
  - **Versão 6:** *Loops* Paralelos
  - **Versão 7:** *Tiling*
  - **Versão 8:** Vetorização do Compilador
  - **Versão 9:** Intel MKL
- Mais otimizações…

# Definição de performance/desempenho

- Performance é a relação entre o trabalho realizado e os recursos consumidos para realizá-lo.
- Esses recursos incluem:
  - Tempo
  - CPU
  - Memória
  - Energia
  - Comunicação (I/O, rede)

# Citação de Donald Knuth



**Fonte:** [1]

"***Premature optimization is the root of all evil.***"
(A otimização prematura é a raiz de todo mal.)
— Donald Knuth



[1] Wikipédia — *Donald Knuth*. Disponível em: https://pt.wikipedia.org/wiki/Donald_Knuth Acesso em: jan. 2026.

# Propriedades de *Software*

- Quais as propriedades de *software* mais importantes que performance?

# Propriedades de *Software*

- Quais as propriedades de *software* mais importantes que performance?
  - Compatibilidade, funcionalidade, confiabilidade, correção, manutenibilidade, robustez, clareza, debuggability, modularidade, portabilidade, testabilidade, usabilidade, …

# Propriedades de *Software*

- Quais as propriedades de *software* mais importantes que performance?
  - Compatibilidade, funcionalidade, confiabilidade, correção, manutenibilidade, robustez, clareza, debuggability, modularidade, portabilidade, testabilidade, usabilidade, …

- Se sacrificamos performance, por que estudar performance?

# Propriedades de *Software*

- Quais as propriedades de *software* mais importantes que performance?
  - Compatibilidade, funcionalidade, confiabilidade, correção, manutenibilidade, robustez, clareza, debuggability, modularidade, portabilidade, testabilidade, usabilidade, …

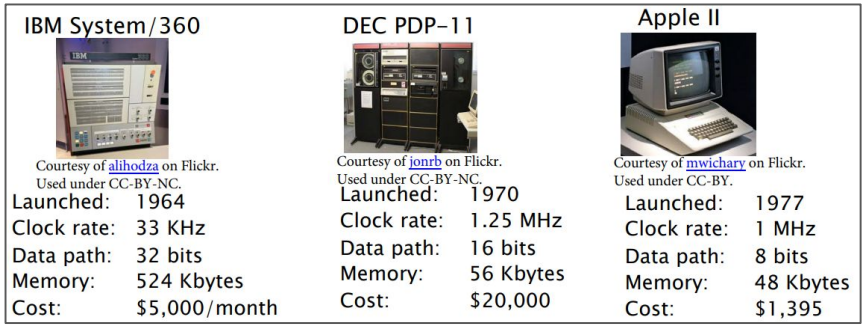
- Se sacrificamos performance, por que estudar performance?
  - Performance é a “moeda” da computação 💰.

# Propriedades de *Software*

- Performance como moeda de troca:
  - **Abstrações** → ganha clareza e modularidade, paga *overhead*
  - **Camadas** → ganha manutenibilidade, paga latência
  - **Verificações** → ganha correção e robustez, paga tempo
  - ***Logs*** → ganha debuggabilidade, paga I/O
  - **Linguagens de alto nível** → ganha produtividade, paga performance

# Primeiros anos da programação de computadores

- Otimização na performance era inseparável da programação, pois os recursos de *hardware* eram tão limitados que muitos programas só existiam graças às otimizações intensivas.



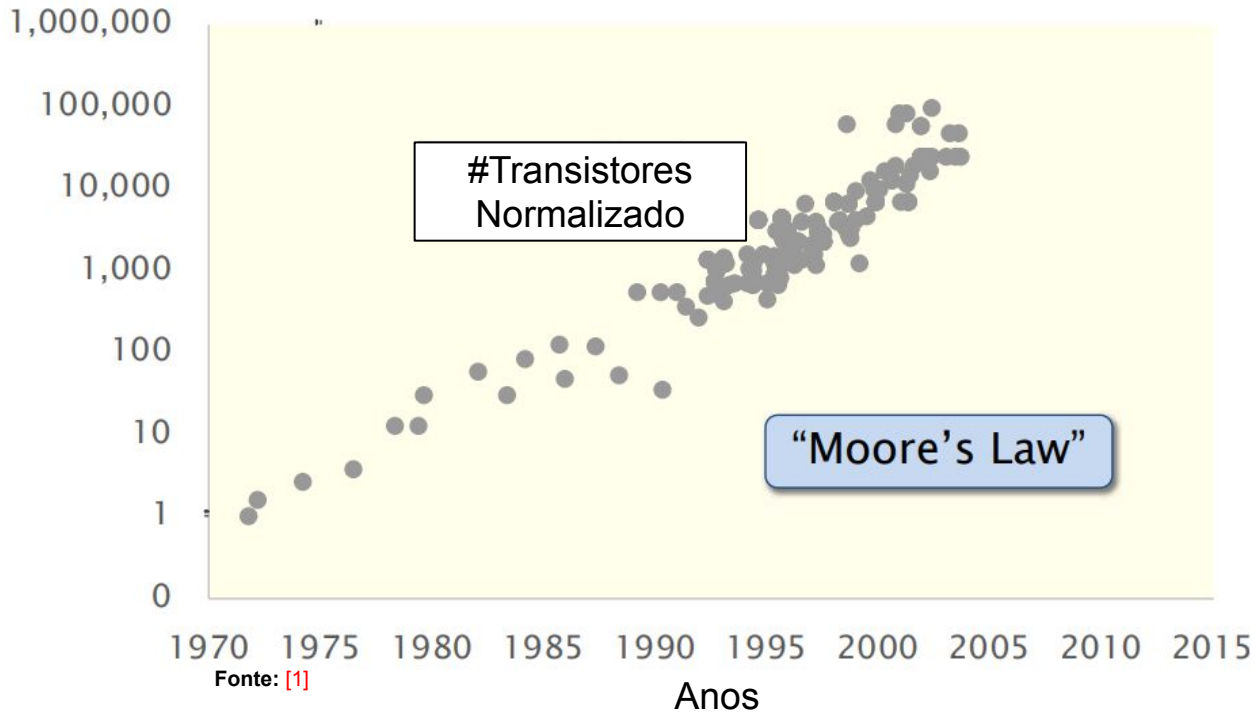

**Fonte:** [1]

- Muitos programas rodavam no limite físico da máquina:
  - O *software* tinham que ser moldado para o *hardware*.
  - Muitos programas poderiam não rodar na máquina sem otimizações intensivas.



[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

# Evolução da tecnologia até 2004

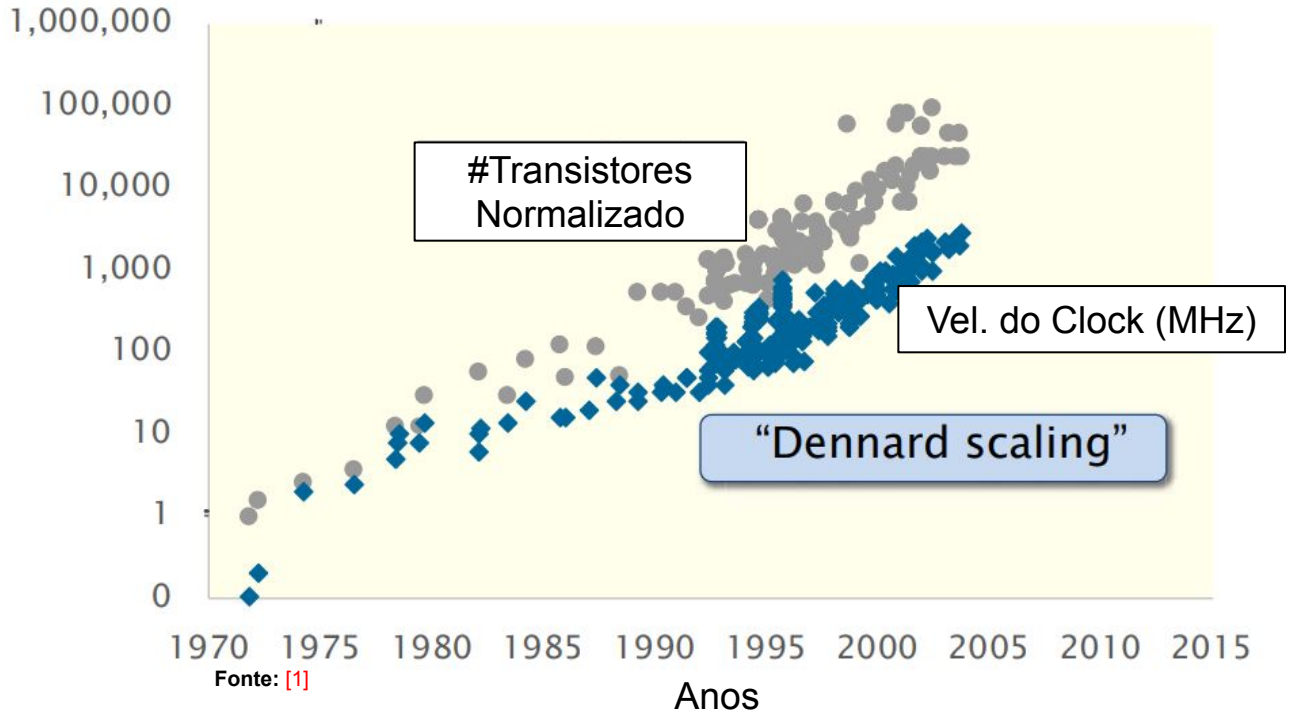



**Fonte:** [1]



[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

# Evolução da tecnologia até 2004



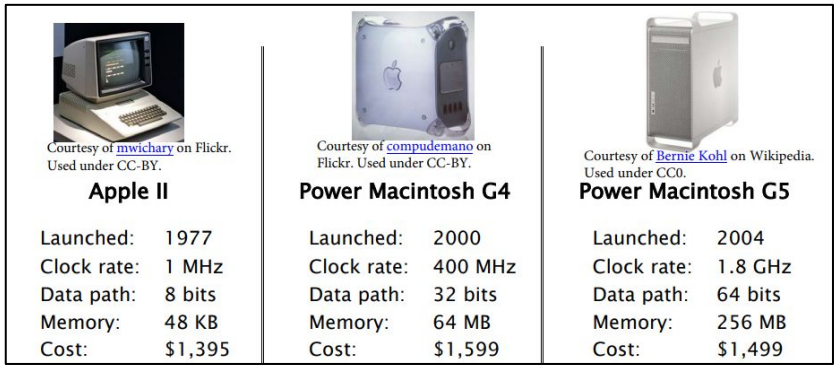

**Fonte:** [1]



[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

# Avanços no *hardware*

- Computadores da Apple com preços similares de 1970 a 2004.



Courtesy of mwichary on Flickr. Used under CC-BY.

**Apple II**

| | |
|---|---|
| Launched: | 1977 |
| Clock rate: | 1 MHz |
| Data path: | 8 bits |
| Memory: | 48 KB |
| Cost: | $1,395 |

Courtesy of compudemano on Flickr. Used under CC-BY.

**Power Macintosh G4**

| | |
|---|---|
| Launched: | 2000 |
| Clock rate: | 400 MHz |
| Data path: | 32 bits |
| Memory: | 64 MB |
| Cost: | $1,599 |

Courtesy of Bernie Kohl on Wikipedia. Used under CC0.

**Power Macintosh G5**

| | |
|---|---|
| Launched: | 2004 |
| Clock rate: | 1.8 GHz |
| Data path: | 64 bits |
| Memory: | 256 MB |
| Cost: | $1,499 |

**Fonte:** [1]

- Mesmo com preços de lançamento semelhantes, um computador de 2003/2004 era várias ordens de magnitude mais capaz do que um da década de 1970.



[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

# Até 2004

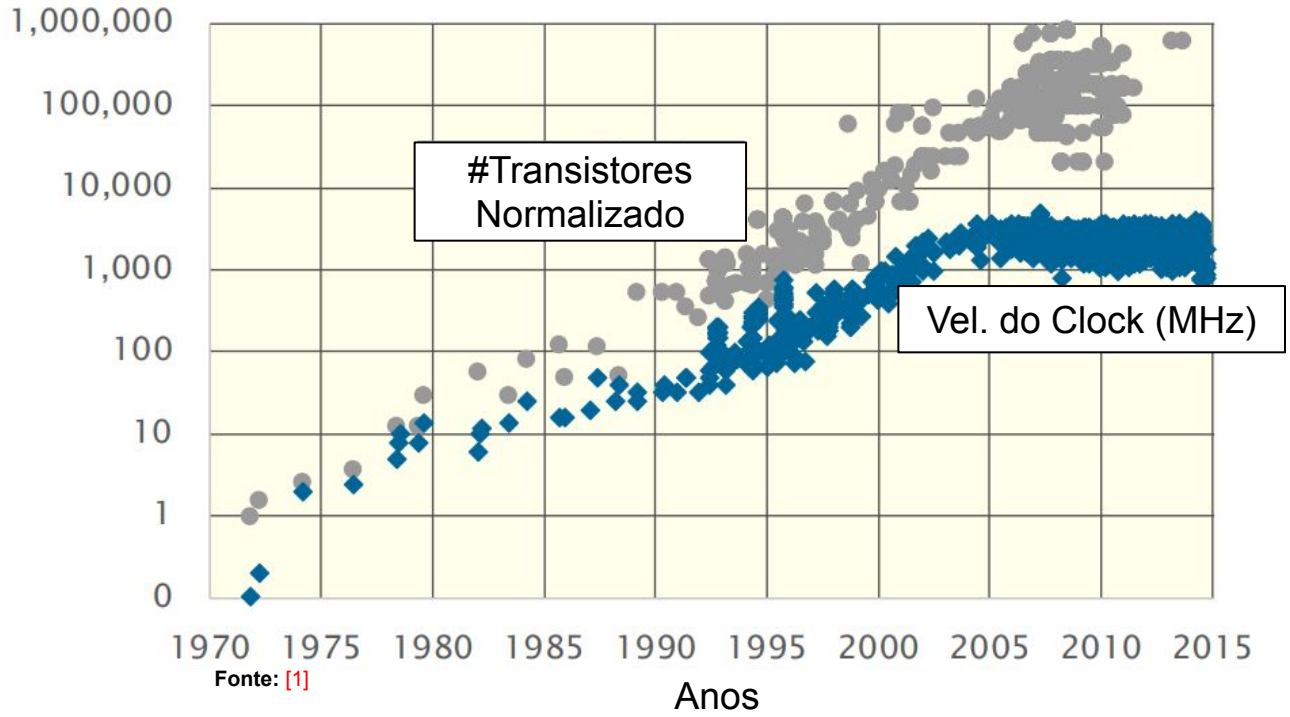
Lei de Moore + Escalonamento da Frequência de Clock (até ~2004)

**=**

“imprensa” de desempenho

- O desempenho crescia quase automaticamente graças ao aumento do número de transistores e da frequência de operação.

# Evolução da tecnologia depois de 2004



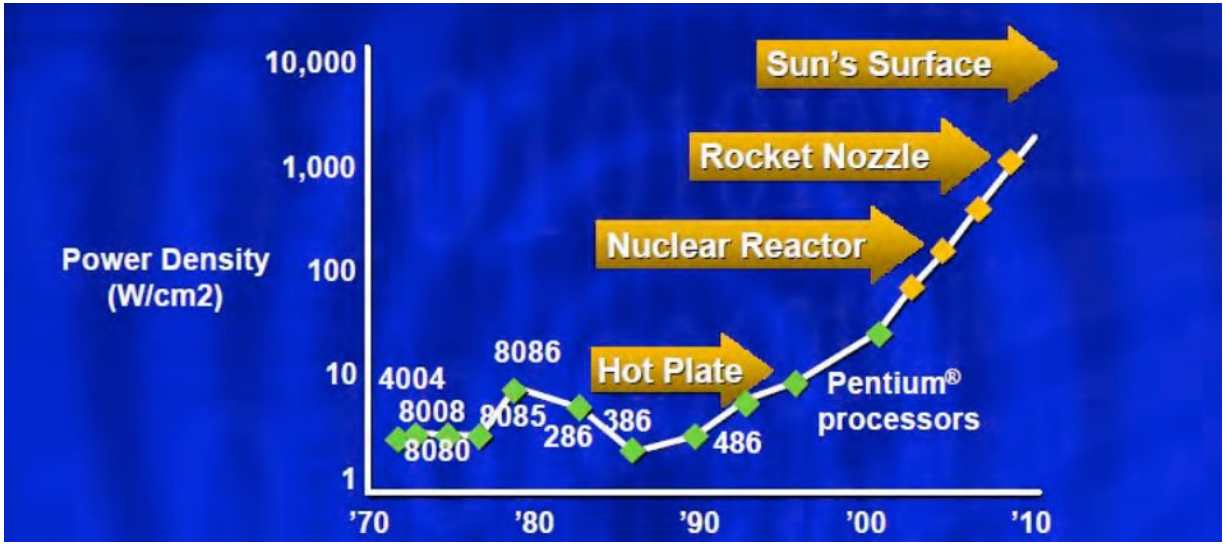

**Fonte:** [1]



[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

# Densidade de Potência

- Se o crescimento da frequência de *clock* (25-30% ao ano) tivesse continuado após 2004, a densidade de potência teria atingido níveis termicamente insustentáveis, estabelecendo o chamado *power wall*.



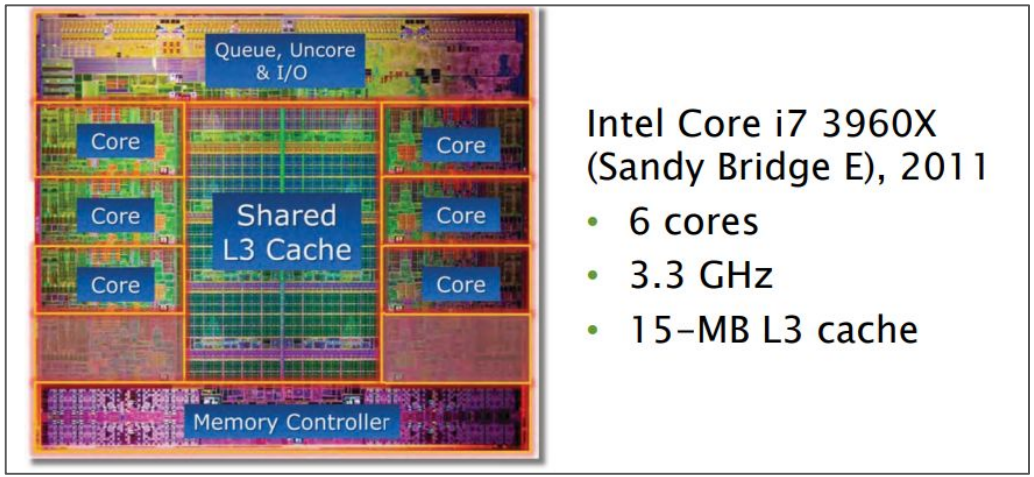

**Fonte:** [1]



[1] Gelsinger, P. Presentation at the Intel Developer's Forum (IDF), Intel Corporation, 2004. Figure reproduced in MIT 6.172 course slides.

# Solução: Multicores

- Os fabricantes mudaram a estratégia para continuar escalando desempenho:
  - **Colocar vários núcleos (*cores*) no mesmo *chip*.**
- Cada geração da Lei de Moore potencialmente dobra o número de *cores*.

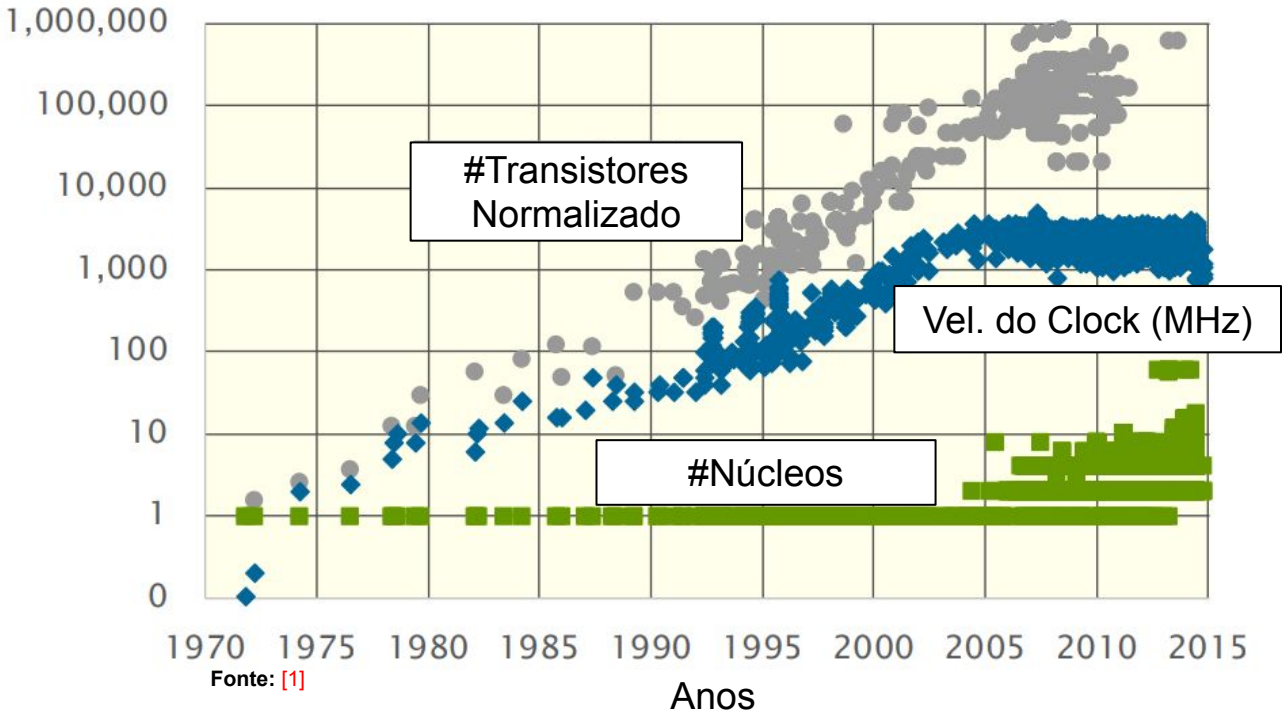



**Fonte:** [1]



[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

# Evolução da tecnologia

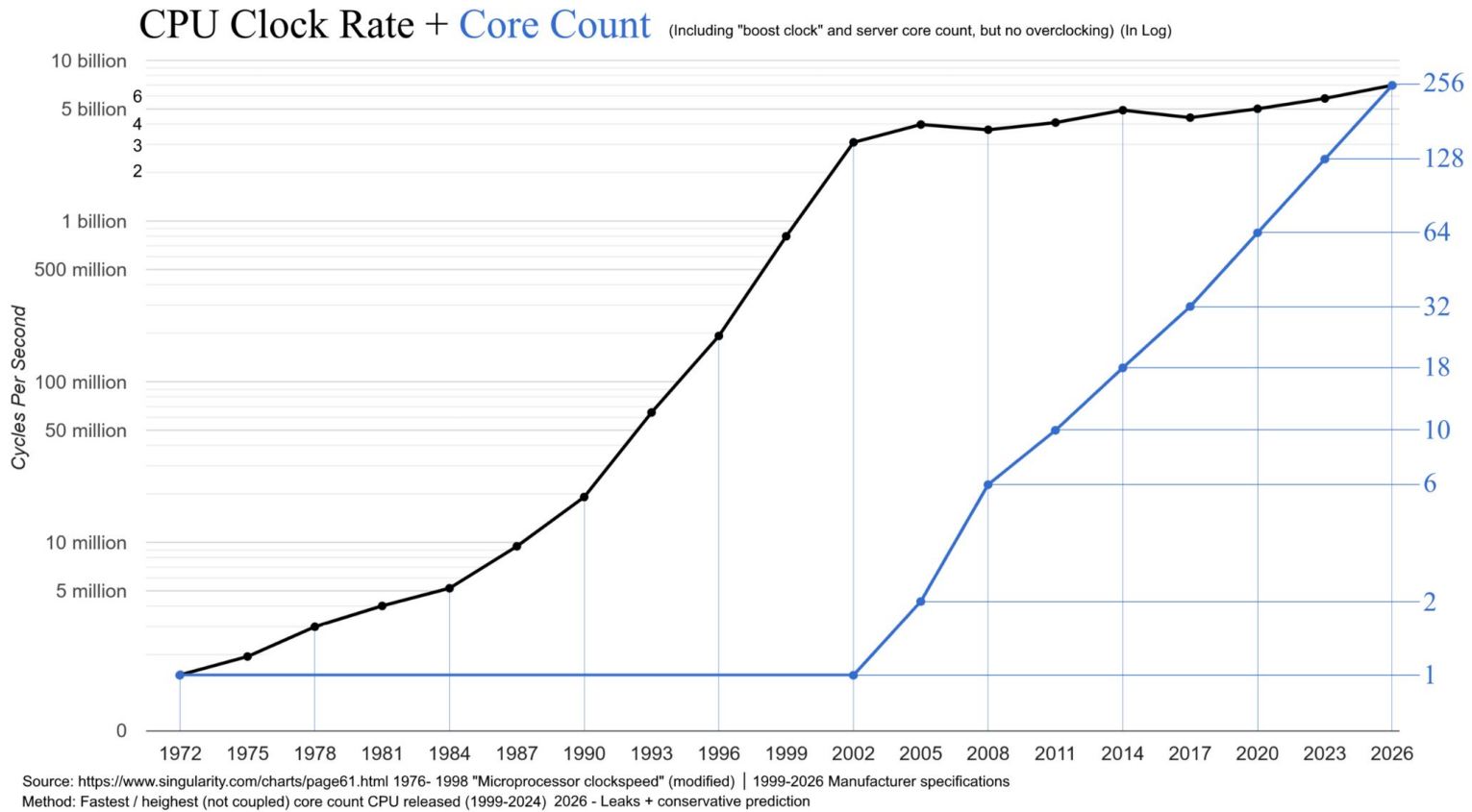



**Fonte:** [1]



[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

# Evolução da tecnologia



# Performance não é mais “gratuito”

- A Lei de Moore ainda entrega transistores, mas o desempenho só aparece quando o *software* é capaz de explorar paralelismo, hierarquia de memória e aceleradores.




# Otimização de ainda é difícil

- Um processador *desktop* multicore moderno contém núcleos de processamento paralelo, unidades vetoriais, caches, prefetchers, GPUs, *hyperthreading*, escalonamento dinâmico de frequência, entre outros recursos.

# Otimização de ainda é difícil

- Um processador *desktop* multicore moderno contém núcleos de processamento paralelo, unidades vetoriais, caches, prefetchers, GPUs, *hyperthreading*, escalonamento dinâmico de frequência, entre outros recursos.

**Como podemos escrever *software* para utilizar o *hardware* moderno de forma eficiente?**

# Agenda

- Motivação
- **Otimização:**
  - **Estudo de Caso e Ambiente Experimental**
  - **Versões 1-3:** *Loops* Aninhados em Python, Java e C
  - **Versão 4:** Ordem dos *Loops*
  - **Versão 5:** *Flags* de Otimização
  - **Versão 6:** *Loops* Paralelos
  - **Versão 7:** *Tiling*
  - **Versão 8:** Vetorização do Compilador
  - **Versão 9:** Intel MKL
- Mais otimizações…

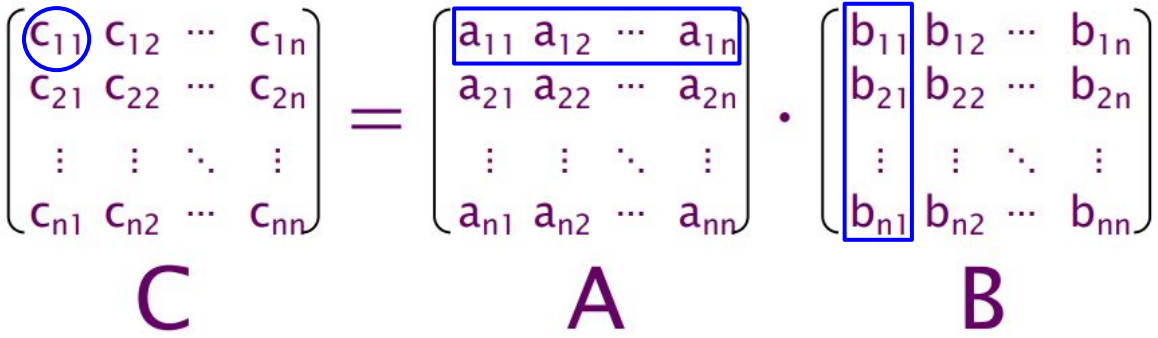
# Multiplicação de matrizes quadradas

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

C A B

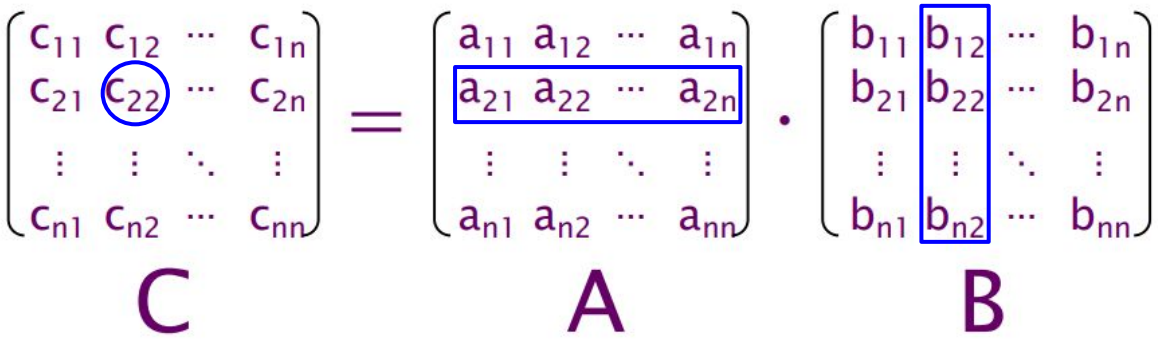
$$C(i,j) = \sum_{k=1}^{N} A(i,k) \cdot B(k,j)$$

# Multiplicação de matrizes quadradas




$$C(i,j) = \sum_{k=1}^{N} A(i,k) \cdot B(k,j)$$

# Multiplicação de matrizes quadradas




$$C(i,j) = \sum_{k=1}^{N} A(i,k) \cdot B(k,j)$$

# Configurações da **sequana_cpu_dev**

| Característica | Especificação |
|---|---|
| Modelo | Intel Xeon Gold 6252 |
| Microarquitetura | Cascade Lake (2ª geração Intel Xeon Scalable) |
| Frequência de clock | 2.10 GHz (Turbo Boost de até ~3.70 GHz) |
| Sockets (chips) | 2 |
| Núcleos de processamento | 24 cores por chip |
| Hyperthreading | 2 threads por core (desabilitado) |
| Unidade de ponto flutuante | FPU vetorial por core, com suporte a AVX (256-bit), AVX2 e AVX-512 (512-bit: 8 doubles ou 16 floats) |
| Tamanho da linha de cache | 64 bytes |
| L1 Instruction Cache (L1i) | 32 KB por core |
| L1 Data Cache (L1d) | 32 KB por core |
| L2 Cache | 1 MB por core |
| L3 Cache (LLC) | 35.75 MB compartilhado |
| DRAM | 376 GiB |

# Configurações da **sequana_cpu_dev**

| Característica | Especificação |
|---|---|
| Modelo | Intel Xeon Gold 6252 |
| Microarquitetura | Cascade Lake (2ª geração Intel Xeon Scalable) |
| Frequência de clock | 2.10 GHz (Turbo Boost de até ~3.70 GHz) |
| Sockets (chips) | 2 |
| Núcleos de processamento | 24 cores por chip |
| Hyperthreading | 2 threads por core (desabilitado) |
| Unidade de ponto flutuante | FPU vetorial por core, com suporte a AVX (256-bit), AVX2 e AVX-512 (512-bit: 8 doubles ou 16 floats) |
| Tamanho da linha de cache | 64 bytes |
| L1 Instruction Cache (L1i) | 32 KB por core |
| L1 Data Cache (L1d) | 32 KB por core |
| L2 Cache | 1 MB por core |
| L3 Cache (LLC) | 35.75 MB compartilhado |
| DRAM | 376 GiB |

$$P_{\text{pico}} = f \times N_{\text{cores}} \times \text{FLOPs/ciclo/core}$$

- f = frequência do *clock* (Hz)
- $N_{cores}$ = número total de cores físicos
- FLOPs/ciclo/core = operações de ponto flutuante que um core executa em um ciclo

# Configurações da **sequana_cpu_dev**

| Característica | Especificação |
|---|---|
| Modelo | Intel Xeon Gold 6252 |
| Microarquitetura | Cascade Lake (2ª geração Intel Xeon Scalable) |
| Frequência de clock | 2.10 GHz (Turbo Boost de até ~3.70 GHz) |
| Sockets (chips) | 2 |
| Núcleos de processamento | 24 cores por chip |
| Hyperthreading | 2 threads por core (desabilitado) |
| Unidade de ponto flutuante | FPU vetorial por core, com suporte a AVX (256-bit), AVX2 e AVX-512 (512-bit: 8 doubles ou 16 floats) |
| Tamanho da linha de cache | 64 bytes |
| L1 Instruction Cache (L1i) | 32 KB por core |
| L1 Data Cache (L1d) | 32 KB por core |
| L2 Cache | 1 MB por core |
| L3 Cache (LLC) | 35.75 MB compartilhado |
| DRAM | 376 GiB |

$$P_{\text{pico}} = f \times N_{\text{cores}} \times \text{FLOPs/ciclo/core}$$

$\text{Pico}_{\text{FP64}} = (2{,}1 \times 10^9) \times 48 \times 32$

$\approx 3{,}23 \times 10^{12}$ FLOPs = **3,23 TFLOPs**

- $f = 2{,}1 \times 10^9$
- $N_{\text{cores}} = 48$ (24 cores × 2 sockets)
- FLOPs/ciclo/core = 32 (8 doubles por AVX-512 × 2 FLOPs por FMA × 2 unidades FMA)

# Configurações da **sequana_cpu_dev**

| Característica | Especificação |
|---|---|
| Modelo | Intel Xeon Gold 6252 |
| Microarquitetura | Cascade Lake (2ª geração Intel Xeon Scalable) |
| Frequência de clock | 2.10 GHz (Turbo Boost de até ~3.70 GHz) |
| Sockets (chips) | 2 |
| Núcleos de processamento | 24 cores por chip |
| Hyperthreading | 2 threads por core (desabilitado) |
| Unidade de ponto flutuante | FPU vetorial por core, com suporte a AVX (256-bit), AVX2 e AVX-512 (512-bit: 8 doubles ou 16 floats) |
| Tamanho da linha de cache | 64 bytes |
| L1 Instruction Cache (L1i) | 32 KB por core |
| L1 Data Cache (L1d) | 32 KB por core |
| L2 Cache | 1 MB por core |
| L3 Cache (LLC) | 35.75 MB compartilhado |
| DRAM | 376 GiB |

$$P_{\text{pico}} = f \times N_{\text{cores}} \times \text{FLOPs/ciclo/core}$$

$\text{Pico}_{\text{FP64}} = (2{,}1\times10^{9}) \times 48 \times 32$

$\approx 3{,}23\times10^{12}$ FLOPs = **3,23 TFLOPs**

- $f = 2{,}1\times10^{9}$
- $N_{\text{cores}} = 48$ (24 cores × 2 sockets)
- FLOPs/ciclo/core = 32 (8 doubles por AVX-512 × 2 FLOPs por FMA × 2 unidades FMA)

$\text{Pico}_{\text{FP32}} = (2{,}1\times10^{9}) \times 48 \times 64$

$\approx 6{,}45\times10^{12}$ FLOPs = **6,45 TFLOPs**

# Agenda

- Motivação
- **Otimização:**
  - Estudo de Caso e Ambiente Experimental
  - **Versões 1-3: *Loops* Aninhados em Python, Java e C**
  - **Versão 4:** Ordem dos *Loops*
  - **Versão 5:** Flags de Otimização
  - **Versão 6:** Loops Paralelos
  - **Versão 7:** Tiling
  - **Versão 8:** Vetorização do Compilador
  - **Versão 9:** Intel MKL
- Mais otimizações…

*Loops* Aninhados

# Versão 1: Python

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
```

*Loops* Aninhados

# Versão 1: Python

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
```

- Tempo de execução (N = 1024) = 201,63 s

*Loops* Aninhados

# Versão 1: Python

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
```

- Tempo de execução (N = 1024) = 201,63 s

Foi rápido?

# Versão 1: Python

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 1 | Python | 201,63 | 1,00 | 0,01065 | 0,00017 |

# Como estimar o %Pico?

$$\%\text{Pico} = \frac{\text{FLOPs do algoritmo}/T}{\text{Pico teórico}} \times 100$$

- FLOPs do algoritmo:
  a. Para cada elemento de C, são realizadas **N multiplicações** + **N-1 somas** = 2N-1 FLOPs

$$C(i,j) = \sum_{k=1}^{N} A(i,k) \cdot B(k,j)$$

  b. Como C tem $N^2$ elementos, temos o custo total é $N^2 \times (2N-1) = 2N^3 - N^2$ FLOPs
  c. Assintoticamente, temos que, **$2N^3$ FLOPs** para valores grande de N

# Como estimar o %Pico?

$$\%\text{Pico} = \frac{\text{FLOPs do algoritmo}/T}{\text{Pico teórico}} \times 100$$

- Para N = 1024:

$$\%\text{Pico} = ( ( 2(\mathbf{2^{10}})^3 \div 201{,}625643 ) \div 6{,}45\times10^{12} ) \times 100$$

$$\approx (1{,}066\times10^{7} \div 6{,}45\times10^{12} ) \times 100$$

$$= \mathbf{0{,}00017\%}$$

# Versão 2: Java

```java
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

# Versão 2: Java

```java
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

N = 1024

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 1 | Python | 201,63 | 1,00 | 0,01065 | 0,00017 |
| 2 | Java | 1,55 | 130,23 | 1,38702 | 0,02150 |

# Versão 2: Java

```java
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

N = 1024

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 1 | Python | 201,63 | 1,00 | 0,01065 | 0,00017 |
| 2 | Java | 1,55 | 130,23 | 1,38702 | 0,02150 |

N = 4096

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 2 | Java | 145,39 | 1,00 | 0,94534 | 0,01466 |

*Loops* Aninhados

# Versão 3: C

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

# Versão 3: C

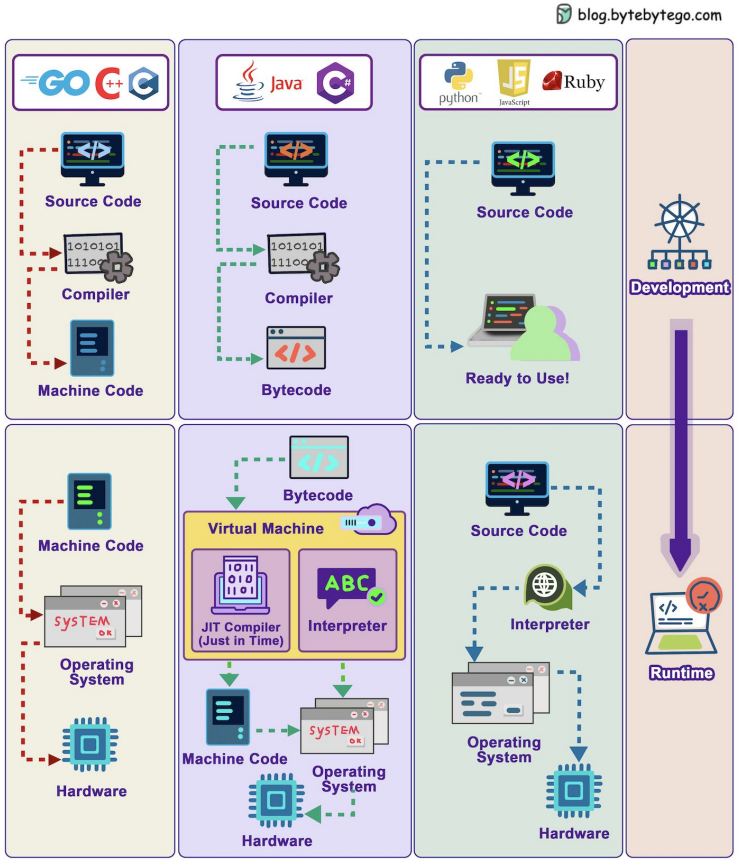
```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

N = 4096

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 2 | Java | 145,39 | 1,00 | 0,94534 | 0,01466 |
| 3 | C (-O3) | 127,10 | 1,14 | 1,08135 | 0,01677 |

# Por que Python é lento e C é rápido?

- Python é interpretado:
  - Código fonte → Interpretador → CPU
- C é compilado:
  - Código fonte → Compilador → Código de máquina
- Java é meio termo:
  - Código fonte → Bytecode → JVM



# Agenda

- Motivação
- **Otimização:**
  - Estudo de Caso e Ambiente Experimental
  - **Versões 1-3:** *Loops* Aninhados em Python, Java e C
  - **Versão 4: Ordem dos *Loops***
  - **Versão 5:** *Flags* de Otimização
  - **Versão 6:** *Loops* Paralelos
  - **Versão 7:** *Tiling*
  - **Versão 8:** Vetorização do Compilador
  - **Versão 9:** Intel MKL
- Mais otimizações…

# Ordem dos *Loops*

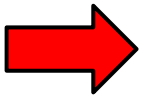
- Nós podemos mudar a ordem dos *loops* sem afetar a corretude do código.

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

# Ordem dos *Loops*

- Nós podemos mudar a ordem dos *loops* sem afetar a corretude do código.

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```



```
for (int i = 0; i < N; i++) {
    for (int k = 0; k < N; k++) {
        for (int j = 0; j < N; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

- A ordem dos *loops* afeta performance?

# Performance em com diferentes ordens

- Sim, a ordem impacta muito no desempenho!

| Ordem | Tempo (s) |
|---|---|
| i, j, **k** | 126,17 |
| i, **k**, j | 20,03 |
| j, i, **k** | 490,82 |
| j, **k**, i | 580,08 |
| **k**, i, j | 21,94 |
| **k**, j, i | 21,83 |

# Performance em com diferentes ordens

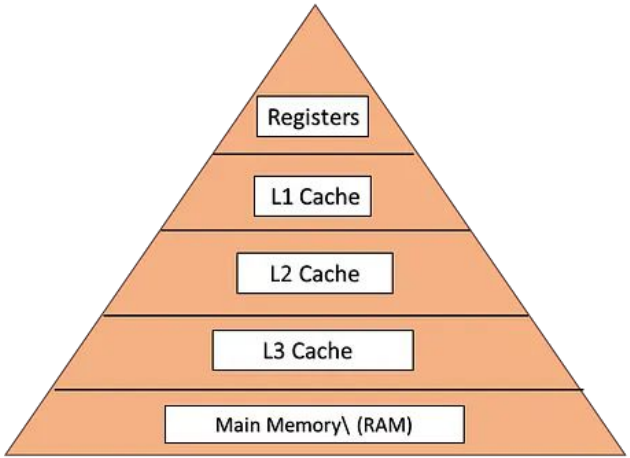
- Sim, a ordem impacta muito no desempenho.
- **Porque?**

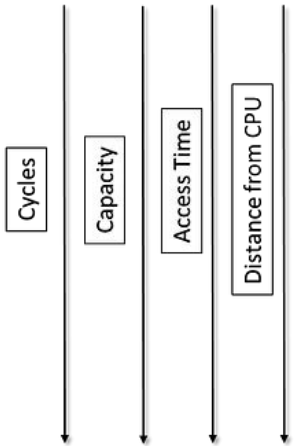
| Ordem | Tempo (s) |
|---|---|
| i, j, **k** | 126,17 |
| i, **k**, j | 20,03 |
| j, i, **k** | 490,82 |
| j, **k**, i | 580,08 |
| **k**, i, j | 21,94 |
| **k**, j, i | 21,83 |

# Performance em com diferentes ordens

- Sim, a ordem impacta muito no desempenho.
- **Porque?**

**Hierarquia de Memória**



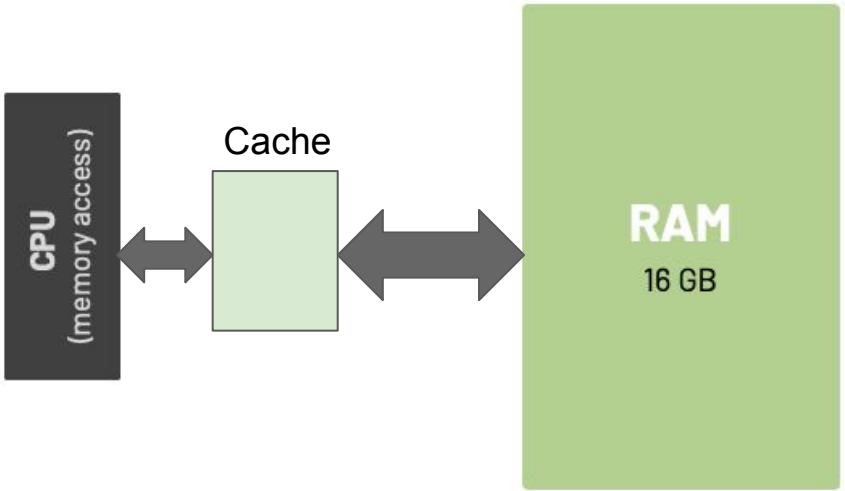

**Fonte:** [1]

| Ordem | Tempo (s) |
|---|---|
| i, j, **k** | 126,17 |
| i, **k**, j | 20,03 |
| j, i, **k** | 490,82 |
| j, **k**, i | 580,08 |
| **k**, i, j | 21,94 |
| **k**, j, i | 21,83 |

[1] CHAUDHARI, Khushboo. *Optimizing matrix multiplication: the impact of loop order in 1D array representation*. Medium, 2023. Disponível em: https://medium.com/@chaudharikhushboo24/optimizing-matrix-multiplication-the-impact-of-loop-order-in-1d-array-representation-4e039d85399d. Acesso em: 15 jan. 2026.

# Memória cache

- Memória cache é uma memória pequena e rápida próxima à CPU que armazena dados usados recentemente para acelerar o acesso à memória principal.




**Fonte:** [1]

[1] PIKUMA. *Understanding computer cache*. Pikuma, s.d. Disponível em: https://pikuma.com/blog/understanding-computer-cache. Acesso em: 15 jan. 2026.

# Princípio de localidade

- **Localidade Espacial:**
  - Endereços de acessos futuros tendem a ser próximos de endereços anteriores
  - **Exemplo:** Códigos e dados agrupados
- **Localidade Temporal:**
  - Posições de memória, uma vez acessadas, tendem a ser acessadas novamente num futuro próximo
  - **Exemplos:** Laços de instruções, acessos a pilha de dados e variáveis

# Exemplo: Princípio de localidade

- Considere os acessos:
  - 1, 2, 3, 4, 5, 6, 7, 6, 7, 492, 493, 494, 6, 7, 6, 7

- Podemos observar:
  - Localidade Espacial
  - Localidade Temporal

# Princípio de localidade - Exemplo

- Considere os acessos:
  - 1, 2, 3, 4, 5, 6, 7, 6, 7, 492, 493, 494, 6, 7, 6, 7

- Podemos observar:
  - **Localidade Espacial**
  - Localidade Temporal

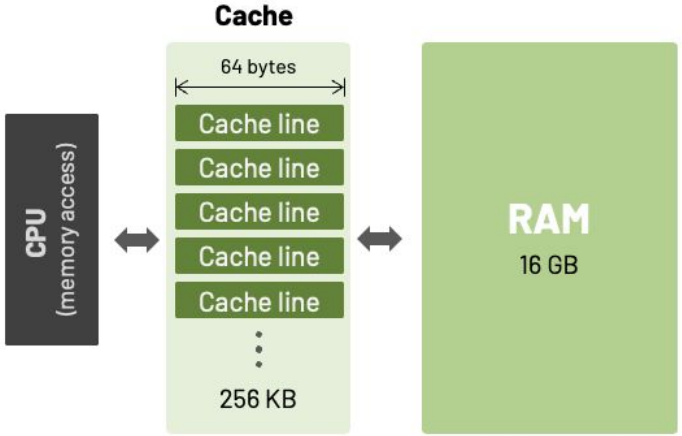
# Princípio de localidade - Exemplo

- Considere os acessos:
  - 1, 2, 3, 4, 5, 6, 7, 6, 7, 492, 493, 494, 6, 7, 6, 7

- Podemos observar:
  - Localidade Espacial
  - **Localidade Temporal**

# Princípio de localidade

- Como explorar os princípios de localidade?
  - **Localidade espacial:**
    - Move **bloco de palavras contíguas** para os níveis da hierarquia mais próximos do processador.
  - **Localidade temporal:**
    - Mantém os dados **mais recentemente acessados** nos níveis da hierarquia mais próxima do processador.

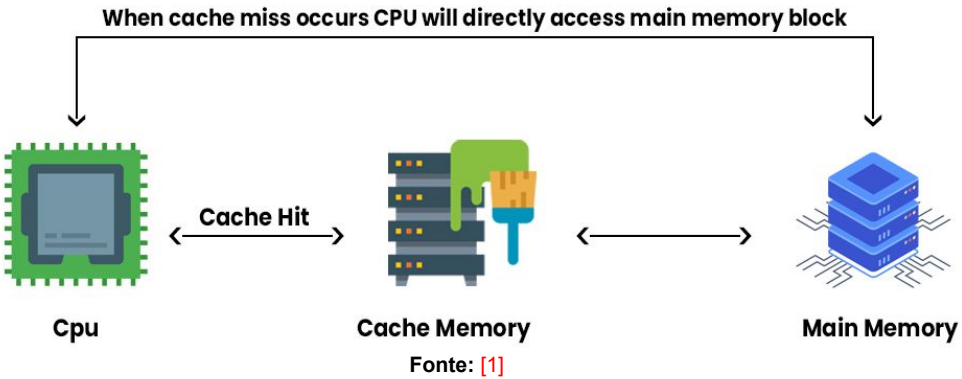



**Fonte:** [1]

[1] PIKUMA. *Understanding computer cache*. Pikuma, s.d. Disponível em: https://pikuma.com/blog/understanding-computer-cache. Acesso em: 15 jan. 2026.

# Acesso a cache

- **Cache *hit*:**
  - Ocorre quando o dado solicitado pela CPU já está na memória cache.
  - Acesso é feito em poucos ciclos de *clock*.
- **Cache *miss*:**
  - Ocorre quando o dado não está na cache.
  - A CPU precisa buscar o dado na memória principal (DRAM).




**Fonte:** [1]

[1] GEEKSFORGEEKS. *Simultaneous and hierarchical cache accesses*. GeeksforGeeks, s.d. Disponível em: https://www.geeksforgeeks.org/computer-organization-architecture/simultaneous-and-hierarchical-cache-accesses/. Acesso em: **15 jan. 2026**.

# Penalidade de Cache

- Custo do atraso causado por um cache *miss*
- Relacionado ao *Average Memory Access Time* (AMAT):
  - Considere caches L1, L2, …, Ln, e, após o último nível, a memória principal.

$$\text{AMAT} = T_{L1} + MR_{L1}\left(T_{L2} + MR_{L2}\left(T_{L3} + \cdots + MR_{L_n} \cdot P_{\text{mem}}\right)\right)$$

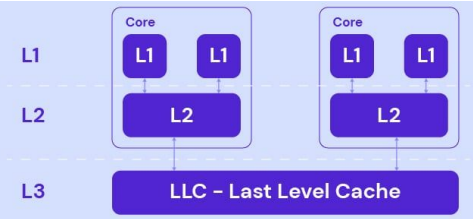
    - $T_{Li}$ = Tempo de acesso (*hit time*) do cache Li
    - $MR_{Li}$ = Taxa de *miss* do cache Li
    - $P_{mem}$ = penalidade/tempo de acesso à memória principal (RAM)

# Penalidade de Cache

- Custo do atraso causado por um cache *miss*
- Relacionado ao *Average Memory Access Time* (AMAT):
  - Considere caches L1, L2, …, Ln, e, após o último nível, a memória principal.

$$\text{AMAT} = T_{L1} + MR_{L1}\left(T_{L2} + MR_{L2}\left(T_{L3} + \cdots + MR_{L_n} \cdot P_{\text{mem}}\right)\right)$$

  - $T_{Li}$ = Tempo de acesso (*hit time*) do cache Li
  - $MR_{Li}$ = Taxa de *miss* do cache Li
  - $P_{mem}$ = penalidade/tempo de acesso à memória principal (RAM)




Fonte: [1]

$$\text{AMAT} = T_{L1} + MR_{L1}\left(T_{L2} + MR_{L2}\left(T_{L3} + MR_{L3} \times P_{mem}\right)\right)$$

[1] HOSTINGER. Cache miss. Hostinger, s.d. Disponível em: https://www.hostinger.com/tutorials/cache-miss. Acesso em: 15 jan. 2026.

# Penalidade de Cache

$$\text{AMAT} = T_{L1} + MR_{L1}\left(T_{L2} + MR_{L2}\left(T_{L3} + MR_{L3} \times P_{mem}\right)\right)$$

- Suponha que:
  - $T_{L1}$ = 4 ciclos, $MR_{L1}$ = 5%
  - $T_{L2}$ = 12 ciclos, $MR_{L2}$ =10%
  - $T_{L3}$ = 40 ciclos, $MR_{L3}$ = 50%
  - $P_{mem}$ = 200 ciclos

$$\text{AMAT} = 4 + 0{,}05\ (12 + 0{,}10\ (\ 40 + 0{,}50 \times 200\ )\ )$$

$$= 4 + 0{,}05\ (\ 12 + 14\ ) = 4 + 1{,}3 = \mathbf{5{,}3\ ciclos}$$

# Penalidade de Cache

$$\text{AMAT} = T_{L1} + MR_{L1}\left(T_{L2} + MR_{L2}\left(T_{L3} + MR_{L3} \times P_{mem}\right)\right)$$

- Suponha que:
  - $T_{L1}$ = 4 ciclos, $MR_{L1}$ = 5%
  - $T_{L2}$ = 12 ciclos, $MR_{L2}$ =10%
  - $T_{L3}$ = 40 ciclos, $MR_{L3}$ = 50%
  - $P_{mem}$ = 200 ciclos

$$\text{AMAT} = 4 + 0{,}05\ (12 + 0{,}10\ (\ 40 + 0{,}50 \times 200\ )\ )$$

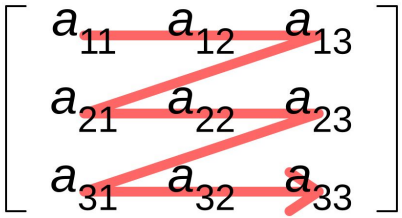
$$= 4 + 0{,}05\ (\ 12 + 14\ ) = 4 + 1{,}3 = \textbf{5,3 ciclos}$$

**Poucos misses já impactam bastante o tempo médio.**

# *Layout* de matrizes na memória

- Em nosso código, a matriz é armazenada em **row-major order**

Row-major order




**Fonte:** [1]

**Memória**

$[a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}]$



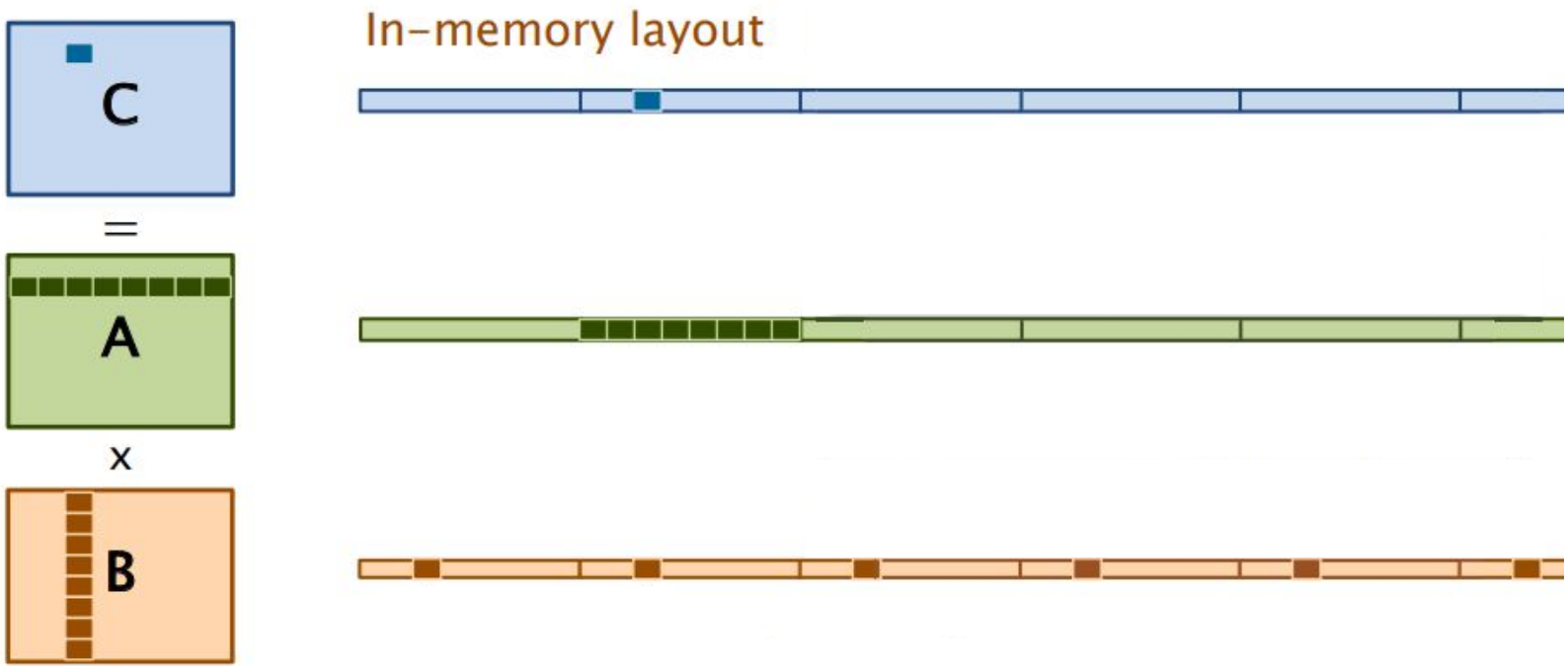
[1] WIKIPEDIA. *Row- and column-major order*. Disponível em: https://en.wikipedia.org/wiki/Row-_and_column-major_order. Acesso em: 21 jan. 2026.

Ordem dos *Loops*

# Padrão de acesso i, j, k

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Tempo (s) = 126,17





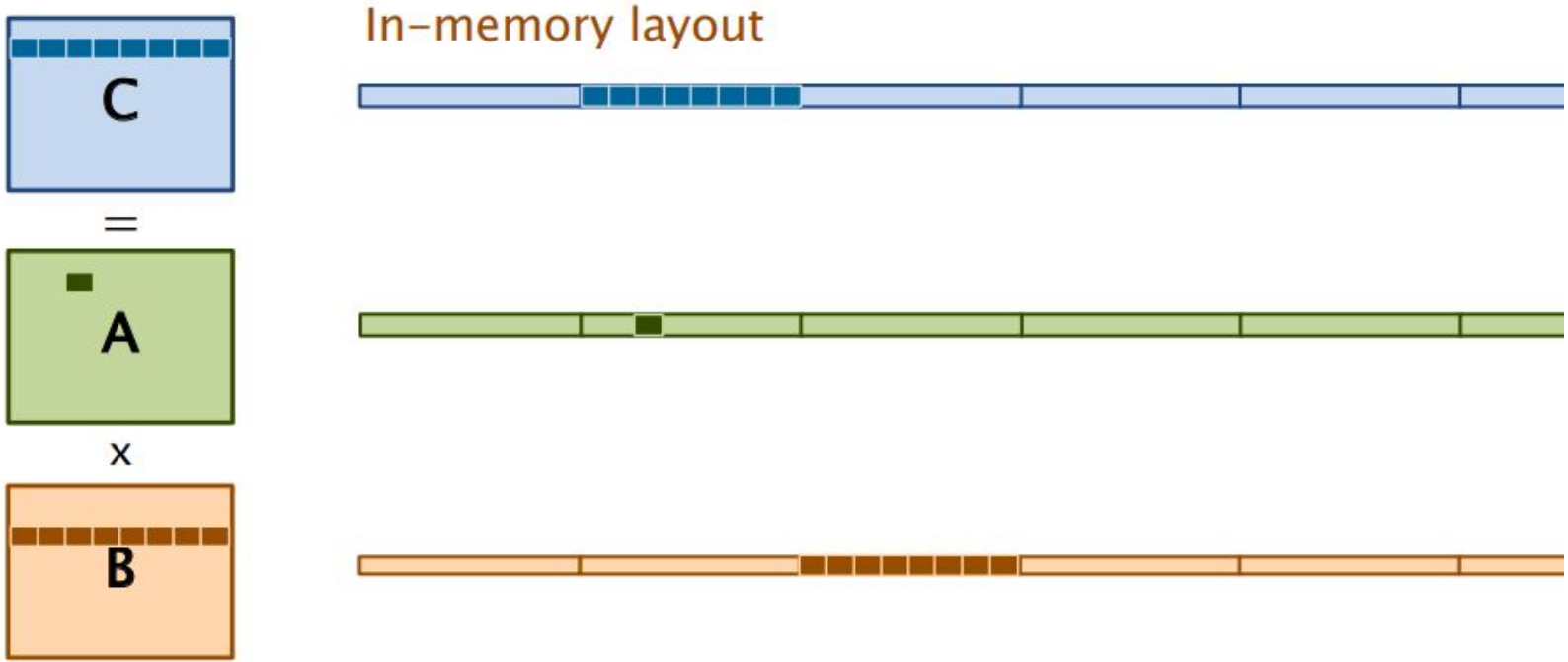

[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

Ordem dos *Loops*

# Padrão de acesso i, k, j

```
for (int i = 0; i < N; i++)
    for (int k = 0; k < N; k++)
        for (int j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```

Tempo (s) = 20,03





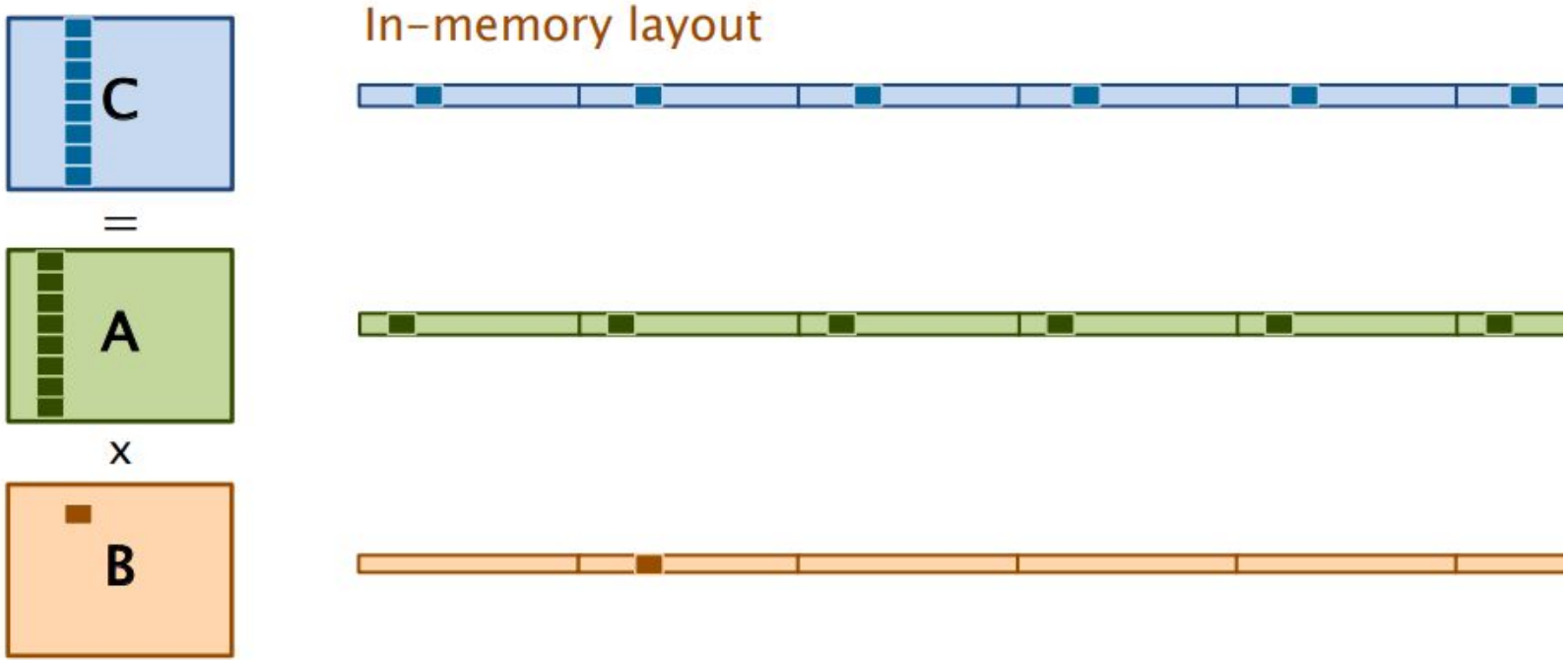

[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

Ordem dos *Loops*

# Padrão de acesso j, k, i

```
for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
        for (int i = 0; i < N; i++)
            C[i][j] += A[i][k] * B[k][j];
```

Tempo (s) = 580,08






[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

# Versão 4: Ordem dos *loops*

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 2 | Java | 145,39 | 1,00 | 0,94534 | 0,01466 |
| 3 | C (-O3) | 127,10 | 1,14 | 1,08135 | 0,01677 |
| 4 | + Ordem dos loops | 20,03 | 7,26 | 6,86025 | 0,10636 |

- **Quais outras otimizações podemos fazer?**

# Agenda

- Motivação
- **Otimização:**
  - Estudo de Caso e Ambiente Experimental
  - **Versões 1-3:** *Loops* Aninhados em Python, Java e C
  - **Versão 4:** Ordem dos *Loops*
  - **Versão 5: *Flags* de Otimização**
  - **Versão 6:** *Loops* Paralelos
  - **Versão 7:** *Tiling*
  - **Versão 8:** Vetorização do Compilador
  - **Versão 9:** Intel MKL
- Mais otimizações…

*Flags* de Otimização

# Otimizações do compilador

- O compilador do GCC oferece algumas opções de *flags* de otimização:
  - **-O0:** Nenhuma otimização
  - **-O1:** Otimizações básicas
  - **-O2:** Otimizações seguras e balanceadas
  - **-O3:** Otimizações agressivas
  - **-Ofast:** Máximo desempenho

- À medida que o nível de otimização aumenta, o compilador aplica transformações cada vez mais agressivas para melhorar o desempenho, podendo sacrificar facilidade de depuração ou precisão numérica.

# Otimizações do compilador

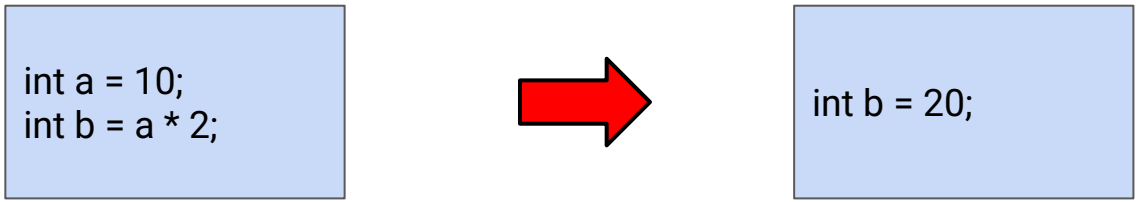
- O compilador do GCC oferece algumas opções de flags de otimização:
  - **-O0:** Nenhuma otimização
  - **-O1:** Otimizações básicas
  - **-O2:** Otimizações seguras e balanceadas
  - **-O3:** Otimizações agressivas
  - **-Ofast:** Máximo desempenho

| Ordem | Tempo (s) |
|---|---|
| -O0 | 169,30 |
| -O1 | 43,14 |
| -O2 | 21,07 |
| -O3 | 20,03 |
| -Ofast | 20,93 |

- À medida que o nível de otimização aumenta, o compilador aplica transformações cada vez mais agressivas para melhorar o desempenho, podendo sacrificar facilidade de depuração ou precisão numérica.

# Otimizações do compilador: -O1

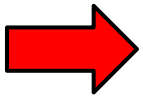
- Remove desperdícios óbvios:
  - Eliminação de código morto
  - Propagação de constantes
  - Simplificação de expressões
- Exemplo: **Simplificação de expressões**



# Otimizações do compilador: -O2

- Melhora bastante o desempenho sem mudar o significado do programa:
  - Uso eficiente de registradores
  - Eliminação de acessos desnecessários à memória
  - Movimento de código invariável para fora do *loop*
- Exemplo: ***Loop* invariável**

```
for (int i = 0; i < N; i++)
    y[i] = x * 2;
```

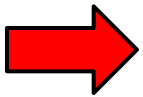


```
int aux = x * 2;
for (int i = 0; i < N; i++)
    y[i] = aux;
```

# Otimizações do compilador: -O3

- Prioriza máximo desempenho computacional:
  - Desenrolamento de *loops*
  - Vetorização automática
  - *Inlining* agressivo
  - Reordenação de instruções
- Exemplo: ***Loop unrolling***

```
for (int i = 0; i < 8; i++)
    a[i] += b[i];
```

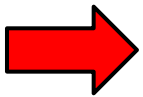


```
a[0] += b[0];
a[1] += b[1];
...
a[7] += b[7];
```

# Otimizações do compilador: -Ofast

- Vai além do -O3, relaxando regras do padrão IEEE 754:
  - Reordena operações de ponto flutuante
  - Assume ausência de NaN/Infs
  - Usa FMA agressivamente
- Exemplo: **Reordenação de Operações**

```
double x = a + b + c;
```



```
double x = (a + b) + c;
ou
double x = a + (b + c);
```

**Resultado numericamente equivalente, mas não idêntico bit a bit.**

# O compilador não faz milagre!

- O compilador otimiza como o código executa, não o algoritmo.
- Cabe ao programador:
  - Escolher a complexidade assintótica
  - Definir a ordem lógica dos *loops*
  - Projetar a estrutura de dados e o *layout* de memória
  - Explorar paralelismo
  - Usar eficientemente a hierarquia de memória
  - Garantir precisão numérica e estabilidade
  - Selecionar bibliotecas otimizadas

# Versão 5: *Flags* de otimização

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 2 | Java | 145,39 | 1,00 | 0,94534 | 0,01466 |
| 3 | C (-O3) | 127,10 | 1,14 | 1,08135 | 0,01677 |
| 4 | + Ordem dos loops | 20,03 | 7,26 | 6,86025 | 0,10636 |
| 5 | + Flags de otimização | 20,03 | 7,26 | 6,86025 | 0,10636 |

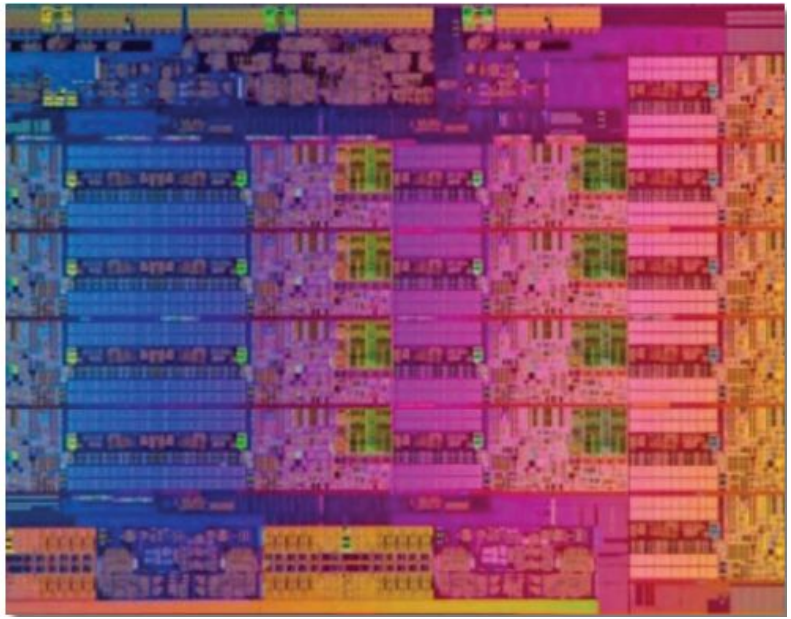
- Com um código simples e otimização do compilador, nós alcançamos apenas **0,1%** do pico de performance da máquina.
- **O que está causando o baixo desempenho?**

# Agenda

- Motivação
- **Otimização:**
  - Estudo de Caso e Ambiente Experimental
  - **Versões 1-3:** *Loops* Aninhados em Python, Java e C
  - **Versão 4:** Ordem dos *Loops*
  - **Versão 5:** *Flags* de Otimização
  - **Versão 6: *Loops* Paralelos**
  - **Versão 7:** *Tiling*
  - **Versão 8:** Vetorização do Compilador
  - **Versão 9:** Intel MKL
- Mais otimizações…

# Paralelismo dos multicores

- Nós estamos usando apenas **1** dos **48** cores disponíveis em nossa máquina.



**Fonte:** [1]

| Característica | Especificação |
|---|---|
| Modelo | Intel Xeon Gold 6252 |
| Microarquitetura | Cascade Lake (2ª geração Intel Xeon Scalable) |
| Frequência de clock | 2.10 GHz (Turbo Boost de até ~3.70 GHz) |
| Sockets (chips) | 2 |
| Núcleos de processamento | 24 cores por chip |
| Hyperthreading | 2 threads por core (desabilitado) |



[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

# *Loops* paralelos

- OpenMP oferece o **#pragma omp parallel for** que permitem as iterações de um *loop* executar em paralelo.
- **Qual *loop* paralelizar?**

```
for (int i = 0; i < N; i++)
    for (int k = 0; k < N; k++)
        for (int j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```

# *Loops* paralelos

- OpenMP oferece o **#pragma omp parallel for** que permitem as iterações de um *loop* executar em paralelo.
- **Qual *loop* paralelizar?**

**#pragma parallel for**

```
for (int i = 0; i < N; i++)
    for (int k = 0; k < N; k++)
        for (int j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```

# Versão 6: *Loops* paralelos

N = 4096

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 2 | Java | 145,39 | 1,00 | 0,94534 | 0,01466 |
| 3 | C (-O3) | 127,10 | 1,14 | 1,08135 | 0,01677 |
| 4 | + Ordem dos loops | 20,03 | 7,26 | 6,86025 | 0,10636 |
| 5 | + Flags de otimização | 20,03 | 7,26 | 6,86025 | 0,10636 |
| 6 | Loops paralelos | 1,24 | 117,20 | 110,79426 | 1,71774 |

- Paralelizando, nós melhoramos em **16,15x** com relação à versão anterior.
- Contudo, alcançamos apenas **1,7%** do pico.
- **O que podemos fazer para melhorar?**

# Versão 6: *Loops* paralelos

N = 4096

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 2 | Java | 145,39 | 1,00 | 0,94534 | 0,01466 |
| 3 | C (-O3) | 127,10 | 1,14 | 1,08135 | 0,01677 |
| 4 | + Ordem dos loops | 20,03 | 7,26 | 6,86025 | 0,10636 |
| 5 | + Flags de otimização | 20,03 | 7,26 | 6,86025 | 0,10636 |
| 6 | Loops paralelos | 1,24 | 117,20 | 110,79426 | 1,71774 |

N = 16384

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 6 | Loops paralelos | 58,32 | 1,00 | 150,81934 | 2,33828 |

# Agenda

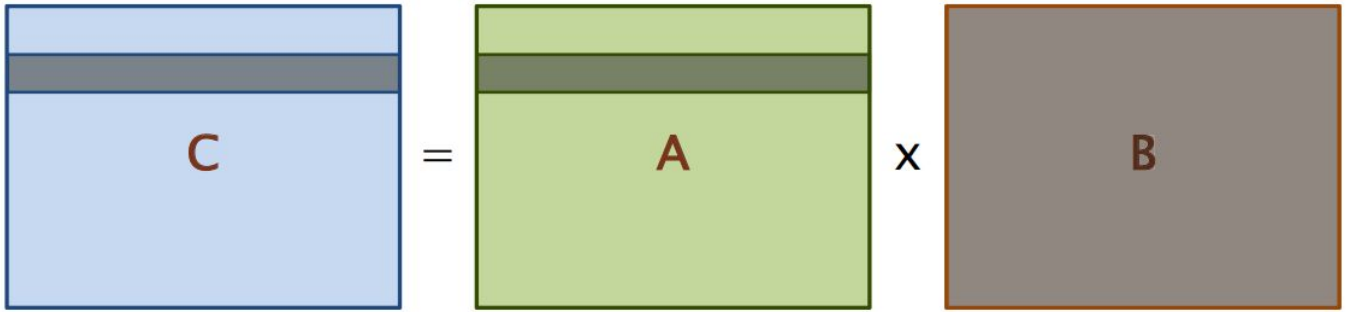
- Motivação
- **Otimização:**
  - Estudo de Caso e Ambiente Experimental
  - **Versões 1-3:** *Loops* Aninhados em Python, Java e C
  - **Versão 4:** Ordem dos *Loops*
  - **Versão 5:** *Flags* de Otimização
  - **Versão 6:** *Loops* Paralelos
  - **Versão 7: *Tiling***
  - **Versão 8:** Vetorização do Compilador
  - **Versão 9:** Intel MKL
- Mais otimizações…

# Localidade na cache

- O desempenho real não é determinado por quantas operações você executa, mas por quantas vezes você precisa acessar a memória para executá-las.
- **Ideia:**
  - Reestruturar a computação para reutilizar os dados na cache o máximo possível:
    - Cache *misses* são lentos
    - Cache *hits* são rápidos
    - Use ao máximo os dados que já estão na cache

# Reuso de dados em *loops*

- Quantos acessos à memória o nosso código deve realizar para computar 1 linha de C?
  - 16.384 * 16.384 = 268.435.456 leituras de C
  - 16.384 * 16.384 = 268.435.456 escritas em C
  - 16.384 * 1 = 16.384 lidas de A
  - 16.384 * 16.384 = 268.435.456 lidas de B
  - Total de **805.306.752** acessos à memória



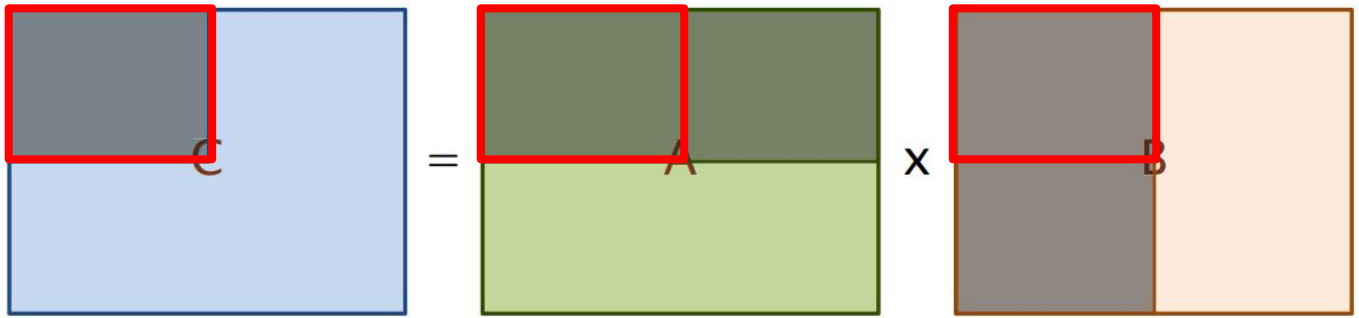

**Fonte:** [1]



[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

# Reuso de dados em blocos

- E se computarmos um bloco de 128×128 de C (acumulando sobre todos os blocos de K)?
  - (128 * 128) * (16.384 ÷ 128) = 2.097.152 leituras de C
  - (128 * 128) * (16.384 ÷ 128) = 2.097.152 escritas em C
  - (128 * 128) * (16.384 ÷ 128) = 2.097.152 leituras de A
  - (128 * 128) * (16.384 ÷ 128) = 2.097.152 leituras de B
  - Total de **8.388.608** acessos à memória






[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

*Tiling*

# Código

```
for (int ii = 0; ii < N; ii += TILE) {
    for (int kk = 0; kk < N; kk += TILE) {
        for (int jj = 0; jj < N; jj += TILE) {

            int i_max = (ii + TILE < N) ? ii + TILE : N;
            int k_max = (kk + TILE < N) ? kk + TILE : N;
            int j_max = (jj + TILE < N) ? jj + TILE : N;

            for (int i = ii; i < i_max; i++) {
                for (int k = kk; k < k_max; k++) {
                    float aik = A[i*N + k];
                    for (int j = jj; j < j_max; j++) {
                        C[i*N + j] += aik * B[k*N + j];
```

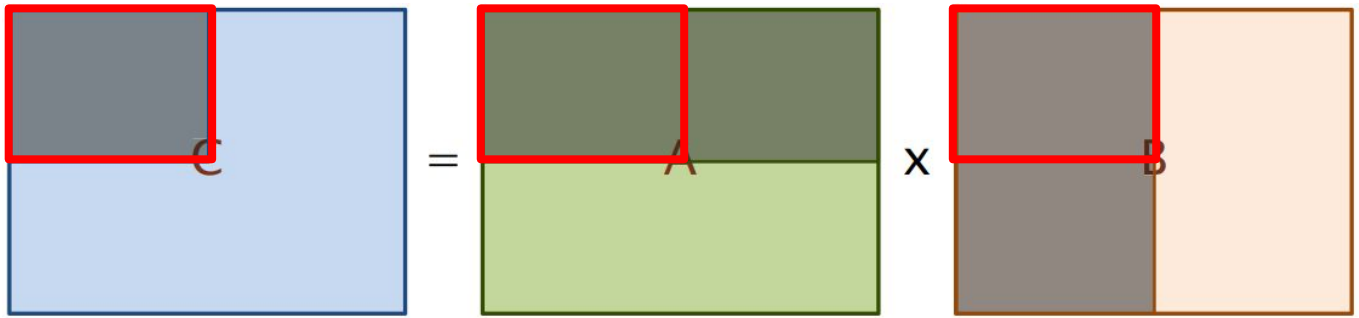
*Tiling*

# Código

```
for (int ii = 0; ii < N; ii += TILE) {
    for (int kk = 0; kk < N; kk += TILE) {
        for (int jj = 0; jj < N; jj += TILE) {

            int i_max = (ii + TILE < N) ? ii + TILE : N;
            int k_max = (kk + TILE < N) ? kk + TILE : N;
            int j_max = (jj + TILE < N) ? jj + TILE : N;

            for (int i = ii; i < i_max; i++) {
                for (int k = kk; k < k_max; k++) {
                    float aik = A[i*N + k];
                    for (int j = jj; j < j_max; j++) {
                        C[i*N + j] += aik * B[k*N + j];
```






[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

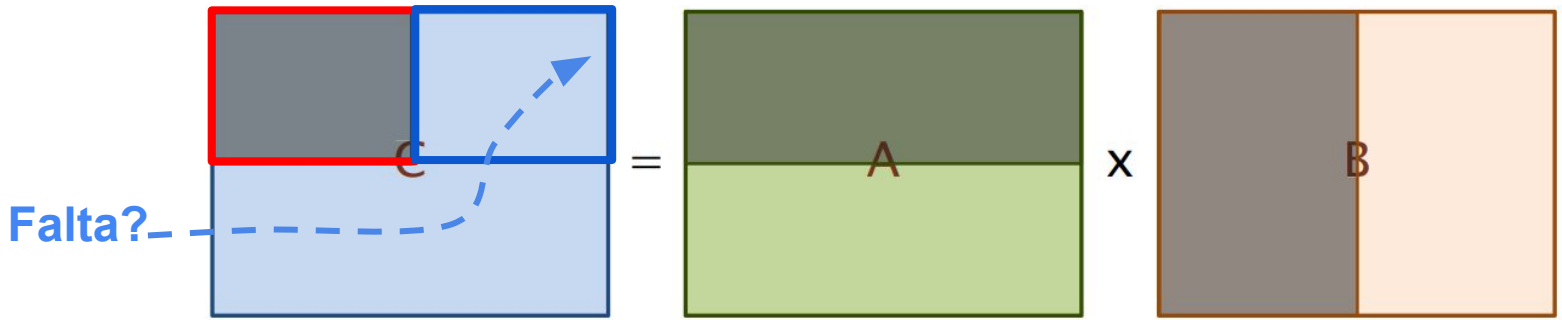
*Tiling*

# Código

```
for (int ii = 0; ii < N; ii += TILE) {
    for (int kk = 0; kk < N; kk += TILE) {
        for (int jj = 0; jj < N; jj += TILE) {

            int i_max = (ii + TILE < N) ? ii + TILE : N;
            int k_max = (kk + TILE < N) ? kk + TILE : N;
            int j_max = (jj + TILE < N) ? jj + TILE : N;

            for (int i = ii; i < i_max; i++) {
                for (int k = kk; k < k_max; k++) {
                    float aik = A[i*N + k];
                    for (int j = jj; j < j_max; j++) {
                        C[i*N + j] += aik * B[k*N + j];
```






[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

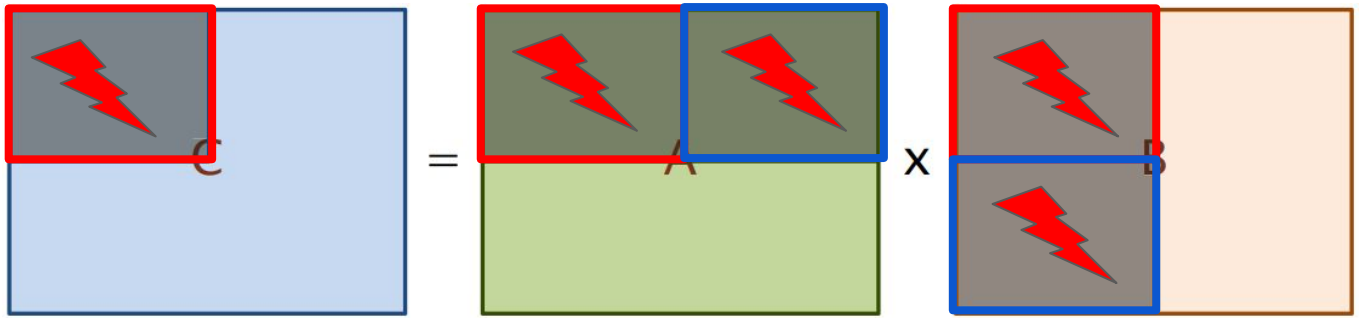
*Tiling*

# Código

```
for (int ii = 0; ii < N; ii += TILE) {
    for (int kk = 0; kk < N; kk += TILE) {
        for (int jj = 0; jj < N; jj += TILE) {

            int i_max = (ii + TILE < N) ? ii + TILE : N;
            int k_max = (kk + TILE < N) ? kk + TILE : N;
            int j_max = (jj + TILE < N) ? jj + TILE : N;

            for (int i = ii; i < i_max; i++) {
                for (int k = kk; k < k_max; k++) {
                    float aik = A[i*N + k];
                    for (int j = jj; j < j_max; j++) {
                        C[i*N + j] += aik * B[k*N + j];
```






[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

*Tiling*

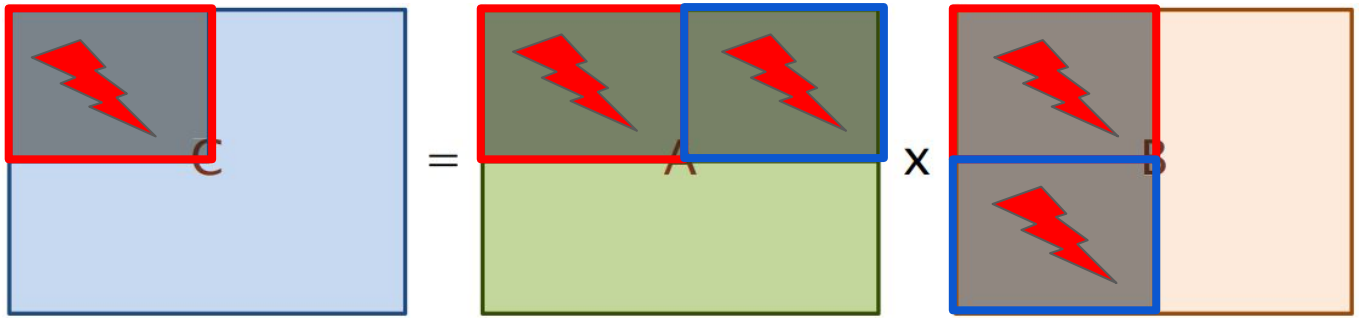
# Código

```
for (int ii = 0; ii < N; ii += TILE) {
    for (int kk = 0; kk < N; kk += TILE) {
        for (int jj = 0; jj < N; jj += TILE) {

            int i_max = (ii + TILE < N) ? ii + TILE : N;
            int k_max = (kk + TILE < N) ? kk + TILE : N;
            int j_max = (jj + TILE < N) ? jj + TILE : N;

            for (int i = ii; i < i_max; i++) {
                for (int k = kk; k < k_max; k++) {
                    float aik = A[i*N + k];
                    for (int j = jj; j < j_max; j++) {
                        C[i*N + j] += aik * B[k*N + j];
```

Como definir?






[1] MIT OpenCourseWare — *6.172 Performance Engineering of Software Systems.* Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.

*Tiling*

# Código

```
for (int ii = 0; ii < N; ii += TILE) {
    for (int kk = 0; kk < N; kk += TILE) {
        for (int jj = 0; jj < N; jj += TILE) {

            int i_max = (ii + TILE < N) ? ii + TILE : N;
            int k_max = (kk + TILE < N) ? kk + TILE : N;
            int j_max = (jj + TILE < N) ? jj + TILE : N;

            for (int i = ii; i < i_max; i++) {
                for (int k = kk; k < k_max; k++) {
                    float aik = A[i*N + k];
                    for (int j = jj; j < j_max; j++) {
                        C[i*N + j] += aik * B[k*N + j];
```

Como definir?

| Tamanho | Tempo (s) |
|---|---|
| 4 | 114,22 |
| 8 | 65,26 |
| 16 | 43,45 |
| 32 | 53,71 |
| 64 | 48,44 |
| 128 | 44,15 |

# Nosso cenário

- Hierarquia de caches:
  - **L1** = 32 KB por core
  - **L2** = 1 MB por core
  - **LLC** = 35,75 MB (compartilhado)
- Dados em float (4 bytes) e 48 threads OpenMP
- Para cada tile/bloco, o kernel precisa manter simultaneamente:

  $A + B + C \Rightarrow 3 \times TILE^2 \times 4$ bytes na cache

# Nosso cenário

- Impacto do tamanho do tile:
  - 4×4 = 192 bytes → blocos minúsculos, *overhead* dominante
  - 8×8 = 768 bytes → cache subutilizada, pouco reuso
  - 16×16 = 3 KB → reuso intenso antes de qualquer evicção (ótimo)
  - 32×32 = 12 KB → tiles médios começam a pressionar a L1
  - 64×64 = 48 KB → blocos não cabem mais na L1
  - 128×128 = 192 KB → blocos não cabem na L1 e pressionam a L2/LLC

| Tamanho | Tempo (s) |
|---|---|
| 4 | 114,22 |
| 8 | 65,26 |
| 16 | 43,45 |
| 32 | 53,71 |
| 64 | 48,44 |
| 128 | 44,15 |

# Versão 7: *Tiling*

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 6 | Loops paralelos | 58,32 | 1,00 | 150,81934 | 2,33828 |
| 7 | + Tiling | 43,45 | 1,34 | 202,44042 | 3,13861 |

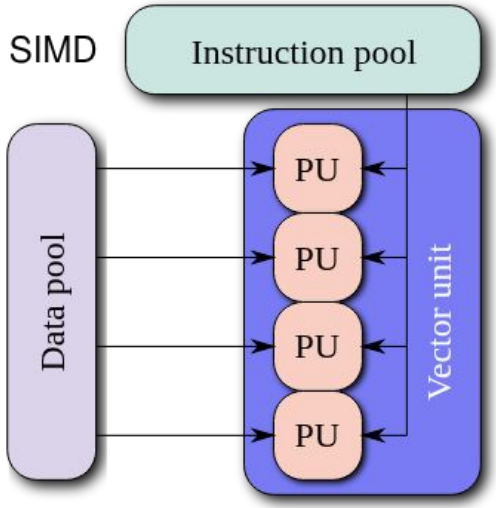
- Ainda estamos usando só **3,1%** do pico.
- **O que otimizar mais?**

# Agenda

- Motivação
- **Otimização:**
  - Estudo de Caso e Ambiente Experimental
  - **Versões 1-3:** *Loops* Aninhados em Python, Java e C
  - **Versão 4:** Ordem dos *Loops*
  - **Versão 5:** *Flags* de Otimização
  - **Versão 6:** *Loops* Paralelos
  - **Versão 7:** *Tiling*
  - **Versão 8: Vetorização do Compilador**
  - **Versão 9:** Intel MKL
- Mais otimizações…

# *Hardware* vetorial

- Microprocessadores modernos possuem *hardware* vetorial para processar dados no modelo de *Single Instruction, Multiple Data* (SIMD).

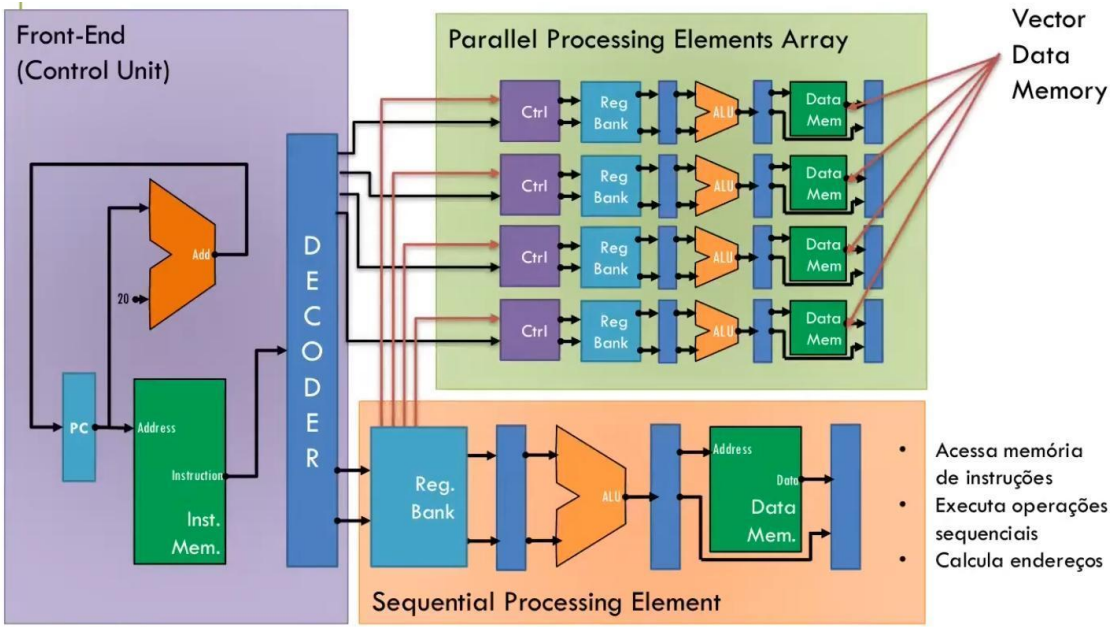



**Fonte:** [1]



[1] **WIKIPEDIA.** *Single instruction, multiple data*. Disponível em: https://en.wikipedia.org/wiki/Single_instruction%2C_multiple_data. Acesso em: 19 jan. 2026.

# *Hardware* vetorial

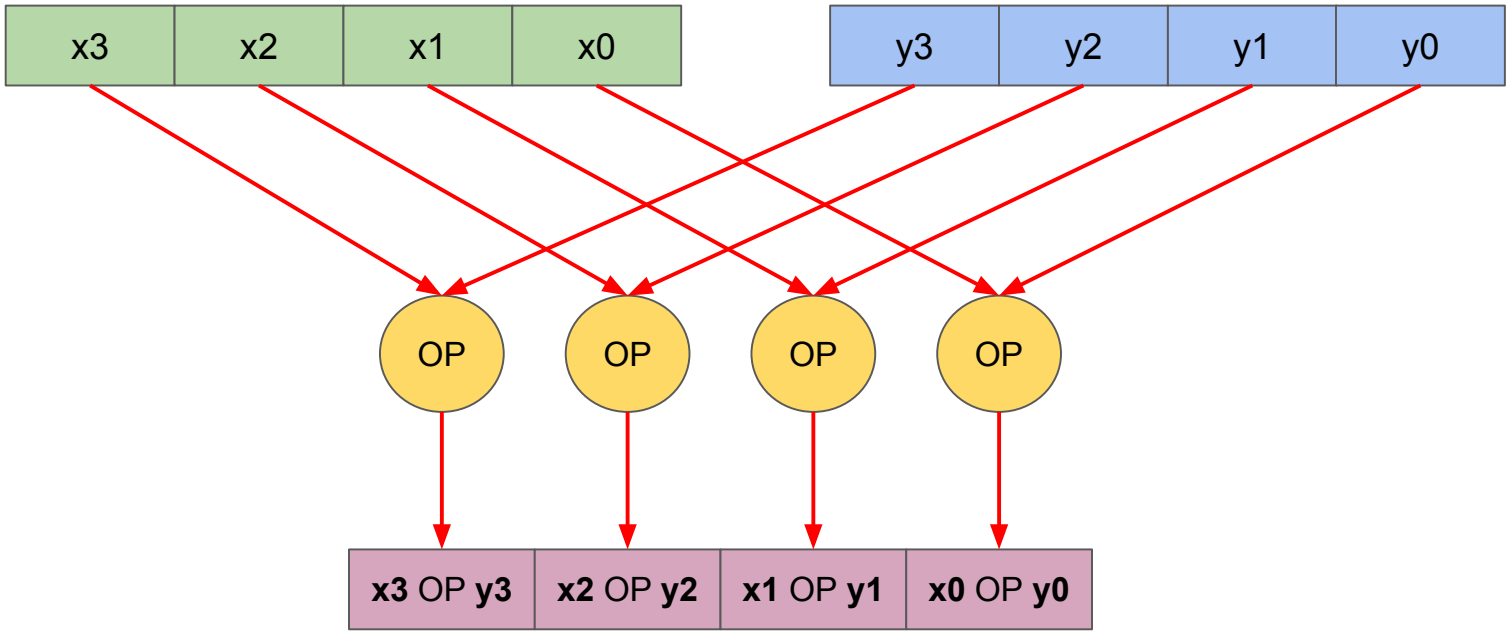



**Fonte:** [1]

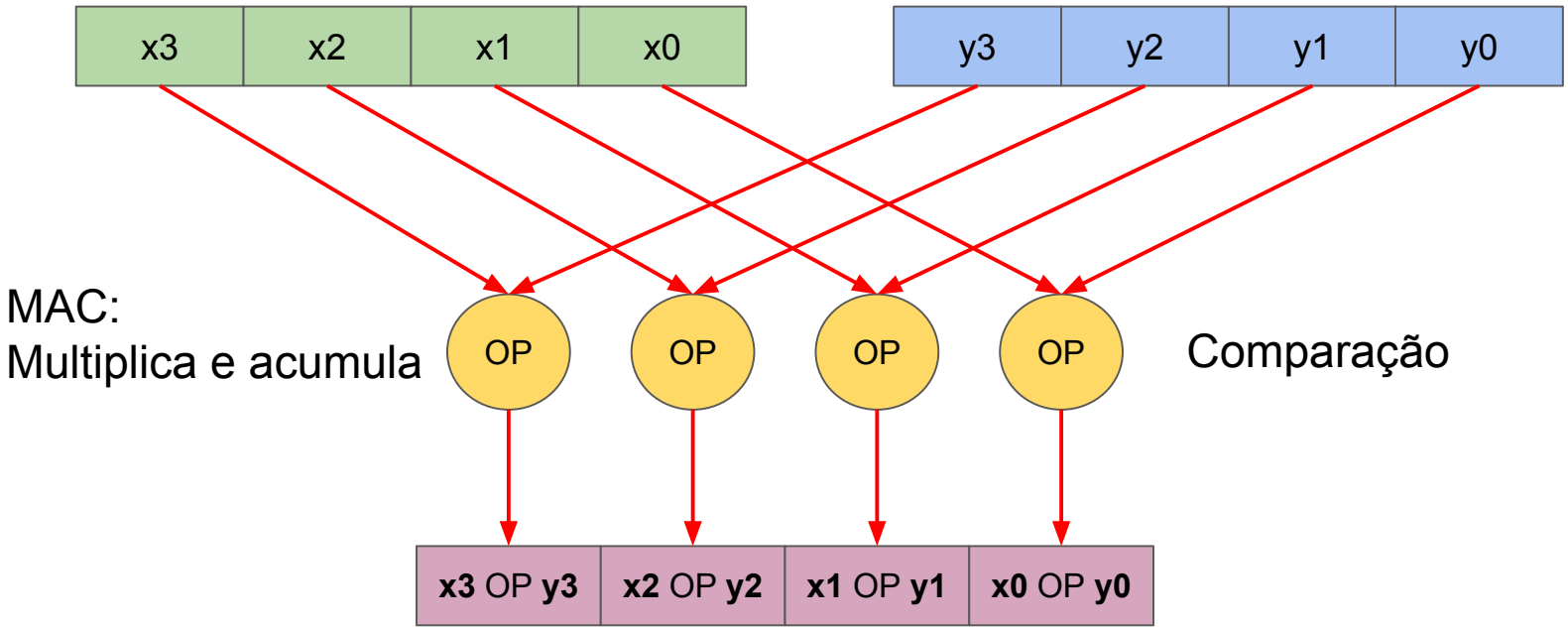


[1] AULA 12 – *Arquitetura de Computadores – Taxonomia de Flynn e Arquiteturas Paralelas* [vídeo]. YouTube, 2021. Disponível em: https://youtu.be/cEX186vzRe8. Acesso em: 21 jan. 2026.

# *Hardware* vetorial



# *Hardware* vetorial



# *Hardware* vetorial

- Principais arquiteturas:
  - **x86:**
    - SSE: registradores de 128 bits
    - AVX: registradores de 256 bits
    - AVX-512: registradores de 512 bits
  - **ARM:**
    - NEON: registradores de 128 bits
    - SVE (S*calable Vector Extension*): largura variável (de 128 até 2048 bits)

# Vetorização automática

- O GCC utiliza instruções vetoriais automaticamente a partir da *flag* -O2
- Mesmo com -O3, o compilador é conservador:
  - Não assume comportamentos arriscados
  - Prioriza a correção semântica
- Nem todas as máquinas suportam as instruções vetoriais mais recentes:
  - O compilador gera código mais genérico
  - Evita usar instruções muito novas
  - Prioriza a portabilidade

# *Flags* de vetorização

- O compilador só utiliza instruções vetoriais modernas se o programador permitir explicitamente:
  - **-mavx:** habilita o uso de instruções vetoriais AVX
  - **-mavx2:** habilita o uso de instruções vetoriais AVX2
  - **-mfma:** habilita instruções vetoriais de multiplicação-e-soma fundidas (FMA)
  - **-march=<arquitetura>:** utiliza todas as instruções disponíveis na arquitetura especificada
  - **-march=native:** utiliza todas as instruções disponíveis na arquitetura da máquina onde o código é compilado

# Vetorização do compilador

| Configuração | Tempo (s) |
|---|---|
| -O3 | 43,45 |
| + -march=native | 36,01 |

- Apenas informar a arquitetura já traz um ganho significativo.

# Vetorização do compilador

| Configuração | Tempo (s) |
|---|---|
| -O3 | 43,45 |
| + -march=native | 36,01 |

- Apenas informar a arquitetura já traz um ganho significativo.
- **Podemos ir além com simd…**
  - Sem informação adicional, o compilador assume possível *aliasing* entre ponteiros
  - A palavra-chave `restrict` informa a semântica dos dados (não há sobreposição)
  - A diretiva `#pragma omp simd` informa que o *loop* pode ser vetorizado com segurança

# Indo além com SIMD…

- restrict:
  - “Esses ponteiros não apontam para a mesma região de memória.”

```
float *restrict A;
float *restrict B;
float *restrict C;
```

- #pragma omp simd:
  - “Este *loop* pode ser vetorizado com segurança.”

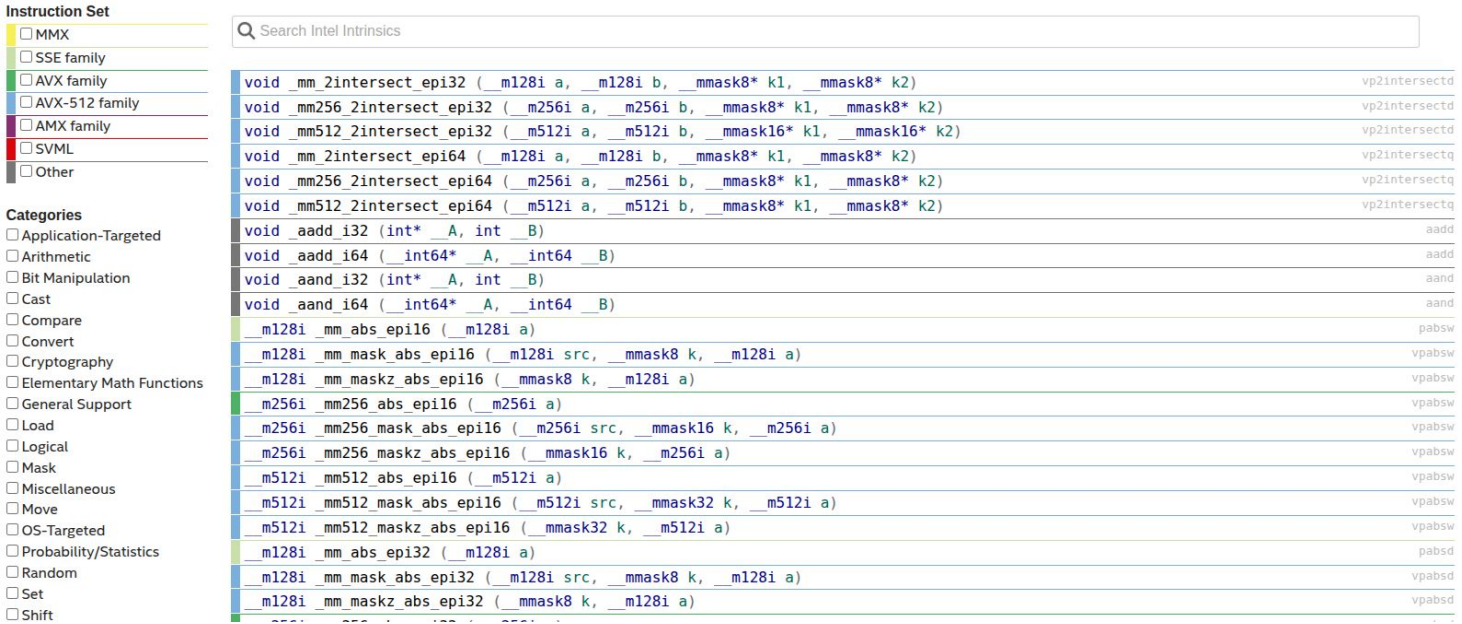
```
#pragma omp simd
for (int j = jj; j < j_max; j++) {
    C[i*N + j] += aik * B[k*N + j];
}
```

# Versão 9: Vetorização do compilador

| Configuração | Tempo (s) |
|---|---|
| -O3 | 43,45 |
| + -march=native | 36,01 |
| + restrict | 29,55 |
| + simd | 28,74 |

# Instruções AVX Intrinsics

- AVX intrinsics são funções em C/C++ fornecidas pela Intel que permitem acessar diretamente as instruções vetoriais do processador, sem escrever assembly.




[1] Intel. *Intel® Intrinsics Guide*. Disponível em: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. Acesso em: 21 jan. 2026.

# Instruções AVX Intrinsic

| Configuração | Tempo (s) |
|---|---|
| -O3 | 43,45 |
| + -march=native | 36,01 |
| + restrict | 29,55 |
| + simd | 28,74 |
| AVX manual | 71,89 |

# Versão 9: Vetorização do compilador

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 6 | Loops paralelos | 58,32 | 1,00 | 150,81934 | 2,33828 |
| 7 | + Tiling | 43,45 | 1,34 | 202,44042 | 3,13861 |
| 8 | + Vetorização do compilador | 28,74 | 2,03 | 306,01495 | 4,74442 |

- Ainda estamos usando só **4,7%** do pico.

# Agenda

- Motivação
- **Otimização:**
  - Estudo de Caso e Ambiente Experimental
  - **Versões 1-3:** *Loops* Aninhados em Python, Java e C
  - **Versão 4:** Ordem dos *Loops*
  - **Versão 5:** *Flags* de Otimização
  - **Versão 6:** *Loops* Paralelos
  - **Versão 7:** *Tiling*
  - **Versão 8:** Vetorização do Compilador
  - **Versão 9: Intel MKL**
- Mais otimizações…

# Version 9: Intel MKL

| Versão | Implementação | Tempo (s) | Speedup | GFLOPs | Pico% |
|---|---|---|---|---|---|
| 6 | Loops paralelos | 58,32 | 1,00 | 150,81934 | 2,33828 |
| 7 | + Tiling | 43,45 | 1,34 | 202,44042 | 3,13861 |
| 8 | + Vetorização do compilador | 28,74 | 2,03 | 306,01495 | 4,74442 |
| 9 | Intel MKL | 2,26 | 25,84 | 3.896,47475 | 60,41046 |

- Apesar de todo esse trabalho, não batemos a Intel MKL…

# Intel Math Kernel Library

- A Intel MKL (*Math Kernel Library*) é uma biblioteca de alto desempenho que implementa rotinas matemáticas fundamentais, otimizadas para CPUs modernas.
- Ela fornece implementações extremamente eficientes de operações como:
  - Multiplicação de matrizes
  - Álgebra linear
  - Transformadas rápidas
  - Operações vetoriais
- **Objetivo:**
  - Extrair o máximo desempenho possível do *hardware*, sem que o programador precise escrever código de baixo nível.

# Intel Math Kernel Library

- A MKL combina automaticamente:
  - Paralelismo entre núcleos (*multithreading*)
  - Vetorização SIMD (AVX, AVX2, AVX-512)
  - FMA (*fused multiply-add*)
  - Cache *blocking* profundo
  - *Prefetch* explícito
  - Escolha dinâmica de microkernels

# Agenda

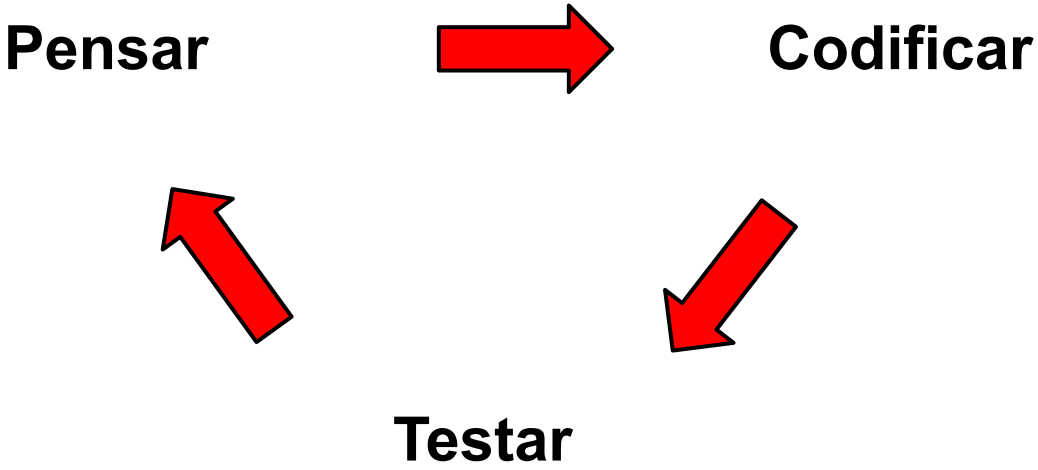
- Motivação
- Otimização:
  - Estudo de Caso e Ambiente Experimental
  - **Versões 1-3:** *Loops* Aninhados em Python, Java e C
  - **Versão 4:** Ordem dos *Loops*
  - **Versão 5:** *Flags* de Otimização
  - **Versão 6:** *Loops* Paralelos
  - **Versão 7:** *Tiling*
  - **Versão 8:** Vetorização do Compilador
  - **Versão 9:** Intel MKL
- **Mais Otimizações…**

Mais Otimizações…

# O que mais podemos fazer?

- Podemos aplicar diversos outros *insights* e técnicas de engenharia de desempenho para tornar esse código mais rápido, incluindo:
  - Pré-processamento
  - Transposição de matrizes
  - Alinhamento de dados
  - Otimizações de gerenciamento de memória
  - Um algoritmo eficiente para o caso base que utiliza explicitamente instruções AVX por meio de intrinsics

# O trabalho continua…



# Referências e materiais adicionais


- MIT OpenCourseWare — 6.172 Performance Engineering of Software Systems. Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/ Acesso em: jan. 2026.
- BORIN, Edson. LNCC14 – Material da disciplina. Universidade Estadual de Campinas (UNICAMP). Disponível em: https://www.ic.unicamp.br/~edson/disciplinas/lncc14/. Acesso em: 19 jan. 2026.