Otimização e Multiplicação de Matrizes

Lucas de Sousa Rosa e Alfredo Goldman

MAC0219 - Programação Concorrente e Paralela

27 de setembro de 2024

Mini EP5, EP1 & Seminários da Pós

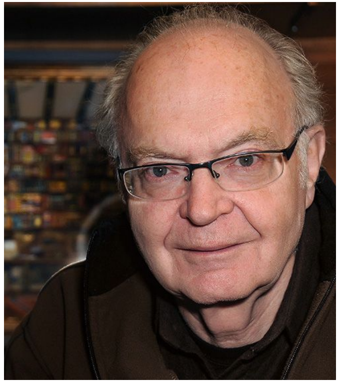
Mais detalhes sobre o Mini EP5

- **Variáveis importantes**: número de tentativas, o contador global e o tamanho do tabuleiro.
- Tipos de impasses (*deadlocks*) possíveis:
  - Não há mais movimentos válidos disponíveis.
  - Inanição (*starvation*) – threads (sapos) que consistentemente não receberem tempo de CPU suficiente, impedindo-as de realizar movimentos, mesmo que tenham opções válidas.
- Perguntas para o relatório (respostas breves):
  - Se o tamanho do tabuleiro for grande (7 já é grande), quantas vezes o problema é resolvido?
  - O que acontece com a chance de sucesso quando você diminui o limite do contador?
  - Por que as mudanças observadas ocorrem?

Motivação

- Vamos ver na prática como aplicar os conceitos desenvolvidos nos últimos Mini EPs.
  - Implementação do problema (Mini EP1).
  - Otimização do código (Mini EP2).
  - Profiling (Mini EP3).
  - Programação paralela (Mini EP4 & Mini EP5).
- Quais propriedades de software são mais importantes que desempenho?
  - Compatibilidade, funcionalidade, corretude, claridade, mantenabilidade, testabilidade, portabilidade, usabilidade… e muitas outras.
- Então, por que se preocupar com desempenho?
  - Desempenho como **moeda de troca** para comprar as outras propriedades.

Algumas Lições dos Anos 70 e 80



Donald Knuth

A otimização prematura é a raiz de todo o mal.

Michael A. Jackson

A primeira regra da otimização de programas: não faça isso. A Segunda regra da otimização de programas (somente para especialistas!): não faça isso ainda.

Estudo de Caso: Multiplicação de Matrizes

Multiplicação de Matrizes

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

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Considere por simplicidade que $n = 2^k$.

Especificação da Máquina `c4.8xlarge` (AWS)

Microarquitetura	Haswell (Intel Xeon E5-2666 v3)
Frequência do relógio	2,9 GHz
Número de núcleos	2×9
Hyperthreading	2 way
Unidade de ponto flutuante	8 operações de double-precision, com *fused-multiply-add*, por núcleo por ciclo
Tamanho da linha de cache	64 B
L1i & L1d cache	32 KB (*private 8-way set associative*)
L2 cache	256 KB (*private 8-way set associative*)
L3 cache (LLC)	25 MB (*shared 20-way set associative*)
DRAM	60 GB

Pico de desempenho (teórico): $(2{,}9 \times 10^9) \times 2 \times 9 \times 16 = 836$ GFLOPS

Versão 1: Laços Aninhados em Python

```
import random
from time import time

n = 4096

A = [[random.random() for row in range(n)] for col in
range(n)]
B = [[random.random() for row in range(n)] for col in
range(n)]
C = [[0 for row in range(n)] for col in range(n)]

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

print("Time: %.6f sec" % (end - start))
```

Quanto tempo esse código leva para rodar?

Versão 1: Laços Aninhados em 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 = 21042 segundos ≅ 6 horas

Isso é rápido?

- Quantas operações de ponto flutuante esse laço realiza? E para n = 4096?
- Quanto de desempenho estamos extraindo do máximo teórico?

Versão 1: Laços Aninhados em 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

= 21042 segundos ≅ 6 horas

Isso é rápido?

- Quantas operações de ponto flutuante esse laço realiza? E para $n = 4096$?
  - Há n^2 elementos na matriz C. Para cada entrada são feitas n multiplicações e $(n - 1)$ adições.
  - $n^2 \times (2n - 1) \cong 2n^3$ operações de ponto flutuante.
  - $2n^3 = 2 \times (4096)^3 = 2 \times (2^{12})^3 \cong 2^{37}$ operações de ponto flutuante.
- Quanto de desempenho estamos extraindo do máximo teórico?

Versão 1: Laços Aninhados em 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 = 21042 segundos ≅ 6 horas

Isso é rápido?

- Quantas operações de ponto flutuante esse laço realiza? E para $n = 4096$?
    - Há n^2 elementos na matriz C. Para cada entrada são feitas n multiplicações e $(n - 1)$ adições.
    - $n^2 \times (2n - 1) \cong 2n^3$ operações de ponto flutuante.
    - $2n^3 = 2 \times (4096)^3 = 2 \times (2^{12})^3 \cong 2^{37}$ operações de ponto flutuante.
- Quanto de desempenho estamos extraindo do máximo teórico?
    - $2^{37}/21042 \cong$ 6,25 MFLOPS.
    - 6,25/836 x 100% [M/G] ≅ 0,00075% do máximo teórico.

Versão 2: Java

```java
import java.util.Random;

public class mm {
   static int n = 4096;
   static double[][] A = new double[n][n];
   static double[][] B = new double[n][n];
   static double[][] C = new double[n][n];

   public static void main(String[] args) {
       Random r = new Random();

       for (int i = 0; i < n; i++) {
           for (int j = 0; j < n; j++) {
               A[i][j] = r.nextDouble();
               B[i][j] = r.nextDouble();
               C[i][j] = 0;
           }
       }

       long start = System.nanoTime();
       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];
               }
           }
       }
       long end = System.nanoTime();

       double tdiff = (end - start) * 1e-9;
       System.out.println("Time: " + tdiff + " sec");
   }
}
```

Tempo de execução

2738 segundos ≅ 46 minutos

8,8x mais rápido que Python

```java
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 3: C

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#define n 4096
double A[n][n], B[n][n], C[n][n];

// ... (funções auxiliares)

int main(int argc, const char *argv[]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = (double)rand() / (double)RAND_MAX;
            B[i][j] = (double)rand() / (double)RAND_MAX;
            C[i][j] = 0;
        }
    }

    struct timeval start, end;
    gettimeofday(&start, NULL);

    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];
            }
        }
    }

    gettimeofday(&end, NULL);
    printf("Time: %f sec\n", tdiff(&start, &end));
    return 0;
}
```

Tempo de execução

≅ 1156 segundos ≅ 19 minutos

2x mais rápido que Java e 18x mais rápido que Python

```
$ clang mm.c -o mm
```

```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];
        }
    }
}
```

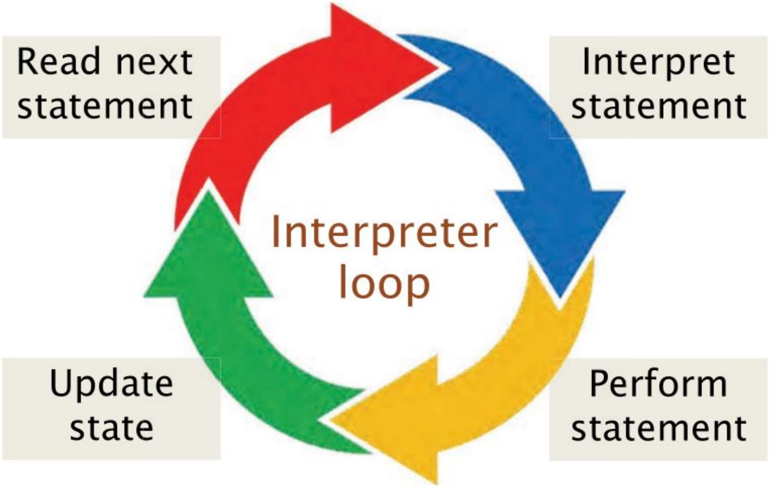
Onde Estamos até Agora

Versão	Implementação	Tempo de Execução (s)	Speedup Relativo	Speedup Absoluto	GFLOPS	% do Pico
1	Python	21041,67	1,00	1	0,007	0,001
2	Java	2387,32	8,81	9	0,058	0,007
3	C	1155,77	2,07	18	0,119	0,014

- Qual a diferença entre as três linguagens?
  - Python é interpretada.
  - C é compilada diretamente para linguagem de máquina.
  - Java é compilada para *bytecode* que é interpretada e compilada *just-in-time* (JIT) para código de máquina.

Interpretadores são Versáteis, mas Lentos

- O interpretador lê, interpreta e executa cada instrução do programa e atualiza o estado da máquina.
- Os interpretadores podem facilmente oferecer suporte a recursos de programação de alto nível — como alteração dinâmica de código — em detrimento do desempenho.



Ordem dos Laços

Podemos alterar a ordem dos laços neste programa sem afetar sua corretude.

```
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 Laços

Podemos alterar a ordem dos laços neste programa sem afetar sua corretude.

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

Ordem dos Laços

Podemos alterar a ordem dos laços neste programa sem afetar sua corretude.

```
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 laços faz diferença no desempenho?

Desempenho de Diferentes Ordens

Ordem do Laço (Externo para o Interno)	Tempo de Execução (s)
i, j, k	1155,77
i, k, j	177,68
j, i, k	1080,61
j, k, i	3056,63
k, i, j	179,21
k, j, i	3032,82

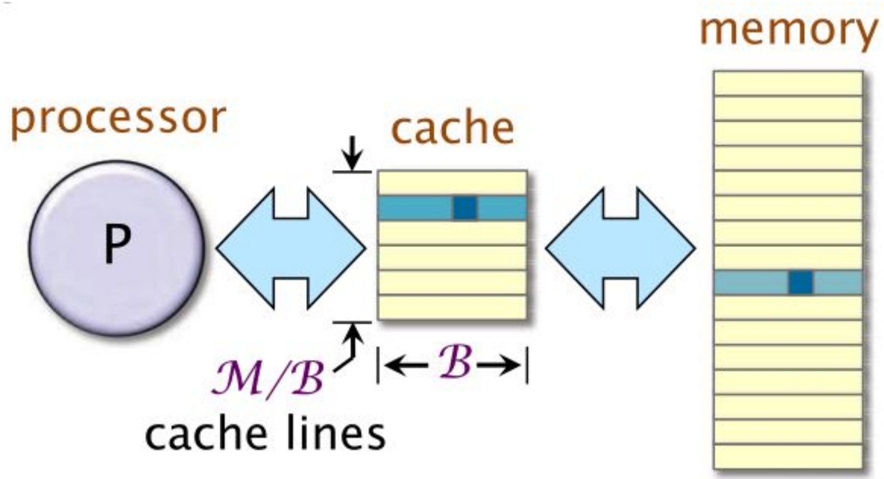
A ordem dos laços afetou o tempo de execução por fator de 18x!

Por isso aconteceu?!

Memória Cache

Cada processador lê e grava na memória principal em blocos contíguos, chamados **linhas de cache**.

- Linhas de cache acessadas anteriormente são armazenadas em uma memória menor, chamada **cache**, que fica perto do processador.
- **Cache hits** — acessos a dados no cache — são rápidos.
- **Cache misses** — acessos a dados que não estão no cache — são lentos.



Layout de Memória de Matrizes

Neste código de multiplicação de matrizes, as matrizes são dispostas na memória em ***row-major order***.

Matriz

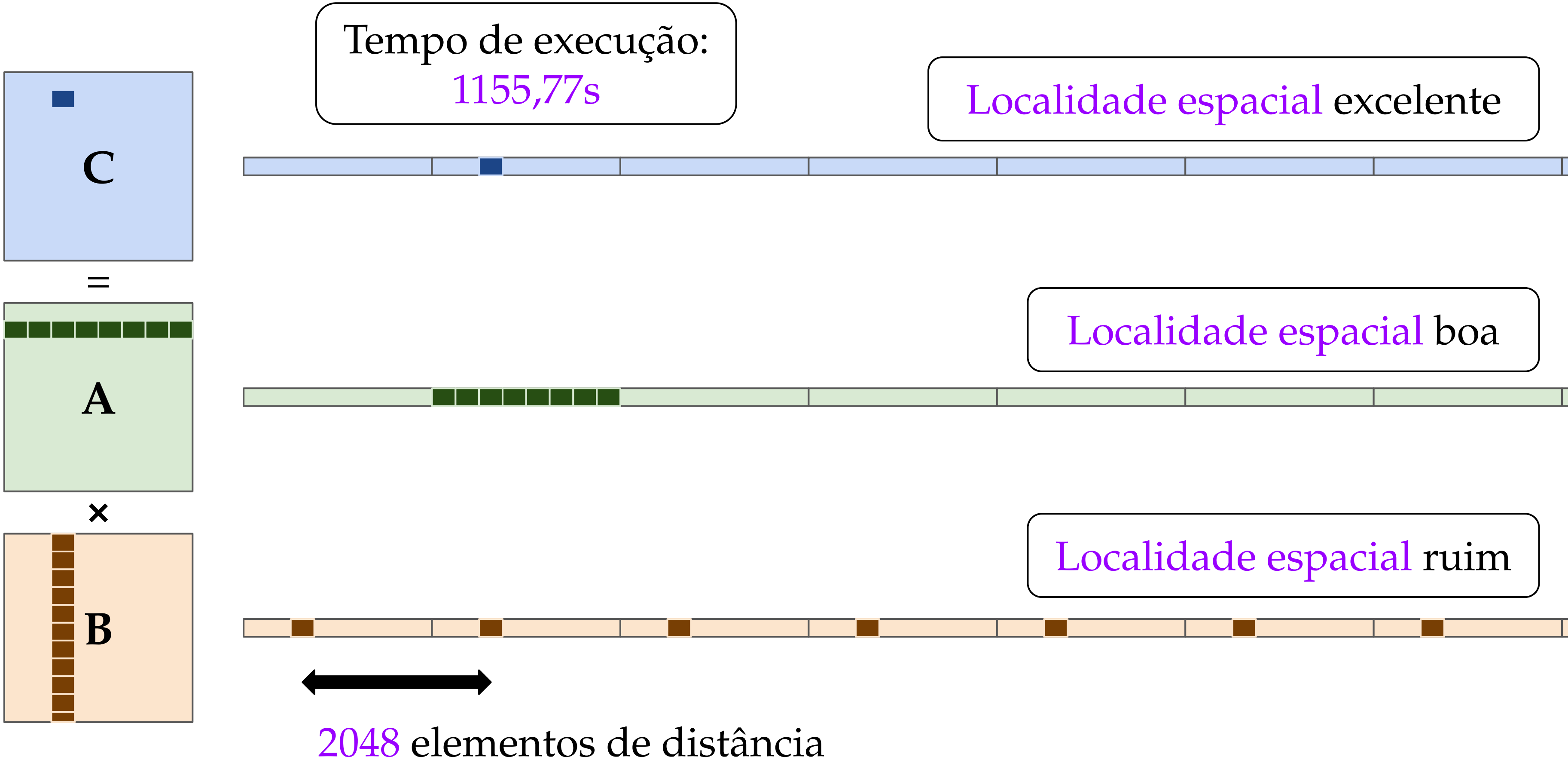
Linha 1
Linha 2
Linha 3
Linha 4
Linha 5
Linha 6

Qual a relação desse layout com a variação de desempenho para as diferentes ordens dos laços?

Memória

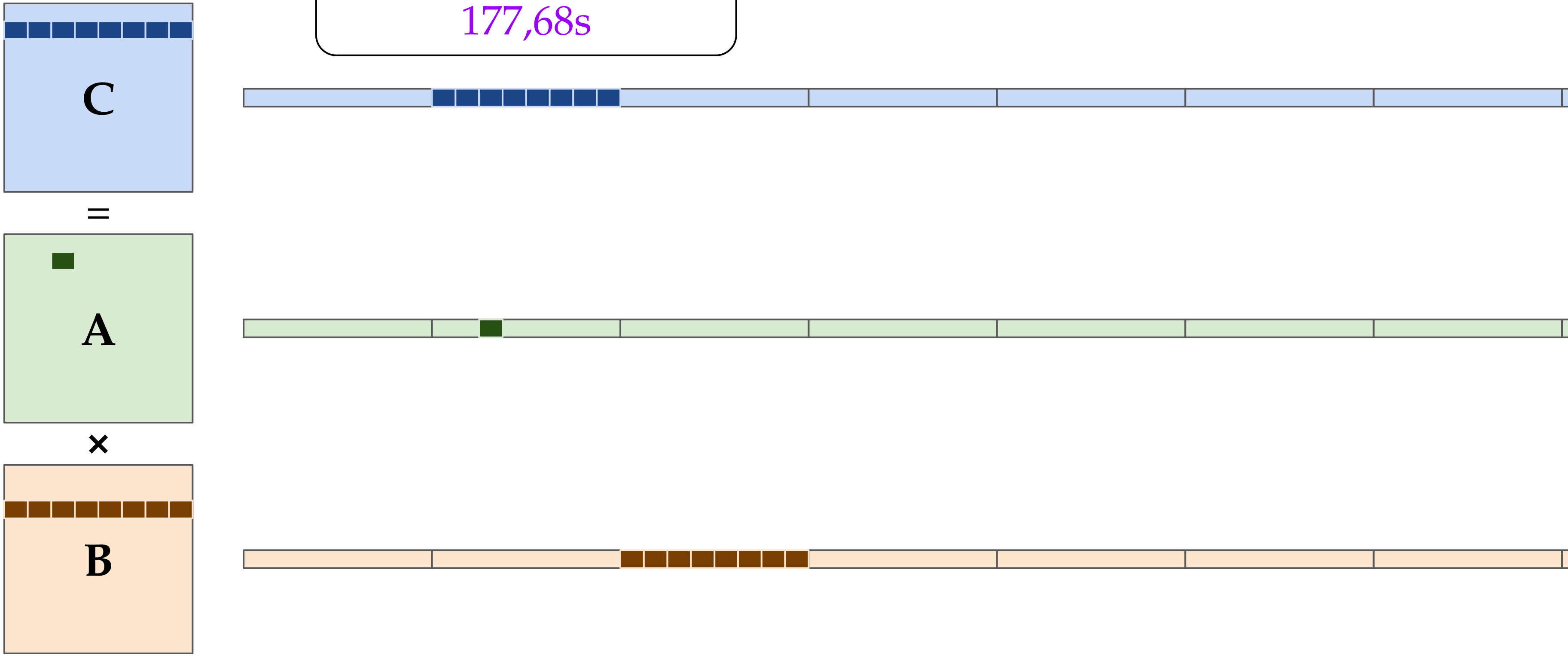
Linha 1	Linha 2	Linha 3	Linha 4	Linha 5

Padrão de Acesso para i, j, k



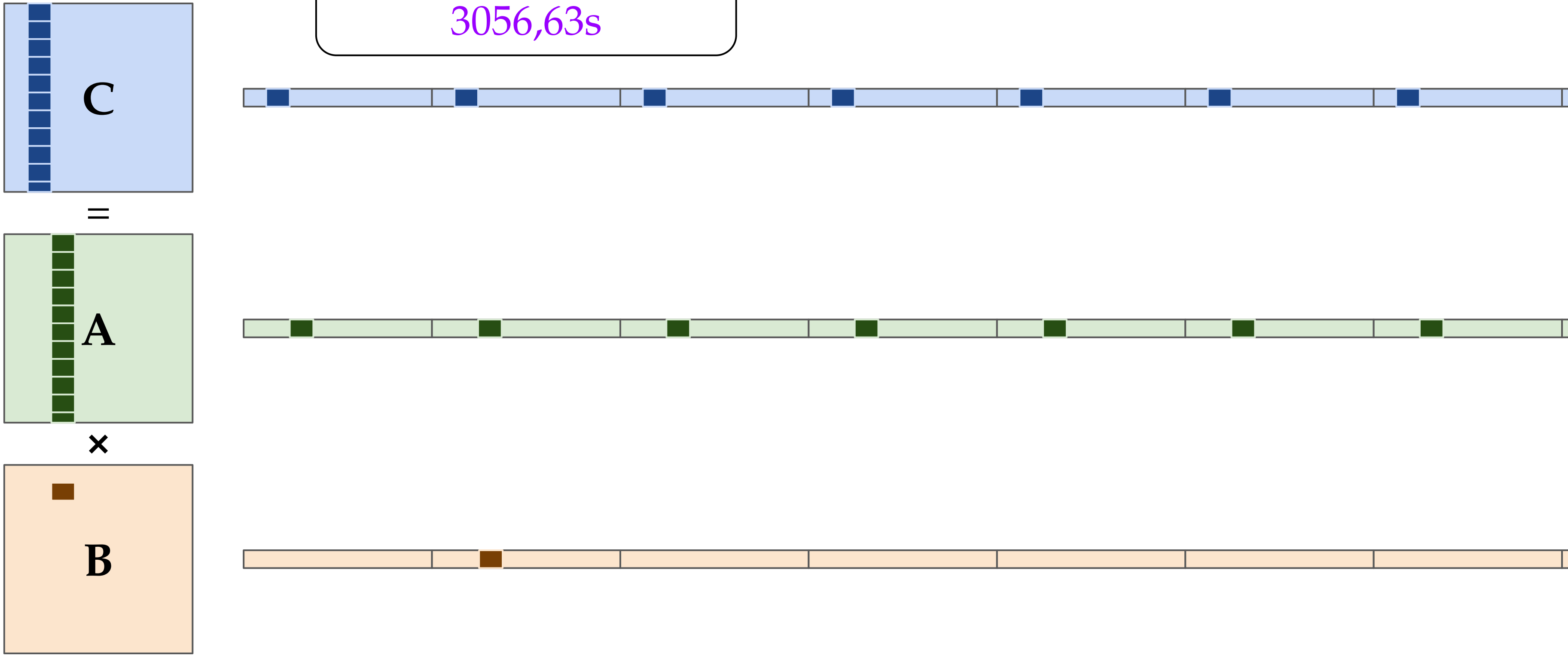
Padrão de Acesso para i, k, j

Tempo de execução:
177,68s



Padrão de Acesso para j, k, i

Tempo de execução:
3056,63s



Desempenho de Diferentes Ordens

Podemos medir o efeito de diferentes padrões de acesso usando o simulador de cache Cachegrind:

```
$ valgrind --tool=callgrind --simulate-cache=yes ./mm
```

Ordem do Laço (Externo para o Interno)	Tempo de Execução (s)	Last-level-cache miss rate
i, j, k	1155,77	7,7%
i, k, j	177,68	1,0%
j, i, k	1080,61	8,6%
j, k, i	3056,63	15,4%
k, i, j	179,21	1,0%
k, j, i	3032,82	15,4%

Versão 4: Troca da Ordem dos Laços

Versão	Implementação	Tempo de Execução (s)	Speedup Relativo	Speedup Absoluto	GFLOPS	% do Pico
1	Python	21041,67	1,00	1	0,007	0,001
2	Java	2387,32	8,81	9	0,058	0,007
3	C	1155,77	2,07	18	0,119	0,014
4	+ troca da ordem dos laços	177,68	6,50	118	0,774	0,093

Que outra simples mudança podemos fazer?

Melhorando o Desempenho de C

- Clang fornece uma coleção de *switches* de otimização. Você pode especificar um *switch* para o compilador para pedir que ele otimize.

Nível de Otimização	Significado	Tempo (s)
`-O0`	Não otimize	177,54
`-O1`	Otimize	66,24
`-O2`	Otimize ainda mais	54,63
`-O3`	Otimize AINDA mais	55,58

- Nem sempre `-O3` será mais rápido!!
  - Compilador usa heurísticas para as otimizações.

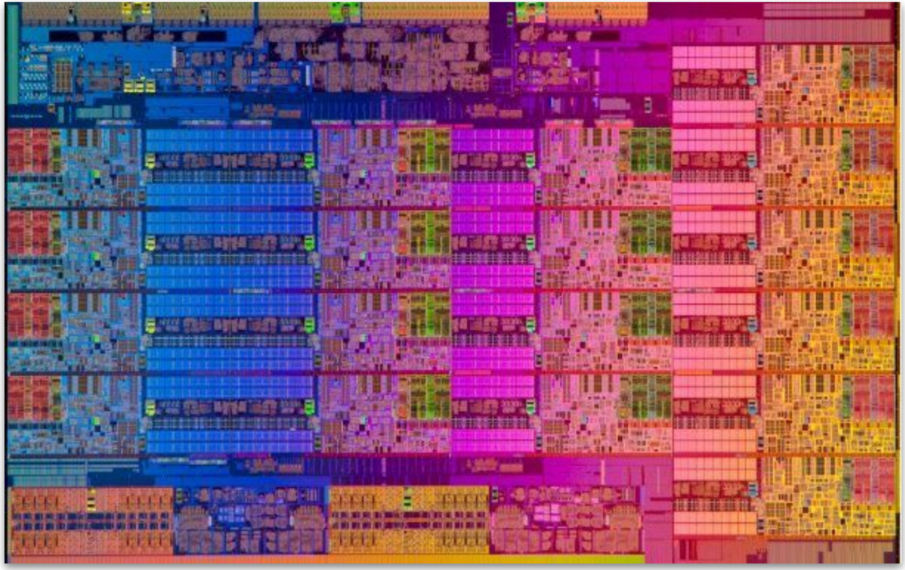
Versão 5: Flags de Otimização

Versão	Implementação	Tempo de Execução (s)	Speedup Relativo	Speedup Absoluto	GFLOPS	% do Pico
1	Python	21041,67	1,00	1	0,007	0,001
2	Java	2387,32	8,81	9	0,058	0,007
3	C	1155,77	2,07	18	0,119	0,014
4	+ troca da ordem dos laços	177,68	6,50	118	0,774	0,093
5	+ flags de otimização	54,63	3,25	385	2,516	0,301

Com um simples código e ajustes na compilação conseguimos miseros 0,3% do máximo teórico.

Por que tão pouco desempenho?

Paralelismo Multicore



Intel Haswell E5: 9 núcleos por chip.

Máquina de testes da AWS possui 2 desses chips.

Estamos usando apenas 1 dos 18 núcleos. Vamos usá-los todos!!

Laços Paralelos e Cilk

Conhecemos Pthreads e OpenMP. Apresento-vos **Cilk**.

- Extensão simples para C (não tão popular quanto OpenMP).
- Apenas **três** palavras-chave básicas.
  - `cilk_for` indica que o laço pode ser executado em paralelo.
  - `cilk_spawn` indica que a subrotina pode ser executada de forma independente e em paralelo com o chamador.
  - `cilk_sync` mecanismo de sincronização. Barreira.
- Mantém a semântica serial.
  - Abstrai a execução paralela, o balanceamento de carga e o escalonamento.
- Fornece algumas "garantias" de desempenho.

Laços Paralelos

```
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];
        }
    }
}
```

Esses laços podem ser (facilmente) paralelizados

Por que esse laço não pode ser paralelizado?

Qual versão paralela é melhor?

Experimentos com Laços Paralelos

```
cilk_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 de execução: 3,18s

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

Tempo de execução: 531,72s

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

Tempo de execução: 10,64s

Regra Prática

Sempre paralelize o laço externo

Versão 6: Laços Paralelos

Versão	Implementação	Tempo de Execução (s)	Speedup Relativo	Speedup Absoluto	GFLOPS	% do Pico
1	Python	21041,67	1,00	1	0,007	0,001
2	Java	2387,32	8,81	9	0,058	0,007
3	C	1155,77	2,07	18	0,119	0,014
4	+ troca da ordem dos laços	177,68	6,50	118	0,774	0,093
5	+ flags de otimização	54,63	3,25	385	2,516	0,301
6	Laços paralelos	3,04	17,97	6921	45,211	5,408

Laços paralelos nos deram quase 18x de *speedup* usando os 18 núcleos! (Nem todo código é fácil de paralelizar).

Versão 6: Laços Paralelos

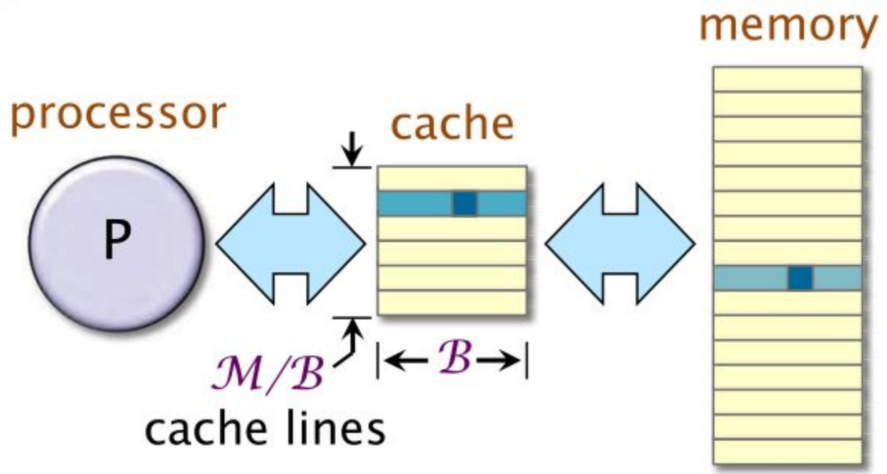
Versão	Implementação	Tempo de Execução (s)	Speedup Relativo	Speedup Absoluto	GFLOPS	% do Pico
1	Python	21041,67	1,00	1	0,007	0,001
2	Java	2387,32	8,81	9	0,058	0,007
3	C	1155,77	2,07	18	0,119	0,014
4	+ troca da ordem dos laços	177,68	6,50	118	0,774	0,093
5	+ flags de otimização	54,63	3,25	385	2,516	0,301
6	Laços paralelos	3,04	17,97	6921	45,211	5,408

Mesmo assim, só 5% do máximo teórico. Por quê?

Memória Cache de Novo

Precisamos otimizar a utilização da memória cache ao máximo.

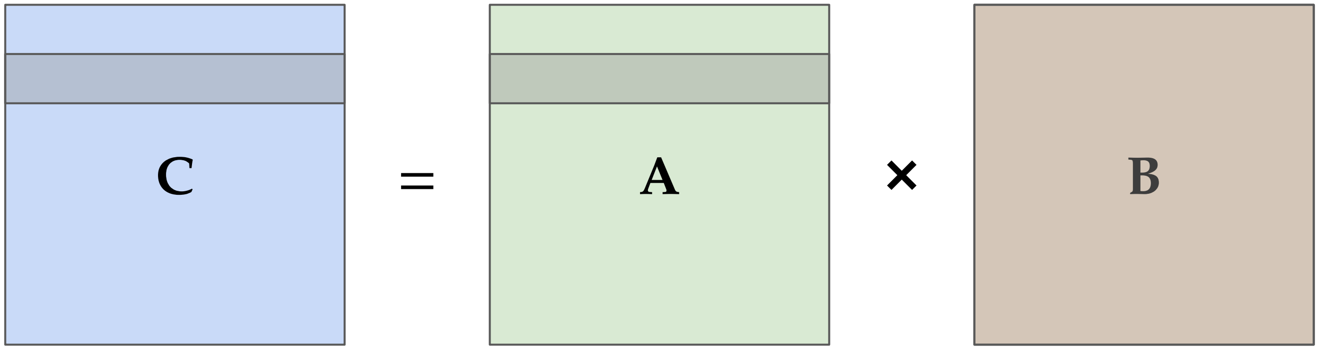
- *Cache misses* são lentos e cache hits são rápidos.
- Vamos tentar utilizar ao máximo a cache, usando os dados que já estão lá.



Aproveitando Melhor os Dados

Quantos acessos a memória são necessários para realizar a computação de **uma** linha inteira de C?

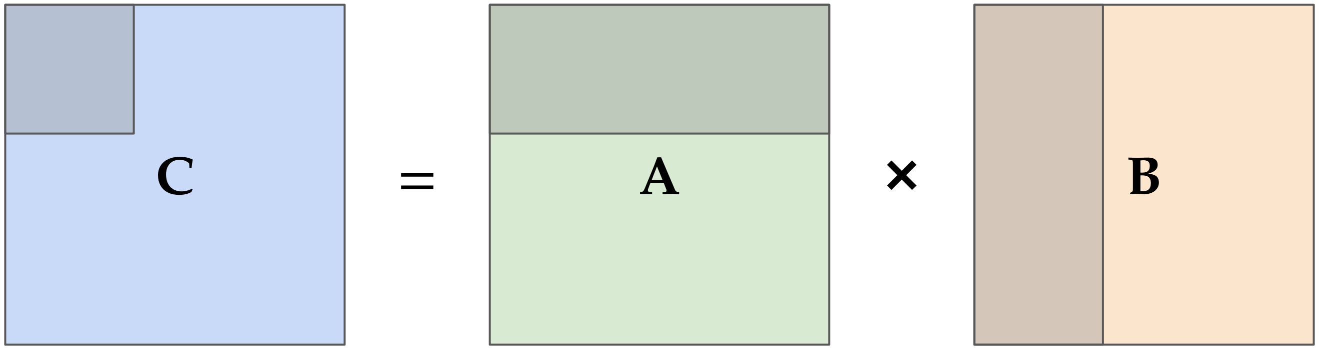
- 4096 x 1 = 4096 escritas em C.
- 4096 x 1 = 4096 leituras de A.
- 4096 x 4096 = 16.777.216 leituras de B.
- 16.785.408 acessos a memória no total.



Aproveitando Melhor os Dados

Quantos acessos a memória são necessários para realizar a computação de um bloco 64x64 de C?

- 64 x 64 = 4096 escritas em C.
- 64 x 4096 = 262.144 leituras de A.
- 4096 x 64 = 262.144 leituras de B.
- 528.384 acessos a memória no total.



Multiplicação de Matrizes por Blocos

```
cilk_for (int ih = 0; ih < n; ih += s)
   cilk_for (int jh = 0; jh < n; jh += s)
       for (int kh = 0; kh < n; kh += s)
           for (int il = 0; il < s; il++)
               for (int kl = 0; kl < s; kl++)
                   for (int jl = 0; jl < s; jl++)
                       C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[kh+kl][jh+jl];
```

- Como descobrimos o melhor valor para s?
  - Experimentos!

Multiplicação de Matrizes por Blocos

Tamanho do bloco	Tempo de Execução (s)
4	6,74
8	2,76
16	2,49
32	1,79
64	2,33
128	2,13

A versão por bloco tem 62% menos referências a cache e resulta em 68% menos *cache misses*.

Implementação	Referências a Cache (milhões)	L1-d cache misses (milhões)	Last-level cache misses (milhões)
Laços paralelos	104.090	17.220	8.600
+ blocos	64.690	11.777	416

Versão 7: Blocos

Versão	Implementação	Tempo de Execução (s)	Speedup Relativo	Speedup Absoluto	GFLOPS	% do Pico
1	Python	21041,67	1,00	1	0,007	0,001
2	Java	2387,32	8,81	9	0,058	0,007
3	C	1155,77	2,07	18	0,119	0,014
4	+ troca da ordem dos laços	177,68	6,50	118	0,774	0,093
5	+ flags de otimização	54,63	3,25	385	2,516	0,301
6	Laços paralelos	3,04	17,97	6921	45,211	5,408
7	+ blocos	1,79	1,70	11772	76,782	9,184

Multiplicação de Matrizes por Blocos

```
cilk_for (int ih = 0; ih < n; ih += s)
   cilk_for (int jh = 0; jh < n; jh += s)
       for (int kh = 0; kh < n; kh += s)
           for (int im = 0; im < s; im += t)
               for (int jm = 0; jm < s; jm += t)
                   for (int km = 0; km < s; km += t)
                       for (int il = 0; il < t; il++)
                           for (int kl = 0; kl < t; kl++)
                               for (int jl = 0; jl < t; jl++)
                                   C[ih+im+il][jh+jm+jl] +=
                                       A[ih+im+il][kh+km+kl] * B[kh+km+kl][jh+jm+jl];
```

Podemos otimizar ainda mais o acesso a cache, considerando os outros níveis.

- Aqui o código já começa a ficar poluído. Será que não tem uma forma melhor?

Multiplicação de Matrizes Recursiva

$$C = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{bmatrix} + \begin{bmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{bmatrix}$$

Criar blocos para **toda** potência de 2.

- 8 multiplicações de matrizes n/2 x n/2.
- 1 soma de matrizes n x n.

Multiplicação de Matrizes Recursiva

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{
   assert((n & -n) == n); // n is a power of 2
   if (n <= 1) {
      *C += *A * *B;
   } else {
#define X(M, r, c)  (M + (r*(n_ ## M) + c)*(n/2)) // M[r][c]
       cilk_spawn mm_dac(X(C, 0, 0), n_C, X(A, 0, 0), n_A, X(B, 0, 0), n_B, n/2);
       cilk_spawn mm_dac(X(C, 0, 1), n_C, X(A, 0, 0), n_A, X(B, 0, 1), n_B, n/2);
       cilk_spawn mm_dac(X(C, 1, 0), n_C, X(A, 1, 0), n_A, X(B, 0, 0), n_B, n/2);
       mm_dac(X(C, 1, 1), n_C, X(A, 1, 0), n_A, X(B, 0, 1), n_B, n/2);
       cilk_sync;
       cilk_spawn mm_dac(X(C, 0, 0), n_C, X(A, 0, 1), n_A, X(B, 1, 0), n_B, n/2);
       cilk_spawn mm_dac(X(C, 0, 1), n_C, X(A, 0, 1), n_A, X(B, 1, 1), n_B, n/2);
       cilk_spawn mm_dac(X(C, 1, 0), n_C, X(A, 1, 1), n_A, X(B, 1, 0), n_B, n/2);
       mm_dac(X(C, 1, 1), n_C, X(A, 1, 1), n_A, X(B, 1, 1), n_B, n/2);
       cilk_sync;
   }
}
```

Caso base é muito pequeno. É preciso **aumentar** o tamanho para evitar o overhead das chamadas recursivas.

Tempo de execução: 93,93s
50x mais devagar que a versão anterior!

Ajustando o Caso Base

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{
   assert((n & -n) == n); // n is a power of 2
   if (n <= THRESHOLD) {
       mm_base(C, n_C, A, n_A, B, n_B, n);
   } else {
#define X(M, r, c)  (M + (r*(n_ ## M) + c)*(n/2)) // M[r][c]
       cilk_spawn mm_dac(X(C, 0, 0), n_C, X(A, 0, 0), n_A, X(B, 0, 0), n_B, n/2);
       cilk_spawn mm_dac(X(C, 0, 1), n_C, X(A, 0, 0), n_A, X(B, 0, 1), n_B, n/2);
       cilk_spawn mm_dac(X(C, 1, 0), n_C, X(A, 1, 0), n_A, X(B, 0, 0), n_B, n/2);
       mm_dac(X(C, 1, 1), n_C, X(A, 1, 0), n_A, X(B, 0, 1), n_B, n/2);
       cilk_sync;
       cilk_spawn mm_dac(X(C, 0, 0), n_C, X(A, 0, 1), n_A, X(B, 1, 0), n_B, n/2);
       cilk_spawn mm_dac(X(C, 0, 1), n_C, X(A, 0, 1), n_A, X(B, 1, 1), n_B, n/2);
       cilk_spawn mm_dac(X(C, 1, 0), n_C, X(A, 1, 1), n_A, X(B, 1, 0), n_B, n/2);
       mm_dac(X(C, 1, 1), n_C, X(A, 1, 1), n_A, X(B, 1, 1), n_B, n/2);
       cilk_sync;
   }
}
```

Apenas um parâmetro para otimizar.

Ajustando o Caso Base

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{
   assert((n & -n) == n); // n is a power of 2
   if (n <= THRESHOLD) {
       mm_base(C, n_C, A, n_A, B, n_B, n);
   } else {
#define X(M, r, c) (M + (r*(n_ ## M)
       cilk_spawn mm_dac(X(C, 0, 0),
       cilk_spawn mm_dac(X(C, 0, 1),
       cilk_spawn mm_dac(X(C, 1, 0),
       mm_dac(X(C, 1, 1), n_C, X(A, 1
       cilk_sync;
       cilk_spawn mm_dac(X(C, 0, 0),
       cilk_spawn mm_dac(X(C, 0, 1),
       cilk_spawn mm_dac(X(C, 1, 0),
       mm_dac(X(C, 1, 1), n_C, X(A, 1
       cilk_sync;
   }
}
```

```
void mm_base(double *restrict C, int n_C,
             double *restrict A, int n_A,
             double *restrict B, int n_B,
             int n)
{
   for (int i = 0; i < n; i++)
      for (int k = 0; k < n; k++)
         for (int j = 0; j < n; j++)
            C[i*n_C + j] += A[i*n_A + k] * B[k*n_B + j];
}
```

Multiplicação de Matrizes por Blocos

Tamanho do bloco	Tempo de Execução (s)
4	3,00
8	1,34
16	1,34

Tamanho do bloco	Tempo de Execução (s)
32	1,30
64	1,95
128	2,08

Implementação	Referências a Cache (milhões)	L1-d cache misses (milhões)	Last-level cache misses (milhões)
Laços paralelos	104.090	17.220	8.600
+ blocos	64.690	11.777	416
Divisão e conquista paralela	58.230	9.407	64

Versão 8: Divisão e Conquista Paralela

Versão	Implementação	Tempo de Execução (s)	Speedup Relativo	Speedup Absoluto	GFLOPS	% do Pico
1	Python	21041,67	1,00	1	0,007	0,001
2	Java	2387,32	8,81	9	0,058	0,007
3	C	1155,77	2,07	18	0,119	0,014
4	+ troca da ordem dos laços	177,68	6,50	118	0,774	0,093
5	+ flags de otimização	54,63	3,25	385	2,516	0,301
6	Laços paralelos	3,04	17,97	6921	45,211	5,408
7	+ blocos	1,79	1,70	11772	76,782	9,184
8	Divisão e conquista paralela	1,30	1,38	16197	105,722	12,646

Vetorização

Processadores modernos são capazes de processar múltiplos dados em uma única instrução (SIMD).

Operação Escalar

$$c_1 = a_1 + b_1$$

$$c_2 = a_2 + b_2$$

$$c_3 = a_3 + b_3$$

Operação Vetorizada

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Flags de Vetorização

Podemos direcionar o compilador para usar instruções vetoriais modernas usando flags como:

- `-mavx`: Usa as instruções de vetorização AVX.
- `-mavx2`: Usa as instruções de vetorização AVX2.
- `-mfma`: Usa a instrução de vetorização *fused multiply-add*.
- `-march`: Usa qualquer instrução disponível na máquina que está compilando o programa.

Por conta de restrições de aritimética de ponto flutuante a flag `-ffast-math` pode ser necessária para a vetorização fazer efeito.

Versão 9: Flags Vetorização

Versão	Implementação	Tempo de Execução (s)	Speedup Relativo	Speedup Absoluto	GFLOPS	% do Pico
1	Python	21041,67	1,00	1	0,007	0,001
2	Java	2387,32	8,81	9	0,058	0,007
3	C	1155,77	2,07	18	0,119	0,014
4	+ troca da ordem dos laços	177,68	6,50	118	0,774	0,093
5	+ flags de otimização	54,63	3,25	385	2,516	0,301
6	Laços paralelos	3,04	17,97	6921	45,211	5,408
7	+ blocos	1,79	1,70	11772	76,782	9,184
8	Divisão e conquista paralela	1,30	1,38	16197	105,722	12,646
9	+ flags de vetorização	0,70	1,87	30272	196,341	23,486

Sendo Mais Espertos que o Compilador

A Intel fornece funções em C, chamadas **instruções intrínsecas**, que fornecem acesso direto às operações de vetorização de hardware.

- https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

E com Algumas Otimizações a Mais

Podemos aplicar vários outros *insights* e truques para fazer esse código rodar mais rápido, incluindo:

- Pré-processamento
- Transposição de matriz
- Alinhamento de dados
- Otimizações no gerenciamento de memória
- Um algoritmo mais inteligente para o caso base que usa instruções intrínsecas AVX explicitamente.

Spoiler do próximo Mini EP 😅.

Versão 10 & 11: AVX Intrinsics & MKL

Versão	Implementação	Tempo de Execução (s)	Speedup Relativo	Speedup Absoluto	GFLOPS	% do Pico
1	Python	21041,67	1,00	1	0,007	0,001
…	…	…	…	…	…	…
9	+ flags de vetorização	0,70	1,87	30272	196,341	23,486
10	+ AVX intrinsics	0,39	1,76	53292	352,408	41,677
11	Intel MKL	0,41	0,97	51497	335,217	40,098

Podemos chegar no nível da biblioteca Math Kernel Library (MKL) desenvolvida por profissionais da Intel 😎.

Conclusões

53.292x

Economia de combustível
em MPG

Geralmente não vemos esse tipo de ganho de desempenho. Multiplicação de matrizes é um problema especial.

Referências

- Essa aula foi preparada com base nos slides do Professor Charles Leiserson "*Lecture 1: Introduction and Matrix Multiplication*" do curso 6.172 Performance Engineering of Software Systems do MIT; nos slides do Professor Ion Stoica "*Cilk and OpenMP*" do curso CS262a Advanced Topics in Computer Systems da UC Berkeley.
- Os códigos-fonte utilizados e outras aulas estão disponíveis no repositório do Github **fredgrub/aulas-PPD**.