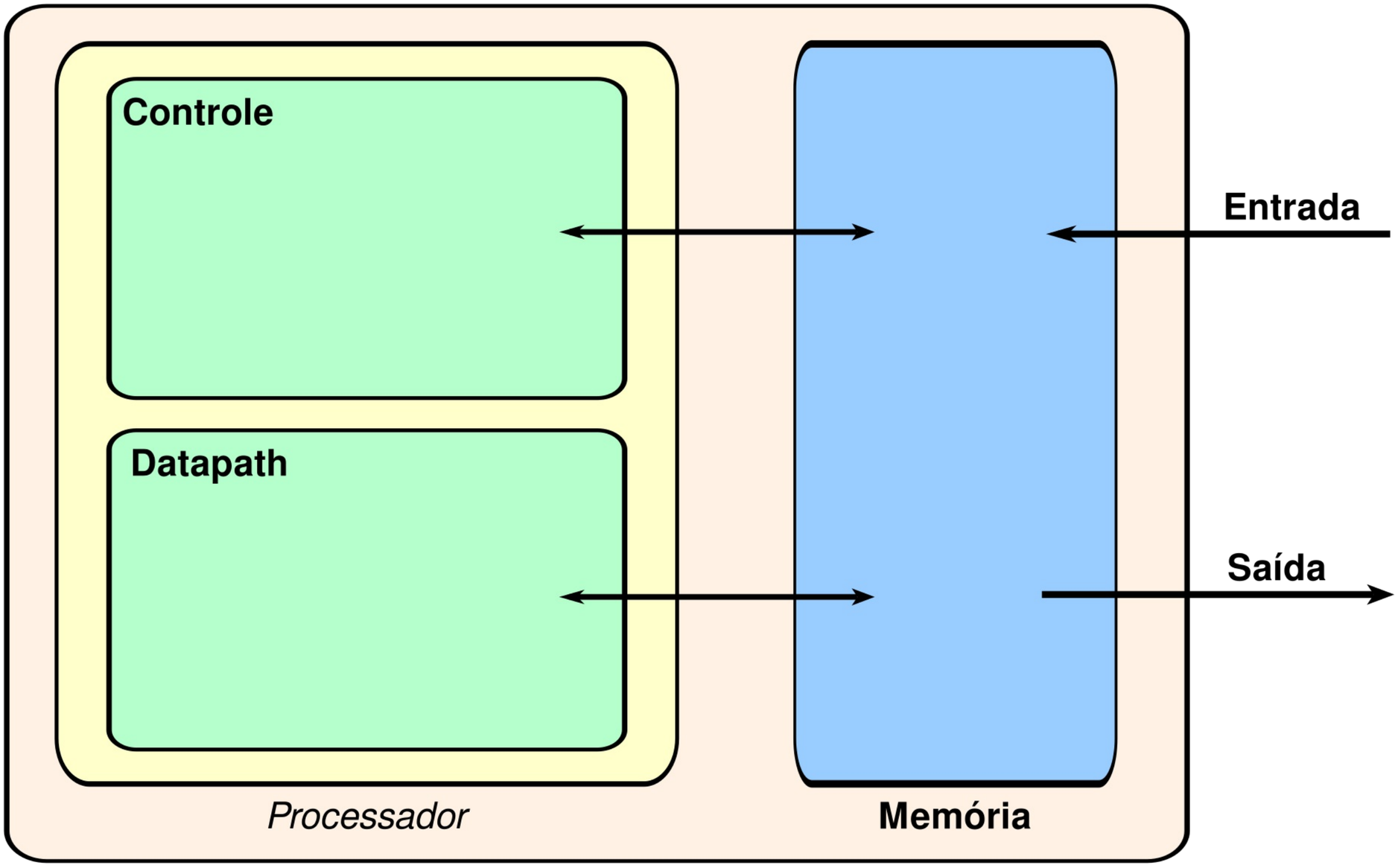
MOTIVAÇÃO


Controle
Datapath
Entrada
Saída
Processador
Memória
COMPUTADOR

MOTIVAÇÃO


Servo ou magia sem controle?
int não é Z; float não é R
Olá, mundo!
Memória importa
Entrada/Saída/Rede
Motivação
Evolução
Rito de passagem
Pilha tecnológica

Entender a memória importa!

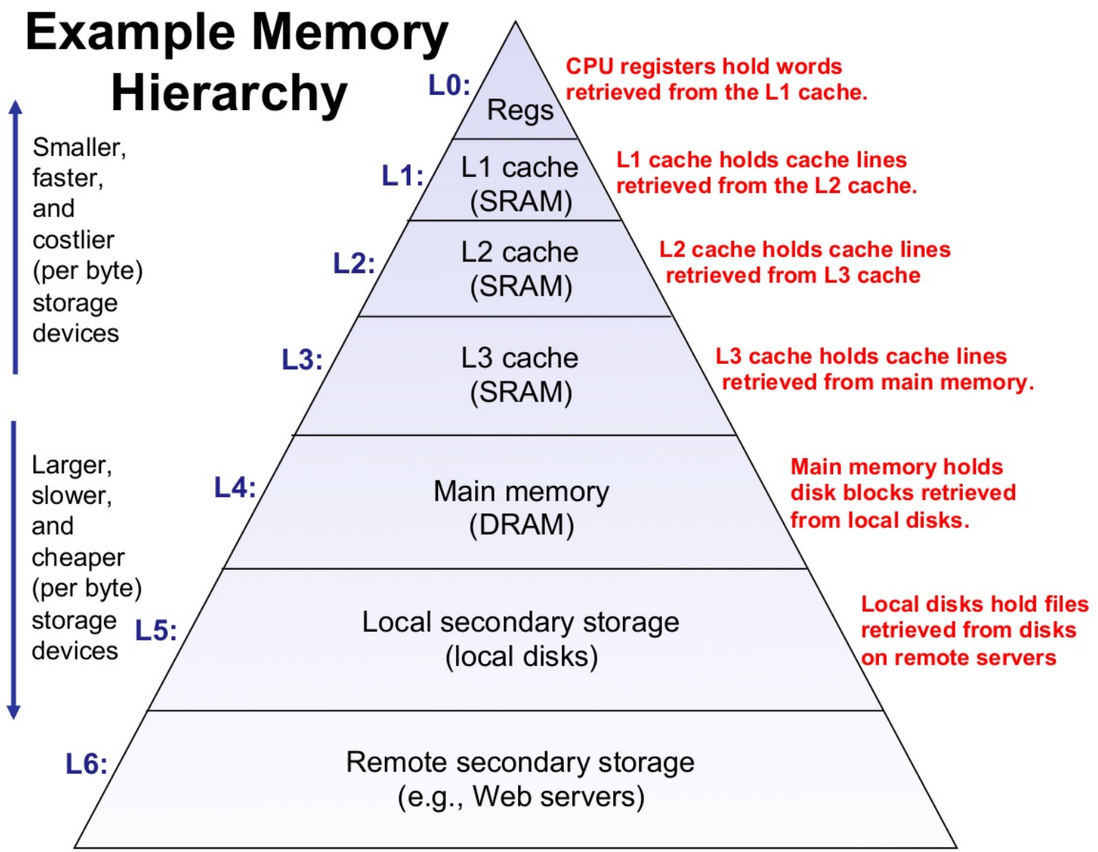
- Muitos programas dependem basicamente da memória RAM, mas a memória RAM não existe, é uma abstração limitada. O que existe é um complexo sistema hierárquico de memórias diferentes que tenta nos dar a impressão de alta capacidade de armazenamento e alta performance.
    * Deve ser gerenciada: alocada e desalocada

- Bugs no uso da memória são perniciosos!
    * Efeitos são distantes no tempo e no espaço
    * Difícil de debugar

- A performance da memória não é uniforme
    * Memória virtual e cache afetam a performance
    * Adaptar um programa às características da memória pode levar a ganhos expressivos de performance

A memória RAM é uma abstração!

- Muitos programas dependem basicamente da memória RAM, mas a memória RAM não existe, é uma abstração limitada. O que existe é um complexo sistema hierárquico de memórias diferentes que tenta nos dar a impressão de alta capacidade de armazenamento e alta performance.
    * Deve ser gerenciada: alocada e desalocada



Bugs no referenciamento da memória são perniciosos!

- Bugs no uso da memória são perniciosos!
    * Efeitos são distantes no tempo e no espaço
    * Difícil de debugar

```c
/**
 * CR6.190A: Arquitetura de Computadores
 * https://cursos.computacaoraiz.com.br
 *
 * erro_de_memoria.c
 * O array não tem fim?
 */

#include <stdio.h>

int main(void)
{
    int a[2] = {22, 33};

    for (int i = 0; i <= 10; ++i)
    {
        a[i] = i + 1;
        printf("a[%d] = %d\n", i, a[i]);
    }

    return 0;
}
```

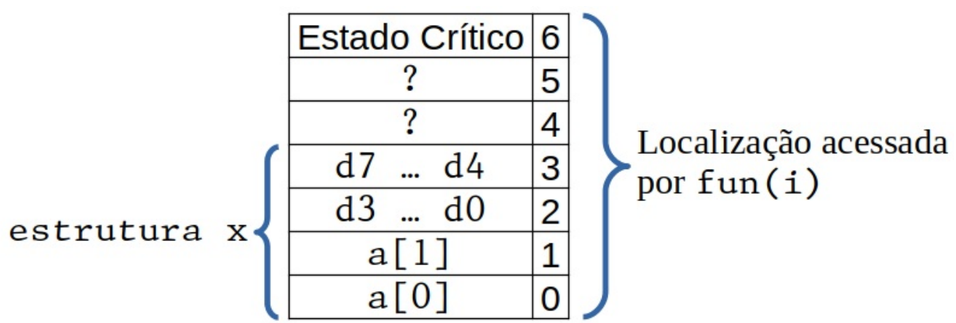
```
[abrantesasf@cosmos ~/cr6.190a/introducao]$ ./erro_de_memoria
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
a[5] = 6
a[6] = 7
a[7] = 8
a[8] = 9
a[9] = 10
a[10] = 11
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Bugs no referenciamento da memória são perniciosos!

- Bugs no uso da memória são perniciosos!
    * Efeitos são distantes no tempo e no espaço
    * Difícil de debugar

```c
/**
 * CR6.190A: Arquitetura de Computadores
 * https://cursos.computacaoraiz.com.br
 *
 * erro_de_memoria2.c
 * Erro intermitente é o pior.
 *
 * Código retirado de:
 * Computer Systems: A Programmer's Perspective, 3ª ed (CS:APP3e)
 * https://csapp.cs.cmu.edu/3e/instructors.html
 */

#include <stdio.h>

typedef struct
{
    int a[2];
    double d;
} estrutura;

double fun(int i)
{
    volatile estrutura x;
    x.d = 3.14;
    x.a[i] = 1073741824;
    return x.d;
}

int main(void)
{
    for (int i = 0; i < 10; ++i)
        printf("fun(%d) = %.2f\n", i, fun(i));
}
```

```
[abrantesasf@cosmos ~/cr6.190a/introducao]$ ./erro_de_memoria2
fun(0) = 3.14
fun(1) = 3.14
fun(2) = 3.14
fun(3) = 2.00
fun(4) = 3.14
fun(5) = 3.14
*** stack smashing detected ***: terminated
Aborted (core dumped)
```



Bugs no referenciamento da memória são perniciosos!

- Bugs no uso da memória são perniciosos!

- C e C++ não fornecem nenhuma proteção de memória
  * não verificam limites de arrays
  * não verificam ponteiros inválidos
  * é fácil abusar de malloc/free

- Levam a bugs estranhos e difíceis de encontrar
  * se um bug terá ou não efeito depende do seu sistema e do compilador, tornando tudo mais difícil (erros intermitentes)
  * ação à distância (tempo e espaço): o objeto corrompido pode não ter relação lógica nenhuma com o objeto sendo acessado

- Como evitar?
  * entender como a memória funciona e como prevenir
  * usar ferramentas para diagnóstico (ex.: valgrind)
  * usar Python, Java, Ruby, ML... 😒

A performance da memória não é uniforme!

- Programadores geralmente estão mais preocupados em saber se o algoritmo é correto, se as estruturas de dados são adequadas, e com a complexidade assintótica de seus programas (big-O). Isso é perfeito e deve ser realmente a preocupação inicial, mas:
    * É necessário também otimizar para o baixo nível, para o uso de memória
    * É importante entender o que o computador faz, como ele faz, e o que faz ele rodar mais rápido ou mais devagar
    * É fácil obter até 10x mais performance dependendo apenas de como o código é escrito
    * Otimizar: algoritmo, estrutura de dados, procedimentos, loops

- A performance da memória não é uniforme
    * Memória virtual e cache afetam a performance
    * Adaptar um programa às características da memória pode levar a ganhos expressivos de performance
    * Fatores constantes também importam
    * É necessário entender o funcionamento para otimizar corretamente

A performance da memória não é uniforme!

$$A_{4096\times4096} = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,4095} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,4095} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{4095,0} & a_{4095,1} & a_{4095,2} & \cdots & a_{4095,4095} \end{pmatrix}$$

$$B_{4096\times4096} = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & \cdots & b_{0,4095} \\ b_{1,0} & b_{1,1} & b_{1,2} & \cdots & b_{1,4095} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{4095,0} & b_{4095,1} & b_{4095,2} & \cdots & b_{4095,4095} \end{pmatrix}$$

$$B = A$$

(16.777.216 operações de cópia)

A performance da memória não é uniforme!

$$\begin{pmatrix} \boxed{b_{0,0}} & b_{0,1} & b_{0,2} & \cdots & b_{0,4095} \\ \boxed{b_{1,0}} & b_{1,1} & b_{1,2} & \cdots & b_{1,4095} \\ \boxed{\vdots} & \vdots & \vdots & \ddots & \vdots \\ \boxed{b_{4095,0}} & b_{4095,2} & b_{4095,3} & \cdots & b_{4095,4095} \end{pmatrix} = \begin{pmatrix} \boxed{a_{0,0}} & a_{0,1} & a_{0,2} & \cdots & a_{0,4095} \\ \boxed{a_{1,0}} & a_{1,1} & a_{1,2} & \cdots & a_{1,4095} \\ \boxed{\vdots} & \vdots & \vdots & \ddots & \vdots \\ \boxed{a_{4095,0}} & a_{4095,2} & a_{4095,3} & \cdots & a_{4095,4095} \end{pmatrix}$$

$$\begin{pmatrix} \boxed{b_{0,0}} & \boxed{b_{0,1}} & \boxed{b_{0,2}} & \cdots & \boxed{b_{0,4095}} \\ b_{1,0} & b_{1,1} & b_{1,2} & \cdots & b_{1,4095} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{4095,0} & b_{4095,2} & b_{4095,3} & \cdots & b_{4095,4095} \end{pmatrix} = \begin{pmatrix} \boxed{a_{0,0}} & \boxed{a_{0,1}} & \boxed{a_{0,2}} & \cdots & \boxed{a_{0,4095}} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,4095} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{4095,0} & a_{4095,2} & a_{4095,3} & \cdots & a_{4095,4095} \end{pmatrix}$$

```c
/**
 * CR6.190A: Arquitetura de Computadores
 * https://cursos.computacaoraiz.com.br
 *
 * copiar_matrizes2.c
 * B = A
 *
 */

#include <stdio.h>

#define N 4096

int main(void)
{
    static int matrizA[N][N];
    static int matrizB[N][N];

    for (int l = 0; l < N; ++l)
        for (int c = 0; c < N; ++c)
            matrizA[l][c] = 1;

    for (int c = 0; c < N; ++c)
        for (int l = 0; l < N; ++l)
            matrizB[l][c] = matrizA[l][c];
}
```

$$B = A$$

$$O(n^2)$$

```c
/**
 * CR6.190A: Arquitetura de Computadores
 * https://cursos.computacaoraiz.com.br
 *
 * copiar_matrizes1.c
 * B = A
 *
 */

#include <stdio.h>

#define N 4096

int main(void)
{
    static int matrizA[N][N];
    static int matrizB[N][N];

    for (int l = 0; l < N; ++l)
        for (int c = 0; c < N; ++c)
            matrizA[l][c] = 1;

    for (int l = 0; l < N; ++l)
        for (int c = 0; c < N; ++c)
            matrizB[l][c] = matrizA[l][c];
}
```

```
[abrantesasf@cosmos ~/cr6.190a/introducao]$ time ./copiar_matrizes2

real    0m0,543s
user    0m0,489s
sys     0m0,052s
```

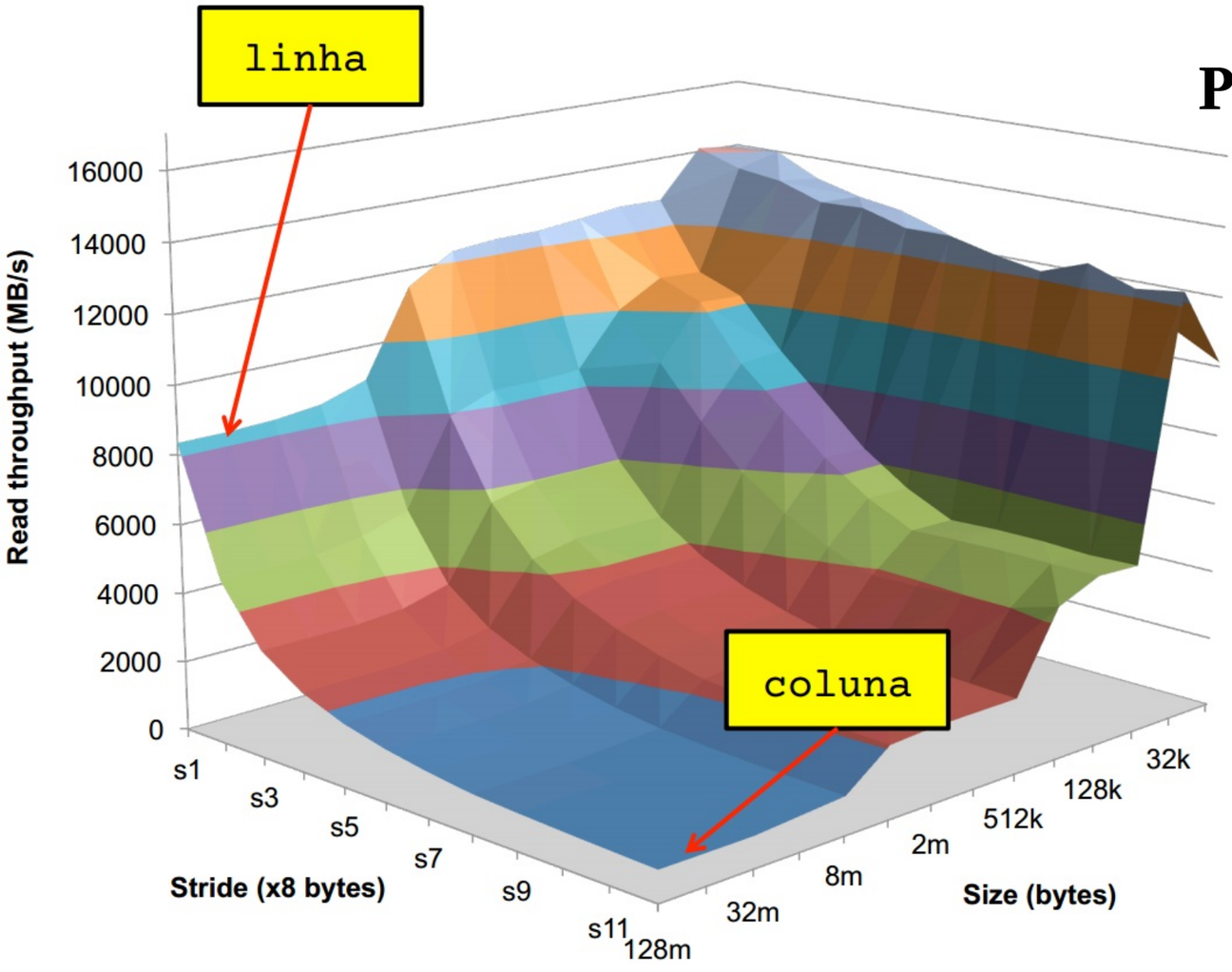
```
[abrantesasf@cosmos ~/cr6.190a/introducao]$ time ./copiar_matrizes1

real    0m0,133s
user    0m0,097s
sys     0m0,037s
```

Copiar por coluna foi 4 VEZES MAIS LENTO do que copiar por linha! Por que isso ocorreu?

A performance da memória não é uniforme!

$$\begin{pmatrix} \boxed{b_{0,0}} & b_{0,1} & b_{0,2} & \cdots & b_{0,4095} \\ \boxed{b_{1,0}} & b_{1,1} & b_{1,2} & \cdots & b_{1,4095} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \boxed{b_{4095,0}} & b_{4095,2} & b_{4095,3} & \cdots & b_{4095,4095} \end{pmatrix} = \begin{pmatrix} \boxed{a_{0,0}} & a_{0,1} & a_{0,2} & \cdots & a_{0,4095} \\ \boxed{a_{1,0}} & a_{1,1} & a_{1,2} & \cdots & a_{1,4095} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \boxed{a_{4095,0}} & a_{4095,2} & a_{4095,3} & \cdots & a_{4095,4095} \end{pmatrix}$$

$$\begin{pmatrix} \boxed{b_{0,0}} & \boxed{b_{0,1}} & \boxed{b_{0,2}} & \cdots & \boxed{b_{0,4095}} \\ b_{1,0} & b_{1,1} & b_{1,2} & \cdots & b_{1,4095} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{4095,0} & b_{4095,2} & b_{4095,3} & \cdots & b_{4095,4095} \end{pmatrix} = \begin{pmatrix} \boxed{a_{0,0}} & \boxed{a_{0,1}} & \boxed{a_{0,2}} & \cdots & \boxed{a_{0,4095}} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,4095} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{4095,0} & a_{4095,2} & a_{4095,3} & \cdots & a_{4095,4095} \end{pmatrix}$$



Para entender porque a lentidão, é necessário:

* **Entender a hierarquia da memória**
* **Entender como a performance depende dos padrões de acesso à memória (incluindo como percorrer um array multidimensional)**

3ª verdade nua e crua: você precisa entender a arquitetura da memória!

Para ser um bom programador:

- **Entender a hierarquia de memória**
- **Entender como a arquitetura da memória e linguagens como C podem levar a bugs de referenciamento de memória que são complicados de debugar e que podem ser distantes do tempo e espaço**
- **Entender que a performance da memória não é uniforme e que é necessário otimizar para o baixo nível também**

"Entender a representação em nível de máquina, na memória, das estruturas de dados e como elas funcionam faz uma grande diferença na sua habilidade de evitar e lidar com problemas de referenciamento de memória e vulnerabilidades no seu programa." Randal Bryant

hacker.jpg (Image by Robinraj Premchand from Pixabay: https://pixabay.com/photos/hacking-cyber-blackandwhite-crime-2903156)