Computação Concorrente (DCC/UFRJ)

Aula 12: Programação concorrente em GPUs: visão geral

Prof. Silvana Rossetto

12 de junho de 2012

Motivação

- **GPU** (*Graphics Processing Units*): hardware inicialmente desenhado para processamento gráfico
- ...hoje coprocessadores programáveis (*massive parallelism*)
  - centenas de processadores
  - milhares de threads
- **GPU Computing**: uso de GPUs para computação de propósito geral (não requer o uso da API gráfica tradicional e do modelo de pipeline gráfico)

Oposição à abordagem precedente "General Purpose Computation on GPU (**GPGPU**)": programação da GPU usando uma API e pipeline gráfico para executar tarefas não gráficas

Arquitetura GPU unificada

- As arquiteturas GPU unificadas são baseadas em um **array paralelo** de vários **processadores programáveis**
- *GPUs manycores*: executar muitas threads paralelas eficientemente em muitos núcleos de processamento

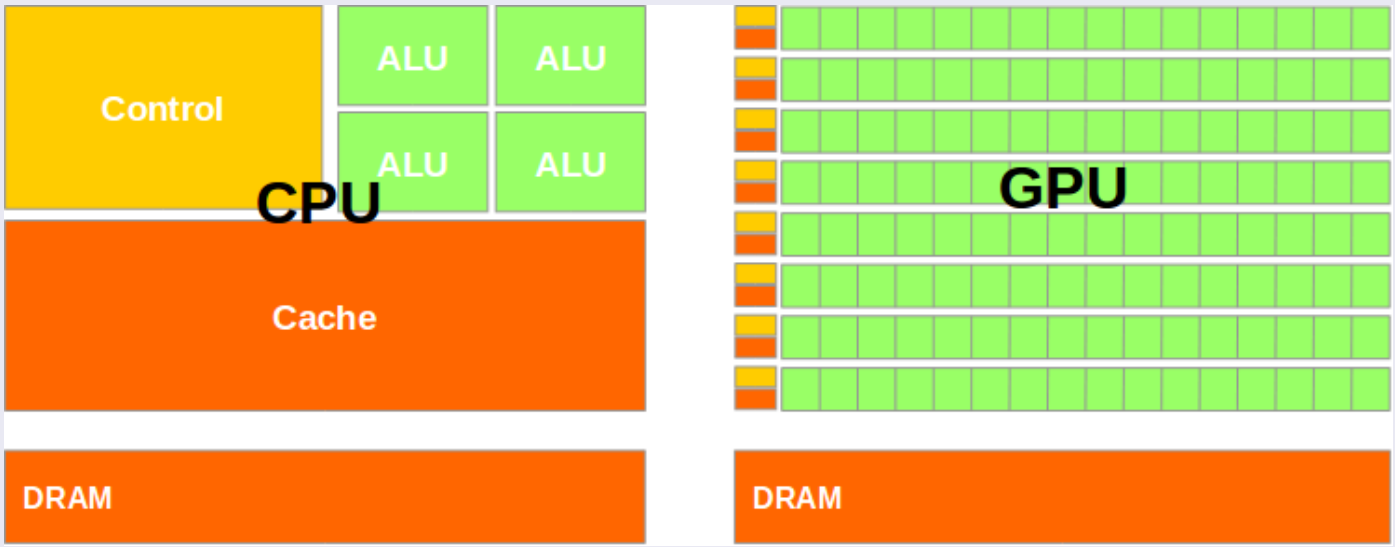


Figura: Programming Massively Parallel Processors: A Hands-on Approach, 2010 David B. Kirk/NVIDIA Corporation and Wen-mei Hwu. Elsevier Inc.

Uso de GPU Computing

- GPUs são em geral adequadas para problemas com forte **paralelização de dados**, em particular para o caso de grandes quantidades de dados
- Processamento de imagem e vídeo (codificação, decodificação, etc.) são boas aplicações de GPUs: acesso aos dados é **regular** e com bom uso de **localidade**

CUDA (*Compute Unified Device Architecture*)

- **Plataforma de software para GPU** e outros processadores paralelos (programação em C ou C++)
- **Modelo de programação SPMD** (*Simple Program Multiple Data*):
  - o programador escreve o código para uma thread, **o ambiente instancia e executa o código em várias threads em paralelo**, nos vários processadores da GPU
- O programador gerencia a transferência de dados entre memórias (CPU e GPU)

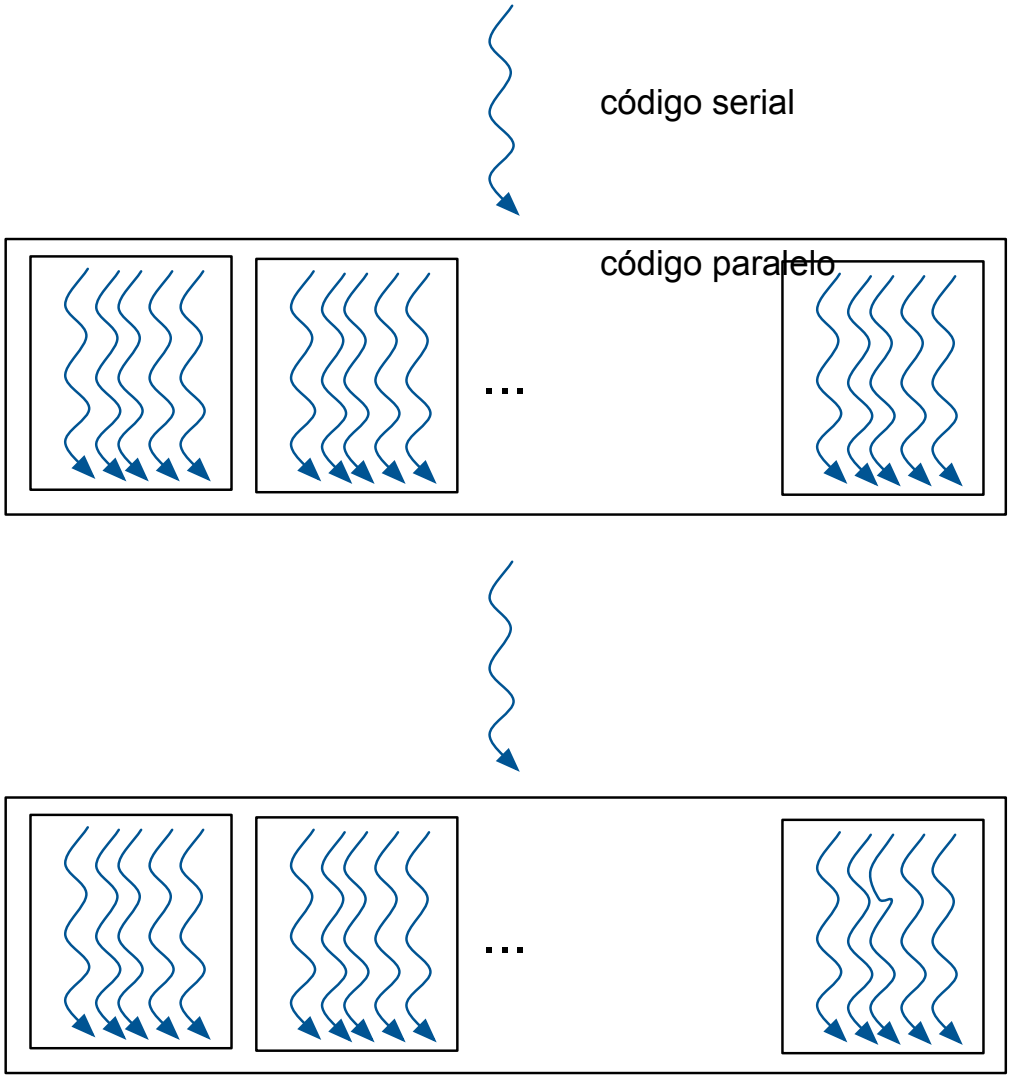
Hoje é possível usar as GPUs como processadores gráficos e processadores convencionais ao mesmo tempo, e combinar esse uso em aplicações de visão computacional

Paradigma de programação CUDA

- **Programas seriais** que chamam **kernels paralelos**
- O programador organiza as threads em uma hierarquia de **blocos de threads** e **grades de blocos de threads**

Um **kernel** executa em paralelo porque é instanciado em um conjunto de threads paralelas

Modelo de execução CUDA



Bloco de threads

Um **bloco de thread** é um conjunto de threads concorrentes que cooperam usando:

- mecanismo de **sincronização por barreira**
- acesso **compartilhado a um espaço de memória privado por bloco**

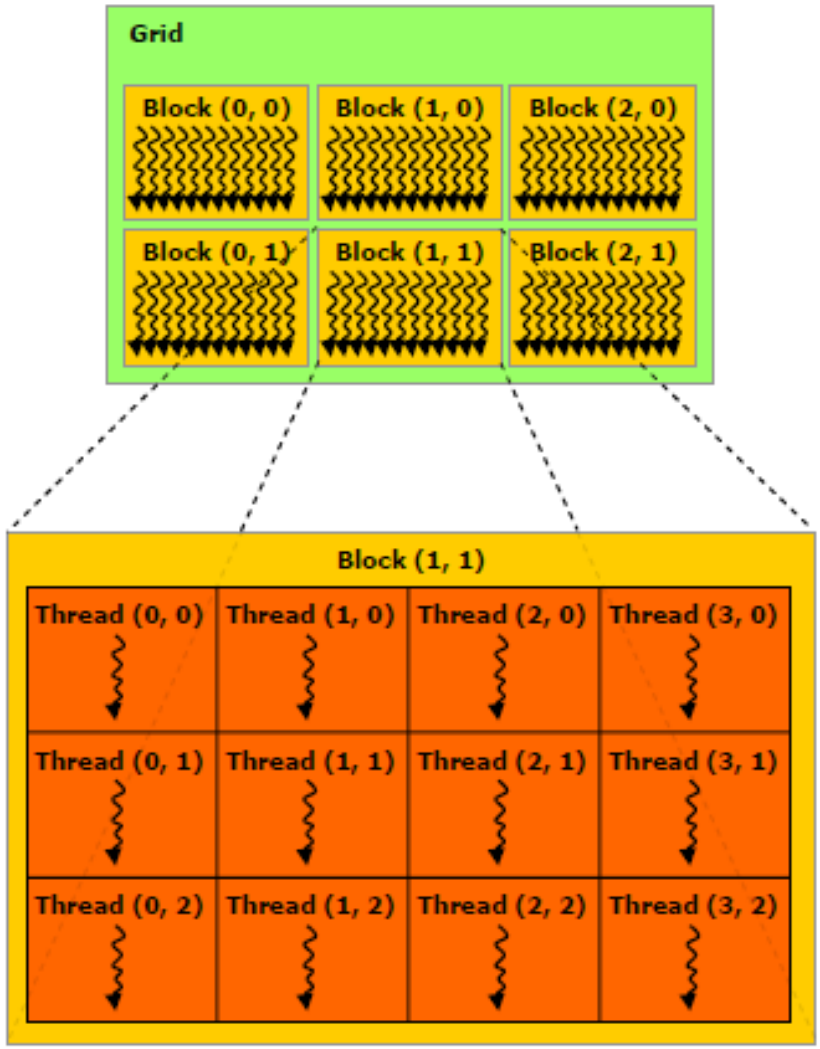
Grade de blocos de thread

- Uma **grade** é um conjunto de blocos de threads que podem ser executados de forma independente e em paralelo
- Quando o programador **invoca um kernel**, ele especifica o **número de threads por bloco** e o **número de blocos na grade**

- A cada thread é atribuído um identificador único — **threadIdx** — dentro do seu bloco (0,1,2,...blockDim-1)
- E a cada bloco de thread é atribuído um identificador único — **blockIdx** — dentro da grade
- (CUDA (atual) permite blocos com até 512 threads)

Blocos de threads e grades podem ter **1, 2 ou 3 dimensões**, acessadas via índices .x, .y e .z

Grade de blocos de threads



C estendido

- A declaração `__global__` indica que o procedimento é um ponto de entrada de um kernel
- Os kernels são disparados com a seguinte sintaxe: `kernel <<<dimGrid, dimBlock>>> (..lista de parâmetros..)`
- **dimGrid** e **dimBlock** são vetores de 3 elementos que especificam as dimensões da grade e do bloco de threads (o default é 1)

Exemplo de código sequencial

Calcular $Y = aX + Y$ em um loop sequencial:

```
void calc_seq(int n, float a, float *x, float *y){
   for(int i=0; i<n; i++)
      y[i] = a * x[i] + y[i];
}
//invocação da função
calc_seq(n, 2.0, X, Y);
```

Exemplo de código paralelo

Calcular $Y = aX + Y$ em paralelo usando CUDA:

```
__global__
void calc_par(int n, float a, float *x, float *y){
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if(i<n)
      y[i] = a * x[i] + y[i];
}
//invocação da função (256 threads por bloco)
int nblocks = (n+255) / 256;
calc_par <<<nblocks,256>>> (n, 2.0, X, Y);
```

Execução de um kernel

- Necessidade de orquestrar a transferência de dados entre memória principal e dispositivo:
  1. transfere dados para dispositivo
  2. executa o kernel
  3. transfere resultados para memória principal
- **cudaMalloc** e **cudaFree**: aloca/desaloca memória global no dispositivo
- **cudaMemcpy**: transfere dados em ambas as direções

Ambiente de execução CUDA

- A execução paralela e a gerência de threads é automática
- As threads de um bloco executam concorrentemente e podem sincronizar suas ações via chamada `__syncthreads()`
- Depois de passarem pela barreira, as threads garantidamente "vêem" todas as últimas operações de escrita executadas palas outras threads antes da barreira

- A coordenação do acesso à memória global pode ser feita usando **operações atômicas**

Aplicações com várias grades

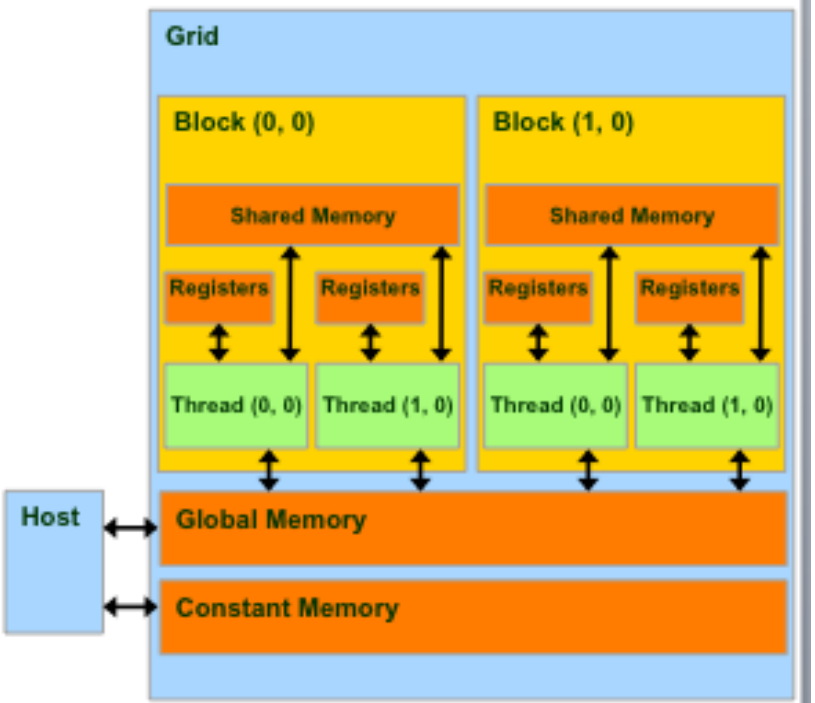
- Uma aplicação pode executar **várias grades independentes, ou dependentes uma da outra**
- Grades independentes podem executar concorrentemente, tendo recursos de hardware suficientes
- Grades dependentes executam sequenciamente, com uma **barreira inter-kernel implícita** entre elas, garantindo que todos os blocos da primeira grade completam antes dos da segunda e sucessivamente

Espaços de memória

- Cada thread possui uma **memória local privada**
- Cada bloco de threads possui uma **memória compartilhada** visível por todas as threads do bloco (mesmo tempo de vida do bloco)
- Todas as threads têm acesso à **memória global**

Programas declaram variáveis nas memórias global e compartilhada usando os qualificadores de tipo: __**device**__ e __**shared**__

Organização de Memória



- threads podem acessar:
  - registradores
    - (ex., para acomodar o máximo de 768 threads, no maximo 10 regs por thread)
  - memória compartilhada (entre threads do bloco)
    - 16K por SM
  - memória local
  - memória global

Restrições

Por questões de **eficiência** e **simplicidade de implementação**, o modelo de programação CUDA tem algumas restrições:

- threads só podem ser criadas invocando um kernel paralelo (simplifica o escalonamento)
- funções recursivas não são (atualmente) permitidas dentro dos kernels (espaço de memória requerido para a pilha de chamadas)

Considerações

- **Modelo de concorrência com granularidade mais fina**:
  - custo baixo de criação de threads
  - overhead de escalonamento zero (multithreaded em hw)
  - sincronização por barreira eficiente (uma única instrução)
- **Programador responsável por muitos detalhes**:
  - controle de memória local e global
  - controle de execução sincronizada
- API razoavelmente baixo nível

Multiplicação de matrizes (CPU, sequencial)

```
1  void Matrix_Mult(float **a1, float **a2, float **a3)
2  {
3      int i = 0;
4      int j = 0;
5      int k = 0;
6          for(i = 0; i < N; i++)
7              for( j = 0; j < N; j++)
8                  for( k = 0; k < N; k++)
9                      a3[i][j] +=  a1[i][k] * a2[k][j];
10 }
```

Multiplicação de matrizes (CPU, multithreading)

Fixa o número total de threads na aplicação e atribui a cada thread o cálculo de uma ou mais linhas da matriz resultante

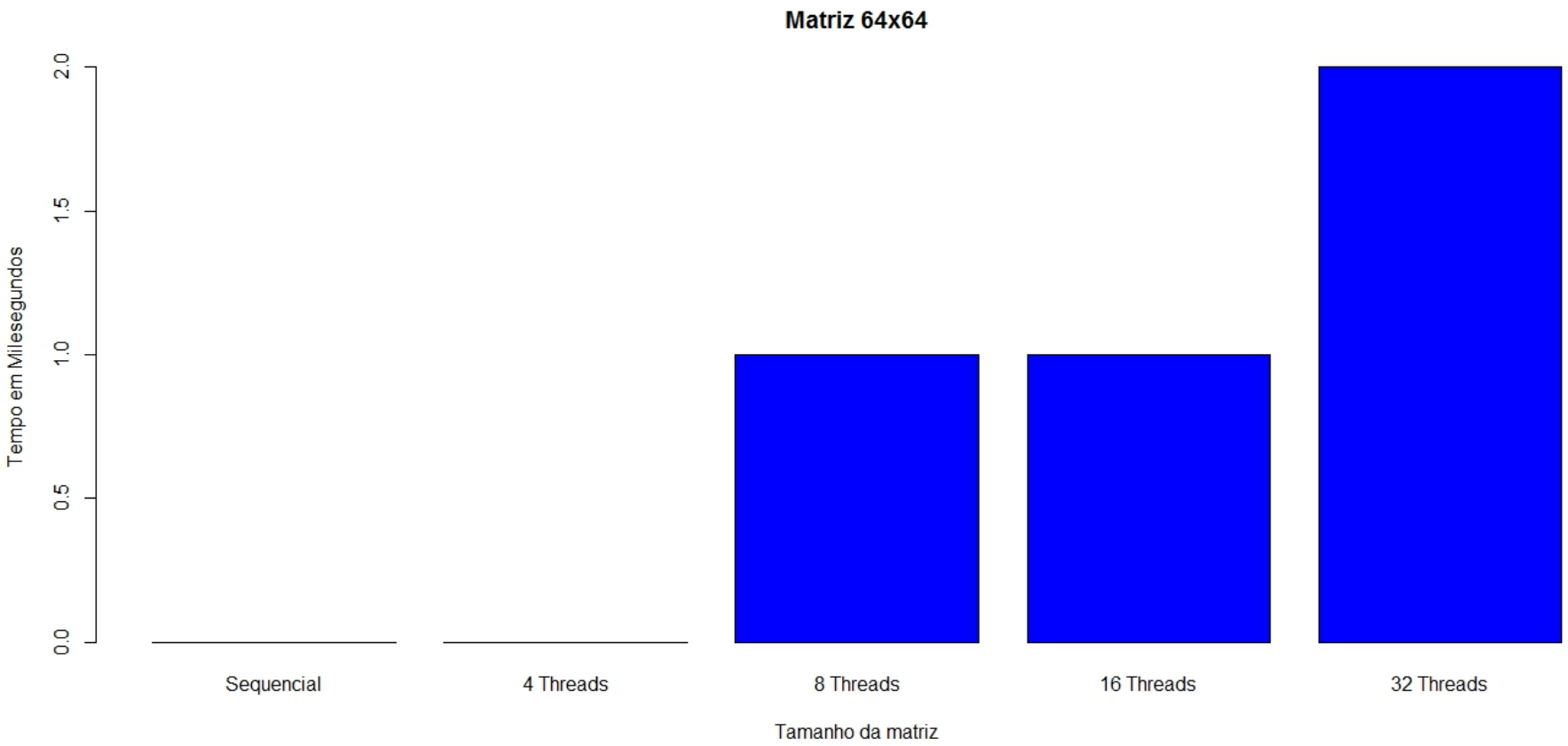
```
1  void *Matrix_Mult(void * arg)
2  {
3      int id = *(int *)(arg);
4      int i = 0;
5      int j = 0;
6      int k = 0;
7          for(i = 0; i < nvezes; i++)
8              for( j = 0; j < N; j++)
9                  for( k = 0; k < N; k++)
10                     C[i+id*nvezes][j] +=  A[i+id*nvezes][k] * B[k][j];
11         return 0;
12 }
```

Multiplicação de matrizes (CPU, multithreading)

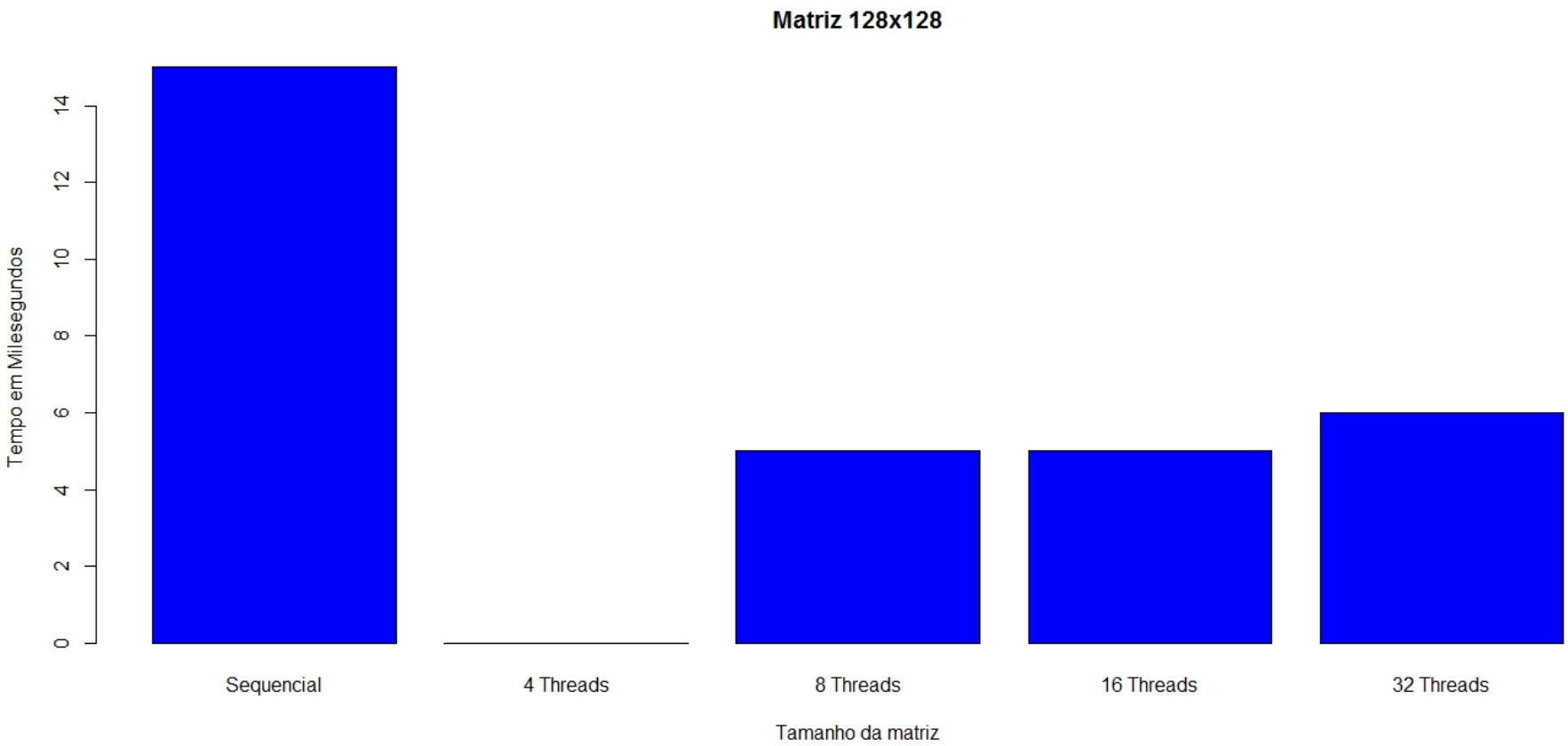
Cada thread calcula um elemento da matriz de saída (i.e., qtde de threads é variável)

```
1  void *Matrix_Mult(void * arg)
2  {
3      int id = *(int *)(arg);
4      int k = 0;
5      int n=N;
6
7      for( k = 0; k < N; k++)
8          C[id/n][id%n] +=  A[id/n][k] * B[k][id%n];
9
10 }
```

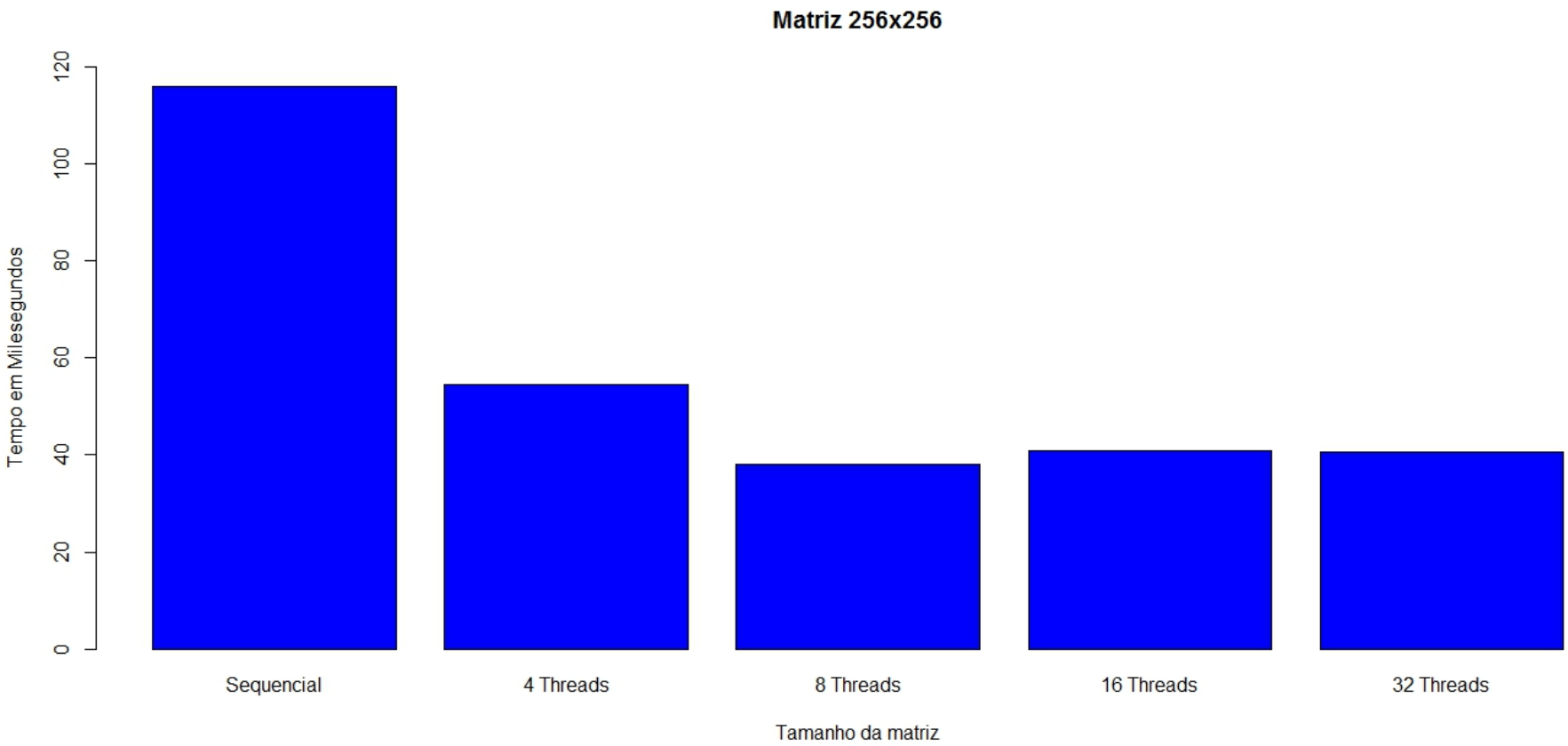
Multiplicação de matrizes: sequencial X multithreading



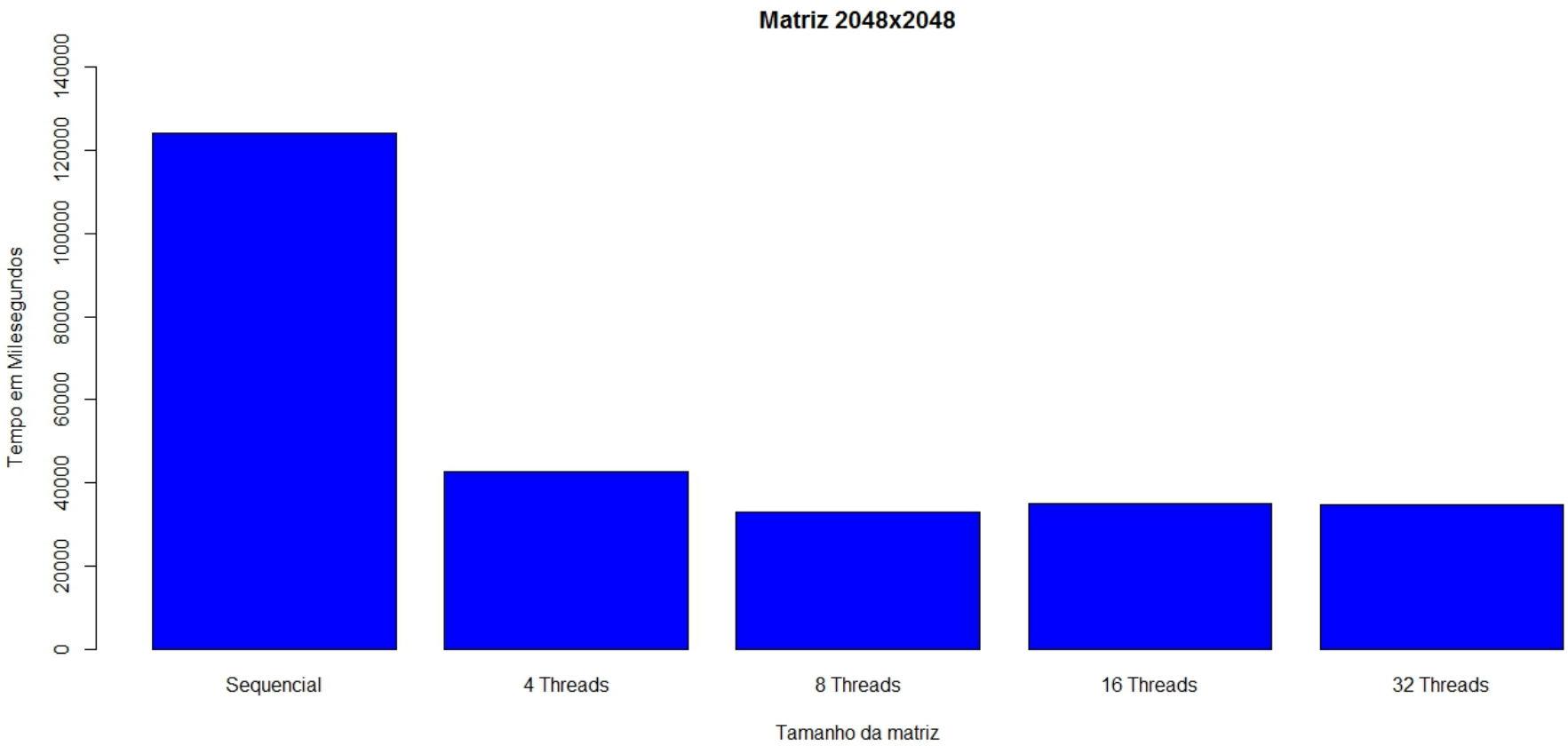
Multiplicação de matrizes: sequencial X multithreading



Multiplicação de matrizes: sequencial X multithreading



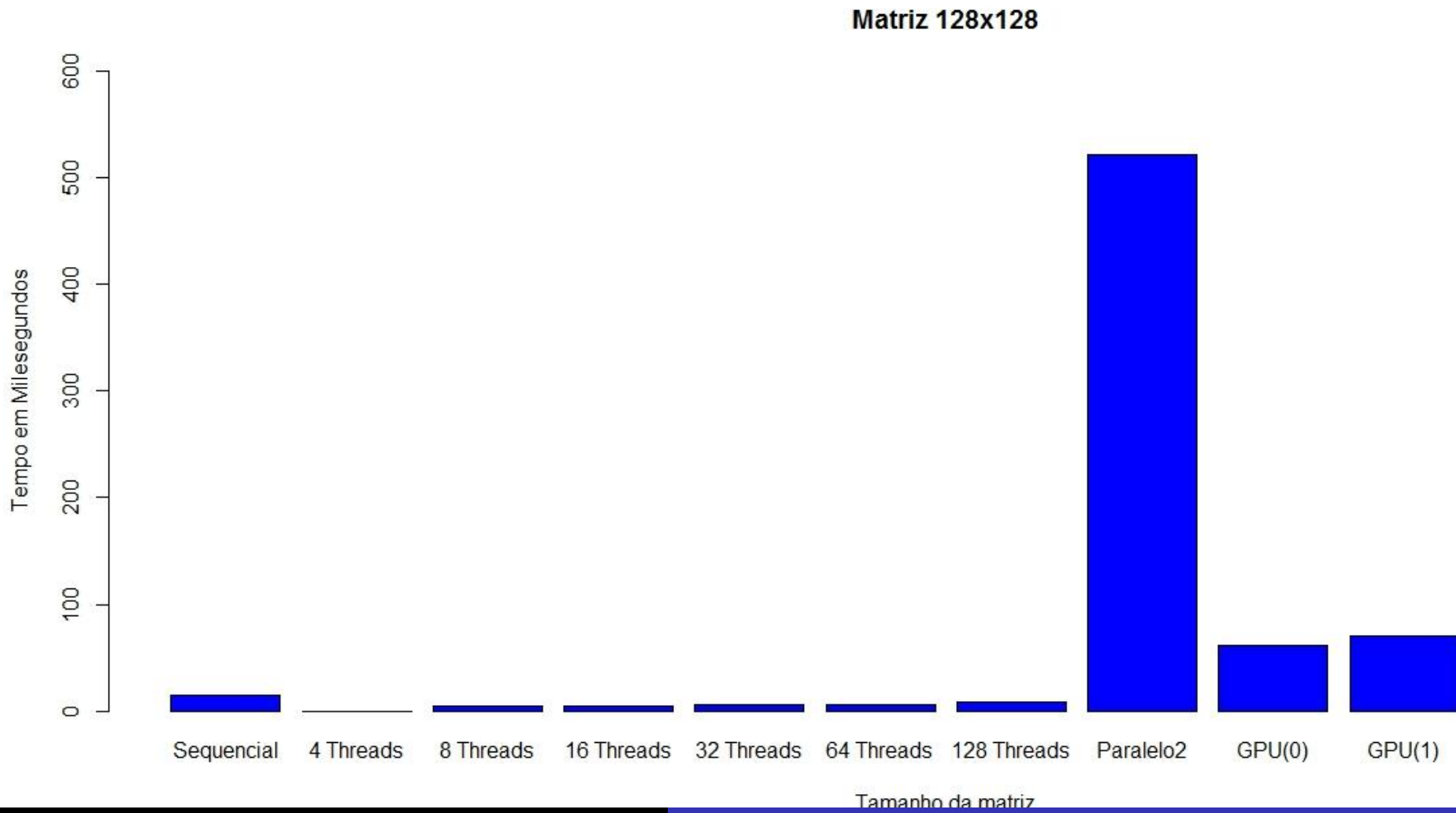
Multiplicação de matrizes: sequencial X multithreading



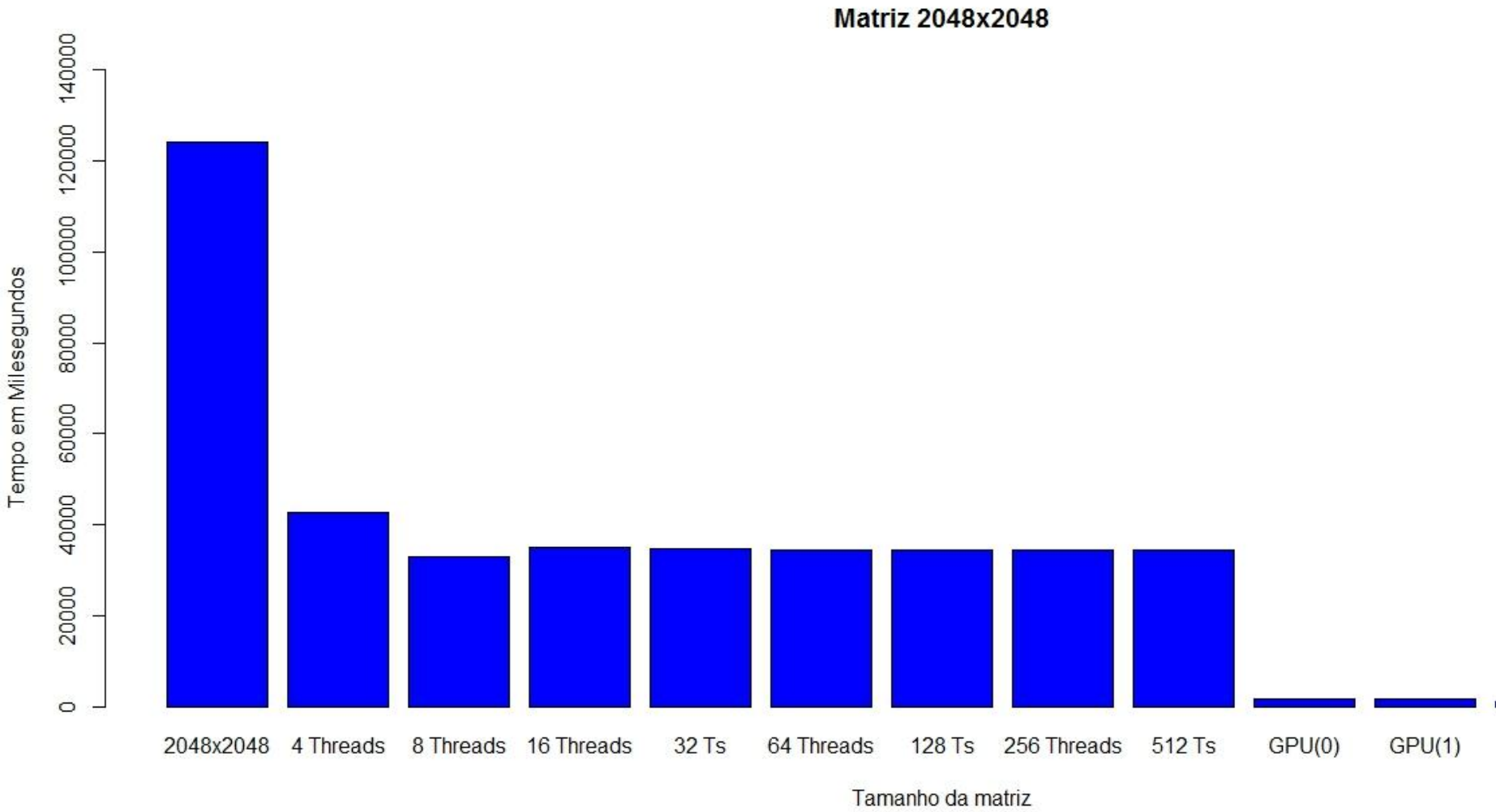
Multiplicação de matrizes (GPU)

```
1   __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
2   {
3
4       int Cvalue = 0;
5       int row = blockIdx.y * blockDim.y + threadIdx.y;
6       int col = blockIdx.x * blockDim.x + threadIdx.x;
7
8       for (int e = 0; e < A.width; ++e){
9           if(row * A.width + e<N*N && e * B.width + col <N*N)
10              Cvalue += A.elements[row * A.width + e]
11                  * B.elements[e * B.width + col];
12      }
13      if((row * C.width + col)<N*N)
14          C.elements[row * C.width + col] = Cvalue;
15
16  }
```

Multiplicação de matrizes: tempos totais

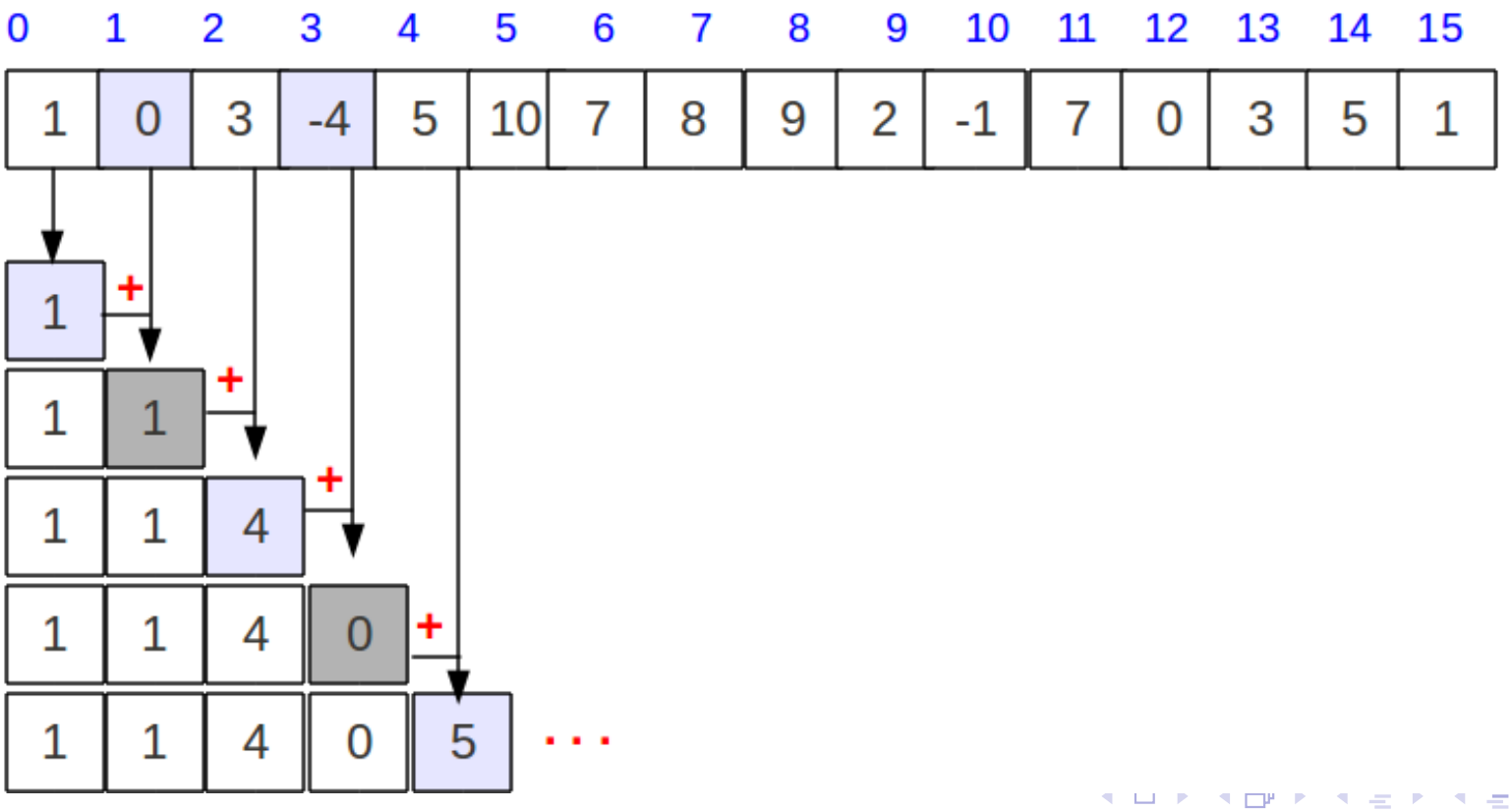


Multiplicação de matrizes: tempos totais



Exemplo: soma de prefixos paralela

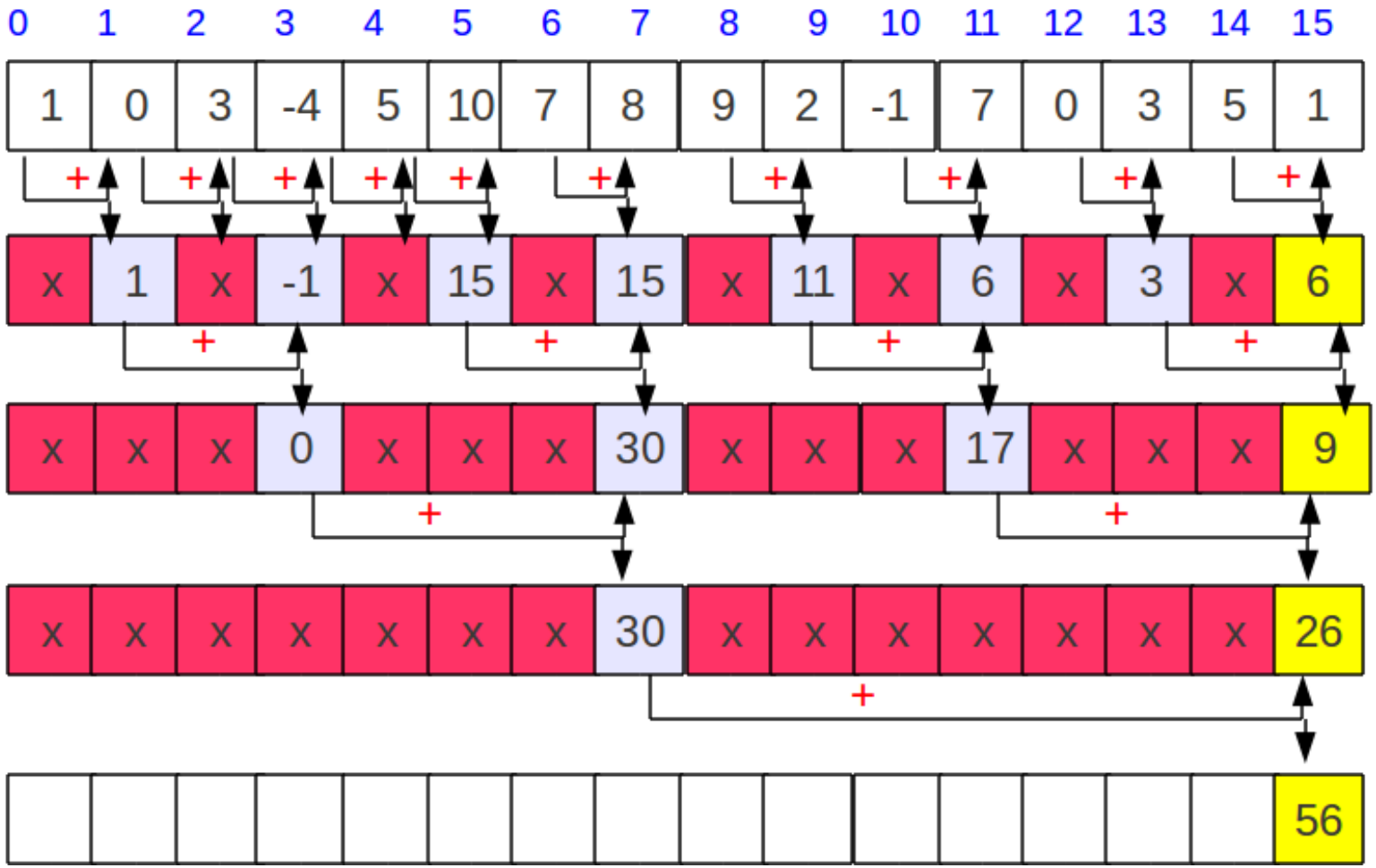
- Operação usual em aplicações paralelas
- Cada elemento do vetor de saída contém a soma de todos os seus antecessores



Código soma de prefixos sequencial

```
__host__ void scan_seq (int *x, int n) {
   for(int i=1; i<n; i++)
      x[i] = x[i-1] + x[i];
}
```

Soma de prefixos pararela



Código soma de prefixos paralela

```
__device__ void scan_par(int *x){
    int i = threadIdx.x;
    int n = blockDim.x;
    for(int offset=1; offset<n; offset*=2) {
      int aux;
      if(i >= offset) { aux = x[i-offset]; }
      __syncthreads;
      if(i >= offset) { x[i] = aux + x[i]; }
      __syncthreads;
    }
}
```

Referências bibliográficas

1. *Computer Organization and Design – the hardware/software interface*, Patterson e Hennessy, ed. 4, 2009
2. Notas de sala de aula da profa. Noemi Rodriguez (Puc-Rio), novembro de 2011
3. *Programming Massively Parallel Procesors*, D. Kirk e W. Hwu, Morgan-Kaufmann, 2010