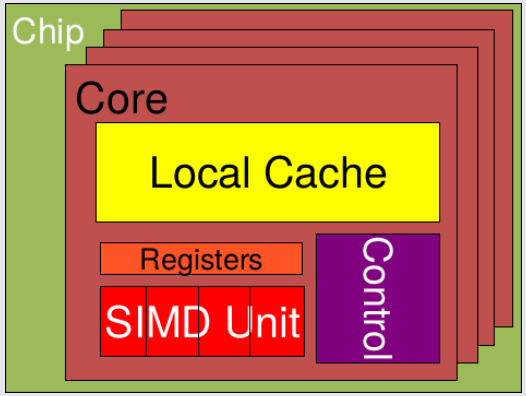
INF 310 – Programação Concorrente e Distribuída

CUDA

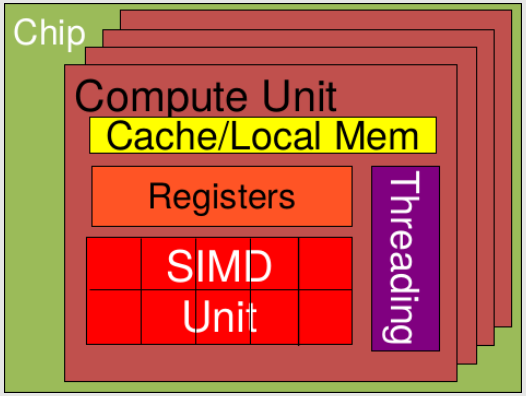
Professor: Vitor Barbosa Souza
vitor.souza@ufv.br

Computação Heterogênea

- Uso da melhor abordagem de acordo com o processamento a ser realizado
- Uso de abordagens com diferentes orientações
  - Orientado a latência
  - Orientado a *throughput*
  - Processamento digital de sinais
  - Blocos lógicos configuráveis, etc.



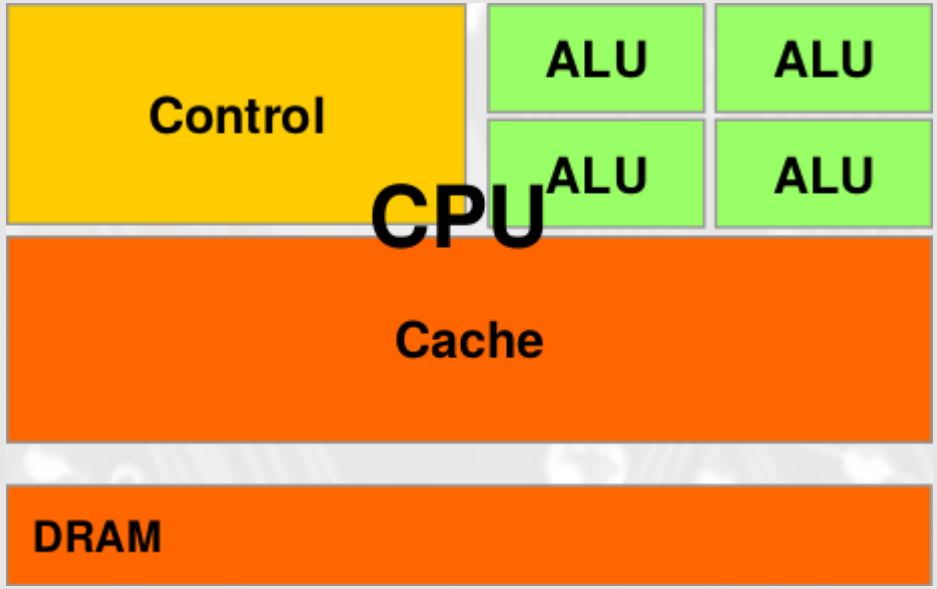
CPU (latência)



GPU (throughput)

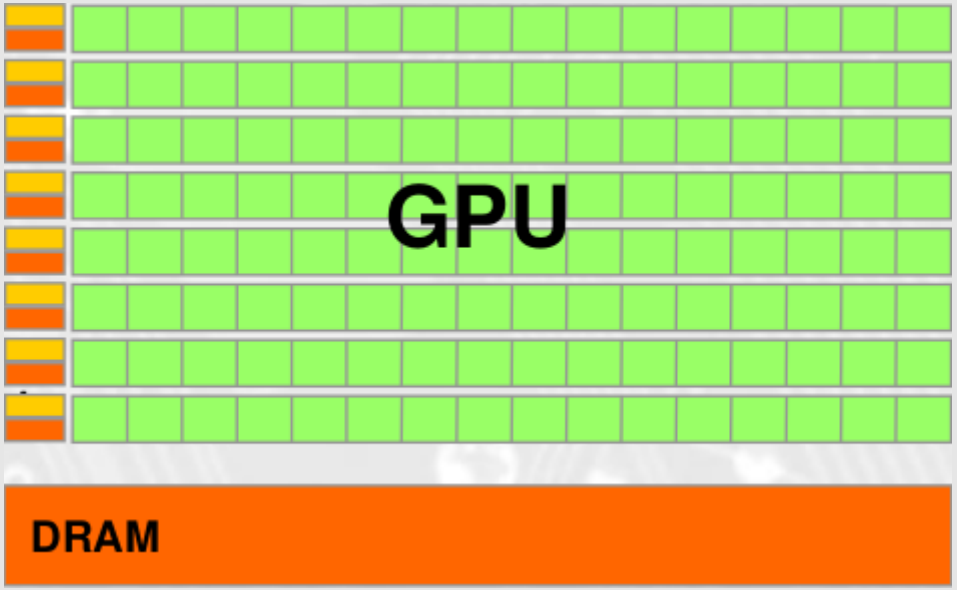
Computação Heterogênea

- CPU
  - Cache grande
    - Evitar latência alta no acesso à memória principal
  - Controle sofisticado
    - *Branch prediction*
    - *Data forwarding*
  - ALU de alta capacidade



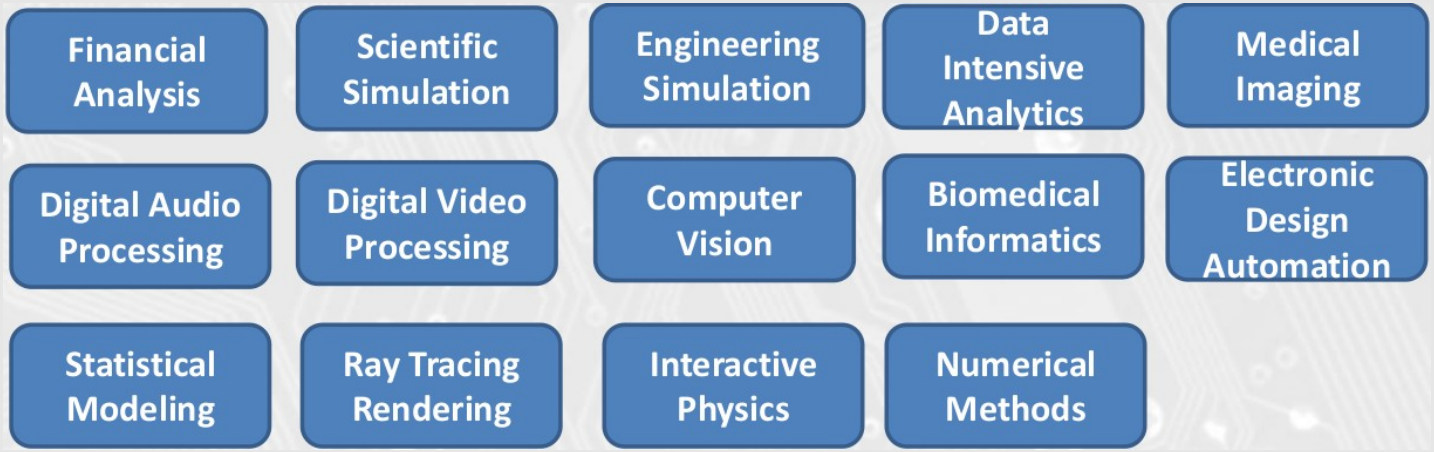
Computação Heterogênea

- GPU
  - Cache pequena
  - Controle simples
    - Sem *branch prediction*
    - Sem *data forwarding*
  - Muitas ALUs
    - Latência alta
    - Pipeline
    - Vazão alta (*throughput*)
  - Grande número de threads para compensar latência



Computação Heterogênea

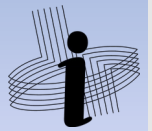
- Quando utilizar CPU ou GPU?
  - Códigos sequênciais: CPUs podem ser ordens de grandeza mais rápidos que GPU
  - Códigos paralelos: GPUs podem ser ordens de grandeza mais rápidos que CPU
- Exemplos de uso de GPU



Paralelismo de dados

- Arquiteturas de hardware
  - SISD (*Single Instruction, Single Data*)
  - SIMD (*Single Instruction, Multiple Data*)
  - MIMD (*Multiple Instruction, Multiple Data*)

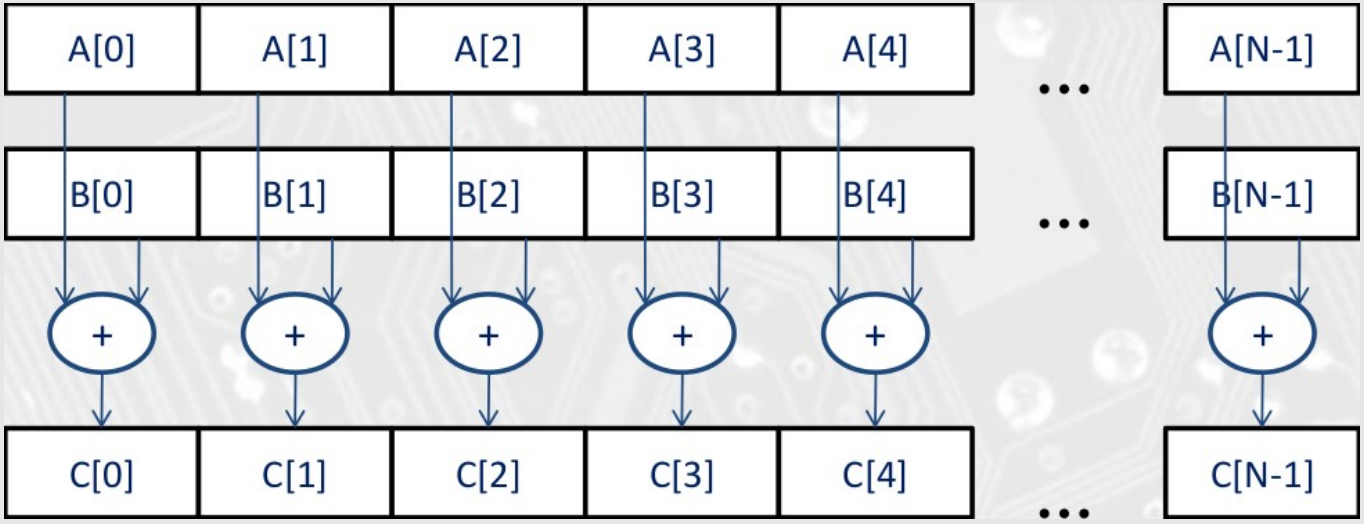
  - CUDA = MSIMD



UFV

Paralelismo de dados

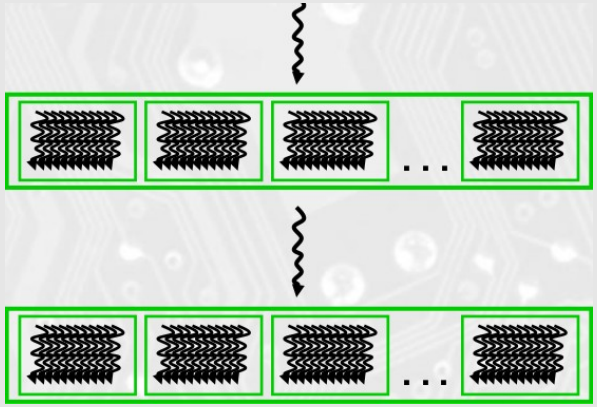
- Soma de vetores
  - Simples porque os dados são independentes



Paralelismo de dados

- Esquema de um programa utilizando CUDA
  - Código serial executa no host
  - Código paralelo (*kernel*) executa na GPU (device)

```
//Código serial (host)

//Código paralelo (device)
kernelA<<<numBlocos,numThreads>>>(args);

//Código serial (host)

//Código paralelo (device)
kernelB<<<numBlocos,numThreads>>>(args);
```



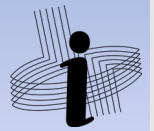
  - Chamada do kernel é assíncrona
    - kernelB só é executado após o fim do kernelA
    - Host pode utilizar o cudaDeviceReset() para esperar a conclusão do kernel

Paralelismo de dados

- O conjunto de blocos que executam o *kernel* é chamado de grid
  - Todas as threads em um grid executam o mesmo código (SPMD)
  - Cada thread deve calcular seu índice

```
tid=blockIdx.x * blockDim.x + threadIdx.x;
```

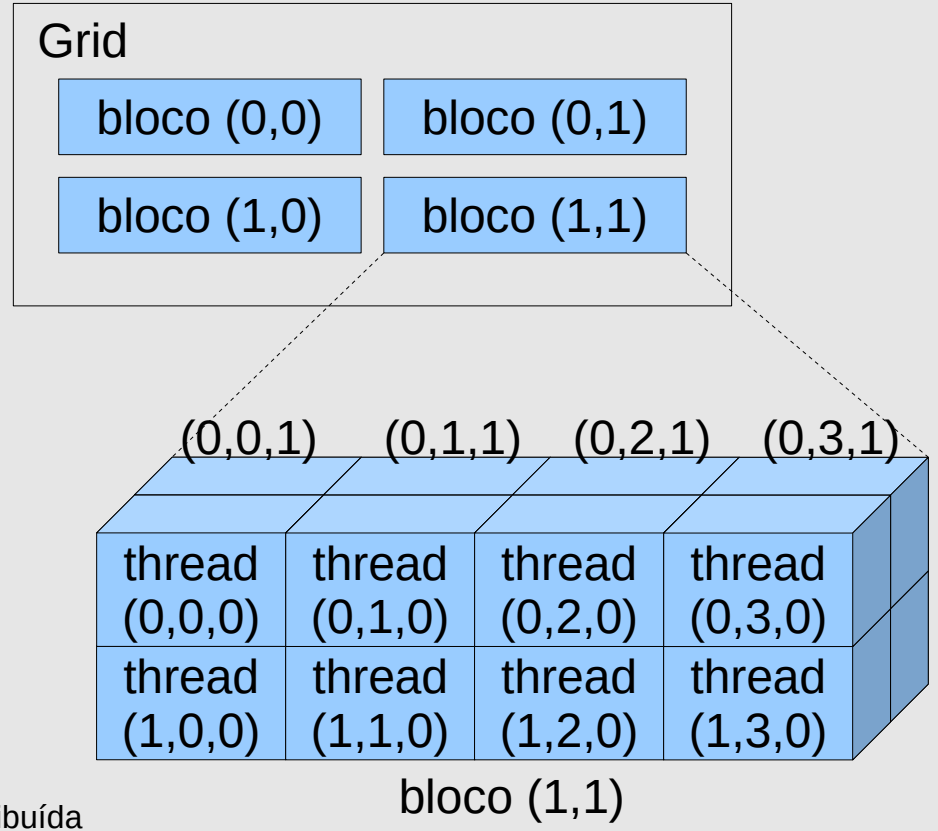
- Threads dentro de um mesmo bloco cooperam através de
  - Memória compartilhada (*shared memory*)
  - Operações atômicas
  - Sincronização de barreira



UFV

Paralelismo de dados

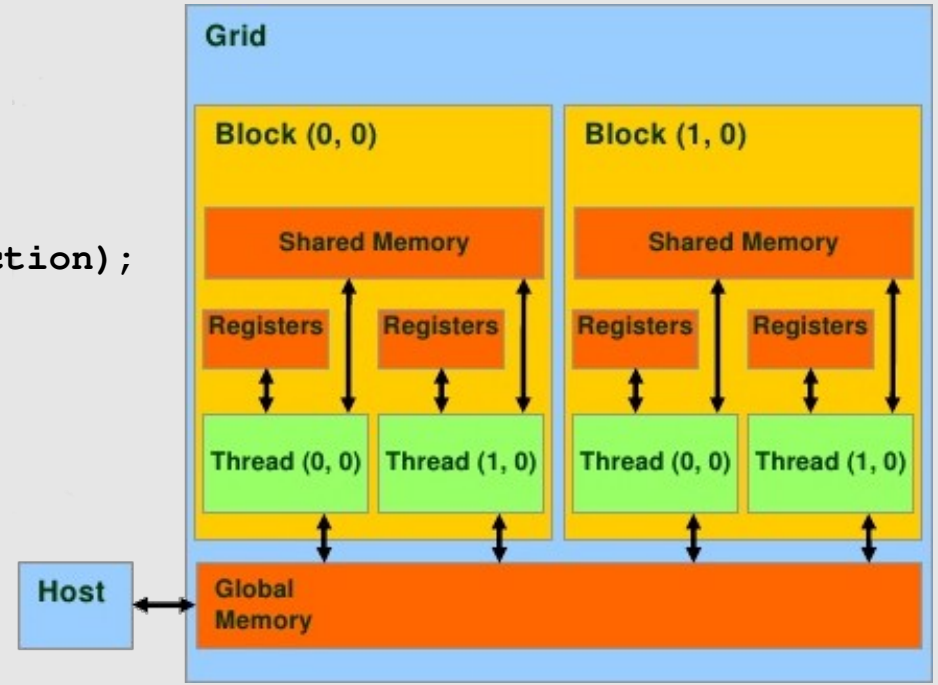
- Grids de até 3 dimensões podem ser utilizados
  - Simplifica processamento de dados multidimensionais



Modelo de memória (visão parcial)

- Código no dispositivo pode
  - Ler e escrever registradores
  - Ler e escrever memória global
- Código no host pode
  - Transferir data para memória global

```
cudaMalloc((void**)&dev,size);
cudaMemcpy(dest,src,size,direction);
cudaFree(dev);
```



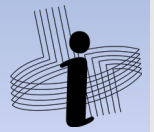
Checagem de erros

- Funções retornam código para indicar sucesso/erro

```
cudaError_t err=cudaMalloc((void**)&d_A, size);

if (err!=cudaSuccess) {
    printf("ERRO: %s\n",cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
```

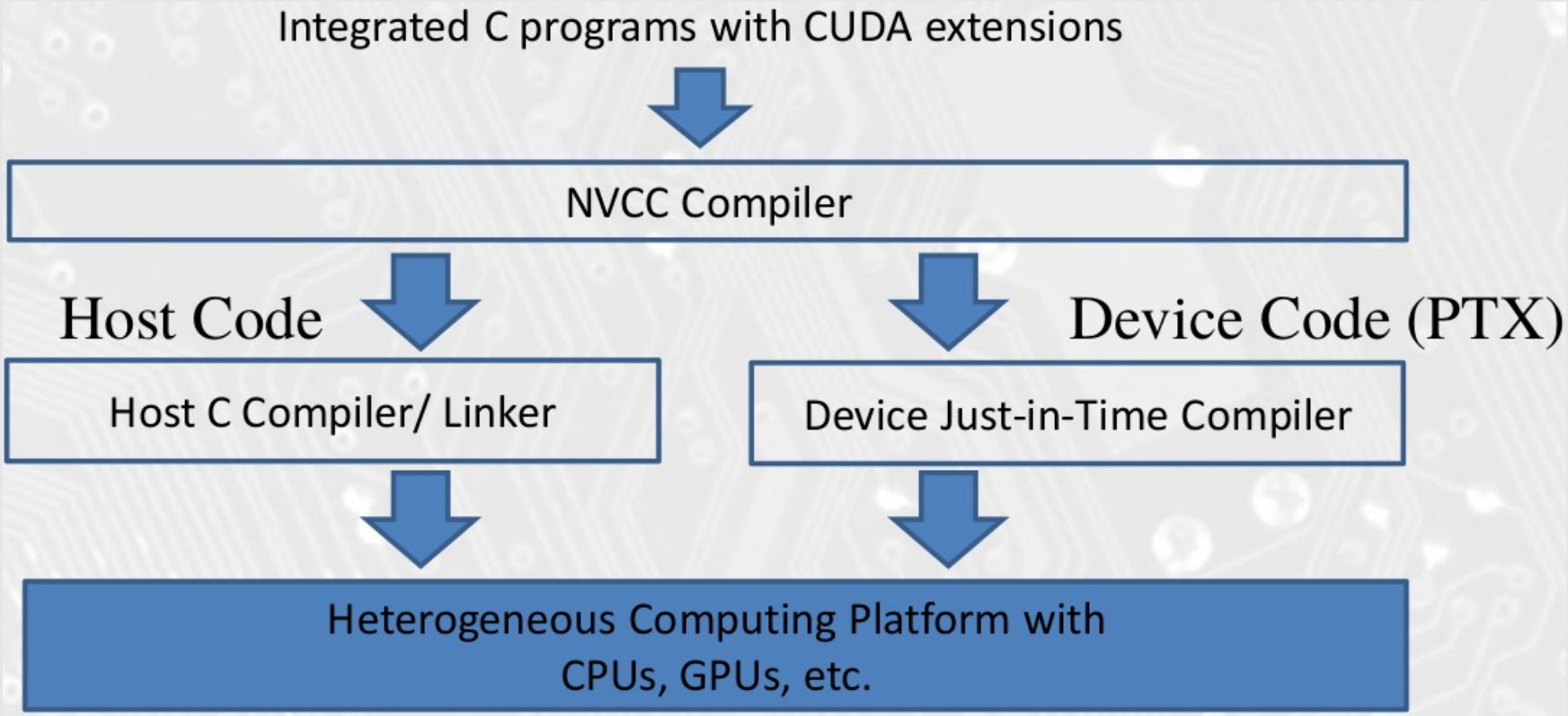
- Definindo uma função para checagem de erros:

```
#define CHECK(ans) { gpuAssert((ans), __FILE__, __LINE__); }

inline void gpuAssert(cudaError_t code, const char *file, int line) {
   if (code != cudaSuccess) {
      fprintf(stderr,"GPUassert: %s in file %s: line %d\n",
              cudaGetErrorString(code), file, line);
      exit(code);
   }
}
```



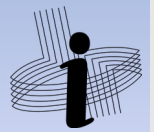
Compilando código CUDA

- Compilador nvcc



Obtendo informações da GPU

```
int dev = 0;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);

printf("Using Device %d:           %s\n", dev, deviceProp.name);
printf("Total global memory:     %u B\n",  deviceProp.totalGlobalMem);
printf("Shared memory per block: %u B\n",  deviceProp.sharedMemPerBlock);
printf("Threads per block:       %d\n", deviceProp.maxThreadsPerBlock);
printf("Maximum Grid Size:       %d,%d,%d\n",deviceProp.maxGridSize[0],
                                             deviceProp.maxGridSize[1],
                                             deviceProp.maxGridSize[2]);
printf("Maximum block size:      %d,%d,%d\n",deviceProp.maxThreadsDim[0],
                                             deviceProp.maxThreadsDim[1],
                                             deviceProp.maxThreadsDim[2]);
printf("Warp size:               %d\n", deviceProp.warpSize);
printf("Clock rate:              %d Khz\n", deviceProp.clockRate);
```



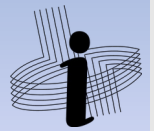
UFV

Marcando tempo de execução

- Por código

```
float elapsed_time;
cudaEvent_t start, stop;     // Declara dois eventos
cudaEventCreate(&start);     // Irá marcar o inicio da execução
cudaEventCreate(&stop);      // Irá marcar o final da execução

cudaEventRecord(start, 0);   // Adiciona na fila (início da marcação)

cudaEventRecord(stop, 0);    // Adiciona na fila (fim da marcação)
cudaEventSynchronize(stop); // Espera o evento terminar
cudaEventElapsedTime(&elapsed_time,start, stop); // Calcula o tempo
```

- nvprof
  - Ferramenta de *profiling* da Nvidia

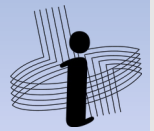


UFV

Adição de vetores

- Código executado no host

```
#include <cuda.h>

void vecAdd(float* h_A, float* h_B, float* h_C, int n) {
    int size = n*sizeof(float);
    float* d_A, d_B, d_C;

    // alocar memória na GPU para A, B e C
    // copiar A e B para a GPU
    // chamar o Kernel
    // copiar C do GPU para a memória principal
}
```

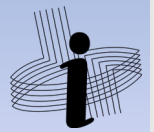


UFV

Adição de vetores

- Código executado no host

```
#include <cuda.h>

void vecAdd(float* h_A, float* h_B, float* h_C, int n) {
    int size = n*sizeof(float);
    float* d_A, d_B, d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    vecAddKernel<<<ceil(n/512.0),512>>>(d_A, d_B, d_C, n);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

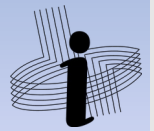


UFV

Adição de vetores

- Código executado na GPU

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n) {
    int i=blockIdx.x * blockDim.x + threadIdx.x;
    if (i<n)
        C[i]=A[i]+B[i];
}
```



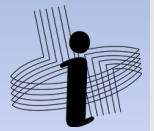
UFV

Declaração de funções

- Diretivas

Diretiva	Onde é executada?	Quem pode chamar?
`__global__`	GPU	host
`__device__`	GPU	GPU
`__host__`	host	host

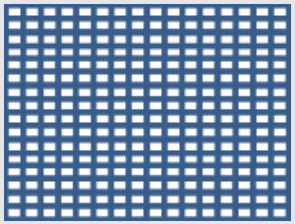
  - A combinação `__device__ __host__` é permitida
  - Uma função `__global__` deve ser sempre do tipo void



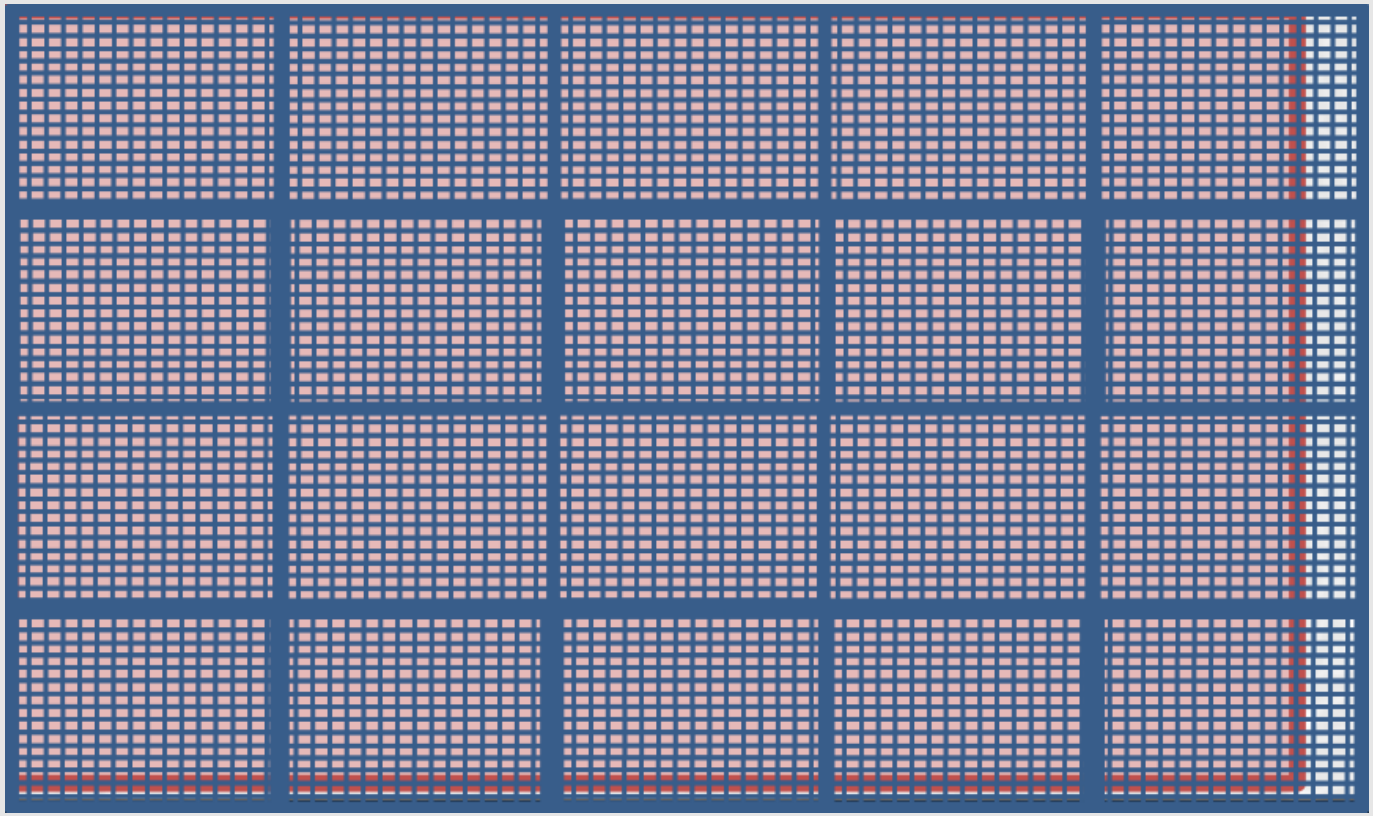
UFV

Grid multidimensional

- Processamento de imagens

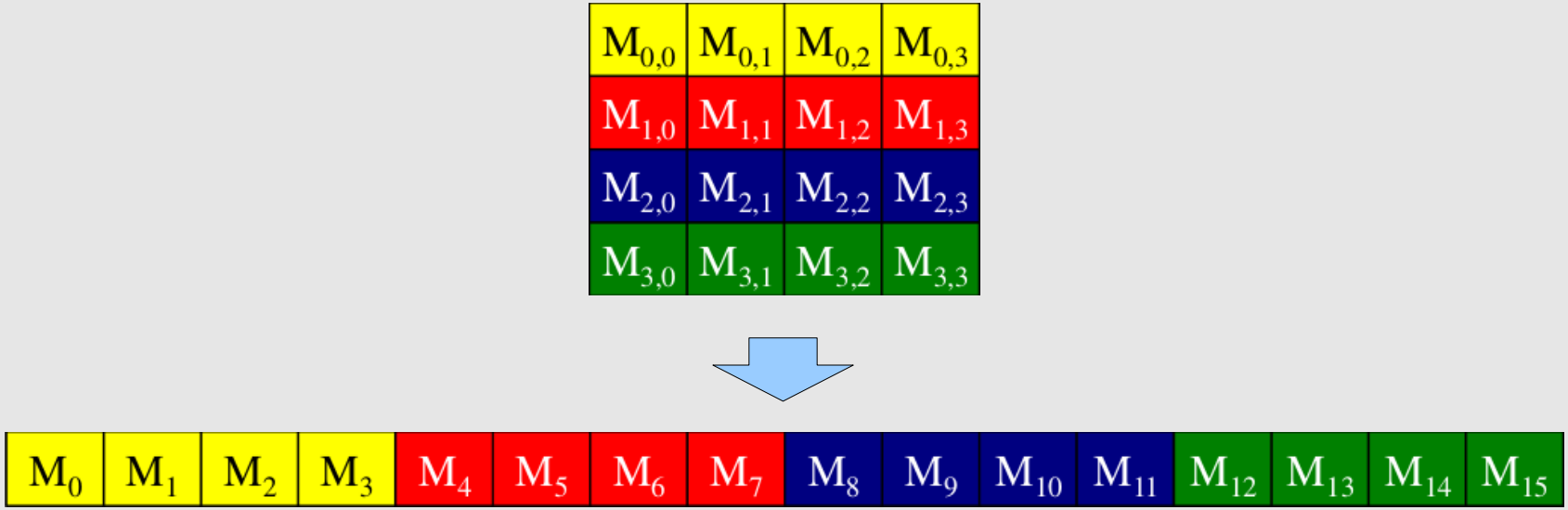


Bloco de 16x16



Grid multidimensional

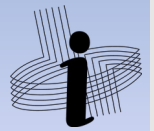
- Estrutura de uma matriz em C/C++



Grid multidimensional

- Exemplo de um *kernel* multidimensional

```
__global__
void PictureKernel(float* d_Pin, float* d_Pout, int m, int n) {
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;

    if ((row < m) && (col < n))
        d_Pout[row*n+col] = 2*d_Pin[row*n+col];
}
```



UFV

Grid multidimensional

- Usando o tipo dim3
  - A chamada do kernel

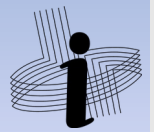
```
vecAddKernel<<<ceil(n/512.0),512>>>(d_A, d_B, d_C, n);
```

  - é equivalente a

```
dim3 grid (ceil(n/512.0),1,1);
dim3 block(512,1,1);
vecAddKernel<<<grid,block>>>(d_A, d_B, d_C, n);
```

- Pode-se criar grids e blocos de 1D até 3D
  - Exemplo em duas dimensões para uma matriz L×C com blocos 32×32

```
dim3 grid (ceil(C/32.0),ceil(L/32.0),1);
dim3 block(32,32,1);
```



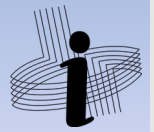
UFV

Grid multidimensional

- Exemplos

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

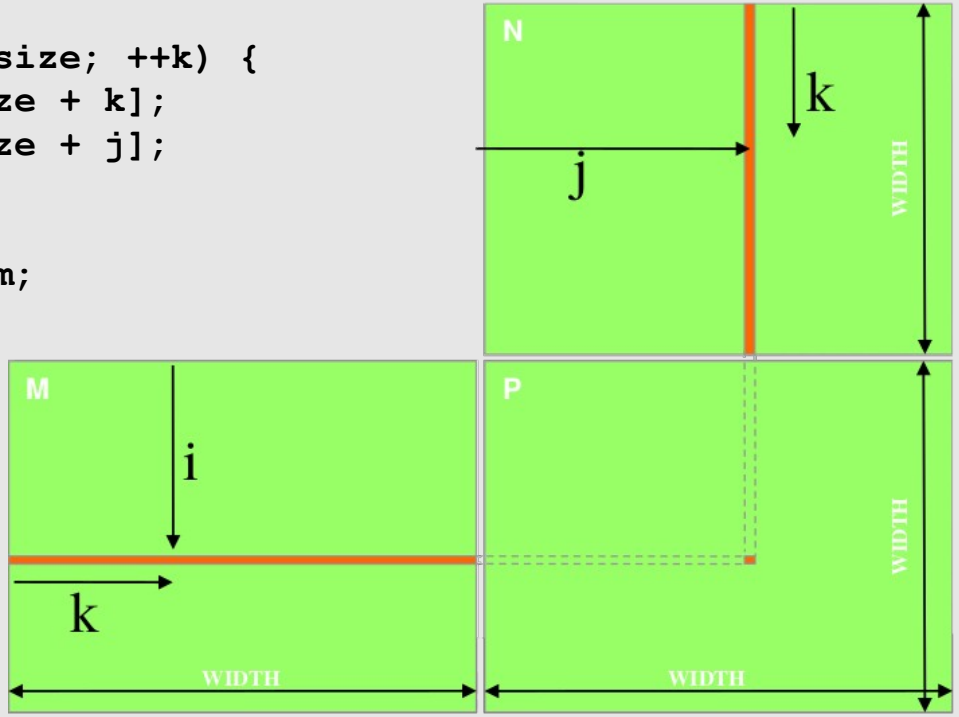
$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$



Multiplicação de matrizes

- Multiplicação executada completamente no host

```
void matrixMultOnHost(int *M, int *N, int *P, int size) {
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j) {
            int sum = 0;
            for (int k = 0; k < size; ++k) {
                int a = M[i * size + k];
                int b = N[k * size + j];
                sum += a * b;
            }
            P[i * size + j] = sum;
        }
}
```



Multiplicação de matrizes

- Calculando linha e coluna de cada thread
  - Bloco 0,0

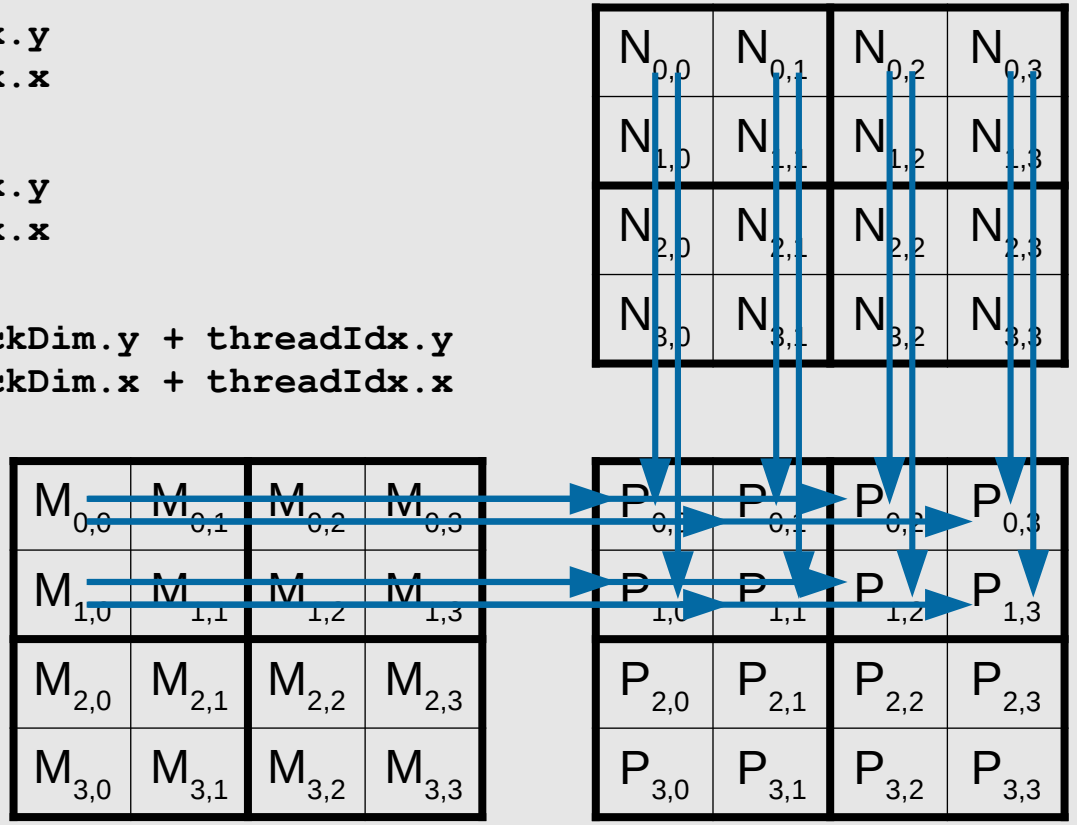
```
linha  = 0 * 2 + threadIdx.y
coluna = 0 * 2 + threadIdx.x
```

  - Bloco 0,1

```
linha  = 0 * 2 + threadIdx.y
coluna = 1 * 2 + threadIdx.x
```

  - Formula geral

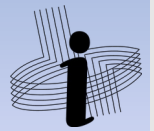
```
linha  = blockIdx.y * blockDim.y + threadIdx.y
coluna = blockIdx.x * blockDim.x + threadIdx.x
```



Multiplicação de matrizes

- Multiplicação executada na GPU

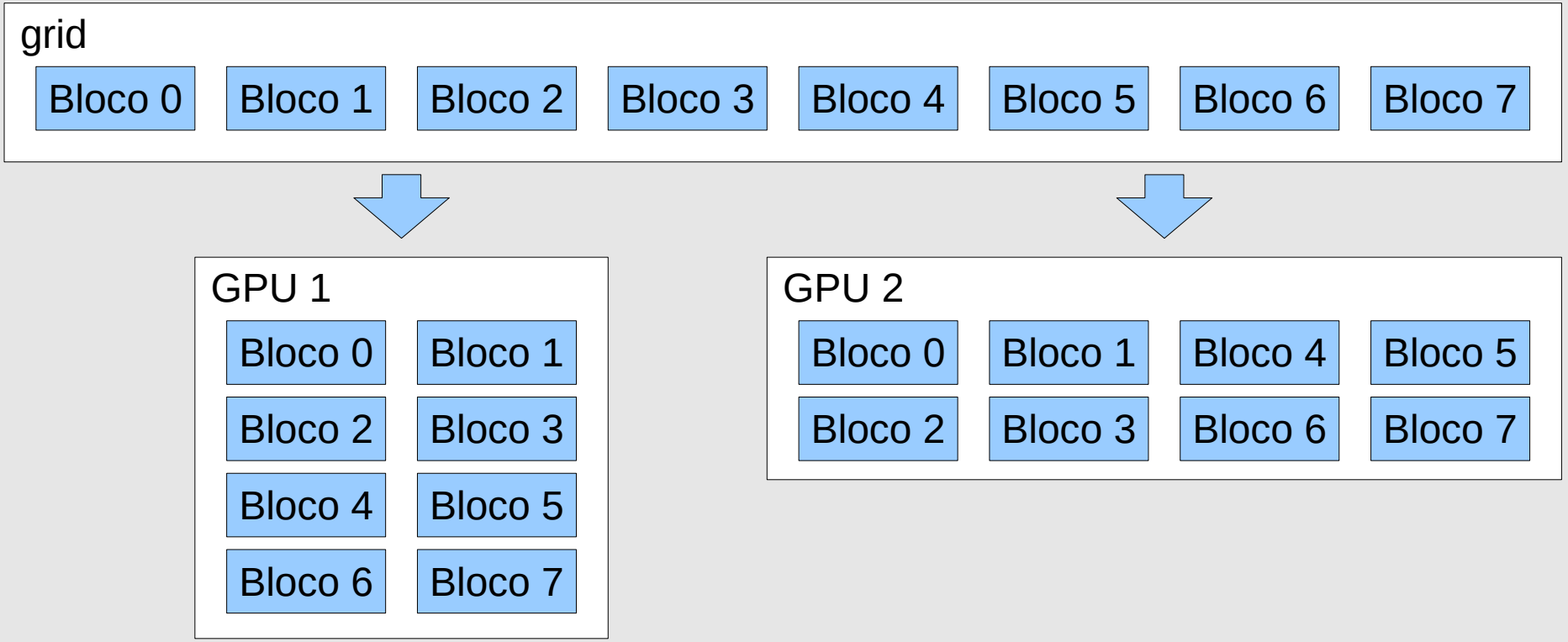
```
__global__
void matrixMultKernel(int *M, int *N, int *P, int size) {

    int row=blockIdx.y*blockDim.y+threadIdx.y;
    int col=blockIdx.x*blockDim.x+threadIdx.x;

    if (row<size && col<size) {
        int v=0;
        for(int k=0; k<size; ++k)
            v+=M[row*size+k]*N[k*size+col];
        P[row*size+col]=v;
    }
}
```



UFV

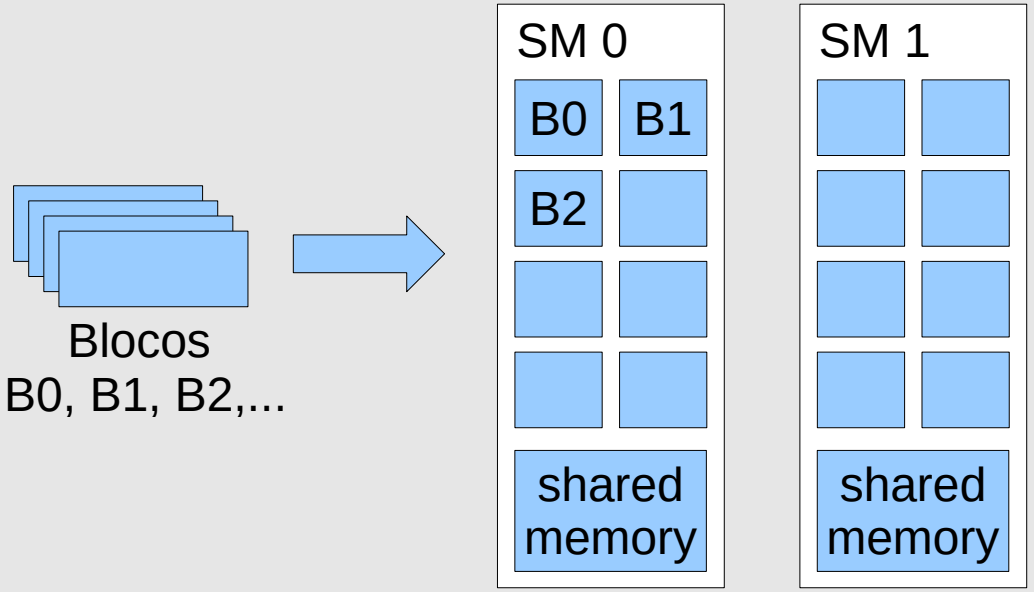
Escalabilidade e escalonamento

- Atribuição de blocos ao processador é definida pelo hardware
- O kernel deve ser escalável para qualquer número de processadores



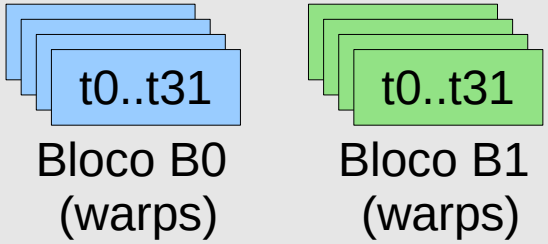
Escalabilidade e escalonamento

- *Streaming Multiprocessors* (SM)
  - Possui um conjunto de processadores e memória compartilhada
  - 1 bloco não pode ser dividido em mais de 1 SM
  - Limitado a um número máximo de blocos e de threads
    - Ex: Fermi SM
      - 8 blocos por SM
      - 1536 threads por SM
        - » 256 * 6
        - » 512 * 3



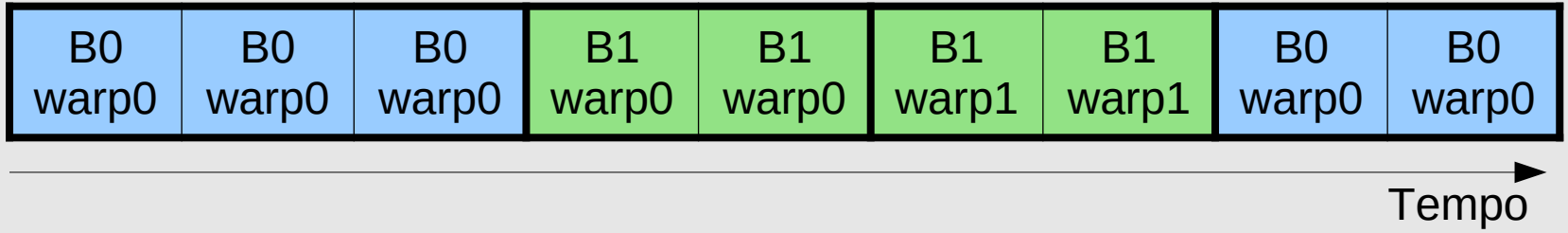
Escalabilidade e escalonamento

- *Warps*
  - Unidade de escalonamento do SM (corresponde a 32 threads)
  - Threads dentro de um bloco são divididas em *warps*
  - Arquitetura SIMD
  - Troca de warps em uma SM não tem overhead
  - Ordem de execução **não** é garantida



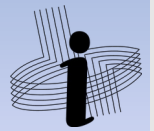
Quantos warps estarão em um SM executando 3 blocos de 256 threads?

Resposta: 24 warps



Escalabilidade e escalonamento

- Pergunta
  - Considerando as características do Fermi SM (8 blocos e 1536 threads por SM), qual o tamanho ideal para o bloco em um kernel de multiplicação de matriz?
    - 8x8 ?
    - 16x16 ?
    - 32x32 ?



UFV

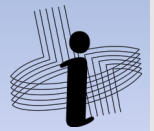
Divergência em instruções de controle

- Blocos são divididos em *warps* de acordo com o índice das threads
- Divergências podem ocorrer quando uma arquitetura SIMD executa uma instrução de controle de fluxo

```
if (threadIdx.x < 2)
    ...
else
    ...
```

  - Exemplo sem divergência

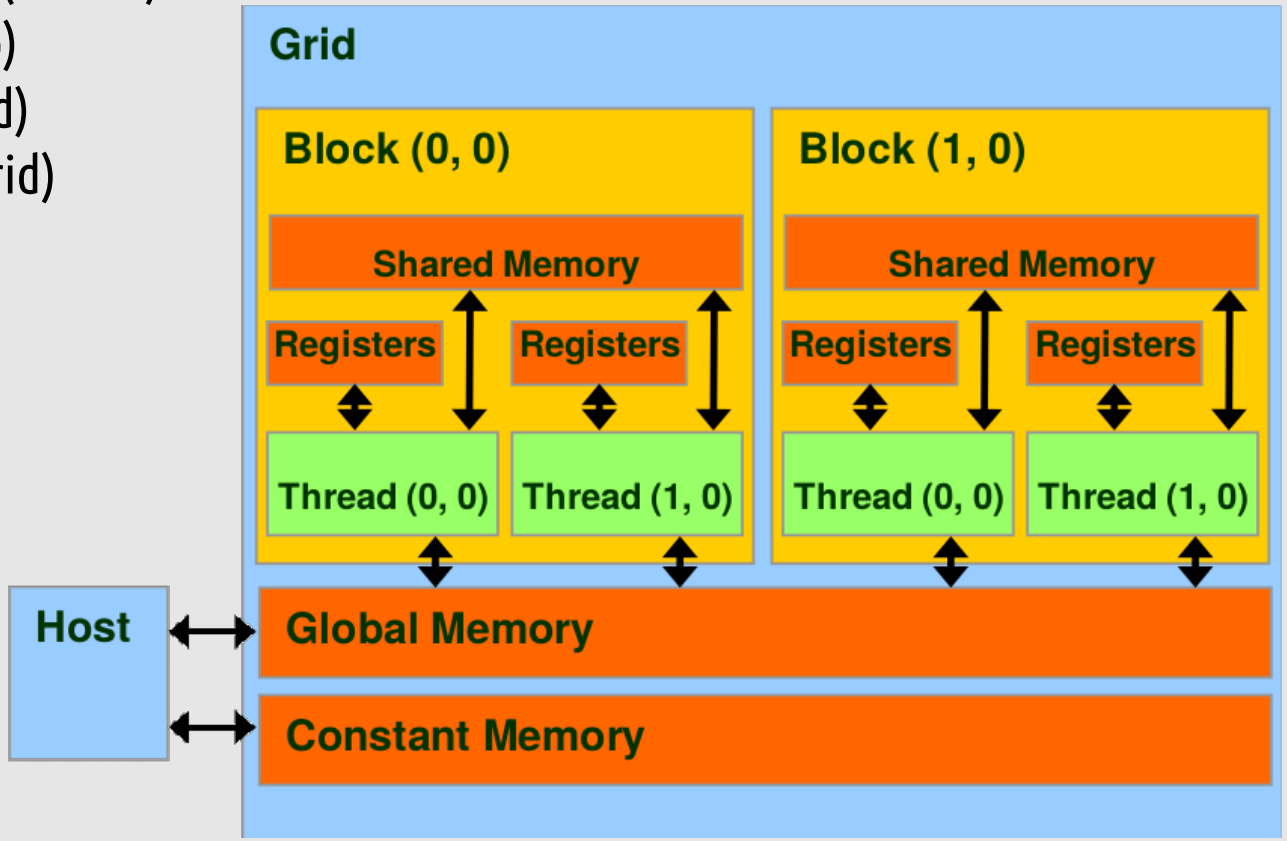
```
if (threadIdx.x / WARP_SIZE > 2)
    ...
else
    ...
```



UFV

Modelo de memória completo

- Custo de acesso e escopo
  - Registradores: ~1 ciclo (thread)
  - Shared: ~5 ciclos (bloco)
  - Global: ~500 ciclos (grid)
  - Constante: ~5 ciclos (grid)

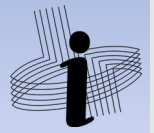


Modelo de memória completo

- Alocando memória

Diretiva		Memória	Escopo	Duração
	`int localVar;`	registrador	thread	thread
`__device__ __shared__`	`int sharedVar;`	shared	bloco	bloco
`__device__`	`int globalVar;`	global	grid	aplicação
`__device__ __constant__`	`int constantVar;`	constante	grid	aplicação

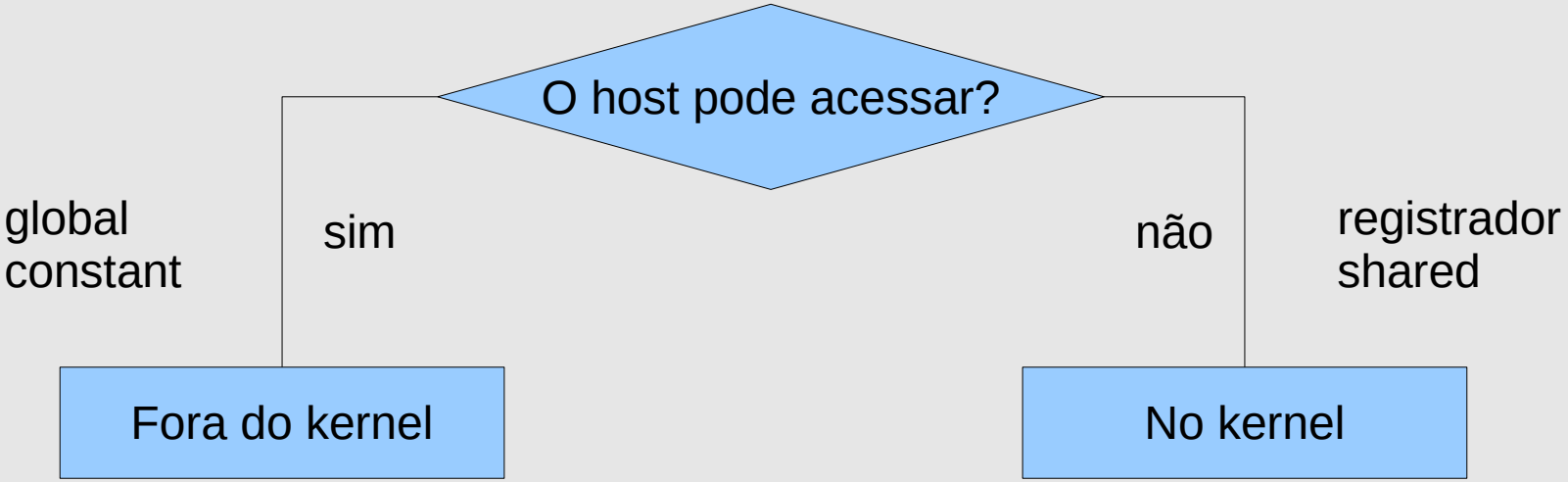
  - A diretiva `__device__` é opcional quando usada com `__shared__` ou `__constant__`
  - Variáveis automáticas são alocadas nos registradores, com exceção dos arrays criados por threads, que são alocados na memória global



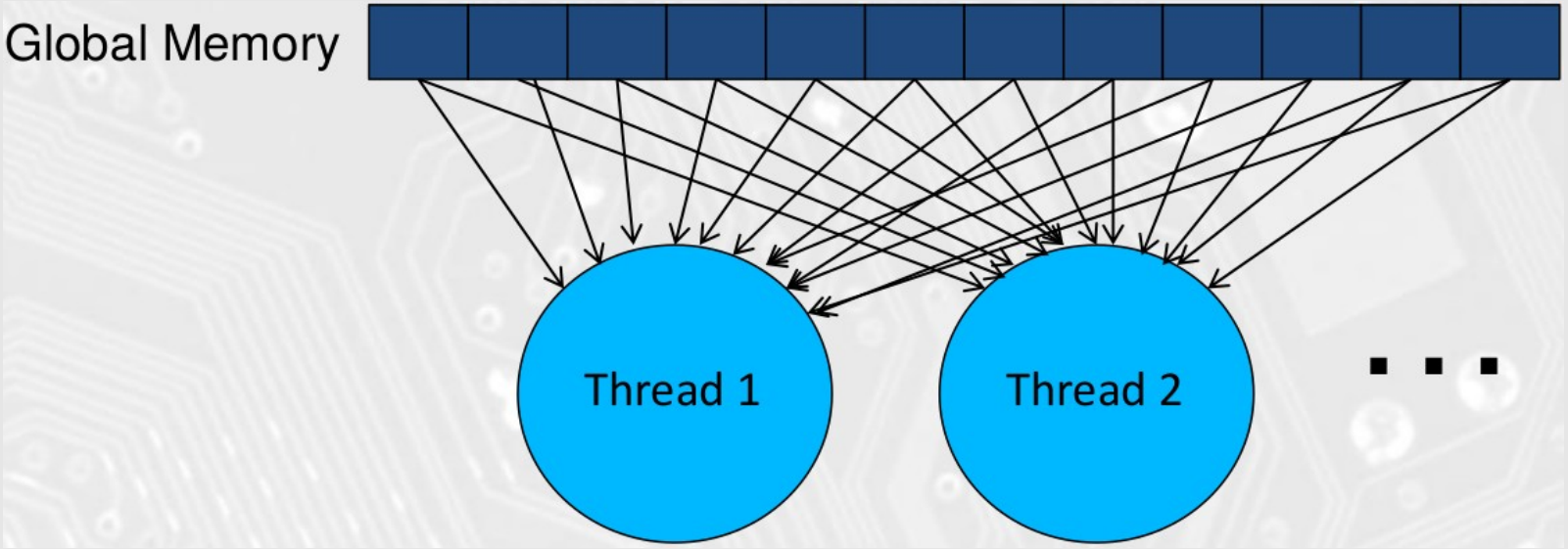
UFV

Modelo de memória completo

- Onde declarar as variáveis?

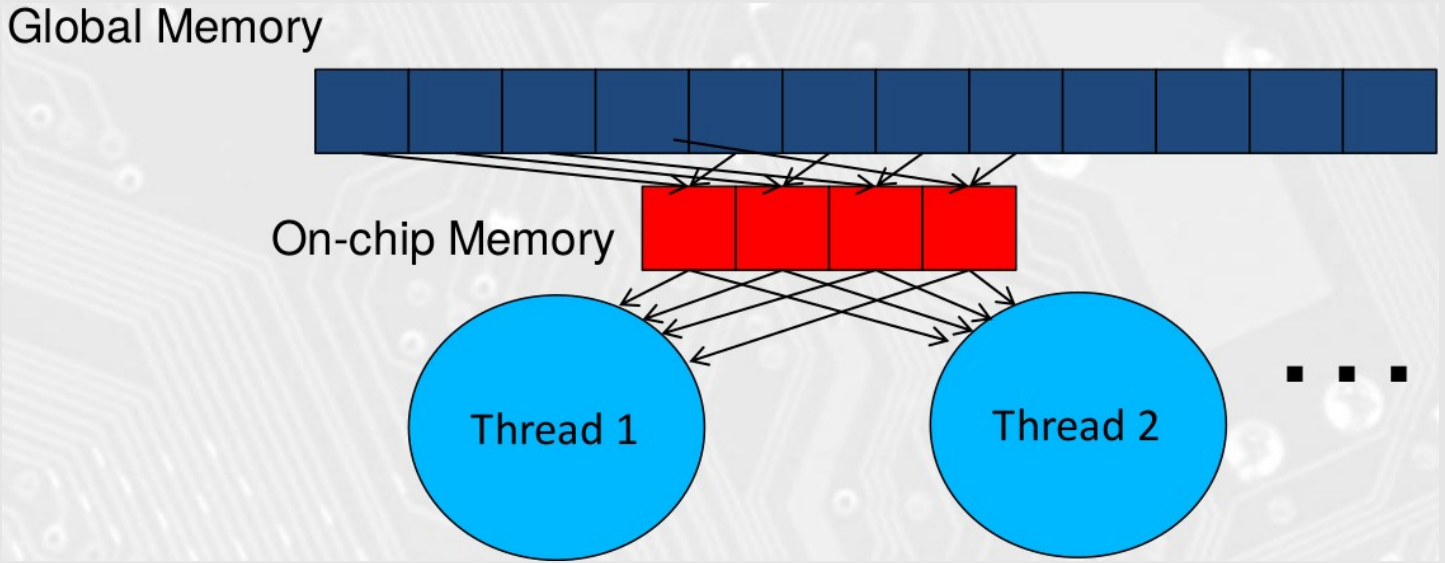


Usando memória compartilhada



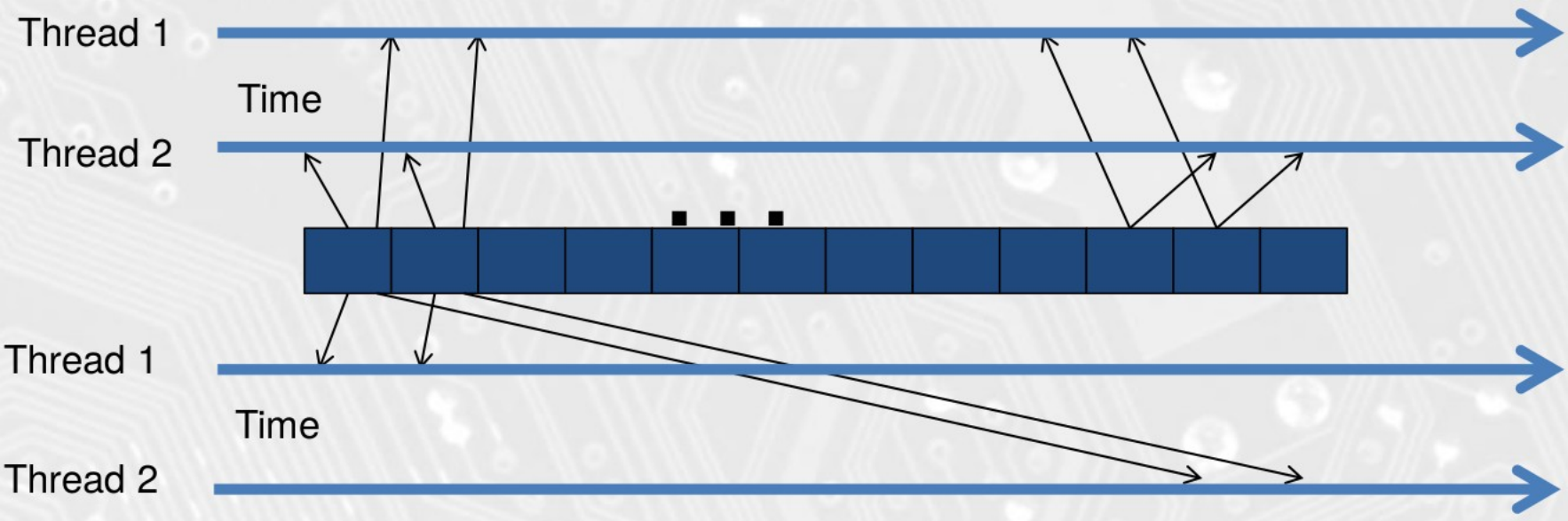
Usando memória compartilhada

- A memória compartilhada é como um rascunho utilizado pelas threads



Usando memória compartilhada

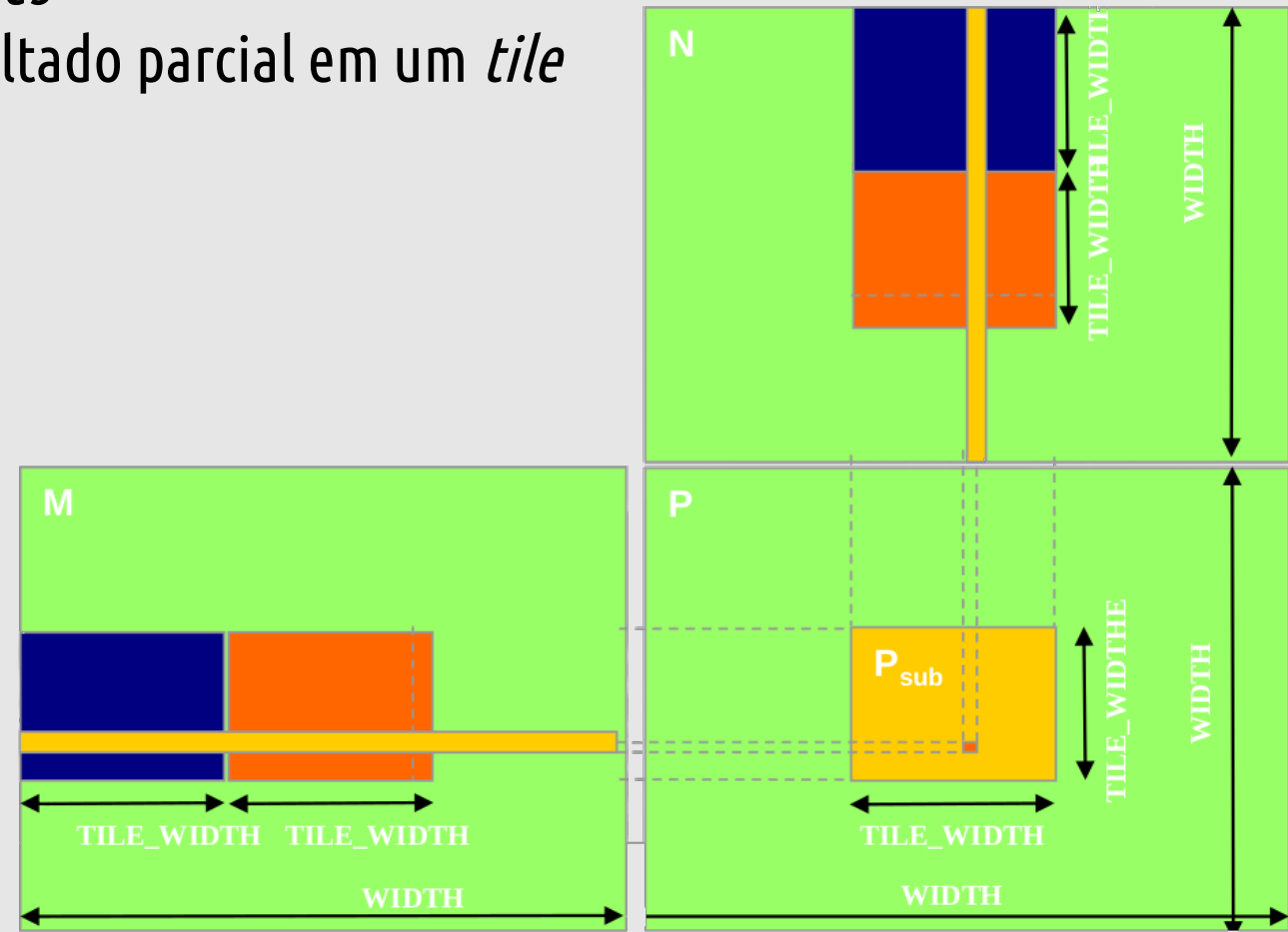
- Cenário bom



- Cenário ruim

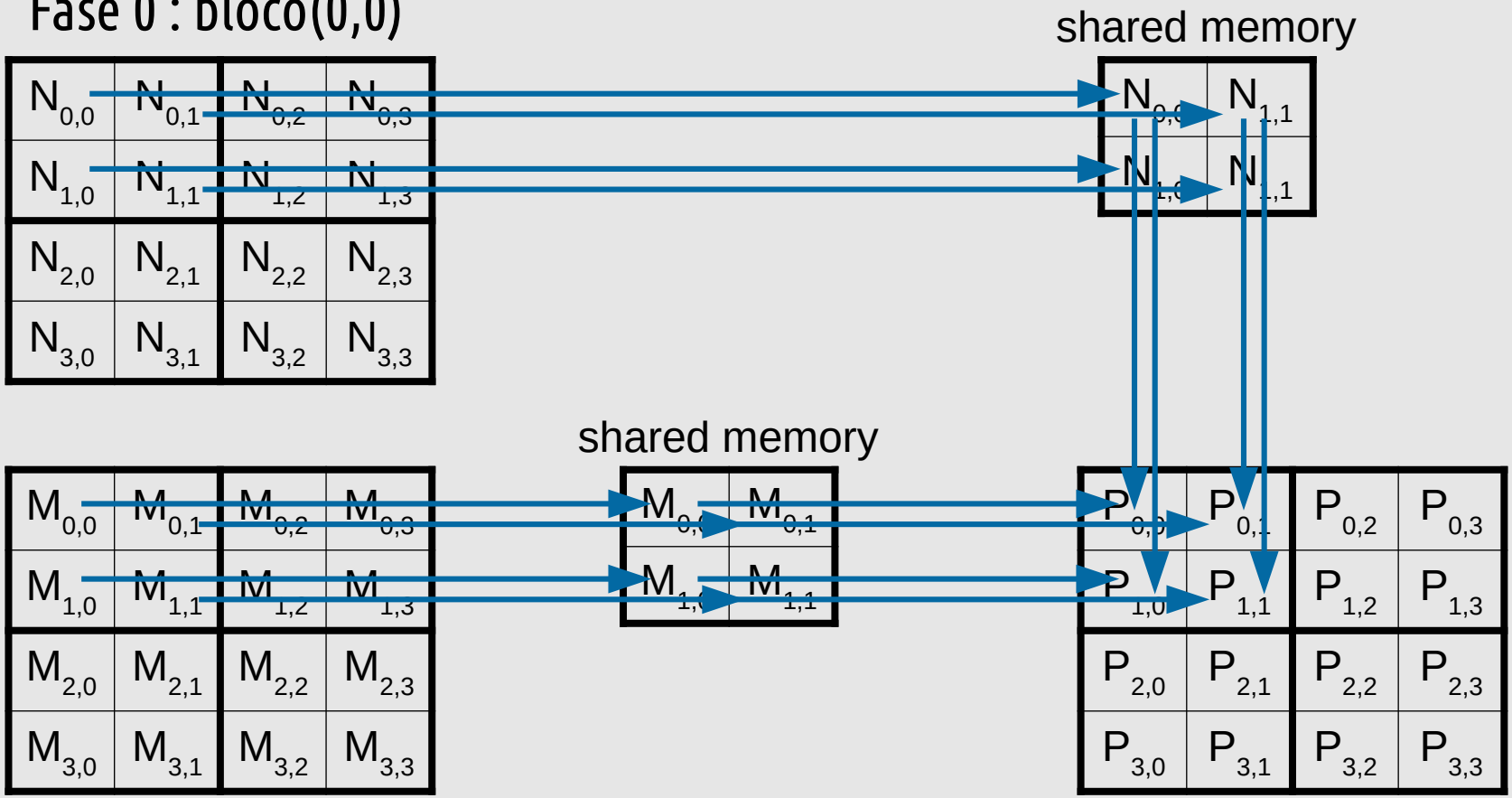
Multiplicação de matrizes usando memória compartilhada

- Divisão da matriz em *tiles*
- Cada fase calcula o resultado parcial em um *tile*



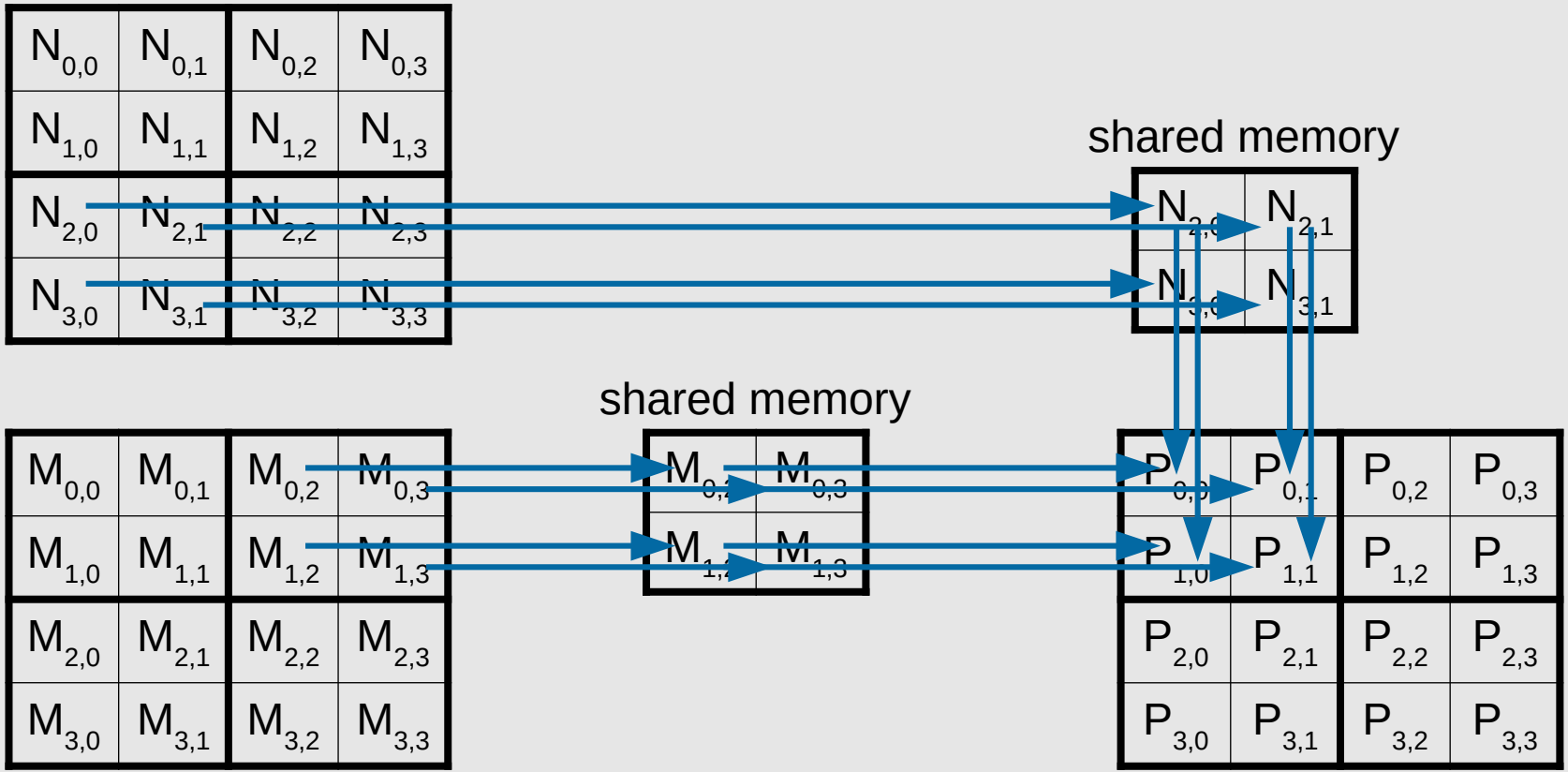
Multiplicação de matrizes usando memória compartilhada

- Fase 0 : bloco(0,0)



Multiplicação de matrizes usando memória compartilhada

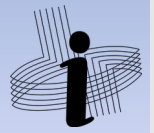
- Fase 1 : bloco(0,0)



Multiplicação de matrizes usando memória compartilhada

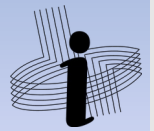
- Código do kernel

```
void multMatriz(int *m, int *n, int *p, int size) {
    __shared__ int sm[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ int sn[BLOCK_SIZE][BLOCK_SIZE];

    int tx=threadIdx.x;
    int ty=threadIdx.y;
    int row=blockIdx.y*blockDim.y + ty;
    int col=blockIdx.x*blockDim.x + tx;
    int valor=0;

    for (int fase=0; fase<size/BLOCK_SIZE; ++fase) {
        sm[ty][tx]=m[row*size+tx+fase*BLOCK_SIZE];
        sn[ty][tx]=n[(ty+fase*BLOCK_SIZE)*size+col];
        __syncthreads();       //barreira para todas as threads do bloco
        for (int k=0; k<BLOCK_SIZE; ++k)
            valor+=sm[ty][k]*sn[k][tx];
        __syncthreads();
    }
    p[row*size+col]=valor;
}
```



Multiplicação de matrizes usando memória compartilhada

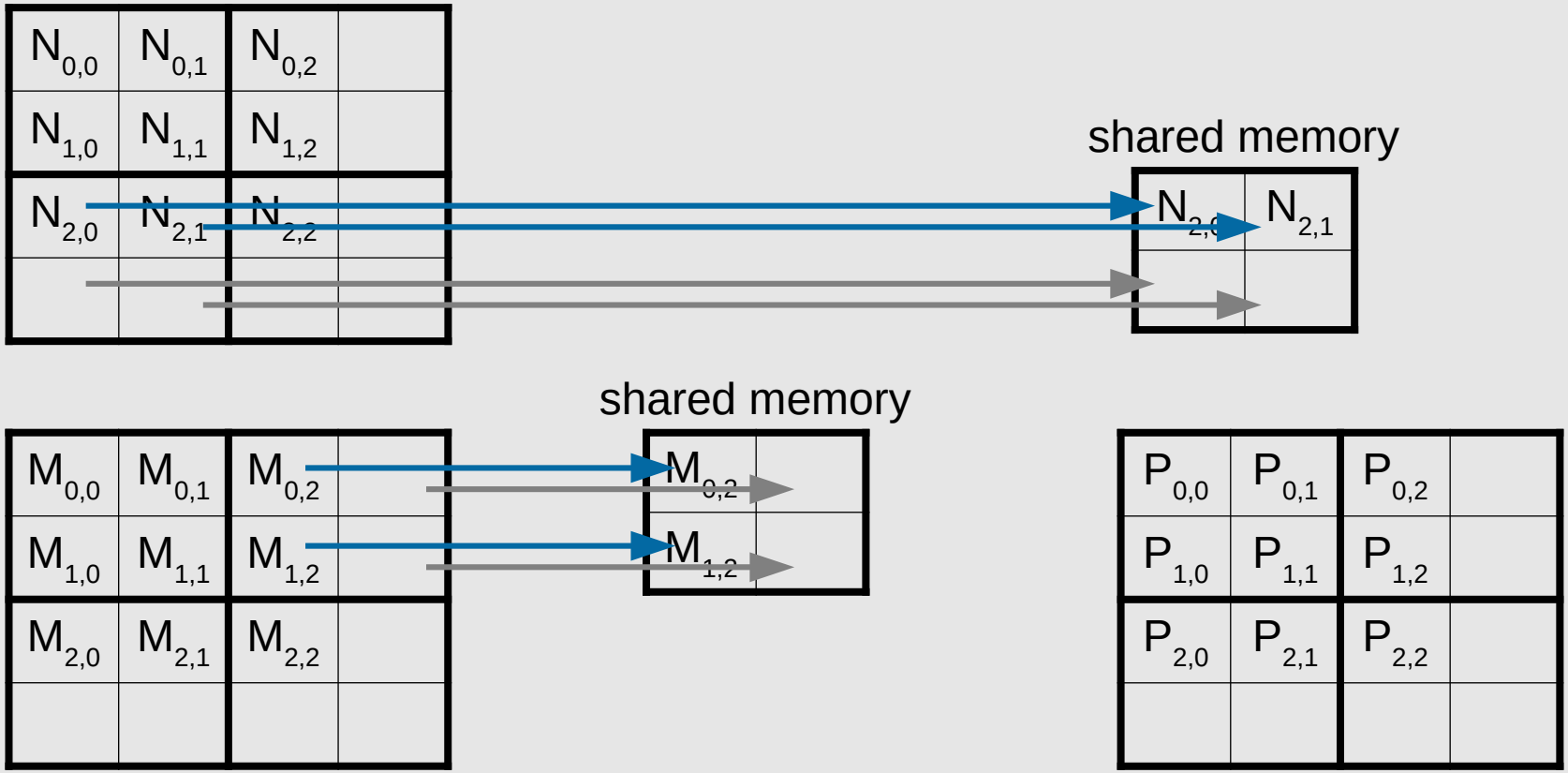
- Análise do tamanho dos blocos
  - 16x16 = 256 threads
    - 2*256 = 512 leituras na memória global
    - 256 * (2*16) = 8192 operações (multiplicação e adição)
  - 32x32 = 1024 threads
    - 2*1024 = 2048 leituras na memória global
    - 1024 * (2*32) = 65536 operações (multiplicação e adição)



UFV

Multiplicação de matrizes com tamanhos arbitrários

- Fase 1 : bloco(0,0)



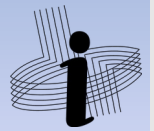
Multiplicação de matrizes com tamanhos arbitrários

- Versão atualizada do kernel

```
__global__
void multMatriz(int *M, int *N, int *P, int size) {
    ...
    for (int fase=0; fase < ceil(1.0*size/BLOCK_SIZE) ; ++fase) {
        if (row<size && (fase*BLOCK_SIZE+tx)<size)
            sM[ty][tx]=M[row*size + fase*BLOCK_SIZE + tx];
        else
            sM[ty][tx]=0;
        if ((fase*BLOCK_SIZE+ty)<size && col<size)
            sN[ty][tx]=N[(fase*BLOCK_SIZE + ty)*size + col];
        else
            sN[ty][tx]=0;
        __syncthreads();
        for (int k=0; k<BLOCK_SIZE ; k++)
            v+=sM[ty][k]*sN[k][tx];
        __syncthreads();
    }
    if (row<size && col<size)
        P[row*size+col]=v;
}
```

Multiplicação de matrizes com tamanhos arbitrários

- Considerações
  - Para cada thread as condições são diferentes para
    - carregar dados de M
    - carregar dados de N
    - calcular/gravar dados de P
  - Efeito da divergência nos blocos *if..else* (considerando blocos de tamanho 32 x 32)
    - Não há divergência nos blocos da última linha
    - Há divergência nos blocos da última coluna



UFV

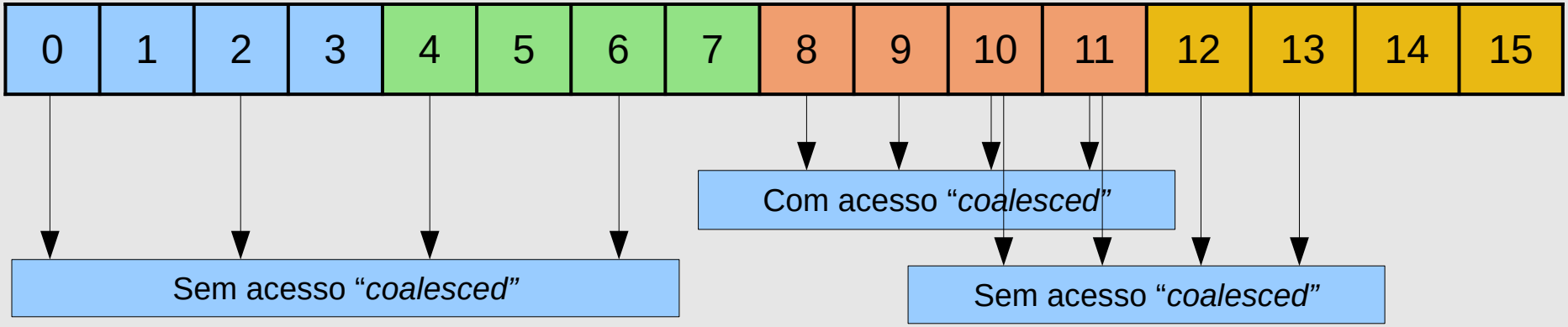
Acesso "*Coalesced*"

- Acesso à memória DRAM é mais lento que a velocidade da interface
- Leitura por rajadas (*burst*)
  - Tempo de acesso
    - Sem rajadas

    - Com rajadas

Acesso "*Coalesced*"

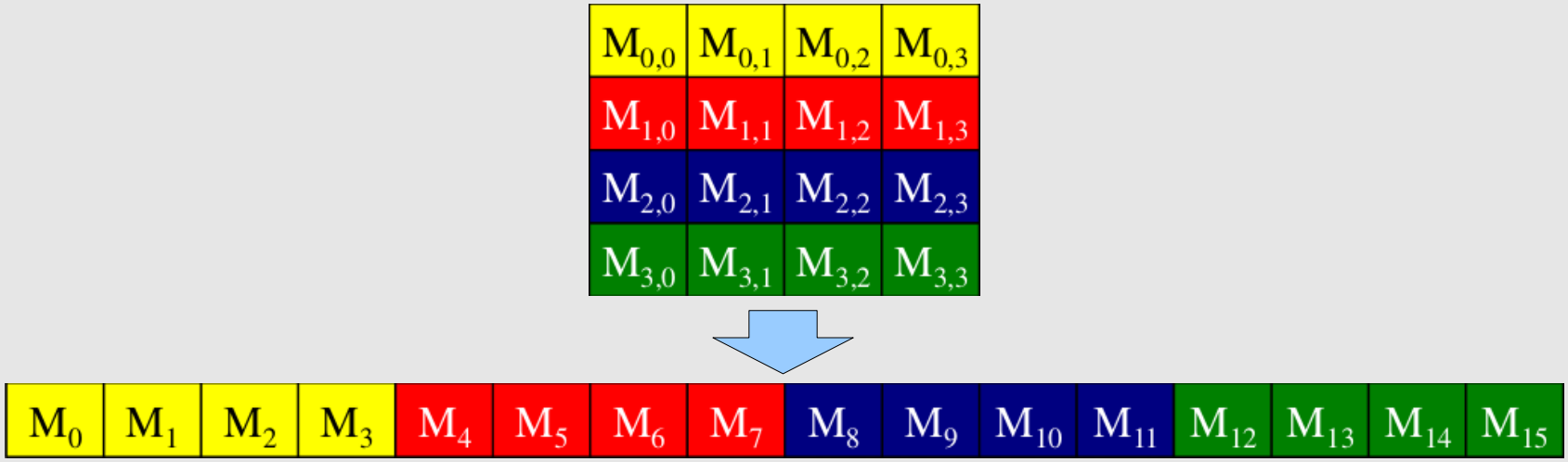
- Espaço de endereçamento dividido em seções do tamanho de uma rajada
- A seção inteira é transferida ao acessar uma localização qualquer da mesma
- *Coalescing* : Agrupar processamento de dados de uma mesma seção
  - Espaço de endereçamento de 16 bytes com rajadas de 4 bytes



  - Na prática, espaços de endereçamento têm pelo menos 4GB com rajadas de 128 bytes

Acesso "*Coalesced*"

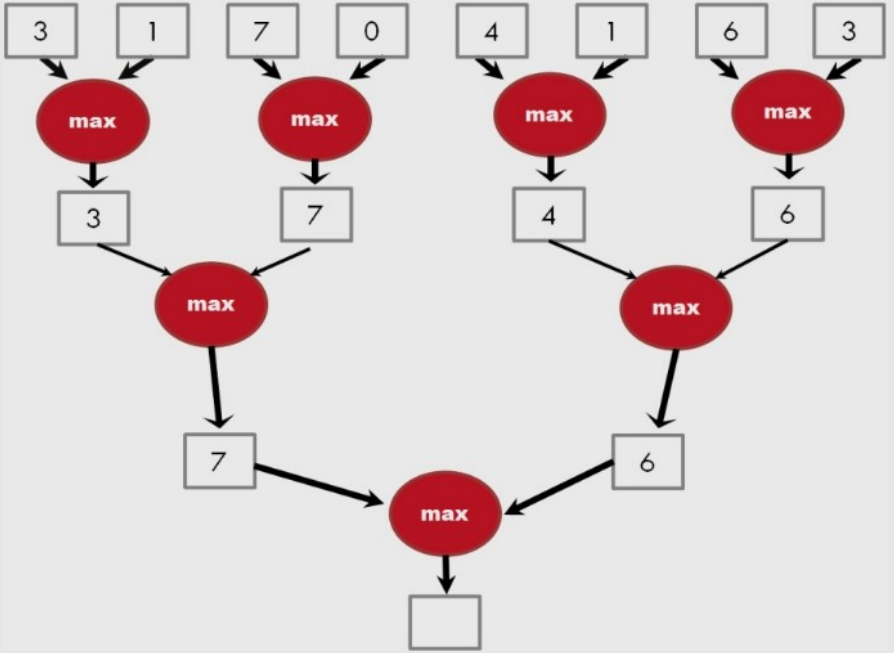
- Acesso *coalesced* na multiplicação de matrizes MxN (usando memória global)
  - Acesso agrupado na matriz N
  - Acesso não agrupado na matriz M (desperdício de largura de banda)



- Uso de memória compartilhada resolve o problema
  - Carregamento dos *tiles* de M e N são *coalesced*
  - Acessos à memória compartilhada SRAM são rápidos e não é feito *burst*

Redução

- Técnica utilizada para processar grandes quantidades de dados
- Resume um conjunto de dados em um valor através de uma operação
  - Soma, produto, mínimo, máximo, etc.
  - Operações definidas pelo usuário, desde que
    - seja associativa
    - seja comutativa
    - tenha um valor identidade bem definido
- Algoritmo sequencial
  - realiza N-1 operações de redução
- Algoritmo paralelo
  - realiza N-1 operações em log(N) passos

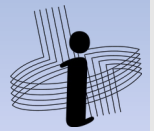


Redução

- Análise

$$\frac{1}{2}N+\frac{1}{4}N+\frac{1}{8}N+\ldots+\frac{1}{N}N=\left(1-\frac{1}{N}\right)N=N-1 \text{ operações}$$

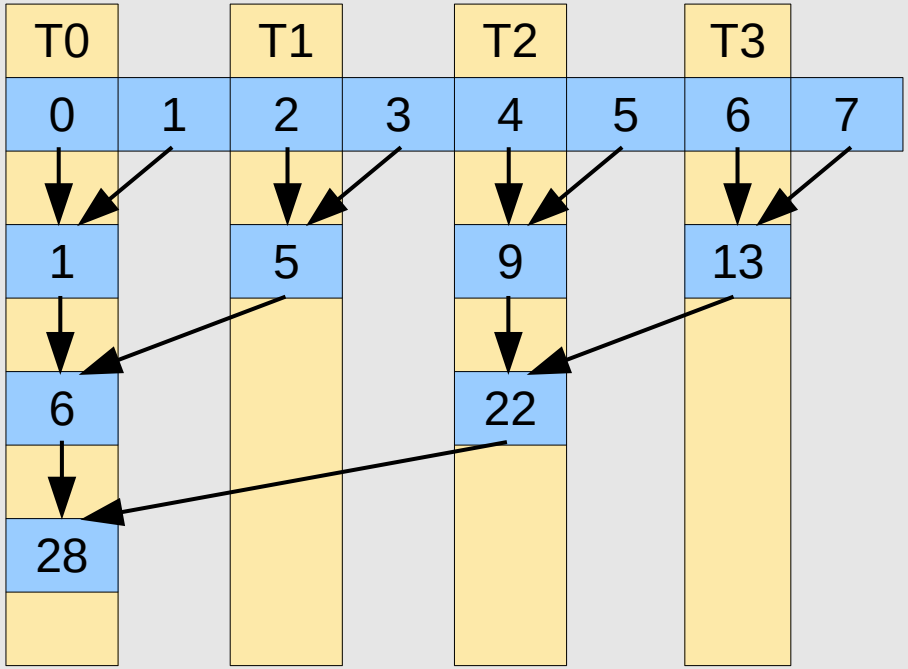
  - N = 1.000.000
  - Redução leva log(N) = 20 passos
  - Paralelismo médio = 50.000 por passo
- No entanto, pico de recursos exigidos é de 500.000



UFV

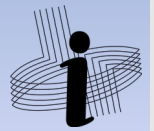
Redução de soma

- Vetor original armazenado na memória global
- Memória compartilhada armazena um vetor de somas parciais
- Resultado final é armazenado na posição 0



Redução de soma

- Voltando para a memória global
  - Thread 0 escreve resultado em um array na memória global
  - Escrita indexada através do índice do bloco
- O resultado do kernel é um novo array menor
  - Processado na CPU
  - Ou utiliza o kernel novamente

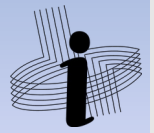


Redução de soma

- Código do kernel (versão 1)

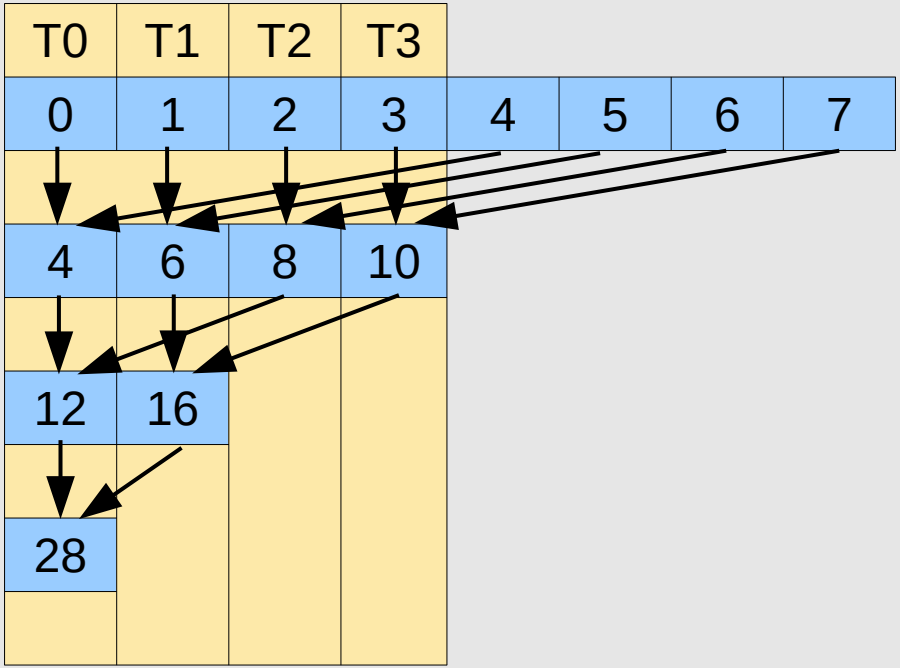
```
__global__
void reductionKernel1(int *a, int *r) {
    __shared__ int s[2*BLOCK_SIZE];
    int tx=threadIdx.x;
    int tid=2*blockIdx.x*blockDim.x+tx;

    s[tx]=a[tid];
    s[tx+BLOCK_SIZE]=a[tid+BLOCK_SIZE];

    for (int dist = 1; dist<BLOCK_SIZE; dist*=2) {
        __syncthreads();
        if (tx%dist==0)
            s[2*tx]+=s[2*tx+dist];
    }

    if (threadIdx.x == 0);
        r[blockIdx.x] = s[0];
}
```

Uso ineficiente dos recursos.
Várias warps terão poucas threads ativas em determinados momentos.



Redução de soma

- Reduzindo divergência de controle nas *warps*
  - Para um bloco de 1024 threads: nenhuma divergência nos primeiros 5 passos
    - 1024, 512, 256, 128, 64, 32

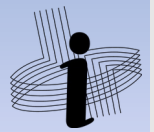


Redução de soma

- Código do kernel (versão 2)

```
__global__
void reductionKernel2(int *a, int *r) {
    __shared__ int s[2*BLOCK_SIZE];
    int tx=threadIdx.x;
    int tid=2*blockIdx.x*blockDim.x+tx;

    s[tx]=a[tid];
    s[tx+BLOCK_SIZE]=a[tid+BLOCK_SIZE];

    for (int dist = BLOCK_SIZE; dist>0; dist/=2) {
        __syncthreads();
        if (tx<dist)
            s[tx]+=s[tx+dist];
    }

    if (threadIdx.x == 0);
        r[blockIdx.x] = s[0];
}
```

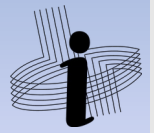
Uso eficiente dos recursos!



Outras considerações em relação à memória

- Memória constante
  - Tamanho mais limitado que memória global (64KB)
  - Apresenta bom desempenho quando threads de uma warp acessam mesmo endereço
  - Desempenho reduzido no acesso serial

```
__constant__ float const_pi;
__constant__ int const_a[100];

__global__ void kernel (int *d_in, int *d_out){
    int tid=blockIdx.x*blockDim.x+threadIdx.x;
    for (int i=0; i<100; ++i)
        d_out[tid] += d_in[tid]*const_pi*const_a[i];
}
int main() {
    ...
    float pi=3.141592;
    cudaMemcpyToSymbol(const_pi,pi,sizeof(float));
    cudaMemcpyToSymbol(const_a,a,100*sizeof(int));
    ...
}
```



Outras considerações em relação à memória

- Alocação dinâmica de memória

```
__shared__ int *s;
if (threadIdx.x==0) s=(int*)malloc(n*sizeof(int));
__syncthreads();
...
__syncthreads();
if (threadIdx.x==0) free(s);
```

  - ou

```
__global__ void kernel(...){
    extern __shared__ int s[];
...
kernel<<<grid, block, n*sizeof(int)>>>(...)
```

  - utilizando um array de int A e um array de float B

```
__global__ void kernel(...){
    extern __shared__ int s[];
    int *sA = s;
    float *sB = (float*)&sA[nA];
...
kernel<<<grid,block,nA*sizeof(int)+nB*sizeof(float)>>>(...)
```