UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS DE QUIXADÁ

Programando Processadores Gráficos Massivamente Paralelos

Prof. João Marcelo Uchôa de Alencar

joao.marcelo@ufc.br

Agenda

1. Introdução
2. Anatomia de uma GPU
3. Ponto de Vista do Desenvolvedor
4. OpenCL

Introdução

- CPUs são projetadas para lidar com tarefas complexas:
  - Desvios condicionais.
  - Compartilhamento de tempo.
- GPUs são projetadas para executar bilhões de operações aritméticas de baixo nível.
- CUDA e OpenCL permitem programar as GPUs de maneira eficiente.

Anatomia de uma GPU

Ideia básica no projeto de uma GPU:

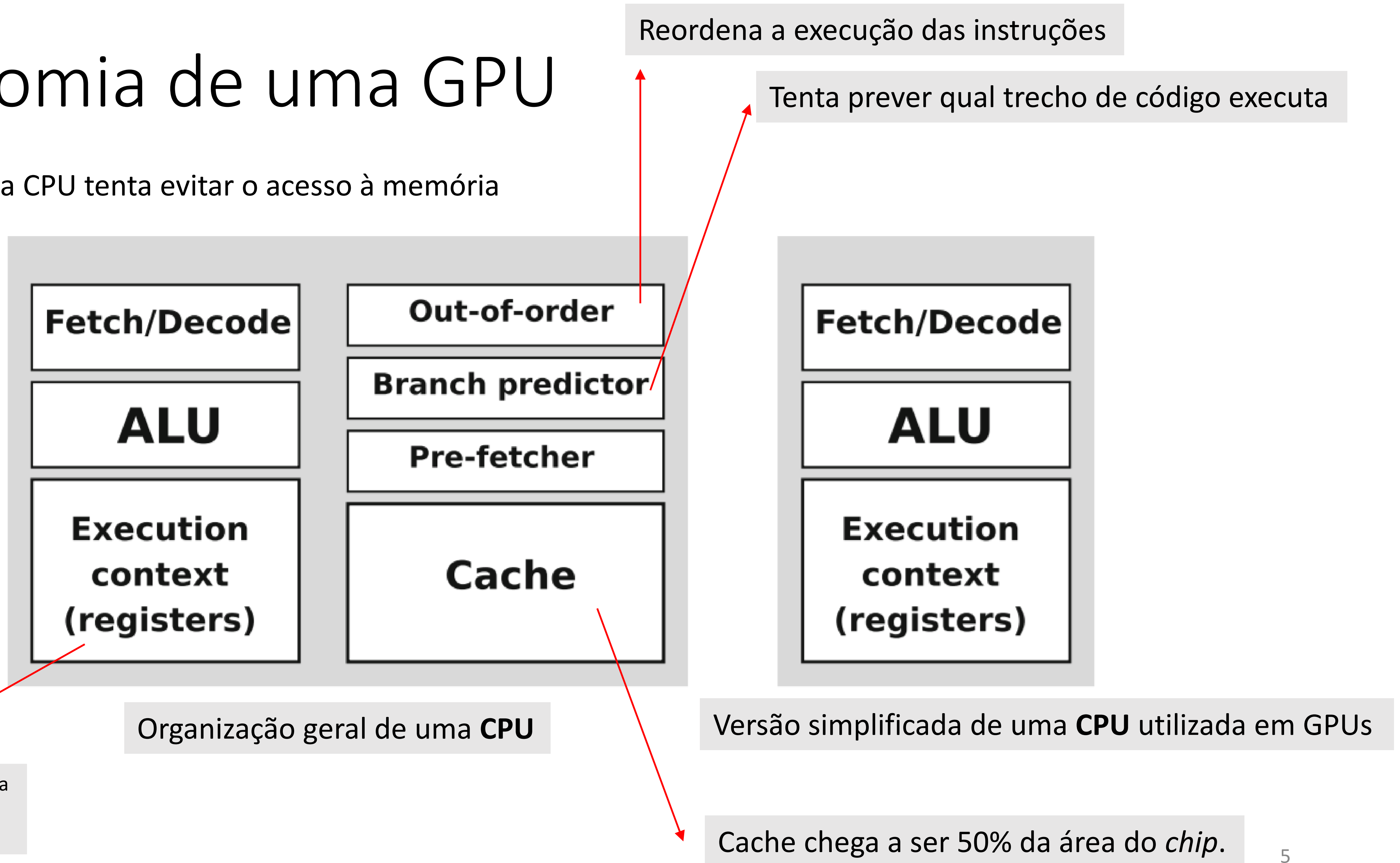
Centenas ou milhares de unidades de processamento simples no lugar de duas ou mais CPUs.

Objetivo:

Inúmeras unidades executando as mesmas instruções, com dados diferentes.

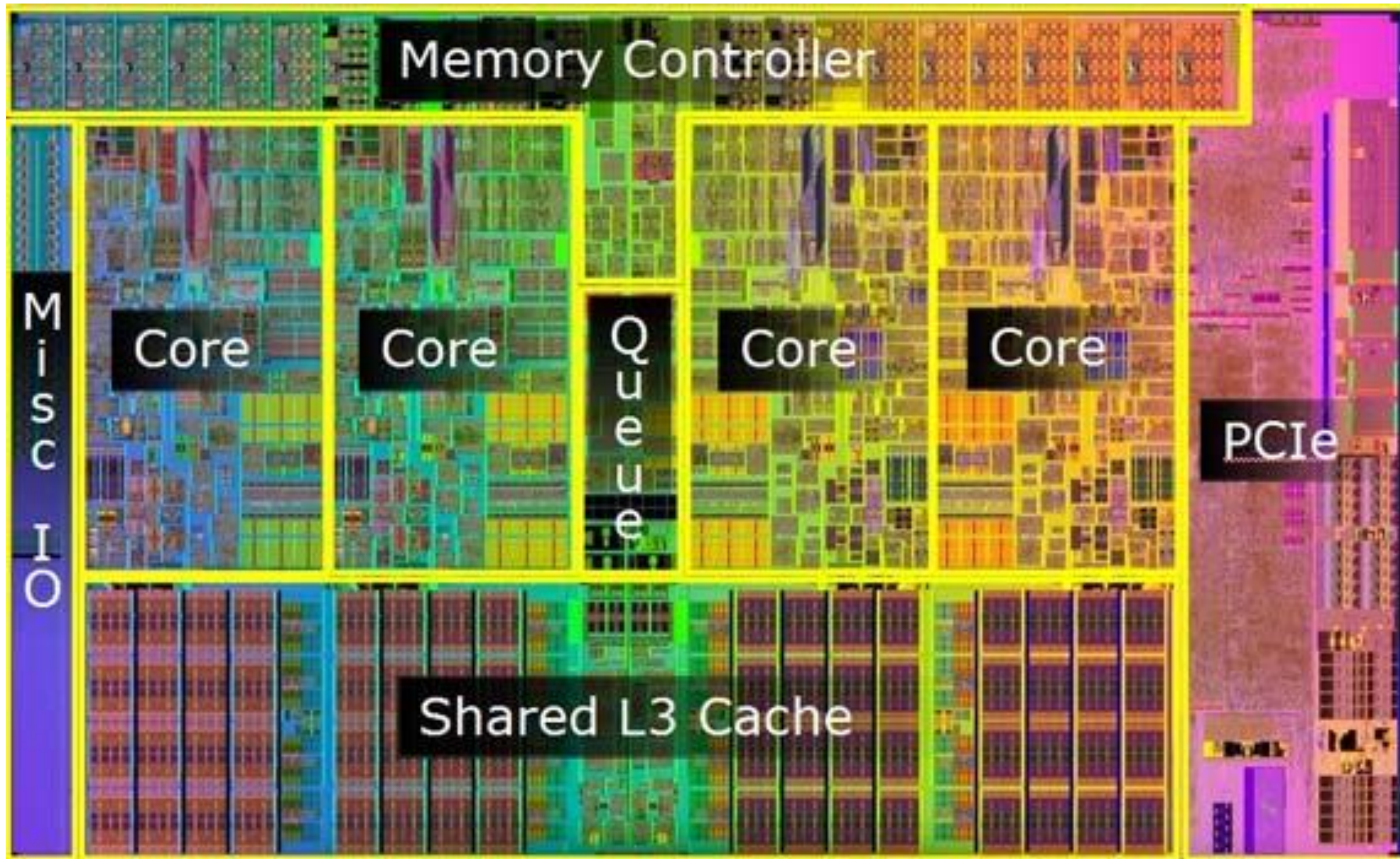
Anatomia de uma GPU

A lógica extra da CPU tenta evitar o acesso à memória



- Contador de programa
- Ponteiro para a pilha
- Registradores

Anatomia de uma GPU



Core i7 Mobile (Clarksfield)

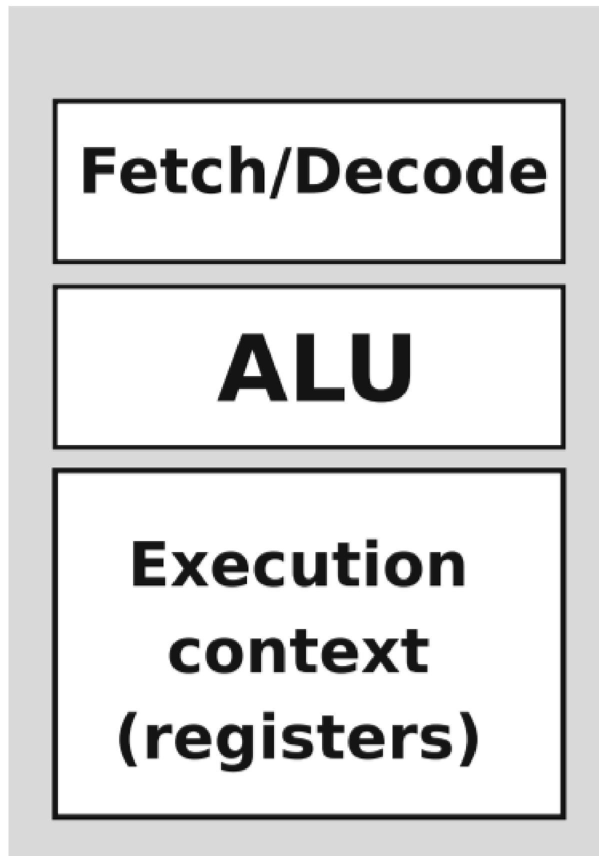
Anatomia de uma GPU

- Uma GPU remove de seu processador mais simples:
  - *Branch-predictor*.
  - *Out of order*.
  - Boa parte do sistema de *cache*.

```
void vectorAdd(float *vecA, float *vecB, float *vecC) {
        int tid = 0;
        while (tid < 128) {
                vecC[tid] = vecA[tid] + vecB[tid];
                tid += 1;
        }
}
```

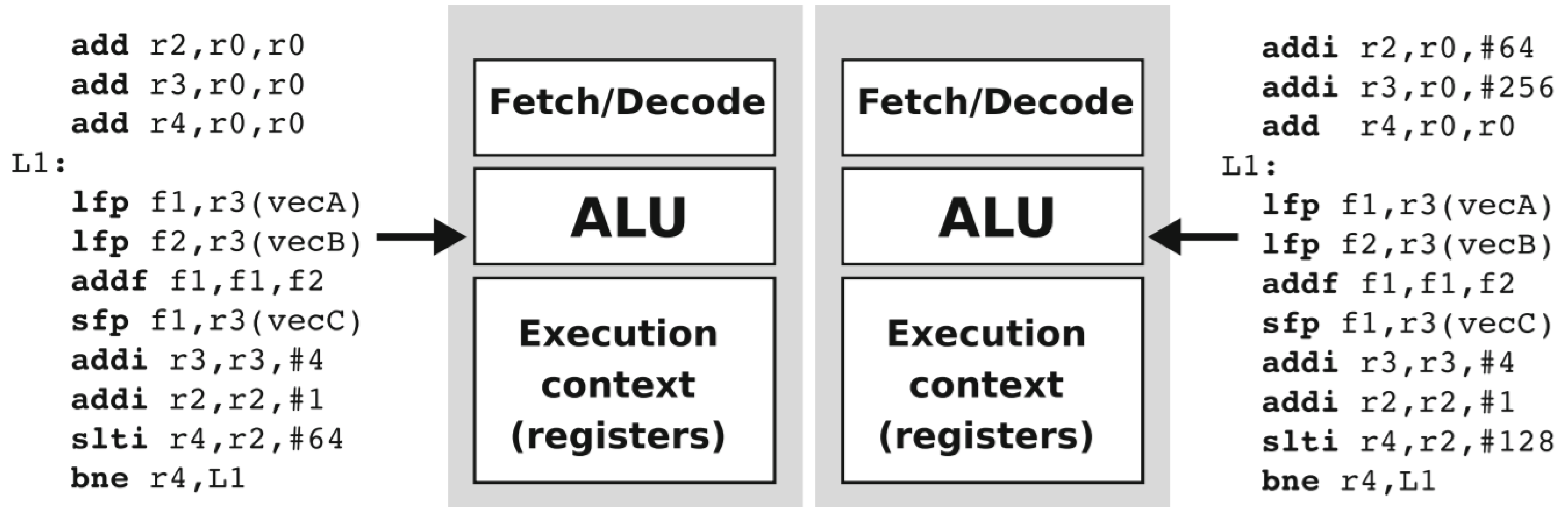
Anatomia de uma GPU

CPU com único núcleo

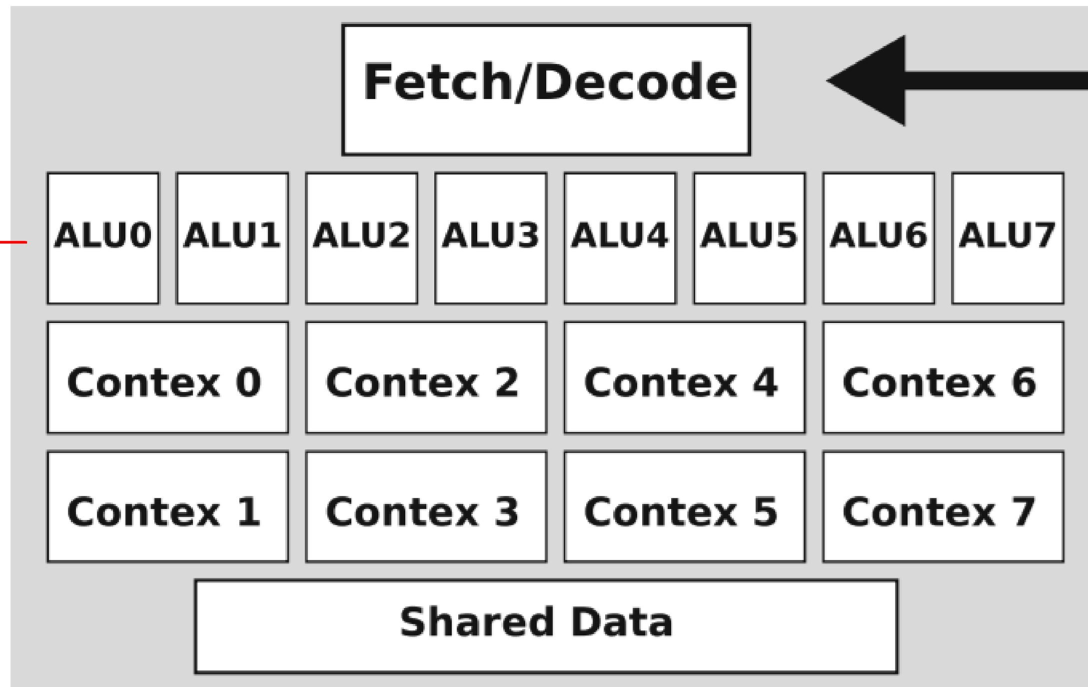



```
    add r2,r0,r0     ; tid=0
    add r3,r0,r0
    add r4,r0,r0
L1:
    lfp f1,r3(vecA)  ; load vectors
    lfp f2,r3(vecB)  ; vecA and vecB
    addf f1,f1,f2    ; add adjacent elements
    sfp f1,r3(vecC)  ; store in vecC
    addi r3,r3,#4
    addi r2,r2,#1    ; tid=tid+1
    slti r4,r2,#128  ;
    bne r4,L1        ; loop back if tid<128
```

Anatomia de uma GPU



Anatomia de uma GPU




```
    addi r2,r0,#tid  ; tid
    addi r3,r0,#tid
    slli r3,r3,#2    ; tid*4
    add r4,r4,r0
L1:
    lfp f1,r3(vecA)  ; add two
    lfp f2,r3(vecB)  ; adjacent elements
    addf f1,f1,f2    ; at tid index
    sfp f1,r3(vecC)
    addi r3,r3,#32
    addi r2,r2,#8    ; tid=tid+8
    slti r4,r2,#128
    bne L1
```

CPU com Unidades Lógicas Aritméticas replicadas

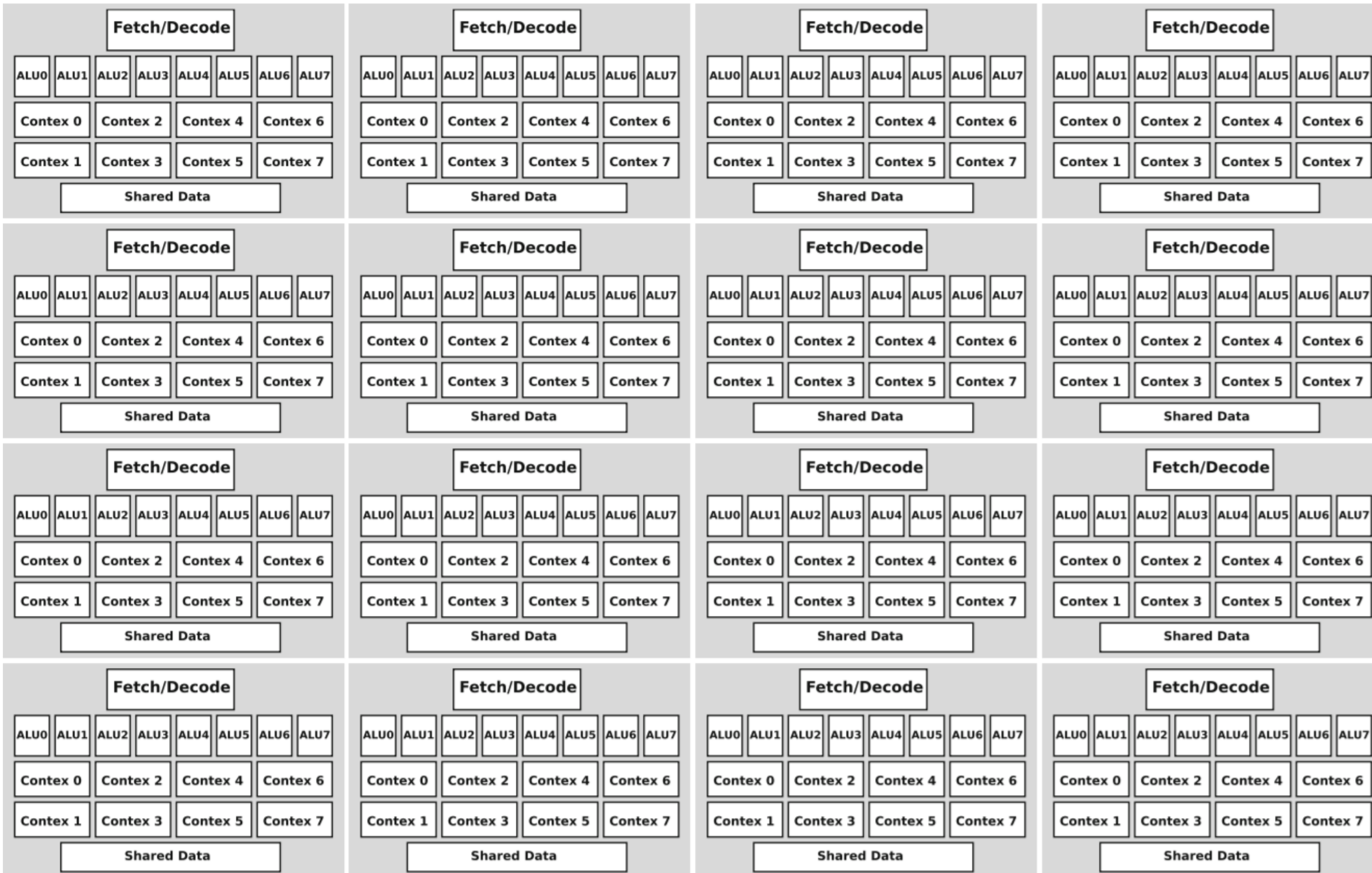
Mesma instrução em cada ALU

Anatomia de uma GPU

- Um núcleo de CPU com unidades ALU replicadas é uma *Unidade de Computação* (**Compute Unit – CU**).
- Cada ALU independente é um Elemento de Processamento (**Processing Element – PE**).
- Ideia básica por trás das GPUs modernas: usar o máximo possível de ALUs para executar o fluxo de instruções em *lock-step*, ou seja, em um **ciclo**, a *mesma instrução* em **dados diferentes**.

Anatomia de uma GPU

```
add r3,r0,#tid
slli r3,r3,#2
lfp f1,r3(vecA)
lfp f2,r3(vecB)
addf f1,f1,f2
sfp f1,r3(vecC)
```



Em cada “núcleo” da GPU

Anatomia de uma GPU

	GeForce GTX280	GeForce GTX580	GeForce GTX780
Microarchitecture	Tesla	Fermi	Kepler
CUs	30	16	12
PEs	240	512	2304
PEs per CU	8	32	192
32-bit registers per CU	16 K	32 K	64 K

AMD Radeon™ RX 5700 XT

Compute Units: 40

Base Frequency: 1605 MHz

Game Frequency*: Up to 1755 MHz

Peak Pixel Fill-Rate: 121.9 GP/s

Peak Half Precision Compute Performance: 19.51 TFLOPs

Peak Single Precision Compute Performance: 9.75 TFLOPs

Stream Processors: 2560

Texture Units: 160

	GEFORCE RTX 2080 Ti FOUNDERS EDITION Especificações do Produto	GEFORCE RTX 2080 Ti Especificações de Referência
Especificações da Engine da Placa de Vídeo:		
NVIDIA CUDA® Cores	**4352**	**4352**

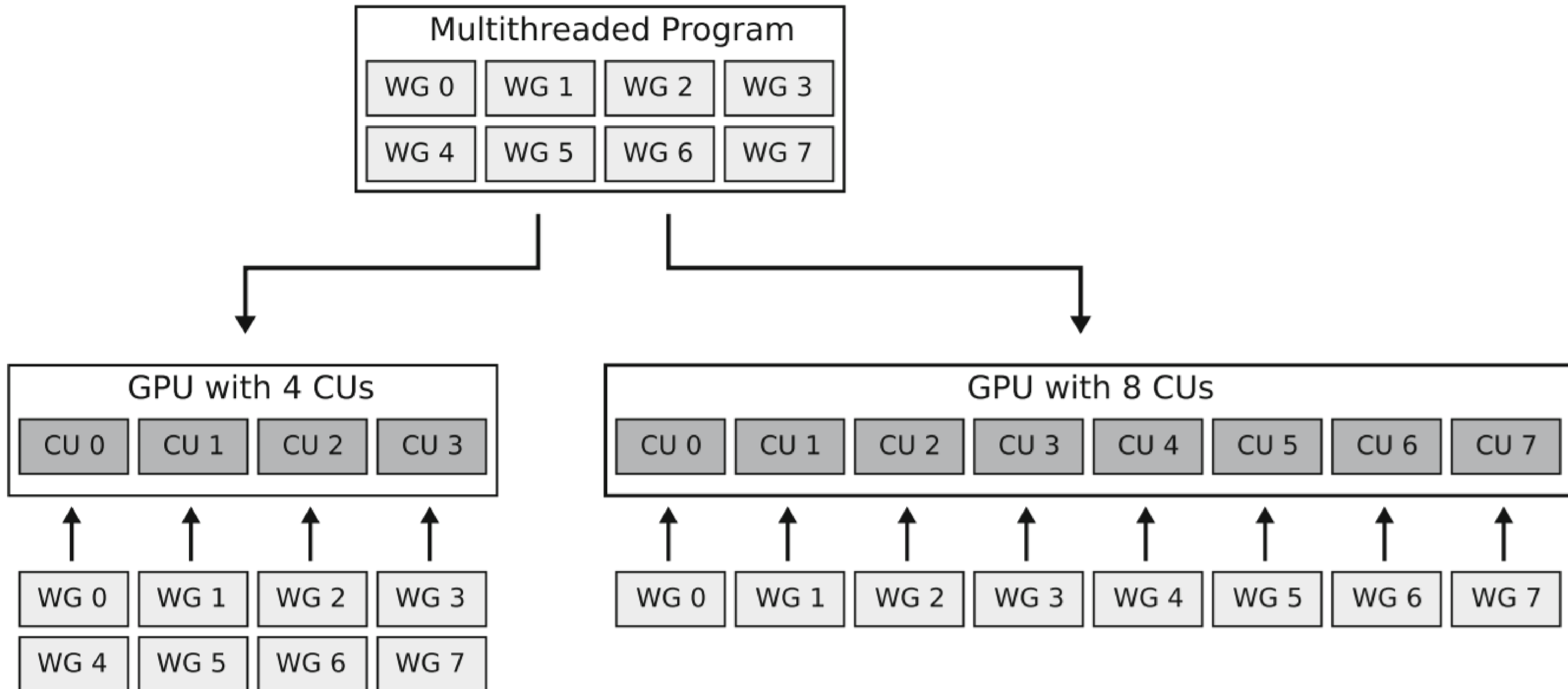
PE nas placas atuais:

- AMD tem *stream processors*.
- NVIDIA tem *cuda cores*.

Anatomia de uma GPU

- No lugar de *threads*, usamos o termo *item de trabalho* (**work-item, WI**)
- Um WI não é escalonado para uma PE de forma individual:
  - Um grupo de WIs é formado (chamado de **work-group, WG**).
  - Um WG é então designado para um **Compute Unit**.
  - Em um mesmo WG, os WIs podem acessar memória compartilhada e sincronizar através de barreiras.
- O desenvolvedor particiona o programa em blocos de *work-items* chamados *work-groups*, que executam de forma independente nas *compute unit*.

Anatomia de uma GPU



Anatomia de uma GPU

Work-items são distribuídos em *Compute Units* em duas etapas:

1. O programador define os work-groups. Após o programa ser compilado, um escalonador em *hardware* distribui *work-groups* para as *compute units*.
2. Dentro das *compute units*, *work-items* são agrupados em subgrupos com tamanho 32, as *warps*. Cada *warp* tem *work-items* com identificadores consecutivos. As *warps* são executadas em paralelo pela *compute unit*.

Associado a cada *warp*, há valores de registradores (contexto) como o contador de programa.

Anatomia de uma GPU

- Todos os WIs em uma *warp* executam a mesma instrução por ciclo, em dados diferentes.
- Eficiência máxima ocorre quando todos os WIs executam a mesma <u>sequência</u> de instruções.
- Uma estrutura de decisão pode complicar a situação.

Se metade (16) dos WIs em uma *warp* precisa ir para o *trecho 01*, então a outra metade “bloqueia”.

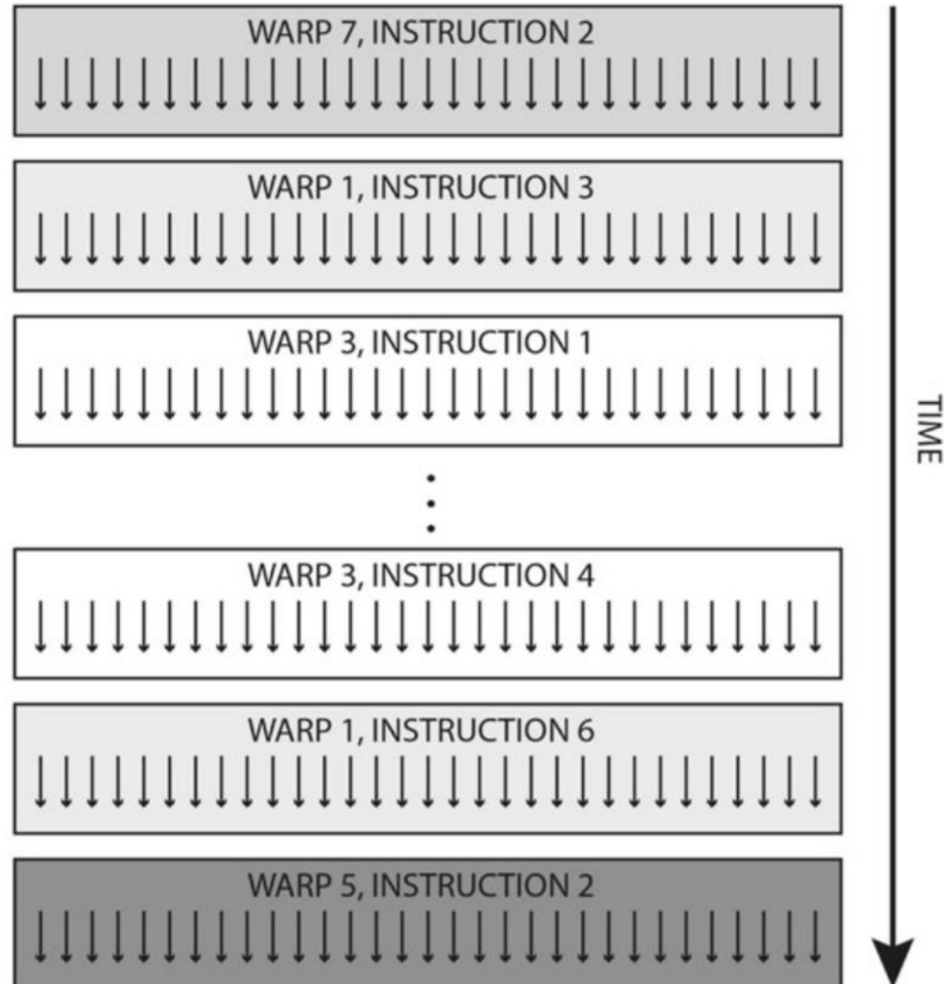
```
if (value < 100) {
		// trecho 01
} else {
		// trecho 02
}
```

O que acaba ocorrendo é a serialização de *trecho 01* e *trecho 02*.

Anatomia de uma GPU

- Apesar de cada *work-item* poder ter seu contador de programa, a *warp* em execução só um registrador para tal valor.
- Um *work-item* pode ser bloqueado pelo acesso à memória:
  - De todos os *warps* criados, o escalonador escolhe um dentre aqueles que tem *work-items* ativos (desbloqueados) para despachar para execução.
  - O número de ciclos necessários para que uma *warp* fique pronta para execução é a latência.
  - Quando todas as *warps* tem algum *work-item* pronto para executar, a latência é ocultada: trabalho é feito até precisar de acesso a memória, a *warp* é “bloqueada”, outra é carregada.
- O contexto de execução de cada *warp* é mantido internamente na *compute unit* enquanto ela não finaliza sua execução.

Anatomia de uma GPU



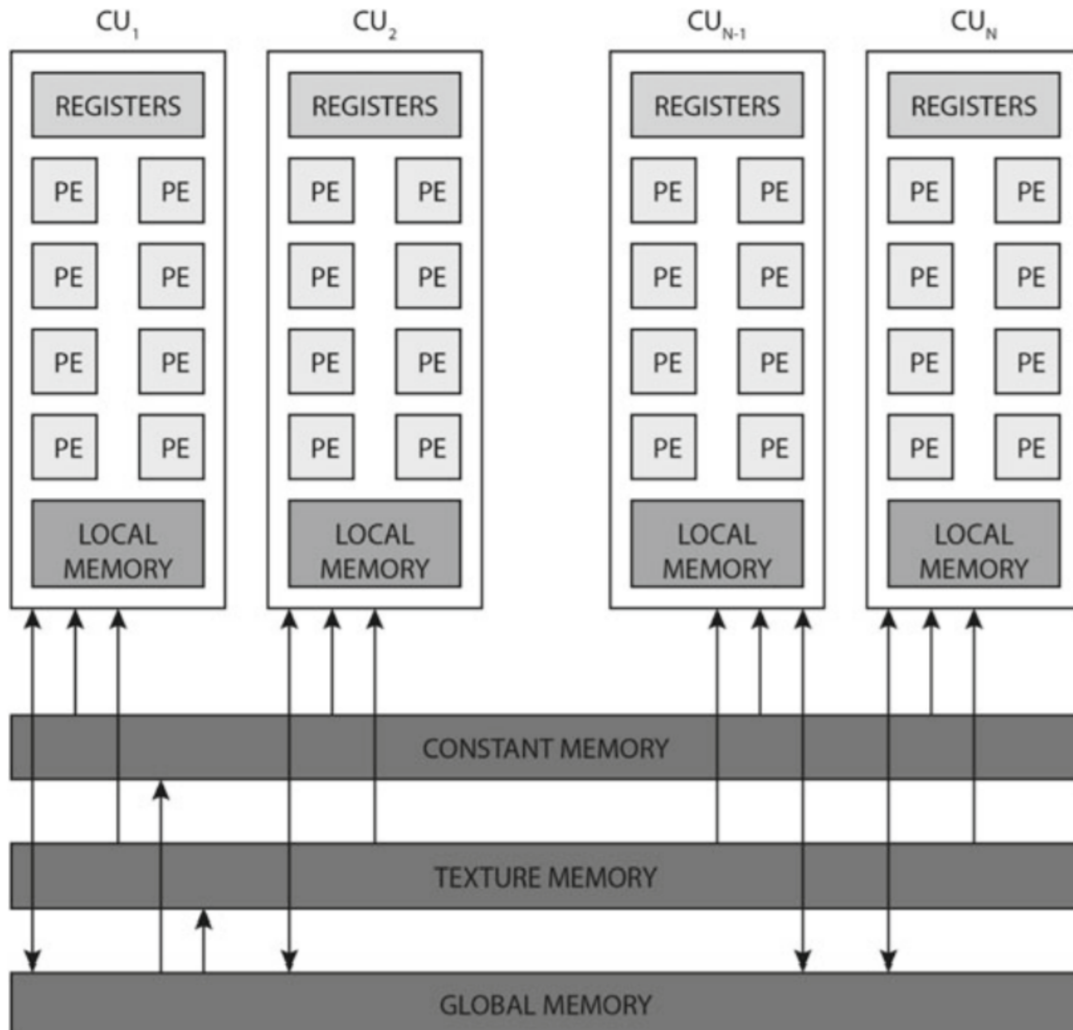
Para efeito didático, imagine as warps como os processos escalonados em um SO, mas bloqueadas pelo acesso à memória ou barreiras de sincronização.

Anatomia de uma GPU

- Tempo de acesso à memória:

Storage type	Access time
Registers	1 cycle
Local memory	1–32 cycles
Texture memory	~500 cycles
Constant memory	~500 cycles
Global memory	~500 cycles

Anatomia de uma GPU



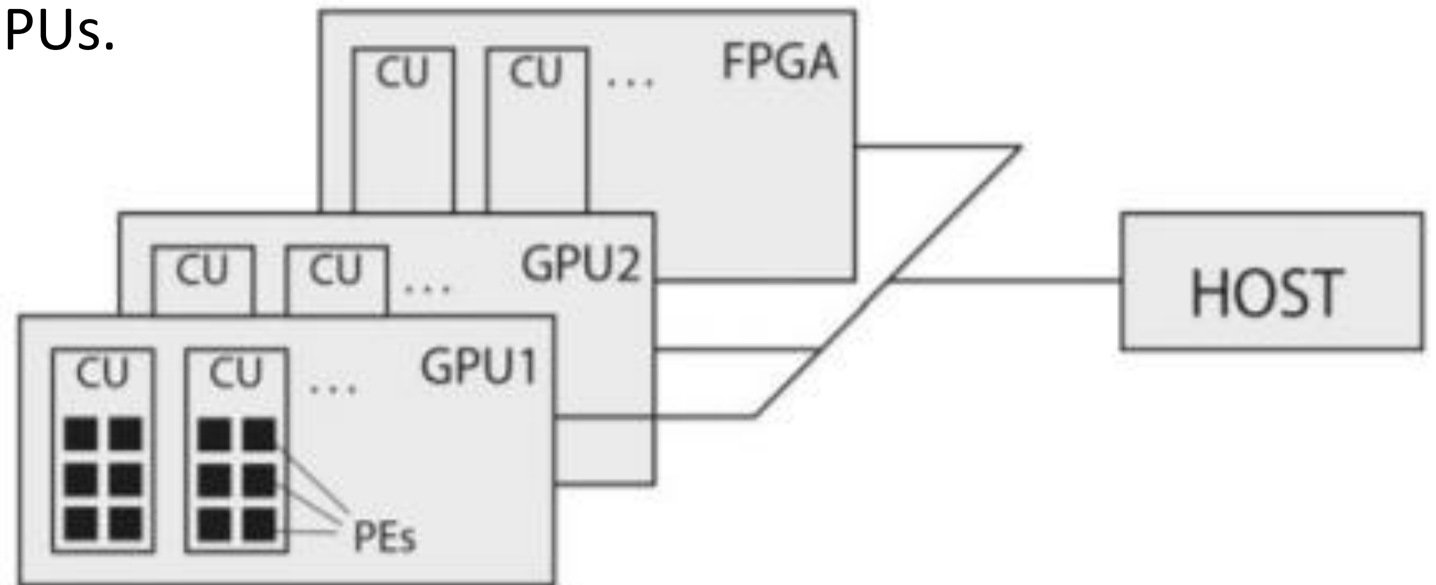
- **Memória global**: GDDR de alta velocidade, acessível tanto pela GPU quanto pela CPU.
    - Leituras e escritas com palavras de 128 *bytes*, os segmentos.
    - Se dois *work-items* estão bloqueados por dados de um mesmo segmento, só é necessário um acesso à memória para desbloqueá-los.
    - *Work-items* de uma mesma *warp* deve acessar elementos contínuos na memória.
    - *Memory coalescing*: tentar agrupar o várias requisições em uma só.
- **Memória constante**: apenas leitura.
    - Tem estrutura de *cache*.
    - Permite broadcast de valores para vários *work-items*.
- **Memória de textura**: otimização para acesso espacial 2D.
    - Texturas e imagens.

Visão do Programador

- É responsabilidade do programador particionar programas em *work-groups* de *work-items*.
- OpenCL (*Open Computing Language*)
  - Linguagem de programação baseada em C.
    - Usada para escrever funções (***kernels***) executadas por todos os *work-items*.
    - Em geral, a sintaxe para escrever *kernels* não é complexa.
  - API para controlar dispositivos e executar programas nesses dispositivos.
    - Muitas funções, com chamadas em baixo nível.
    - Não é fácil de aprender.
  - Modelo de <u>plataforma</u>, modelo de <u>execução</u> e modelo de <u>memória</u>.

Visão do Programador

- Sistema Heterogêneo
  - Programa *host*, em execução na CPU
  - *Kernels*, em execução nas GPUs.



Visão do Programação

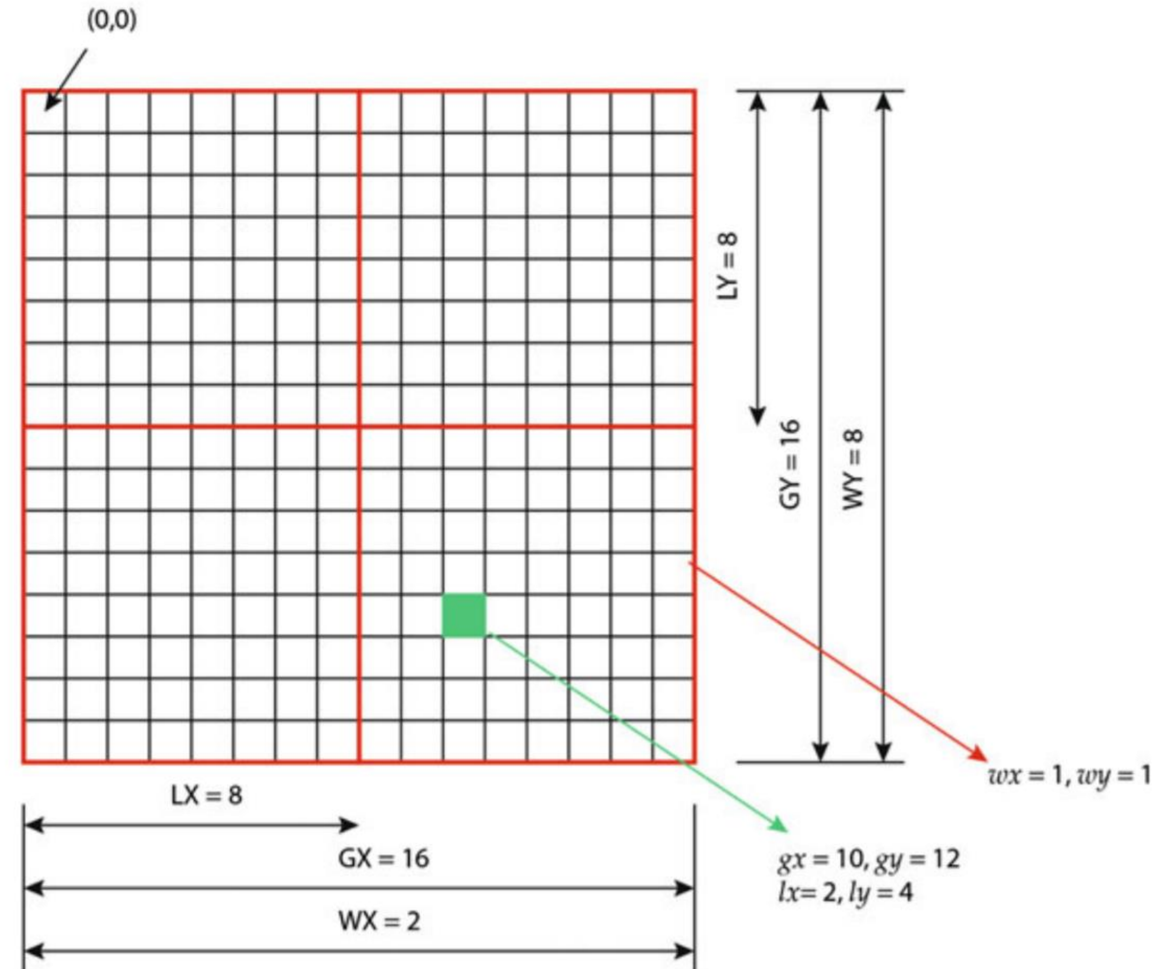
- Modelo de Execução
  - Execução NDRange (*N-Dimensional Range*)
    - Os *work-items* são organizados em, no máximo, três dimensões.
    - Cada *work-item* tem seu identificador (coordenada) em cada dimensão.
    - A execução não obedece a ordem em qualquer dimensão, em geral, é paralela.
  - Como já afirmamos, *work-items* são organizados em *work-groups.*
    - Compartilhamento de Dados.
    - Sincronização.
    - Todos *work-items* de um *work-group* são executados em um mesmo *Compute Unit*.
  - *Global work size*: total de *work-items*.
  - *Local work size*: quantidade de *work-items* em um *work-group*.

Visão do Programação

- A função *kernel* é escrita no *host*.
- O programa *host* compila e submete para execução no dispositivo.
  - Esse programa é responsável por criar a coleção de *work-items*.
- Considere uma organização em duas dimensões:
  - Cada *work-item* tem um índice global (g_x, g_y).
  - Dentro de um *work-group,* cada *work-item* tem um índice local (l_x, l_y).
  - E cada *work-group* tem seu próprio índice (w_x, w_y).
  - Finalmente, os próprios *work-groups* são organizados nas dimensões (WX, WY).

Visão do Programador

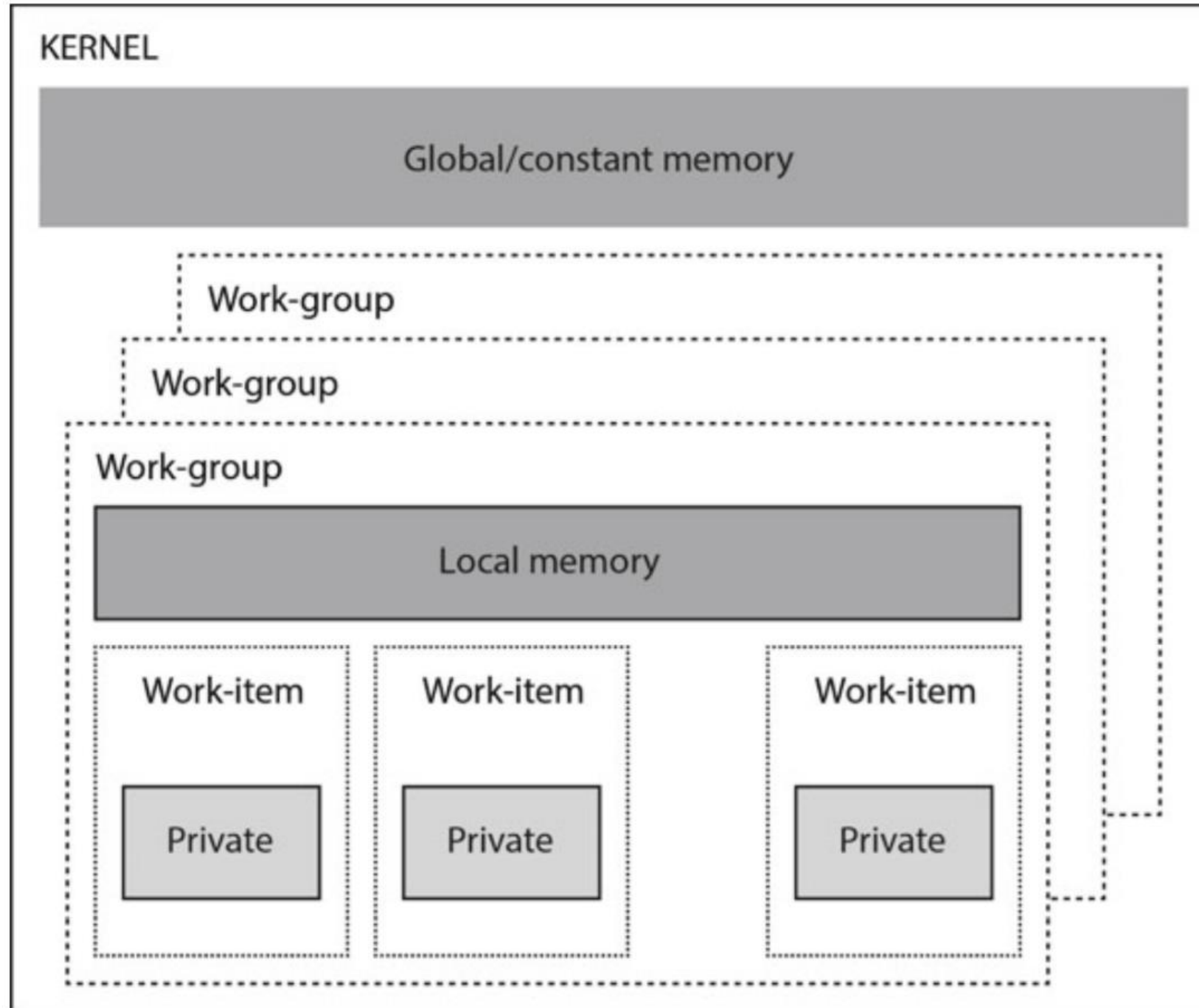
Exemplo de uma organização *NDRange* em duas dimensões



Visão do Programador

- Modelo de Memória
  - **Privada**: região de memória exclusiva do *work-item*. Em uma GPU, está fisicamente dentro da *Compute Unit*.
  - **Local**: região de memória compartilhada dentro de um *work-group*. Todos os *work-items* tem acesso leitura/escrita. Em uma GPU, está fisicamente dentro da *Compute Unit*.
  - **Global**: região de memória acessível por todos os *work-items* em todos os *work-groups*. Implementada em GDDR5, alocada pelo *host.*
  - **Constante**: semelhante a global, mas apenas de leitura pelos *work-items*, escrita pelo *host.*
- No código do *kernel*, os operadores *__global*, *__constante* e *__local* indicam o tipo de memória, sem eles o padrão é acessar a privada.

Visão do Programador



Programando em OpenCL

- Simples adição de dois vetores.
  - Uma única dimensão, para começar.
  - Vamos configurar na executar na GPU.
- Precisamos criar:
  - Função *kernel.*
  - Programa *host.*

```
void VectorAdd( float *a,
                float *b,
                float *c,
                int iNumElements) {
    int iGID = 0;
    while (iGID < iNumElements) {
        c[iGID] = a[iGID] + b[iGID];
        iGID += 1;
    }
}
```

Programando em OpenCL

- Função Kernel
  - https://github.com/jmhal/parallel/blob/master/opencl/vector_addition/VectorAdd.cl
  - A ideal geral é substituir o iterações com invocações da função por cada *work-item.*
  - O qualificador __*kernel* indica que a função irá executar no dispositivo.
  - Você pode colocar a função em um arquivo .cl:
    - https://github.com/jmhal/parallel/tree/master/opencl/vector_addition
  - Ou colocar o código em uma *string* C:
    - https://github.com/jmhal/parallel/tree/master/opencl/vector_addition_inline

Programando em OpenCL

- Etapas que o código *host* deve realizar:
  1. Descobrir o *hardware* OpenCL (plataformas e dispositivos).
  2. Recuperar as características dos dispositivos.
  3. Carregar o programa fonte do *kernel*.
  4. Alocar memória no dispositivo e transferir dados da CPU.
  5. Compilar e executar o *kernel*.
  6. Recuperar da memória do dispositivo o resultado.
- É um código muito poluído, baixo nível, mas pelo menos pode ser reutilizado para diversos *kernels* com alterações pontuais.
- https://github.com/jmhal/parallel/blob/master/opencl/vector_addition/host.c

Programando em OpenCL

1. Prepara e inicializar dados no *host*.
2. Descobrir e inicializar os dispositivos.
3. Criar um contexto.
4. Criar uma fila de comando.
5. Criar o objeto programa no contexto.
6. Construir o programa OpenCL.
7. Criar os *buffers* no dispositivo.
8. Escrever dados nos *buffers* do dispositivo.
9. Criar e compilar o *kernel.*
10. Configurar os argumentos do *kernel.*
11. Definir o modelo de execução e enfileirar o *kernel* para execução.
12. Recuperar o resultado da memória do dispositivo para o *host.*

Variam de acordo com a aplicação.
O resto é fixo.

Programando em OpenCL

- Cada aplicação *host* tem cinco estruturas de dados principais:
  - *cl_device_id*: identificador do dispositivo a ser usado para executar o *kernel.*
  - *cl_context*: estado das transferências de memórias entre CPU e o dispositivo.
  - *cl_command_queue*: fila que permite enviar comandos (ações) para o dispositivo.
  - *cl_program*: objeto que reúne um ou mais *kernels*.
  - *cl_kernel*: função a ser executada pelos *work-items.*
- Existem chamadas na API do OpenCL para popular e manipular cada uma delas.

Programando em OpenCL

- Descobrir e Inicializar dispositivos:
  - clGetDevicesIDs()
    - Primeira invocação: recuperar o número de dispositivos na plataforma.
    - Segunda invocação: recuperar uma lista dos dispositivos.
  - Parâmetros:
    - *cl_platform_id platform*: identificador da plataforma ou NULL.
    - *cl_device_type device_type:* se deseja recuperar CPU, GPU ou qualquer um.
    - *cl_uint num_entries*: o número de entradas para *devices*.
    - *cl_device_id *devices*: a lista de dispositivos. Se for NULL, retorna a quantidade de dispositivos em *num_devices*.
    - *cl_uint *num_devices:* a quantidade de dispositivos.
  - Retorna CL_SUCCESS se executada corretamente.

Programando em OpenCL

- Descobrir e Inicializar dispositivos:
  - clGetDeviceInfo()
    - Retorna informações sobre um dispositivo.
  - Parâmetros:
    - *cl_device_id device*: identificador do dispositivo.
    - *cl_device_info param_name:* uma enumeração que informa qual informação é desejada.
    - *size_t param_value_size*: o tamanho em *bytes* do valor a ser retornado.
    - *void *param_value*: buffer que irá armazenar a informação recuperada.
    - *size_t *param_value_size_ret:* tamanho exato retornado.
  - Retorna CL_SUCCESS se executada corretamente.

Programando em OpenCL

- Criar um contexto:
  - clCreateContext()
    - Agrupar dispositivos e objetos de memória em uma estrutura, para permitir emissão de comandos de comandos.
- Cria uma fila de comandos:
  - clCreateCommandQueue()
    - Uma fila de ações por dispositivo.
    - A ação mais comum é submeter um *kernel* para execução.

Programando em OpenCL

- Criar um objeto programa:
  - clCreateProgramWithSource()
    - Cria um objeto programa com um ou mais *kernels* e funções auxiliares.
    - Pode usar um binário compilado externamente, carregar as funções de um arquivo ou de uma variável *string*.
- Constrói o Programa:
  - clBuildProgram()
    - Compila o código do programa.
    - Além da compilação, pode fazer ligação com as bibliotecas necessárias.

Programando em OpenCL

- Aloca memória no dispositivo:
  - clCreateBuffer()
    - Cria ponteiros para a memória alocada no dispositivo.
    - Não faz sentido referenciá-los no código *host.*
    - São passados como argumentos para o *kernel.*
- Transfere os dados para o dispositivo:
  - clEnqueueWriteBuffer()
    - Transfere objetos da memória da CPU para a GPU.
- Criar e compilar o *kernel.*
  - clCreateKernel()
    - Escolhe uma função dentro do programa para ser o *kernel* a ser lançado.
    - Faz a compilação final específica para o dispositivo.
    - Cria o objeto *kernel* na memória do dispositivo.

Programando em OpenCL

- Configura os argumentos do *kernel*
  - clSetKernelArg()
- Coloca o *kernel* na fila de execução.
  - clEnqueueNDRangeKernel()
    - Fornece a quantidade de dimensões para organização dos *work-items.*
    - Também é necessário informar a quantidade total de *work-items* e o tamanho do *work-group.*
- Recupera o resultado para o *host.*
  - clEnqueueReadBuffer()

Soma de Vetores com Tamanho Arbitrário

- Nós definimos a quantidade de *work-items* (512 * 512) e o tamanho dos *work-groups* (512):
  - São criados 262.144 *work-items*, agrupados de 512 em 512, em 512 *work-groups.*
  - Lembrando que cada *work-group* é designado a um *Compute Unit*.
- O padrão OpenCL não determina como é feito o mapeamento para o *hardware.*
  - Cada *work-group* de 512 será repartido em *warps* de 32.
  - Mas, em teoria, não há limites para a quantidade de *work-items* e *work-groups.*
  - Na prática, limitações de memória e arquitetura impõem restrições.

Soma de Vetores com Tamanho Arbitrário

- Restrições em relação ao *work-group:*
    1. O número de *work-items* deve ser divisível pelo tamanho do *work-group.*
    2. O tamanho do *work-group* deve ser no máximo CL_DEVICE_MAX_WORK_GROUP_SIZE (obtido por *clGetDeviceInfo*).
- O número máximo de *work-groups* por *Compute Unit* também é limitado de acordo com a placa.
- Também há um limite para a quantidade de *warps* ativas.
- Ocupação: a média de *warps* ativas por *Compute Unit* em relação ao máximo permitido.

Soma de Vetores com Tamanho Arbitrário

- Somar vetores de 512 x 512 elementos (*iNumElements*) usando 512 x 512 *work-itens* (*szGlobalWorkSize*) é direto:
  - Cada *work-item* soma um elemento do vetor.
  - Mas como somar vetores com mais elementos do que o limite de *work-items* que a placa suporta?
- A solução é fazer com que cada *work-item* realize mais do que uma única soma.

Soma de Vetores com Tamanho Arbitrário

```
__kernel void VectorAdd(__global float* a,
                        __global float* b,
                        __global float* c,
                        int iNumElements) {

        // Achar o índice global
        int iGID = get_global_id(0);

        while (iGID < iNumElements) {
                c[iGID] = a[iGID] + b[iGID];
                iGID += get_global_size(0);
        }
}
```

- Teremos menos *work-items* do que posições no vetor.
- A soma será feita em várias etapas:
  - Em cada etapa, os *work-items* cooperam para somar uma partição do vetor.
  - Em seguida, cada *work-item* “pula” um número de posições equivalente ao total de *work-itens*.
- Alguns *work-itens* ficam ociosos na última etapa se o tamanho do vetor não for divisível pela quantidade de *work-itens*, mas em geral são poucos.
- Podemos então ter números menores para quantidade total de *work-items* e de *work-groups*
  - *szLocalWorkSize = 256*
  - *szGlobalWorkSize = 512 * 256*

Produto Escalar em OpenCL

- Produto escalar entre dois vetores
  - Primeiro uma versão simples, quase a mesma coisa da soma.
  - Depois, vamos usar a memória local da *Compute Unit* para otimização.
- O algoritmo geral funciona em duas etapas:
  1. Multiplicamos os elementos correspondentes nos dois vetores.
  2. Fazemos a soma total das multiplicações.
- Cada *work-item*:
  - Irá multiplicar os elementos que é responsável.
  - Manterá uma soma parcial da sua contribuição.

Produto Escalar em OpenCL

```
__kernel void DotProductNaive(__global float* a,
                              __global float* b,
                              __global float* c,
                              int iNumElements) {

        // Achar o índice global
        int iGID = get_global_id(0);
        int index = iGID;

        while (index < iNumElements) {
                c[iGID] += a[index] * b[index];
                index += get_global_size(0);
        }
}
```

- Como cada *work-item* mantém sua soma parcial, precisamos de um vetor C final apenas com tamanho igual a quantidade de *work-items*.
- Vamos copiar este vetor, menor, de volta à CPU e somá-lo.

Medindo o Desempenho de um *Kernel*

- O OpenCL tem um mecanismo de criação de perfil de desempenho embutido.
- Os eventos são usados para medir o tempo que o dispositivo (GPU) gasta para completar ações inseridas na fila de comandos.
  - Transferência de dados.
  - Execução de *Kernel*.
- É preciso criar a fila com a *flag* CL_QUEUE_PROFILING_ENABLE ativada.
- Para cada iteração com a fila, um evento é anexado para podermos monitorar a ação associada.

Medindo o Desempenho de um *Kernel*

- *clGetEventProfilingInfo()*
  - Quando o segundo parâmetro é CL_PROFILING_COMMAND_START, retorna o tempo em nano segundos quando o evento começou a ser tratado pelo dispositivo.
  - Quando o segundo parâmetro é CL_PROFILING_COMMAND_END, retorna o tempo em nano segundos quando o evento terminou de ser tratado pelo dispositivo.
- Calculando a diferença, sabemos quanto tempo o dispositivo levou para executar a ação.

Produto Escalar em OpenCL – Memória Local

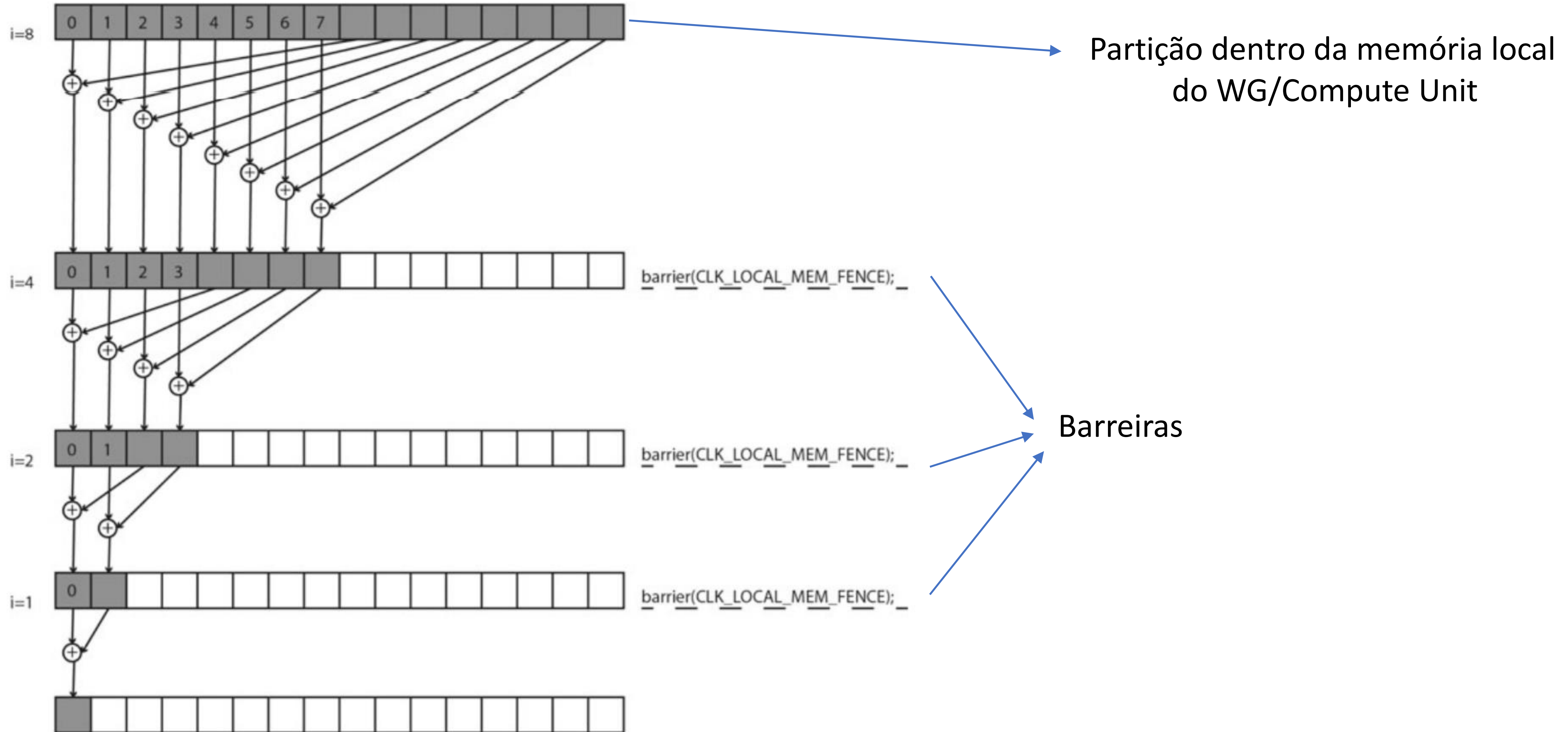
- Os *work-items* podem cooperar através da memória local na *Compute Unit* através de sincronização.
- Essa memória, apesar de ser bem menor que a memória global da GPU, é mais rápida.
- Vamos criar dentro de um *Compute Unit* um vetor temporário (tamanho *szLocalWorkSize)* para guardar o resultado parcial dos *work-items* de um *work-group* escalonado para o *Compute Unit.*

```
__kernel void DotProductNaive(__global float* a,   __global float* b,   __global float* c,
                                                                    __local* ProductsWG,
                                                                    int iNumElements) {
```

Produto Escalar em OpenCL – Memória Local

- Como é memória local da *Compute Unit*, a alocação é feita na definição do parâmetro do *kernel*.
- Ao final da execução dos produtos, precisamos somar todos os valores parciais em *ProductsWG*, que agora não é na memória global, portanto não pode ser copiado para a CPU diretamente.
- Faremos uma operação de **redução**:
  1. Todos os produtos são realizados, com as somas parciais em *ProductsWG*.
  2. Os *work-items* sincronizam em uma barreira.
  3. Alguns *work-items* interagem para somar os elementos do vetor *ProductsWG*.

Produto Escalar em OpenCL - Redução

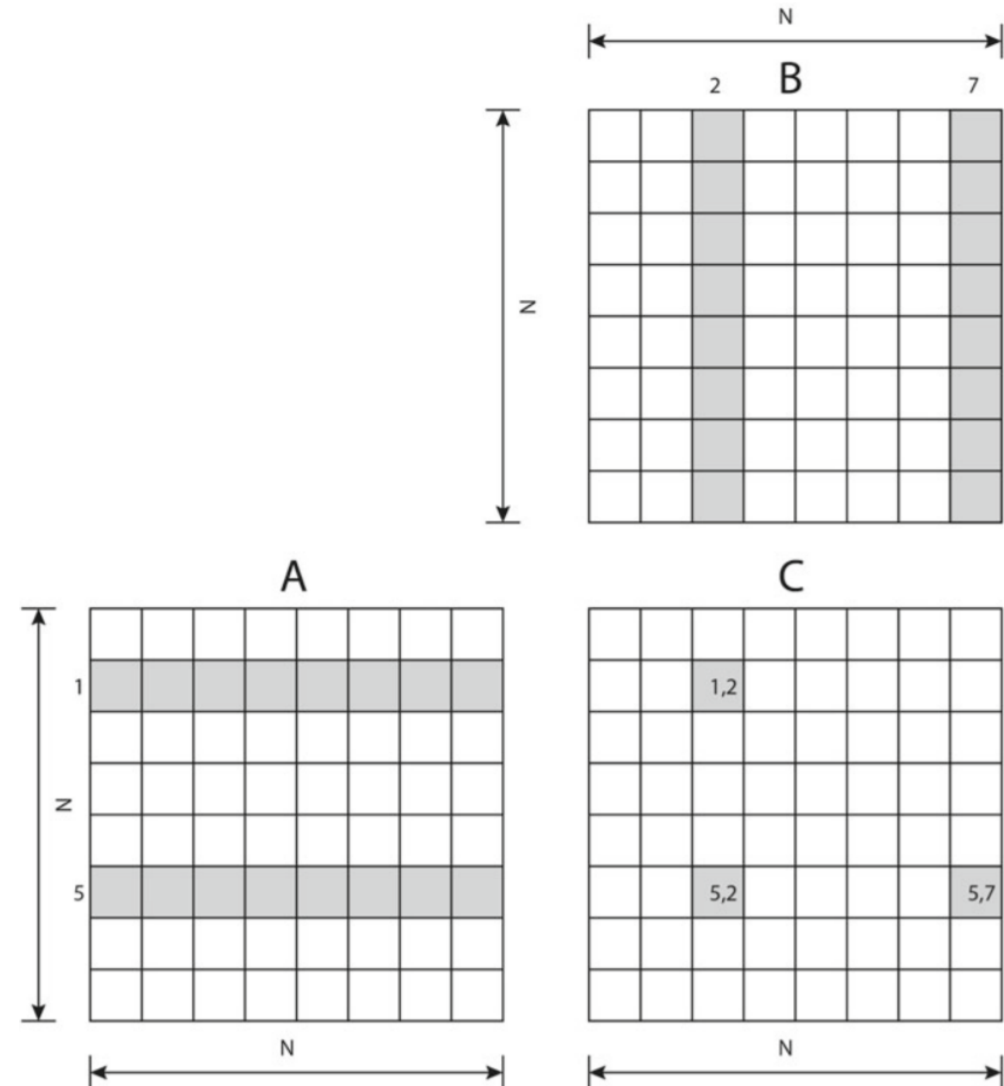


Multiplicação de Matrizes em OpenCL

- Vamos descrever uma versão inicial da Multiplicação de Matrizes.
- Memória global é utilizada para armazenar a entrada e saída.
- Matrizes quadradas com dimensão múltipla de 32.
- Distribuição de Computação:
  - Cada *work-item* calcula um elemento da matriz final.
  - Os *work-items* agora são organizados em duas dimensões.
    - *Work-item* (*i, j*) calcula elemento $c_{i,j}$ usando linha *i* de A e coluna *j* de B.
    - *N x N work-items*.
- A matriz A é lida *N* vezes da memória global.
- A matriz B é lida *N* vezes da memória global.

Multiplicação de Matrizes

- Vamos criar um kernel em duas dimensões.
- Como os *work-items* são organizados no plano?
- Lembre-se que a origem (0,0) fica no canto superior esquerdo.



Multiplicação de Matrizes em OpenCL

```
void matrixmul( float * matrixA , float *matrixB ,
                    float *matrixC , int N) {
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++) {
     for (int k = 0; k < N; k++) {
       matrixC[i*N + j] +=  matrixA[i * N + k] *  matrixB[k * N + j];
      }
   }
 }
}
```

Versão serial

```
__kernel void MatrixMultNaive(__global float* matrixA, __global
float*          matrixB, __global float* matrixC, int N) {
  // Achar os índices globais
  int xGID = get_global_id(0); // coluna
  int yGID = get_global_id(1); // linha
  float dotprod = 0.0;

  for (int i = 0; i < N; i++) {
    dotprod += matrixA[yGID * N + i] * matrixB[i * N + xGID];
  }
  matrixC[yGID * N + xGID] = dotprod;
}
```

Kernel OpenCL

Multiplicação de Matrizes

- Mudanças no código *host:*
    1. Definir a dimensão das matrizes.
    2. Alocar as matrizes de forma correta.
    3. Informar o novo código do *kernel.*
    4. Ajustar os parâmetros do *kernel*.
    5. Recuperar e verificar os valores.
- Boa parte do código anterior é reaproveitado.

Multiplicação de Matrizes – Memória Local

- No *kernel* para multiplicação de matrizes, cada *work-item* carrega para o cálculo *2 x N* elementos da memória global:
  - Dois em cada iteração do laço.
  - Um elemento da linha de A e um elemento da coluna de B.
- O acesso a memória global é mais lento, tendo efeito negativo no desempenho.
- No cálculo de um elemento de C em uma coluna:
  - Usamos a mesma coluna de A.
  - *Work-items* em um *work-group* usam a mesma coluna de B.

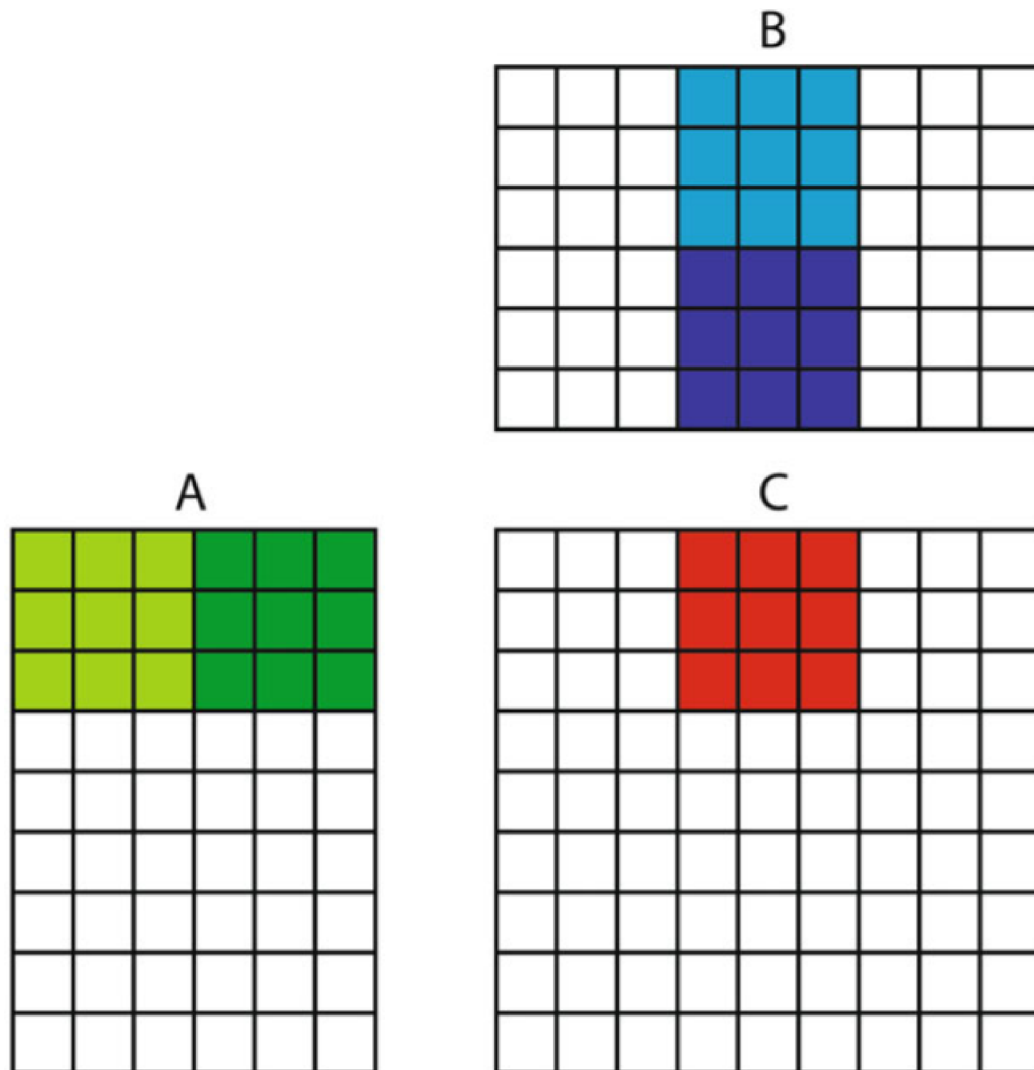
Multiplicação de Matrizes – Memória Local

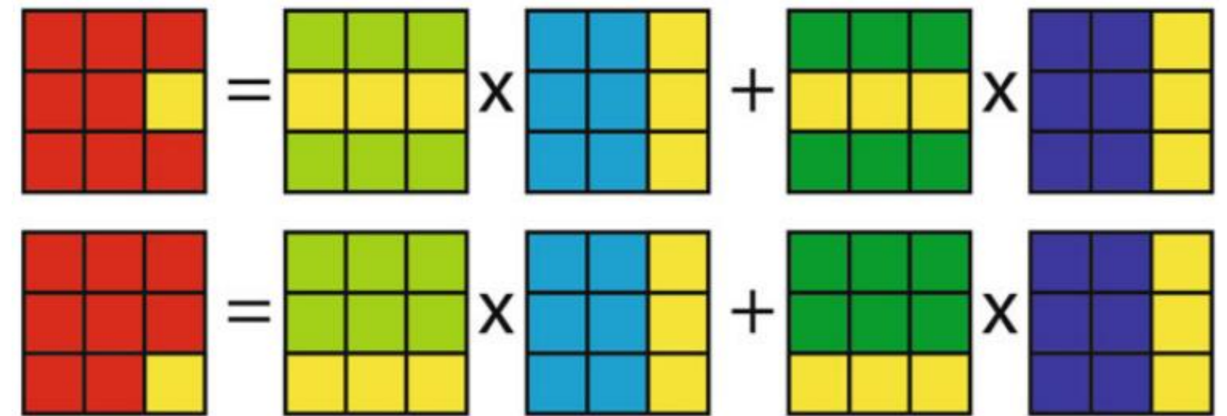
- Além do acesso frequente a memória, o mesmo nem sempre é sequencial (*coalescing*):
  - O descolamento (*stride*) dos elementos de A são em uma unidade do tamanho do tipo.
  - Agora o deslocamento em B são de N em N unidades do tamanho do tipo.
- O ideal é que *work-items* da mesma *warp* acessem elementos contínuos da memória global para minimizar a latência.
- As placas atuais tem sistema de *cache* que minimizam o problema, mas se o desenvolver conseguir carregar os dados na memória local da *Compute Unit* o desempenho será superior.

Multiplicação de Matrizes – Memória Local

- Dividir as matrizes em submatrizes (partições, “ladrilhos”, *tiles*)
  - Realizar a multiplicação das submatrizes.
  - Somar os resultados parciais para obter a matriz final.
- Definimos as submatrizes de tamanho que caiba na memória local.
- O número de *work-items* continua sendo igual ao número de elementos da matriz resultado.
- O *work-group* terá o tamanho da partição.

Multiplicação de Matrizes – Memória Local





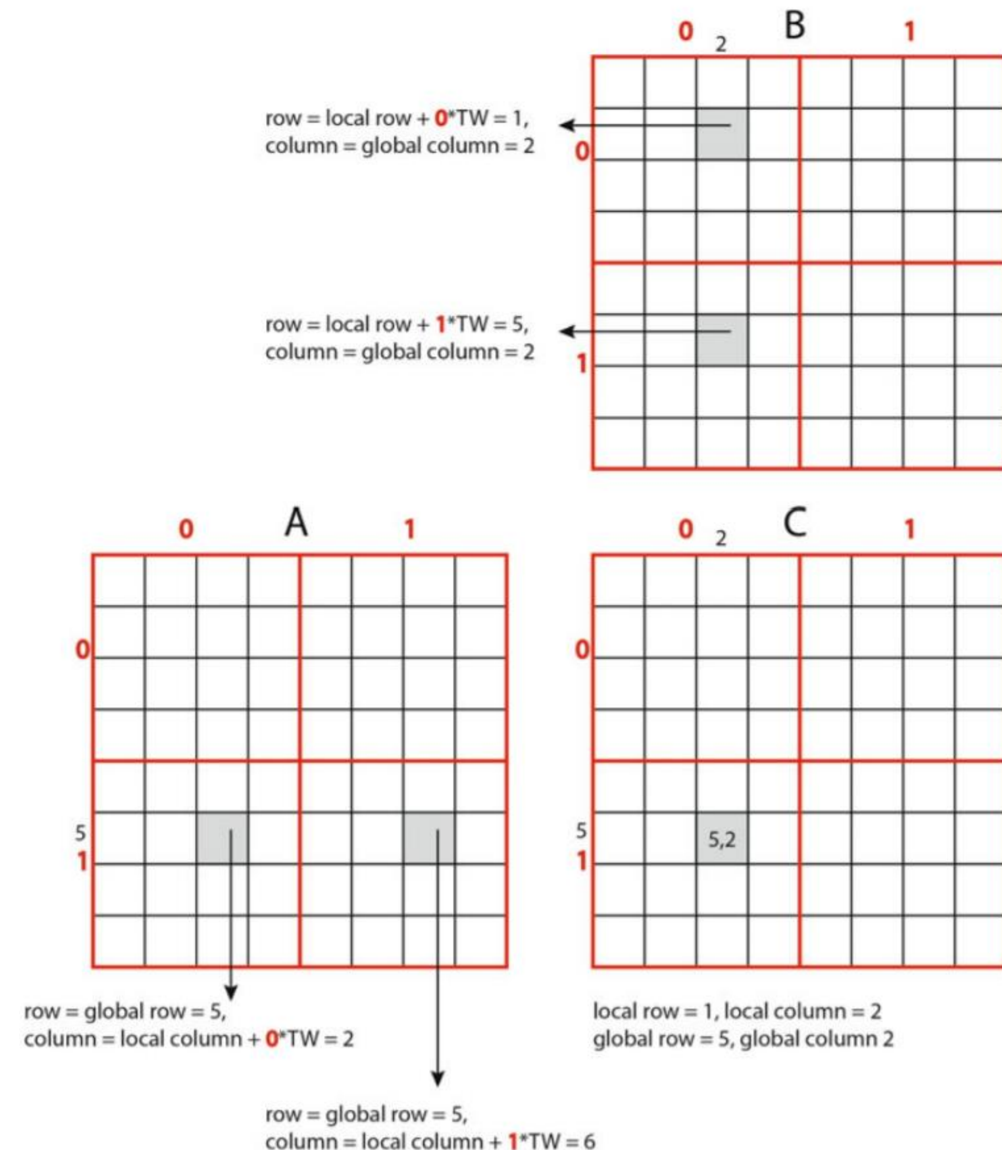
- Partição (*tile*) de tamanho 3 por 3.
- Uma partição ou submatriz de C:
  - Soma dos produtos de partições:
    - Uma partição de A.
    - Uma partição de B.
  - O processo se repete enquanto existirem partições:
    - N / (tamanho da partição)

Multiplicação de Matrizes – Memória Local

- Uma vez que cada *work-group* executa o produto parcial de duas partições:
  1. Vamos fazer com que cada *work-item* **carregue** uma parte da partição para a memória local.
  2. Uma barreira sincroniza o *work-group* para garantir que os dados estejam na memória local.
  3. A multiplicação é feita.
- A questão é definir qual elemento o *work-item* deve carregar.

Multiplicação de Matrizes – Memória Local

- Dada uma partição 4x4 em matrizes 8x8.
- Considere o elemento [5, 2] da matriz C:
  - Está na partição (0, 1) de C.
  - Tem coordenadas globais [5, 2] e coordenadas locais [1, 2].
  - Para seu cálculo no primeiro produto parcial:
    - Segunda linha da partição (0, 1) de A.
    - Terceira coluna da partição (0, 0) de B.
  - Para seu cálculo do segundo produto parcial:
    - Segunda linha da partição (1, 1) de A.
    - Terceira coluna da partição (1, 0) de B
- Existe um mapeamento da posição do *work-item* no *work-group* para a posição do elemento que irá carregar na partição vigente.



Multiplicação de Matrizes – Memória Local

- Carga colaborativa.
- O *work-item* na i-ésima linha e na j-ésima coluna da partição realiza duas cargas:
  - O elemento (*i, j*) da partição correspondente em A.
  - O elemento (*i, j*) da partição correspondente em B.
- A partição é carregada por completo na memória local.
- Enquanto estiver na memória, o acesso será mais rápido.

FIM