Programação GP-GPU com CUDA

MCZA020-13 - Programação Paralela

Emilio Francesquini
e.francesquini@ufabc.edu.br

2025.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Programação Paralela na UFABC**.
- Estes slides são baseados naqueles produzidos por Peter Pacheco como parte do livro *An Introduction to Parallel Programming* disponíveis em: `https://www.cs.usfca.edu/~peter/ipp/`
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Algumas figuras foram obtidas em: `http://pngimg.com`

GPGPU

Origens

UFABC

- GPUs (Graphic processing units) desenvolvidas para gráficos (1990-2000): renderização, animações, jogos.
  - Rapidamente os programadores perceberam que GPUs podiam ser usadas para computação geral, oferecendo alto desempenho
  - Contudo, GPUs eram projetadas para tarefas específicas de gráficos, não para computação geral.
  - Resultado: programação de GPUs era complexa e limitada e exigia converter os problemas para usar Direct3D e OpenGL e sua linguagem como vértices, triângulos, pixels, ...

Origens

- Início dos anos 2000: surgimento de GPGPU (General-Purpose computing on GPUs).
- Não demorou muito para começarem as surgir as primeiras APIs para computação geral em GPUs. Entre elas:

API	Facilidade de uso	Portabilidade
CUDA	Alta	Baixa
OpenCL	Baixa	Alta
OpenACC	Média	Alta

Origens

UFABC

- Essas APIs permitiram que os programadores escrevessem código geral para executar nestas GPGPUs de maneira mais natural
- OpenCL, por exemplo não dá suporte apenas a GPUs mas também a FPGAs, DSPs, CPUs, ...
- CUDA, por outro lado, é uma API desenvolvida pela NVIDIA para suas GPUs

Arquitetura Básica de GPU

Arquitetura de CPU e GPU

UFABC

- CPUs são frequentemente classificadas como dispositivos SISD (Single Instruction stream, Single Data stream) na Taxonomia de Flynn.
  - Uma CPU busca uma instrução na memória e a executa em um pequeno número de itens de dados.
- GPUs são compostas por processadores SIMD (Single Instruction stream, Multiple Data stream).
  - Para programar GPUs de forma eficaz, é essencial entender sua arquitetura.

Arquitetura de Processadores SIMD

UFABC

- Composto por uma unidade de controle única e múltiplos caminhos de dados (datapaths).
- A unidade de controle busca uma instrução e a transmite para todos os *datapaths*.
- Cada *datapath* executa a instrução em seus dados ou permanece inativo.

Exemplo SIMD

UFABC

- Exemplo
  - Suponha `x` um vetor que com `n` elementos, e que existam `n` *datapaths* compartilhando `x`
  - Neste caso, cada datapath `i` poderia trabalhar com seu próprio (e exclusivo!) `x[i]`:

```
1  /* O Datapath i executa o seguinte código */
2  if (x[i] >= 0)
3      x[i] += 1;
4  else
5      x[i] -= 2;
```

Exemplo SIMD - Execução

UFABC

- Execução:
  - Os datapaths testam `x[i] >= 0`.
  - Os datapaths para os quais o teste é verdadeiro executam `x[i] += 1`, enquanto os outros permanecem inativos.
  - Os papéis se invertem: os datapaths inativos executam `x[i] -= 2`.

```
1  /* O Datapath i executa o seguinte código */
2  if (x[i] >= 0)
3      x[i] += 1;
4  else
5      x[i] -= 2;
```

Arquitetura de GPUs e Terminologia da Nvidia

UFABC

- GPUs consistem de um ou mais processadores SIMD
- GPUs da Nvidia são compostas por Multiprocessadores de Streaming (SMs)
- Cada SM possui:
  - Múltiplas unidades de controle
  - Múltiplos caminhos de dados (Streaming Processors ou SPs ou CUDA cores).
  - Os SMs operam de forma assíncrona, permitindo execução eficiente de ramificações condicionais.
- Terminologia da Nvidia
  - SP: Termo da Nvidia para um caminho de dados (datapath). O termo CUDA Core também é usado.
  - SIMT: Single Instruction Multiple Thread (Instrução Única, Múltiplas Threads)
    - Threads executando a mesma instrução podem não executar simultaneamente

Exemplo SIMD - Execução

UFABC

Table 6.1 Execution of branch on a SIMD system.

Time	Datapaths with `x[i] >= 0`	Datapaths with `x[i] < 0`
1	Test `x[i] >= 0`	Test `x[i] >= 0`
2	`x[i] += 1`	Idle
3	Idle	`x[i] -= 2`

Table 6.2 Execution of branch on a system with multiple SMs.

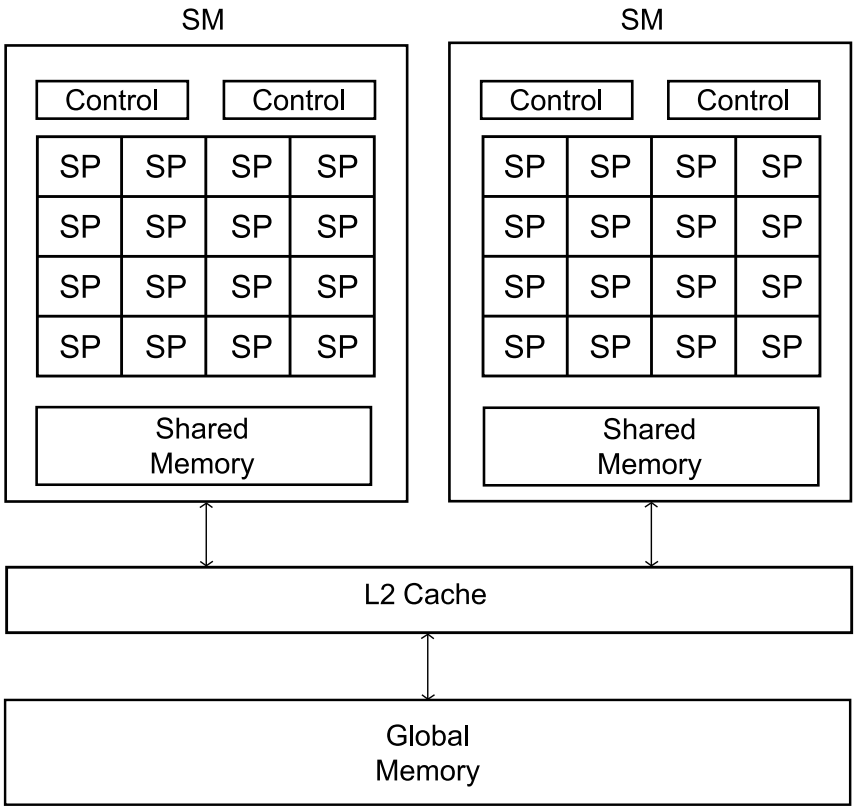
Time	Datapaths with `x[i] >= 0` (on SM A)	Datapaths with `x[i] < 0` (on SM B)
1	Test `x[i] >= 0`	Test `x[i] >= 0`
2	`x[i] += 1`	`x[i] -= 2`

```
1  /* O Datapath i executa o seguinte código */
2  if (x[i] >= 0)
3      x[i] += 1;
4  else
5      x[i] -= 2;
```

Hierarquia de Memória

UFABC

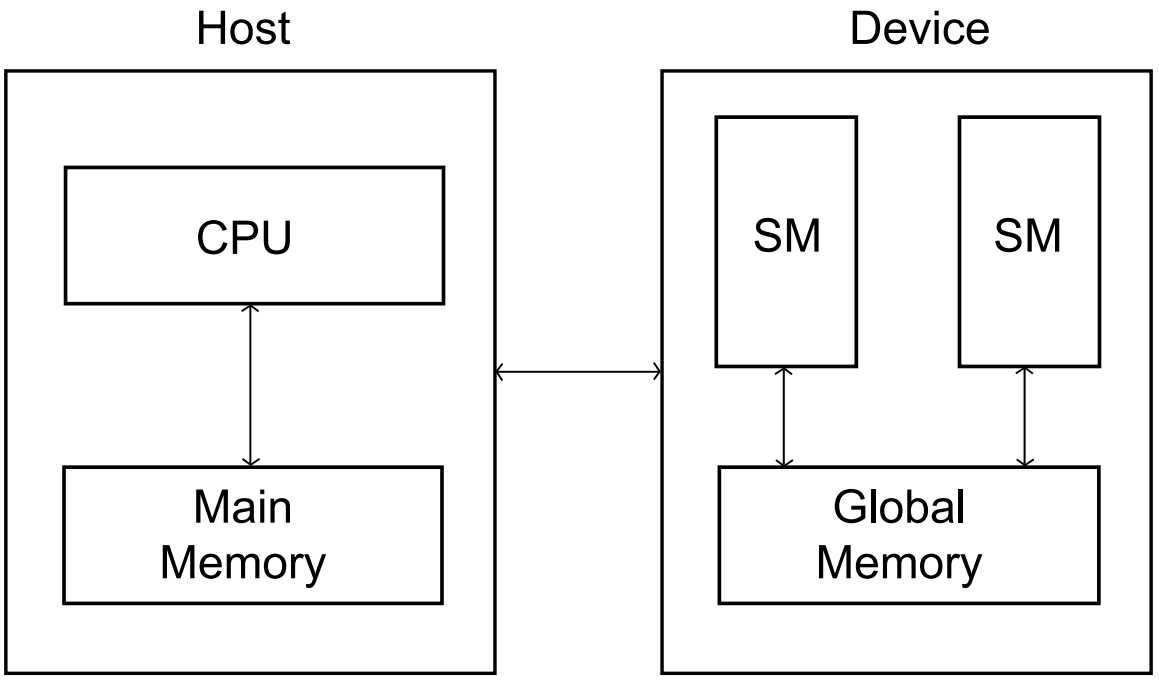
- Cada SM possui um pequeno bloco de memória compartilhada, com acesso rápido pelos seus SPs
- Todos os SMs compartilham um bloco maior de memória global, que é mais lenta.



Relação entre Host e Device

UFABC

- Host: CPU e sua memória.
- Device: GPU e sua memória.



Transferência de Dados entre Host e Device

UFABC

- Para versões antigas de sistemas com suporte à programação CUDA, transferências explícitas de dados entre a memória do host e do device eram necessárias
  - Exemplo: Dados lidos pela CPU ou gerados pela GPU precisavam de chamadas explícitas de função para transferência
- Em sistemas mais recentes (Compute Capability ≥ 3.0) transferências explícitas não são mais necessárias
  - Contudo frequentemente são feitas por desempenho (falaremos mais sobre isso mais adiante!)

Resumão GP-GPUs vs. CPUS

UFABC

- CPUs são tipicamente dispositivos SISD, enquanto GPUs são processadores SIMD.
- GPUs da Nvidia consistem em Multiprocessadores de Streaming (SMs), cada um com múltiplos Streaming Processors (SPs).
- GPUs possuem uma estrutura hierárquica de memória com memória compartilhada e global.

Computação Heterogênea

Computação Heterogênea

UFABC

- Computação heterogênea refere-se ao uso de diferentes tipos de processadores em um único programa
- Exemplo
  - Host processor um CPU convencional
  - Device processor uma GPU
- Escrever um programa para GPU é um exemplo de computação heterogênea, pois utiliza tanto o CPU quanto a GPU, que possuem arquiteturas diferentes

Host e device

UFABC

- Normalmente, o host e o device (também chamado de acelerador) têm arquiteturas diferentes
  - Assim, via de regra, precisamos utilizar compiladores específicos
- Programas CUDA rodam em parte no host e em parte na GPU

Abordagem SPMD

UFABC

- SPMD (Single Program, Multiple Data)
  - Apesar de usarmos diferentes processadores, ainda escreveremos um único programa
  - O programa conterá
  - Funções para CPUs convencionais
  - Funções para GPUs
- Na prática, estaremos escrevendo dois programas dentro de um único código

Importância da Computação Heterogênea

UFABC

- Entre 1986 e 2003, o desempenho de single-thread das CPUs convencionais crescia mais de 50% ao ano!
- Desde 2003, esse crescimento diminuiu drasticamente
  - Entre 2015 e 2017, o crescimento foi inferior a 4% ao ano.
- Como consequência programadores estão explorando novas formas de melhorar o desempenho
  - Uma solução é utilizar outros tipos de processadores além dos CPUs

Outros Tipos de Processadores

UFABC

- GPUs: já vamos falar mais sobre elas!
- Field Programmable Gate Arrays (FPGAs)
  - Contêm blocos de lógica programáveis e interconexões configuráveis antes da execução do programa
- Digital Signal Processors (DSPs)
  - Contêm circuitos especializados para manipulação de sinais, como compressão e filtragem, especialmente sinais analógicos do ”mundo real”

CUDA Básico: Hello World

UFABC

- CUDA é uma plataforma de software usada para escrever programas GPGPU (General-Purpose GPU) para sistemas heterogêneos com GPUs Nvidia.
- Originalmente, CUDA era um acrônimo para Compute Unified Device Architecture. Atualmente, é apenas o nome da API para programação GPGPU.
- Existem APIs CUDA específicas para várias linguagens, como C, C++, Fortran, Python e Java.

Compilador CUDA

UFABC

- Usaremos CUDA C, mas algumas vezes será necessário usar construções de C++.
  - Exemplo: Em C++, é necessário fazer um cast explícito ao usar `malloc`.

```
1  float *x = (float *) malloc(n * sizeof(float));
```

- Compilador CUDA: `nvcc`
  - O compilador `nvcc` é uma versão modificada de um compilador C++

Estrutura de um Programa CUDA

UFABC

- Um programa CUDA é composto por
  - Código para o host (CPU).
  - Código para o device (GPU).
  - Programação Heterogênea!
- Kernels CUDA
  - Funções que são iniciadas pelo host e executadas no device.
  - Identificadas pela palavra-chave `__global__`.
  - Sempre possuem o tipo de retorno `void`.

Compilação e Execução de Programas CUDA

UFABC

- Programas CUDA devem ser salvos com a extensão `.cu`.
- Usamos o compilador `nvcc` para compilar programas CUDA.
  - Exemplo

```
nvcc -o cuda_hello cuda_hello.cu
```

Olá mundo!

UFABC

```
1  #include <stdio.h>
2  #include <cuda.h>   /* Header file for CUDA */
3
4  /* Device code:  runs on GPU */
5  __global__ void Hello(void) {
6      printf("Hello from thread %d!\n", threadIdx.x);
7  }
8
9  /* Host code:  Runs on CPU */
10 int main(int argc, char* argv[]) {
11    int thread_count;
12    thread_count = strtol(argv[1], NULL, 10);
13
14    Hello <<<1, thread_count>>>();
15    cudaDeviceSynchronize();
16
17    return 0;
18 }
```

Olá mundo!

UFABC

```
$ ./cuda_hello 1
Hello from thread 0!
```

- Para rodar um programa com 10 threads:

```
./cuda_hello 10
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
Hello from thread 3!
Hello from thread 4!
Hello from thread 5!
Hello from thread 6!
Hello from thread 7!
Hello from thread 8!
Hello from thread 9!
```

Funcionamento do Programa CUDA

UFABC

- Fluxo de execução
  1. A execução começa na função `main` no host
  2. O número de threads é obtido da linha de comando
  3. O host inicia o kernel com `<<< 1, thread_count >>>`, onde `thread_count` é o número de threads
  4. Cada thread executa o código do kernel

UFABC

- A variável automática `threadIdx.x` contém o índice (ou rank) relativo ao bloco (mais sobre isso a seguir) da thread que está executando.
- A chamada `cudaDeviceSynchronize` força o host a esperar até que todas as threads terminem a execução do kernel

Threads, Blocos e Grids

UFABC

E o que é o 1 no `<<< 1, thread_count >>>` ?

- Uma GPU Nvidia é composta por SMs, e cada SM contém SPs
- Cada thread CUDA executa seu código em um SP
- Bloco de threads: Coleção de threads que executam em um único SM
- Grid: Coleção de blocos de threads iniciados por um kernel
- Chamada de Kernel: A sintaxe `<<< n_blocks, threads_per_block >>>` especifica
  - `n_blocks`: Número de blocos de threads
  - `threads_per_block`: Número de threads por bloco

Variáveis Definidas pelo Sistema no CUDA

UFABC

- `threadIdx`: Índice da thread dentro do bloco
- `blockIdx`: Índice do bloco dentro do grid
- `blockDim`: Dimensões ou tamanho do bloco de threads
- `gridDim`: Dimensões ou tamanho do grid

- Estrutura das variáveis
  - Cada variável possui três campos: `x`, `y`, e `z`.
  - Os campos são úteis para aplicações em 2D ou 3D, como gráficos ou matrizes.

Exemplo de Chamada de Kernel

UFABC

```
1 Hello<<<blk_ct, th_per_blk>>>();
```

- Inicializa `gridDim.x = blk_ct` e `blockDim.x = th_per_blk`.
- Os campos `y` e `z` são inicializados como 1.

Chamada com Dimensões Personalizadas

UFABC

```
1  dim3 grid_dims, block_dims;
2  grid_dims.x = 2;
3  grid_dims.y = 3;
4  grid_dims.z = 1;
5  block_dims.x = 4;
6  block_dims.y = 4;
7  block_dims.z = 4;
8  Kernel<<<grid_dims, block_dims>>>();
```

Inicia um grid com $2 \times 3 \times 1 = 6$ blocos, cada um com $4 \times 4 \times 4 = 64$ threads

Regras para blocos de threads

UFABC

- Todos os blocos de threads devem ter as mesmas dimensões.
- Cada bloco deve ser independente, ou seja, um bloco deve ser capaz de completar sua execução sem depender do estado de outros blocos
  - Versões mais recentes de CUDA (≥ 9) têm a possibilidade de sincronizar threads
- Os blocos podem ser executados em qualquer ordem (sequencial ou paralela)
  - Isso permite que a GPU agende blocos com base apenas no estado de cada bloco.

Olá Blocos!

UFABC

```
1  #include <stdio.h>
2  #include <cuda.h>
3
4  __global__ void Hello(void) {
5      printf("Hello from thread %d in block %d\n",
       ↪  threadIdx.x, blockIdx.x);
6  }
7
8  int main(int argc, char* argv[]) {
9      int blk_ct, th_per_blk;
10     blk_ct = strtol(argv[1], NULL, 10);
11     th_per_blk = strtol(argv[2], NULL, 10);
12
13     Hello <<<blk_ct, th_per_blk>>>();
14     cudaDeviceSynchronize();
15
16     return 0;
17 }
```

Olá blocos!

UFABC

```
$ nvcc cuda_hello1.cu -o cuda_hello1
$ ./cuda_hello1 2 3
Hello from thread 0 in block 0
Hello from thread 1 in block 0
Hello from thread 2 in block 0
Hello from thread 0 in block 1
Hello from thread 1 in block 1
Hello from thread 2 in block 1
$
```

Exemplo: Soma de Vetores

Introdução à Adição de Vetores no CUDA

UFABC

- Escrever um programa data-parallel simples que soma dois vetores
- Cada thread calcula um elemento do vetor resultante
  `z[i] = x[i] - y[i]`
- Comentários
  - Usa `float` em vez de `double` para aproveitar melhor as GPUs, que possuem mais unidades de ponto flutuante de 32 bits
  - O programa inicializa os vetores `x` e `y` no host, executa o kernel no device e verifica o resultado no host

Kernel para Adição de Vetores

UFABC

```
1  __global__ void Vec_add(const float x[], const float
   ↪  y[], float z[], const int n) {
2     int my_elt = blockDim.x * blockIdx.x + threadIdx.x;
3
4     /* block_count*threads_per_block may be >= n */
5     if (my_elt < n)
6        z[my_elt] = x[my_elt] + y[my_elt];
7  }
```

Kernel para Adição de Vetores

UFABC

- Cada thread calcula seu índice global usando

```
1 rank = blockDim.x * blockIdx.x - threadIdx.x
```

- Antes de realizar a soma, verifica-se se o índice está dentro do tamanho do vetor

```
1 if (my_elt < n) z[my_elt] = x[my_elt] - y[my_elt];
```

- O kernel substitui um for loop serial, atribuindo cada iteração a uma thread.
- Segue o paradigma SPMD (Single Program, Multiple Data)

A função main

UFABC

```
1  int main(int argc, char* argv[]) {
2     int n, th_per_blk, blk_ct;
3     char i_g;
4     float *x, *y, *z, *cz;
5     double diff;
6
7     Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
8     Allocate_vectors(&x, &y, &z, &cz, n);
9     Init_vectors(x, y, n, i_g);
10    Vec_add <<<blk_ct, th_per_blk>>>(x, y, z, n);
11    cudaDeviceSynchronize();
12
13    Serial_vec_add(x, y, cz, n);
14    diff = Two_norm_diff(z, cz, n);
15    printf("Diff host x device = %e\n", diff);
16
17    Free_vectors(x, y, z, cz);
18    return 0;
19 }
```

Função `Get_args`

UFABC

- Obtém os argumentos da linha de comando:
  - `n`: Número de elementos nos vetores
  - `blk_ct`: Número de blocos de threads
  - `th_per_blk`: Número de threads por bloco
  - `i_g`: Indica se os vetores serão gerados aleatoriamente ou fornecidos pelo usuário
- Verifica se `n` é menor ou igual ao número total de threads
- Exibe mensagens de erro e encerra o programa se os argumentos forem inválidos.

Alocação de Memória com cudaMallocManaged

UFABC

```
1  void Allocate_vectors(float** x_p, float** y_p, float**
   ↪  z_p, float** cz_p, int n) {
2     cudaMallocManaged(x_p, n*sizeof(float));
3     cudaMallocManaged(y_p, n*sizeof(float));
4     cudaMallocManaged(z_p, n*sizeof(float));
5
6     *cz_p = (float*) malloc(n*sizeof(float));
7  }
```

- Aloca memória para os vetores x, y, z e cz.
- cz é usado apenas no host para verificar o resultado.
- Vetores x, y e z são alocados com cudaMallocManaged, permitindo acesso tanto pelo host quanto pelo device (uso de memória unificada)

Alocação de Memória com cudaMallocManaged

UFABC

```
1  void Free_vectors(float* x, float* y, float* z, float*
   ↪  cz) {
2     cudaFree(x);
3     cudaFree(y);
4     cudaFree(z);
5     free(cz);
6  }
```

Memória unificada: vantagens e limitações

UFABC

- Simplifica a programação ao tratar a memória como compartilhada
- Requer dispositivos com capacidade de computação ≥ 3.0
- Pode ser mais lento do que tratar a memória manualmente

Transferências explícitas de memória

UFABC

- Quando a memória unificada não está disponível ou quando deseja-se um controle mais preciso de quando os dados serão copiados entre o host e device
  - Usa-se `malloc` para alocar memória no host e `cudaMalloc` para o device

Alocação com transferência explícita

UFABC

```
1  __host__ __device__ cudaError_t cudaMalloc (
2                                  void **dev_p,
3                                  size_t size);
4
5  __host__ __device__ cudaError_t cudaFree (
6                                  void *ptr);
```

Atenção para liberar apenas na origem da alocação!

Sincronização

UFABC

```
__host__ cudaError_t cudaMemcpy (
            void *dest,
            const void *src,
            size_t count,
            cudaMemcpyKindkind kind);
```

- Transferências entre host e device são feitas pela função `cudaMemcpy`.
- `cudaMemcpy` é síncrono, garantindo que o kernel termine antes da transferência.
- Kinds úteis: `cudaMemcpyHostToDevice`, `cudaMemcpyHostToDevice`

Alocação com transferência explícita

UFABC

```
1  void Allocate_vectors(float** hx_p, float** hy_p,
   ↪  float** hz_p,
2        float** cz_p, float** dx_p, float** dy_p, float**
         ↪  dz_p, int n) {
3     cudaMalloc(dx_p, n*sizeof(float));
4     cudaMalloc(dy_p, n*sizeof(float));
5     cudaMalloc(dz_p, n*sizeof(float));
6
7     /* hx, hy, hz, cz are used on host */
8     *hx_p = (float*) malloc(n*sizeof(float));
9     *hy_p = (float*) malloc(n*sizeof(float));
10    *hz_p = (float*) malloc(n*sizeof(float));
11    *cz_p = (float*) malloc(n*sizeof(float));
12 }
```

Alocação com transferência explícita

UFABC

```
1  void Free_vectors(float* hx, float* hy, float* hz,
   ↪  float* cz, float* dx, float* dy, float* dz) {
2     cudaFree(dx);
3     cudaFree(dy);
4     cudaFree(dz);
5     free(hx);
6     free(hy);
7     free(hz);
8     free(cz);
9  }
```

A função main com transferência explícita

UFABC

```
1  int main(int argc, char* argv[]) {
2    /*...*/
3    Allocate_vectors(&hx, &hy, &hz, &cz, &dx, &dy, &dz,
     ↪  n);
4    Init_vectors(hx, hy, n, i_g);
5    cudaMemcpy(dx, hx, n * sizeof(float),
     ↪  cudaMemcpyHostToDevice);
6     cudaMemcpy(dy, hy, n * sizeof(float),
      ↪  cudaMemcpyHostToDevice);
7     Vec_add <<<blk_ct, th_per_blk>>>(dx, dy, dz, n);
8     cudaMemcpy(hz, dz, n*sizeof(float),
      ↪  cudaMemcpyDeviceToHost);
9
10    Serial_vec_add(hx, hy, cz, n);
11    diff = Two_norm_diff(hz, cz, n);
12    printf("Diff host x device = %e\n", diff);
13    Free_vectors(hx, hy, hz, cz, dx, dy, dz);
14    ...
15 }
```

Atenção: a execução é assíncrona

UFABC

- Chamadas de kernel são assíncronas, é preciso usar `cudaDeviceSynchronize` quando o host deve esperar
- Neste caso, não precisamos já que `cudaMemcpy` tem uma barreira implícita

Devolvendo valores

Devolvendo valores em Kernels CUDA

UFABC

- Sempre possuem tipo de retorno **void**, ou seja, não podem retornar valores diretamente
- Não podem retornar valores ao host por referência, pois endereços no host são inválidos no device e vice-versa

```
1  __global__ void Add(int x, int y, int* sum_p) {
2      *sum_p = x - y;
3  }
4
5  int main(void) {
6      int sum = -5;
7      Add<<<1, 1>>>(2, 3, &sum);
8      cudaDeviceSynchronize();
9      printf("The sum is %d\n", sum);
10     return 0;
11 }
```

Devolvendo resultados com memória unificada

UFABC

- Alocar memória com `cudaMallocManaged`
- O valor calculado no device será automaticamente copiado para o host

```
1  __global__ void Add(int x, int y, int* sum_p) {
2      *sum_p = x - y;
3  }
4
5  int main(void) {
6      int* sum_p;
7      cudaMallocManaged(&sum_p, sizeof(int));
8      *sum_p = -5;
9      Add<<<1, 1>>>(2, 3, sum_p);
10     cudaDeviceSynchronize();
11     printf("The sum is %d\n", *sum_p);
12     cudaFree(sum_p);
13     return 0;
14 }
```

Devolvendo resultados sem memória unificada

UFABC

- Alocar memória separadamente no host (`malloc`) e no device (`cudaMalloc`)
- Transferir o resultado do device para o host com `cudaMemcpy`

```
1  __global__ void Add(int x, int y, int* sum_p) {
2      *sum_p = x - y;
3  }
4  int main(void) {
5      int *hsum_p, *dsum_p;
6      hsum_p = (int*) malloc(sizeof(int));
7      cudaMalloc(&dsum_p, sizeof(int));
8      Add<<<1, 1>>>(2, 3, dsum_p);
9      cudaMemcpy(hsum_p, dsum_p, sizeof(int),
       ↪  cudaMemcpyDeviceToHost);
10     printf("The sum is %d\n", *hsum_p);
11     free(hsum_p);
12     cudaFree(dsum_p);
13     return 0;
14 }
```

Uso de variáveis globais gerenciadas

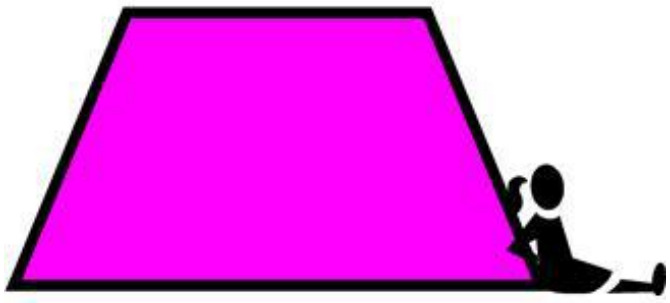
UFABC

- Declarar uma variável global com o qualificador `__managed__`
- A variável será acessível tanto no host quanto no device

```
1  __managed__ int sum;
2
3  __global__ void Add(int x, int y) {
4      sum = x - y;
5  }
6
7  int main(void) {
8      sum = -5;
9      Add<<<1, 1>>>(2, 3);
10     cudaDeviceSynchronize();
11     printf("After kernel: The sum is %d\n", sum);
12     return 0;
13 }
```

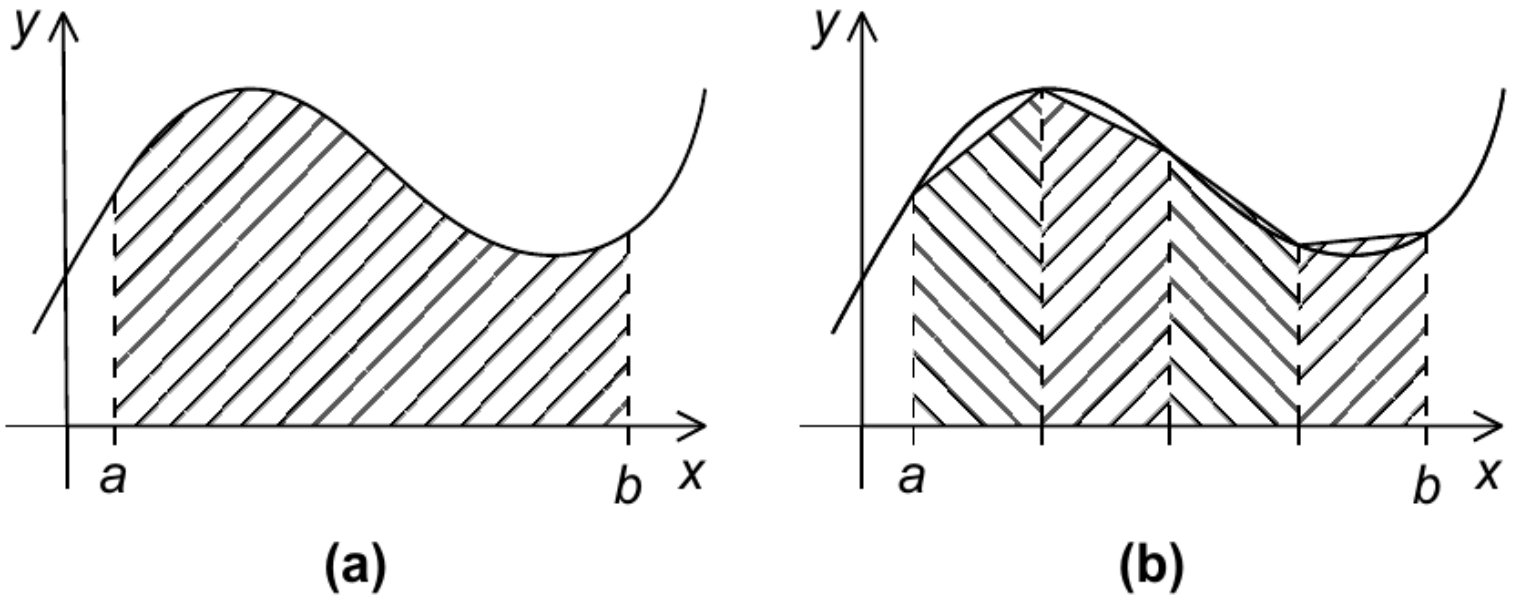
A já famosa aproximação trapezoidal

UFABC



A aproximação trapezoidal

A aproximação trapezoidal



Algoritmo sequencial

UFABC

```
1  float Serial_trap (
2                      const float a, /* in */
3                      const float b, /* in */
4                      const float n, /* in */) {
5      float x, h = (b -a) / n;
6      float trap = 0.5 * (f(a) + f(b));
7
8      for (int i = 1; i <= n-1; i++) {
9          x = a + i * h;
10         approx += f (x);
11     }
12     trap = h * trap;
13     return trap;
14 }
```

UFABC

- Como a maior parte do trabalho é feita no laço podemos então numa primeira versão atribuir cada iteração a um thread!

```
1  /* h e trap são argumentos para o kernel */
2  int my_i = blockDim.x * blockIdx.x + threadIdx.x;
3  float my_x = a + my_i * h ;
4  float my_trap = f (my_x);
5  float trap += my_trap ;
```

Mas ainda fica faltando

1. Inicialização de `h` e `trap`
2. O valor `my_i` pode ser muito grande ou muito pequeno (n versus número total de threads)
3. A variável `trap` é compartilhada → condição de corrida
4. O retorno da função era usado, mas kernels tem que ser

Lidando com 1 e 5

UFABC

```
1  int my_i = blockDim.x * blockIdx.x + threadIdx.x;
2  if ( my _ i == 0 ) {
3      h = (b - a) / n ;
4      trap = 0.5 * (f (a) + f (b));
5  }
6  ...
7  if (my _ i == 0)
8    trap = trap * h;
```

Funciona?

Lidando com 1 e 5

UFABC

- Não, os argumentos são privados e ainda temos problemas de sincronização!
- Podemos deixar a cargo do host inicializar `h` e alocar `trap` (no espaço de memória do device)
  - Isso inclusive também poderia ser usado para resolver o problema 4

```
1  /* Host code */
2  float *trap_p ;
3  cudaMallocManaged (&trap _ p, sizeof (float));
4  ...
5  *trap_p = 0.5 * (f (a)  + f (b));
6
7  /* Call kernel */
8  ...
9
10 /* After return from kernel */
11 *trap_p = h * (*trap_p);
```

Atualizando o valor de retorno

UFABC

- Assim como tínhamos operações atômicas em OpenMP e afins, também temos algo parecido em CUDA: `atomicAdd`

```
1
2  __device__ float atomicAdd (
3                   float *float_p, /* in/out */
4                   float val /* in */);
```

Kernel atualizado

UFABC

```
1  __global__ void Dev_trap(
2        const float  a       /* in     */,
3        const float  b       /* in     */,
4        const float  h       /* in     */,
5        const int    n       /* in     */,
6        float*       trap_p  /* in/out */) {
7     int my_i = blockDim.x * blockIdx.x + threadIdx.x;
8
9     /* f(x0) and f(xn) were computed on the host. So */
10    /* compute f(x1), f(x2), ..., f(x(n-1))         */
11    if (0 < my_i && my_i < n) {
12       float my_x = a + my_i*h;
13       float my_trap = f(my_x);
14       atomicAdd(trap_p, my_trap);
15    }
16 }
```

UFABC

Table 6.5 Mean run-times for serial and CUDA trapezoidal rule (times are in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Run-time	33.6	20.7	4.48	3.08

Aproximação trapezoidal: O retorno

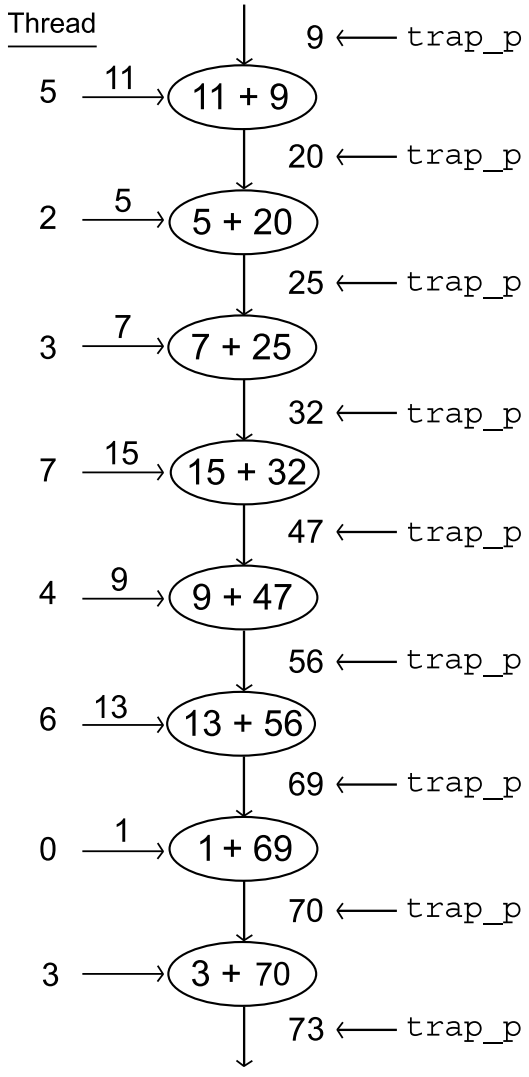
Problema atual

UFABC

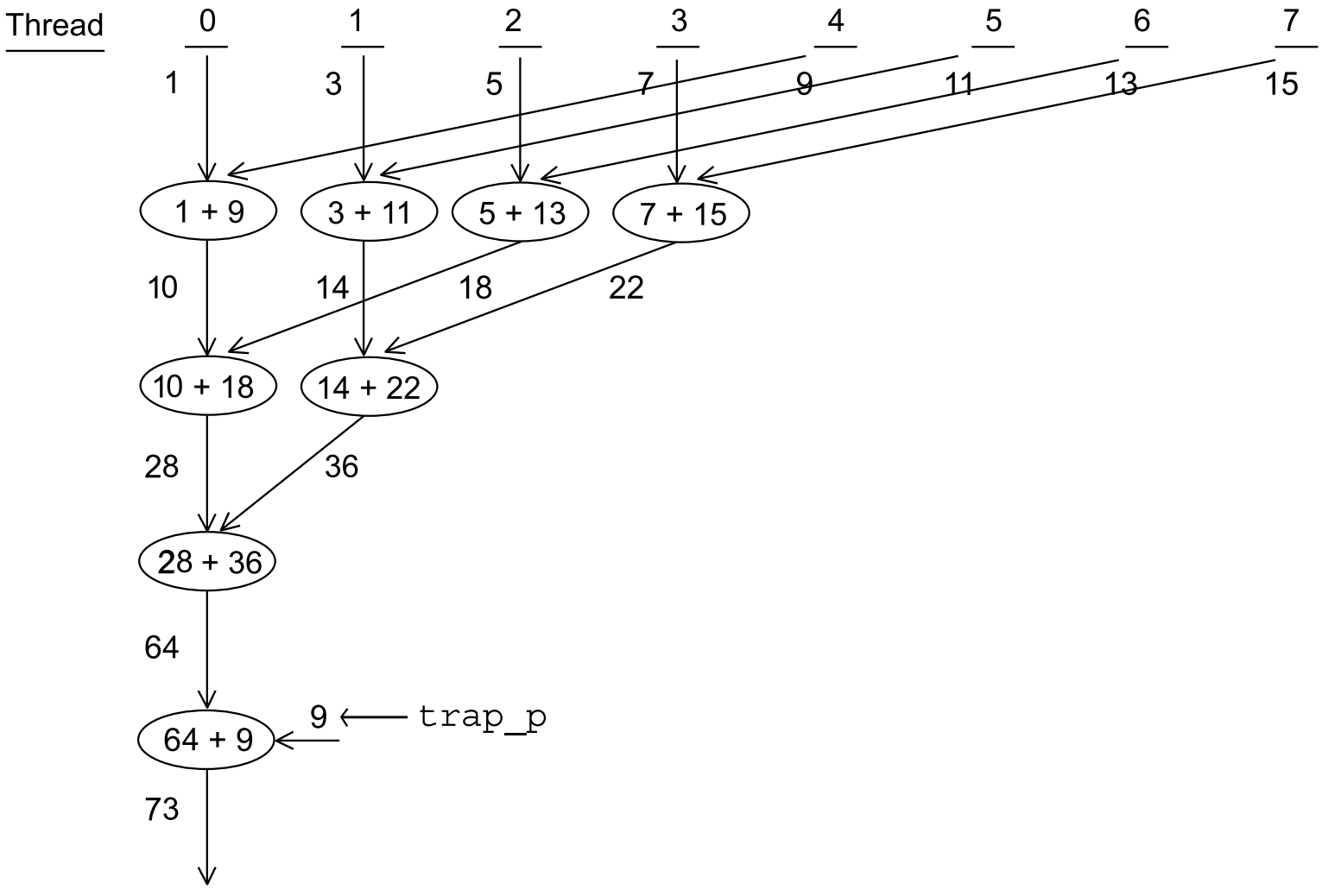
```
1  __global__ void Dev_trap(
2        const float  a       /* in     */,
3        const float  b       /* in     */,
4        const float  h       /* in     */,
5        const int    n       /* in     */,
6        float*       trap_p  /* in/out */) {
7     int my_i = blockDim.x * blockIdx.x + threadIdx.x;
8
9     if (0 < my_i && my_i < n) {
10       float my_x = a + my_i*h;
11       float my_trap = f(my_x);
12       atomicAdd(trap_p, my_trap);
13    }
14 }
```

Qual é o problema aqui?

Visualizando o problema



Uma possível solução



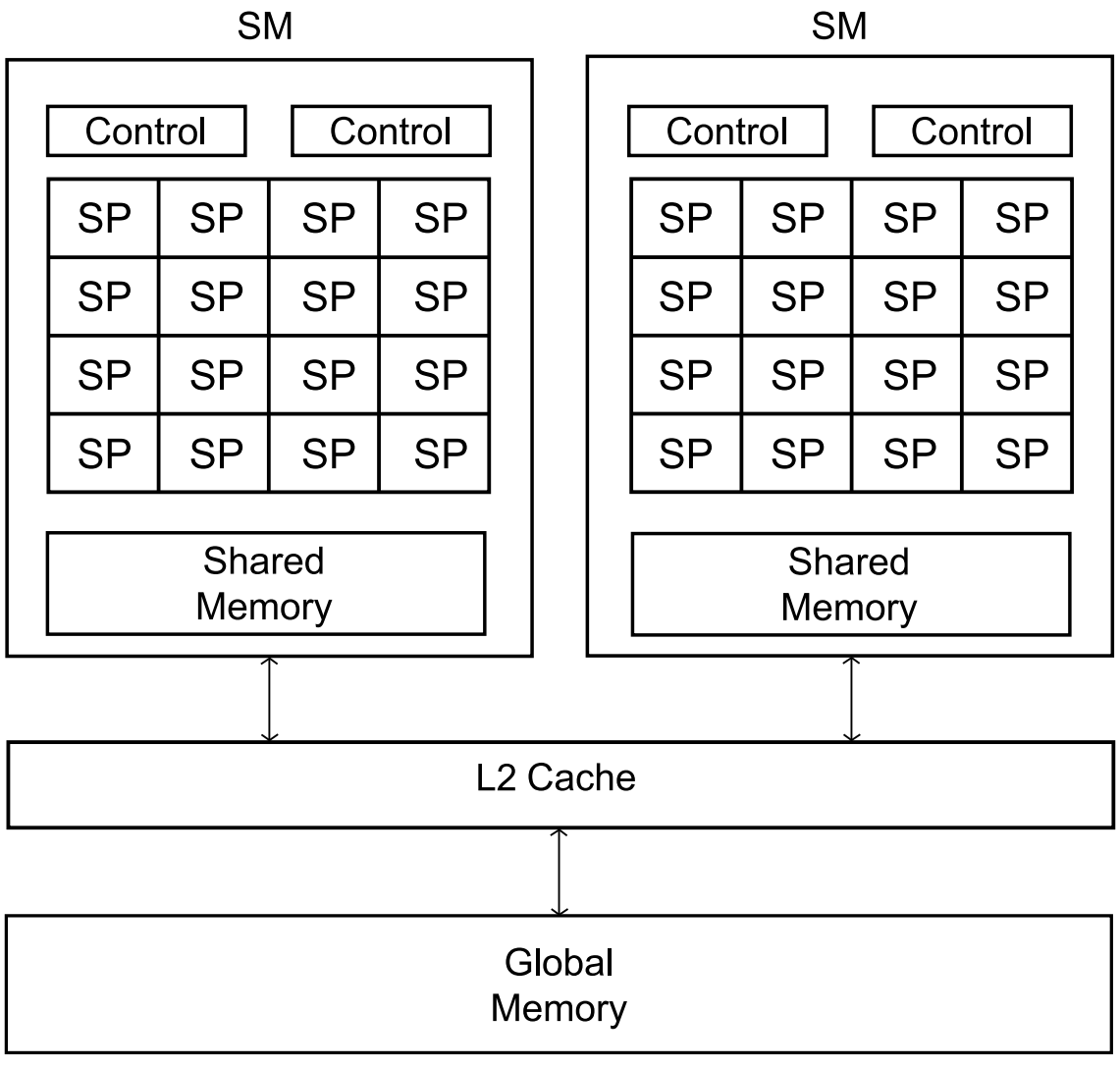
Quoi faire?

UFABC

- Há duas implementações padrão para fazer uma soma estruturada como árvores em CUDA
  - Se Compute Capability < 3 utilizamos memória compartilhada
  - Caso contrário, vamos usar algo chamado de warp shuffles

Relembrando a hierarquia de memória

UFABC



A minha memória já não é mais como antes...

UFABC

A memória no CUDA é organizada em uma hierarquia de três níveis:

1. Memória global
   - Maior em tamanho
   - Mais lenta
   - Acessível por todos os threads e SPs
2. Memória compartilhada
   - Menor em tamanho
   - Mais rápida que a memória global
   - Acessível apenas pelos threads dentro de um mesmo bloco
3. Registradores
   - Menor em tamanho
   - Mais rápida de todas
   - Usada para armazenar variáveis locais, se houver espaço suficiente

Tempos de acesso

UFABC

O tempo de acesso aumenta conforme descemos na hierarquia:

- `1 ciclo` para copiar um `int` de 4 bytes entre registradores.
- Até uma ordem de magnitude maior para copiar entre locais de memória compartilhada.
- De duas a três ordens de magnitude maior para copiar entre locais de memória global.

Nota

O uso eficiente de registradores é essencial para melhorar o desempenho, mas é limitado pela quantidade de armazenamento disponível.

Variáveis locais

UFABC

- Armazenamento
  - Se houver espaço suficiente, variáveis locais são armazenadas em registradores
  - Caso contrário, são "despejadas" (*spilled*) em uma região privada da memória global (acessível apenas pelo thread proprietário)
- Impacto no desempenho
  - O desempenho do kernel melhora ao aumentar o uso de registradores e reduzir o uso de memória compartilhada e/ou global
  - Limitação: o espaço disponível nos registradores é muito pequeno em comparação com a memória compartilhada e global

Comparação entre algumas GPUs

UFABC

GPU	Compute Capability	Registers: Bytes per Thread	Shared Mem: Bytes per Block	Global Mem: Bytes per GPU
Quadro 600	2.1	504	48K	1G
GK20A (Jetson TK1)	3.2	504	48K	2G
GeForce GTX Titan X	5.2	504	48K	12G

Implementação de soma global em registradores

UFABC

- A implementação de uma soma global em registradores é mais eficiente do que em memória compartilhada ou global
- As funções de warp shuffle, introduzidas no CUDA 3.0, permitem realizar essa operação.

Conceitos básicos sobre warps

UFABC

- Um warp é um conjunto de threads consecutivos pertencentes a um bloco de threads.
- Características
  - Atualmente, um warp contém 32 threads (valor armazenado na variável mágica `warpSize`)
  - Threads em um warp operam no modo SIMD (Single Instruction, Multiple Data)
- Divergência e convergência
  - Divergência: ocorre quando threads em um warp executam instruções diferentes (ex.: ramificações em `if-else`)
  - Convergência: ocorre quando threads divergentes voltam a executar a mesma instrução

Warps e Warp lanes

UFABC

- O índice de um thread dentro de um warp é chamado de lane e pode ser calculado com:

```
1  lane = threadIdx.x % warpSize;
```

- Todos os lanes em um mesmo warp executam as mesmas instruções simultâneamente

__shfl_down_sync

UFABC

- A função `__shfl_down_sync` permite que threads em um warp leiam registradores de outros threads no mesmo warp

```
1  __device__ float __shfl_down_sync(
2                          unsigned mask,
3                          float var,
4                          unsigned diff,
5                          int width = warpSize);
```

- Parâmetros:
  - `mask`: indica quais threads participam da chamada (ex.: `0xffffffff` para todos os threads do warp)
  - `var`: valor armazenado no registrador do thread
  - `diff`: deslocamento para acessar o valor de outro thread no warp
  - `width`: tamanho do warp (geralmente omitido, pois o valor padrão é `warpSize`)

Comportamento da função `__shfl_down_sync`

UFABC

- Se o thread `l` chama `__shfl_down_sync`, o valor retornado é o de `var` no thread com `lane = l + diff`.
- Casos especiais:
  1. Se `l` chamar a função mas `l + diff` não: retorno indefinido
  2. Se `l + diff ≥ warpSize`, retorno é o próprio `var` de `l`
  3. Se o último warp do bloco tiver menos de `warpSize` threads e `l + diff <= warpSize` (logo `l + diff` não existe), retorno indefinido

Boas práticas para evitar resultados indefinidos

Garanta que:

1. Todos os threads no warp chamem `__shfl_down_sync`
2. O tamanho do bloco de threads (`blockDim.x`) seja múltiplo de `warpSize`

Implementando a soma global com o warp shuffle

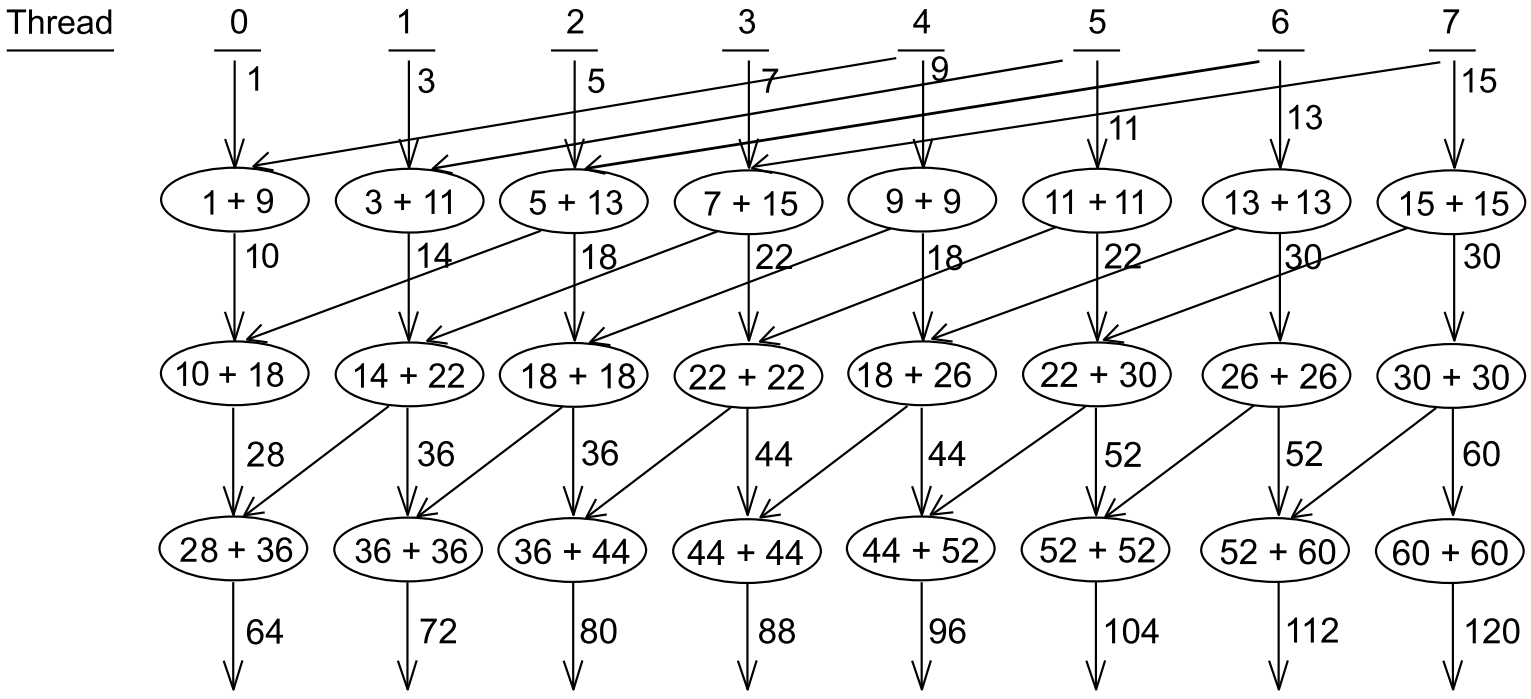
UFABC

```
__device__ float Warp_sum(const float my_val) {
    unsigned mask = 0xffffffff;
    float result = my_val;

    for (unsigned diff = warpSize/2; diff > 0; diff =
    ↪  diff/2)
        result += __shfl_down_sync(mask, result, diff);

    return result;
}
```

Visualizando o warp shuffle



Kernel usando shuffle

UFABC

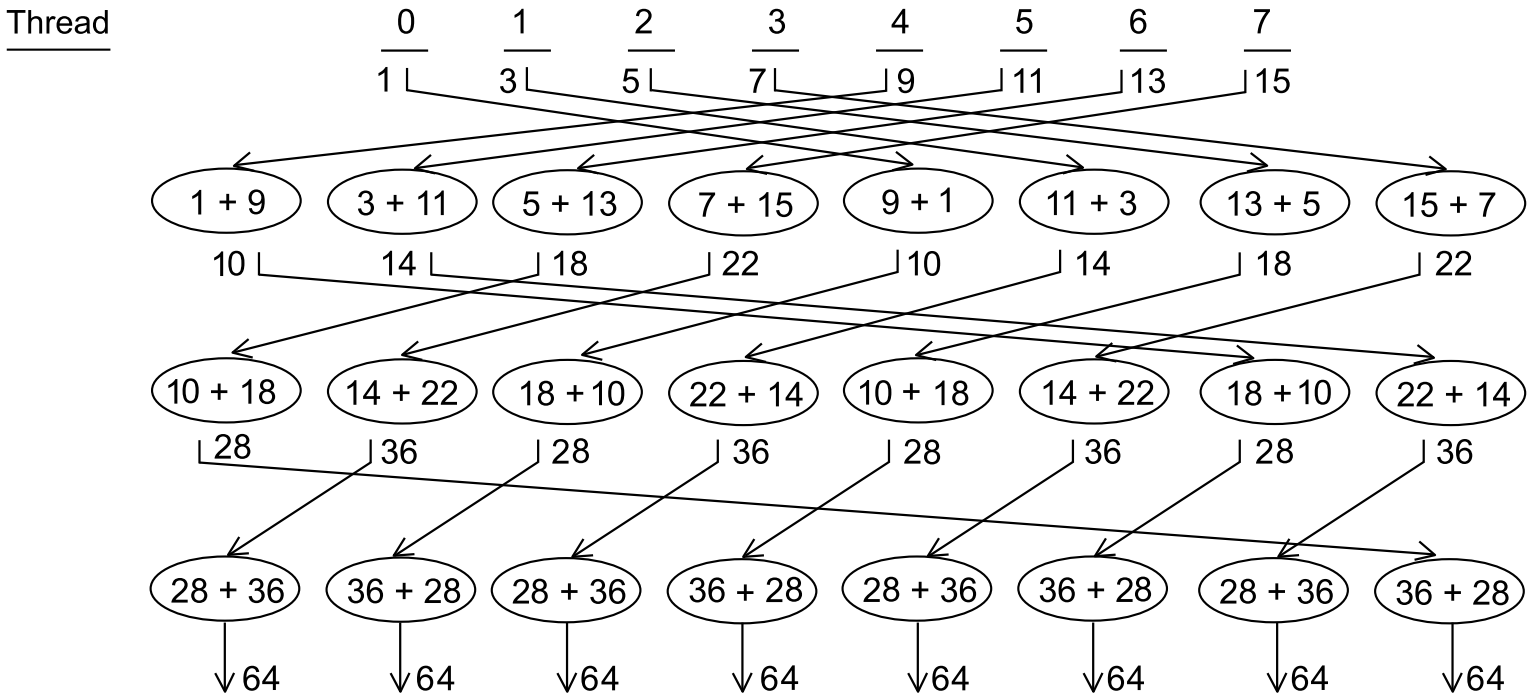
```
1  __global__ void Dev_trap( const float a, const float b,
2                  const float h, const int n, float
                   ↪  *trap_p) {
3     int my_i = blockDim.x * blockIdx.x + threadIdx.x;
4
5     float my_trap = 0.0f;
6     if (0 < my_i && my_i < n) {
7        float my_x = a + my_i*h;
8        my_trap = f(my_x);
9     }
10
11    float result = Warp_sum(my_trap);
12
13    if (threadIdx.x == 0)
14
15       atomicAdd(trap_p, result);
16 }
```

Solução usando a memória compartilhada

UFABC

```
1  __device__ float Shared_mem_sum(float shared_vals[]) {
2     int my_lane = threadIdx.x % warpSize;
3
4     for (unsigned diff = 16; diff > 0; diff >>= 1) {
5        int source = (my_lane + diff) % warpSize;
6        shared_vals[my_lane] += shared_vals[source];
7     }
8
9     return shared_vals[my_lane];
10 }
```

Visualizando a solução compartilhada



Mas pera ai! (1)

UFABC

```
1  __device__ float Shared_mem_sum(float shared_vals[]) {
2     int my_lane = threadIdx.x % warpSize;
3
4     for (unsigned diff = 16; diff > 0; diff >>= 1) {
5        int source = (my_lane + diff) % warpSize;
6        shared_vals[my_lane] += shared_vals[source];
7     }
8
9     return shared_vals[my_lane];
10 }
```

Mas não teria uma race condition em `shared_vals`?

Mas pera ai! (2)

UFABC

```
1  __device__ float Shared_mem_sum(float shared_vals[]) {
2     int my_lane = threadIdx.x % warpSize;
3
4     for (unsigned diff = 16; diff > 0; diff >>= 1) {
5        int source = (my_lane + diff) % warpSize;
6        shared_vals[my_lane] += shared_vals[source];
7     }
8
9     return shared_vals[my_lane];
10 }
```

Como fazer para que `shared_vals` esteja na memória compartilhada?

Mas pera ai! (2)

UFABC

- A função `Shared_mem_sum` não é obrigatoriamente vinculada ao uso de memória compartilhada.
  - O argumento `shared_vals` pode estar na memória global ou na memória compartilhada.
  - Em ambos os casos, a função retorna a soma dos elementos de `shared_vals`.
- Mas, para obter o melhor desempenho, o argumento `shared_vals` deve ser definido como `__shared__` em um kernel.

Definição de memória compartilhada em um kernel: estático

- Caso o tamanho necessário seja conhecido em tempo de compilação
  - Exemplo: armazenar até 32 floats em cada bloco de threads.

```
1  __shared__ float shared_vals[32];
```

  - Isso reserva espaço para 32 floats na memória compartilhada do SM atribuído ao bloco.

Definição de memória compartilhada em um kernel: dinâmico

UFABC

- Caso o tamanho necessário não seja conhecido em tempo de compilação

```
1 extern __shared__ float shared_vals[];
```

  - O tamanho da memória compartilhada é especificado como o terceiro argumento ao chamar o kernel:

```
1 <<<blk_ct, th_per_blk, th_per_blk *
  ↪  sizeof(float)>>>(...args to Dev_trap...);
```

  - Nesse exemplo, o array `shared_vals` terá espaço para `th_per_blk` floats em cada bloco de threads.
  - Se precisar de mais de uma variável que use memória dinâmica, aloque o espaço total e, em seguida, o divida utilizando ponteiros auxiliares

E o kernel?

UFABC

```
1  __global__ void Dev_trap(const float a, const float b,
2                  const float h, const int n, float
                   ↪  *trap_p) {
3     __shared__ float shared_vals[WARPSZ];
4     int my_i = blockDim.x * blockIdx.x + threadIdx.x;
5     int my_lane = threadIdx.x % warpSize;
6
7     shared_vals[my_lane] = 0.0f;
8     if (0 < my_i && my_i < n) {
9        float my_x = a + my_i*h;
10       shared_vals[my_lane] = f(my_x);
11    }
12
13    float result = Shared_mem_sum(shared_vals);
14    /* ATENÇÃO Assume tamanho do bloco = tamanho warp*/
15    if (threadIdx.x == 0)
16       atomicAdd(trap_p, result);
17 }
```

UFABC

Executando o mesmo teste anterior

- 2^{20} = 1.048.576 trapézios
- Como queremos fixar o tamanho do bloco como o tamanho do warp (32), precisamos de 32.768 blocos $(32 \times 32768 = 1048576)$

Table 6.8 Mean run-times for trapezoidal rule using block size of 32 threads (times in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Original	33.6	20.7	4.48	3.08
Warp Shuffle		14.4		0.210
Shared Memory		15.0		0.206

Aproximação trapezoidal: A revanche

Qual é o nosso problema?

UFABC

- Usar blocos com apenas 32 threads reduz a flexibilidade e o desempenho dos programas CUDA.
- Dispositivos com capacidade de computação ≥ 2.0 permitem blocos com até 1024 threads, ou de maneira equivalente, cada bloco pode ter até 32 warps
  - Dado o índice de um thread, é fácil determinar o seu warp:

```
1  int my_warp = threadIdx.x / warpSize;
```

Rascunho de solução

UFABC

Para múltiplos blocos com até 1024 threads/32 warps:

1. Cada thread calcula a sua contribuição
   - No nosso exemplo, calcula f
2. Cada warp calcula a soma de seus próprios threads
   - Já fizemos isso no exemplo anterior
3. O warp 0 de cada bloco calcula a soma dos resultados de todos warps de seu próprio bloco
4. Somamos os resultados obtidos por todos os blocos

Mas, temos um probleminha...

Rascuno 2 de solução

UFABC

Para múltiplos blocos com até 1024 threads/32 warps:

1. Cada thread calcula a sua contribuição
   - No nosso exemplo, calcula f
2. Cada warp calcula a soma de seus próprios threads
   - Já fizemos isso no exemplo anterior
3. Espera todos os warps de um bloco acabar o passo 2
4. O warp 0 de cada bloco calcula a soma dos resultados de todos warps de seu próprio bloco
5. Somamos os resultados obtidos por todos os blocos

Como implementar 3?

__syncthreads

UFABC

```
1 // Sincroniza todos os threads de um bloco
2 __device__ void __syncthreads(void);
```

- Seu funcionamento é semelhante às barreiras que já vimos
  - MPI: `MPI_Barrier` sincroniza processos em um comunicador
  - OpenMP: `#pragma omp barrier` sincroniza threads em uma região paralela
  - CUDA: `__syncthreads` sincroniza todos os threads de um bloco
- Assim como seus análogos, TODOS precisam sincronizar para que o processamento continue
- Atente-se ao fato que só são sincronizados os threads do mesmo bloco, threads de outros blocos não são afetados

Mas ainda tem uma coisinha...

UFABC

- Pergunta: Como fazer o warp 0 de cada bloco ter acesso aos resultados calculados pelos demais warps daquele bloco?
  - Resposta: memória compartilhada!

Esqueleto da solução

UFABC

```
1  ...
2  __shared__ float warp_sum_arr[32];
3  int my_warp = threadIdx.x / warpSize;
4  int my_lane = threadIdx.x % warpSize;
5
6  // PASSO 1 - Cada thread calcula sua contribuição
7  ...
8  // PASSO 2 - Soma dos resultados de cada warp
9  float my_result = Warp_sum(my_trap);
10 if (my_lane == 0) {
11     warp_sum_arr[my_warp] = my_result;
12 }
13 // PASSO 3 - Aguarda todos os warps do bloco
14 __syncthreads();
15 /* PASSO 4 - O warp 0 de cada bloco calcula a soma
16    dos resultados de todos warps de seu próprio bloco
17    usando a variável warp_sum_arr */
18 ...
```

Esqueleto da solução - Segunda parte

UFABC

```
1  // PASSO 1 - Cada thread calcula sua contribuição
2  // PASSO 2 - Soma dos resultados de cada warp
3  // PASSO 3 - Aguarda todos os warps do bloco
4  /* PASSO 4 - O warp 0 de cada bloco calcula a soma
5     dos resultados de todos warps de seu próprio bloco
6     usando a variável warp_sum_arr */
7  if (my_warp == 0) {
8      if (threadIdx.x >= blockDim.x/warpSize)
9          warp_sum_arr[threadIdx.x] = 0.0;
10     my_trap = warp_sum_arr[threadIdx.x];
11     blk_result = Warp_sum(my_trap);
12 }
13 // PASSO 5 - Soma os resultados de todos os blocos
14 if (threadIdx.x == 0)
15     atomicAdd(trap_p, blk_result);
```

UFABC

- Aida é o mesmo teste
  - 2^{20} = 1.048.576 trapézios
  - E portanto precisamos de 2^10 threads
- Mas, dessa vez, vamos usar blocos de 1024 threads
  - Precisamos de 1024 blocos ($1024 \times 1024 = 1048576$)

Table 6.10 Mean run-times for trapezoidal rule using arbitrary block size (times in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Original	33.6	20.7	4.48	3.08
Warp Shuffle, 32 ths/blk		14.4		0.210
Shared Memory, 32 ths/blk		15.0		0.206
Warp Shuffle		12.8		0.141
Shared Memory		14.3		0.150

Conclusão

- GPGPU (General Purpose Programming on Graphics Processing Units)
  - Surgiram no início do século XXI para explorar o poder computacional das GPUs para computação de propósito geral
    - Tornou-se uma das formas mais importantes de computação paralela

Arquitetura básica de uma GPU

UFABC

- GPUs são processadores do tipo Single Instruction Multiple Data (SIMD)
  - Executam a mesma instrução em múltiplos caminhos de dados (datapaths), onde cada caminho pode ter seus próprios dados
- Em GPUs Nvidia
  - Os caminhos de dados são chamados de Streaming Processors (SPs)
  - Os SPs são agrupados em Streaming Multiprocessors (SMs)

Organização da execução

UFABC

- Analogias com software
  - SPs correspondem a threads
  - SMs correspondem a thread blocks
  - Diferenças
    - Um thread block é executado em um único SM
    - Múltiplos blocos independentes podem ser executados em um único SM
    - Os threads em um bloco não precisam executar em lockstep
- Warp
  - Subconjunto de threads em um bloco que executa em lockstep
  - Atualmente, um warp consiste em 32 threads

Tipos de memória em GPUs Nvidia

UFABC

- Memória compartilhada
  - Pequena e rápida
  - Compartilhada entre os threads de um bloco
- Memória global
  - Grande e mais lenta
  - Compartilhada entre todos os threads

CUDA (Era: Compute Unified Device Architecture)

UFABC

- API usada para programação GPGPU
- Extensão de C/C++
- Modelo de execução
  - O sistema possui uma CPU (host) e uma GPU (device)
  - A função principal (`main`) é executada na CPU
  - Um `kernel` é uma função CUDA chamada pelo host, mas executada na GPU
- Programação heterogênea
  - Um programa CUDA geralmente é executado em CPU e GPU, que possuem arquiteturas diferentes

Outras considerações

UFABC

- A memória unificada simplifica o acesso à memória entre host e device, mas pode não ser tão eficiente quanto o uso manual de memória
- O uso eficiente de registradores e memória compartilhada é crucial para o desempenho
- As funções de warp shuffle permitem operações eficientes entre threads em um warp
- A sincronização entre threads é essencial para evitar condições de corrida e garantir a consistência dos dados
- A programação CUDA é uma forma poderosa de explorar o potencial das GPUs, mas requer atenção a detalhes como gerenciamento de memória, sincronização e eficiência de acesso à memória