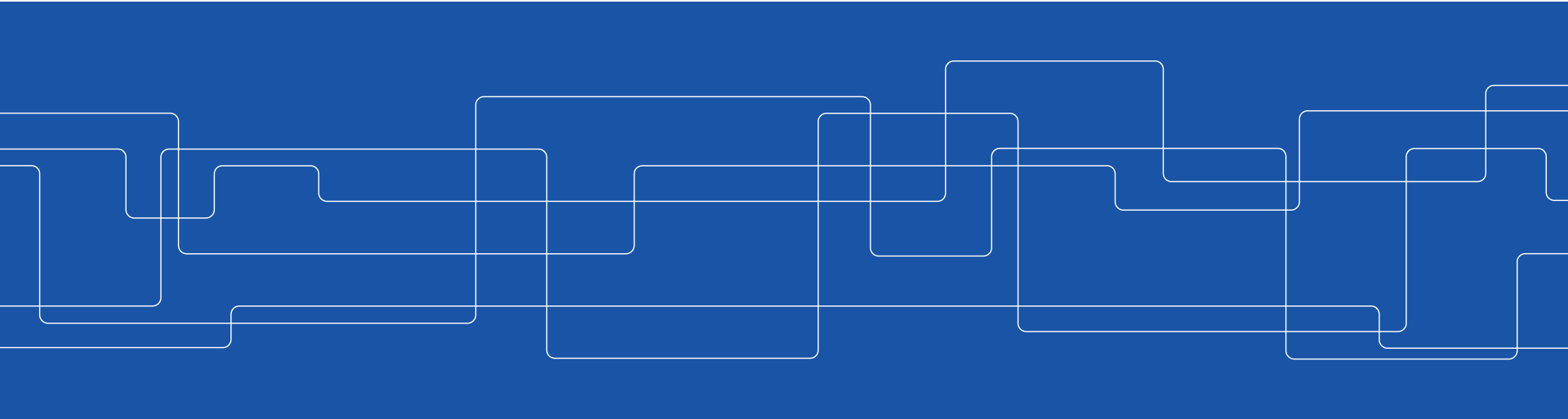
CUDA – Stencils, Shared Memory and Reduction Operations

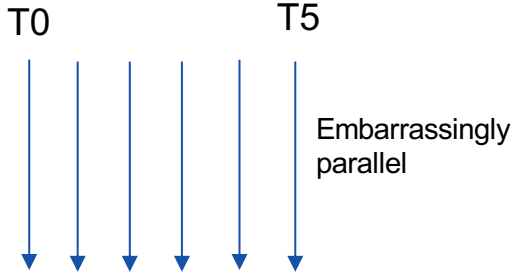
Stefano Markidis

KTH Royal Institute of Technology

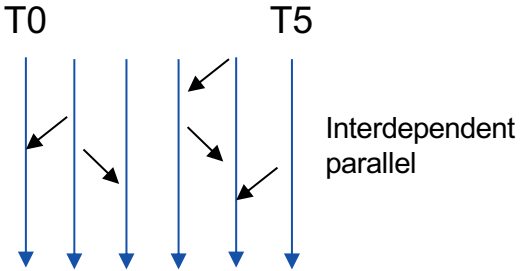


Embarrassingly Parallel Problems

So far we have only worked with independent tasks that didn't require data from other threads. This kind of problems are also called **embarrassingly parallel** problems.



However, the majority of applications have **interdependent threads**. We will deal with them now.

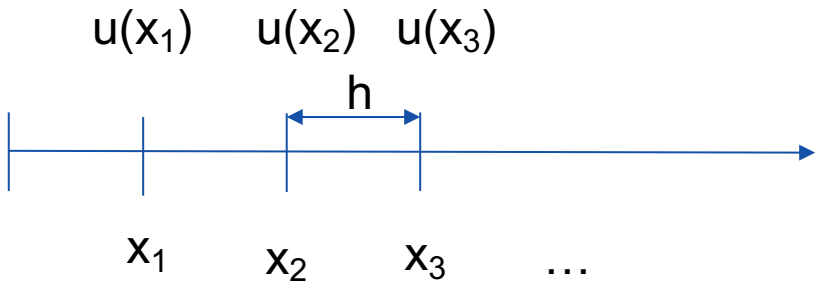


Example: 2nd Order Derivative on 1D Grid

Problem: we have a function `u(x)= sin(x)` defined on discrete points of a 1D grid with grid spacing `h`. The second order derivative at the point x_i can be calculated as

$$(d^2/dx^2\ u)|_i = (u_{i+1} - 2u_i + u_{i-1})/h^2$$

We want to implement a code to calculate this using CUDA



Interdependency of threads

If we are in thread `i`, we want to calculate:

$$(d^2/dx^2\ u)|_i = (u_{i+1} - 2u_i + u_{i-1})/h^2$$

Problem: *compute the difference between the value of a variable in this thread and the value of the same variable in the adjacent grid*

Two types of GPU memory so far ...

So far we have used two types of memories without identifying them:

- **Register memory** where the local variables for each thread are stored. Register memory is as close to the SM as possible, so **it is fastest but its scope is local** only to a single thread. In the previous slide, we ”assumed” `u` as register variable.

- **Global memory** is the GPU memory that has been allocated with `cudaMalloc()`. It provides **the bulk of the device memory capacity** but **it is far from the GPU** so it slower than register memory. However, this memory is accessible by all threads.

Question: is `x` a register or global variable ?

```
#include <stdio.h>
#define N 64
#define TPB 32


__device__ float scale(int i, int n)
{
  return ((float)i)/(n - 1);
}


__device__ float distance(float x1, float x2)
{
  return sqrt((x2 - x1)*(x2 - x1));
}


__global__ void distanceKernel(float *d_out, float ref, int len)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float x = scale(i, len);
  d_out[i] = distance(x, ref);
  printf("i = %2d: dist from %f to %f is %f.\n", i, ref, x, d_out[i]);
}
```

```
int main()
{
  const float ref = 0.5f;

  // Declare a pointer for an array of floats
  float *d_out = 0;

  // Allocate device memory to store the output array
  cudaMalloc(&d_out, N*sizeof(float));

  // Launch kernel to compute and store distance values
  distanceKernel<<<N/TPB, TPB>>>(d_out, ref, N);

  cudaFree(d_out); // Free the memory
  return 0;
}
```

Question: is `x` visible by all the threads?

Question: is `d_out` a register or global variable ?

```
#include <stdio.h>
#define N 64
#define TPB 32


__device__ float scale(int i, int n)
{
  return ((float)i)/(n - 1);
}


__device__ float distance(float x1, float x2)
{
  return sqrt((x2 - x1)*(x2 - x1));
}


__global__ void distanceKernel(float *d_out, float ref, int len)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float x = scale(i, len);
  d_out[i] = distance(x, ref);
  printf("i = %2d: dist from %f to %f is %f.\n", i, ref, x, d_out[i]);
}
```

```
int main()
{
  const float ref = 0.5f;

  // Declare a pointer for an array of floats
  float *d_out = 0;

  // Allocate device memory to store the output array
  cudaMalloc(&d_out, N*sizeof(float));

  // Launch kernel to compute and store distance values
  distanceKernel<<<N/TPB, TPB>>>(d_out, ref, N);

  cudaFree(d_out); // Free the memory
  return 0;
}
```

Question: is `d_out` visible by all the threads?

Going back to our initial problem

Problem: we have a function $u(x) = \sin(x)$ defined on discrete points of a 1D grid with grid spacing h. The second order derivative at the point x_i can be calculated as

$$(d^2/dx^2\ u)|_i = (u_{i+1} - 2u_i + u_{i-1})/h^2$$

Question: Should u be register or global?

CUDA code

```
#include <stdio.h>
#define TPB 64

__global__
void ddKernel(float *d_out, const float *d_in, int size, float h)
{
  const int i = threadIdx.x + blockDim.x*blockIdx.x;
  if (i >= size) return;
  d_out[i] = (d_in[i - 1] - 2.f*d_in[i] + d_in[i + 1]) / (h*h);
}
```

Question: are `d_in` really visible by all threads?

```
int main() {
  const float PI = 3.1415927;
  const int N = 150;
  const float h = 2 * PI / N;
  float x[N] = { 0.0 };
  float u[N] = { 0.0 };
  float result_parallel[N] = { 0.0 };

  for (int i = 0; i < N; ++i) {
    x[i] = 2 * PI*i / N;
    u[i] = sinf(x[i]);
  }
 ddParallel(result_parallel, u, N, h);
 }

// second order derivative
void ddParallel(float *out, const float *in, int n, float h) {
  float *d_in = 0, *d_out = 0;

  cudaMalloc(&d_in, n*sizeof(float));  // on global memory
  cudaMalloc(&d_out, n*sizeof(float)); // on global memory
  cudaMemcpy(d_in, in, n*sizeof(float), cudaMemcpyHostToDevice);

  ddKernel<<<(n + TPB - 1)/TPB, TPB>>>(d_out, d_in, n, h);

  cudaMemcpy(out, d_out, n*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(d_in); cudaFree(d_out);
}
```

Efficiency of this approach

To use global variables is not the most efficient approach because:

- **Memory traffic and synchronization:** When a large grid is launched, there may be millions of threads trying to read and write values to and from the same arrays (`d_in[i]` gets requested as the right-hand neighbor value, center value, and left-hand neighbor value by threads with index `i-1`, `i`, and `i+1` respectively)

- **Access** to global memory is **relatively slow** because far from the GPU.

A third kind of GPU memory: Shared Memory

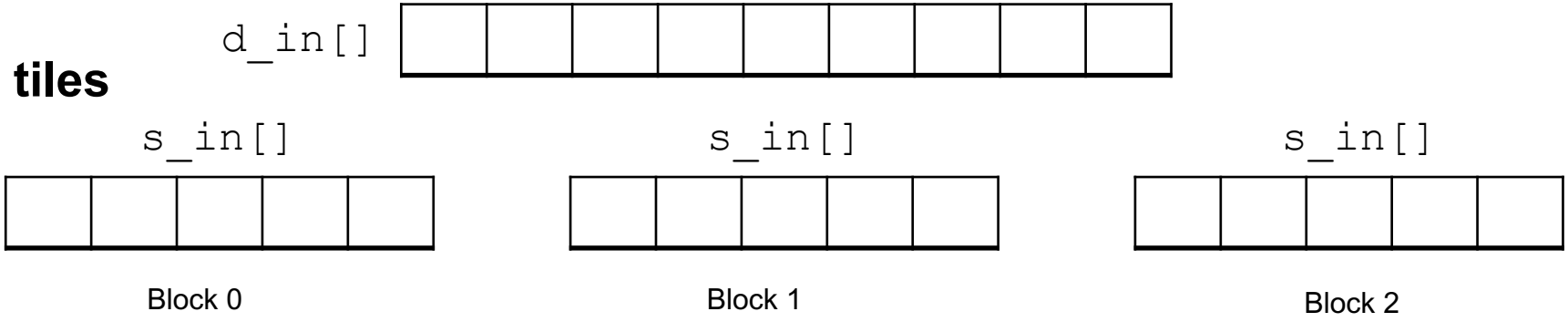
GPUs also have **shared memory** which aims to bridge the gap in memory speed between global and register. Its usage leads to a performance increase in most of the cases.

Shared memory resides adjacent to the SM and provides up to 48KB of storage that can be accessed efficiently **by all threads in a block**.

Shared data tiles from global array

When using GPU shared memory, the basic approach is to **break the large global array** into **tiles** of **data providing all the input required for a computational block**.

3 tiles



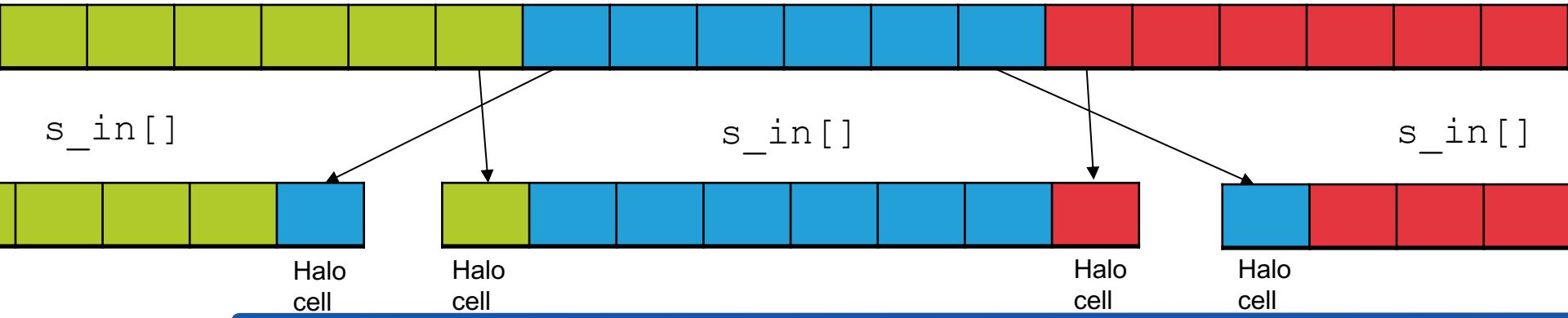
Halo/Ghost Cells

We divide the global array in smaller arrays of equal size

Problem: to compute $(u_{i+1} - 2u_i + u_{i-1})/h^2$ on the boundary cells of the tile requires to know the values on adjacent parts.

Solution: include array elements at the boundary of the adjacent parts in the shared array

Question: what is the size of the shared array if we divide global array of size `N` by the number of blocks?



How to declare shared variables – 1. Fixed Size

Shared variables are declared in the kernel function.

We declare shared arrays using the `__shared__` qualifier.

If you create your shared **array with a fixed size**, the array can be created simply prepending `__shared__` to the type of the array, e.g:

```
__shared__ float s_in[34];
__shared__ float s_in[blockDim.x + 2];
```

2. Shared Array Allocated Dynamically

If we **allocate the array dynamically**, the declaration requires the keyword `extern` as follows:

```
extern __shared__ float s_in[];
```

Question: how we specify the size of `s_in`?
In the execution configuration as third argument!

Launching a Kernel using a Shared Variable

If we are using a shared variable in the kernel, the kernel launch **requires a third argument** within <<< … >>> to **specify the size of the shared memory allocation in bytes**

```
int smemSize = (TPB + 2)*sizeof(float);
ddKernel <<< (n+TPB-1)/TPB, TPB, smemSize>>> (args)
```

Indexing when Using Shared Arrays

The use of shared (to the block) array make indexing little bit more complicated. To keep the bookkeeping simple, a good approach is to

- maintain the usual index `i` (which we call **global index** because it identifies the corresponding element of the array in **global memory**)
- to introduce a **local index** `s_idx` to keep track of where things are stored in the **shared array**.

Indexing with Halo Cells

We can introduce the radius (`RAD`) of the stencil, that is the number of cells we need in either of the boundary. In the second order derivative problem, `RAD = 1`.

When handling a stencil with radius `RAD`, in addition to one element for each thread in the block, **the shared array must also include `2*RAD` halo cells added to each end.**

The first thread in the block need to leave room for `RAD` neighbors and therefore get local index `s_idx = RAD`. The general relation between the local index and thread index is

```
s_idx = threadIdx.x + RAD;
```

Filling the Shared Arrays

Once the shared array is allocated, we are ready to **transfer the data from global memory to shared memory.**

Each thread requests the entry in the input array whose index matches the thread's global index.

```
__global__
void ddKernel(float *d_out, const float *d_in, int
size, float h) {

  // global index
  const int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i >= size) return;
  // local index
  const int s_idx = threadIdx.x + RAD;

  // declared shared
  extern __shared__ float s_in[];

  // fill
  s_in[s_idx] = d_in[i];

  …
```

Filling the shared arrays: Halo cells

The values **for the halo cells still need to be obtained and stored**:

```
 // fill
 s_in[s_idx] = d_in[i];
 // Halo cells
if (threadIdx.x < RAD) {
  s_in[s_idx - RAD] = d_in[i - RAD];
  s_in[s_idx + blockDim.x] = d_in[i + blockDim.x];
}
```

Thread `0` has `s_idx = RAD`, so:

- `s_idx-RAD` is `0` and the leftmost neighbor gets stored in the leftmost halo cell at the beginning of the shared array
- `s_idx+blockDim.x` is `blockDim.x` and the immediate neighbor to the right gets stored in the leftmost halo cell at the end of the array

Synchronization of Shared Arrays

Kernel launches are asynchronous: we can't assume that all the input data has been loaded in the shared memory array before threads execute the statement using shared arrays.

To ensure that all the data has been properly stored, we employ the CUDA function:

```
__synchthreads();
```

This forces all the threads in the block to complete the previous statements before any thread in the block proceeds further.

```
…
// Regular cells
s_in[s_idx] = d_in[i];
// Halo cells
if (threadIdx.x < RAD) {
  s_in[s_idx - RAD] = d_in[i - RAD];
  s_in[s_idx + blockDim.x] =
      d_in[i + blockDim.x];
}
__syncthreads();
…
```

Question: what is the difference between `__synchthreads()` and `cudaDeviceSynchronize()`?

GPU Code – Kernel Launching

```
#define TPB 64
#define RAD 1 // radius of the stencil
…

void ddParallel(float *out, const float *in, int n, float h) {
  float *d_in = 0, *d_out = 0;
  cudaMalloc(&d_in, n * sizeof(float));
  cudaMalloc(&d_out, n * sizeof(float));
  cudaMemcpy(d_in, in, n * sizeof(float), cudaMemcpyHostToDevice);

  // Set shared memory size in bytes
  const size_t smemSize = (TPB + 2 * RAD) * sizeof(float);
  ddKernel<<<(n + TPB - 1)/TPB, TPB, smemSize>>>(d_out, d_in, n, h);
  cudaMemcpy(out, d_out, n * sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(d_in);
  cudaFree(d_out);
}
```

GPU code – Kernel Definition

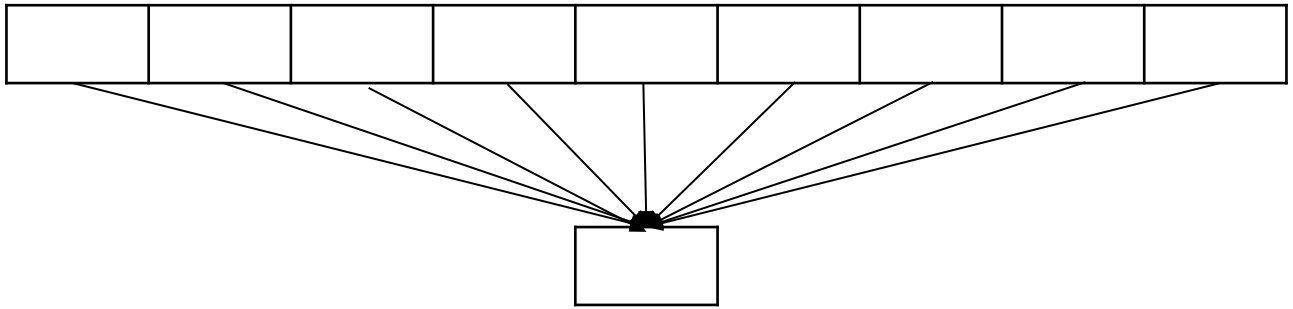
```
__global__ void ddKernel(float *d_out, const float *d_in, int size, float h) {
  const int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i >= size) return;

  const int s_idx = threadIdx.x + RAD;
  extern __shared__ float s_in[];

  // Regular cells
  s_in[s_idx] = d_in[i];
  // Halo cells
  if (threadIdx.x < RAD) {
    s_in[s_idx - RAD] = d_in[i - RAD];
    s_in[s_idx + blockDim.x] = d_in[i + blockDim.x];
  }
  __syncthreads();
  d_out[i] = (s_in[s_idx-1] - 2.f*s_in[s_idx] + s_in[s_idx+1])/(h*h);
}
```

Reduction Operation

We look now at a different and very common type of problem: reduction. In the reduction, elements of an input array are combined to obtain a single output



When do you need reductions ? **dot products**, **image similarity measures**, **integral properties** (remember `pi.c` in the OpenMP lab), and **histograms** require reduction

Parallel reduction: parallel dot product

We now want to parallelize with CUDA this serial CPU code to calculate the dot product of two arrays, `a` and `b`, of size `N`:

```
for (int i = 0; i < N; ++i) {
    cpu_res += a[i] * b[i];
}
```

What would be one easy implementation?

- Move `a` and `b` to GPU memory
- Create a `d_res[]` array on the GPU to hold the result of single element multiplication `d_res[i] = a[i]*b[i]`
- Move `d_res[]` to `res[]` in the CPU memory
- Sum up all the elements of `res[]` in one scalar value `sum`.

Question: Can we do better ? Any better strategy?

Possible solution: tiles and shared

The global memory traffic can be reduced by:

- taking a tile approach in which we break the large input vectors up to in `N/Number of blocks` pieces.

Each tile would consists of **threads_per_block-sized shared arrays**.

Question: do we need halo/ghost cells?

- updating `d_res` once per block

The Shared “Strategy”: Create array and fill it

- Create a **shared memory array** to store the product of corresponding entries in the tiles of the input arrays. Fill the array:

```
__shared__ int s_prod[TPB];
s_prod[s_idx] = d_a[idx] * d_b[idx];
```

Question: What do we need to do next?

- **Synchronize** to ensure that the shared array is completely filled before proceeding!

The Shared “Strategy”: Sum partial sum over the blocks

- Assign **one thread** (e.g. with `threadIdx.x == 0`) to loop over the shared (whose data is accessible by any thread in the block) **to accumulate the contribution from the block** into a register variable `blockSum`

- Perform **one global memory transaction** (for the whole block) to add `blockSum` to the value stored in the accumulator `d_res`

CUDA Code

```
#include <stdio.h>
#define TPB 64

void dotLauncher(int *res, const int *a, const int *b, int n) {
  int *d_res;
  int *d_a = 0;
  int *d_b = 0;

  cudaMalloc(&d_res, sizeof(int));
  cudaMalloc(&d_a, n*sizeof(int));
  cudaMalloc(&d_b, n*sizeof(int));

  cudaMemset(d_res, 0, sizeof(int));
  cudaMemcpy(d_a, a, n*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, n*sizeof(int), cudaMemcpyHostToDevice);

  dotKernel<<<(n + TPB - 1)/TPB, TPB>>>(d_res, d_a, d_b, n);
  cudaMemcpy(res, d_res, sizeof(int), cudaMemcpyDeviceToHost);

  cudaFree(d_res);
  cudaFree(d_a);
  cudaFree(d_b);
}
```

Question: why the third argument is missing?

```
__global__ void dotKernel(int *d_res, const int *d_a, const
int *d_b, int n) {

  const int idx = threadIdx.x + blockDim.x * blockIdx.x;
  if (idx >= n) return;

  const int s_idx = threadIdx.x;

  __shared__ int s_prod[TPB];
  s_prod[s_idx] = d_a[idx] * d_b[idx];
  __syncthreads();


  if (s_idx == 0) {
    int blockSum = 0;
    for (int j = 0; j < blockDim.x; ++j) {
      blockSum += s_prod[j];
    }
    printf("Block %d, blockSum = %d\n", blockIdx.x, blockSum);
    *d_res += blockSum;

  }
}
```

Question: Is this code correct?

Code Correctness

Thread `0` in each block reads a value of `d_res` from global memory, adds its value of `blockSum` and stores the results back into the memory location where `d_res` is stored.

Problem: the outcome of these operations depends on the sequence in which they are performed by each thread!

Question: Have you seen this problem before?

```
__global__ void dotKernel(int *d_res, const int *d_a, const
int *d_b, int n) {

  const int idx = threadIdx.x + blockDim.x * blockIdx.x;
  if (idx >= n) return;

  const int s_idx = threadIdx.x;

  __shared__ int s_prod[TPB];
  s_prod[s_idx] = d_a[idx] * d_b[idx];
  __syncthreads();


  if (s_idx == 0) {
    int blockSum = 0;
    for (int j = 0; j < blockDim.x; ++j) {
      blockSum += s_prod[j];
    }
    printf("Block %d, blockSum = %d\n", blockIdx.x, blockSum);
    *d_res += blockSum;

  }
}
```

Race Condition

This situation, in which the order of operations whose sequencing is uncontrollable, is called a **race condition**, and the race condition results in undefined behavior (most of times results in data corruption).

We saw this problem in the OpenMP lectures and lab exercises.

Problem: How did we solve this problem in OpenMP?

CUDA Atomic Functions

The solution is to use CUDA atomic functions (we don't have critical session in CUDA).

Atom in ancient Greek means *uncuttable* or *indivisible*. An atomic function performs **read-modify-write sequence** of operations **as an indivisible unit**.

Using `atomicAdd()` to solve the race condition

```
__global__ void dotKernel(int *d_res, const int *d_a, const int *d_b, int n) {
  const int idx = threadIdx.x + blockDim.x * blockIdx.x;
  if (idx >= n) return;
  const int s_idx = threadIdx.x;


  __shared__ int s_prod[TPB];
  s_prod[s_idx] = d_a[idx] * d_b[idx];
  __syncthreads();


  if (s_idx == 0) {
    int blockSum = 0;
    for (int j = 0; j < blockDim.x; ++j) {
      blockSum += s_prod[j];
    }
    printf("Block_%d, blockSum = %d\n", blockIdx.x, blockSum);

    atomicAdd(d_res, blockSum);
    }
}
```

Other CUDA Atomic Operations

Together with `atomicAdd()`, CUDA offers 10 other atomic functions: `atomicSub()`, `atomicExch()`, `atomicMin()`, `atomicMax()`, `atomicInc()` ,`atomicDec()`, `atomicCAS()` (CAS = compare and swap) and three bitwise functions `atomicAnd()`, `atomicOr()` and `atomicXor()`.

Problem: Atomic operations force some **serialization** and **slow down** things a bit. Use only when really needed!

CUDA Advanced Features

We have reached now the end of our introduction to CUDA. We haven't had time cover other CUDA more advanced features, like:

- CUDA Unified Memory
- CUDA Dynamic Parallelism
- CUDA Streams
- CUDA performance optimization: branching and memory coalescence

However, you know now all the important CUDA concepts and acquired all the terminology to also understand other CUDA concepts