

# CUDA Memories, Shared and Atomics

Stefano Markidis and Sergio Rivas-Gomez

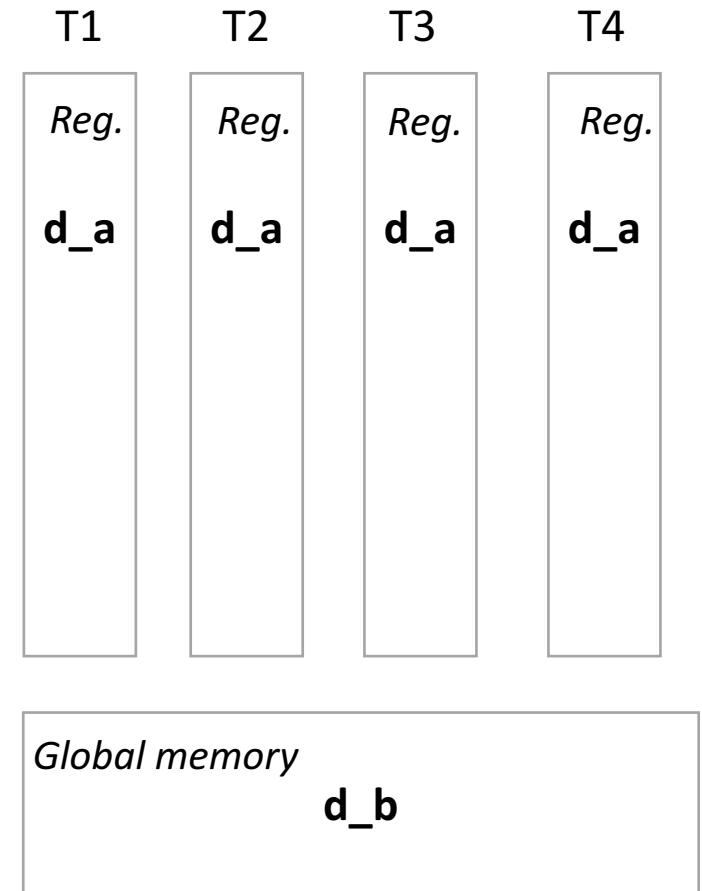
# Three Key-Points

- GPU has three kinds of memories: register, local to the thread, global, accessible by all threads, and shared, accessible only by threads on the SM.
- Shared memory is local to the SM, faster than global and it can be used for performance optimization
- When using global memory, we need to use atomic operations to avoid race conditions

# 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.
- **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.



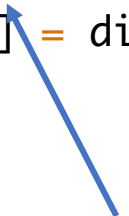
# Question: is x a register or global variable ?

```
#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);
}
```



```
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
    cudaMalloc(&d_out, N*sizeof(float));

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

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

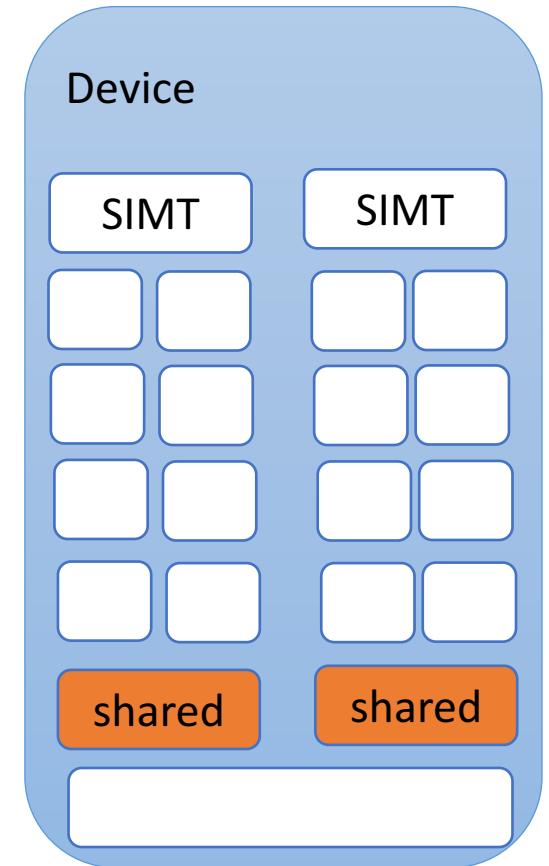
# Global Memory is very Convenient. However, ...

1. **Memory traffic:** when a large grid is launched, there may be millions of threads trying to read and write values to and from the same arrays
2. Access to global memory is relatively **slow** because **far** from the GPU.
3. **Concurrent update** of global memory might result in **data corruption** (race condition)



# 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.



# How to Declare Shared Variables (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];
```

# How to Declare Shared Variables (Dynamic All.)

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!



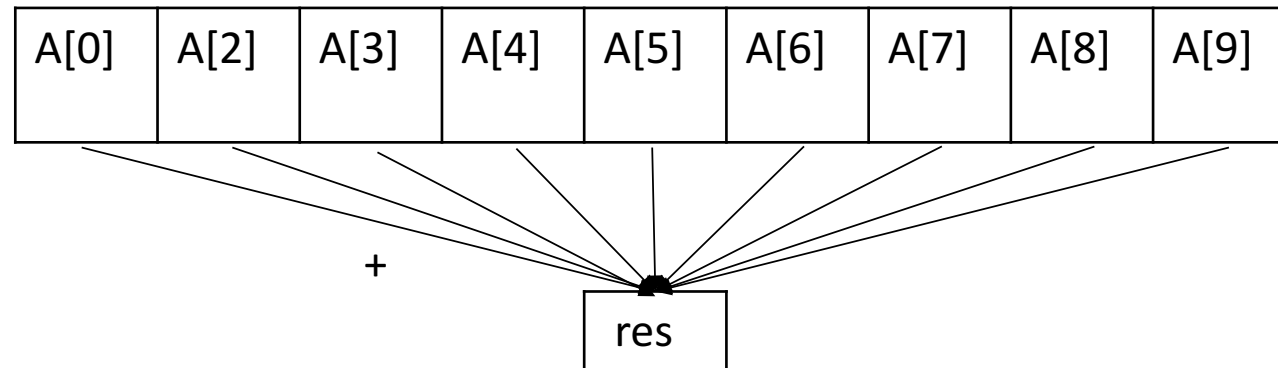
# Launching a Kernel Using a Shared Variable

If we are using a dynamic 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)*sizeof(float);  
ddKernel <<< (n+TPB-1)/TPB, TPB, smemSize>>> (args)
```

# Reduction Operation

We look now at a very common type of problem where we can use shared: the operation 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 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 for GPU?

- 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 serially all the elements of `res[]` in one scalar value `sum`.

# Possible solution: tiles and shared

- The global memory traffic can be reduced by:
  - **Taking a tile approach:** we break the large input vectors up to in `N/Number of blocks` pieces.
- Each tile would consist of **threads\_per\_block-sized shared arrays** to store the result of the multiplication.

# The Tile Approach I

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];
```

# 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:

```
__syncthreads();
```

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

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

```

#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!

```
__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).

# CUDA Atomics Functions

The solution is to use CUDA atomic functions. 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()`.

# To Summarize

- GPU has three kinds of memories: register, local to the thread, global, accessible by all threads, and shared, accessible only by threads on the SM.
- Shared memory is local to the SM, faster than global and it can be used for performance optimization
- When using global memory, we need to use atomic operations to avoid race conditions

# CUDA Advanced Features

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

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