LESSON 3

Shared Memory and Reduction Pattern

TOPICS

- How to use the shared memory
- How to synchronize threads
- One way of programming the Mean Square Error
- How to manipulate threads to do a reduction

- *Key words: __shared__, __syncthreads(), reduction.*

More of Threads and Blocks

- In the matrix manipulation, we started to play with the threads and blocks, where for every block there was a copy of the kernel to be executed for the N threads declared in the *execution configuration*.

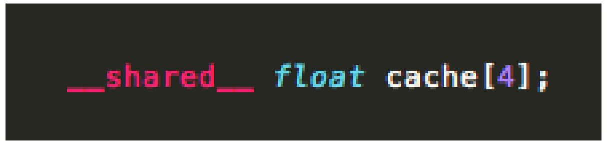
- Each thread just made one operation and stored the result in other variable; there was not communication between the threads in the same block, but what if this is needed?

- For this purpose we have the shared memory. This memory is used by threads in the same block. If you declare a shared variable, this is copied in every block and the threads can manipulate it, but they cannot manipulate the shared variable from other blocks.

- For example, if we are working with different threads maybe we need that thread *A* do some duty, while thread *B* waits for thread *A*. In this case we are creating a race condition and we need a way to synchronize the threads, a way to avoid this is to use the function *__syncthreads()*.

- It's important to be aware that calling __syncthreads() in divergent code is undefined and can lead to deadlock—all threads within a thread block must call __syncthreads() at the same point.

- To use the shared memory we need to use the keyword __shared__.



```
__shared__ float cache[4];
```

- In the other examples, we have made use of the global memory. The global memory is the biggest one in the device, but it is kind of slow; in contrast, the shared memory is much faster than the global memory, but it is smaller.

- Each thread can:
  - Read / write per thread **registers**
    - ~1 cycle, extremely high throughput
  - Read / write per thread **shared memory**
    - ~5 cycle, high throughput
  - Read / write per thread **global memory**
    - ~500 cycle, modest throughput

Static and Dynamic Shared Memory

- The static memory is defined at compile time when we can explicitly declare an array of fixed amount size.

- The dynamic shared memory can be used when the amount of shared memory is not known at compile time. In this case the size must be specified in bytes using an optional third execution configuration parameter.

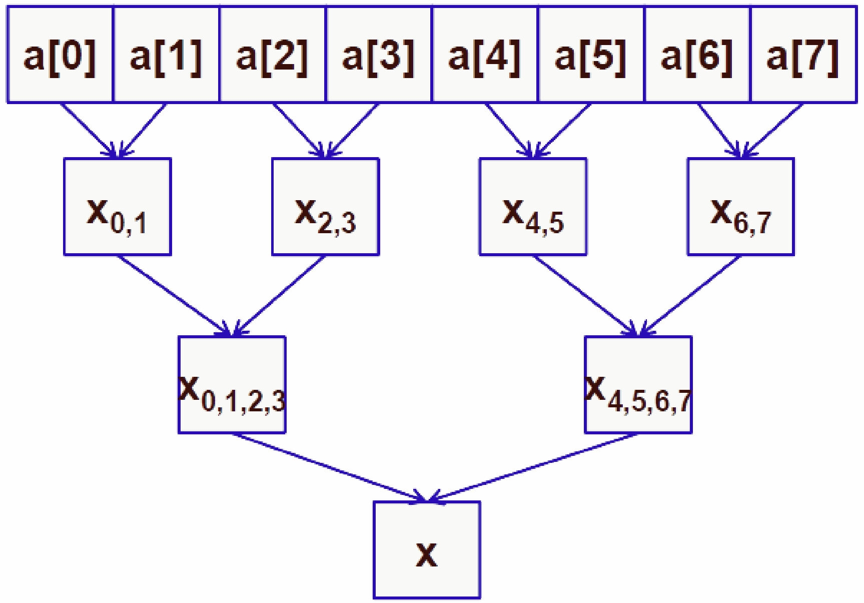
Example

- 00static_and_dynamic: This example shows the use of static and dynamic memory.

Reduction

- The reduction is a way of recollect results from some operations (collective operations).

- It is possible to get from the reduction
  - Min value
  - Max value
  - Sum reduction
  - Product reduction



Example

- 01reduction_shared: This example computes the dot product between two vectors using shared memory. This includes the Vector.h header.

A Common Programming Strategy

- Partition data into subsets or tiles that fit into shared memory.

- Use one thread block to handle each tile by:
  - Loading the tile from global memory to shared memory, using multiple threads.
  - Performing the computation on the subsets from shared memory, reducing traffic to the global memory.
  - Copying results from shared to global memory.

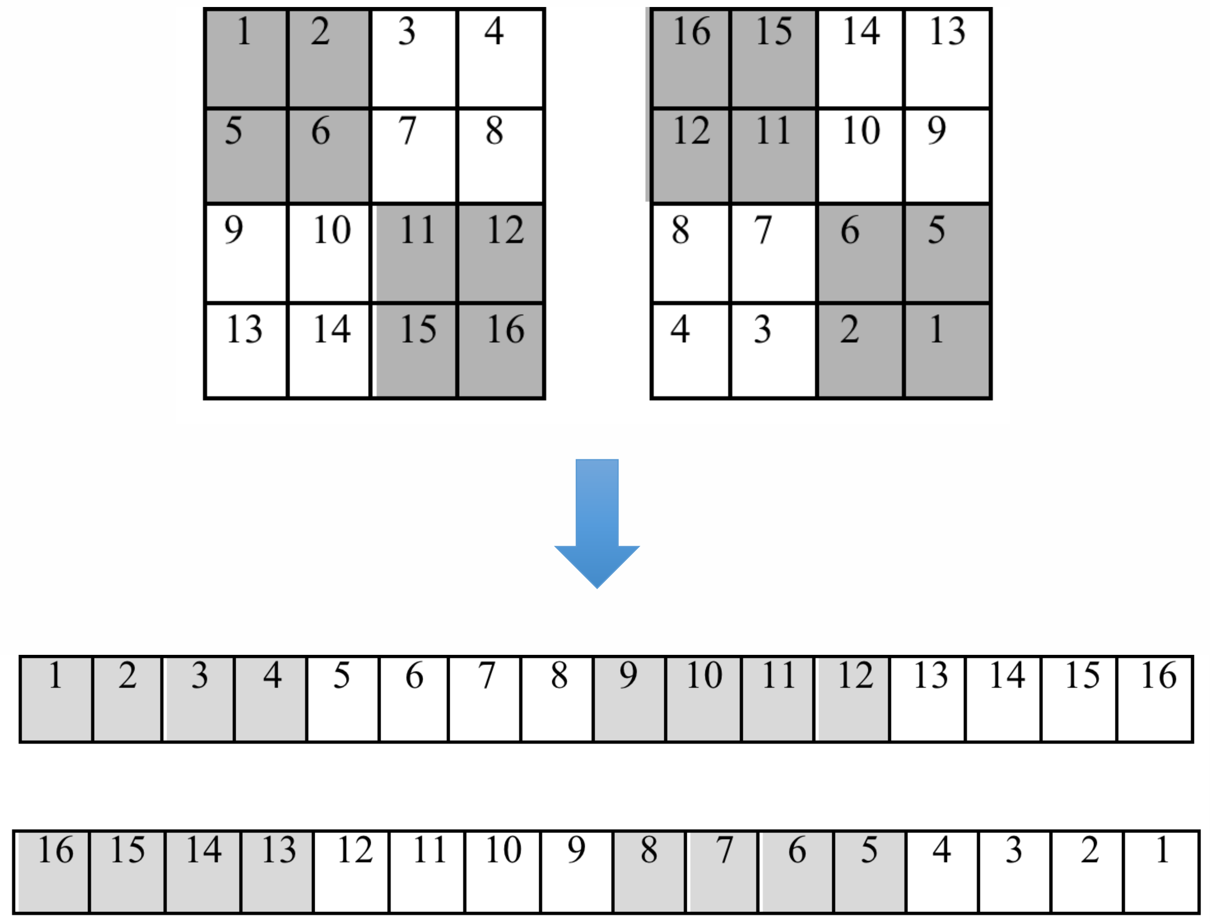
Practice

- 02mse_shared: This code must obtain the Mean Square Error (MSE) from different sections of a matrix. The matrices are linearized in vectors.

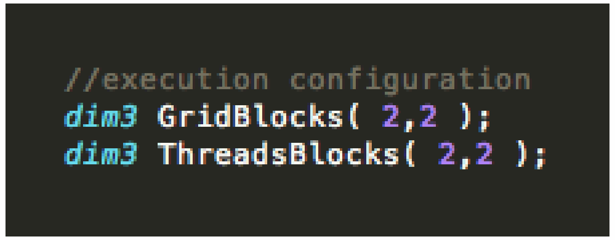
$$MSE = \sum_{0}^{n-1} (A_1 - A_2)^2$$

- We are going to use two small arrays of sixteen elements —to have a better outlook of how to deal with the data—. The data will be divided in four blocks of four elements each one, and we are going to use the same procedure as in the matrix manipulation practice.

- The data will be mapped in the device as shown in the next tables.

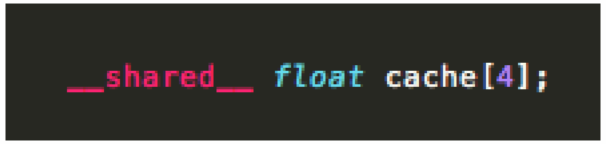


- It is necessary to create a grid of four blocks and four threads in each one.



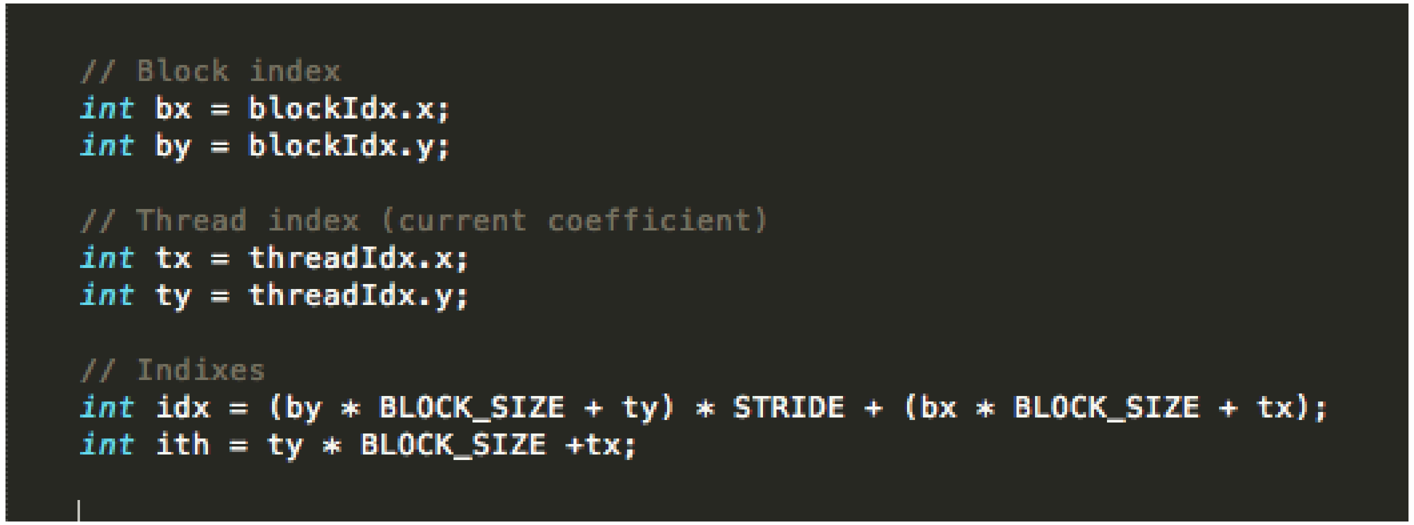
```
//execution configuration
dim3 GridBlocks( 2,2 );
dim3 ThreadsBlocks( 2,2 );
```

- To create the kernel subroutine we are going to use the shared memory declaring an array of four elements —*cache[4]*—, so, every block has one array of four elements where the threads can share data —if and only if the threads are in the same block—.



```
__shared__ float cache[4];
```

- Now, we need to generate the indexes to play with the data. They will be generated like in the matrix manipulation practice, but in this case we are going to need two indexes, one to extract the data from the arrays (*idx*), and one to store the result in the *cache* array (*ith*).

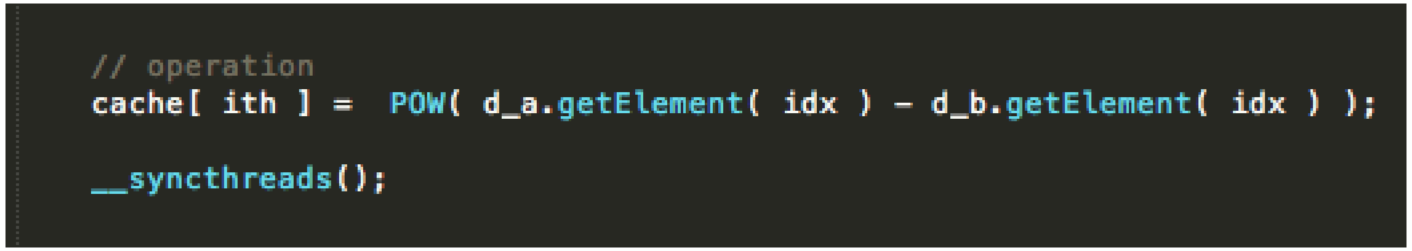


```
// Block index
int bx = blockIdx.x;
int by = blockIdx.y;

// Thread index (current coefficient)
int tx = threadIdx.x;
int ty = threadIdx.y;

// Indixes
int idx = (by * BLOCK_SIZE + ty) * STRIDE + (bx * BLOCK_SIZE + tx);
int ith = ty * BLOCK_SIZE +tx;
```

- The next step is to compute the subtraction and raising it to the power of 2, the result is stored in the *cache* array.

- After this point we need to be sure that the threads store their outcomes in the *cache*, due to the fact that we have to execute the sum, and it is necessary to have all the values in the *cache* array placed.

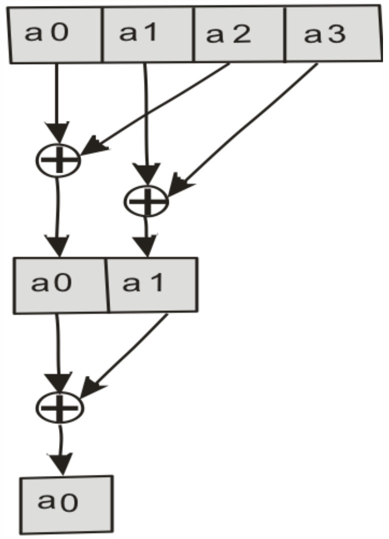
- This point is where we need to use the *__syncthreads()* in order to guarantee the right execution of this part of the code, before the hardware executes the next instruction on the threads.



```
// operation
cache[ ith ] =  POW( d_a.getElement( idx ) - d_b.getElement( idx ) );

__syncthreads();
```

- Once that we are sure that all the threads finished their first task, we are ready to do the reduction. For our code this is the tricky part, we have to manipulate our threads to use most of them.

- In the *cache* array we have four values, what we can do is to use half of the threads, and each thread will take two values from the *cache* array; after this we have two values to add, and we need just one thread to do it; you can see this procedure in the next figure.



- At the end of the operation, we are going to have the result saved in *cache[0]*.

- As in the last code, we need to be sure that the threads will finish their work before continuing with the next step, which bring us to use *__syncthreads()* again.

- In the next code *i=2* indicates that we are going to use two threads, to be precise, the two threads with index *0* and *1*. For the first iteration, the conditional *if* states that if the index *ith* < *i*, then:

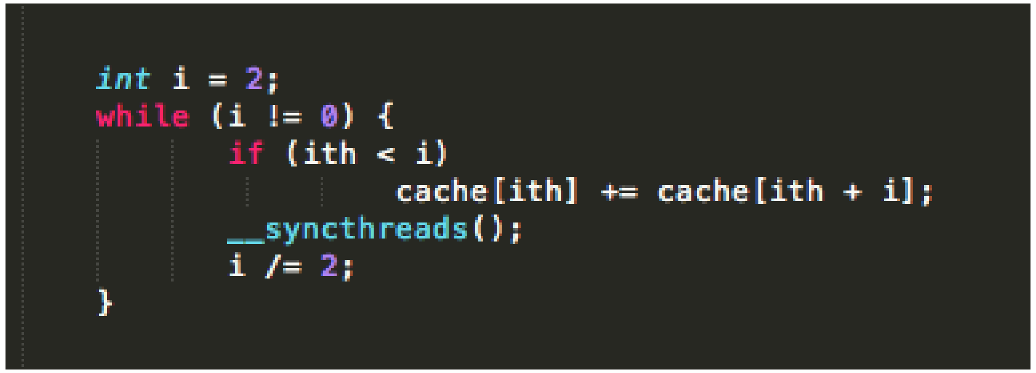
cache[0] += cache[0 + 2];

cache[1] += cache[1 + 2];

- At this moment, we are just using two threads to add two values but, what about the other threads that are not working?

- Divergence is when some threads execute some instructions while other do not. Here we have to take care of what we are doing; the *__syncthreads()* guarantees that all the threads must be executed, but if we have free threads, they will never reach the *__syncthreads()*.

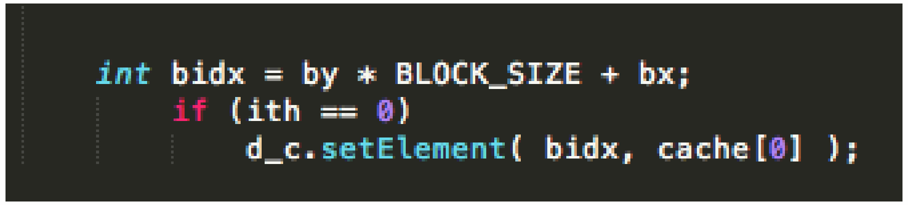
- This implies that *__syncthreads()* must be out of the *if*, on the other case the hardware will be waiting for the threads that have not executed the function forever.



```
int i = 2;
while (i != 0) {
        if (ith < i)
                cache[ith] += cache[ith + i];
        __syncthreads();
        i /= 2;
}
```

- Finally we need to set back the data to the global memory, to return the result to the host.

- We are not returning the final value; we are returning an array with one value calculated by block. This shows an example that sometimes if the data is too small we do not need to perform all the steps in the code, just the ones that require more processing.



```
    int bidx = by * BLOCK_SIZE + bx;
        if (ith == 0)
            d_c.setElement( bidx, cache[0] );
```

Tiling

- Tiling is a technnique used to store small data from the global memory into the shared memory. Process the data on the shared memory, and finally return the data to the global memory.

- This technique avoid to spend time reading-processing-storing data from and to the GPU's global memory.

- The disadvantage is amount of shared memory to use.

Practice

- 03tilev1 and 04tilev2: Apply the use of tiling to change the code of the different kernels.