Programming in CUDA

Part IV. Optimizing CUDA — 1

Frédéric S. Masset

Instituto de Ciencias Físicas, UNAM

Escuela de Verano GPU 2014

Outline

Outline

Accessing the global memory : good and bad ways — 1

Remember : access to global memory is **slow**. It takes 400-600 clock cycles. Example :

```
__global__ void simplekernel (float *x, int n) {
   int i = threadIdx.x + blockDim.x * blockIdx.x; Registers:
  float y;                                        fast access
  y = x[i]; x[i] in global memory: 400-600 clock cycles to read
  y = y + 1.0;
  x[i] = y; x[i] in global memory: 400-600 clock cycles to write
}
```

Note that there is no special manner to transfer global memory between the multiprocessors and the global memory. The mere invocation of a variable in global memory (an array element) achieves the transfer.

Optimizing global loads or stores with coalescence

load: from global memory to multiprocessor

store: from multiprocessor to global memory

- If subsequent threads access subsequent array elements in an orderly manner, the transaction is said *coalesced*, or *coherent*.
- The data can travel in parallel in a coalesced transaction (up to 16 parallel transactions – a half-warp)

Optimizing global loads or stores with coalescence

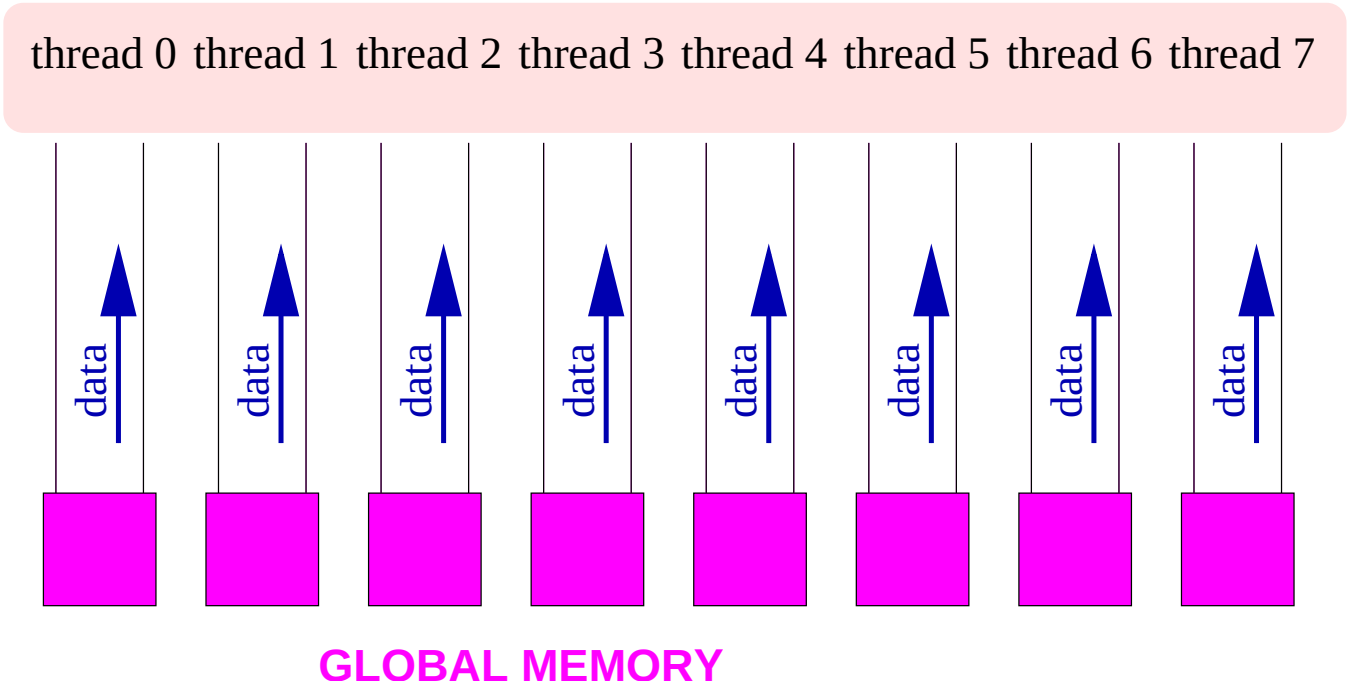
load: from global memory to multiprocessor

store: from multiprocessor to global memory

- If subsequent threads access subsequent array elements in an orderly manner, the transaction is said *coalesced*, or *coherent*.
- The data can travel in parallel in a coalesced transaction (up to 16 parallel transactions – a half-warp)

A naive model of a coalesced transaction

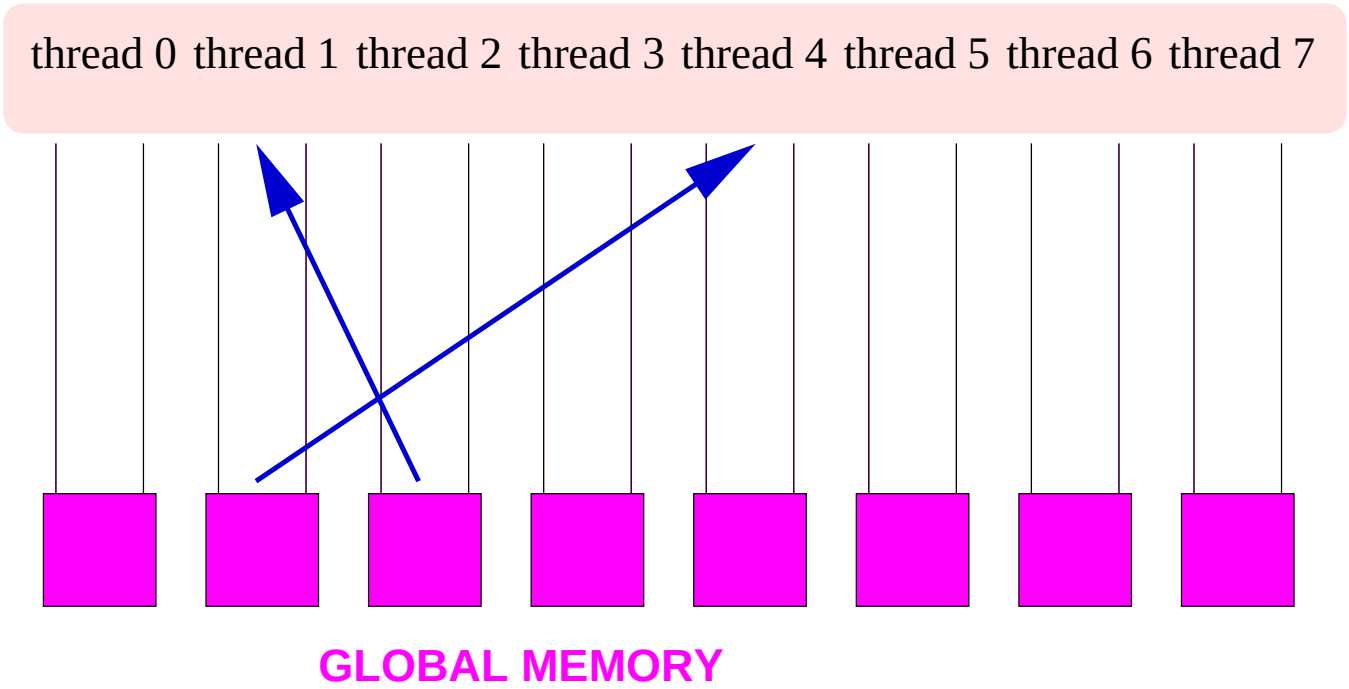
Think of the bus between the multiprocessor and the global memory as a motorway, with several lanes.



No risk of collision : data can travel together

A naive model of a coalesced transaction

Think of the bus between the multiprocessor and the global memory as a motorway, with several lanes.



Risk of collisions : data must travel one by one

Requirements for coalescence

Requirements for coalescence are of course more complex than the naive model shown previously. They depend on the compute capabilities of the device (1.x, 2.0). In any case, if subsequent aligned data is accessed by subsequent threads, the transaction is coalesced.

```
__global__ void simplekernel (float *x, int n) {
   int i = threadIdx.x + blockDim.x * blockIdx.x; Registers:
  float y;                                        fast access
  y = x[i]; x[i] in global memory: 400-600 clock cycles to read
  y = y + 1.0;
  x[i] = y; x[i] in global memory: 400-600 clock cycles to write
}
```

Requirements for coalescence

Requirements for coalescence are of course more complex than the naive model shown previously. They depend on the compute capabilities of the device (1.x, 2.0). In any case, if subsequent aligned data is accessed by subsequent threads, the transaction is coalesced.

```
__global__ void simplekernel (float *x, int n) {
   int i = threadIdx.x + blockDim.x * blockIdx.x; Registers:
  float y;                                        fast access
  y = x[i]; x[i] in global memory: 400-600 clock cycles to read
  y = y + 1.0;
  x[i] = y; x[i] in global memory: 400-600 clock cycles to write
}
```

Are the transactions above coalesced ?

Requirements for coalescence

Requirements for coalescence are of course more complex than the naive model shown previously. They depend on the compute capabilities of the device (1.x, 2.0). In any case, if subsequent aligned data is accessed by subsequent threads, the transaction is coalesced.

```
__global__ void simplekernel (float *x, int n) {
   int i = threadIdx.x + blockDim.x * blockIdx.x; Registers:
  float y;                                        fast access
  y = x[i]; x[i] in global memory: 400-600 clock cycles to read
  y = y + 1.0;
  x[i] = y; x[i] in global memory: 400-600 clock cycles to write
}
```

Are the transactions above coalesced ? **YES**

Coming back to `sandbox2`...

Remember the last topic of practice `sandbox2` : each thread treated sequentially two “pixels” of the matrix.

- How was the change in performance ?
- Does that correspond to coalesced transaction ? Why ?

Reminder : access time to memory

- Shared memory: 4 clock cycles
- Global memory: 400-600 clock cycles

How do we use the shared memory ?
Qualifier `__shared__` within the kernel. Example:

```
__shared__ float local_small_array[BLOCKSIZE];
```

All threads within the same block will have access to the array `local_small_array`.
The shared memory may be filled by reading the content of a chunk of global memory (preferentially using coalescence...)

Scope of shared memory

- The scope of the shared memory is the kernel.
- You cannot declared `__shared__` memory outside of a kernel
- The shared memory within a kernel is accessible to all threads of the same block (only !)

Usage of shared memory

The shared memory can be regarded as a manual cache. You have a full control of its behavior. You fill it from the global memory using coalescence.

Then you can essentially use and reuse at will the content of the shared memory. Access to its data is as fast as access to registers (provided there is no *bank conflict*)
Usually each thread transfer one data element to the shared memory. They do this half-warp by half-warp, in no specific order.

Thread synchronization

Before we use the data in the shared memory, we must ensure that the transfer from `gmem` to `shmem` is complete.

Thread synchronization

Before we use the data in the shared memory, we must ensure that the transfer from `gmem` to `shmem` is complete.

Once a given thread has achieved its part of the transfer, it must way for all others to have done the same

Thread synchronization

Before we use the data in the shared memory, we must ensure that the transfer from `gmem` to `shmem` is complete.

Once a given thread has achieved its part of the transfer, it must way for all others to have done the same

After `gmem` to `shmem` transfer, put an execution barrier: all threads must reach it before they can proceed.

__syncthreads ()

The instructions to force all threads *within a same block* to wait for each other is `__syncthreads ()`. It is an *execution barrier*.

```
__global__ void littlekernel (float *array, ...) {
//array is on gmem
__shared__ float sharray[BLOCKSIZE];
gid = blockDim.x*blockIdx.x + threadIdx.x; global id
lid = threadIdx.x;                          local id
sharray[lid] = array[gid];     gmem to shmem transfer
__syncthreads ();                    execution barrier
//now we can use at will sharray[]...
}
```

to sum up...

- Coalescence
- Shared memory (`__shared__`)
- Synchronization (`__syncthreads ()`)

are the three major basic techniques to achieve good performance. They often work together. We shall see finer tuning techniques, but these ones are essential.

Practice : matrix convolution `convol` and `convol2`

Purpose: simple low-pass filter on a given matrix $(A_{ij})_{i,j\in[0,N-1]}$.
The output matrix B is such that:

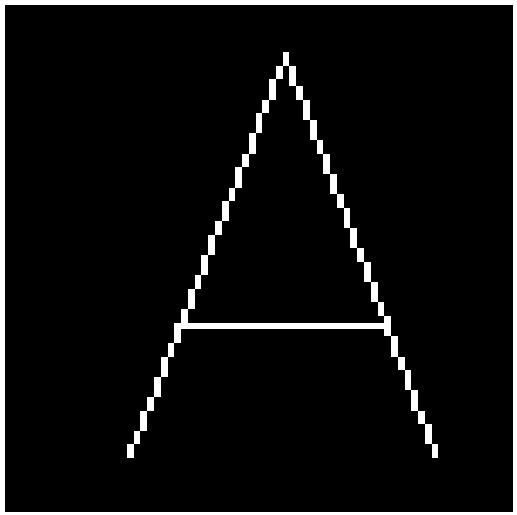
$$B_{ij} = \frac{1}{5}(\quad A_{i-1j} + A_{ij-1} + A_{ij} + A_{ij+1} + A_{i+1j} \quad) \text{ if } i > 0 \text{ and } i < N-1 \text{ and } j > 0 \text{ and } j < N-1$$
$$= A_{ij} \text{ otherwise}$$

In English : each element of the matrix is substituted by the average of itself and its 4 neighbors, except edge elements which are kept as such $\Rightarrow$ if matrix represents an image, it becomes “unfocused”

Practice : matrix convolution `convol` and `convol2`

Purpose: simple low-pass filter on a given matrix $(A_{ij})_{i,j\in[0,N-1]}$.
The output matrix B is such that:

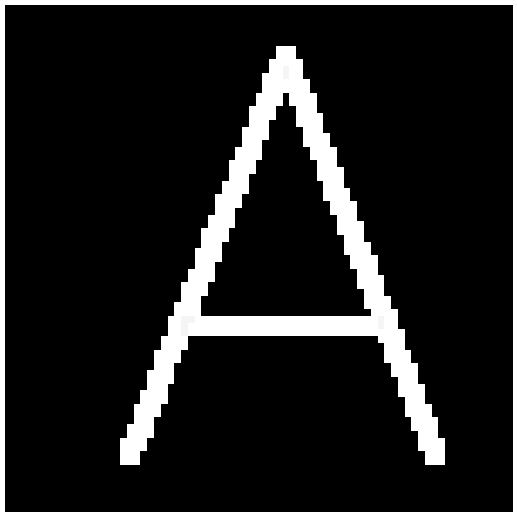
$$B_{ij} = \frac{1}{5}(\quad A_{i-1j} + A_{ij-1} + A_{ij} + A_{ij+1} + A_{i+1j} \quad) \text{ if } i > 0 \text{ and } i < N-1 \text{ and } j > 0 \text{ and } j < N-1$$
$$= A_{ij} \text{ otherwise}$$



Practice : matrix convolution `convol` and `convol2`

Purpose: simple low-pass filter on a given matrix $(A_{ij})_{i,j\in[0,N-1]}$.
The output matrix B is such that:

$$B_{ij} = \frac{1}{5}(A_{i-1j} + A_{ij-1} + A_{ij} + A_{ij+1} + A_{i+1j}) \text{ if } i > 0 \text{ and } i < N-1 \text{ and } j > 0 \text{ and } j < N-1$$
$$= A_{ij} \text{ otherwise}$$



Practice : matrix convolution `convol` and `convol2`

Purpose: simple low-pass filter on a given matrix $(A_{ij})_{i,j\in[0,N-1]}$.
The output matrix B is such that:

$$B_{ij} = \frac{1}{5}(A_{i-1j} + A_{ij-1} + A_{ij} + A_{ij+1} + A_{i+1j}) \text{ if } i > 0 \text{ and } i < N-1 \text{ and } j > 0 \text{ and } j < N-1$$
$$= A_{ij} \text{ otherwise}$$

Practice : matrix convolution `convol` and `convol2`

Purpose: simple low-pass filter on a given matrix $(A_{ij})_{i,j\in[0,N-1]}$.
The output matrix B is such that:

$$B_{ij} = \frac{1}{5}(A_{i-1j} + A_{ij-1} + A_{ij} + A_{ij+1} + A_{i+1j}) \text{ if } i > 0 \text{ and } i < N-1 \text{ and } j > 0 \text{ and } j < N-1$$
$$= A_{ij} \text{ otherwise}$$

Practice : matrix convolution `convol` and `convol2`

Purpose: simple low-pass filter on a given matrix $(A_{ij})_{i,j\in[0,N-1]}$.
The output matrix B is such that:

$$B_{ij} = \frac{1}{5}(A_{i-1j} + A_{ij-1} + A_{ij} + A_{ij+1} + A_{i+1j}) \text{ if } i > 0 \text{ and } i < N-1 \text{ and } j > 0 \text{ and } j < N-1$$
$$= A_{ij} \text{ otherwise}$$

Practice : matrix convolution `convol` and `convol2`

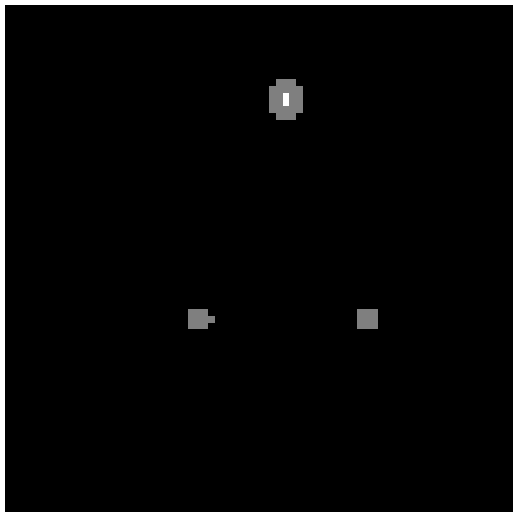
Purpose: simple low-pass filter on a given matrix $(A_{ij})_{i,j\in[0,N-1]}$.
The output matrix B is such that:

$$B_{ij} = \frac{1}{5}(A_{i-1j} + A_{ij-1} + A_{ij} + A_{ij+1} + A_{i+1j}) \text{ if } i > 0 \text{ and } i < N-1 \text{ and } j > 0 \text{ and } j < N-1$$
$$= A_{ij} \text{ otherwise}$$

Practice : matrix convolution `convol` and `convol2`

Purpose: simple low-pass filter on a given matrix $(A_{ij})_{i,j\in[0,N-1]}$.
The output matrix B is such that:

$$B_{ij} = \frac{1}{5}(A_{i-1j} + A_{ij-1} + A_{ij} + A_{ij+1} + A_{i+1j}) \text{ if } i > 0 \text{ and } i < N-1 \text{ and } j > 0 \text{ and } j < N-1$$
$$= A_{ij} \text{ otherwise}$$



Convolution: the slow and the fast

- We tackle this practice in two different ways: without using the shared memory (first part, easier, `convol`), then using the shared memory (second part, a bit more difficult, `convol2`).
- As usual, the convolution is also performed on the CPU for comparison purposes.
- Read the CPU code (`main.c`) carefully. Make sure you understand the maths of indexes.
- This practice already includes time measurements to get an idea of the performance.

`convol` - Simple convolution on GPU

In this first part (`convol`), you will have to edit `convol.cu` in order to:

- Find the `i` index of the thread (we have here a 2D topology). You can use the example of the `j` index.
- Find the offset of the element considered (when you regard `gpu_a` as a 1D array)
- Find the missing indexes in the convolution formula.
- Find the number of blocks... Nothing new with respect to previous practices. Use a formula that is general (*i.e.* that works also when the matrix size in *x* or *y* is not a multiple of `BLOCK_X` or `BLOCK_Y`.)
- Find the missing argument in the kernel invocation

convol2 - More sophisticated convolution on GPU

In the second part (convol2), you will have to edit convol.cu in order to:

- Find the local indexes of a thread in the shared array
- Answer verbally some questions asked in the source file
- Fill up some partial or missing statements.
- Is there any difference in main.c with respect to the previous practice convol ? And in the wrapper function gpu_convol () ?

One `convol` and `convol2` work...

...we can try and understand the results of the benchmarking.

- For each block, how many coalesced reads to we have in `convol` ? How many uncoalesced reads ? How many coalesced writes ?
- Deduce how many `gmem` transactions we have in `convol`, per block.
- Same question for `convol2`.
- How does the ratio of number of `gmem` transactions compare to the ratio of execution time ? Conclusion ?

One `convol` and `convol2` work...

...we can try and understand the results of the benchmarking.

- For each block, how many coalesced reads to we have in `convol` ? How many uncoalesced reads ? How many coalesced writes ?
- Deduce how many `gmem` transactions we have in `convol`, per block.
- Same question for `convol2`.
- How does the ratio of number of `gmem` transactions compare to the ratio of execution time ? Conclusion ?

Our kernel is memory-bound