Array transposition in CUDA shared memory

Mike Giles

February 19, 2014

Abstract

This short note is inspired by some code written by Jeremy Appleyard for the transposition of data through shared memory. I had some difficulty getting my head around it, and decided it would be helpful to have a few figures to explain it. I've also extended it slightly to cover more general cases.

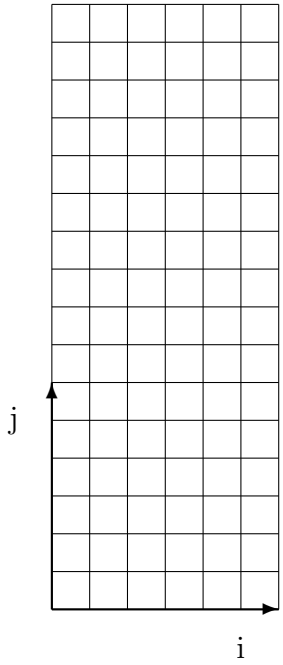

Figure 1: Illustration of array to be written into, and read from

1 Objective

As illustrated in the figure above we want to work with a shared memory array which is mathematically of width I, and height 32 (equal to the warp size).

We want to effectively transpose some data by writing into it with the threads in the thread block filling it row-wise, working across the first row, then the second, and so on in ascending order of $i + jI$, and then (after a synchronisation) reading data out of it column-wise, working up the first column, then the second and so in in ascending order of $j + 32i$. Or, vice versa, we might want to fill it by columns, and then read it out by rows.

The main application for this is for loading in (or storing) data which is stored as a Array-of-Structs, each of size I. To achieve a coalesced read from device memory using a single warp, the threads can load in contiguous vectors from device memory and fill the shared memory array by rows. Then they can read it back into registers from columns so that each thread gets its required struct data. The process would be reversed for writing back to device memory. In both cases, the array index in device memory would correspond to $i+jI$.

The second application arises in the ADI solvers which Jeremy and I are working on. Here, for part of the calculation, in order to maximise coalesced memory transfers it is efficient for threads to work on the array row-wise initially, but there is then a middle section in which it is necessary to work on columns with a separate warp for each column, before finally reverting to the original thread mapping for the final stage.

The challenge is to come up with a mapping $(i, j) \to k$ to an index k in the linear shared memory array so that there are no memory bank conflicts when accessing the data in either direction.

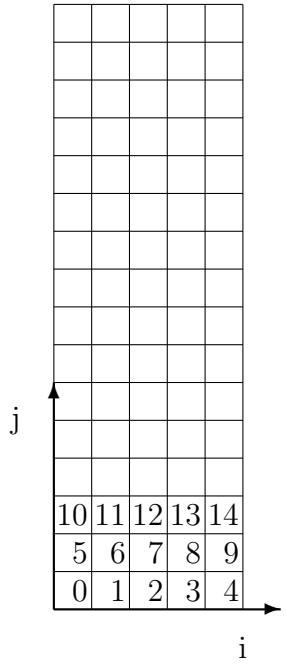

Figure 2: Shared memory indices when $I = 5$.

2 I odd

When I is odd we can define

$$k = i + j\, I.$$

This naturally gives no bank conflicts when reading row-wise, since each warp gets 32 contiguous addresses and current NVIDIA GPUs have 32 shared memory banks.

Furthermore, there are no bank conflicts in each column, because $j = 32$ is the smallest strictly positive integer such that

$$j\, I \bmod 32 = 0$$

which would lead to a bank conflict with the element $j = 0$.

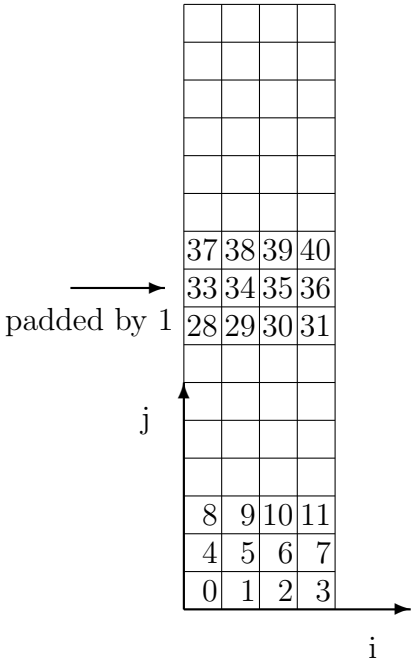

Figure 3: Shared memory indices when $I = 4$.

3 I a power of 2

When I is a power of 2, then the definition

$$k = i + j\, I$$

would lead to bank conflicts along each column. The first bank conflict is when $i = 0, j = 8, k = 32$. This suggests the idea of padding by 1 after every 32 elements, giving the mapping

$$k = i + j\, I + (i + jI)/32$$

where the division is interpreted in the integer sense (i.e. discarding the remainder).

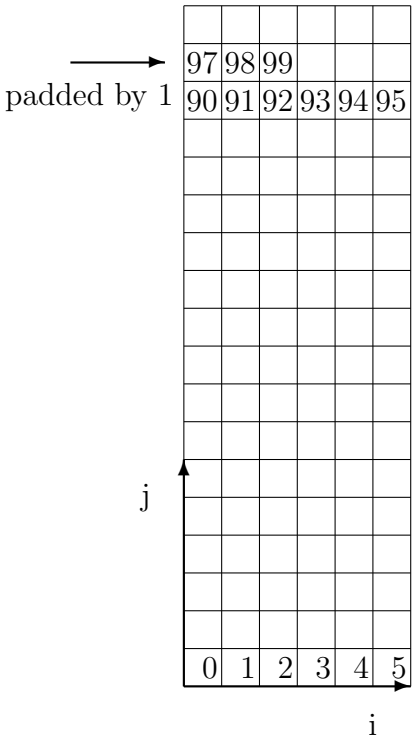

Figure 4: Shared memory indices when $I = 6$.

4 General I

Having handled the two extreme cases, now we consider the general case in which $I = P\,F$, where P is a power of 2, and F is odd. In this case

$$k = i + j\,I$$

leads to the first bank conflict when $i = 0$ and

$$j\,I \bmod 32 = 0.$$

If $P \leq 32$ then this implies

$$j\,F \bmod (32/P) = 0,$$

which happens first when $j = 32/P$, and hence $jI = 32F$. Thus, the padded definition to avoid conflicts is

$$k = i + j\,I + (i + j\,I)/(32F).$$

(Note: when $I = F$, then $(i + j\,I)/(32F) = 0$ which correctly gives us back the unpadded version.)

Alternatively, if $P > 32$ then the first bank conflict occurs when $j = 1$, and an appropriate padding is

$$k = i + j\,I + j.$$

Implementation notes

The simplest thing is to define the (i, j) pairs for loading and storing, and then compute the k for each.

At worst, the padding increases the shared memory requirements by approximately 3%.

The computation of k requires 2 additional integer operations, a bit-shift and an addition. An alternative, when I is even, is to use the mapping

$$k = i + j\,(I+1).$$

This avoids the 2 additional integer operations, but at the expense of using more shared memory.