High Performance Computing

GPU Programming with CUDA: Memory Hierarchy and Streams

Στολτίδης Αλέξανδρος, Κουτσούκης Νικόλαος

January 6, 2022

1 Memory Optimizations

To optimize the code even further, we can take advantage of the GPU's memory hierarchy. Data that it's value is never changed can be stored into the constant memory where read operations are much faster. The constant memory however is too small to store large chunks of data and thus, only the filter can stored there. We set the constant memory's size to it's maximum acceptable value and the filter can have a maximum length of 65536 Bytes (8192 Doubles). To store more data and still maintain a low read operation latency we can use the GPU's available shared memory. To understand how the shared memory works we must first understand how the L1 cache is structured. The L1 cache is divided into two parts. The software controlled part is called shared memory while the rest of the L1 cache is hardware controlled. By pulling data from global memory into shared memory the read latency is minimized. An image of the GPU's hierarchy can be seen bellow.

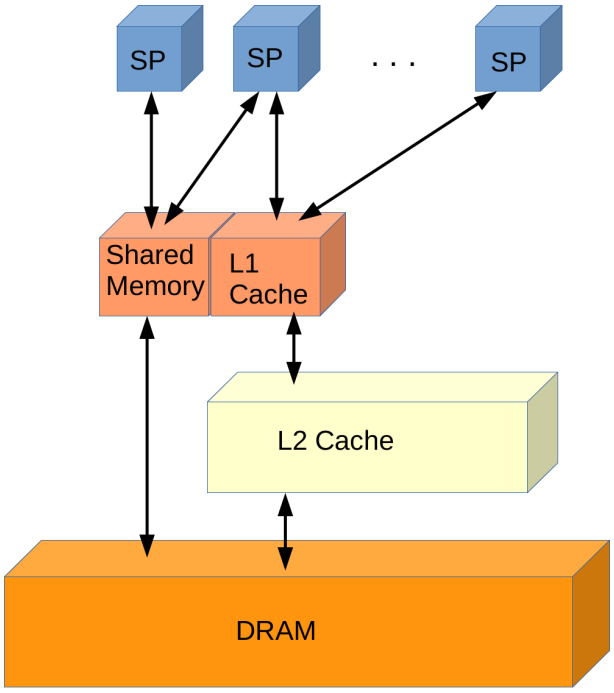

Figure 1: Fermi Memory Architecture

1.1 Padding in Shared Memory

The original image does not need to be padded since the padding is added directly at the shared memory. Different padding strategies are applied in row and column convolutions.

1.1.1 Row Convolution Padding

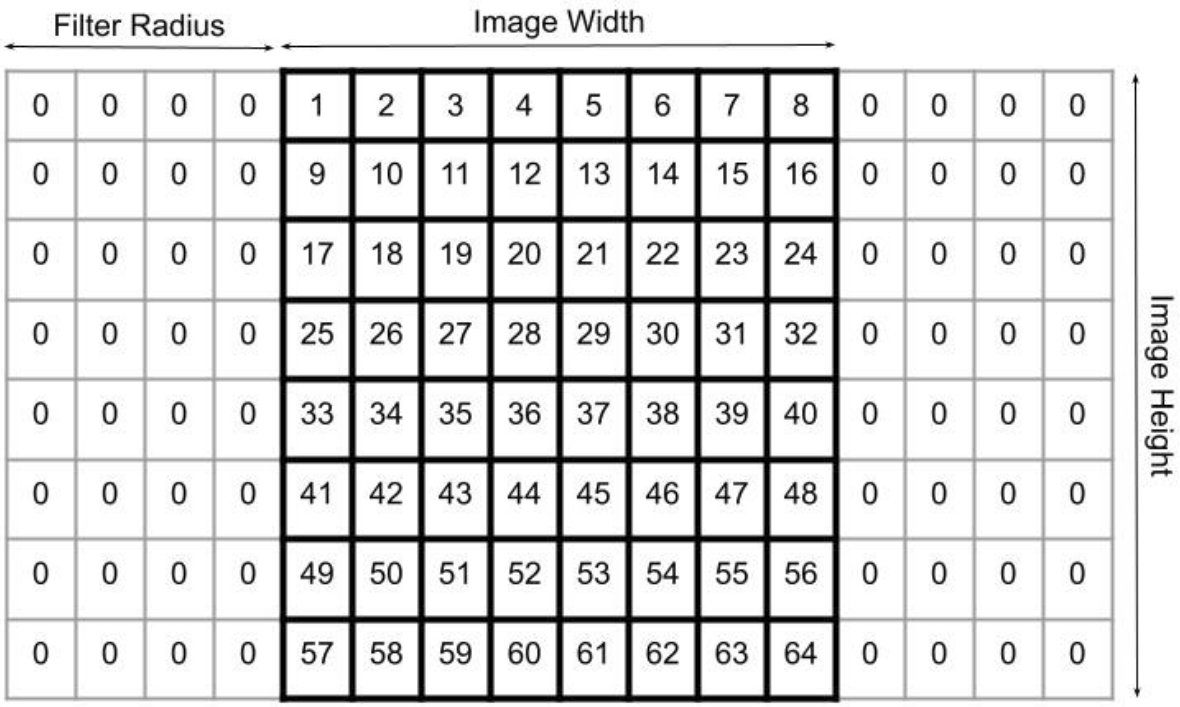

Figure 2: Padded with Zeroes only Left and Right Sides

When the filter is applied, on an edge pixels of the image, the conditional statements are avoided since multiplications with zero cells of padding are allowed and do not alter the final result. Since row convolution only multiplies the pixels that are located left and right of the actual pixel we do not need to add padding on the top or the bottom sides of the image.

1.1.2 Column Convolution Padding

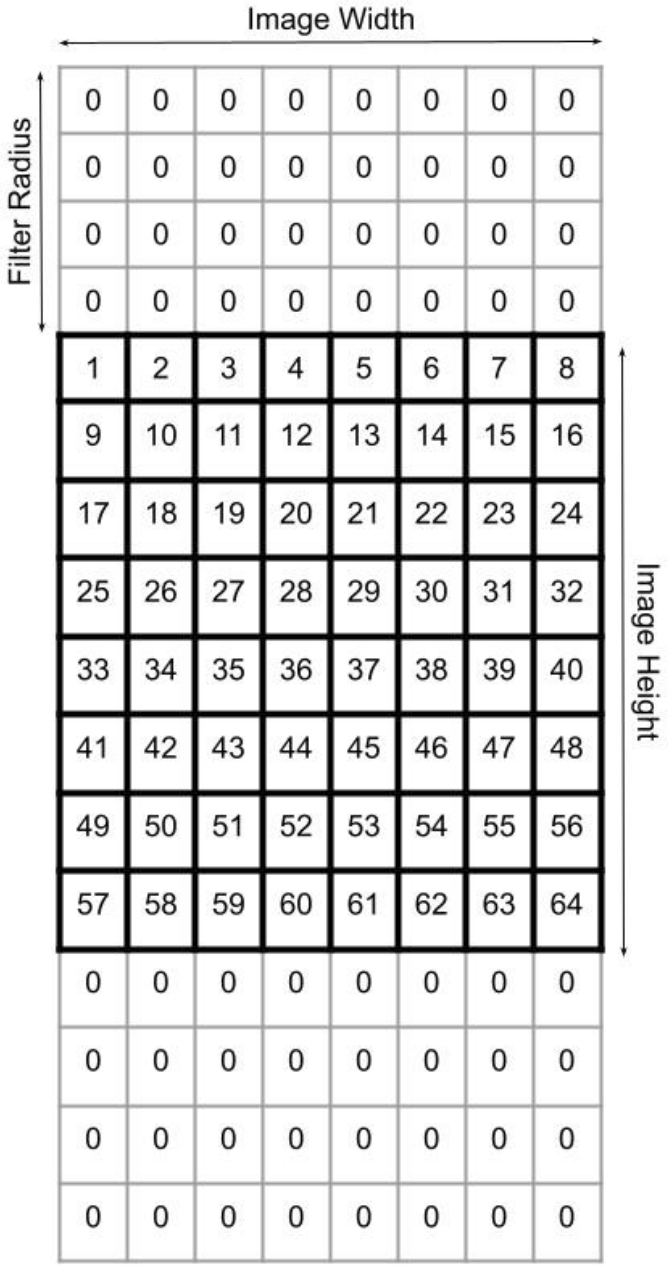

Figure 3: Padded with Zeroes only Top and Bottom Sides

Column convolution only multiplies the pixels that are located above and bellow the actual pixel with the filter and thus we do not need to add padding on the left or the right sides of the image.

1.2 Unified GPU-CPU Memory

Some additional optimizations that can be performed is switching to a unified memory model. As we already know the CPU and GPU memories are totally different and data must be copied from main memory to the global memory before any kernel launch. When the kernel finally finishes it's execution the data must be copied back to main memory to be used by the CPU. These memory operations take a lot of time since they can not executed concurrently with the actual kernel computation. To avoid this problem we take advantage of the DMA module that the motherboard provides. With this module the GPU gets data on demand directly from main memory when the actual computation needs it. This saves a lot more time since data transfer operations are executed only when necessary alongside with the actual GPU computation.

1.3 Measuring Time

As expected when the filter length is increased the time it takes for the actual computation also increases. This can be attributed to the fact that more padding in the image is added and thus more multiplications in the convolution process are performed. We bench-marked our code with different filter lengths and we decided that the best filter radius that better represents the average time that the convolution takes is when the filter radius is set to 32. For a filter with a radius of 32 we calculated the *computation/memory* and the graph bellow is extracted.

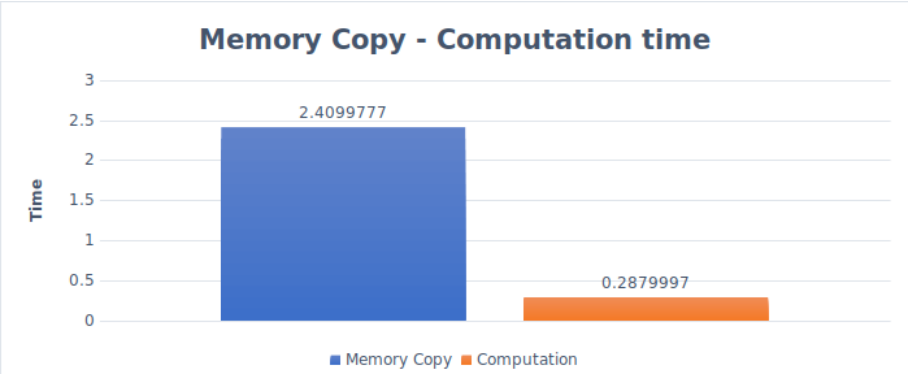

Figure 4: Computation / Memory = 0.12

We also wanted, just for fun, to compare by how much the unified memory model outperforms the manual data transfers and the result can be also seen bellow.

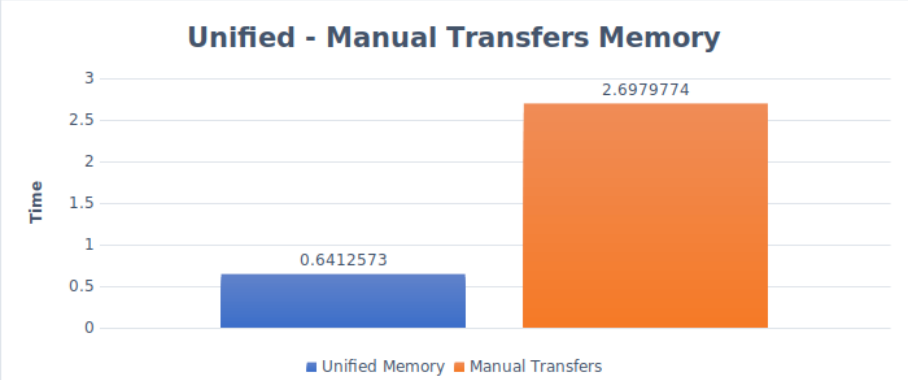

Figure 5: UM is 4.2 Times Faster

2 Image Tiling

To support even larger images than before we can split the image into multiple parts, tiles, and launch multiple kernels to process each tile separately. After a kernel finishes it sets the correct pixels on the final image which is the same size as the original image. To solve the tiling problem we have to make a few assumptions though. At first we only handle images that their width and height is a multiple of 32 ($width = height = 32, 64, 128...$). In addition, the total amount of tiles that the image can be split in is $horizontal = width/32$ provided that $width \mod 32 = 0$ and $vertical = height/32$ provided that $height \mod 32 = 0$ for horizontal and vertical tiles respectively. Optimization methods like padding in shared memory, using constant memory to store the filter and utilizing the DMA to minimize data transfers are also used in tiling.

2.1 Tiling

In order to split the image into multiple tiles we must first allocate memory for each tile. Since all kernels use the default stream they all run sequentially from the perspective of the CPU and thus only one tile can be allocated and later be reused by all kernels. Each tile is consisted of two parts. The first part is the actual image that the tile is extracted from. The second part is the padding added to the tile that is used to avoid divergence. If the tile is located on the edges of the image the padding from the edge side should be initialized with zeroes, else it is initialized with the neighbouring pixel's values. An example of tiling and padding can be seen below.

Figure 6: Row Convolution Tiling with 4 Tiles

The above example shows how each tile is initialized for the row convolution. When the actual convolution in the GPU begins the tile's data is transferred to shared memory and the actual computation begins. In row convolution the padding is added to the left and to the right edges of each tile (like the image above). On the other side, in column convolution the padding is added above and below the tile. Divergence is completely avoided since all 32 threads of each warp always execute the same instructions. The execution however can still be optimized since only one kernel is active on the GPU at a time. To avoid this kernel serialization we take advantage of the multiple streams that the GPU provides.

3 Streams

Since each kernel execution is asynchronous our first priority is to transfer data correctly from main memory to the GPU. To avoid data inconsistency each tile must take its own memory space. To achieve this we create an array of tiles. Each kernel uses its own stream and tile to calculate the actual convolution. When all kernels finish the device is finally synchronized.

3.1 Default Only vs Multiple Streams

Tiling the image into 32 Horizontal and 32 Vertical tiles and launching 32 Kernels with 32 streams leads to the results below.

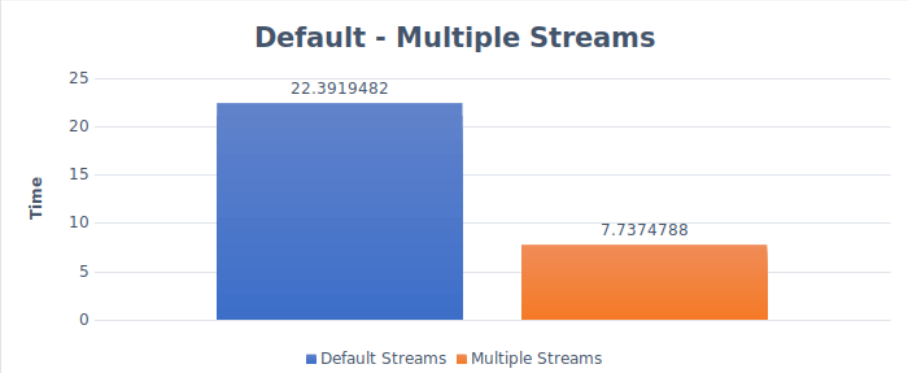

Figure 7: Multiple Streams are 2.88 Times Faster

4 Tiling Disadvantage

Since each tile is handled by its own kernel the more tiles we have the greater the number of kernel actually is. For a small number of kernels we observe that the execution is fast since a lot more time is spent on actual computation and lot less in spawning kernels. When a lot more kernels that are necessary are spawned the execution times suffer immensely. This can be attributed to the fact that there is a great kernel launch overhead.

5 Final Limiting Factor

Now the limiting factor is not actually the GPU itself. The problem lies in the main memory and especially the amount of memory that the machine has available. The maximum image size we can now support is 32768x32768 pixels before we run out of RAM.