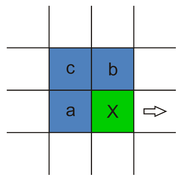
Using GPUs for Lossless-JPEG real-time image decompression of high-throughput microscope data

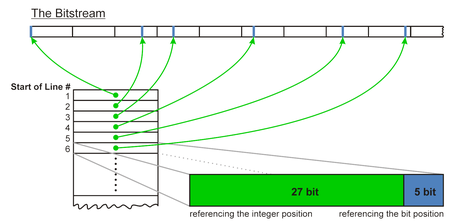
Timo Bernard, Guillermo Marcus, Markus Gipp, Reinhard Männer
Dpt. Of Computer Science V, ZITI, University of Heidelberg

The Algorithm



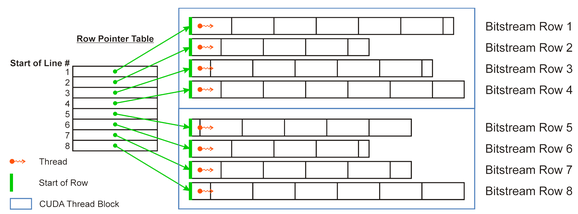
The Lossless JPEG (L-JPEG) compression process pixels in standard (zig-zag) order, and for each pixel, produces a residual based in a chosen codec function of its neighbor pixels. Then, the residual is encoded following a fixed table.
In the serial version, the decompressor walks the bitstream in order, decoding the residual value and reconstructing the pixel value, using the reconstructed values of the neighbors. This creates a dependency chain for the decoding of any pixel.

Row Pointer Table



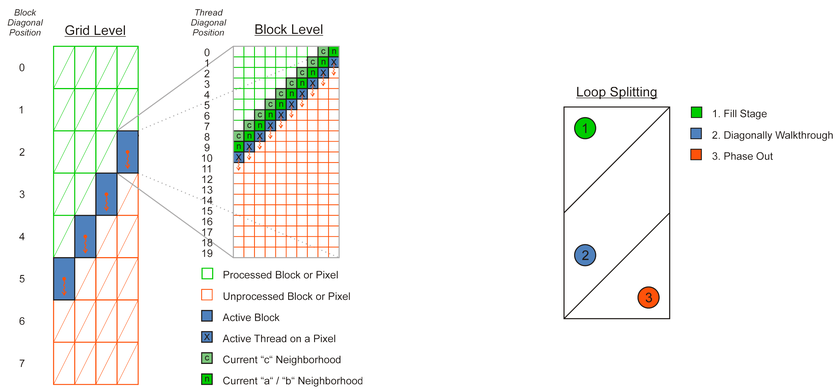
The row pointer table contains the starting position of every row in the image. Each entry encodes the position as a 27-bit integer plus 5-bit offsets. The integer position corresponds to the 32-bit word aligned position in the bitstream array, and the bit position is the bit offset inside this word.
This array serves as a Look-Up Table (LUT) for fast access to the row of interest, without the need to parse the bitstream serially.

First Kernel: Decoding the residual value



This kernel reads the residual from the bitstream and decodes it as integer values.
In order to do it efficiently, every CUDA thread process in parallel one row of the reconstructed image. This is possible because the residual value is encoded with a fixed table, which does not depends in the value of previous pixels. Also, being the image scanned in a zig-zag pattern, values for a row are stored in a consecutive subset of the bitstream.
Each thread walks the bitstream subset serially, decodes the residual value and writes it as an integer into the image buffer for further processing by kernel 2.

Second Kernel: Reconstructing the pixels



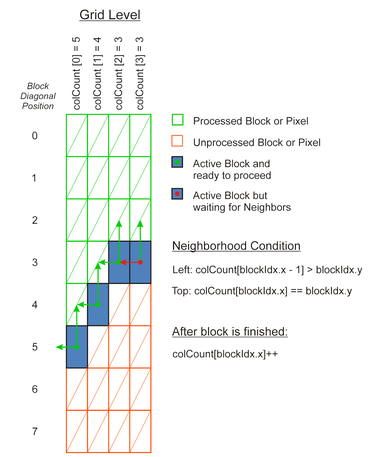
The second kernel performs the reconstruction of the pixel values, based on the neighboring pixels and the residual values of kernel 1.

In order to parallelize this task, the target image is divided in rectangular blocks, and they correspond to CUDA blocks. The kernel is launched several times, one launch per block diagonal. In this way, we ensure block with neighboring pixels are processed in the proper order, without the need for synchronization, as Kernel launches occur one after the other.

Every CUDA block process in itself pixels in a similar fashion, following a diagonal. The two previous diagonals (neighbors c and n in the diagrams) are cached in shared memory for improved performance. In practice, every column in the block belongs to a thread.

In order to further increase the performance, the loop is splat in 3 sections, to avoid unnecessary checks of border conditions during the diagonal walkthrough (point 2).

Alternative: Atomic units?



One alternative algorithm is to run one block per column, and synchronize their dependency via an atomic counter per column (counting the finished row block). Then, each block can wait until the neighboring block has finished processing the required row. We discover this solution is as slow as processing the blocks serially, which means there is only one atomic unit in the memory controller and therefore cannot take advantage of multiple atomic operations over non-overlapping counters.

Results

- Speedups of 15-30 times, compared to the host.
- Able to process data at high data rate.
- Bitstream is not modified, standard L-JPEG.
- Row table pointer is a byproduct of the FPGA compression, no extra cost.