Implementation of 2D Convolution on FPGA, GPU and CPU

Ben Cope
Department of Electrical & Electronic Engineering, Imperial College London
benjamin.cope@imperial.ac.uk

Abstract

The 2D convolution algorithm is a memory intensive algorithm with a regular access structure. Implementation on an FPGA can exploit data streaming and pipelining. The GPU is unable to hold onto previously accessed data, this report exemplifies this limitation.

Designs for implementations on FPGAs, GPUs and the CPU are shown and results of their performance analysed. We find the FPGA to have a more consistent and higher throughput than both the GPU and CPU.

1 Introduction

The 2D convolution algorithm addresses one of the key drawbacks of using GPUs for video processing. That is their memory usage is inefficient. They have no internal storage mechanism to hold onto previously accessed pixel data for later use in processing another pixel.

Memory usage for FPGAs is more efficient. Previously accessed data from external memory can be held locally in on-chip memory. The flexibility of FPGAs can be exploited to implement an arbitrary data path for efficient streaming of data.

Cache and internal registers in the CPU can be exploited to reduce the required number of external memory accesses. The rigid instruction set of CPUs means they are less flexible in how the data can be used.

This report will describe how the 2D convolution algorithm can be implemented on GPUs, FPGAs and CPUs. These will form the first three sections. Following this are the results and an analysis of the figures.

2 GPU Implementation

For each of the three implementations we can divide the 2D convolution task into two main sections, shown in figure 1. The first is the retrieval of data (i.e. that covered by the mask) this is from some external video memory. The second is processing that data and outputting.

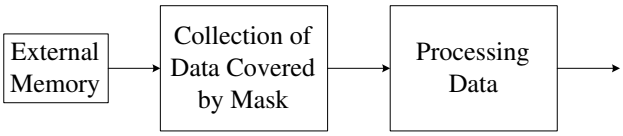

Figure 1. High Level Block diagram for all Implementations

2.1 Data Fetch

Each frame is bound as a 2D texture. This can then be accessed by the GPU as a texture lookup. For each pixel a lookup is required for each of the locations it desires. A GPU is capable of performing a texture lookup in the same clock cycle as simple arithmetic calculations. In the compiled assembly code these lookups are therefore distributed within the first processing steps. An example lookup command is shown below:

$$float3\ pColor\_0\_p1 = tex2D(testTexture,\\ float2(IN.decalCoords.x, IN.decalCoords.y)).rgb;$$

This accesses the current pixel being processed. Offsets on the IN coordinates are used to access locations around the current pixel. If this falls outside of the pixel window the value is determined by the current memory access mode. This can for example mirror the texture or repeat the edge values for those outside of the image boundary.

2.2 Processing Data

The example of 2D convolution size 8×8 is used to show how to optimally process data. It is remembered that GPUs are optimal when processing data in matrices or as vectors. Figure 2 shows an 8×8 array of data to be processed. The bold lines indicate how it is divided. For different sized convolutions the array is divided into arrays of size 4×4 and where necessary 3×3 or smaller.

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7

Figure 2. How an array of 8×8 data items is divided for processing in the GPU

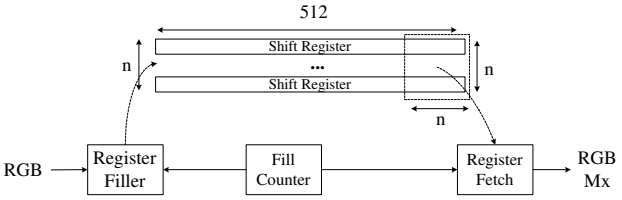

Figure 3. How data fetch is implemented in the FPGA design

Each of these smaller arrays is then multiplied by a constant non-zero array of the same size. Additions of vectors are performed, avoiding scalar addition.

Reordering of data has no cycle cost therefore data can be arranged arbitrarily for optimal additions and multiplications.

3 FPGA Implementation

This has the same high level block diagram as for the GPU, see figure 1. The implementation of each therefore shall be considered as above.

3.1 Data Fetch

This is where the main advantage of the FPGA over the GPU becomes evident. That is that the FPGA can hold onto the previously accessed data for later processing. This comes at the cost of extra complexity in the design and the requirement for on-chip storage. These factors increase with the size of convolution. Figure 3 shows how the data is stored for a filter size n.

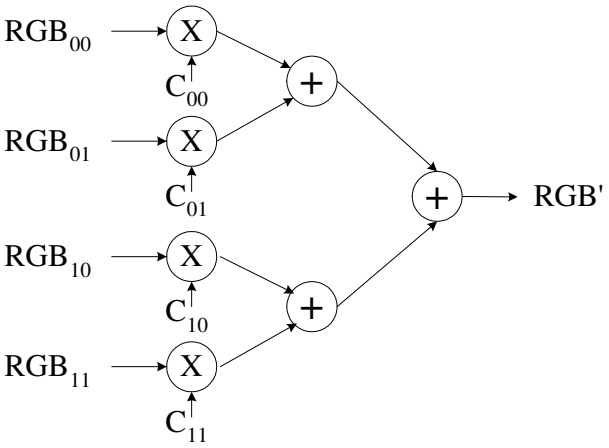

Figure 4. Example of how data is processed in FPGA design for convolution size 2×2

The register filler block takes the current RGB input and feeds this into the current fill register indicated by fill counter. This RGB data is subsequently moved through the shift register. After a whole row is input the filler begins on the next shift register, this is circular for the entire frame.

If a current shift register isn't being filled then its data items are rotated in a circular buffer manner. This is to ensure the correct alignment of data items. If a shift register is the one currently being filled then pixels 'drop off' the end of the register.

Register fetch always takes the $n \times n$ pixel data values from the bottom of the shift register. These are ordered according to the current output of fill counter. This matrix is then passed onto the processing block.

3.2 Processing Data

For this block we again take an example of one size of convolution which scales for others. The choice is size 2×2, the implementation can be seen in figure 4. First each of the elements of the array are multiplied by their respective coefficients. An adder tree is then used to sum up the results with the minimum possible delay, at a cost of area.

4 CPU Implementation

Unlike the FPGA and GPU implementations care must be taken about the edge of a frame. Min and max operations are used to ensure that a memory address doesn't exceed the edges of a frame, hence leave the address range. However one can execute different portions of the code depending on current pixel location, to reduce this overhead. The general

Size	6800 Ultra	6800 GT	6600 GT	5900 Ultra	5800 Ultra	5200 Ultra	3.0 GHz Pentium 4
2×2	1070	933	667	150	62.5	25	14
3×3	278	243	174	17	10.7	4.3	9.7
4×4	110	96.5	69	4.3	3.2	1.3	6.8
5×5	54	47.5	34	2.6	1.7	0.7	5.1
6×6	33	29	21	3	2.3	0.9	3.5
7×7	22	19	14	1.5	1	0.4	2.6
8×8	11	10	7.1	1.1	0.8	0.3	2.1
9×9	9	8	5.6	1.2	1	0.4	1.6
10×10	5.5	5	3.5	0.7	0.5	0.2	1.3
11×11	4.7	4	3	0.6	0.4	0.16	1.2

Table 1. Throughput (MP/s) of 2D convolution implemented on varying GPUs and a CPU

structure is again similar to that for the FPGA and GPU shown in figure 1.

4.1 Data Fetch

An array is created for the $n \times n$ data items to be processed in the next section. Data fetching is performed in two main groups. First, those not around the edge of the image. Second, those around the edge of the image and requiring special attention.

The first set of pixels are processed as follows. Pixels in the first $n \times n$ region are loaded into the array. In the next loop pixels are shifted along to the left one location and new pixels added on the right-hand side of the array. At the end of a row all the pixels for the first array region of the next row are looked up again. It is expected that the memory cache will hold some of this data reducing the time for memory access. This process is repeated over all of the image.

Secondly the edges of the image are considered. For this we introduce range limiting on the pixel addressing. That is if at the edge of the image the pixel on that edge will be repeated for those in the region outside of the edge boundary.

4.2 Processing Data

Processing for the CPU implementation is the most simply implemented of the three hardware types. The following code is used:

$//Filter$
$R\_sum = 0;$
$G\_sum = 0;$
$B\_sum = 0;$

$for(int\ pos = 0;\ pos < dim * dim;\ pos++)$
$\{$
$R\_sum += R[pos] * filter[pos];$
$G\_sum += G[pos] * filter[pos];$
$B\_sum += B[pos] * filter[pos];$
$\}$

4.3 Considerations if using SSE Instructions

In an SSE implementation four pixels are processed in parallel. This means an array size of $n \times n \times 4$. This is passed to the processing function. The increment in the x direction for lookup is altered to four from one.

The processing is slightly more complicated. It is required to multiply each of the vectors of four elements in the array with the corresponding constant. The results are then added, this part of the code needed to be changes for each value of n.

5 Results

A GeForce 6800 GT was used to test the 2D convolution algorithm. The speed of this device along with other GPUs was predicted from the NVIDIA nvshaderperf tool. This assumes 1 cycle texture lookup assuming that there are no memory clashes.

The FPGA design was written in VHDL and synthesis / place and routing was done in Xilinx Project Navigator. The in-built synthesis / PAR tool XST were used for this. ModelSim was used to simulate the behaviour of the filter for test.

A 3.0GHz Pentium 4 was used as the CPU implementation of the 2D convolution filter. The 'read time stamp counter' assembly instruction was used to calculate the number of cycles taken for the CPU.

The results for the GPU and FPGA implementations will now be presented alongside the results for the CPU.

Size	Spartan 3	Virtex II-Pro	3.0GHz Pentium 4
2×2	190	221	14
3×3	139	202	9.7
4×4	120	182	6.8
5×5	112	162	5.1
6×6	107	152	3.5
7×7	90	110	2.6
8×8	65	80	2.1
9×9	73	61	1.6
10×10	45	52	1.3
11×11	23	48	1.2

Table 2. Throughput (MP/s) of 2D convolution implemented on two FPGAs and a CPU

5.1 GPU Results versus the CPU

The results for varying GPUs, compared to the CPU, can be seen in table 1. It is seen that for the new GPU architectures on the left hand side of the table their throughput exceeds that of the CPU for all filter sizes. Their improved memory access rate over older generations facilitates this.

For the three older architectures (right hand size of table 1 throughput rate drops below the CPU's at small filter sizes (3×3 or 4×4). It is also noted that their throughput rate is not always falling but sometimes rises, for example from size 5×5 to size 6×6. This is due to the efficiency of their implementation of vector versus scalar multiplication. In the 5×5 case a larger number of smaller vectors / arrays is required slowing down the processing.

5.2 FPGA Results versus the CPU

Notice that the throughput rate of the FPGAs drops off however this isn't as sharp a decrease as seen with the GPU. This is due to the availability of arbitrary pipelining and parallelism in the FPGA design. The reason that the throughput drops off is due to increased complexity leading to an increase in fan outs and the routing length of the critical path.

6 Conclusion

The implementation of 2D convolution on GPUs, FPGAs and the CPU has been shown. Following this a presentation and brief discussion of the results.

It was seen that the throughput rate for the GPU dropped off sharply. This is due to their inefficient memory usage with a data fetch required for all mask elements for each pixel processed. The older GPUs were seen to have a lower throughput than the CPU for convolution size greater than 4×4. The FPGA throughput rate dropped off less sharply. This is due to its exploitation of pipelining and parallelism. Its performance dropped off due to increased complexity which in turn leads to longer critical path(s) and fan outs.