FPGA based Volume Ray-Casting

David Dolman MEng. MIET (david.dolman@alpha-data.com)

November 10, 2017

1 Introduction

This document accompanies Alpha Data's SC2017 FPGA based Volume Ray-Casting demonstration. This demonstration shows how a compute-intensive algorithm can be implemented on a FPGA card with a PCI Express interface, utilising Vivado HLS and Alpha Data's ADB3 PCIe Bridge. The presented system is shown to be scalable and power efficient.

The FPGA platform is built from a combination of a top-level IP Integrator (IPI) system, IP cores for on/off-chip data flow and board-specific host communications. These are combined with the ray-casting IP cores written in C++ and synthesised with Xilinx's Vivado HLS tool. Some important aspects of these IP are discussed.

At a high level, the algorithm has three phases:

1. Uploading data and splitting the task.
2. Executing the rendering algorithm.
3. Downloading and reassembling the results.

During the execution phase, each pixel of the output can be calculated independently of any of the other pixels; this makes scalability straightforward by splitting up the task (decomposition) and distributing the pieces across multiple execution units. It should be noted that the structure of this demonstration FPGA design is suitable for any other compute-intensive algorithm that permits the task to be decomposed into independent work items. Figure 1 shows an example of the output.

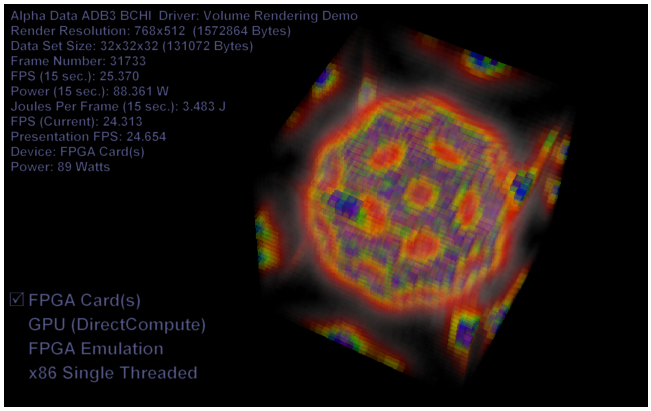

Figure 1: Demo running on a ADM-PCIE-KU3

This paper explores how the overall achieved performance, in terms of iterations per second, scales with the number of FPGA cards available. Results show improved energy efficiency for the FPGA implementation compared to CPU alternatives. Scaling by increasing the FPGA to CPU ratio within the node results in further energy efficiency improvements at the node level.

In the current version of the demonstration, the limit on the number of FPGA cards that can be used is determined by the number of PCI Express slots in the host system, but in principle, the task could be decomposed over any number of FPGA cards with heterogeneous interconnect (such as Ethernet). It is expected, however, that several bottlenecks will place practical limits on the scalability of the system as a whole, and this may be investigated in future work.

2 Architecture

Execution of Volume Ray-Casting is split into three phases:

1. Uploading the volumetric data set to the device and setting up rendering parameters.
2. Executing the rendering algorithm, using "C to gates" IP generated by Vivado HLS.
3. Downloading the results (the rendered image) from the device.

From an academic viewpoint, phases 1 and 3 are not particularly interesting and represent nothing new. Phase 2, however is the focus of this paper. The pipeline for rendering is made up of a set of components, each written in C++ and synthesised to a netlist using the Vivado HLS tool. These components are connected together, along with an instance of the ADB3 PCIe Bridge (also known as the BCHI; Board Control and Host Interface), using the plumbing provided by Vivado's Block Diagram editor, to create the complete rendering architecture seen in figure 2.

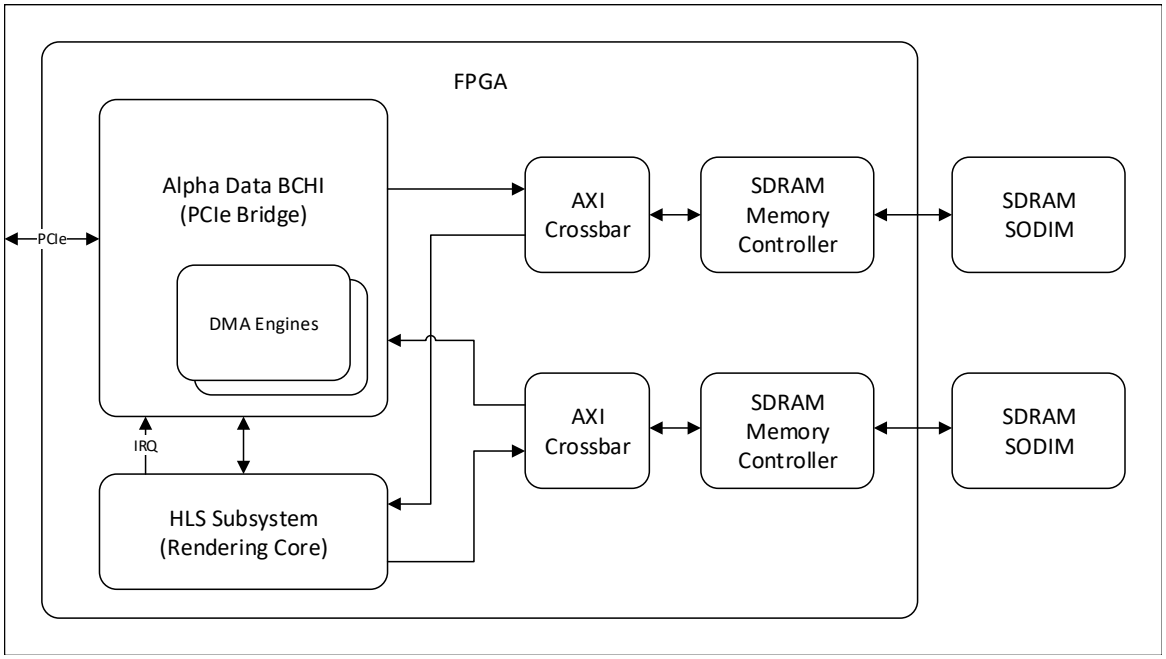

Figure 2: Simplified top-level of the FPGA design

The FPGA card chosen for this example is the ADM-PCIE-KU3, which has two SODIMM slots that provide off-chip memory for the FPGA. A memory controller is required for each SODIMM. Crossbars enable shared access to the memory, by both (a) the host system (for uploading and downloading data and results), and (b) the HLS Subsystem which executes the rendering algorithm.

In each FPGA card, the host interface is provided by the BCHI. This IP provides a PCI Express endpoint containing up to four DMA engines, and also permits the CPU to read and write registers within the FPGA that control the HLS Subsystem. A facility to interrupt the host is also provided by the BCHI; this is used by the HLS Subsystem to notify the host of completion of a work item.

3 Rendering Algorithm

The Volume Ray-Casting algorithm used in this example is as follows:

For each frame to be rendered, each pixel of the output image is cast as a ray from the eye point through the location of the pixel (on the screen). As the ray is projected from the pixel's location, an initial culling calculation is performed to determine whether or not the ray intersects the dataset's volume (a cuboid encompassing the 3-d space represented by the dataset). If the result is negative, the output pixel is set to the background colour and does not require further processing.

The volumetric data set (3-d array of voxels) can be represented as three stacks of 2-d textured planes: XY, XZ and YZ, as opposed to considering the volume as a three-dimensional set of voxels. This representation is key to how the algorithm works.

Rays that pass the initial culling test (see above) are cast against the three stacks of planes, ordering them such that the first check for each plane stack will return the collision closest to the screen (resulting in the shortest ray). Collisions are iterated on for each stack until the first collision in each stack is detected within bounds of the volume (remembering the planes are infinite in size). Comparing the results of collisions for each plane stack, the earliest one is selected as the next intersection.

Once a particular 2-d plane and a point within that 2-d plane has been determined, the point is then converted into the coordinates of 3-d voxel in the volume data set. The memory containing the volumetric data is then accessed to read the colour value of the voxel. This colour value is blended with the current colour of the ray. If the voxel has a transparent colour, the ray will continue its journey. Rays that hit opaque voxels, or whose an alpha value becomes such that no deeper penetration of the volume would be visible, are written out to the frame buffer after having the colour value updated to reflect the final collision that they encounter.

Rays that continue deeper into the sets of planes are processed further, against the three stacks of planes, until a termination condition is reached: (i) the journey ends due to an opaque collision, (ii) further collisions do not have a visible outcome, or (iii) the ray exits the volume completely. When this condition is reached, no further processing of a ray is required, and its colour value is written out to the frame buffer.

When all pixels have been computed (i.e. all rays have been cast and terminated), the frame is ready for download and presentation (viewing).

4 Rendering Implementation

The description of the rendering algorithm above suggests that any implementation will be arithmetic-intensive and require random-access to memory. The key advantage offered by Vivado HLS and other so-called "C to Gates" tools, over coding everything in HDL, is the ability to code such an algorithm in a high-level language.

Vivado HLS can be a very powerful tool when used correctly. Like any EDA tool or programming language, it also has limitations that must be understood. A key strength of Vivado HLS is the rapid development of IP blocks can be created to perform complicated arithmetic functions; these would take a long time to code in traditional HDL languages such as VHDL or Verilog. Once written, HLS code can be easily modified, whereas modifications to HDL code that require structural changes to the design might require an order of magnitude or more engineering time.

For example, in C/C++, a software engineer could take a matter of minutes to write a function that evaluates a complicated arithmetic expression. In order to code the same expression in a traditional HDL language, the engineer must consider not only space, but also time, as well as performance considerations as pipelining and the need to achieve timing closure.

Vivado HLS has some limitations. It cannot magically analyse code written for a sequential processor, or arrays of processors (such as GPU cores), and create the optimal hardware architecture to execute

the algorithm described by the code. Naively converting an algorithm to HLS code without any consideration to HLS's limitations may create a correct implementation but with poor performance. To create a good architecture for an algorithm, some analysis must be performed in order to decompose the algorithm into sub-components that can form a pipeline.

A secondary consideration is the FPGA architecture; a knowledge of how the algorithm is likely to map to primitives within the FPGA is useful in order to avoid consuming FPGA resources unnecessarily.

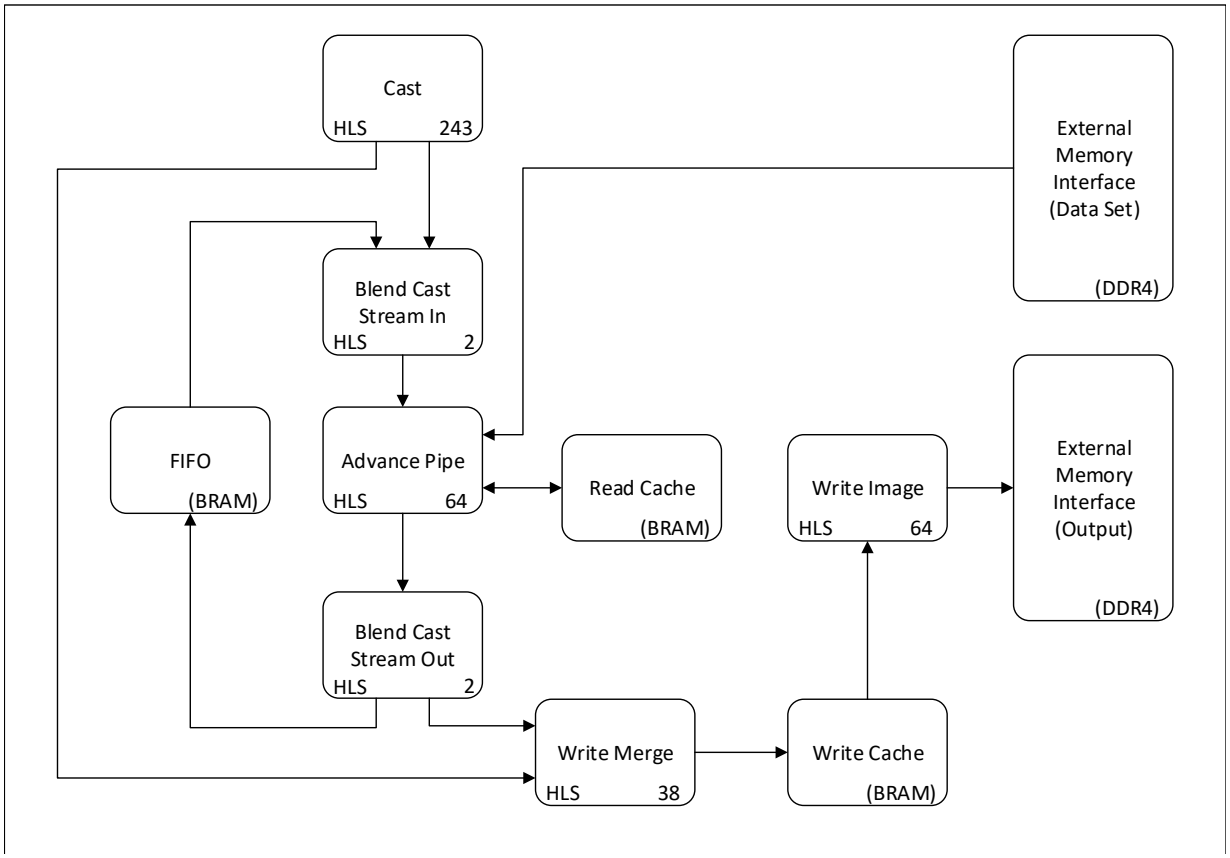

Figure 3: Simplified view of HLS Subsystem (Rendering core)

Figure 3 shows the connectivity of the HLS-generated components together with non-HLS components. Each HLS component has been annotated with the depth of its pipeline.

The Cast component generates a stream of initial rays to be cast into the volume data. For each pixel on the screen (work item), it performs 3-d geometric calculations to project and initialise rays that are used in other components. Each processed work item is output on one of two possible output streams: (i) the bypass stream, for rays that are calculated to miss the volume data, and (ii) a stream for rays that intersect the volume data and thus require processing. The Cast component is mathematically the heaviest of the HLS-generated components. HLS generated a 243-stage pipeline for this component, and it is capable of starting a new work item every clock cycle as long as the output streams are not blocked. The rate at which work items are processed is often referred to as the Iteration Interval (II). Cast can be said to have a latency of 243 and an II of 1.

Several of the HLS components are very simple stream merge entities; Blend Cast Stream Out and Blend Cast Stream In. Creating them as HLS IPs (as opposed to writing them in HDL) has the advantage that they have interfaces that are guaranteed to be compatible with those other HLS IPs, and they can be used when emulating the FPGA design on a microprocessor during functional verification. Both Blend Cast Stream In and Blend Cast Stream Out have IIs of 1.

Advance Pipe advances the rays though the data set to modify the colour of the ray based on the voxels that the ray passes though. Some of the required computations are in common with those that have already been performed in the Cast component. Therefore, instead of recalculating these

results, the results from the Cast component are passed as data along with the ray. This reduces the complexity of the Advance Pipe, so a pipeline length of only 64 is required. This is not without cost, because the extra bus width added to the stream out of the Cast component consumes additional FPGA routing resources. Extremely wide buses may result in difficulty in achieving timing closure.

The Advance Pipe has an II of 1, but its accesses to external memory can cause it to stall if the external memory does not respond quickly. To reduce the number of access to external memory and thus reduce the probability of a stall, a BlockRAM-based cache is attached to the component. Control of the cache is implemented in HLS code, but the cache was coded in HDL in order to have full control over its implementation. On each pass though the Advance Pipe component, one update to the colour of the ray is computed. If the ray requires further processing, it is pushed into the FIFO component by the Blend Cast Stream Out component, and it re-enters Advance Pipe at a later time. If the ray has completed its journey though the data set it is push into the Write Merge component.

Write Merge has two jobs. It takes a completed work item from either the cast bypass stream or the stream of completed Advanced Pipe work items, merging the two streams together, and writes the items into a cache. This component has an II of 1.

The Write Image component writes cache lines from the Write Cache when lines are ready. This component has an II of 2. This does not result in a bottleneck because the width of the memory is 512 bits, whereas the items input to Write Image are 32 bits wide. Therefore, Write Merge could run with an II of up to 16 without introducing a bottleneck into the system.

This section has now outlined all the HLS components required for the Volume Ray-Casting. Because they each have an II of 1 (except for Write Image, as noted above), the maximum possible throughput is achieved.

5 Host Interfacing

FPGAs, Vivado and Vivado-HLS has shown be to effective tools to building a custom processor dedicated to executing a single algrothem. However having the execution units is only half of the solution to creating an application to using that algrothm. The other part of the problem is how to move data in and out of the execution unit and control it.

Alpha Data's ADB3 Bridge and Driver provide a number of features demonstrated in this example that streamline the common task of interacting with the custom FPGA accelerator device; Direct Slave access to the device, Direct Memory Access(DMA) (in both directions between the host and FPGA device), interrupt handling, and communicating with multiple ADB3 devices via a common API to allow splitting the task between devices.

6 Host Software

The presentation layer of the application is using Direct3D (part of the Microsoft DirectX API). This is being used to display the frames rendered from the FPGA on the screen with minimum latency. In addition to presentation of the frames rendered from the FPGA card, a Direct Compute and single thread x86_64 version of the rendering algorithm can also be run.

Interaction with the FPGA card is performed in the render loop synchronising acquisition of frames with displaying them. On each iteration of the render loop a rendered frame is collected from the FPGA card, view parameters are updated, and rendering of the next frame is started.

6.1 FPGA Scalability

The demonstration application when running the FPGA version of the rendering routine will enumerate all the FPGA devices present in the system. The rendering task will then be divide between the available devices. In the case of this application is just splits the work items evenly between the available card. For example if two cards are present in the system, the frame is divided into two half frames, and each FPGA card will render one of these half frames. The half frames are then combine back into a full frame as part of the DMA transfer from the FPGA cards back to host memory.

7 Execution and Evaluation

There are many different parameters that can be varied in the Volume Ray-Casting example. The results presented below show how an FPGA card can outperform a traditional microprocessor-based implementation whilst being more energy-efficient.

The host system used to record the results detailed below consisted of an i7-4770S 3.1GHz CPU with 16 GiB of DDR4 SDRAM on a MSI Z87-G45 consumer-class motherboard. Power measurements were taken using sensors built into the Corsair AX860i supply powering the system. In each experiment, unused accelerator devices were removed from the system so that a fair comparison could be made without unused devices consuming static power in their idle state.

The volumetric data used to obtain the results below is a 32 x 32 x 32 voxel set of a translucent Buckyball such as the one seen in figure 1. Rendering was performed continuously for a period of 15 seconds, and the mean frame rate, power, and energy consumption were recorded. In addition to rendering on a set of FPGA cards, rendering was also performed on two other computing devices using a version of the algorithm ported appropriately: (i) a single thread on the system's processor, and (ii) the processor's integrated GPU, under Microsoft DirectCompute.

7.1 Results

Table 1 shows the mean achieved frame rate of the system at various sizes of output image (higher figures are better). It can be seen that, in all cases, the FPGA cards outperform the CPU.

Frame Size	KU3	2xKU3	3xKU3	x86_64	DXCompute
128x128	228.3	425.02	526	32.96	93.23
256x256	60.65	116.8	143.5	8.27	24.17
512x512	15.35	30.04	37.55	2.131	6.226
1024x1024	3.861	7.624	9.5	0.535	1.587

Table 1: Rendering speed, Frame Per Second (FPS)

Table 2 shows the mean power in each experiment. This is the mean total power of the system (including power used to display the output image). The x86_64 singled-threaded version required the least power, but this is perhaps not surprising given that it achieved the lowest frame rate.

Frame Size	KU3	2xKU3	3xKU3	x86_64	DXCompute
128x128	101.9	149.5	184.5	76.63	117.4
256x256	89.97	129.6	158.3	78.08	112.5
512x512	87.71	117	141.7	81.39	114.2
1024x1024	84.72	116.4	143	76.11	112

Table 2: System Power (Watts)

Table 3 and 4 show the energy cost of each frame, and each pixel, for each experiment.

Frame Size	KU3	2xKU3	3xKU3	x86_64	DXCompute
128x128	0.446	0.352	0.351	2.325	1.26
256x256	1.483	1.109	1.104	9.441	4.656
512x512	5.716	3.893	3.774	38.2	18.34
1024x1024	21.95	15.27	15.04	142.2	70.6

Table 3: Energy Per Frame (Joules)

Frame Size	KU3	2xKU3	3xKU3	x86_64	DXCompute
128x128	27.22	21.48	21.42	141.91	76.90
256x256	22.63	16.92	16.85	144.06	71.04
512x512	21.80	14.85	14.40	145.72	69.96
1024x1024	20.93	14.56	14.34	135.61	67.33

Table 4: Energy Per Pixel (µJoules)

These results show that the FPGA cards were more energy-efficient and performed faster than a x86_64 CPU for the presented application.

8 Conclusion

This demonstration has shown that a compute-intensive algorithm can be implemented in FPGA-based hardware using high-level tools, and scaled to run across multiple FPGAs, providing a relatively energy-efficient solution compared to a standard x86_64 CPU.

Please come an talk to us at SC2017 to find out more about the design or how an application may be be ported to an FPGA device or set of devices. We can also answer your questions via contacting us at sales@alpha-data.com.

9 References

Alpha Data's ADM-PCIe-KU3: https://www.alpha-data.com/dcp/products.php?product=adm-pcie-ku3

ADB3 Driver and SDK https://www.alpha-data.com/esp/softwareg3.php

Xilinx HLS: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html