RECONFIGURABLE SYSTEMS FOR VIDEO PROCESSING

by

BENJAMIN THOMAS COPE

A report submitted in fulfilment of requirements for the MPhil to PhD transfer examination.

Circuits and Systems Group
Dept of Electrical and Electronic Engineering
Imperial College London
University of London
Abstract

Video processing provides exciting research potential. The results of implementations are accessible to both professionals and the wider public. Due to the increasingly high video frame resolutions there is a higher demand on systems to process more frame data. The required processing speed is further increased by consumer demand for increasingly higher video quality.

Current systems exploit the dataflow and parallelism of video processing applications. The contributions proposed by this report are to define what type of new processing cores and FPGA embedded modules could be introduced to improve performance. These will be developed by considering graphics hardware. Graphics hardware is attractive as it’s technology is driven forward by a consumer market. It also improves in processing performance at a rate of up to 2.5 times per annum.

Over the next six chapters the objectives of the thesis are presented. Preliminary work towards these objectives is reported in chapters 3 and 4. Chapter 5 proposes a plan of future work, with a supporting Gantt chart. This demonstrates how the objectives will be fulfilled over the next twelve months.

It is found that there are complimentary advantages and disadvantages of using a graphics processing unit (GPU) versus a field programmable gate array (FPGA) for video processing. The GPU core in the GeForce 6800 Ultra is shown to be capable of implementing primary colour correction and 2D convolution (up to size $7 \times 7$) in real time. The benefits of the GPU over the FPGA are primarily in its ability to perform floating point vector computations very quickly. A downfall of the GPU is in local data storage and memory access. An FPGA can implement arbitrary storage and memory access patterns.

In summary the research area is to reassess video processing systems, from the angle of the benefits of graphics hardware. Video processing is an ambiguous and large set of algorithms. The specific area of broadcast video is approached.
Contents

Acknowledgements 2
List of Publications 3

1 Introduction 5

2 Research Area and Objectives 7
  2.1 Short-Term Objectives .................................................. 7
  2.1.1 Interconnects .................................................. 7
  2.1.2 Graphics Hardware for Video Processing ......................... 8
  2.1.3 Image Re-sizing .................................................. 8
  2.2 Long-Term Objectives .................................................. 9

3 Literature Review 10
  3.1 Architectures .................................................. 10
     3.1.1 FPGA .................................................. 11
     3.1.2 GPU .................................................. 16
  3.2 Interconnects .................................................. 22
     3.2.1 Buses .................................................. 22
     3.2.2 Crossbar Switch .............................................. 23
     3.2.3 Network-on-Chip .............................................. 24
  3.3 Example Applications .................................................. 25
     3.3.1 FPGA .................................................. 25
     3.3.2 GPU .................................................. 28
  3.4 Multiple or Mixed Core Systems ...................................... 29
     3.4.1 Mixed Core Solutions .............................................. 31
  3.5 Debugging tools and coding ............................................ 31
     3.5.1 Hardware Coding .............................................. 31
     3.5.2 Hardware Testing .............................................. 32
     3.5.3 GPU Coding .............................................. 32
     3.5.4 GPU Testing .............................................. 34
  3.6 Summary .................................................. 34

4 Progress to Date 36
  4.1 Bus Model .................................................. 36
     4.1.1 Multiplier Model .............................................. 36
     4.1.2 AMBA Bus .................................................. 37
  4.2 Comparison of FPGAs, GPUs and a CPU ................................ 41
## CONTENTS

4.2.1 Comparison Method ................................................. 42  
4.2.2 Relative Advantages of GPUs to FPGAs .......................... 43  
4.2.3 Primary Colour Correction ....................................... 44  
4.2.4 2D Convolution .................................................. 51  
4.2.5 Comparison of the Case Studies on the GPU ...................... 58  
4.2.6 Current ‘State of the Art’ Graphics Hardware ..................... 59  
4.2.7 Further Factors to Comparison ................................... 59  
4.3 Resizing Algorithms ................................................ 60  
4.3.1 Non-Edge Adaptive ............................................. 60  
4.3.2 Edge Adaptive .................................................. 62  
4.3.3 The Comparison ................................................ 63  
4.4 Summary ............................................................ 66  

5 Future Work  .......................................................... 67  
5.1 Further Algorithms .................................................. 67  
5.1.1 Interpolation .................................................... 68  
5.2 Enhancements to Video Processing Systems ......................... 70  
5.2.1 A GPU as part of a system-on-chip ................................ 70  
5.2.2 Embedded components for an FPGA ................................ 71  
5.2.3 Development of Tools .......................................... 71  
5.3 Gantt Chart and Thesis Layout ..................................... 71  

6 Conclusion .............................................................. 73  

A Research paper to appear in proceedings of Field Programmable Technology 2005  ............................................. 74  

B Gantt Chart for Future Work ............................................ 83  

Bibliography .............................................................. 88
Acknowledgements

I gratefully acknowledge the support provided by the Donal Morphy Scholarship, the UK Engineering and Physical Sciences Research Council, and Sony Broadcast & Professional Europe. Also support given by my supervisors professors Peter Y.K. Cheung and Wayne Luk.
List of Publications

The following publication has been written during the course of this work:

- B. Cope, P.Y.K. Cheung, W. Luk and S. Witt, “Have GPUs made FPGAs redundant in the field of Video Processing?,” to be included in proceedings of Field Programmable Technology, 2005 (see appendix A)
Chapter 1

Introduction

A video processing system is comprised of a set of functions performed over frames of pixels. These functions include compression, decompression and editing. A common system requirement is that this processing is performed in real time. Further, that there is no perceivable delay between a video stream entering and leaving the system.

Our desire for video with greater resolution and faster refresh rates, means that for real time processing a high throughput rate$^1$ is required. For example, high definition video (resolution $1920 \times 1080$ pixels), at a refresh rate of 30 frames per second, needs over 63 million pixels per second throughput rate for real time processing. This has led designers to consider alternatives to the general purpose central processing unit (CPU), which is not capable of such performance.

Video processing systems are susceptible to parallelism and pipelining at function, pixel and frame levels. Field programmable gate arrays (FPGAs) can be used to implement arbitrary levels of parallelism and pipelining. This makes them a viable solution. They have been used since the early 1990s [1] and are still used to prototype designs and often as the final solution. A downfall of these devices is in implementing complex arithmetic or floating point computation. This requires a large device area.

Graphics processing units (GPUs) are another alternative to implement video processing. They have a limited number of parallel pipelines, specific to a particular device. In contrast to FPGAs, video processing type GPU algorithms have only been explored since 2003 [2]. GPUs are designed for computer graphics and their increase in performance is driven by the video games industry. Operations performed in graphics applications have similarities to those performed in video processing functions. Both perform, among other operations, floating point vector and matrix operations on RGB pixel vectors.

$^1$Throughput rate is taken to be how many pixels are processed per second
1. Introduction

It is also becoming increasingly popular to use a number of hardware cores on the same chip, this is referred to as a system-on-chip (SoC). The decision variables for an SoC are the number and type of processing cores to use and how to interconnect them. An example is the PowerPC processor cores in Xilinx Virtex II-Pro and Virtex 4 FPGAs, which use the IBM CoreConnect bus architecture.

Processors are not the only solution to which processing core to use. Other options include custom hardware and reconfigurable logic.

The bus is not the only method for core interconnects either. Constructs of network-on-chip (NoC) and crossbar switch are also available to designers. NoC uses techniques and topologies from computer networks and is a scalable interconnect structure. The ‘crossbar switch’ is also scalable if a lesser known structure.

The aims of this report are: 1) Demonstrate my specific area of research 2) Define a number of research objectives 2) Show a foundation in the research area in terms of background reading 3) Present work as progress towards my thesis.

This report is organised as follows. Chapter 2 proposes the research area and objectives. Background work relative to the research area is shown in chapter 3. Chapter 4 details work carried out to date towards the aims of the research area. A projection of future work is shown in chapter 5. Chapter 6 concludes the report.
Chapter 2

Research Area and Objectives

This chapter can be split into two sections of short term and long term research objectives. Short term objectives cover, in part, work which will be presented in chapters 3 and 4. Long term objectives look more broadly at the aim and direction of the thesis.

The research area is reconfigurable systems for video processing. More specifically looking at the use of graphics hardware for video processing tasks. Using the product of this to reconsider current reconfigurable systems and improve their performance for video processing. The term ‘video processing’ is open ended and ambiguous. One can think of many applications which could fall under this title. Therefore the sub-set of algorithms associated with broadcast video are considered in the research area.

2.1 Short-Term Objectives

2.1.1 Interconnects

The interconnect between cores in a design is a common bottleneck. It is important to have a good model of the interconnect, to either eliminate or reduce this delay in the design stage. There have been many architectures proposed and developed for module interconnects (group-able as bus, crossbar switch and network), which are discussed in chapter 3. This leads to the first research objective: investigate suitable interconnect architectures for mixed core hardware blocks and find adequate ways to model interconnect behaviour. A model is important to decide the best interconnect for a task without the need for full implementation, which is costly in design time.

Sections 3.2 and 4.1 indicate steps towards fulfilling this objective. This is through a literary review of interconnects and a bus model respectively.
2.1 Short-Term Objectives

2.1.2 Graphics Hardware for Video Processing

The potential of graphics hardware has long been exploited in the gaming industry, focusing on its high pixel throughput and fast processing. It has been shown, to be particularly efficient where there is no inter-dependence between pixels. Programming this hardware was historically difficult. One could use assembly level language, in which it takes a long time to prototype. The alternative is an API, such as OpenGL, which limits a programmers choice to a set of predefined functions. In 2003 Nvidia produced a language called Cg. This facilitates high level programming without sacrificing the control of assembly level coding. Following the introduction of Cg, non-graphical applications of GPUs were explored. For example, the frequently cited FFT Algorithm by Moreland and Angel [2].

The adaptability of graphics hardware to non-standard tasks leads to the second research objective: to further investigate graphics hardware use as part of a video processing system. This takes advantage of the price-performance ratio of graphics hardware, whilst maintaining current benefits of using FPGA and processor cores. FPGA cores allow for high levels of parallelism and flexibility. This is because many designs can be implemented on the same hardware. Processors are optimised for an instruction set and can run many permutations of instruction. The fetch-decode-execute cycle of a processor is traded-off against the reprogramming time of an FPGAs.

This objective is covered in sections 3.1.2, 3.3.2 and 4.2.1. The architecture of the GPU is reviewed in the first. Example previous implementations are covered in the second. The third covers new comparisons of colour correction and 2D convolution.

2.1.3 Image Re-sizing

When one wishes to resize an image there are many possibilities for determining new values for pixels, filtering and interpolation being two. An example filtering method is to use a sinc function, where the complexity scales with the number of filter taps. Interpolation could be a bi-linear or bi-cubic method, the later being the more complex. The final research objectives is: investigate the perceived quality versus computational complexity of resizing methods. Theory suggests filtering, of a ‘long enough’ tap length, should produce a smoother result. However, this may not however be the best result perceptively or it may be too computationally complex. Edge adaptive non-linear methods are also potential choices [3, 4].

Sections 3.3.2 and 4.3 work towards meeting this objective. These cover some filtering considerations for a GPU implementation and a comparison of resizing methods respectively.
2.2 Long-Term Objectives

Chapter 4 will demonstrate that the limitations of GPUs and FPGAs are different and sometimes complimentary.

GPUs perform optimal floating point vector calculations. However, they have limited parallelism and cannot reuse data that has been read from memory for previously processed pixels.

FPGAs are highly parallelable and flexible in number representation. However, they are limited for floating point and complex operations, for which they require a large device area.

A system for performing arbitrary video processing tasks could benefit from containing both GPU and FPGA cores. Alternatively, an FPGA core with further embedded components. Such components benefiting from the years of research and development that has led the GPU to be a powerful floating point vector processing unit.

The interconnect structure of such a system is one concern, with a choice of bus, crossbar switch or network-on-chip architectures. The interconnect bandwidth of GPUs to a host CPU is frequently cited as a bottleneck in a design. For example, by Kelly in 2004 in the implementation of motion estimation on graphics hardware [5]. Kelly also suggests that a CPU may share memory with the GPU to overcome some of this bottleneck.

The proposal of a mixed or embedded core solution, aims to overcome the area and speed constraints associated with floating point calculation on an FPGA. However, to decide on the optimum combination of hardware or processor components, typical implementations must be considered. In chapter 4 applications of image resizing, colour correction and 2D convolution are exemplified. Further applications must be considered to broaden the examples and validate any decisions made as to optimum embedded or mixed core solutions.

In summary the long term objectives are: to investigate the improvement of video processing systems through (a) the use of multiple processing cores and (b) new embedded components for FPGAs. It is likely to be necessary to alter current tools or produce new tools to meet the objectives. This is included in objectives (a) and (b).

This report approaches these objectives as follows. Chapter 3 looks at background to the architectures, their interconnects and example implementations. Supporting algorithms for proposed new systems are presented in chapter 4. Chapter 5 explains how these objectives will be fulfilled over the next twelve months. A review of current tools for design and test is presented in section 3.5.
Chapter 3

Literature Review

This section aims to give a background to the research area through a review of research and development in the field. The following will be covered. Architectures of FPGAs and GPUs. Interconnect methods, both ones used today and proposed architectures. Example algorithms implemented on FPGAs and GPUs. Systems where multiple, or both, FPGAs and GPUs have been used. Finally a review of some of the tools used for implementing designs in each.

The implementation of video processing applications is moving away from being predominantly software focused to a hardware focused solution. This is shown through the increased popularity in the replacement of micro-processors with reconfigurable systems, such as Sonic [6].

There have also been advances in research into interconnect structures, the bus is no-longer seen as the only solution to connect hardware cores together. Switch-boxes and networks are also possible solutions. It is likely that we will see more topologies being considered in the future, than just the bus, and some of the new ideas today becoming common-place.

3.1 Architectures

This section will cover background to the architectures of the GPU and FPGA. In particular the focus on the embedded modules in the FPGA and the components which make up the GPU. This is preliminary research to fulfill objective (b) in section 2.2.

A primary design choice is whether to use hardware (for example FPGAs) or software (for example run on a CPU). Software implementations are beneficial for irregular, data-dependent or floating point calculations. A hardware solution is beneficial for regular, parallel computation at high speeds [7]. Hardware acceleration is particularly suited to video processing applications due to the parallelism and regular memory accesses.

Advancements in hardware means that some of the problems associated with floating point
calculations and alike have been overcome. Hardware can now perform such tasks equally or even better than software [8].

Due to the improvements in hardware and other factors co-design of hardware and software is becoming increasingly popular [9]. To make co-design possible the software designer needs a good software model of the hardware. The hardware designer requires a good level of abstraction [10].

Application specific integrated circuits (ASICs) solutions for processing tasks are considered to be optimal in speed, power and size [7], however, are expensive and inflexible.

Digital signal processors (DSPs) allow for more flexibility and can be reused for different applications. They are however, energy inefficient and throughput limited by their serial task implementation and subsequently clock rate.

It is often favourable to implement video processing applications in a reconfigurable unit. This give a good trade off of flexibility and optimal device speed, power and size. Reconfigurable devices are also capable of arbitrary levels of parallelism and pipelining.

GPUs have lots of computational power, which comes from specialised processing units, parallelism and pipeline. GPUs are part of a consumer product (the graphics card) which is mass-produced. This drives GPU development and helps maintain low unit cost. The FPGA is specialised and requires the development of dedicated FPGA platforms, which are expensive [11].

Two architectures are considered, representing the two design alternatives of hardware and software (processor based). Firstly, the FPGA as a reconfigurable hardware module. Second, the GPU which is the processing core of modern graphics cards.

FPGA architectures have been designed with tasks such as video processing in mind. The GPU is designed primarily for graphics applications.

### 3.1.1 FPGA

FPGAs were first designed to be as flexible as possible, comprising configurable logic blocks (CLBs) and switch-box interconnects. CLBs consist of four input lookup tables (LUTs), registers, multiplexors and other logic which can be used to implement macros and other functions [12]. Switch-box interconnects are a collection of transistors which allow interconnect lines to be multiplexed. As part of the switch matrix in an FPGA, switch-box’s help to overcome some of the net delay [12].

In future products manufacturers introduced embedded components, such as embedded mem-
ory blocks and in the latest FPGAs processors. This trades some of the flexibility for possible improvement in performance, when utilising these components. There is potential for future work in this area, to develop new embedded blocks which could improve functionality. The following subsections cover current embedded modules and proposed modules which could be included in future FPGAs.

Memory

In video applications storage of frames of data is important. It is useful therefore to be able to store data efficiently. The first FPGAs contained only LUTs and registers (in CLBs) as storage mechanisms. LUTs are used to implement logic functions and can be cascaded for larger functions. Registers are used for buffering data, as they are in close proximity to the four input LUTs used to implement functions. The use of CLBs for storing large amounts of data, for example rows of a video frame, is not optimal as logic in the CLBs is wasted. Due to customer demand for such storage FPGAs manufacturers introduced embedded memory modules.

The Xilinx Virtex II Pro [13] has 18Kb Block Select RAM modules. These are laid out in columns within the FPGA. Multiple memory blocks can be grouped together to form bigger memory modules. Dual port synchronous and asynchronous accesses are possible. Each port can be grouped together in any aspect ratio from $16K \times 1$ bit through $512 \times 36$ bit configurations.

The Virtex 4 [14] has built-in FIFO support in their memory modules which eliminated the need for external counter and status signals.

It is likely this technology will progress further, perhaps to an autonomous memory block (AMB) [15], which can create its own memory address. This is facilitated through an address generation unit which allows many access patterns, including FIFO and striped access. These are typically required in a DSP or video processing application. The advantage of the AMB is that a memory address does not need to be externally generated from or routed to the memory block.

Multipliers

The motivation for use of embedded multipliers is that implementation of binary multiplication in FPGAs is often too large and slow [16]. An alternative is to use embedded memory for lookup tables. This is suitable for small word length constant multiplication. However, the memory requirement becomes large for long word length non-constant multiplication.

Embedded multiplier blocks are the solution to these limitations. These sacrifice device flexibility and area for improved performance in multiply operations. Multiply operations are present
in many FPGA applications. Xilinx’s Virtex II Pro [13] contains up to 444 18 × 18 bit signed multiplier blocks. These are distributed in columns, placed next to the 18Kb block select RAMs as seen in figure 3.1. The multipliers and RAMs can be used together or independently. An example of their use together is for multiply accumulate operations on data from memory, such as in filtering. Memory, multipliers and the FPGA fabric are connected through a switch matrix.

A development of this is in the Virtex 4 [14] which contains embedded DSP Blocks. In addition to the multiplier the DSP blocks consist of adder logic and a 48-bit accumulator. Two DSP blocks along with extra logic and routing form a tile. Each tile is the height of four CLBs. These can be cascaded together to form a DSP function, for example a filter.

A limitation of current embedded multipliers is that they are fixed to 18 bit inputs, therefore space is wasted if smaller operands are used. A more flexible possibility is presented by Haynes [16] which used reconfigurable multiplier blocks. He presents a flexible array block (FAB) capable of multiplication of two 4 bit numbers. FABs can be combined to multiply numbers of lengths 4n and 4m. The speed of the FABs is comparable to that of non-configurable blocks, at a cost of them being twice the size and having twice the number of interconnects. The interconnects are not a problem due to the many metal layers in an FPGA. The FABs are also smaller than an equivalent CLB implementation.
A modification was proposed later to use a radix-4 overlapped multiple-bit scanning algorithm [17]. This has improved speed and area efficiency. The modified FAB (MFAB) multiplies two numbers with eight bit word lengths together. The length must be greater than seven to make a space saving on the FAB. The blocks are one thirtieth the size of the equivalent pure FPGA implementation and need only forty percent usage to make them a worthwhile asset.

Current embedded modules focus on fixed point arithmetic. There is the potential for similar floating point modules if an application demanded so. This is an area to explore in objective (b) in section 2.2.

**Processors**

The processor core used in the Xilinx Virtex II Pro [13] and Virtex 4 [14] is the PowerPC 405-D5 reduced instruction set computer (RISC). This is specially designed to run at low power and a core frequency of 300+ MHz. On-chip memory controllers (OMCs) handle communication to the FPGA block RAM which can be used to store either data or instructions. The dual port feature of these RAM modules can be used to communicate between the FPGA fabric and the processor.

Another useful feature is that the core is compatible with the IBM CoreConnect™ bus architecture. There are three main components to this, the processor local bus (PLB), on-chip peripheral bus and the device control register. Bus architectures will be considered in more detail in sub-section 3.2.1.

The high-level blocks are as follows. Instruction and data cache linked to the PLB or an OMC. A memory management unit which can be disabled, if not required. A fetch-decode unit with static branch prediction. Linked to this an execution unit with registers, an Arithmetic Logic Unit and a Multiply-accumulate block. Finally timer and debug blocks.

The PowerPC communicates with the FPGA fabric by two main methods, each through Block-RAM memory. One through the processor local bus (PLB) and the other the previously mentioned OMC. A PLB requires bus arbitration. The OMC provides non-cacheable memory accesses and does not require arbitration. There are separate OMC modules for instruction and data sides.

Figure 3.2 shows the high-level diagram for a PowerPC processor core.

Firstly considering the cache block. The PowerPC has three channels, two for data and one for instructions, each with an interface to a PLB or OCM controller.

Next the memory management unit. This contains a transaction look-aside buffer for data and instructions. It is used to control memory translation and protection.
3.1 Architectures

The fetch-decode-execute unit is where the main body of the processor is. The processor is an instruction set processor which fetches its instructions from memory using the PLB or OMC. It is active in that it ‘looks’ for its next instruction. In comparison graphics hardware required the prompt of a clip rectangle (CR) rendering command. Its instructions are then structured as a number of rendering instructions and a final write notifier (WN) instruction indicates the end of a render.

![Diagram of PowerPC processing core]

A PowerPC core is estimated to take up a die area of 14mm$^2$. This is estimated through measuring the percentage area a single core consumes of the largest device in Xilinx Project Navigator (0.012 percent). Then assuming the largest device die area to be 1.5 inch by 1.5 inch or 1444mm$^2$ [18].

Current FPGA processor cores are limited to the PowerPC processor or similar. There is scope for the investigation of alternative processing cores. An investigation into this forms part of objective (a) in section 2.2.

**Implementing Floating Point Computation**

Fixed point representation is normally adopted on FPGAs, however floating point calculations are also possible. Multiply, addition and multiply-accumulate operations are commonly used in video processing. Over a six year period (1997 to 2003) FPGAs have improved in performance, on these operations, at a rate of two to five times per year [19]. This exceeds Moore’s Law of an improvement of 1.5 times per year for CPUs [20]. This growth has continued and all these
operations are faster on FPGAs than CPUs [8].

Performing floating point calculations on FPGAs has a large area cost. Floating point multiplication at 140MHz, on a Virtex II Pro, requires 592 slices [19]. That is 12 percent of the slices in the XC2VP7 device. Embedded multipliers are used to speed up floating point arithmetic [19].

This shows that the potential of floating point arithmetic in FPGA's has been explored. However, as previously identified there is potential for future embedded modules to further improve performance for floating point arithmetic.

![Graphics Processing Subsystem](image)

Figure 3.3: Graphics processing subsystem with one (solid lines) or two (all lines) GPU core(s) [21]

### 3.1.2 GPU

Research into the use of GPUs for general-purpose ‘computation’ began on the Ikonas machine [22] which was developed in 1978. This was used for the genesis planet sequence in ‘Star Trek: The Wrath of Khan.’ It has also been used in television commercials and the ‘Superman III’ video game. This highlighted early on the potential for graphics hardware to do much more than output successive images.

More recently Moreland and Angel [2] implemented an FFT routine using GPUs. This was the first image or video processing use of graphics cards.

Use for such computation has been made possible by graphics hardware manufacturers (Nvidia...
and ATI being the largest) who now allow programmers more control over the GPU. This is facilitated through shader programs (released in 2001) written in languages such as Cg (C for graphics) by Nvidia or HLSL (high level shader language) by Microsoft. There is still room for future progress as some hardware functionality is still hidden from the user, there is no pointer support and debugging is difficult. Recognition for non-graphical applications of GPUs was given at the Siggraph Eurographics Hardware Workshop in San Diego (2003) showing its emergence as a recognised field.

![Figure 3.4: Memory structure of a graphics system [23]](image)

**How the GPU fits into a Computer System**

The GPU acts as a slave processor to the CPU. That is it operates in response to commands from a driver program executing on the CPU master. See figure 3.3 for the example system.

The processing commands can either be stored in RAM local to the GPU or in system memory. In the former, the CPU uses direct memory access (DMA) to store commands in memory on the graphics card. In the latter, the GPU uses DMA to fetch commands from system memory. The command buffer queues commands, received from the CPU via the system bus, in a circular FIFO. Access of command streams by the GPU is performed asynchronously to the GPU clock.

The GPU RAM is also used to store the display buffer (two for double buffering mode in which one is edited whilst the other is output). This is usually updated by the GPU, however the display buffer can also be updated by the CPU. The scan-out control logic (SCL) passes data in the display
buffer to the display device. A memory interface and bridge facilitate the access of memory by the GPU, SCL and CPU. Texture data is also stored in the onboard RAM.

The architecture of graphics hardware is equivalent to that of stretch computers (designed for fast floating point arithmetic). The GPU adheres to the stream processing model requiring a sequence of data in some order. This method exploits the dataflow in the organisation of the processing elements to reduce caching (CPUs are typically 60 percent cache [24]). Other features are the exploitation of spatial parallelism of images and that pixels are generally independent.

Figure 3.5: High-level diagram of GPU internal structure [23]

Memory Hierarchy

The memory structure of a CPU is very flexible, one can allocate or free local or global memory at any instance. The main forms of memory are registers, local memory, global memory and disk. Local memory is used to maintain a stack, global memory for a heap, registers for operand storage and the disk for data.

For GPUs the memory structure is much more restricted. Memory can only be allocated or freed before computation starts. There is limited access to the memory during computation. Local memory does not exist so kernel programs cannot maintain a stack. Global memory (in the form of textures) is read only during computation and write only at the end of computations. Disk access also does not exist. The reason for these restrictions is down to the optimisations of the GPU for graphics applications.

The memory model of the GPU and CPU can be seen in figure 3.4. The CPU model is well known and a simple flow of CPU main memory to caches to registers. The GPU model has a similar flow of video memory to caches to registers. However, it also has separate constant registers for values which are static over all pixels.
For an indication of the speed of the memory interface: the GPU is capable of 35GB/s (thousand million bytes per second) bandwidth to onboard DRAM. The CPU is capable of 6.4GB/s or more to system DRAM. The link from the CPU and system DRAM to the GPU is capable of up to 8GB/s bandwidth. The figures are given for a GeForce 6800 Ultra with a PCI Express bus interconnect and a Pentium 4 with an 800MHz front side bus speed [23].

Figure 3.6: Detailed high-level diagram of the Nvidia GeForce 6 series GPU internal structure [23]

**GPU Internal structure**

Figure 3.5 shows a simplified high-level view of the internal structure of a GPU. Initially data must be fed to the GPU from an external source to begin the rendering pass. The vertex data is passed to the vertex processor, previously this did not have access to textures, however from the GeForce 6 series onwards it can. Vertices are then rasterised to fragments (updates for pixels). The final
processing element in the flow is the fragment processor which performs texture mapping and other operations on the fragments. The output is then fed to the frame buffer of, which there may be more than one if double buffering or multiple render targets are implemented. After a rendering pass, data from the display buffer can be written to textures or back to the vertex buffer for a second pass. This feedback feature is useful for non-graphics applications.

Above is only a simplified view of what is happening in the GPU. A more detailed view of the GPU can be seen in figure 3.6. The diagram is for a GeForce 6 series GPU, however the 7 series is of a similar architecture with an additional eight fragment processing units. The first point of interest is the feed from the texture cache to the vertex processors. This facilitates the new vertex texture read feature.

Following the vertex processor are features common to graphics applications of cull, clip and vertex setup. These do not have to be completely ignored as they may be useful to avoid computation of a certain set of pixels. Z-cull may also be used to do this [23].

Next are the fragment processors, notice how they are organised in a 4×4 structure of processing elements [23]. That is there are four cores each processing squares of four pixels (called quad) at the same time. Combined they work on hundreds of pixels at once, in a SIMD processing fashion. Following this fragments update the display buffer in the DRAM, by either blending with the current value or replacing it.

**Fragment Processors**

Both the vertex and fragment processor provide floating point computational horsepower. However, it is normally common to use the floating point processor for per pixel processing and thus image or video processing. For this reason it is shown in more detail, see figure 3.7.

The fragment processor is given a number of initialisation parameters, or constants, at the beginning of computation. Following this a fetch, decode, execute loop occurs. Within this loop for each clock cycle: registers can be read, values mapped (including the re-ordering of vectors), optional texture lookup performed, input textures filtered and math operations evaluated. Following these possibilities the result is stored. Once all instructions have been executed the loop is exited.

**Where development is leading?**

The intentions of the manufactures is clear, in an article in July 2002 [26], NVIDIA’s CEO announced teaming with AMD to develop nForce. This will be capable of handling multimedia
tasks and will bring theatre-style DVD playback to the home computer. Previously the CPU off-loaded tasks, such as video processing, onto plug-in cards which were later shrunk and encapsulated within the CPU. This minimisation was beneficial to the likes of Intel and less so to GPU producers such as Nvidia. The implementation of multimedia applications on graphics cards (and their importance to the customer) means the screen is now seen as the computer, rather than the network it sits on. In development of Microsoft’s Xbox more money was given to Nvidia than Intel, this trend is likely to continue to show a power shift to graphics card manufacturers. Graphics hardware has progressed from being display hardware to allow user programmability. This has led to use for non-graphics applications.
3.2 Interconnects

In section 2.1 the first objective is to assess the interconnects of processing cores. This section covers background reading to aid in meeting this objective. It also contributes to objective (a) in section 2.2. It defines the type of interconnects that could be used to connect cores in a system on chip (SoC) design.

The design of the interconnect structure of a system is becoming more important as design sizes increase. The international technology roadmap for semi-conductors 2003 highlighted global interconnects and skew rates as a design challenge for the future. The requirement for scalability of an interconnect structure is becoming increasingly important to keep the design time low.

This section will describe the merits and failures of three interconnect types bus, crossbar switch and network-on-chip.

3.2.1 Buses

A bus has three main components. An ‘arbiter’ decides who ‘owns’ the bus. A ‘master’ attempts to own the bus. A ‘slave’ responds to the masters’ communications.

One bus structure is the AMBA AHB Bus [27] which is a processor controlled model [28]. Where the processor decides when it can transmit and arbitrates for the bus. The opposite is an arbiter controlled model where the arbiter holds requests from the processors and then decides when they can transmit.

There are other bus architectures such as the open core Wishbone bus and the IBM CoreConnectTM which have different features. All buses have the above three components and adhere to the common concept of request by master and grant by arbiter.

GPU view

GPU components are implemented in conjunction with CPUs acting as graphics sub-systems and working as co-processors with the CPUs. To do this a high speed interface is required as GPUs can process large amounts of data in parallel. The bandwidth requirement doubles every two years [29]. The AGP standard has progressed through 1× to the current 8× model (peaking at 2.1GB/s). However, with new GPUs working at higher bit precisions (128 bit for RGBα in the GeForce 6800 series) greater throughput was required. AGP uses parallel point to point interconnections with timing relative to the source. As the transfer speed increased, the capacitance and inductance on connectors needed to be reduced, this became restrictive past 8×. A new transfer
method was required, serial differential point to point offers a high speed interconnect at 2.5GB/s. This became the first PCI Express generation, overcoming these capacitance and inductance issues [29]. PCI Express 16× offers 8GB/s peak bandwidth (4GB/s in each direction) and is the new standard for CPU to GPU communications.

**FPGA view**

The PowerPC processor mentioned in sub-section 3.1.1 is compatible with the IBM CoreConnect™ bus. This is a soft core which can be interfaced to the two masters of the PowerPC (for instructions and data). The embedded development kit for the FPGA includes such soft cores so this introduces only a small amount of design time. The PowerPC is a 32-bit processor, the 32-bit CoreConnect™ bus has a peak bandwidth of 264MB/s at a maximum clock frequency of 66MHz. An advantageous feature of this bus is that it requires no tri-state on-chip buffers. This makes the soft core implementation possible.

**3.2.2 Crossbar Switch**

A crossbar switch architecture is defined as size N×N where N is the number of switch ports [28]. The control, address and data are communicated in one packet and a central control unit arbitrates between masters on the switch ports. Different size switch and port widths can be used. Xu et al [28] simulated these using OPNET (a communications network package which they modified to work with a NoC). In creating their architecture model they make assumptions about how long certain blocks take to function. An arbiter or switch control for example takes one cycle. They showed that a switch was better than the bus model for more than eight bits per port with a high effective throughput and low latency. Their metrics, which may be useful to analyse interconnects in general, are as follows [28]:

\[
\text{Utilisation} = \frac{\text{Avg} - \text{throughput}}{\text{Max} - \text{throughput}} \quad (3.1)
\]

\[
\text{System frequency} = \frac{3 \times 10^8 \times 150}{\text{Processed} - \text{Frames}} \quad (3.2)
\]

Where throughput is proportional to 1/utilisation.

Xu et al found the optimum for the Smart SoC Camera system to be a 3×3 crossbar switch with two shared memories, where processors are split across the memory. They draw the conclusion that the bottleneck in a SoC design is the interconnect. This justifies a large amount of design time on this.
A crossbar switch should not be confused with switch-box interconnects, which are used in FPGAs. These facilitate the connection of two interconnect wires (between CLBs) and have no packet structure.

### 3.2.3 Network-on-Chip

The connection of a number of cores (for example DSPs) to a number of resources (for example memory) raises questions of the best interconnect structure. We require the maximum possible throughput at the lowest possible ‘cost’. The concept cost can be power, area or even money.

The advantages of a NoC are that it has high performance (or bandwidth), modularity, can handle concurrent communications and has better electrical properties than a bus or crossbar switch. As the size of chips increases global synchrony becomes infeasible as it takes a signal several clock cycles to travel across a chip. The NoC overcomes this problem as it is globally-asynchronous locally-synchronous (GALS) architecture.

Dally [30] proposes a mesh structured NoC interconnect as a general purpose interconnect structure. The advantage of being general purpose is that the frequency of use would be greater justifying more effort in design. The disadvantage is that one could do better by optimising to an certain application, though this may not be financially viable.

In Dally’s example he divide a chip into an array of sixteen tiles, numbered 0 through 3 in each axis. Interconnections between tiles are made as a folded torus topology (i.e. in the order 0,2,3,1.) This attempts to minimise the number of tiles a packet must pass through to reach it’s destination. Each tile therefore has a N,S,E,W connection and each has an input and output path to put data into the network or take it out respectively. The data, address and control signals are grouped and sent as a single item called a ‘flit’. Area is dominated by buffers (6.6 percent of tile area in their example). The limitations are opposite to computer networks. There is less constraint on number of interconnections, but more on buffer space. The network can be run at 4GB/s (at least twice the speed of tiles) to increase efficiency, however this would increase space required for buffers.

The disadvantage of Dally’s work is that the tiles are not always going to be of equal size. Therefore space is wasted for smaller tiles. Jantsch [31] proposes a solution which overcomes this, using a similar mesh structure. The main differences are that he no longer uses the torus topology but connects tiles to their neighbours only. He also provides a region wrapper, around a block considerably larger than others, which emulates the original same size tile network being present.
Jantsch suggests two possibilities for the future, either many NoC designs for many applications (expensive in design time) or one NoC design for many applications (inflexible). The later would justify the design cost, however one would need to decide on the correct architecture (mix of cores), language, operating protocol and design method for a set of tasks.

Other interconnect methods are suggested. One is a honeycomb structure [32] where each component connects to six others. Next a scalable, programmable, interconnect network (SPIN) [33] with a tree structure of routers. The communicating nodes being the leaves of the tree. Dobkin et al [34] propose a similar mesh structure to Jantsch however include bit-serial long-range links. They use a non-return to zero method for the bit-serial connection and believe it to be best for NoC.

This shows a snapshot of the NoC ideas for which there are many topology ideas, as with personal computer networks.

3.3 Example Applications

Example applications cover background work necessary in deciding which algorithms to chose for the second objective in section 2.1. This is also important for choosing supporting algorithms for the long term goals laid out in section 2.2.

3.3.1 FPGA

When deciding which hardware to use for a video processing system there is a trade-off between operating speed and flexibility. Custom processors maximise speed whilst general purpose processors are maximally flexible. To be viable a new system must give more flexibility than custom processors and be faster than a general purpose processor. The need for flexibility is justified as one may need a change an algorithm, for example for a new compression standard, post manufacture. By utilising its re-programmability a small FPGA can appear as a large and efficient device.

Use as a graphics processor

Singh and Bellac in 1994 [35] implemented three graphics applications on a FPGA. These were to draw the outline of circle, filled circle and a sphere. They found a RAM based FPGA was favourable due to the large storage requirements. The performance in drawing the circle was seen to be satisfactory. It out-performed a general purpose display processor (TMS34020) by a factor of
3.3 Example Applications

six (achieving 16 million pixels per second throughput). It was however worse in the application
of fast sphere rendering at only 2627 spheres per second versus 14,300 from a custom hardware
block. Improvements are however expected with new FPGAs, such as the Virtex 4, which have
larger on-chip storage and more processing power. FPGAs today also have more built-in blocks to
speed up operations such as multiplication, as shown in sub-section 3.1.1.

To extend the instruction set of a processor

FPGA fabric can also be used to extend the instruction set of a host-processor, i.e. as virtual
hardware. This idea is approached by Vermeulen et al [36] where a processor is mixed with some
hardware to extend its instruction set. In general this ‘hardware’ could be another processor or
an ASIC component, again there are issues with finding ways to get the components to commu-
nicate and work together. Design issues in this include how to get the two modules to efficiently
communicate their state. A possible solution is the use of shared status registers.

The auxiliary processor unit (APU) controller, associated with the PowerPC processor in the
Virtex 4, facilitates the extension of the instruction set by a soft processing core in the FPGA
fabric [12]. This is an example of how such instruction set extension is already used for embedded
processors in FPGAs.

Sonic

Extensions to section 3.3.1 are the Sonic and UltraSonic [6, 37] architectures, developed at Impe-
rial College London in conjunction with Sony Broadcast Laboratory.

The challenges involved, highlighted in [6, 38] are: Correct hardware and software partition-
ing, spatial and temporal resolution, hardware integration with software, keeping memory accesses
low and real-time throughput. Sonic approaches these challenges with plug in processing elements
(PIPEs) with three main components of an engine (for computations), a router (for routing, for-
matting and data access) and memory for storing video data.

The PIPEs act as plug-ins. These are analogous to software plug-ins, and provides a sensible
model for the software programmer. This overcomes a previous problem with re-configurable
hardware that there were no good software models.

Algorithms can be efficiently divided between hardware and software on this architecture [39].
It was found that an average 83 percent reduction in processing speed, over a CPU implementation,
can be made through such task partitioning.
On a Task-level parallelism is often ignored in designs, by proposing a design method focused on the system dataflow Sedcole [37] hopes to exploit this. Taking Sonic as an example spatial parallelism is implemented through distributing parts of each frame across multiple hardware blocks, PIPEs in this case. Temporal parallelism can be exploited by distributing entire frames over these blocks.

**Partial Reconfiguration**

Singh and Bellac [35] also consider partial reconfiguration and propose partitioning the FPGA into zones. Each zone would have good periphery access to the network and be a different size. The capability of partial reconfiguration is important if a new task is required only a section of the FPGA need be reconfigured, leaving other sections untouched for later reuse. A practical example of this is seen with the Sonic architecture. The router and engine are implemented on separate FPGAs. If a different application required only a different memory access pattern (for example 2*1D implementation of a 2D filter [10]) only the router need be reconfigured, this separation also provides abstraction.

More recently with the development of partial reconfigurability a single chip version of Sonic has been proposed [40]. Each of the PIPEs is now a zone of the FPGA, each reconfigurable whilst leaving the rest of the system unchanged.

**Summary**

The requirements of a reconfigurable implementation are to be flexible, powerful, low cost, run-time and partial reconfigurable and to fit in well with software. The current FPGA limitations are configuration time, debugging, number of gates, partial reconfiguration (Altera previously had no support) and PCI Bus Bottleneck [6, 10]. These considerations are also important if considering an FPGA’s implementation with other hardware in a ‘mixed system’.

An architecture where the bus bottleneck problem is reduced is seen in [7] where a daughter board is incorporated to perform D/A conversion. The sharing of data and configuration control path reduces the bottleneck, however, data-loss occurs during configuration phase but this is seen as acceptable.
3.3 Example Applications

3.3.2 GPU

Processing Power

The rate of increase of processing performance of GPUs has been 2.8 times per year since 1993 [24] (compared to two times per 1.5 years for CPUs according to Moore’s law) a trend which is expected to continue till 2013. The GeForce 5900 performs at 20G/flops, which is equivalent to a 10GHz Pentium Processor [24]. This shows the potential of graphics hardware to out-perform CPUs, with a new generation being unveiled every six months. It is expected that TFLOP performance will be seen from graphics hardware in 2005. For Example in Strzodka and Garbe’s implementation of Motion Estimation [41] they out-perform a P4 2GHz processor by four to five times, with a GeForce 5800 Ultra.

A GPU has a higher memory bandwidth, more floating point units and single instruction-stream, multiple data-stream (SIMD) processing than a CPU. The Nvidia 6800 Ultra has a peak performance of 40 GFlops per second compared to 6.8 GFlops per second for a 3GHz Pentium 4 [20].

Filtering

Increases in performance also benefit filtering type operations. In the previously mentioned FFT implementation [2] the potential for frequency domain filtering was shown. The amount of computations required to do filtering are reduced by performing them in the frequency domain from an $O(NM^2)$ problem to an $O(NM) + FFT + IFFT$ (about $O(MN(log(M) + log(N)))$) one. Moreland and Angel implemented ‘clever tricks’ with indexing (dynamic programming), frequency compression and splitting a 2D into two 1D problems to achieve this speed up. With the rapidly increasing power of graphics cards it can be expected that the computation time will be reduced from one second to allow real-time processing. A final factor which aids this is that 32-bit precision calculations are now possible on GPUs which are vital for such calculations.

Another benefit of the GPU is cost, a top end graphics card (capable of near TFLOP performance) can be purchased for less than 300 pounds. Such a graphics card can perform equivalently to image generators costing 1000’s pounds in 1999 [24]. This gives the opportunity for real-time video processing capabilities on a standard workstations.

Computer Vision Related

Strzodka and Garbe in their paper on motion estimation and visualisation on graphics cards [41] show how a parallel computer application can be implemented in graphics hardware. They iden-
tify GPUs as not the best solution but to have a better price-performance ratio than other hardware solutions. In such applications the data-stream controls the flow rather than instructions, facilitating a cache benefit. Moreland and Angel [2] go further in branding the GPU as no-longer a fixed pipeline architecture but a single instruction stream multiple data stream (SIMD) parallel processor, which highlights the flexibility in the eyes of the programmer.

Further Fung and Mann [42] implement a real-time projective camera motion tracking algorithm. They use the GeForce FX 5900. This algorithm requires a high degree of local processing per pixel and is therefore well suited to the GPU.

### 3.4 Multiple or Mixed Core Systems

![Example command stream for a system of multiple GPUs](image)

Parallelism can also be exploited by using multiple cores [42, 43, 44]. Work will now be discussed which uses multiple FPGAs and GPUs as well as both together.

This section looks specifically at the use of multiple or mixed cores in a system. This is cohesive with objective (a) in section 2.2. That is to use multiple cores in a new video processing system, particularly graphics hardware.

#### Use of multiple GPUs

Figure 3.3 shows an example of a system with two GPUs. In this situation if we have M rows in a frame rows 1 to P will be processed on GPU0 and rows P+1 to M on GPU1 [21]. Each GPU must wait for the other to finish a current frame before moving to the next. An equal split of rows is not always optimal. For example in a video game most of the action occurs in the bottom half of the screen, thus requiring more processing. Dynamic adjustment of this load balance is required for optimal adjustment.

The main modules to facilitate a multi-processor system are: command stream generator (CSG), imbalance detecting module (IDM) and partitioning module (PM) [21]. The CSG provides commands to both GPUs. The IDM detects when one GPU is waiting for the other and re-addresses
3.4 Multiple or Mixed Core Systems

the balance of rows to each through the PM.

A simplified system model is shown in figure 3.3. In a typical implementation the two GPUs and their individual memory and memory interface are on separate cards. The scan out logic on the primary graphics card is used to access the memory on both cards and output display data.

Figure 3.8 shows an example of a command stream for such a system. The first command CR is a clip rectangle (view space) command, which is sent to both GPUs. Next some rendering instructions are sent to both GPUs. Following this the master GPU (GPU0) is stalled whilst the secondary GPU sends its rendered data to the display buffer of memory associated with GPU0. Finally a write notifier (WN) command is sent to identify the end of a rendering pass.

A GPU example is a motion tracking algorithm implemented on a GeForce 5900 [42], with a core clock speed of 400MHz. This performs 3.5 times faster than a 2.0GHz Athlon Processor with one device. Five GPUs perform the same task 4.5 times faster than a single GPU [42].

Use of Multiple FPGAs

Another graphics application example is presented by Fender [45]. They implement a ray-tracing engine on multiple Virtex-E devices. They show this implementation to be up to 23 times faster than a 2.0GHz Pentium 4 implementation. They project that newer Virtex II Pro could achieve rendering at 288 times faster than the CPU.

Multiple FPGAs are also used by Gause in a shape-adaptive template matching (SA-TM) algorithm implemented on the Sonic architecture [44]. This shows potential for up to 28,600 times speedup over a 1.4GHz Pentium 4 when using multiple Virtex 1000E devices.

The following are general issues, Sedcole et al propose to be considered in such large scale implementations:

- Design complexity
- Modularisation - the allocation and management of resources
- Connectivity and communication between modules
- Power minimisation (indicates low memory accesses)

_Sonic and Ultra-Sonic:_ Typical applications are the removal of distortions introduced by watermarking an image [38], 2D filtering [10] and 2D convolution [37]. In the latter an implementation
at half the clock rate of state-of-the-art technology, capable of the same task, was adequate suggesting a lower power solution. 2D filtering was split into two 1D filters and showed a 5.5 times speed up when using one PIPE, and greater speed up with more PIPEs.

### 3.4.1 Mixed Core Solutions

Alternatively FPGAs, GPUs and CPUs can be used together in a mixed core system. Sepia [46] is one such system applied to volume rendering. In a Sepia module the GPU renders individual objects; the FPGA merges objects from a module and its adjacent module and the CPU prepares data and instructions. A network of Sepia modules is used to render an entire space.

CT Reconstruction [47] is another example of a mixed core system. The FPGA performs backward projection and the GPU forward projection. This makes optimum use of the GPU, whilst the FPGA is used for a task not suitable for the GPU.

This work is similar to what is being proposed in the objective (b) of section 2.2. The difference however is that in these examples the GPU has been used solely as part of the graphics card. From the proposed long term objective (b) the use of a GPU core as part of an SoC architecture is to be explored.

### 3.5 Debugging tools and coding

This section will look at the practicality of implementing designs on FPGAs and GPUs. It is important to understand how each is used to implement a design to contrast their effectiveness as a medium for performing the tasks. Firstly hardware coding and testing are considered, followed by GPU coding and testing. This is not directly related to the objectives, however it is paramount to understand how a hardware core is to be programmed and debugged.

This review is also be useful for developing any design tools, that are necessary as a product of meeting the objectives set in section 2.2.

#### 3.5.1 Hardware Coding

FPGAs can be programmed through well known languages such as VHDL and Verilog at the lower level. MATLAB system generator, and more recently SystemC (available from systemc.org) and HandleC at the higher level.
3.5 Debugging tools and coding

3.5.2 Hardware Testing

Pre-load

A downside to FPGAs over ASICs is in pre-load particularly pre-place and route. In ASIC design only wiring capacitance is missing from pre-synthesis tests. For the FPGA, module placement is decided at place and route. The path lengths can vary dramatically, in this step, due to the layout of resources. This can drastically effect timing.

The most widely known pre-load test environments are ModelSim (Xilinx) and Quartus (Altera). COMPASS (Avant) is an example of an automated design tool, creating a level of abstraction for the user. The benefits are that the user can enter a design as a state machine or dataflow. Therefore implement at the system (high) level rather than circuit (low) level [35].

Post-load

The issue of post-load testing is currently approached by using part of the FPGA space for a debugging environment, invoked during on-board test. A previously popular test strategy was ‘Bed of Nails.’ In this pins are connected directly to the chip and a logic analyser. Due to the large pin count on today’s devices this is impractical, even if possible it would significantly alter the timing.

Following was ‘Boundary Scanning’ by the joint test action group (JTAG). However this only probed external signals.

Better still is Xilinx Chipscope. This is an embedded black box which resides inside the FPGA, as a probe unit. The downside is that is uses the slow JTAG interface to communicate readings.

An example of an on-chip debugging environment, which uses a faster interface (the PCI bus), is the SONICmole [48] used with UltraSonic. This takes up only four percent of a Virtex XVC1000 chip (512 slices). It’s function is to act as a logic analyser, viewing and driving signals, whilst being as small as possible and having a good software interface. This uses the PIPE (the processing unit of Sonic and UltraSonic) memory to store signal captures. It has been implemented at the UltraSonic maximum frequency of 66MHz [48]. It is also portable to other reconfigurable systems.

3.5.3 GPU Coding

Initially there were two choices for implementing a GPU design. Firstly at the high level application programmer interface (API) instructions could be used. These are a limited set of instructions, facilitated through DirectX or OpenGL. Secondly at a low level, programmers could write assem-
bly code. This has the advantage of being more flexible. However for large designs the time taken becomes impractical. In 2003 a number of high level languages were produced for GPUs, one of the most popular being Cg developed by Nvidia.

**Cg: The tool**

Cg [49] was so that developers could program GPUs in a C-like manner. The features of C beneficial for an equivalent GPU programming tool are: performance, portability, generality and user control over machine level operations. The main difference to C is the stream processing model for parallelism in GPUs.

When developing Cg Nvidia worked closely with other companies (such as Microsoft) who were developing similar tools. An aim of Cg was to support non-shading uses of GPU, this is of particular interest. Fernando and Kilgard [25] provide a tutorial on using Cg to program graphics hardware). For the non-programmable parts of a GPU CgFX [49] handles the configuration settings and parameters.

Cg supports high level programming, however is linkable with assembly code for optimised units this gives the programmer more control. Cg supports user defined compound types (for example arrays and structures) which are useful for non-graphics applications. It also allows vectors and matrices of floating point numbers up to dimension four (for example for colour space RGB\(\alpha\)). A downside is Cg does not support pointers or recursive calls (as there is no stack structure). Pointers may be implemented at a later date.

**Cg: How to program**

Nvidia separates programming of the two types of GPU processor units (vertex and fragment) to avoid branching and loop problems and so they are accessed independently. The downside is optimisations between the processor cores are not possible. A solution is to use a meta-programming system to merge this boundary.

Nvidia introduced the concept of profiling for handling differences in generations of GPUs. Each GPU era has a profile of what it is capable of implementing. There is also a profile level which represents all GPUs necessary for portable code.

There are two programming models for graphics hardware [41]:

- **Flowware**: Assembly and Direction of dataflow
- **Configware**: Configuration of processing elements
For comparison these two features are implemented together in an FPGA, however in graphics hardware these are explicitly different. Careful implementation of GPU code is necessary for platform (for example DX 9.0) and system (for example C++) independence. APIs also handle flowware and configware separately. This becomes important if considering programming FPGAs and GPUs simultaneously.

### 3.5.4 GPU Testing

Relative to other programming languages Cg is still in its infancy. Therefore the debugging and test skills for it are still in development. The suggested interface is to use a C or C++ coding environment to link the Cg code with API calls to the graphics hardware. This is through either OpenGL or DirectX.

For debugging there is a dedicated Cg compiler which indicates the line in the code which contains a particular bug. Further there are a number of tools developed by Nvidia to assess performance of your hardware and debug in run-time. One such tool is NVShaderPerf which allows the user to estimate the performance of a GPU for a particular piece of Cg code. This is also linked into the Cg compiler.

### 3.6 Summary

This chapter has identified the key literary areas which are necessary to investigate to meet the research objectives of chapter 2.

Firstly the architectures of both the FPGA and GPU are explored to identify their current component parts. Specifically for the FPGA how such embedded components are connected to the rest of the FPGA fabric. For the GPU how the dataflow of algorithms is exploited to fully utilise its processing capability.

Interconnect structure are detailed to explore how one may wish to design a system on chip. This follows onto the bus model which will be presented in section 4.1.

Sample implementations for both architectures are shown to identify their strengths and weaknesses. Also to give perspective on the relative age of implementations on each of the hardware. FPGA implementations are more mature field that those on GPUs.

Implementations for systems with multiple cores are shown. Particularly the Sonic and Ultra-Sonic systems which are a good example of dataflow exploitation. As part of this a two mixed core solutions are shown which highlight similar work to that proposed in objective (a) in section 2.2.
The difference is noted in that the authors of the cited designs focus on using the graphics card as a complete entity. This objective focuses on using the GPU core individually.

Finally the coding and debugging tools for each of the hardware are presented. This review is to show which tools it may be necessary to modify, to model the performance of systems proposed to meet the thesis objectives of section 2.2.
Chapter 4

Progress to Date

Progress can be grouped into four key areas. Initially a model of the ARM advanced microcontroller bus architecture (AMBA) bus was produced to give background to bus interconnects. A comparison of implementations of a primary colour correction algorithm on FPGAs, GPUs and a CPU followed. A 2D convolution filter was implemented to demonstrate another example to compare the architectures. Finally a comparison of interpolation methods for the re-sizing of video frames with the intension of efficient implementation on the hardware detailed in chapter 3. These will now be discussed individually in more detail.

4.1 Bus Model

My first project was a high level model of the ARM AMBA bus. This model predicts bus performance for varying numbers of masters (for example processors) and slaves (for example memory) and is cycle accurate. SystemC, a hardware modelling library for C or C++, was used for this. This contributes to the first short term objective in section 2.1.

The motivation came from a paper by Vermeulen and Catthoor [36]. They proposed using an ARM7 processor to allow for up to ten percent post manufacture functional modification to a custom hardware design.

4.1.1 Multiplier Model

To demonstrate the design process with the SystemC library a multiply function, for a communicating processor and memory, was modelled. Two values to be multiplied are loaded from memory in consecutive clock cycles. These are then multiplied and returned to memory using a point to point interconnect. This is the simplest bus model with one master and one slave. SystemC outputs a value change dump (VCD) file, this can then be displayed in a waveform viewer.
The results are seen in figure 4.1. This shows two identical multiply cycles of the design. Two values $\times 2$ and $\times 17$ are read from memory locations $\times 1$ and $\times 2$ respectively. The product $\times 2e$ is returned to location $\times 3$.

Figure 4.1: Waveform for multiplier implementation

### 4.1.2 AMBA Bus

From the AMBA bus specification [27] two sub-types were of interest, namely the advanced high performance bus (AHB) and advanced system bus (ASB). Both types allow multiple numbers of masters and slaves, have high performance and are pipelined. The advantages of the AHB over the ASB are that it allows burst transfers and split transactions. The AHB is also the newer architecture and is chosen for the design.

A typical design consisting of one master with two slaves, taken from the specification [27], is a good starting point for specifying the model (see figure 4.2). The decoder decides which slave should process the data by considering the current address. Signals are (de-)multiplexed between the slaves and the master. The arbiter (not shown) trivially grants the single master bus access all the time. A possible implementation of this architecture is for slave one (memory) to hold
instructions for the master (processor) and slave two (memory) data (operands and results). The routing and decoding between the blocks, along with arbiter signalling comprises a ‘bus’.

Next I considered a model of custom processors (named SimpleScalar) written by Ray Cheung [50]. We worked together to propose the possibility of integrating the AMBA bus model and his processor model. We developed Vermeulen and Catthoor’s work [36] suggesting a fully flexible bus and processor model. The model is flexible to include any bus configuration. After discussion with Ray Cheung I focused on the design of a flexible bus model.

**AMBA AHB Design**

A physical interpretation of how the AMBA AHB bus blocks fit together can be seen in figure 4.3. Global clock and reset signals have been omitted from the figure. HWDATA and HRDATA are write and read data respectively. The control signals are requests from masters and split (a resume transfer function) signals from slaves.

![Block diagram of the AMBA AHB bus](image)

**Figure 4.3: Block diagram of the AMBA AHB bus**

The challenge in implementing the slave and master multiplexer blocks was in making them general. The master multiplexer uses a delayed master select signal from the arbiter to pipeline address and data buses. One master can use the data bus whilst another controls the address bus. For the decoder an assumption was made about how a slave is chosen. The address most
significant bits (MSBs) are used to decide which slave to use, \( \text{ceil} (\log_2(\text{number slaves})) \) are required. The decimal value of the MSBs indicates which slave to use, for example b01 would be slave number one.

The arbiter implementation demonstrates the power of using a high level language such as SystemC. It is the most complex of the blocks because it must hold past information about which buses have recently used the bus. The initial design uses a fixed priority scheme where the master with the lowest number has priority. For example master three has priority over master four. Later this can be extended to include techniques such as ‘round-robin’ and ‘last-used’. The arbiter must also be reactive to the split requests of the slave devices. A split request follows a slave holting a transfer by asserting that it is not ready for more than a predefined number of cycles. The request is to resume the transfer indicating that it is now ready to receive.

**AMBA AHB Results**

A test procedure was produced, this loads stimulus from a text file, with the result viewed as a waveform, as with the multiplexer example. The file consists of lines of either, ‘variable’ and ‘value’ pairs or ‘tick’ followed by a number of cycles to run for. Initially, simple tests were carried out, to check for correct reset behaviour and that the two multiplexers worked with a setup of one master and two slaves. An example of a test output is shown in 4.4.

![Figure 4.4: Test output showing reset and the bus request and grant procedure](image-url)
In the example as HSEL signals change at the bottom of the waveform, the two read data signals are multiplexed. When reset all outputs are set to zero irrespective of inputs, this is what would be expected. When the master requests the bus the arbiter waits until HREADY goes high before granting access, through HGRANT. In the case of more than one master, the HMASTER signal changes immediately (with HBUSREQ) to the correct master, allowing for multiplexing and so the slaves know which master is communicating.

The model was further tested with two masters and two slaves, a common configuration. Within this, the sending of packets consisting of one and multiple data items was experimented with along with split transfers and error responses from slaves. Below is a list of all scenarios which were tested.

- Both masters sending packets of burst length one word
- Master 0 sending bursts of varying length and master 1 bursts of length one (and vice versa)
- Both masters sending packets of varying lengths
- Split transfer sequences by both slave devices
- Error response from both slaves
- Suspend the transfer by toggling ready signal from active slave

The correct, cycle accurate, results were seen. The waveforms for these become complicated and large very quickly, however are of a similar form to figure 4.4. An example is shown in figure 4.5 for the test of multiple packets of varying length from both masters. Notice that the grant signals are of different lengths representing the change in burst length. The binary data has been converted to integer representation so that it can be seen on the waveform.

**Extension to a general bus model**

Further to this work a general bus model was proposed. This removes the limitation of just implementing the AMBA AHB bus features. Such a model would focus on the throughput potential of the bus and not on the cycle accuracy as in section 4.1.2. The motivation for this was to decide the bus features which are best to suit a particular application. Note a bus with these features may not exist. Such a model was implemented as a summer project by Stephen Spain [51].
4.2 Comparison of FPGAs, GPUs and a CPU

Video processing algorithms comprise of a set of operations performed over frames of pixels. When accelerating such algorithms, FPGAs are used to prototype designs and often as the final solution. Their flexibility, with which the parallelism of an algorithm can be exploited, makes FPGAs an obvious solution. Operations, pixels or even frames can be processed in parallel. This section demonstrates that GPUs also offer a viable solution, capable of exploiting parallelism and meeting the application throughput rate. This is demonstrated through case studies of primary colour correction and variable sized 2D convolutions.

The first case study is primary colour correction targeted at progressive high definition video (1920 × 1080 frame size at 30 frames per second.) The algorithm consists of the following range of operations: vector arithmetic, complex arithmetic, branching statements and memory accesses. It is also separable into modules which contain different proportions of the above operations. From this the performance of each hardware core can be evaluated.

A second case study, an \(n \times n\) 2D convolution targeted at a frame size of 512 × 512 pixels
also at 30 frames per second, focuses on memory accesses. This is a significant bottleneck for the GPU [2].

For real-time application target throughput rates of 63MP/s (1920 * 1080 * 30 fps) and 8MP/s (512 * 512 * 30 fps) are required for primary colour correction and 2D convolution respectively.

This section highlights two algorithms which are useful in demonstrating the limitations of GPUs. These can be used as part of a set of representative algorithms to justify the long term objectives in section 2.2.

### 4.2.1 Comparison Method

When considering which hardware to use for a task one considers many metrics including throughput, area and power. The case studies focus on throughput, but also consider die area. The aim is to see whether a device is capable of real time implementation of a target application.

#### Throughput Rate

For video processing on all hardware throughput is considered in terms of MP/s. This is how many million pixels can be processed per second.

For FPGA implementations one pixel is clocked in and out per clock cycle so throughput is the maximum clock rate.

For the CPU throughput is defined as the clock rate divided by the number of clock cycles taken for the application. If SSE or MMX instructions are used this is multiplied by the degree of parallelism (i.e. four).

Throughput for the GPU is the number of parallel pipelines multiplied by clock rate divided by number of cycles taken by the task.

#### Choice of Devices

The scenario is set of a designer deciding whether to use a GPU or an FPGA for accelerating a video processing task. The capability of each device to perform the task is the primary concern.

Firstly, Nvidia’s GeForce 6800 GT and Xilinx’s Virtex II Pro are chosen for comparison. These are the current ‘state of the art’ available to all designers (data taken May 2005). Secondly, the GeForce 6600 GT and the Spartan 3 are chosen as ‘low cost’ alternatives. The GPUs are refered to as 6800 GT and 6600 GT as apposed to the chip names of NV40-GT and NV43-GT respectively.

The Pentium 4 3.0GHz processor is chosen to benchmark the comparison.
The 6600 GT has half the number of parallel pipelines of the 6800 GT (eight versus sixteen). The 6600 GT is on a smaller process technology of 110 nm compared to 130 nm for the 6800 GT. It also runs at a faster core clock rate, 500MHz compared to 350MHz for the 6800 GT.

New Virtex II Pro devices have a hybrid process technology of 90 nm high speed transistors and 130 nm metal layers. The Spartan 3 is on a 90 nm process. Both these devices can exploit arbitrary pipelining and parallelism, within area limitations.

For comparison the Pentium 4 is on a process technology of 90 nm. GPU manufacturers are behind FPGA and CPU manufacturers in shrinking their technology size to 90 nm.

This choice of devices covers two scenarios. One, a designer wishes to get the best performance. Two, they are also concerned about low cost.

### 4.2.2 Relative Advantages of GPUs to FPGAs

If a design is well matched to the instruction set of the GPU one would expect a GPU implementation to be advantageous. Arithmetic operations on vectors and matrices of dimensionality three or four are the most apparent. Less apparent operations supported by the instruction set include: vector range limiting, $x^y$ and euclidean distance. Branching statements must be avoided by replacement with blending based conditional statements [20]. This is a first check in considering whether to use a GPU or an FPGA.

Next, consider memory access. In graphics applications memory accessed by one pixel is in close proximity to that accessed by its neighbours. The number of accesses is also low. In a rendering example [25] one may require a texture lookup, specular detail and parameters for shadow and diffuse mapping per pixel. An application requiring random memory accesses such as Huffman decoding would not be suited to the GPU. Low memory usage in a regular manner is well suited to the GPU.

FPGA designs can be implemented to make efficient use of on-chip memory. This is not possible on a GPU and makes the FPGA a more attractive solution.

For both FPGAs and GPUs the synthesis tool or compiler used must make optimum use of the hardware. For the FPGA this includes the minimisation of fan outs and routing delays as well as optimum resource usage. A compiler for the GPU must optimally convert high-level functions into GPU assembly instructions. Cg includes a library of high level functions to facilitate this [25].
Die Area

Die area is often compared as it gives an indication of scalability for future improved performance. The GeForce 6600GT has a die area of $156 \text{mm}^2$ [18].

The 6800 GT has more parallel pipelines and a larger process size than the 6600 GT. They are from the same family (GeForce 6 series) therefore have very similar architectures. From this we assume the 6800 GT to be larger than the 6600 GT.

The largest Virtex II Pro device is assumed to be 1.5 inch by 1.5 inch ($1444 \text{mm}^2$) [18]. Die area is scaled, using CLBs (configurable logic blocks), to a smaller device. The XC2VP7 has 1,232 CLBs compared to 13,904 CLBs for the XCV2P125. This gives an estimated area for the XC2VP7 of $128 \text{mm}^2$.

Assume the largest Virtex 4 device to also be $1444 \text{mm}^2$. The Virtex 4 is on the same process technology and has the same architecture as the Spartan 3. We scale in a similar manner as above from the Virtex 4 LX devices. The XC4VLX200 has 22,272 CLBs compared to 6,912 CLBs for the XC3S4000. This gives an estimated area of $448 \text{mm}^2$.

The Pentium 4 has a die area of $112 \text{mm}^2$ [52].

Test Conditions

A GeForce 6800 GT was used to test the algorithms. The speed of this device along with other GPUs was predicted from the NVIDIA NVshaderperf tool. This assumes one cycle texture lookup assuming that there are no memory clashes.

The FPGA design was written in VHDL with synthesis and place and routing performed in Xilinx Project Navigator. The in-built synthesis and 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.

4.2.3 Primary Colour Correction

The design consists of three modules: input correct (IC), histogram equalisation (HE) and colour balance (CB). Each performs non-linear functions on an RGB signal from various inputs (see figure 4.6).

IC and CB include colour space conversion of $\text{RGB to HSL}$ (hue, saturation and luminance)
space, alterations of HSL and conversion back to RGB space. HE performs, among other operations, range limiting and a power function based on gamma. The modifications in HE and CB are performed conditionally on R, G, B or all channels. The algorithm is now described in more detail, identifying optimisations.

The Design

Colour Space Conversion Sub-Module. RGB to HSL space conversion can be broken down into conversion from RGB to yCrCb then yCrCb to HSL, and vice versa for HSL to RGB space.

Conversion of RGB to yCbCr space is a matrix multiply operation [53]. y is clamped to range [16,235] and (Cb,Cr) to range [16,240]. yCbCr to RGB conversion also requires a matrix multiply [53]. After conversion RGB is clamped to range [0,255].

The conversion between yCbCr and HSL colour spaces is computationally expensive. For this reason another colour space XYL is defined, derived from yCbCr, which is equivalent to HSL.

The first terms X and Y are related to Cb and Cr as shown in equations 4.1 and 4.2.

\[
X = (Cb - 128)/112 \tag{4.1}
\]

\[
Y = (Cr - 128)/112 \tag{4.2}
\]

(X,Y) is related to hue and saturation as a co-ordinate representation of the polar space (H,S). H and S being vector angle and length respectively.
4.2 Comparison of FPGAs, GPUs and a CPU

Equation 4.3 shows how luminance relates to \( y \).

\[
L = \frac{(y - 16)}{(235 - 16)} \tag{4.3}
\]

The inverse of these equations is performed for conversion from XYL to \( \gamma \)CbCr space, range limiting \( \gamma \)CbCr as above.

***(X,Y) Modifications.*** The required modifications are: saturation multiplied by SatGain\((m)\) and added to SatShift\((\Delta S)\). HueShift\((\Delta H)\) is added to hue.

These modifications translate to a rotation and scaling of \((X,Y)\). The new values \((X',Y')\) are calculated using equation 4.4.

\[
\begin{bmatrix} X' \\ Y' \end{bmatrix} = K \begin{bmatrix} \cos \Delta H & -\sin \Delta H \\ \sin \Delta H & \cos \Delta H \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} \tag{4.4}
\]

where \( K = (m + \frac{\Delta S}{S}) \). \( S \) is euclidian\((X,Y)\), the pre modification saturation. The euclidian\((X',Y')\) (new saturation), by definition, must be of range \([0,1]\). If euclidian\((X',Y')\) is greater than one it is normalised to one. It is always greater than zero.

The modifications are simplified as SatGain is only applied in IC and SatShift only in CB.

**Input Correct Module.** IC consists of: Colour space conversion to XYL space. The \((X,Y)\) modifications as shown above. Luminance is multiplied by LumGain, added to LumShift and range-limited to \([0,1]\). Finally conversion back to RGB space is performed.

**Histogram Equalisation Module.** The first HE modification to RGB is summarised by equation 4.5.

\[
RGB' = (RGB - BlackLevel) \ast BWDIFF \tag{4.5}
\]

\( BWDIFF \) is BlackLevel minus WhiteLevel. Next \( RGB' \) is range limited to \([0,255]\). \( RGB' \) is then set to the power \( 1/Gamma \). The result is range limited to \([\text{OutputBlackLevel}, \text{OutputWhiteLevel}]\). ChannelSelection selects whether to apply the changes to R, G, B or all channels.

For all hardware the power function, for varying gamma, is pre-computed and stored in memory.

**Colour Balance Module.** CB is similar to IC, except Luminance is only effected by LumShift and range limited to \([0,1]\). As with HE it also includes conditional modification to decide which
of the channels to modify. The original luminance (i.e. L after initial conversion to XYL space) is tested to be either \( < TH_1 \), in range \([TH_1, TH_2]\), or \( > TH_2 \) depending on AreaSelection. If the condition is satisfied then RGB is modified as above. Otherwise it is left unchanged. Another AreaSelection condition allows modifications independent of L.

**Number Representation.** For the GPU and CPU floating point numbers are used. This maintains accuracy and gives satisfactory results.

For the FPGA we use a bit-width optimised fixed point representation. Each RGB channel is represented as eight unsigned bits. For XYL a colour space conversion model was produced and bit-width optimised. The error dropped off exponentially to \( 1/2 \) RGB bit width when there is 18 bit per XYL component. This is a fair choice to maintain accuracy.

**Results for the GPU**

As the GPU is the least familiar hardware a separate section is devoted to it. This highlights the benefits of optimisations performed to GPU algorithms. Also shown is the impressive increase in performance of GPUs over the last three years.

Table 4.1 shows the performance results for the initial and optimised designs using various generations of GPUs. It is seen that there is a large variation in the throughput rates of the devices, although there is only two to three years difference in device age. For more information on the optimisation of the primary colour correction algorithm see [53].

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Throughput (Final) MP/s</th>
<th>Throughput (Initial) MP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>6800 Ultra</td>
<td>120.75</td>
<td>44.14</td>
</tr>
<tr>
<td>6800 GT</td>
<td>105.66</td>
<td>38.62</td>
</tr>
<tr>
<td>6600</td>
<td>72.73</td>
<td>27.59</td>
</tr>
<tr>
<td>5700 Ultra</td>
<td>12.67</td>
<td>2.12</td>
</tr>
<tr>
<td>5200 Ultra</td>
<td>7.08</td>
<td>1.24</td>
</tr>
</tbody>
</table>

Table 4.1: Performance comparison on GeForce architectures for the optimised (final) and initial designs

For efficient optimisation of an algorithm it is important to understand the performance penalty of each section. A detailed breakdown of the primary colour correction algorithm, in terms of delay, was carried out. Some performance bottlenecks in the implementation were ‘compare’ and ‘clamping’ operations. The Colour Balancing function, which includes many of each of these, was
Table 4.2: Effect on performance of each block of the primary colour correction algorithm

seen to be the slowest of the three main blocks. The conversion between colour spaces was seen to have a large delay penalty due mainly to the conversion from RGB to XYL space. In Histogram Equalisation ‘pow’ was seen also to add greatly to the delay and accounts for almost half of the delay (0.00089s/MP).

The register usage, although minimal, was seen to be larger in calculations than compare operations. This is due to the large number of min-terms in the calculations and there being fewer intermediate storages required in compares. In this case the register usages was not a limiting factor to the implementation, however it may be for other algorithms. The breakdown of delay for each block can be seen in table 4.2, for more detail see [53].

The optimisations for a GPU can be summarised below:

- Perform calculations in vectors and matrices
- Use in-built functions to replace complex maths and conditional statements
- Pre-compute uniform inputs, where possible, avoiding repetition for each pixel
- Consider what is happening at assembly code - decipher code if necessary
- Do not convert between colour spaces if not explicitly required

Results with all hardware

Firstly the full algorithm will be considered, followed by the separate blocks. The differences in hardware performance on the algorithm, primarily between the GPU and FPGA, will be noted.

Full Algorithm. The results for the primary colour correction algorithm are seen in figure 4.7. Throughputs for two GPU and two FPGA architectures are shown along with a CPU implementation as a benchmark.
4.2 Comparison of FPGAs, GPUs and a CPU

Both FPGA implementations perform faster than the GPUs. This shows the advantage of the flexible pipelining and parallelism of FPGAs, over the fixed number of parallel pipelines in the GPU. The GeForce 6800 GT approaches the throughput of the Spartan 3 device. The high performance GPU is approaching the speed of a budget FPGA. FPGAs and GPUs perform faster than the Pentium 4.

The benefits of the GPU over the CPU are an optimised instruction set and up to 16 pipelines. The 6800 GT is clocked at 350MHz, 8.5 times slower than the CPU. It is 18 times faster in throughput. This shows the operational efficiency advantage of the optimised instruction set to be at least nine times.

FPGAs benefit from a fixed point bit-width optimised design, another reason for their superior performance. GPUs and CPUs are disadvantaged for algorithms amenable to bit-width modifications, due to their rigid instruction set. A floating point FPGA implementation would have a larger area requirement and potentially slower throughput.

The XC2VP7 Virtex II Pro device (75 percent slice utilisation) was used for implementing the colour correction algorithm. This is three times smaller than the 6600 GT and even smaller than the 6800 GT.
We have shown that the FPGA has a higher throughput than the GPU in the primary colour correction application. However, as we see from the figure 4.7 both the 6800 GT and 6600 GT devices are capable of performing at the target throughput rate (indicated by the bold line). Therefore in terms of suitability both FPGAs and both GPUs are capable of performing the task.

<table>
<thead>
<tr>
<th>Block</th>
<th>Vector</th>
<th>Arith.</th>
<th>Memory</th>
<th>Branch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>L</td>
</tr>
<tr>
<td>HE</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>CB</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>R2X</td>
<td>H</td>
<td>N</td>
<td>N</td>
<td>L</td>
</tr>
<tr>
<td>X2R</td>
<td>H</td>
<td>N</td>
<td>N</td>
<td>L</td>
</tr>
<tr>
<td>FULL</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

Table 4.3: Classification of primary colour correction blocks (N - null, L - low, M - medium and H - high usage) [54]

**Individual Blocks.** The blocks represent different types of algorithms, see table 4.3.

Where N, L, M, and H represent Null, Low Medium and High usage. R2X is *RGB to XYL* conversion and X2R is *XYL to RGB* conversion. The FULL algorithm is shown last for completeness. Arith. and Branch. are arithmetic and branching instructions respectively. Results by module can be seen in figure 4.8.

![Figure 4.8: Maximum throughput (MP/s) of colour correction modules for FPGAs, GPUs and a CPU (IC - input correct, HE - histogram equalisation, CB - colour balance, R2X - RGB to XYL conversion, X2R - XYL to RGB conversion, FULL - full design)](image-url)
It is seen that the FPGA has more consistent throughput rates than the GPU, due to the pipelining nature of hardware. The throughput rate of a GPU and CPU scales with the number of cycles performed. For example the FULL algorithm takes 53 cycles, whereas IC takes 16 cycles, on the GPU. This has the same ratio as their throughput rates 105.66 to 350MHz respectively.

The GPU is fast at R2X and X2R conversions. Both contain a matrix-vector multiplication, vector addition and range limiting. R2X is slow, relative to X2R, due to inconsistent range limiting over XYL (X,Y have a different range than L), not suited to the instruction set. The X2R module range limits RGB to [0,255]. The instruction set is optimised to perform identical operations on vectors. Throughput of R2X and X2R is almost identical for FPGAs and the Pentium 4.

IC and CB, implemented on the GPU, show the most noticeable performance different. CB is slower due to additional branching instructions to conditionally alter RGB based on luminance. The 6800 GT 23 clock cycles to implement CB compared to 16 for IC. The FPGA implementation has similar throughput for both modules due to pipelining. The CPU implements CB slower than IC, the difference is less noticeable than the GPU due to its general purpose instruction set.

HE performs the fastest, on the GPU, of the three modules. This is because it takes only 12 cycles to compute. However it does contain a higher ratio of memory access and branching instructions which are associated with poor GPU performance. The FPGAs are again much slower than the GPU due to its higher clock rate and only one pixel pipeline.

For both IC and CB the FPGA implementations approach the performance of the GPUs. Half of the XC2VP4 device slices are utilised for these modules, we estimate this to be six times smaller than the GeForce 6600 GT. Similar can be said for HE which requires the smallest Virtex II Pro XC2VP2, twelve times smaller than the GeForce 6600 GT.

The results show consistently higher throughput for the GPU over the FPGA. This is because of its higher clock rate and the small number of instructions relative to the full algorithm. The FPGA however, requires a die area much smaller than the GPU.

### 4.2.4 2D Convolution

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 the FPGA is more efficient. Previously accessed data from external memory can be held locally in on-chip memory. The flexibility of the FPGA 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 section will describe how the 2D convolution algorithm can be implemented on GPUs, FPGAs and CPUs. Followed by the results and analysis.

The Design

The algorithm for 2D convolution, of mask size $n \times n$, is shown in equation 4.6.

$$p'(x, y) = \sum_{i=-L}^{M} \sum_{j=-L}^{M} h(i, j)p(x + i, y + j)$$  (4.6)

If $n$ is odd: $L = M = (n - 1)/2$. If $n$ even: $L = n/2, M = (n/2) - 1$. $p'$ is the new image, $p$ the original image and $h$ the filter mask.

For the purpose of our experiment filtering is performed to each R, G and B component. The filter is assumed to be non-separable. The video is assumed to be a non-interlaced progressive scanned input.

GPU Implementation

For each of the three implementations the 2D convolution task can be divided into two main sections, shown in figure 4.9. 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.

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:

```c
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.

![High Level Block diagram for all Implementations](image)

**Processing Data** The example of 2D convolution size $8 \times 8$ 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 4.10 shows an $8 \times 8$ 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 $4 \times 4$ and where necessary $3 \times 3$ or smaller.

![How an array of $8 \times 8$ data items is divided for processing in the GPU](image)

Each of these smaller arrays is 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.
FPGA Implementation

This has the same high level block diagram as for the GPU, see figure 4.9. The implementation of each shall be considered as with the GPU.

![Diagram](image)

**Figure 4.11: How data fetch is implemented in the FPGA design**

**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 4.11 shows how the data is stored for a filter size $n$.

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 is not 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.

**Processing Data** For this block we again take an example of one size of convolution which scales for others. The choice is size $2 \times 2$, the implementation can be seen in figure 4.12. 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.
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 does not 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 structure is again similar to that for the FPGA and GPU shown in figure 4.9.

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 Comparison of FPGAs, GPUs and a CPU

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

```c
// 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];
}
```

**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 $n$.

<table>
<thead>
<tr>
<th>Size</th>
<th>6800 Ultra</th>
<th>6800 GT</th>
<th>6600 GT</th>
<th>5900 Ultra</th>
<th>5800 Ultra</th>
<th>5200 Ultra</th>
<th>3.0 GHz P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 x 2</td>
<td>1070</td>
<td>933</td>
<td>667</td>
<td>150</td>
<td>62.5</td>
<td>25</td>
<td>14</td>
</tr>
<tr>
<td>3 x 3</td>
<td>278</td>
<td>243</td>
<td>174</td>
<td>17</td>
<td>10.7</td>
<td>4.3</td>
<td>9.7</td>
</tr>
<tr>
<td>4 x 4</td>
<td>110</td>
<td>96.5</td>
<td>69</td>
<td>4.3</td>
<td>3.2</td>
<td>1.3</td>
<td>6.8</td>
</tr>
<tr>
<td>5 x 5</td>
<td>54</td>
<td>47.5</td>
<td>34</td>
<td>2.6</td>
<td>1.7</td>
<td>0.7</td>
<td>5.1</td>
</tr>
<tr>
<td>6 x 6</td>
<td>33</td>
<td>29</td>
<td>21</td>
<td>3</td>
<td>2.3</td>
<td>0.9</td>
<td>3.5</td>
</tr>
<tr>
<td>7 x 7</td>
<td>22</td>
<td>19</td>
<td>14</td>
<td>1.5</td>
<td>1</td>
<td>0.4</td>
<td>2.6</td>
</tr>
<tr>
<td>8 x 8</td>
<td>11</td>
<td>10</td>
<td>7.1</td>
<td>1.1</td>
<td>0.8</td>
<td>0.3</td>
<td>2.1</td>
</tr>
<tr>
<td>9 x 9</td>
<td>9</td>
<td>8</td>
<td>5.6</td>
<td>1.2</td>
<td>1</td>
<td>0.4</td>
<td>1.6</td>
</tr>
<tr>
<td>10 x 10</td>
<td>5.5</td>
<td>5</td>
<td>3.5</td>
<td>0.7</td>
<td>0.5</td>
<td>0.2</td>
<td>1.3</td>
</tr>
<tr>
<td>11 x 11</td>
<td>4.7</td>
<td>4</td>
<td>3</td>
<td>0.6</td>
<td>0.4</td>
<td>0.16</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Table 4.4: Throughput (MP/s) of 2D convolution implemented on varying GPUs and a CPU
Results of GPU with CPU

As with the primary colour correction algorithm the GPU is the least familiar of the hardware. This will be analysed briefly before comparing results to the FPGA. The CPU is used to benchmark the performance.

The results for varying GPUs, compared to the CPU, can be seen in table 4.4. 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 side of table 4.4 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.

![Throughput vs Filter Size for 2D Convolution comparing varying GPUs, FPGAs and a CPU](image)

Figure 4.13: Maximum throughput (MP/s) of 2D convolution for GPUs, FPGAs and a CPU

Results with all hardware

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

The logarithm of the resultant throughput is shown due to the greater than exponential decrease of throughput for GPUs (see figure 4.13).

The FPGAs maintain a constant throughput rate relative to the GPUs. This is due to flexible par-
4.2 Comparison of FPGAs, GPUs and a CPU

Parallelism, pipelining and streaming of data. The inefficient memory access of the GPU mentioned earlier is verified.

At convolution sizes of $10 \times 10$ and $11 \times 11$ the designs became very large. For the Spartan 3 this meant a greater decrease in performance, 99 percent of the device was used. The Virtex II Pro showed a more constant throughput for larger convolution sizes.

Both GPUs and FPGAs are capable of performing at the target throughput rate for sizes up to $7 \times 7$. After this the throughput rate of the GPUs continues to fall off rapidly and they become unusable for the application.

The Pentium 4 performance drops below the target throughput rate at size $4 \times 4$. The performance with respect to the GPU shows improvement in memory access speed of new 6 series GPUs over the old 5 series [54, 2].

To stream 2D convolution data the FPGA stores rows of pixels, for a $512 \times 512$ frame size $512 \times n \times 24$ bits (8 bits per R, G, B) are required. As $n$ increases the design size and synthesis time increase rapidly. The $11 \times 11$ convolution requires the largest device of each family. For a larger frame size, such as high definition video and for large $n$ such data streaming may not be possible.

As $n$ increased the number of multiplies ($3 \times n^2$) exceeded the number of multipliers in the largest Spartan 3. LUTs can be used instead. Where both options were possible the one with the fastest throughput rate was taken.

At 2D convolution size $7 \times 7$ the Virtex II Pro XCV2P40 was required, using the same analysis on the die size (it has 43,632 logic cells) we estimate its area to be $212mm^2$. This is larger than the GeForce 6600 GT, however we are unable to compare it precisely to the 6800 GT.

4.2.5 Comparison of the Case Studies on the GPU

The GPU implementation of the primary colour correction algorithm takes 132 instructions and on average 53 cycles per pixel. 2D convolution ranges from 28 instructions and 5 cycles to 811 instructions and 1364 clock cycles for $2 \times 2$ and $11 \times 11$ respectively. Primary colour correction on average computes 2.5 instructions per clock cycle. 2D convolution size $11 \times 11$ computes 0.6 instructions per cycle. This is due to the 121 ($11 \times 11$) memory accesses required per pixel. We see for size $2 \times 2$ the instructions per cycle is high, the GPU can perform texture lookups in the same cycle as arithmetic operations.
4.2 Comparison of FPGAs, GPUs and a CPU

4.2.6 Current ‘State of the Art’ Graphics Hardware

The above analysis was performed in June 2005. More recently the Nvidia GeForce 7800 GTX device has been released. For comparison this can process the full primary colour correction function at 303.53MP/s. This outperforms all FPGA devices. For the 2D convolution filter its decrease in performance is equivalent to that of the older GeForce GPUs. For $2 \times 2$ convolution it performs at over 2GP/s and for $11 \times 11$ 8MP/s. This demonstrates a performance increase of over 2.5 times between the 6800 and 7800 generation of GPUs. This is a time period of approximately one year.

4.2.7 Further Factors to Comparison

Identified are six factors which should also be considered to decide the ‘best’ hardware for a particular application.

**Power consumption.** FPGA devices are predicted to operate at a lower power than both GPUs and CPUs. FPGAs have lower clock speeds and do not normally require heat sinks and fans.

**Die Area.** Die area for a GPU and CPU is fixed. Many device sizes are available for FPGAs. For small designs, such as 2D convolution size $3 \times 3$, we predict the FPGA would use a smaller die area.

**Design time.** This is expected to increase the appeal of GPUs. Cg code is higher level than hardware description languages and closer to proof of concept designs, typically in C/C++. Languages such as HandleIC and SystemC aim to reduce the design time gap.

**Redundancy.** GPUs and CPUs are already in a computer system, FPGAs are added at extra cost. The GPU or CPU can be considered a free resource, if the current GPU or CPU is not fully utilised.

**Multiple pipelines within FPGAs.** Multiple pixels can be computed in parallel on an FPGA, if the video memory bandwidth is sufficient. We did not explore this as the FPGAs met our target throughput for each case study.

**Multiple cores.** If a device is not capable of the target throughput rate a number of them could be used [44, 43, 42]. Further experimentation showing the speedup would be required. Many GPUs may be capable of performing $11 \times 11$ 2D convolution at the target throughput rate.
4.3 Resizing Algorithms

One of the fundamental blocks in a video processing system is one of format conversion (including frame size and whether it is interlaced or progressive scanned). One application is converting broadcast video into a format which can be displayed on a local device. With the introduction of higher resolution formats improved interpolation (up size) and decimation (down size) methods are required.

There are different issues associated with each direction of video resizing. For interpolation one aims to minimise the effect of ‘blocky’ edges (moire) or excessive smoothness. These relate to a disproportional amount of high and low frequency components. For decimation one primarily aims to reduce the effect of aliasing caused by trying to represent frequency components of the original image which are not possible in the new sampling rate, this causes ‘ringing effects.’

Downsizing can be performed by firstly low pass filtering to remove frequency components which it is not possible to display at the new sampling rate. New samples can be taken from this filtered version of the original frame. This has the undesirable effect of causing smoothness as edge detail is held in the high frequencies. In this section we shall be primarily concerned with frame up-sizing however downsizing is mentioned here for completeness.

Up-sizing algorithms can primarily be divided into non-edge adaptive and edge adaptive methods. These are typically linear and non-linear respectfully. Non-edge adaptive techniques trade-off the effects of moire and smoothness for the most visually pleasing result. Edge adaptive techniques try to achieve the best of each by changing the interpolation as a function of whether a pixel is near an edge or not. Both techniques shall now be considered in further detail.

This section details the start of work which shall fulfill the third short term objective of section 2.1. The results will also form part of the algorithm set proposed in section 4.2.1. These will justify the solutions to the long term goals of section 2.2.

4.3.1 Non-Edge Adaptive

The simplest method is nearest neighbour approach. This takes the closest pixel from the original frame to determine the new pixel value in the target frame. This is often unsatisfactory as it causes blocky artifacts.

Better are polynomial methods such as bi-linear and bi-cubic methods. These attempt to fit a polynomial between pixels to best estimate the value of new pixels. As the polynomial order increases the complexity of the algorithm scales exponentially. Bi-cubic interpolation (polynomial
degree three) is observed to give pleasing results in many cases and is often used as an arbitrary scaling technique.

\[
X = Adx^3 + Bdx^2 + Cdx + D \tag{4.7}
\]

\[
X = \alpha X_{-1} + \beta X_0 + \gamma X_1 + \sigma X_2 \tag{4.8}
\]

The algorithm is extended to the 2D case as seen in equation 4.13. Where \(X_{2d}\) is the resultant pixel value and the subscript to \(\alpha-\sigma\) denotes if interpolating in the \(x\) or \(y\) direction the values of these is as of equations 4.9–4.12. This also demonstrates that the algorithm is separable.

\[
\alpha = -dx^3 + 3dx^2 - 2dx \tag{4.9}
\]

\[
\beta = 3dx^3 - 6dx^2 - 3dx + 6 \tag{4.10}
\]

\[
\gamma = -3dx^3 + 3dx^2 + 6dx \tag{4.11}
\]

\[
\sigma = dx^3 - dx \tag{4.12}
\]
\begin{equation}
X2d = \sum \sum \begin{bmatrix}
X_{-1,0} & X_{-1,1} & X_{-1,2} & X_{-1,3} \\
X_{0,0} & X_{0,1} & X_{0,2} & X_{0,3} \\
X_{1,0} & X_{1,1} & X_{1,2} & X_{1,3} \\
X_{2,0} & X_{2,1} & X_{2,2} & X_{2,3}
\end{bmatrix} \ast \begin{bmatrix}
\alpha_{dx} & \beta_{dx} & \gamma_{dx} & \sigma_{dx} \\
\alpha_{dy} & \beta_{dy} & \gamma_{dy} & \sigma_{dy}
\end{bmatrix}
\end{equation}

Filtering

The next method is termed the filtering method. This requires the use of a low pass filter to determine new pixel values. This is also a popular technique due to the ease with which a filter can be implemented in hardware.

This method is again derived from the 1D case. A windowed sinc function is used, with its centre located at the position of \( X \) (our new pixel), see figure 4.7. An arbitrary number of input pixels can be used for the filtering. For the comparison four pixel locations are used, as this is the number used in bi-cubic interpolation.

The resolution of the filter is constant (at \( 1/16 \) in this example) to reduce the possible number of permutations of filter coefficients. That is the sinc function can be shifted to have its centre position falling over one of sixteen possible locations. The 2D kernel is formed from taking the product of the 1D kernel with its transpose, as in the bi-cubic case. This again demonstrates that this function is also separable.

For the comparisons coefficients from the Sony Anycast Station are used [55]. The window used is a blackman window slightly enlarged by a factor of \( 1/0.85 \).

4.3.2 Edge Adaptive

Edge-adaptive techniques often build on linear techniques such as bi-cubic to alter their behaviour around the edges of objects within a frame. Some of these will now be discussed.

LAZA [3], which is short for ‘locally adaptive zooming algorithm,’ is one such technique. It performs the following passes to an image to enlarge by a factor of two, it can be generalised to other scaling factors:

- Insert zeros between each pixel
- Loop through all pixels with even \( x \) and \( y \) coordinates. Determine if horizontal, vertical or diagonal edges and average those pixels which fall on the correct side of the edge boundary. The correct side is the side of the edge on which the target pixel falls. If the range of surrounding pixels is below a threshold interpolate average all pixels.
4.3 Resizing Algorithms

- Loop through all pixels with \( x \) or \( y \) odd. Check for edges as in the recent pass. There is however no modification if an edge is not present.

- Finally two more passes are made first through the even pixels then through the odd pixels as above. For each undefined pixel a histogram is made of four neighbouring pixels. These histogram bins are then averaged to give a pixel value for the undefined location. This process is known as re-bining.

For more information on this implementation see the original algorithm[3].

4.3.3 The Comparison

To determine which algorithms look best there are two factors to consider: which test images to use and how to decipher which is best. Historically the algorithm which gives a mathematically close result is not always the perceptively best. Therefore a wide range of test images including a zone plate, text based pictures and pictures containing regular patterns are used. The zone plate is a standard test image used for broadcast video with concentric circles of increasing frequency. This is useful for comparison of linear functions. Text based images exhibit the degradation of object edges well. This is because the original image has sharp edges. Images with regular patterns can also exhibit unwanted artifacts when up sampling.

Non-interlaced progressively scanned frames will be used for the comparison. For perception tests both still images and video will be compared. Video is important to ensure the non-linear functions do not introduce flickering artifacts.

The algorithms to be considered are as follows:

- Bi-cubic interpolation
- Filtering with ‘Anycast’ coefficients
- LAZA- locally adaptive zooming algorithm [3]

The first examples shown are for the three algorithms on text based images, see figure 4.15. The scaling is integer to account for the LAZA algorithm, which was implemented to only scale by multiples of two. The enlargement factor is times four as little difference is seen at two times.

The LAZA algorithm does a good job of finding the edges however introduces an artifact on the letter \( i \). The dot on the \( i \) is particularly well resized by the LAZA algorithm.
Comparing the filtering and bi-cubic approaches it is seen that the filtering gives a more smooth and satisfactory result. As with all image and video processing analysis this is subjective.

Figure 4.15: Sample outputs for resizing text by a factor of four (left to right: Anycast coefficients, Bi-cubic and LAZA methods)

Next is a scene showing the handle of an ‘umbrella’. Notice that the LAZA method does a good job of finding the bottom edge of the jacket, see figure 4.16. It does however introduce artifacts of small white and black dots. There are a number of variables for threshold values. These can be altered to trade edge detection against artifacts. There is naturally an optimum set of these coefficients for every different image.

An interesting comparison here is between the filtering and bi-cubic methods. It is noticed that the filtering method does a better job at preserving the handle on the umbrella. The bi-cubic method (centre) smudges the boundary. However there is more noise in the filtering method due to the ringing effects.

In summary for the edge enhanced method the result is not very pleasing and further methods should be prototyped to find a more satisfactory algorithm for comparison.

Due to the non-linearity of the LAZA algorithm comparison using the zone-plate is not meaningful. However the bi-cubic and Anycast coefficient methods are compared. This is an interesting result from an industrial view point as these are two frequently considered alternatives.

The zone plate shows the variation between the bi-cubic and filtering methods, see figure 4.17. Shown is is a subspace of the zone plate taken in the top right hand corner of the frame. It is seen that the filtering method has a larger amount of the ringing effect which is seen to be more prominent in the left hand image. The benefit of the filtering method however is the sharper line resolution. The bi-cubic method is seen to give more blurry edges.
4.3 Resizing Algorithms

Figure 4.16: Sample outputs for resizing a general image by a factor of four (left to right: Anycast coefficients, Bi-cubic and LAZA methods)

Figure 4.17: Sample outputs for resizing by a factor of 2.667 (left to right: Anycast coefficients and Bi-cubic methods)

The three methods were presented to a panel of experts at Sony Broadcast and Professional. The filtering method was identified as overall the most satisfactory over three sets of video clips.
The clips contained data from the first two examples of figures 4.15 and 4.16 as well as a scene panning Stockholm city centre. Re-sizing scalings of 2, 4 and 8 were used.

### 4.4 Summary

This chapter has covered three case study algorithms. These are primary colour correction, 2D convolution and image resizing. Further to this it has also looked at a model of the ARM AMBA bus architecture.

The bus model is work towards meeting objective (a) in section 2.2 and the first objective in section 2.1. This was shown to be effective for a simple model of two masters and two slaves communicating through a variable feature AMBA bus. This work has been continued over summer 2005 and a general bus model produced [51].

The primary colour correction and 2D convolution algorithms form part of a research paper to be presented at FPT 2005, see appendix A. These are two case studies which demonstrate the benefits and weaknesses of FPGAs and GPUs. The GPU has large computational processing power, but is limited in the number of memory accesses per pixel processed. An FPGA can handle memory more efficiency however requires an increasingly large device area as application sizes increase. Such case studies will be used to justify design decisions and assumptions made in completing the thesis objectives.

Finally, the image resizing section approaches an interesting industrial question of the best image resizing technique. This work is to progress to analyse the computational complexity of implementing ‘good’ techniques on different hardware. Good is defined by a qualitative analysis of the results. This again works towards the thesis aims by providing another sample case study algorithm.
Chapter 5

Future Work

In chapter 2 the research area and questions were presented. In this chapter the work which will be carried out to meet these requirements will be explained. The two key areas are:

- Implementation of further test applications on current hardware to demonstrate their strengths and weaknesses
- Enhancements to video processing systems which will improve performance of video processing applications

This chapter shall be divided into these two considerations. A brief discussion of the timescale for completing these tasks and the thesis layout follows.

5.1 Further Algorithms

Initially the work started on resizing algorithms will be completed. This will complement the primary colour correction and 2D convolution algorithms already investigated. Further to this other algorithms will be investigated which highlight the differences of FPGAs and GPUs. These will be used to back up the proposals which shall be made as suggested in section 5.2.

The further algorithms are chosen to be a good representative set of the type of functions performed in broadcast video. Colour correction, 2D convolution and resizing algorithms fit well to this model. Multiple resizing algorithms are explored to get a perspective view on the best of the algorithms and compare this to their implementation complexity. As a byproduct of choosing a broad range of algorithms a representative set of the types of functions performed in these algorithms is produced. Therefore there are algorithms with different combinations of number of memory accesses, vector arithmetic and other operations.
5.1 Further Algorithms

5.1.1 Interpolation

Following the unsatisfactory results from the LAZA edge adaptive resizing method seen in section 4 further methods will be implemented. Firstly an anisotropic diffusion method which uses bi-cubic interpolation [4]. Next one of my own proposed methods which adapts the current bi-linear method for edge adaptive interpolation.

Once satisfactory edge adaptive techniques have been found these shall be compared with bi-cubic and filtering based resizing. The aim is to qualitatively compare results result and indicate the computational complexity of each. That is with a view to implementation on an FPGA or GPU. This is towards producing a complexity model for comparisons of algorithms on both types of hardware, which is objective (a) of section 2.2.

Anisotropic Diffusion

There are three stages to this interpolation method. Firstly the image is resized, using the bi-cubic method, to be larger than is required. For the example of two times overall scaling, the image is initially increased by a factor of eight. The edges in this image are then sharpened using anisotropic diffusion as presented by Leu [56]. Finally the image is low pass filtered and downsized to the target resolution. The reason for over sizing the image is that noise is introduced in the anisotropic diffusion stage of the algorithm. The effect of down sizing after the image resize reduces the unwanted noise.

The interesting part of this algorithm is the second stage which is edge sharpening. There are three phases to this part of the algorithm.

The first phase determines the gradient magnitude and direction of each pixel. This is by convolving the image with the ‘Sobel’ operator in the horizontal and vertical directions and combining this result for the angle-magnitude representation.

Phase two generates intensity and magnitude indices for each pixel. This uses a $3 \times 3$ neighbourhood around the current pixel. There are three sets of indices for each pixel representing the low, middle and high position on an object edge. For further detail on this see the paper [56].

The final phase uses these indices to alter each pixel intensity. A pixel edge is a slope of increasing intensity in one direction. Intuitively if a pixel falls towards the bottom of this slope its intensity is decreased a little and if it falls towards the top of the edge it is increased. This has the effect of making the slope appear steeper and hence the edge sharper.

Further enhancements can be made to anisotropic diffusion. One is to avoid computation for
flat areas of the image and thus reduce computation time. This is done by checking the range of intensities over a pixel region before processing. Another is to over enhance the edges. That is by increasing the amount which is subtracted or added to a particular intensity.

The intuitiveness of the sharpening of the edges in this method would suggest a promising result.

**Approximation to edge-adaptation (bi-linear)**

An edge-adaptive techniques based on bi-linear interpolation is presented. This is a single pass techniques and can be converted from a discrete function to a continuous function weighting. Discrete algorithms often exhibit harsh blocky artifacts at object boundaries whereas with continuous functions a smooth transition is seen. There is potential to extend such an algorithm to bi-cubic interpolation however the computational complexity is likely to scale dramatically.

The LAZA algorithm attempts to avoid interpolation across an edge boundary. That is if the difference between two pixels is too large then they should not be interpolated between. A threshold determines what ‘too large’ means.

A bi-linear edge adaptive approach is proposed. Firstly the algorithm is considered discrete and converted to continuous.

Intensity differences are taken in the horizontal and vertical directions. A threshold is set to determine if there is an edge in either direction. The presence of a vertical or horizontal edge is indicated by the boolean $V$ and $H$ respectively. Further a $dx$ and $dy$ boolean are introduced to indicate which of the input pixels a target pixel is closest to.

A function is created for each pixel to determine whether it should be used in calculation or not. The functions for each pixel are shown below, where location $(0, 0)$ is the top left pixel and $(1, 1)$ the bottom right.

\begin{align*}
P_{00} &= \bar{V}.H + \bar{dx}.\bar{dy} + \bar{V}.\bar{dx} + \bar{H}.\bar{dy} \\
P_{01} &= \bar{V}.H + \bar{dx}.\bar{dy} + \bar{V}.\bar{dx} + \bar{H}.\bar{dy} \\
P_{10} &= \bar{V}.H + \bar{dx}.\bar{dy} + \bar{V}.\bar{dx} + \bar{H}.\bar{dy} \\
P_{11} &= \bar{V}.H + \bar{dx}.\bar{dy} + \bar{V}.\bar{dx} + \bar{H}.\bar{dy}
\end{align*}

These can be converted to a continuous function through two possible fuzzy logic principles. The first to replace logical ‘and’ with multiply and logical ‘or’ with addition. The second replacing
with maximum and minimum functions respectively. The choice is determined by the computational complexity for a particular hardware and the perceived quality of the result. The values \( V, H, dx \) and \( dy \) are also converted to continuous sense. The difference between pixel intensities is used for \( V \) and \( H \) and the actual offset for \( dx \) and \( dy \).

The min term results of bi-linear interpolation are weighted by these functions. The result is that a pixel contributes less to the result if it is likely to be across an edge boundary from the new pixel which is being calculated.

### 5.2 Enhancements to Video Processing Systems

In section 4 the benefits of using graphics hardware for video processing applications were shown. The fragment processor is the key component of the graphics hardware used in these implementations. The high bandwidth texture fetching available to the processor also helps provide a fast implementation. The fragment processor is a SIMD processor specialised for graphics type applications. There are two possible enhancements which can be made. Firstly embedded components which could be added to an FPGA to overcome current limitations. Secondly using a GPU in a system-on-chip.

#### 5.2.1 A GPU as part of a system-on-chip

Considering a system at a coarse level one may consider a system-on-chip design including both FPGAs and GPUs. As shown in chapter 4 these have complimentary features. The features make each suitable to video processing applications in different manners.

In his thesis Wiangtong [9] looks at the partitioning algorithms between hardware and software. He focuses particularly on the ultra-sonic reconfigurable platform. An extension to this is to look at the optimal method for partitioning an algorithm between GPUs and FPGAs. This can only be clarified through application of many typical video processing algorithms on both devices.

In extension such a comparison must also include some metric as to how the devices are connected. This can be built into such a model as a third variable. For example, a question may be asked does the performance benefit of dividing a task between an FPGA and GPU outweigh the degradation due to its interconnect?
5.2.2 Embedded components for an FPGA

The second objective set out in section 2.2 is to define what sort of components, developed by considering the GPU, could be embedded in an FPGA to improve performance for video processing applications. This includes features which facilitate the GPU’s computational horsepower for floating point vector operations. Such units would require some additional logic, as with a PowerPC, to communicate with the FPGA fabric. This divides the work neatly into two tasks of what sort of units could be implemented and how these would communicate to the fabric.

The features which were seen to be most appropriate in chapter 4 are the fragment processing core and its adjoining texture caching unit. Here the question to solve is what sort of embedded components are good for video processing applications?

5.2.3 Development of Tools

In fulfilling the research objectives it is likely to be necessary to alter existing tools or create new ones.

In section 3.5 existing FPGA and GPU tools are presented. To allow for simulation of using both cores together (as in objective (a) of section 2.2) it would be necessary to embed implementation of both hardware together. This would be facilitated through using a wrapper for one of the modules. The details of this are the subject of future work.

For the aim of new embedded components (objective (b) of section 2.2) a suitable hardware model of such components would be implement in the testing environment of the FPGA. From this measurements of projected performance could be made. This may require the editing of existing FPGA design tools. Again this is the subject of future work.

5.3 Gantt Chart and Thesis Layout

A Gantt chart detailing the expected time frame of the future work is detailed in appendix B. This is a plan of when I expect to complete the objectives.

During the first three months of 2006 the work and analysis on the interpolation algorithms is expected to be completed. This will include devising a method to analyse the complexity of the algorithms, drawing on past implementation experience. The preceptively best schemes will be implemented on the GPU and FPGA.

Later analysis leading to the proposal of new mixed core systems and the types of embedded cores will be performed. This is divided to allow three months for each. These sections will also
include further implementations to justify any assumptions and design decisions.

Previous work on interconnects and analysis from the research paper will be used to complete long term objective (a). That is to decide how best an algorithm may be divided between a GPU and an FPGA.

The interpolation algorithm and complexity analysis will lead to a definition of the type of embedded components which may be useful in an FPGA. That is to complete long term objective (b).

Throughout the next twelve months work reports shall be produced at the end of each milestone of work. This will be used to assess progress towards answering the objectives set out in chapter 2.

A brief outline of how the thesis will be laid out is as follows:

- Chapter 1: Introduction
- Chapter 2: Literature Review - extended version of chapter 3.
- Chapter 3: Proposed model of a mixed core architecture including its interconnects
- Chapter 4: Supporting case studies of chapter 3 model plus developed design tools
- Chapter 5: Proposed model of the type of components that would be useful for embedding in an FPGA
- Chapter 6: Supporting case studies of chapter 5 model plus developed design tools
- Chapter 7: Conclusion

The literature review is an updated and extended version of what has already been created. Chapters 3 and 4 include new work however this builds on the work already undertaken in chapter 4. Chapters 4 and 6 will include the implementations already proposed in section 4.2.1 along with the image resizing algorithms and further example case studies. Any tools that are developed or altered shall be included in these chapters.

It is seen that these chapters fit with the Gantt chart in appendix B. Long term objectives (a) and (b) are focused on in separate time frames.
Chapter 6

Conclusion

This report has summarise the thesis objectives, achievements so far and future work.

In chapter 2 the research area and objectives were described. Short term and long term goals were established. The short term goals are to investigate interconnects, graphics hardware for video processing and frame resizing schemes. These combine to support the long term goal of improving video processing systems. This goal is achieved through two objectives. The first to define the sort of embedded modules, for FPGAs, which may further improve performance. Secondly to investigate a mixed core solution of FPGAs and GPUs and it’s interconnect structure. Specifically learning from the fast floating point vector processing capability of graphics hardware.

A comprehensive literature survey covering the architectures and implementations of FPGAs and GPUs was given in chapter 3. The architecture of the GPU was demonstrated to have a number of parallel pipelines and processing elements. For the FPGA embedded components were highlighted specifically with how they are interconnected to the rest of the FPGA fabric. Sample applications utilising individual and multiple FPGAs and GPUs were shown.

Next chapter 4 showed in detail the work covered over the last 14 months. In summary. A bus model was created for the ARM AMBA AHB bus. Primary colour correction and 2D convolution were implemented on the FPGA, GPU and CPU. Frame resizing schemes of bi-cubic interpolation, filtering and an edge adaptive method have been compared. This last section is work on-going.

Future work was proposed in chapter 5. Presented is what will be required to answer the research questions. The work is divided into two sections of the sample video processing algorithms and developing an improved video processing system. The first of these areas is justification for the assumptions and design choices which will be made in the second.

To summarise the aim of my thesis is to improve current video processing systems through the introduction of processing cores and/or components, developed by considering graphics hardware.
Appendix A

Research paper to appear in proceedings of Field Programmable Technology 2005
Have GPUs made FPGAs redundant in the field of Video Processing?

Ben Cope¹, Peter Y.K. Cheung¹, Wayne Luk² and Sarah Witt³

¹Department of Electrical & Electronic Engineering, Imperial College London
²Department of Computing, Imperial College London
³Sony Broadcast & Professional Europe

{benjamin.cope, p.cheung, w.luk} @imperial.ac.uk, sarah.witt@eu.sony.com

Abstract

Since the 1990s FPGAs have been popular for accelerating video processing applications. This paper presents GPUs (Graphics Processing Units) as another viable solution, comparing their throughput with that of FPGAs. Previous work on using FPGAs and GPUs for video processing is analysed, grounds for comparison formulated and then exemplified through case studies of primary colour correction and 2D convolution. GPUs are seen to be advantageous in applications utilising their optimised instruction set and which have a low number of memory accesses. We found that for 2D convolution, the throughput of the GPUs exceeds that of FPGAs if the mask dimension is $2 \times 2$. For mask sizes greater than $4 \times 4$ and for primary colour correction FPGAs have higher throughput. The GPU implementation of primary colour correction is shown to be capable of a throughput of 63 MP/s (million pixels per second) enough for high definition video at 30f/s (frames per second). For 2D convolution GPUs achieved the target throughput rate of 8MP/s ($512 \times 512$ frames at 30f/s) up to size $7 \times 7$. The Spartan 3 FPGA was capable of over 60 MP/s throughput and the Virtex II Pro over 110MP/s at this size of convolution.

1. Introduction

Video processing algorithms comprise of a set of operations performed over frames of pixels. When accelerating such algorithms, we use FPGAs to prototype designs and often as the final solution. Their flexibility, with which the parallelism of an algorithm can be exploited, makes FPGAs an obvious solution. Operations, pixels or even frames can be processed in parallel. This paper demonstrates that GPUs also offer a viable solution, capable of exploiting parallelism and meeting the application throughput rate. This is demonstrated through case studies of primary colour correction and variable sized 2D convolutions.

The original contributions of this paper are: 1) the first comparison of FPGAs and GPUs in their use for video processing; 2) a discussion of grounds for comparison; and 3) sample case studies that identify relative strengths and weaknesses of FPGAs and GPUs.

The paper is organised as follows. Section 2 discusses background work in the area. Grounds for comparison are presented in Section 3. Section 4 describes the case study algorithms. Results and their analysis is given in Section 5. Section 6 concludes and suggests future work.

2. Background

When analysing GPUs and FPGAs, it is natural to compare their performance against that of CPUs. Because of its familiarity the CPU is chosen as a benchmark. We shall discuss previous work on GPUs and FPGAs comparing performance with CPUs.

GPUs are designed for computer graphics applications, but have been used to implement video processing applications since 2003 [1]. FPGAs were first used in the early 1990s [2]. The use of CPUs pre-dates both GPUs and FPGAs [2].

The late start for GPUs relative to FPGAs in video processing is due to the accessibility of their architecture. The introduction of high level shader languages, such as Cg [3] in 2003, made GPUs more accessible.

2.1. Architectures

The dataflow of an application is exploited in FPGAs through parallelism and pipelining. CPUs have limited potential for parallelism through SSE/MMX instructions or similar. The Pentium 4 can compute four operations of the same type concurrently using these instructions. FPGAs have one to two orders of magnitude greater throughput rates than CPUs [4, 5, 6, 7].

The graphics pipeline is the standard dataflow for the GPU designed for efficient independent processing of pixels [8]. A pipeline consists of vertex and fragment processors, the latter being used for video processing. Multiple pipelines (16 in the new GeForce 6800 series) are used to exploit system level parallelism in the GPU.
2.2. Computation

Here we consider factors in addition to parallelism and pipelining which speedup FPGA and GPU designs relative to the CPU.

An FPGA design implements a data path without instruction fetch and decode cycles. Embedded modules, for example multipliers, can be used to further improve throughput.

A GPU has a higher memory bandwidth, more floating point units and SIMD (Single Instruction Multiple Data) processing. The Nvidia 6800 Ultra has a peak performance of 40 GFlops per second compared to 6.8 GFlops per second for a 3GHz Pentium 4 [12].

2.3. Memory Access

The number of external memory accesses required in an FPGA design can be minimised through data streaming and reuse. This is made possible by the flexibility given by the reconfigurability of FPGAs. CPUs can also reuse previously accessed data within the limitations of local cache and register size.

Memory is used less efficiently in GPUs than in CPUs and FPGAs. For example, in computing 2D convolution a lookup is required for all mask elements, for each pixel, on each iteration. Two examples where the CPU is faster than the GPU are, local principle component analysis (PCA) size $7 \times 7$ [9] and the FFT [1]. For PCA and FFT the CPU was 1.5 and 4 times better respectively. Both designs used GeForce 5 series GPUs. Comparison was made to a 2GHz Pentium 4 for PCA and a 1.7GHz Intel Zeon for the FFT.

Current GPUs have a higher memory bandwidth, which will be shown to partly overcome these issues. The GeForce 6800 Ultra has a memory bandwidth of 35.2 GBytes per second compared to 6 GBytes per second for the 3.0GHz Pentium 4 [12].

2.4. Clock Rate

CPUs typically have 6 to 15 times faster clock rates [13] than FPGAs. This can even be higher (50 to 60 times) depending on the application. The CPU has a 6 to 8 times faster clock rate than a GPU. A GPU has a higher clock rate than FPGAs, 400 to 500MHz compared to 60 to 200MHz for new GPUs and FPGAs respectively. High clock rate is not always associated with a faster throughput rate [9, 13]. It is however linked with a higher power requirement.

2.5. Multiple Cores

Parallelism can also be exploited by using multiple cores [7, 14, 15].

An example for FPGAs is the Shape-Adaptive Template Matching (SA-TM) algorithm implemented on the Sonic architecture [14]. This shows potential for up to 28,600 times speedup over a 1.4GHz Pentium 4 when using multiple Virtex 1000E devices.

A GPU example is a motion tracking algorithm implemented on a GeForce 5900 [15], with a core clock speed of 400MHz. This performs 3.5 times faster than a 2.0GHz Athlon Processor with one device. Five GPUs perform the same task 4.5 times faster than a single GPU [15].

2.6. Number Representation

Fixed point representation is normally adopted on FPGAs, however floating point calculations are also possible. Multiply, addition and multiply-accumulate operations are commonly used in video processing. Over a six year period (1997 to 2003) FPGAs have improved in performance, on these operations, at a rate of two to five times per year [16]. This exceeds Moore’s Law of an improvement of 1.5 times per year for CPUs [12]. This growth has continued and all these operations are faster on FPGAs than CPUs [17]. Performing floating point calculations on FPGAs has a large area cost. Floating point multiplication at 140MHz, on a Virtex II Pro, requires 592 slices [16], 12% of slices in the XC2VP7 device.

The number representation on a GPU is normally floating point, however half and double precision are available [8]. GPU performance exceeds Moore’s Law also, improving at 2 to 2.5 times per year [12].

3. Comparison of Hardware

When considering which hardware to use for a task one considers many metrics including throughput, area and power. In our case study we focus on throughput, but shall also consider die area. We are interested in whether a device is capable of real time implementation of a target application.

3.1. Throughput Rate

For video processing on all hardware we consider throughput in terms of MIPS. This is how many million pixels can be processed per second.

For FPGA implementations one pixel is clocked in and out per clock cycle so throughput is the maximum clock rate.

For the CPU throughput is defined as the clock rate divided by the number of clock cycles taken for the application. If SSE/MMX instructions are used this is multiplied by the degree of parallelism (i.e. four).
Throughput for the GPU is the number of parallel pipelines multiplied by clock rate divided by number of cycles taken by the task.

### 3.2. Choice of Devices

The scenario we set is of a designer deciding whether to use a GPU or an FPGA for accelerating a video processing task. We consider primarily the capability of each device to perform the task.

Firstly, we choose Nvidia’s GeForce 6800 GT and Xilinx’s Virtex II Pro to compare. These are the current ‘state of the art’ available to all designers. Secondly, we consider the GeForce 6600 GT and the Spartan 3 as ‘low cost’ devices. We refer to the GPUs as 6800 GT and 6600 GT as opposed to the device names of NV40-GT and NV43-GT respectively.

The Pentium 4 3.0GHz processor is chosen to benchmark the comparison.

The 6600 GT has half the number of parallel pipelines of the 6800 GT (eight versus sixteen). The 6600 GT is on a smaller process technology of 110nm compared to 130nm for the 6800 GT. It also runs at a faster core clock rate, 500MHz compared to 350MHz for the 6800 GT.

New Virtex II Pro devices have a hybrid process technology of 90nm high speed transistors and 130nm metal layers. The Spartan 3 is on a 90nm process. Both these devices can exploit arbitrary pipelining and parallelism, within area limitations.

For comparison the Pentium 4 is on a process technology of 90nm. GPU manufacturers are behind FPGA and GPU manufacturers in shrinking their technology size to 90nm.

This choice of devices covers two scenarios. One, a designer wishes to get the best performance. Two, they are also concerned about low cost.

### 3.3. Relative Advantages of GPUs to FPGAs

If a design is well matched to the instruction set of the GPU one would expect a GPU implementation to be advantageous. Arithmetic operations on vectors and matrices of dimensionality three or four are the most apparent. Less apparent operations supported by the instruction set include: vector range limiting, $x^0$ and euclidean distance. Branching statements must be avoided by replacement with blending based conditional statements [12]. This is a first check in considering whether to use a GPU or an FPGA.

Next, we consider memory access. In graphics applications memory accessed by one pixel is in close proximity to that accessed by its neighbours. The number of accesses is also low. In a rendering example [3] one may require a texture lookup, specular detail and parameters for shadow and diffuse mapping per pixel. An application requiring random memory accesses such as Huffman decoding would not be suited to the GPU. Low memory usage in a regular manner is well suited to the GPU.

FPGA designs can be implemented in such a way to make efficient use of on-chip memory. This overcomes many of the limitations of the GPU and can make the FPGA a more attractive solution.

For both FPGAs and GPUs the compiler or synthesis tool used must make optimum use of the hardware. For the FPGA this includes the minimisation of fan outs and routing delays as well as optimum resource usage. A compiler for the GPU must optimally convert high-level functions into GPU assembly instructions. Cg includes a library of high level functions to facilitate this [3].

### 3.4. Die Area

Die area is often compared as it gives an indication of scalability for future improved performance. The GeForce 6600GT has a die area of $156\text{mm}^2$ [18]. The 6800 GT has more parallel pipelines and a larger process size than the 6600 GT. They are from the same family (GeForce 6 series) therefore have very similar architectures. From this we assume the 6800 GT to be larger than the 6600 GT.

The largest Virtex II Pro device is assumed to be 1.5 inch by 1.5 inch ($144\text{mm}^2$) [19]. We scale the die area, using CLBs (configurable logic blocks), to a smaller device. The XC2VP7 has 1,232 CLBs compared to the XC2VP125’s 13,904 CLBs. This gives an estimated area of $128\text{mm}^2$.

We assume the largest Virtex 4 device to also be $144\text{mm}^2$. The Virtex 4 is on the same process technology and has the same architecture as the Spartan 3. We scale in a similar manner as above from the Virtex 4 LX devices. The XC4VLX200 has 22,272 CLBs compared to 6,912 CLBs for the XC3S4000. This gives an estimated area of $448\text{mm}^2$.

The Pentium 4 has a die area of $112\text{mm}^2$ [20].

### 4. Case Study Algorithms

Our first case study is primary colour correction targeted at high definition video ($1560 \times 1080$ frame size at 30 frames per second.) The algorithm consists of the following range of operations: vector arithmetic, complex arithmetic, branching statements and memory accesses. It is also separable into modules which contain different proportions of the above operations. From this we can evaluate the performance of each hardware core.

The second case study, an $n \times n$ 2D convolution targeted at a frame size of $512 \times 512$ pixels also at 30 frames per second, focuses on memory accesses. This is a significant bottleneck for the GPU [1].

For real-time application we require target throughput rates of 63MP/s and 8MP/s for primary colour correction and 2D convolution respectively.
4.1. Primary Colour Correction

The design consists of three modules: Input Correct (IC), Histogram Equalisation (HE) and Colour Balance (CB). Each performs non-linear functions on an RGB signal from various inputs (see fig. 1).

IC and CB include colour space conversion of RGB to HSL (Hue, Saturation and Luminance) space, alterations of HSV and conversion back to RGB space. HE performs, among other operations, range limiting and a power function based on gamma. The modifications in HE and CB are performed conditionally on R, G, B or all channels. We now describe the above in more detail, identifying optimisations.

**Colour Space Conversion Sub-Module.** RGB to HSL space conversion can be broken down into conversion from RGB to yCrCb then yCrCb to HSL, and vice versa for HSL to RGB space.

Conversion of RGB to yCrCb space is a matrix multiply operation [21]. y is clamped to range [16,235] and (Cb,Cr) to range [16,240]. yCrCb to RGB conversion also requires a matrix multiply [21]. After conversion RGB is clamped to range [0,255].

The conversion between yCbCr and HSL colour spaces is computationally expensive. For this reason we define another colour space XYL, derived from yCbCr, which is equivalent to HSL.

The first terms X and Y are related to Cb and Cr as shown in equations 1 and 2.

\[
X = (C_b - 128)/112 \quad (1)
\]

\[
Y = (C_r - 128)/112 \quad (2)
\]

(X,Y) is related to hue and saturation as a co-ordinate representation of the polar space (H,S). H and S being vector angle and length respectively.

Equation 3 shows how luminance relates to y.

\[
L = \frac{(y - 16)}{(235 - 16)} \quad (3)
\]

The inverse of these equations is performed for conversion from XYL to yCbCr space, range limiting yCbCr as above.

**(X,Y) Modifications.** The required modifications are: saturation multiplied by SatGain(m) and added to SatShift(∆S). HueShift(∆H) is added to hue.

These modifications translate to a rotation and scaling of (X,Y). The new values (X', Y') are calculated using equation 4.

\[
\begin{bmatrix}
X' \\
Y'
\end{bmatrix} = K \begin{bmatrix}
\cos \Delta H & -\sin \Delta H \\
\sin \Delta H & \cos \Delta H
\end{bmatrix} \begin{bmatrix}
X \\
Y
\end{bmatrix}
\quad (4)
\]

where \( K = (m + \Delta S'). \) S is euclidian(X,Y), the pre modification saturation. The euclidian(X',Y') (new saturation), by definition, must be of range [0,1]. If euclidian(X',Y') is greater than one it is normalised to one. It is always greater than zero.

The modifications are simplified as SatGain is only applied in IC and SatShift only in CB.

**Input Correct Module.** IC consists of: Colour space conversion to XYL space. The (X,Y) modifications as shown above. Luminance is multiplied by LumGain, added to LumShift and range-limited to [0,1]. Finally conversion back to RGB space is performed.

**Histogram Equalisation Module.** The first HE modification to RGB is summarised by equation 5.

\[
RGB' = (RGB - BlackLevel) * BWDIFF \quad (5)
\]

BWDIFF is BlackLevel minus WhiteLevel. Next RGB' is range limited to [0,255]. RGB' is then set to the power 1/Gamma. The result is range limited to [0,255]. ChannelSelection selects whether to apply the changes to R, G, B or all channels.

For all hardware the power function, for varying gamma, is pre-computed and stored in memory.

**Colour Balance Module.** CB is similar to IC, except: Luminance is only effected by LumShift and range limited to [0,1]. As with HE it also includes conditional modification to decide which of the channels to modify. The original luminance (i.e. L after initial conversion to XYL space) is tested to be either < TH1, in range [TH1, TH2], or > TH2 depending on AreaSelection. If the condition is satisfied then RGB is modified as above. Otherwise it is left unchanged. Another AreaSelection condition allows modifications independent of L.

**Number Representation.** For the GPU and CPU we use floating point. This maintains accuracy and gives satisfactory results.

For the FPGA we use a bit-width optimised fixed point representation. Each RGB channel is represented as eight unsigned bits. For XYL a colour space conversion model was produced and bit-width optimised. The error dropped off exponentially to 1/2 RGB bit width at 18 bits for each XYL component. This is a fair choice to maintain accuracy.
4.2. 2D Convolution

The algorithm for 2D convolution, of mask size \( n \times n \), is shown in equation 6.

\[
p'(x, y) = \sum_{i=-L}^{M} \sum_{j=-L}^{M} h(i, j) p(x + i, y + j) \quad (6)
\]

If \( n \) is odd: \( L = M = (n - 1)/2 \). If \( n \) even: \( L = n/2, M = (n/2) - 1 \). \( p' \) is the new image, \( p \) the original image and \( h \) the filter mask.

For the purpose of our experiment we perform the filtering to each R, G and B component. We also assume the filter to be non-separable. The video is assumed to be of non-interlaced raster scan input.

4.3. General Optimisations

To ensure the GPU implementation is optimal the use of vectors and matrices during multiplication, addition and range limiting is maximised. Branching statements are replaced by blending-based conditional assignments. Inputs which require the same modification, independent of pixel value, are pre-computed on the CPU.

The FPGA implementation is fully pipelined in both case studies. This achieves maximum throughput at the cost of latency. In synthesis we make efficient use of embedded multipliers and LUTs. For 2D convolution the mode change to SSE extensions is not worthwhile, due to the small number of operations to memory lookups.

For the CPU implementation of primary colour correction, SSE extensions to the MMX assembly functions are used. For 2D convolution the mode change to SSE extensions is not worthwhile, due to the small number of operations to memory lookups.

Details and the actual code of the implementations are available elsewhere [21].

5. Results and Analysis

5.1. Test Conditions

Cg (compiler version 1.3) was used to implement the case studies for the GPU. Throughput was predicted for all GPUs using NVIDIA’s performance tool NVShaderPerf. The tool assumes one cycle texture lookup which is valid if there are no memory clashes.

The FPGA implementation was written in VHDL and Xilinx ISE Project Navigator 7.1i used. The in-built XSL synthesis/place and route (PAR) tool was used to optimise for speed with high effort. Post PAR speeds are used to predict throughput.

A Pentium 4 3.0 GHz hyper-threading enabled CPU with 1GB RAM was used. The design was implemented in C++.

5.2. Results of Primary Colour Correction

Full Algorithm. The results for the primary colour correction algorithm are seen in fig. 2. Throughputs for two GPU and two FPGA architectures are shown along with a CPU implementation as a benchmark.

Both FPGA implementations perform faster than the GPUs. This shows the advantage of the flexible pipelining and parallelism of FPGAs, over the fixed number of parallel pipelines in the GPU. The GeForce 6800 GT approaches the throughput of the Spartan 3. The high performance GPU is approaching the speed of a budget FPGA. Both FPGAs and GPUs perform faster than the Pentium 4.

The benefits of the GPU over the CPU are an optimised instruction set and up to 16 pipelines. The 6800 GT is clocked at 350MHz, 8.5 times slower than the CPU. It is 18 times faster in throughput. This shows the operational efficiency advantage of the optimised instruction set to be at least 9 times.

FPGAs benefit from a fixed point bit-width optimised design. This is another reason for their superior performance. GPUs and CPUs are disadvantaged for algorithms amenable to bit-width modifications, due to their rigid instruction set. A floating point FPGA implementation would have a larger area requirement and potentially slower throughput.

The XC2VP7 Virtex II Pro (75% slice utilisation) was used for implementing the colour correction algorithm. This is three times smaller than the 6600 GT and even smaller than the 6800 GT. The Spartan 3 used was the XC3S4000 which is over two times larger than the 6600 GT. The design uses 43 embedded multipliers but only 13% of slices. The Virtex II Pro has 44 multipliers so fits well to the application. The Spartan 3 has 96 multipliers and is the smallest device with over 43 multipliers. By replacing multipliers with LUTs a smaller device could be used at a cost of throughput rate.

The FPGA is shown to have a higher throughput
than the GPU for primary colour correction. However, as we see from the fig. 2 the 6800 GT and 6600 GT devices are also capable of performing at the target throughput rate (indicated by the bold line).

Individual Blocks. The blocks represent different types of algorithms. We classify them in Table 1, where N, L, M, and H represent Null, Low Medium and High usage. R2X is RGB to XYL conversion and X2R is XYL to RGB conversion. The FULL algorithm is shown last for completeness. Arith. and Branch. are arithmetic and branching instructions respectively. Results by module can be seen in fig. 3.

We see that the FPGA has more consistent throughput rates than the GPU, due to flexible pipelining and parallelism. The throughput rate of a GPU and CPU scales with number of clock cycles. R2X and X2R both contain a matrix-vector multiplication, vector addition and range limiting. R2X is slow, relative to X2R, on the GPU due to inconsistent range limiting over XYL (X,Y have a different range than L). This is not suited to the instruction set.

The X2R module range limits RGB to [0,255]. The instruction set is optimised to perform identical operations on vectors. Throughput of R2X and X2R is almost identical for the FPGAs and the CPU. Performance difference of IC and CB is most noticeable in the GPU implementations. CB is slower due to unavoidable branching statements. It takes the 6800 GT 23 clock cycles for CB compared to 16 for IC. The CPU is also slower for CB. The difference is less significant due to its general purpose instruction set. The FPGA implementation has similar throughput for both modules due to pipelining.

HE performs the fastest, on the GPU, of the three modules. This is because it takes only 12 cycles to compute. However it does contain a higher ratio of memory access and branching instructions which are associated with poor GPU performance. The FPGAs are again much slower than the GPU due to its higher clock rate and only one pixel pipeline.

The results show consistently higher throughput for the GPU over the FPGA. This is because of its higher clock rate and the small number of instructions relative to the full algorithm.

5.3. Results of 2D Convolution

The logarithm of the resultant throughput is shown due to the greater than exponential decrease of throughput for GPUs (see fig. 4).

The FPGAs throughput rate is more consistent relative to the GPUs. This is due to flexible parallelism, pipelining and streaming of data. This verifies the inefficient memory access of the GPU.

For convolution size $10 \times 10$ and $11 \times 11$ the designs are very large. For the Spartan 3 this meant a
greater decrease in performance. 99% of the device was used for size 11 × 11. The Virtex II Pro maintains a more constant throughput.

GPUs and FPGAs are capable of performing at the target throughput rate (8MP/s) for sizes up to 7 × 7. After this the throughput rate of the GPUs continues to fall and they become unusable for our application.

The Pentium 4 performance drops below 8MP/s at size 4 × 4. The performance with respect to the GPU shows improvement in memory access speed of the new 6 series GPUs over the old 5 series [1, 9].

To stream 2D convolution data the FPGA stores rows of pixels, for a 512 × 512 frame size, 512 × n × 24 bits (8 bits per R, G, B) are required. As n increases the design size and synthesis time increase rapidly. The 11 × 11 convolution requires the largest device of each family. For a larger frame size, such as high definition video and for large n such data streaming may not be possible.

As n increased the number of multiplies (3 + n²) exceeded the number of multipliers in the largest Spartan 3. LUTs can be used instead. Where both options were possible the one with the fastest throughput rate was taken.

For 2D convolution size 7 × 7 the Virtex II Pro XC2VP40 was required, it has 4848 CLBs so we estimate its area to be 503 mm². The Spartan 3 XC3S2000 was required which has 5120 CLBs and an estimated 332 mm² die area. Both FPGAs are larger than the 6600 GT. We assume the XC2VP40 to be larger than or equivalent in size to the 6800 GT.

5.4. Considerations for the GPU

The GPU implementation of the primary colour correction algorithm takes 132 instructions and 53 cycles/pixel. 2D convolution ranges from 28 instructions and 5 cycles/pixel to 811 instructions and 1364 cycles/pixel for 2 × 2 and 11 × 11 respectively.

Primary colour correction on average computes 2.5 instructions per clock cycle. 2D convolution size 11 × 11 computes 0.6 instructions per cycle. This is due to the 121 (11 × 11) memory accesses required per pixel. We see for size 2 × 2 the instructions per cycle is high. The GPU can perform texture lookups in the same cycle as arithmetic operations.

5.5. Further Factors to Compare

We identify factors which should also be considered to decide the ‘best’ hardware for a target application.

Power consumption. FPGA devices are predicted to operate at a lower power than both GPUs and CPUs. FPGAs have lower clock speeds and do not normally require heat sinks and fans.

Design time. This is expected to increase the appeal of GPUs. Cg code is higher level than hardware description languages and closer to proof of concept designs, typically in C/C++. Languages such as HANDLEC and SystemC aim to reduce the design time gap.

Redundancy. GPUs and CPUs are already in a computer system, FPGAs are added at extra cost. The GPU or CPU can be considered a free resource, if the current GPU or CPU is not fully utilised.

Multiple pipelines within FPGAs. Multiple pixels can be computed in parallel on an FPGA, if the video memory bandwidth is sufficient.

Multiple cores. If a device is not capable of the target throughput rate a number of them could be used [7, 14, 15]. Further experimentation showing the speedup would be required. Many GPUs may be capable of performing 11 × 11 2D convolution at the target throughput rate.
6. Conclusion and Future Work

We have presented an analysis of previous work on the acceleration of video processing tasks on FPGAs and GPUs. These were shown in comparison to the CPU. Later we identified our grounds for comparison. Test cases of primary colour correction and 2D convolution were used to demonstrate the strengths and weaknesses of each.

‘Have GPUs made FPGAs redundant in the field of Video Processing?’ Our conclusion is ‘No.’ GPUs do not give adequate throughput for applications with high memory usage. For example for 2D convolution size greater than $7 \times 7$ GPUs are not capable of our target throughput rate of 8MP/s and the FPGA is the only solution.

A high throughput is obtained with GPUs by efficient use of the instruction set, parallel pipelines and high core clock rate. For the FPGA pipelining, parallelism, data streaming and appropriate choice of number representation gives a high throughput.

The downfall of the GPU is seen in applications of high memory load with a more than exponential decrease in performance for 2D convolution. A limitation of FPGAs is area requirement for complex arithmetic functions, floating point arithmetic and storage for filters of large mask sizes.

FPGAs perform best for primary colour correction and 2D convolution of size $4 \times 4$ upwards. GPUs perform better than FPGAs for individual sections of colour correction and 2D convolution of size $2 \times 2$. GPUs cannot meet the target throughput rate of 8MP/s past convolution size $7 \times 7$.

For primary colour correction the Virtex II Pro FPGA device used was smaller in die area than both GPUs. For convolution size $7 \times 7$ the Virtex II Pro used was larger than the GeForce 6600 GT.

Future work would involve investigating the further factors to compare in section 5. Altera and ATI devices would also be tested to broaden the range of test cases.

Acknowledgements. We gratefully acknowledge the support provided by the UK Engineering and Physical Sciences Research Council, and Sony Broadcast & Professional Europe.

7. References


Appendix B

Gantt Chart for Future Work
## APPENDIX B - GANTT CHART FOR DEC 2005 - NOV 2006

### Compare Resizing Methods
- Anisotropic Diffusion
- Own and other methods
- Comparison of Results / Complexity

### Joint FPGA & GPU Systems
- Review work to date
- Investigate separable tasks
- Model both hardware cores

### Embedded FPGA modules
- Review current and proposed modules
- Investigate possibility of new modules
- Model performance of modules

### Reporting
- Report (possible paper) on resizing
- FPT '05 Presentation and Conference
- Report on joint systems
- Report on embedded modules
- Preparation for writing up

### Research Activities
- Ben Cope (Circuits and Systems)

---

**KEY:**
- Complete Task
- Holidays
- Slack

**Research Activities**
- Ben Cope (Circuits and Systems)
Bibliography


