

An FPGA Vector Co-Processing Core for Rapid Algorithm Development

Kurt Morgan, Simon Maskell
QinetiQ Ltd
Malvern, Worcestershire, WR14 3PS

Abstract

This paper describes a versatile vector co-processor implemented as a field programmable gate array (FPGA) core. Benefits are quantified on image processing operations, where the benefits of parallel processing are shown to be significant.

Introduction

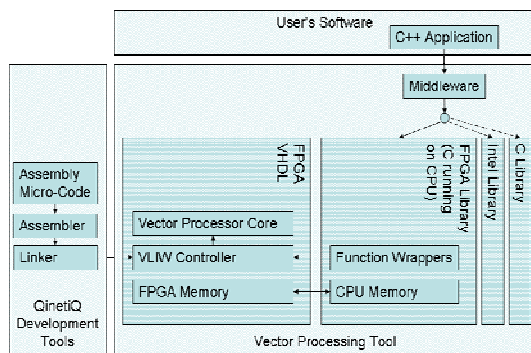
Moore's law growth in processing speed is currently growing by increasing the number of processors in devices. Dual and quad core processors are becoming commonplace in commodity computers. An extreme extension of this is the use of FPGAs, silicon chips which contain thousands of digital logic blocks which can be dynamically wired up to perform the stages of a specific algorithm in parallel.

FPGAs are commonly used in low volume military systems where they offer some of the performance benefits of completely customised silicon chips, whilst having the costs of off-the-shelf components. MOD research budgets are spent developing complex algorithms. FPGAs are the platform that provides the necessary processing power within the demanding size, weight and power constraints of the battle-field. A key problem that keeps these algorithms in the lab is the cost of developing FPGA designs for them, which cannot be justified without demonstrating their battle-field advantages in an FPGA implementation. There are many tools that already convert an algorithm from existing source code (in languages such as C or Matlab) to an FPGA implementation: GDAE, AccelChip or Handel-C (and is

being investigated by other EMRS DTC research [1]). The issue that prevents these tools from replacing the current low-level labour-intensive FPGA design is their runtime performance. Once the performance of an FPGA design has been addressed, redesign is also difficult. Automatic tools for laying-out (place and routing) hand optimised digital logic in an FPGA take hours to run. This means that FPGA designs can only be implemented once the design of an algorithm is stable.

The question this project addresses is how to demonstrate real-time performance within research projects. Commodity multi-core processors and coprocessors (eg graphics cards or ClearSpeed cards) are potential approaches. There are many algorithms which fit into these processors. There are other algorithms which do not partition efficiently onto separate parallel processors. This motivates the FPGA vector coprocessor. The FPGA fabric can be used to provide the customised data paths between parallel processors and customised logical operations on processors: The processor can be moulded to the algorithm. The coprocessor architecture provides a means to instantly reprogram the coprocessor as algorithms develop, without lengthy redesign cycles.

System Architecture



Co-Processor System

The Vector Co-Processor Core fits in a framework of software designed to allow an algorithm developer to benefit from FPGA acceleration, without needing to understand all the detail of the hardware. Prior to this project, QinetiQ developed an image processing library with implementations in C, using the Single Instruction Multiple Dataset (SIMD) extensions of Intel Processors and Graphics Cards. The vector co-processor has been integrated into this library, allowing existing applications to rapidly be converted to use the coprocessor as soon as the required instructions are added to the software library. The Image Processing Library has been enhanced to use Copy on Write (COW) pointers with a Least-Recently Used (LRU) cache: this minimises unnecessary copying of data.

The Vector Co-Processor Core is implemented on a Xilinx Virtex-II Pro FPGA, although the core could be used on other FPGAs with suitable libraries. It comprises of a very long instruction word (VLIW) controller which connects to a vector of processors. The processors have digital logic and data paths specific to image processing applications. The VLIW controller runs a micro-code program which controls all aspects of the operations performed by the processors. The core can be connected to any other hardware connected to the FPGA. It has been used as a coprocessor for a power PC processor

embedded inside an FPGA and for a desktop computer connected to the FPGA using gigabit ethernet.

If the vector co-processor does not contain the functionality needed as an algorithm develops, it can be customised in two ways:

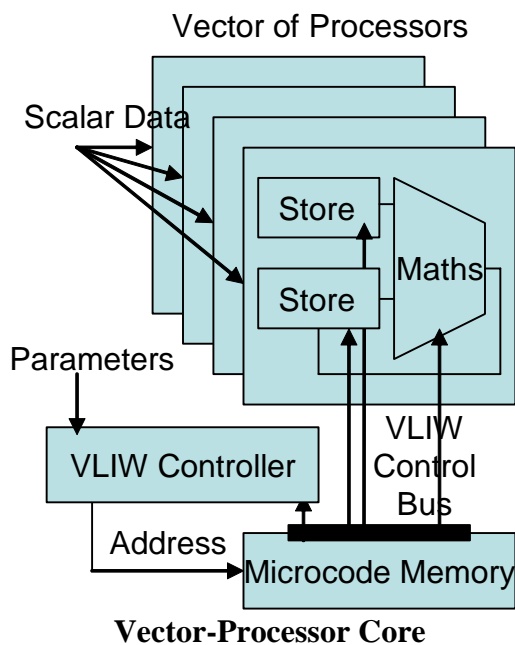
Without Hardware Changes - As the VLIW controller controls the processors at a very low level, a co-processor designed for a signal processing domain, such as image processing, can be used for many developments of an algorithms within that domain using only microcode changes, provided that the necessary hardware connections and digital operations are available inside the processor. For example hardware to do simple adding and subtracting has been used for pipelined differentiation using microcode changes. This does not require FPGA development tools or expertise.

With Hardware Changes - For use on significantly different algorithms, additional hardware can be added to the coprocessor, at the cost of a place and route cycle and requiring FPGA development tools. For example hardware designed originally for pixel arithmetic has been enhanced to operate on images with a mask. Additional processing to determine if the processed pixels are valid, as is needed in mosaicing images, takes place at the same time as pixel arithmetic by adding an extra bit to the images and adding hardware to process the bits in parallel with the existing arithmetic pixel operations. This would be particularly useful in adding bespoke data-paths where processors need to access data held in "adjacent" processors (eg to support novel parallel divide-and-conquer algorithms such as in [2]).

Vector Co-Processing Core

The vector co-processors implemented uses a vector of identical processor, connected to one VLIW controller. This is single

instruction multiple data-set (SIMD) processing. It has the benefit of being simple to understand and using a lot of parallelism, whilst only having to design one processor. As the core is implemented in a general purpose hardware description language (VHDL), other architectures could be developed, using multiple vectors of processors, non-identical processors, or a non-vector architecture, whilst still benefiting from the instant reprogramming of a VLIW controller. In the super-computing field, processors are often connected in a 3D torus. This allows nearest neighbour communication between processors, which allows pipelined communications in divide and conquer algorithms. The interconnections between processors are also used to perform logical operations. Again the vector processing core allows flexibility in this. As an example of this flexibility, tree logic has been connected to the vector of processors: this allows a vector to be added up separately from the VLIW controller.



The VLIW bus is so called because it needs to be over 100 bits wide to control all of the processors functionality. Most of the control signals of the processors are connected to the VLIW bus. The VLIW

controller implemented as a component controlled by the VLIW bus, just like the Vector of Processors. It is different in that it maintains the program counter, the pointer to the current address in the micro-code memory. The VLIW controller has some of the features of a simple general purpose processor, being able to conditionally add and subtract from the program counter and keep a stack of addresses needed for calling functions, using control 'parameters' input to the co-processor. The VLIW controller also synchronises the core with external interfaces, pausing it until data is available, or data can be written.

The Microcode Memory has been designed to use the RAM circuits embedded in Virtex-II FPGAs. This allows the microcode to be changed independently of the rest of the design, not requiring a place and route of the design. The time to place and route co-processors of different sizes is shown below. Note that each reprogramming of the VLIW controller saves a four-hour design iteration (Similar tools do exist for the embedded RAM circuits of other FPGA vendors). The microcode memory could be connected to another hardware interface to allow the microcode to be changed at run time or to allow for larger memories.

Parallel Processors	Build Time
4	13 min
32	237 min

Times for Place and Route

The very wide instruction bus does pose problems on FPGAs which are not designed for signals (nets) connected to many inputs (high fan-out). As the fan-out of a net increases the capacitance of the net increases, meaning that it operates at a slower speed. This problem has been overcome by putting the signal through a tree structure of digital buffers. This reduces the fan-out on each net at the expense of delaying the VLIW signals. This delay is only significant if the outputs of an

operation are required to start the next operation. Otherwise the delay is a pipelined step and does not affect the rate of processing. The buffer tree does use some of the finite digital resources on the FPGA. There is a trade-off between the resources used in a buffer-tree and the resources that would be used in duplicating the Microcode Memory and VLIW controller. Duplicating the controller would also reduce the place and route time by reducing the lengths of the nets to each processor. Vector co-processors with up to 64 processors have been implemented using the buffer-tree.

Programming System

The lowest level software libraries control the FPGA coprocessor, calling individual functions in micro-code memory. A flexible C interface to the vector co-processor has been designed which abstracts the way in which data is transferred to the co-processor, be that the QinetiQ Quixstream protocol over gigabit Ethernet or a direct connection to an embedded power-PC (similar to [3]). This allows programs to be written based purely on an understanding of what data inputs the micro-code of the VLIW controller requires.

The microcode functions can be written using spreadsheet software at a machine code level. Each segment of the VLIW bus is present as a row in the spreadsheet. The columns in the spreadsheet are the addresses in micro-code memory. The conditions which must be met for the next micro-code operation to be executed are also bits on the VLIW bus, which the VLIW controller tests against its internal status and external control signals. Laying out the microcode operations on a spreadsheet allows patterns in the instructions to be visualised and the data flow through the processors to be pipelined. The Excel spreadsheet application can be used to calculate parameters and manage the addresses of data in the code.

The micro-code programs in spreadsheets are linked together using a program that converts the numbers and symbols used in the spreadsheets into the binary contents of the micro-code memory. This tool is also responsible for calculating any function addresses which are marked as labels in the spreadsheet programs.

An assembly level tool has been written as part of another MOD project. This allows mnemonics to be assigned to sequences of instructions controlling components of the processors. A program can then be written as a sequence of these instructions. The tool is programmed with information about the dependencies of components in the processors, which allows it to automatically pipeline the instructions. The use of this system for describing components with the vector-processors developed in this project is the subject of future activities.

An application level developer will never need to use the micro-code functions which can be accessed in the lowest level software libraries, as they can use high level functions appropriate to their signal processing domain. This interface is however useful to those developing new algorithms as it shows them the sorts of functions which the hardware can efficiently implement and helps them to think in-parallel. Algorithms developed to use existing low-level operations will be easy to further accelerate by making new pipelined micro-code functions or adding new hardware to the processors.

Parallel Operations

A vector coprocessor has been implemented to perform the following image processing operations on masked images:

- Differentiation Pixels along Rows;
- Adding Pixels;
- Subtracting Pixels;
- Multiply Pixels;

- Calculating the Absolute Value of Pixels;
- Threshold Pixels And Store Result in Mask;
- Zero Pixels based on Mask.

These operations are used in one of QinetiQ's video tracking algorithms and are already implemented in C, using the Intel Performance Primitives (as part of the aforementioned Image Processing Library).

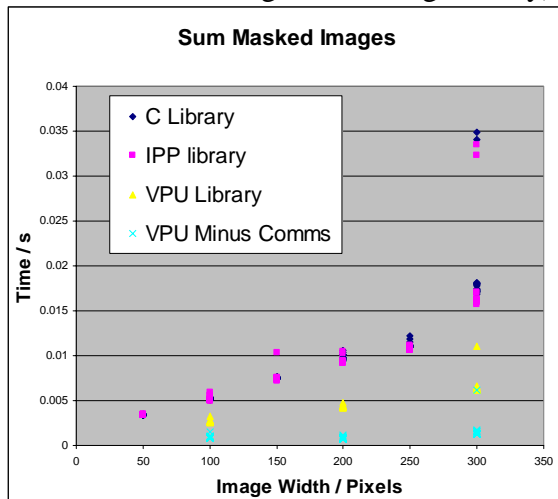


Image Processing Performance

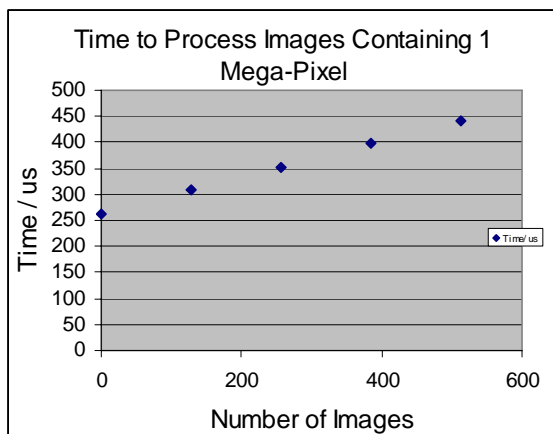
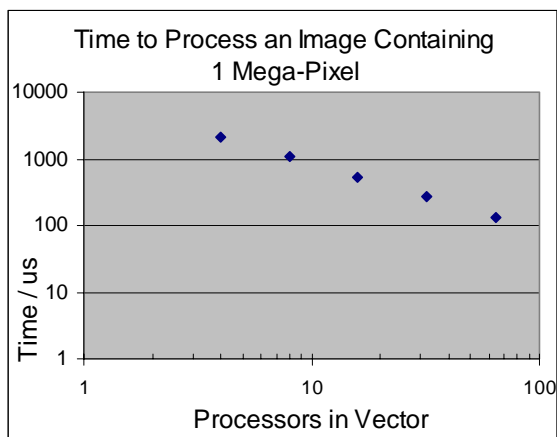
These speeds of the functions were compared for the different libraries. For the tests, the images were copied once to the Vector-Coprocessor for 128 operations, which is a realistic number of processing steps for an image processing algorithm. The summing operation is typical of the pixel arithmetic operations: the 32 way parallel processor (operating at 125MHz) completed the operations 2.8x faster than a desktop PC with a 2.2GHz Pentium VI Processor. The underlying processing operated 11x faster than the PC, but the performance of the co-processor was dominated by the bottle-neck of loading and storing images. This impressive speed improvement was possible because the process of calculating pixel addresses, loading pixels and performing the arithmetic happened in a pipeline. The mask images were also processed in parallel with the data. Non-masked images were processed 2x faster by the PC than

masked images. A similar FPGA with a more direct connection to the PC processor, such as the 16x PCI-X links used by graphics cards would have a communications bandwidth up to 32x faster, allowing the underlying speed improvement to be realised.

The factors limiting the underlying performance include the parallelism of the vector processor and the time taken to call individual functions. The VLIW controller does not support dynamic loop conditions, which means that the 'host processor' needs to make multiple calls to the co-processor to process a whole image from instructions that operate on fixed size patches of image. A more complete VLIW controller implementation with general purpose logic to control pointer arithmetic and allow the co-processor to complete large vector-tasks without host-processor communication is necessary to allow the host-processor and vector co-processor to work efficiently in parallel. A simple solution to this problem that was tried: a processor embedded in the FPGA controlled the co-processor. The instructions of this processor could not simply be pipelined with the co-processor processor's as they were programmed in different tools and connected over a bus. The timing of a more complete VLIW controller unit would be as well understood as the existing simple one, allowing it to work without stalling the co-processor.

The project so far has focused on parallel processing operations which are very simple to schedule on a vector co-processor. If an image is striped across a vector co-processors memory such that rows of image are put in increasing memory addresses, processing a 1024x1024 image on 32 processors could simply be a matter of scheduling 1024/32 operations which operate on whole rows of image. Processing 1024x1024/32 images each only 32x1 pixels would require 1024x1024/32 operations to be scheduled. Operating on

smaller data sets is more challenging as there many more new operations with setting-up overheads. As each address of VLIW microcode controls the whole pipeline of the processors, the entire pipeline has to be flushed before a new operation can begin, unless the next operation can be guaranteed to complete the remaining processing steps. The graphs below show theoretical times to differentiate 2^{10} pixels along image rows, assuming that they could all fit in local processor memory. For one large image, processing time is inversely proportional to the number of parallel processors. For a fixed number of parallel processors, processing time rises linearly with the number of images that the 2^{10} pixels are broken into.



Theoretical Along Row Differentiation Performance

The second year to this project will consider divide-and-conquer algorithms, where the number of processors is similar

to the same of number of data points. The design of the VLIW controller will need to be considered to improve the scaling of performance on small data-sets. Future work in a more applied MOD project will consolidate the image processing functionality developed thus far.

Conclusions

An FPGA co-processor core has been developed that can perform image processing operations. It has been shown that an FPGA Vector-Coprocessor can give a 3x processing speed improvement over a conventional processor on simple image processing operations and that performance improvements of 10x are achievable if appropriate hardware is used.

What is yet to be fully quantified is that a satisfactory trade-is made between Execution Time improvement and Development Time and how this compares to competing acceleration technologies.

Acknowledgements

The work reported in this paper was funded by the Electro-Magnetic Remote Sensing (EMRS) Defence Technology Centre, established by the UK Ministry of Defence and run by a consortium of SELEX Galileo, Thales UK, Roke Manor Research and Filtronic.

References

- [1] S Maskell, B Alun-Jones and M Macleod. "A Single Instruction Multiple Data Particle Filter". *Proceedings of Nonlinear Statistical Signal Processing Workshop 2006*.
- [2] A. Meena, J. McAllister and R. Woods, "Rapid Implementation of High End DSP System on FPGA-Centric Embedded Platforms", *EMRS*

DTC 4th Annual Technical Conference, Edinburgh, May 2007

- [3] Satnam Singh, “Programming Models for Parallel Systems: The Programmer's Perspective”. *IET FPGA Developer's Forum, October 2007*