An overview of FPGAs and FPGA programming; Initial experiences at Daresbury

November 2006 Version 2.0

Richard Wain, Ian Bush, Martyn Guest, Miles Deegan, Igor Kozin and Christine Kitchen
Computational Science and Engineering Department, CCLRC Daresbury Laboratory, Daresbury, Warrington, Cheshire, WA4 4AD, UK

Abstract

This report will provide a brief introduction to Field Programmable Gate Arrays (FPGAs), the key reasons for their emergence into the High Performance Computing (HPC) market and the difficulties of assessing their performance against that of conventional microprocessors. It will also discuss FPGA programming tools and the key challenges involved in programming these devices. As well as providing some background information on FPGAs and FPGA programming this report will cover our initial experiences of FPGA programming with specific reference to the Cray XD1 system.

Contents

1 Introduction 2
1.1 FPGAs and HPC 2
1.2 FPGA Performance 3
2 Programming an FPGA 4
2.1 Hardware vs. Software Design Flow 5
2.2 Hardware vs. HPC Software Practice 6
3 Programming Languages and Tools 7
3.1 VHDL & Verilog 7
3.1.1 Xilinx tools 8
3.1.2 FPGA Advantage Mentor graphics tools 8
3.2 Pseudo-C High level languages 8
3.2.1 Mitrion-C and Mitrion-IDE 9
3.2.2 Handel-C and Celoxica DK Design Suite 10
3.2.3 Nallatech DIME-C and DIMETalk 11
3.3 Other languages and tools 12
4 Initial experiences of FPGA programming at Daresbury 12
4.1 Development board VHDL 13
4.2 Development board Handel-C 13
4.3 Cray XD1 VHDL 14
4.4 Cray XD1 Handel-C 16
4.5 DIME-C 17
4.6 Conclusions 18

1 Introduction

1.1 FPGAs and HPC

A Field-Programmable Gate Array or FPGA is a silicon chip containing an array of configurable logic blocks (CLBs). Unlike an Application Specific Integrated Circuit (ASIC) which can perform a single specific function for the lifetime of the chip an FPGA can be reprogrammed to perform a different function in a matter of microseconds. Before it is programmed an FPGA knows nothing about how to communicate with the devices surrounding it. This is both a blessing and a curse as it allows a great deal of flexibility in using the FPGA while greatly increasing the complexity of programming it. The ability to reprogram FPGAs has led them to be widely used by hardware designers for prototyping circuits. Over the last two to three years FPGAs have begun to contain enough resources (logic cells/IO) to make them of interest to the HPC community. Recently HPC hardware vendors have begun to offer solutions that incorporate FPGAs into HPC systems where they can act as co-processors, so accelerating key kernels within an application. Cray were one of the first companies to produce such a system with the XD1, a system originally designed by OctigaBay [1]. SGI quickly emerged as the main competitor to Cray producing the RASC Module (v1) and later the RASC RC100 Blade each of which could be integrated with SGI Altix servers [2]. SRC computers produces a range of reconfigurable systems based on it's IMPLICIT + EXPLICIT architecture and MAP processor [3]. There are a number of other companies producing reconfigurable hardware for HPC, one of the most recent to enter the market being DRC computers who produce FPGA modules that can be plugged directly into an HTX slot on an AMD Opteron processor [4]. DRC have been working with Cray to implement this technology in the follow up to the XT3 [5]. Another interesting piece of hardware that has recently been announced by Nallatech is the H100 series of FPGA based expansion blades for the IBM BladeCenter [6].

All of the systems mentioned above are based on FPGAs supplied by a single company, Xilinx. Xilinx are the largest global producer of programmable logic devices [7] and the Xilinx Virtex series has been adopted as the FPGA of choice for HPC. The reality of double precision floating-point capable FPGAs really arrived in 2002 with the Virtex II Pro FPGA, which contains up to one hundred thousand logic cells as well as dedicated DSP slices containing 18-bit by 18-bit, two's complement multipliers. These chips can be clocked at up to 400MHz [8]. The Virtex 4, released in 2004, has built considerably on the Virtex II Pro with up to two hundred thousand logic cells, more DSP slices and more 18-bit by 18-bit multipliers, and now clocking up to 500 MHz [9]. Unfortunately, the 18-bit by 18-bit multipliers in the Virtex II Pro and Virtex 4 FPGAs are not well suited to the requirements of floating point IP cores. Xilinx have attempted to redress this problem by providing a larger number of 25-bit by 18-bit multipliers on their new Virtex 5 range of FPGAs [10]. The Virtex 5 is the first FPGA on a 65nm process and once again provides increased clock speeds, higher logic density and a number of other optimizations over previous generations. One of the distinct advantages of FPGAs over conventional microprocessors is that they still benefit from a clock-speed increase with each new generation of hardware. As an FPGA consists of an array of CLBs the array can simply be expanded as the process shrinks below 65nm avoiding the requirement for extensive re-engineering of the architecture for each new generation of chip.

The recent surge in FPGA interest from the HPC community (Spearheaded by OpenFPGA [11]) has come at a time when conventional microprocessors are struggling to keep up with Moore's law. This slowing of performance gains in microprocessors along with the increasing cost of servicing their considerable power requirements has led to an increased interest in any technology that might offer a cost-effective alternative. The use of FPGAs in HPC systems can provide three distinct advantages over conventional compute clusters. Firstly, FPGAs consume less power than conventional microprocessors; secondly, using FPGAs as accelerators can significantly increase compute density; and thirdly, FPGAs can provide a significant increase in performance for a certain set of applications.

1.2 FPGA Performance

While the headline performance increase offered by FPGAs is often very large (>100 times for some algorithms) it is important to consider a number of factors when assessing their usefulness for accelerating a particular application. Firstly, is it practical to implement the whole application on an FPGA? The answer to this is likely to be no, particularly for floating-point intensive applications which tend to swallow up a large amount of logic. If it is either impractical or impossible to implement the whole application on an FPGA, the next best option is to implement those kernels within the application that are responsible for the majority of the run-time, which may be determined by profiling. Next, the real speedup of the ***whole*** application must be estimated once the kernel has been implemented in a FPGA. Even if that kernel was originally responsible for 90% of the runtime the total speed-up that you can achieve for your application cannot exceed 10 times (even if you achieve a 1000 times speed up for the kernel), an example of Amdahl's law [12], that long time irritant of the HPC software engineer. Once such an estimate has been made, one must decide if the potential gain is worthwhile given the complexity of instantiating the algorithm on an FPGA. Then, and only then, one should consider whether the kernel in question is suited to implementation on an FPGA. In general terms FPGAs are best at tasks that use short word length integer or fixed point data, and exhibit a high degree of parallelism [13], but they are not so good at high precision floating-point arithmetic (although they can still outperform conventional processors in many cases). The implications of shipping data to the FPGA from the CPU and *vice versa* [14] must also come under consideration, for if that outweighs any improvement in the kernel then implementing the algorithm in an FPGA may be an exercise in futility.

FPGAs are best suited to integer arithmetic. Unfortunately, the vast majority of scientific codes rely heavily on 64 bit IEEE floating point arithmetic [15] (often referred to as double precision floating point arithmetic). It is not unreasonable to suggest that in order to get the most out of FPGAs computational scientists must perform a thorough numerical analysis of their code, and ideally re-implement it using fixed point arithmetic or lower precision floating-point arithmetic. Scientists who have been used to continual performance increases provided by each new generation of processor are not easily convinced that the large amount of effort required for such an exercise will be sufficiently rewarded. That said the recent development of efficient floating point cores [16] has gone some way towards encouraging scientists to use FPGAs. If the performance of such cores can be demonstrated by accelerating a number of real-world applications then the wider acceptance of FPGAs will move a step closer. At present there is very little performance data available for 64-bit floating-point intensive algorithms on FPGAs. To give an indication of expected performance we have therefore used data taken from the Xilinx floating-point cores (v3) datasheet [16]. Table 1 illustrates the theoretical peak performance for basic double precision floating point operations in the largest Xilinx Virtex-5 FPGA[1].

Device	Multiply performance Gflop/s	Add performance Gflop/s	Fused Add/Mult performance Gflop/s
Xilinx Virtex-5 LX330	**38**	**111**	**53**

Table 1 Performance of basic floating point operations running on a Xilinx Virtex-5 FPGA [16].

A 3.0GHz Intel Xeon (Woodcrest) dual-core processor has a theoretical peak of 24Gflop/s for double precision floating-point operations[2]. We can see that a Virtex-5 FPGA could potentially provide between 1.5 and 5 times the performance of the Xeon dual-core processor depending on the ratio of add/subtract to multiply/divide operators in a given FPGA design. This is a crude calculation but it provides some basis for believing that floating point calculations are in no way beyond the realms of

[1] To incorporate the maximum number of floating point operators on a single FPGA requires the use of 'logic only' cores rather than cores utilising the dedicated DSP slices available on the Virtex-5.

[2] This is the performance of both cores combined.

possibility for FPGAs. Recent work has provided further evidence of this [17-23]. In much of this work the performance data that are available are educated guesses based on the results of design synthesis rather than measurements from an implemented design. This data, like that presented in Table 1, is often impressive when compared to current high end microprocessors but still has to be proven on real hardware. As another example Figure 1 provides a summary of DGEMM performance on a range of existing and novel architectures [17, 24, 25]. The results for Virtex II Pro FPGA and the Cell processor are very impressive but again they are also theoretical

The aim of our work is in part to generate some meaningful performance values for kernels such as matrix multiplication running on the current generation of FPGAs. It is only when verified performance figures such as these are available that we will be able to make a judgement on the true value of FPGAs to floating-point intensive HPC applications. The proof is, after all, in the pudding!

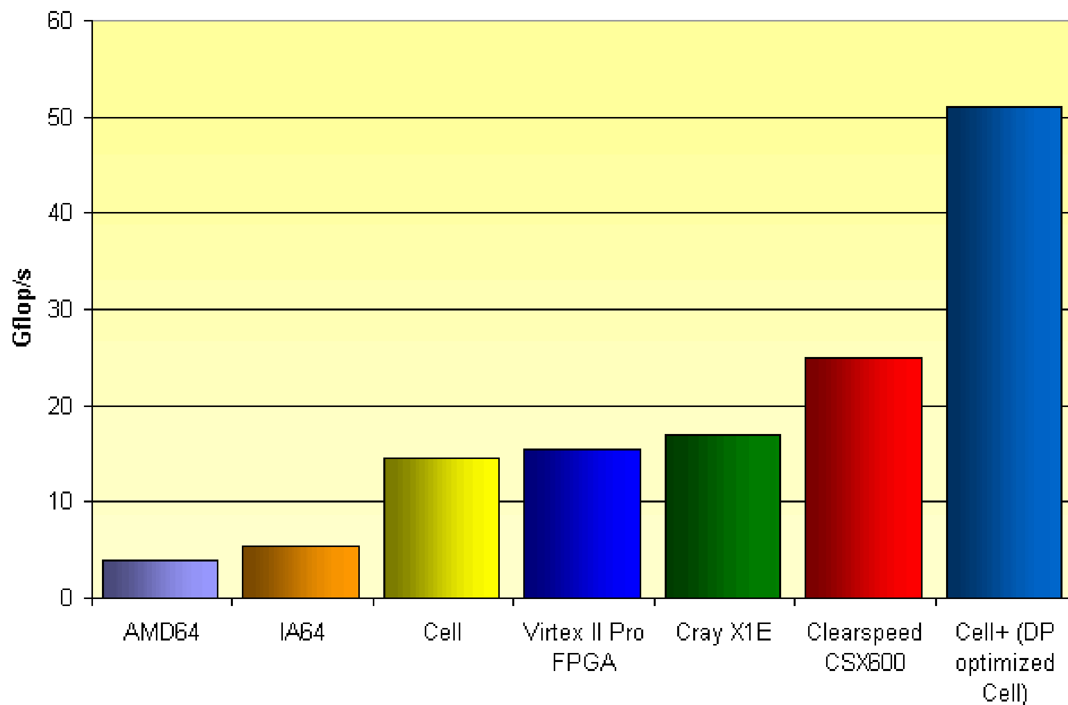

Figure 1 Performance comparison of DGEMM on a range of architectures [17, 24, 25].

2 Programming an FPGA

Although FPGA technology has been around in some form for many years it is only in the last two to three years that the technology has begun to make any inroads into the HPC market. In the past the vast majority of FPGA users would have been hardware designers with a significant amount of knowledge and experience in circuit design using traditional Hardware Description Languages (HDL) like VHDL or Verilog. These languages and many of the concepts that underpin their use are unfamiliar to the vast majority of software programmers. In order to open up the FPGA market to software programmers, tools vendors are providing an increasing number of somewhat C-like FPGA programming languages and supporting tools. These pseudo-C languages all provide a more familiar development flow that, in many cases, may provide a significant level of abstraction away from the underlying hardware. A number of traditional hardware design languages/tools and pseudo-c languages/tools are covered in greater depth later in this report.

2.1 Hardware vs. Software Design Flow

There are several differences between the traditional software design flow and the established Verilog/VHDL design flow for FPGAs. After designing and implementing a hardware design there is a multistage process to go through before the design can be used in an FPGA. The first stage is Synthesis, which takes HDL code and translates it into a netlist. A netlist is a textual description of a circuit diagram or schematic. Next, simulation is used to verify that the design specified in the netlist functions correctly. Once verified, the netlist is converted into a binary format (Translate), the components and connections that it defines are mapped to CLBs (Map), and the design is placed and routed to fit onto the target FPGA (Place and route). A second simulation (post place and route simulation) is performed to help establish how well the design has been placed and routed. Finally, a *.bit file is generated to load the design onto an FPGA. A *.bit file is a configuration file that is used to program all of the resources within the FPGA. Using tools such as Xilinx Chipscope it is then possible to verify and debug the design while it is running on the FPGA. The software design flow has no requirement for a pre-implementation simulation step.

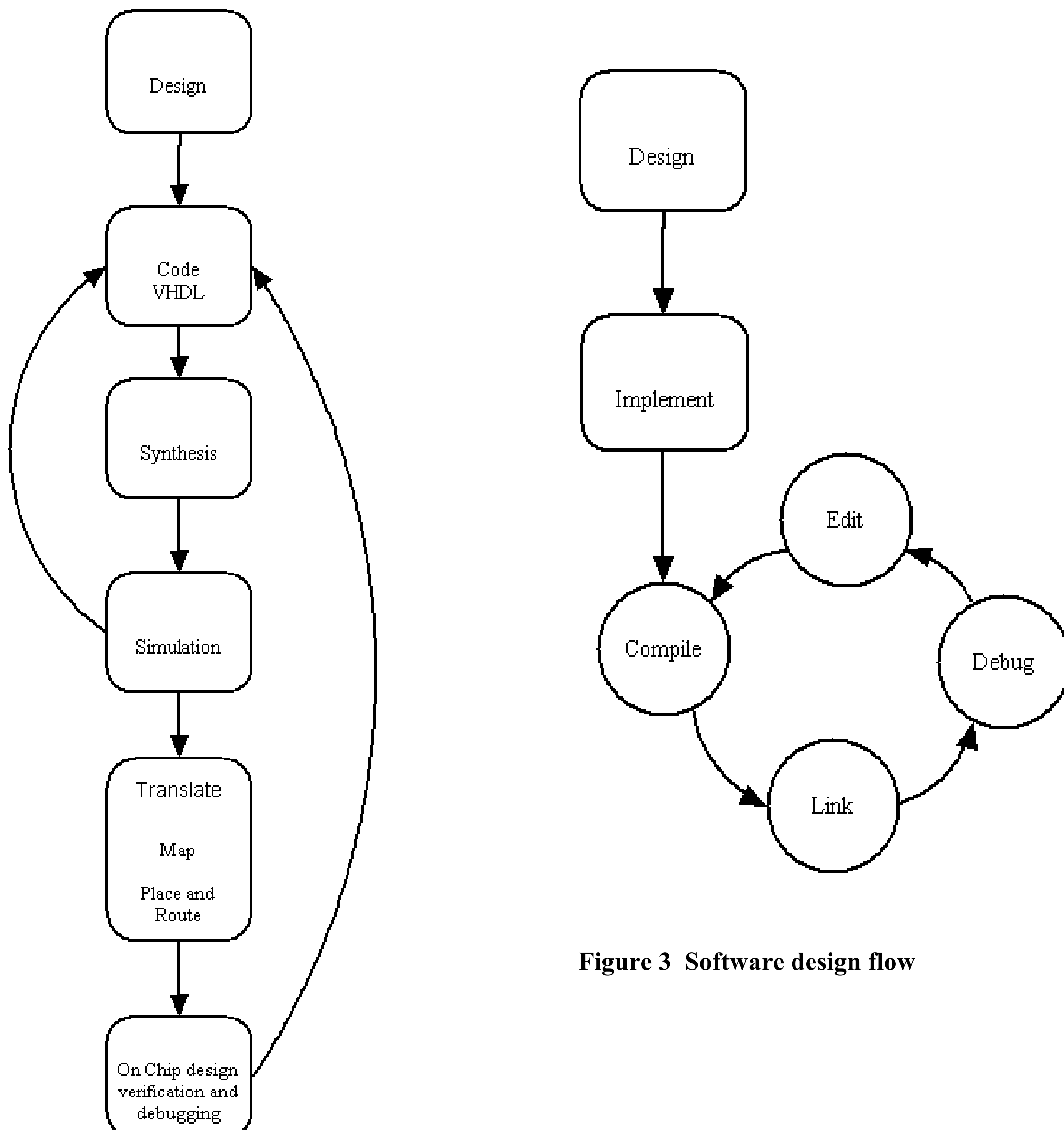

Figure 2 Hardware design flow

Figure 3 Software design flow

Compile times for software are much shorter than implementation times for hardware designs so it is practical to recompile code and perform debugging as an iterative process. In hardware it is very important to establish that a design is functionally correct prior to implementation as a broken design

could take a day or more to place and route and could potentially cause damage to system components. Figures 2 and 3 illustrate the differences between software and hardware design flows.

2.2 Hardware vs. HPC Software Practice

Despite the above comments, to the experienced HPC software engineer the design flow of the two paradigms does not seem to be fundamentally dissimilar. Though for hardware there are appreciably more steps, in general this is more often due to having to explicitly perform steps that are nowadays redundant or hidden in the software case. After all it is not so long ago that one of the authors was debugging software on a simulator!

This observation in many ways summarizes the differences in design flow between the software and hardware environments. To put it simply, and bluntly, HPC reconfigurable hardware programming is in its infancy, and much of what the software engineer would expect has yet to evolve to the level for which he/she would hope.

As a prime example HPC software engineers expect, indeed demand, cross-platform standards. These abstract the scientific problem from the hardware, thus enabling the software engineer to design his or her solution independently of the hardware so enabling the software to be run in many different environments. Nowadays this is the case whether the software is serial or parallel, message passing or multithreaded: For a careful and experienced HPC software engineer code once, run anywhere is now a truly achievable goal. However for hardware this is, as yet, a very far distant goal.

As usual the problem is I/O, using the latter term in its most general sense. In fact in all of the languages considered here the real problem is not how to express the scientific algorithm, it is much more how to get data to that algorithm, and how to get the results back. For HPC this is now a very well defined problem. One uses the facilities defined in the ISO standard of your language of choice for disk access, MPI to obtain data from another process, and the = sign to get data from memory. And that sums up the problem; FPGAs *a priori* know **nothing**. When you get them they are a blank chunk of semiconductor for you to warp and change to your desire. So they know nothing about memory; they don't know what kinds of memory are available, what pins the memory is attached to, or anything else you might like to name. And so while you can easily express your problem, the first thing you may have to do is write code to access the outside world. It's like when you sit down to write Fortran having first to devise your own memory, disk and video controller. FPGAs don't even know *a priori* where they are getting a clock from.

Of course it is not quite as bad as all that; vendors often provide cores for such low level I/O operations. However these are often specific to a given part, thus hampering portability of even the simplest code and so appreciably increasing the cost of maintenance of the application. Further these are often designed with the hardware engineer in mind, and to a software person the "API" may often appear at best cumbersome, at worst impenetrable. For instance does the software engineer really know enough to handle parity checking the data read from the RAM? All in all, the road from software to hardware programming is long and precipitous, and just learning to "talk the talk" is an achievement in itself.

But why should this be so? Well it is partly because FPGAs are not designed with HPC in mind, and in fact their use by HPC is another example of the long tradition of HPC trying to use any hardware that looks "interesting." FPGAs are more often used in the electronics industry in situations where the cost of development of an ASIC can not be justified. As such once debugged the FPGA is never re-programmed, and the program will be specific for this electronic device; portability is just not an issue. It is also because the field is very new and standards have not evolved, and as yet are showing no sign of evolving. In fact there is a strong similarity to the early days of the use of parallel computing in HPC. But ultimately one of main problems is, in fact, the power of the FPGA itself. This derives very directly from its great flexibility, and as such programming a given FPGA will strongly depend not only on the FPGA itself, but also on how it is packaged, what components are directly attached to it, and how it is connected to the host CPU, resulting in a plethora of details with which the neophyte must become acquainted before the item of interest, the scientific algorithm, can be implemented

3 Programming Languages and Tools

3.1 VHDL & Verilog

Both VHDL and Verilog are well established hardware description languages. They have the advantage that the user can define high-level algorithms and low-level optimizations (gate-level and switch-level) in the same language. A basic example of VHDL code, the evaluation of the Fibonacci series, is shown below, and it is a good example of the points made above. The code itself is reasonably straightforward for a software programmer to understand, provided that he/she understands that this is a truly parallel language and all lines are executing "at once". It is also straightforward to simulate a simple design of this nature. However it is surprisingly difficult to implement it in hardware and this difficulty is a direct result of I/O issues. As noted above for a design to work in hardware access is required to resources that are external to the FPGA, such as memory, and an FPGA is, by its very nature, unaware of the components to which it is connected. If you want to retrieve a value from main memory and use it on the FPGA then you need to instantiate a memory controller. While systems such as the Cray XD1 provide cores for communicating with memory, such cores are still complex and unfamiliar to software programmers. Our early experiences with VHDL have indicated that it should only be used for FPGA development if you are in a position to work closely with experienced hardware designers throughout the development process.

Fibonacci series example in VHDL [26]:

Adapted from http://en.wikipedia.org/w/index.php?title=VHDL&oldid=115145

```
1 : entity Fibonacci is
2 : port
3 : (
4 :     Reset       : in    std_logic;
5 :     Clock       : in    std_logic;
6 :     Number      : out   unsigned(31 downto 0)
7 : );
8 : end Fibonacci;
9 :
```

In VHDL the `'entity'` declaration is like a declaration of a function in C++. In this case, the '`entity`' declaration defines a device called Fibonacci. The block of code in parentheses after the word `'port'` describes the I/O behaviour of the entity. Lines 4 and 5 define the input ports `'Reset'` and `'Clock'` and declare them to be of type '`std_logic`' (i.e. 1 bit wide). Line 6 defines the output port `'Number'` as an unsigned 32 bit value output. Line 8 tells us that we are at the end of the description for the entity called Fibonacci.

```
10: architecture Fib of Fibonacci is
11:
12:     signal  Previous    : natural;
13:     signal  Current     : natural;
14:     signal  Next_Fib    : natural;
15:
16: begin
17:
18:     Adder:
19:     Next_Fib <= Current + Previous;
20:
21:     Registers:
22:     process (Clock, Reset) is
23:     begin
24:         if Reset = '1' then
25:             Previous <= 1;
26:             Current  <= 1;
27:         elsif rising_edge(Clock) then
```

```
28:              Previous <= Current;
29:              Current  <= Next_Fib;
30:          end if;
31:      end process Registers;
32:
33:      Number <= to_unsigned(Previous, 32);
34:
35: end Fib;
```

The 'architecture' statement in VHDL is like the actual function in C++, it describes the Fibonacci series logic. Line 10 defines the architecture name of the Fibonacci entity to be 'Fib'. Lines 12 to 14 define signals used within the logic. These signals are much like variables in C++. Everything from the 'begin' statement on line 7 to the 'end' statement on line 26 defines the algorithm itself. Line 19 assigns the sum of 'Current' and 'Previous' to 'Next_Fib'. Lines 21 to 31 define a process called 'Registers' which executes when there is a change in the 'Clock' or 'Reset' signal. If 'Reset' is 1 then 'Previous' and 'Current' are 'Reset'. If 'Reset' is not 1 then on each clock cycle, 'Previous' will be assigned the value of 'Current' and 'Current' will be assigned the value of 'Next_Fib'. Finally line 33 assigns the value of 'Previous' to the output port 'Number'.

3.1.1 Xilinx tools

The Xilinx tools provide a fully functional VHDL and Verilog development environment with a full range of editing, synthesis, simulation and implementation tools. These tools are required regardless of whether you actually use the editing/synthesis/simulation parts of the tool suite, because all of the tools (including the pseudo-C tools) mentioned here require vendor specific place and route tools in order to place and route designs onto that vendor's FPGAs. There are some third party tools that can place and route designs onto Xilinx chips, but usually the Xilinx tools are someway ahead in terms of compatibility with newer hardware. The Xilinx tools are relatively user friendly but for the design section of the tool flow they are not as powerful as those provided by Mentor.

3.1.2 FPGA Advantage Mentor graphics tools

The Mentor graphics tools are used widely by electronic engineers in preference to the Xilinx tools. They provide advanced visual design tools and the market standard simulation tool (modelsim) integrated into a single graphical environment. While it is easy to understand why hardware designers are so enthusiastic about the Mentor tools it would be difficult for any beginner with a software background to make effective use of them.

3.2 High level languages and tools

Without close collaboration between hardware and software designers it is highly unlikely that HDLs will prove viable for the development of accelerated scientific codes. In many universities and laboratories hardware and software engineers are separated geographically and by a significant technical language barrier. In such cases scientists are likely to want to adopt an alternative programming model relying on high level languages so that they can carry out hardware design tasks themselves. There are now a number of maturing pseudo-C languages that can be used for hardware design. These languages and their associated tools usually provide a much faster time to solution for a given problem but they do tend to sacrifice efficient use of the FPGAs resources to achieve this.

But why develop new languages when the vast majority of HPC software is written in Fortran, C or C++? To program FPGAs efficiently a very high degree of very fine grained parallelism is required, and either the compiler must be able to detect such parallelism, or the language express it, or both. It is well known in the HPC community that this is very difficult to achieve for standard languages, and auto-parallelization is rarely used by that community because of its deficiencies. Further the standard HPC methods of expressing parallelism, MPI or openMP, are poorly suited to FPGAs given this requirement for very fine granularity of parallelism. As a result if one wishes to use a high level

programming approach to FPGAs it is usually necessary to use a specially designed language. This is not always the case and there are some high-level approaches – notably SRC Carte, Nallatech DIME-C compiler and the open source Trident compiler – that do take the auto-parallelization approach. These compilers take input based on a subset of ANSI-C (and/or Fortran in the case of SRC Carte) and compile it to hardware. We have evaluated a number of high level tools while working with FPGAs and further details of these are presented below.

3.2.1 Mitrion-C and Mitrion-IDE

Mitrion-C is a C-like FPGA programming language that has been developed recently by Mitrionics [27]. It is very much pitched towards the needs of scientific programmers. The tools provided include the Mitrion Integrated Development Environment (IDE) and the Mitrion Virtual Processor. The IDE provides all the tools necessary for synthesizing and simulating Mitrion-C code as well as standard libraries and libraries for hardware support and simulation. The only tools that Mitrionics do not provide are the place and route tools that are specific to the vendor of the targeted FPGA, as the output of the Mitrion tools is a VHDL IP core for the target architecture [28]. This IP core is, effectively, your algorithm targeted for the Mitrion Virtual Processor. This provides full abstraction from hardware by providing a reconfigurable processor that automatically fits your algorithm into the resources on the FPGA and connects inputs and outputs to appropriate pins so that communication with memory and other external devices can take place. By doing this the Mitrion Virtual Processor allows the user to concentrate on the design of the algorithm while spending less time worrying about integrating the algorithm with the surrounding system components. However, this abstraction from hardware does come at a cost. The Mitrion Virtual Processor restricts the clock speed for a design to less than half of the peak clock speed for a given FPGA, significantly reducing the performance that can be achieved. The Mitrion-C language itself is relatively painless to learn.

Fibonacci series example in Mitrion-C [29]:

```
uint:22<30> main()
{
  uint:22 prev = 1;
  uint:22 fib = 1;

  uint:22<30> fibonacci = for(i in <1..30>)
  {
    fib = fib+prev;
    prev = fib;
  } ><fib;
} fibonacci;
```

At first glance this appears to be similar in some ways to C. In fact it is very different. There is not sufficient room here to go into the details, but these stem from the language having to express parallelism that a compiler can detect. The main difference one should note is that, like VHDL or Verilog, everything is running in parallel, i.e. all lines of the code are running “at once,” and it is up to the compiler to detect data dependencies to work out how to link things together. This requirement on the compiler places further restrictions on the language, the most important of which being the complete lack of pointers. At this point it is clear that the resemblance to C is purely superficial. However, the language does allow the developer to express very fine grained parallelism in a very straightforward way. In particular not only is “wide parallelism” readily expressed, so is “deep” parallelism. The latter is probably less familiar to the HPC software engineering than the former, which is simply numerous execution units all acting in parallel on different data sets. Deep parallelism is, in fact, pipelining, and this method is vital to obtaining the best performance from a FPGA.

Another feature of Mitrion-C that is slightly unusual in the FPGA world is that not only are floating point data types native to the language, arbitrary width is supported. As such not only is full 64 bit IEEE floating point arithmetic immediately available to the programmer, it is also straightforward to investigate widths between 32 and 64 to see if they are accurate enough for the algorithm [30], and if such widths are appropriate it will be possible to fit more onto the FPGA.

3.2.2 Handel-C and Celoxica DK Design Suite

Handel-C is another C-like FPGA programming language, but again it is in fact somewhat different. In fact the closest similarity is to Occam (used for programming transputers), which is unsurprising as both were developed at Oxford University, and both were designed to express parallelism. Now developed commercially by Celoxica [31], the language is more mature than Mitrion-C, having a 10 year history, and has been primarily focused on the embedded market. In the last two to three years Handel-C has been pushing further into the high performance reconfigurable computing (HPRC) domain. However because Celoxica do not provide an equivalent to the Mitrion Virtual Processor, interfacing with memory and other resources external to the FPGA requires more consideration than for Mitrion-C (though Celoxica do provide good platform support libraries (PSL) for a wide variety of FPGA boards that very much help this). This along with a number of syntactic oddities makes Handel-C more cumbersome to work with, but there are several distinct advantages to the language. Firstly, it provides greater flexibility to make low level optimizations once a design is functioning correctly. Secondly, there is a greater body of work and therefore a greater network of support available for Handel-C, due to its greater maturity. Thirdly, the language is more C-like than Mitrion-C which may provide an easier migration path. Finally, and possibly most importantly, the language and its associated tools are available to academic institutions for a fraction of the cost of the Mitrion-C tools. In our opinion the Mitrionics tools may provide the simplest means of getting code to run on an FPGA but the Celoxica tools provide the best value for money option.

Fibonacci series example in Handel-C [32]:

```
void main()
{
  unsigned int 100 prev, fib;
  unsigned int 7 n;

  par { fib = 1; prev = 0; n = 0; }

  do
  par {
    fib += prev;
    prev = fib;
    n++;
  }
  while (n != 30);
}
```

As noted above, it can be seen that Handel-C is more C like in its appearance than Mitrion-C, though Occam programmers will immediately recognize the `par` construct, and will be pleased to hear that there is a corresponding `seq`! This points at the major difference between Handel and Mitrion-C; in Handel-C you have to explicitly declare that a number of statements will be executed in parallel, while in Mitrion-C they will always be. This is actually a mixed blessing, for while Handel-C provides the software engineer with a language much closer to one that he/she is used to it does make it somewhat more cumbersome to express very fine grained parallelism in a natural way. The similarity may also lead the programmer into coding in a "C-style," because of his/her prior experience, rather than using the extra facilities that Handel-C provides to obtain good performance on a FPGA. However Handel-C is an excellent tool and given a good platform support library should enable an experienced software engineer to program a FPGA relatively easily.

One must also note that unlike Mitrion-C, Handel-C does not support floating point entities as native data types in the language. Support for 32 bit floating point data is available through a set of macros, but unfortunately Handel-C does not yet support 64-bit floating point arithmetic. However, this support is soon to be provided as Celoxica are developing their own floating point libraries that are already available as a beta release.

3.2.3 Nallatech DIME-C and DIMETalk

Nallatech's DIME-C language is based on a subset of ANSI-C. This gives it a number of obvious advantages over both Handel-C and Mitrion-C. Firstly, the programmer does not need to learn the syntax or semantics of a new language in order to use DIME-C. They simply have to learn which parts of ANSI-C cannot be used. Secondly, DIME-C code can be compiled and debugged using a standard C compiler.
The DIME-C compiler tool provides an IDE for DIME-C development compilation and debugging. The tool has some problems. Compiler warnings and error messages are not as informative as those provided by most mature C compilers and the code editor has limited functionality when compared to most popular IDE's. Despite these issues the DIME-C tool provides a reasonable environment for development and is well integrated with Nallatech's 'Application Builder' tool DIMETalk. DIMETalk enables the user to create a network of hardware components on an FPGA. These components might include block rams, user code, PCI-X bridges, etc. Components can be wired together in DIMETalk and a bit stream for a finished FPGA design can be generated. Figure 4. shows a screen shot from the DIMETalk tool displaying a simple network.

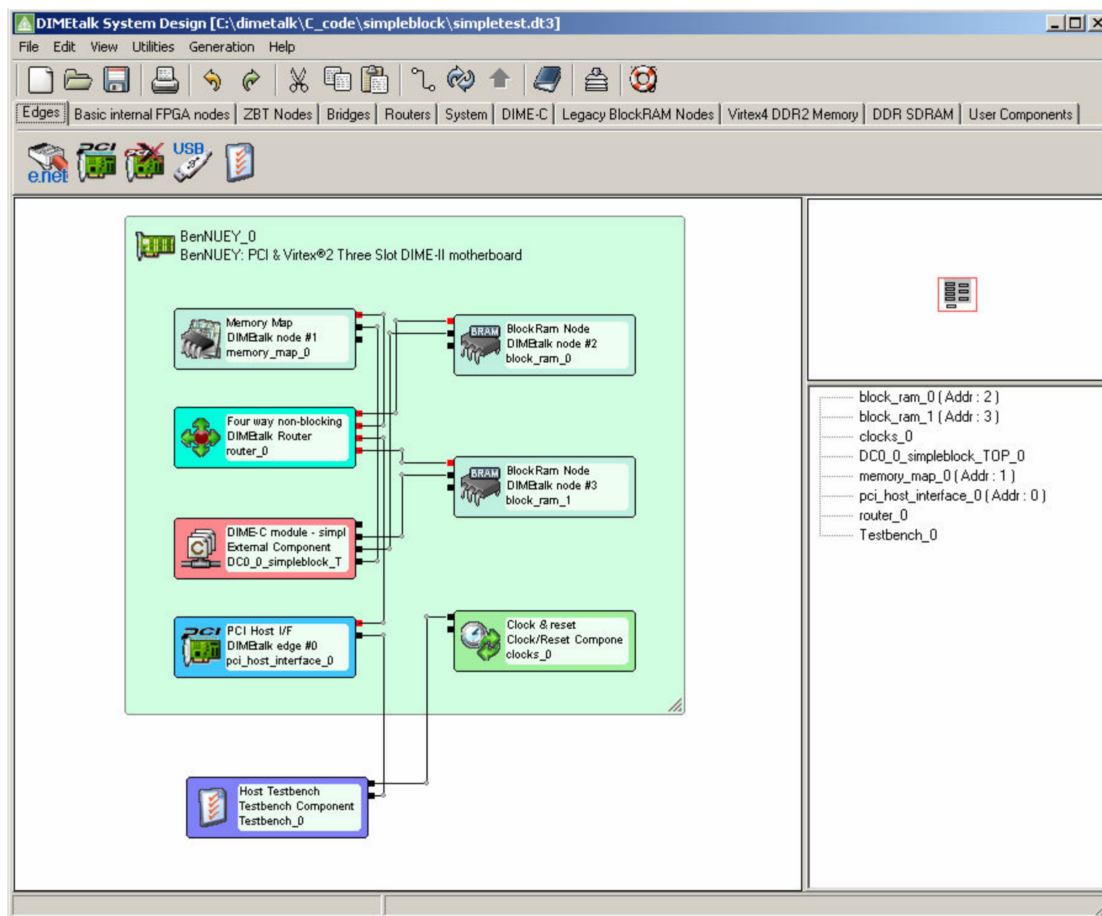

Figure 4. DIMETalk screen shot showing a network of components on an FPGA.

The application builder allows the user to link their code with other FPGA resources and so construct a complete FPGA design targeted at a specific piece of Nallatech hardware. As part of the build process DIMETalk generates an example project demonstrating basic API calls to test the FPGA design. The interface is straightforward to use but as yet we have not had access to Nallatech hardware in order to test the results. DIME-C is not directly compatible with the Cray XD1. The VHDL output from the DIME-C compiler must undergo some alterations before it will function on the Cray [33].

3.3 *Other languages and tools*

While Handel-C and Mitrion-C have become established high level tools in the HPRC arena thanks to their support for Cray and SGI reconfigurable hardware there are plenty of other options out there. Unlike the tools mentioned above the Trident compiler is a freely available open source tool for generating VHDL from C input. Much like the DIME-C compiler, Trident is only compatible with a subset of ansi-C. Currently the range of tools and languages available for programming FPGAs is somewhat mind boggling and this is likely to remain the case until standardization. One might again compare this with the number of programming paradigms and implementations in the early days of parallel HPC. Here is a list of some of the other available languages/tools.

- **SystemC - Open SystemC Initiative (OSCI) - http://www.systemc.org/**
- **Catapult C - Mentor Graphics - http://www.mentor.com/products/c-based_design/**
- **Impulse C - Impulse Accelerated Technologies - http://www.impulsec.com/**
- **Carte - SRC Computers - http://www.srccomp.com/CarteProgEnv.htm**
- **Streams C - Los Alamos National Laboratory - http://www.streams-c.lanl.gov/**
- **AccelChip - MATLAB DSP Synthesis - http://www.accelchip.com/**
- **Starbridge - VIVA - http://www.starbridgesystems.com/**
- **NAPA-C - National Semiconductor - http://portal.acm.org/citation.cfm?id=795813**
- **SA-C - Colorado State University - http://www.cs.colostate.edu/cameron/compiler.html**
- **CoreFire - Annapolis Micro Systems - http://www.annapmicro.com/**
- **Trident compiler - Los Alamos National Laboratory - http://trident.sourceforge.net/**
- **Reconfigurable Computing Toolbox - DSPlogic - http://www.dsplogic.com/home/products**
- **CHiMPS - Xilinx Research Labs - http://gladiator.ncsa.uiuc.edu/PDFs/rssi06/presentations/13_Dave_Bennett.pdf**

Details of a number of these FPGA programming tools can be found on the University of Florida's High Performance Computing and Simulation Research Centre web pages http://docs.hcs.ufl.edu/xd1/app_mappers

4 Initial experiences of FPGA programming at Daresbury

Work in progress within the Distributed Computing and Advanced Research Computing Groups at Daresbury is focused on producing FPGA implementations of kernels that are key to a wide range of scientific applications and to use our findings to inform the user community about the benefits and drawbacks of FPGA computing. Our initial goal was to implement an optimised 64-bit floating point matrix multiplication algorithm on Virtex II Pro and Virtex-4 FPGAs in our Cray XD1 system and we have taken a number of steps towards achieving this which are summarised below. This work, undertaken primarily in VHDL and Handel-C, uncovered a number of problems with both approaches. Our work has now refocused on using FPGA programming solutions which address the needs of scientific HPC. This includes tools such as the DIME-C compiler and the Trident compiler which can produce hardware from a subset of ANSI-C.

This work has certainly helped to confirm that programming FPGAs is a non-trivial task, as outlined above. Over the last year high level programming tools have matured to the point that many now provide support for double precision floating point arithmetic and Virtex-4/Virtex-5 FPGAs. Despite this it is still far from straightforward to get code running on the XD1 platform. Celoxica have promised a platform support library for Handel-C on the Cray XD1 but the library is still under development with several key features not yet implemented. This is somewhat typical of a marketplace in which the lack of standardization has made cross-compatibility between tools, chips and HPRC platforms very difficult to achieve. Until such standards materialise, programmers will still need to make a choice between relying on the assistance of hardware designers or splashing out on high level tools such as those provided by Mitrion in order to get code running on the XD1.

The Cray XD1 itself has a couple of significant drawbacks. The FPGA related tools, examples and documentation provided by Cray are focused solely on VHDL programming. Other systems such as the SRC Map-station are provided with vendor specific C-Like programming tools. Cray have also

recently reported that the XD1 does not form part of their future plans which does not bode well for future programming tool developments on the system. Despite this the system is powerful and with a considerable amount of effort from a combined team of software and hardware designers it should provide a substantial speed-up for some applications.

Below is a summary of the work that we have undertaken using both VHDL and Handel-C on development platforms and on the Cray XD1. Also summarised is our work with Nallatech DIME-C.

4.1 Development board VHDL

Original implementations of VHDL designs were tested on a Xilinx ML401 development board. This board contains a single Xilinx Virtex-4 lx25 FPGA and comes pre-loaded with a number of demonstration designs for driving peripherals using the FPGA. These include designs for testing the onboard LEDs and display, as well as driving an external VGA display and communicating with a PC via the RS232 serial interface. The board can be programmed from a remote computer using a JTAG interface. For the initial investigation phase of the work and as a learning tool for VHDL this board proved very useful. We produced a series of simple test designs which activated devices on the board. These started with the simplest of designs which flashed a single LED on the board and progressed to designs incorporating 64-bit floating point cores.

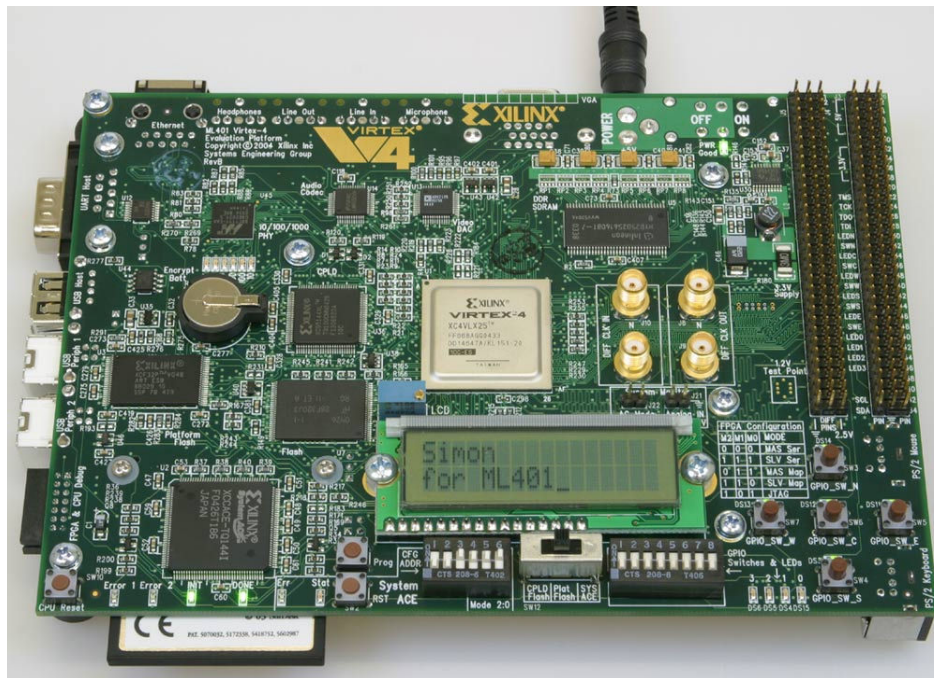

Figure 5 Xilinx ML401 Development Platform

4.2 Development board Handel-C

Early work with Handel-C was targeted towards the Celoxica RC10 development board. The RC10 is based around a single Xilinx Spartan 3 3S1500L-4 FPGA. The Spartan series of FPGAs provide a low cost, entry level alternative to the Virtex series [34]. The board is equipped with 16MB of onboard Flash memory, USB 2.0 port, CAN bus, RS232 port, PS/2 Keyboard and Mouse ports, a 5-way mini Joystick and a JTAG connector. A seven segment display and eight programmable LEDs are also provided [35]. This board proved much simpler to program than the ML401 due to the excellent support for Handel-C on this system. The main factor in the ease of programming the board was the platform support library from Celoxica. The PSL removed much of the work required to control communication with external resources using HDLs. In fact this board gave the authors a glimpse of what may be possible once the HPRC market begins to mature: given a reasonably familiar language and library support for the platform it is possible for the programmer to focus on the scientific

algorithm rather than on interfacing the hardware components together. However our experience so far suggests that this situation is currently the exception rather than the rule.

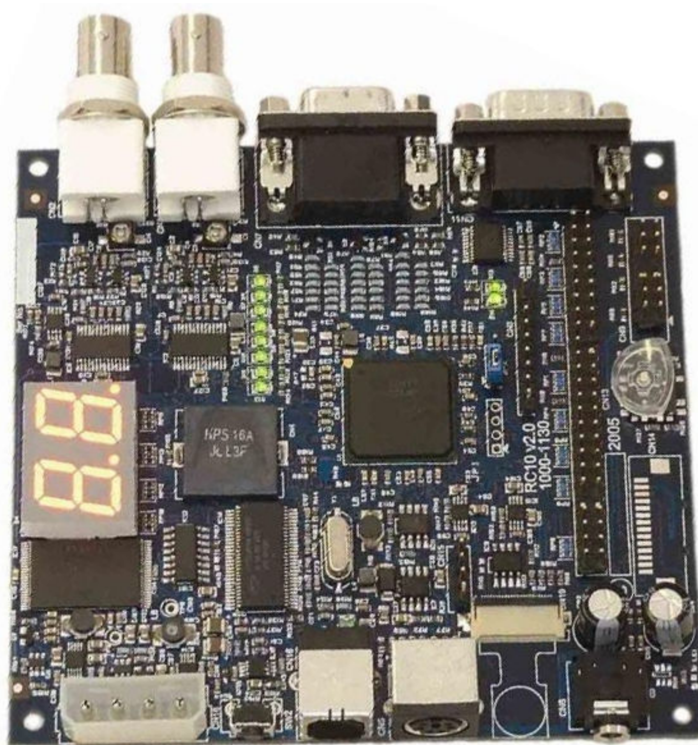

Figure 6 Celoxica RC10 Development Platform

4.3 Cray XD1 VHDL

Our primary FPGA resource is a 6 chassis Cray XD1 system. Each chassis comprises of six nodes, where a node is a dual processor AMD Opteron 250 with a clock speed of 2.4GHz and 4GB local memory, all connected by Cray's proprietary RapidArray fabric. Three of the XD1 chassis are equipped with FPGAs. In total there are twelve Xilinx Virtex 4 (LX160) FPGAs and six Xilinx Virtex 2 Pro FPGAs attached to the system

Currently attempts are being made to implement a 64-bit floating point matrix multiplication kernel on the Xilinx Virtex-4 FPGAs. This kernel forms the basis of the level 3 BLAS [36] and is therefore crucial to the acceleration of many scientific applications. For instance the widely used LAPACK library [37] depends upon a fast level 3 BLAS for its efficiency. This work follows on from work that has been carried out elsewhere to implement double precision floating point matrix multiplication on smaller FPGAs [17]. The Virtex-4 LX160 FPGAs in the Cray XD1 provide considerably greater resources for floating point calculations than the Virtex II Pro devices used in previous work and should therefore be able to provide in the order of a 2X speedup over such devices.

The basic design for our matrix multiplication architecture is shown in figure 7. The block highlighted in grey represents a single MAC pipeline which can be repeated as many times as the FPGA resources will allow. Each MAC pipeline contains block ram elements to buffer input, floating point adder and multiply cores and a sequencer to control data flow through the pipeline. Preliminary calculations based on the number of pipelines that could theoretically fit onto a Virtex 4 LX160 FPGA indicate that sustained performance of 15 to 20 GFlop/s could be achieved using this design.

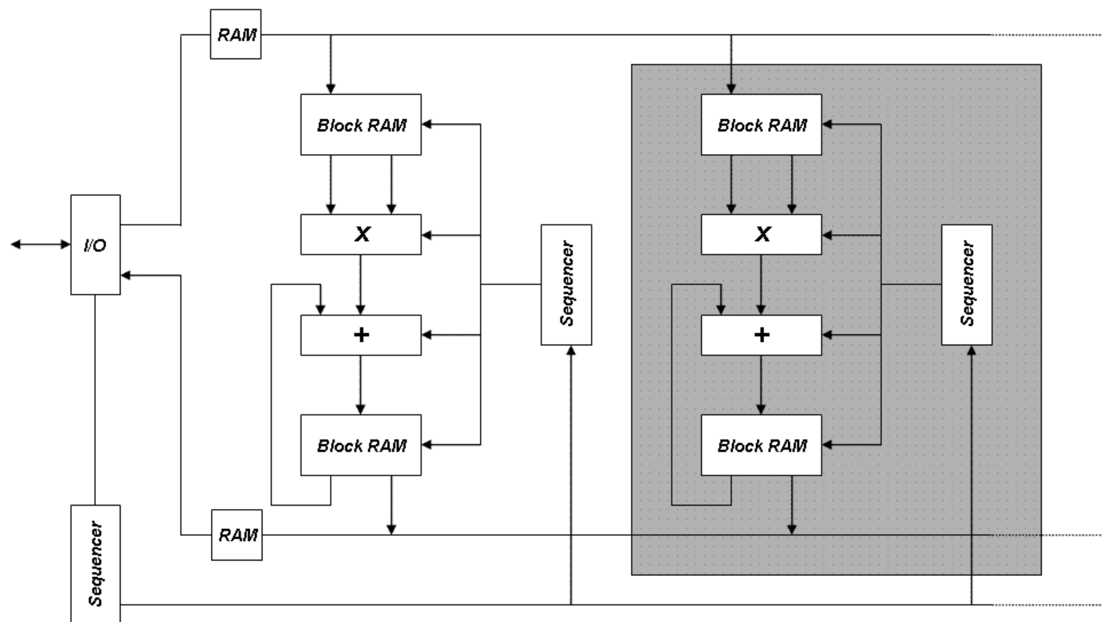

Figure 7 FPGA Matrix multiplication architecture

In the initial phase of this work we carried out a number of sanity checks to verify that all software and hardware functioned as expected. The first of these sanity checks involved rebuilding and running all of the reference codes provided by Cray. This enabled us to check the validity of the Cray VHDL/C source code and ISE project files. It also allowed us to verify that the Cray designs were compatible with the current version of the Xilinx tools (ISE Foundation 8.1i). Finally, this test allowed us to verify that all pre-built bitstreams provided with the reference designs functioned identically to the bitstreams that we generated from the source code.

Cray provides the following FPGA reference designs as part of the Cray XD1 software:

- hello – Demonstrates the basic operation of interfacing the FPGA to the Opteron Processor.
- mince – Diagnostic tool for testing the FPGA memory and bus interfaces.
- mta - Mersenne Twister random number generator.
- swa - FPGA implementation of Smith-Waterman algorithm.
- dma – DMA test application based on 'hello'

As well as these reference designs Cray provides IP cores to interface the FPGA with the RapidArray interconnect (RT core) and the QDR II SRAM (QDR2 core). Figure 8 illustrates how user applications such as 'hello', 'mince' and 'mta' are combined with the RT core and the QDR2 core to form a complete design.

All Virtex II Pro reference designs were built and run, with each producing output that could be verified using the pre-built bit streams provided by Cray. When this work was conducted, only the 'mince' design had been implemented by Cray on Virtex 4. Unfortunately, ISE project settings for this design were inconsistent with the pre-built bit stream provided by Cray. We had to alter the target part for the design from xc4vlx60 to xc4vlx160 and alter the 'macro search path' in the translate properties menu to point at the Virtex 4 versions of the RT and QDR cores. Having made these changes to the project we were able to successfully build and run the Virtex 4 version of the 'mince' design. N.B. These changes are no longer necessary as Cray have addressed this inconsistency in the final release of their FPGA support software. This support software also contains Virtex-4 versions of all the reference designs listed above.

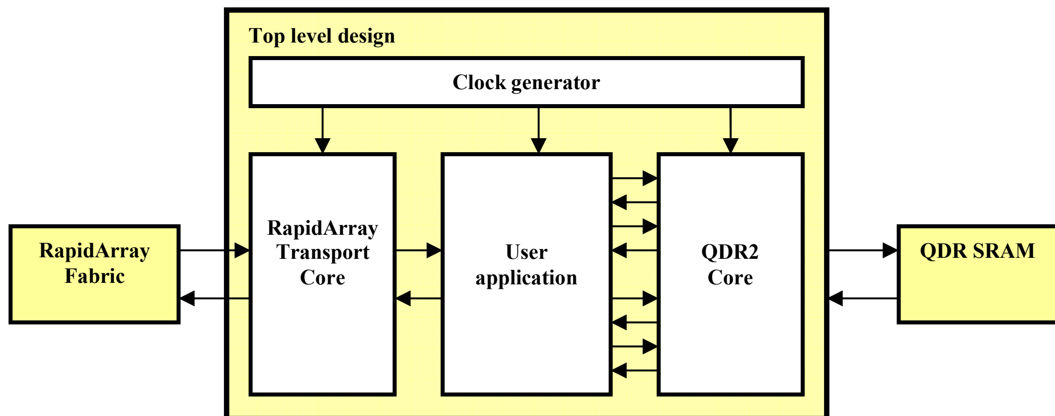

Figure 8 FPGA organization [38]

Next, we performed a test of portability between Virtex II Pro and Virtex 4 designs. The interface between the Cray cores and the user application is identical for both Virtex II Pro and Virtex 4 versions of the cores. We wanted to test that the 'hello' and 'mta' reference designs could be easily ported from Virtex II Pro to Virtex 4 by simply dropping the user application code from the Virtex II Pro design into the framework for the Virtex 4 design (RT core, QDR core and clock management). Once the Virtex II Pro user application code and the Virtex 4 framework had been combined it was necessary to regenerate any Xilinx Core Generator cores which formed part of the user application (see online coregen documentation [39]). These cores needed to be generated from scratch using the Xilinx Core Generator tool as the necessary configuration files for the cores were not provided by Cray. Once the netlist files for the regenerated cores had been added to the working directory we were ready to implement the Virtex 4 version of the design. We encountered some problems with the Xilinx tools refusing to recognise the user constraints file (.ucf) when implementing the ported designs but we overcame this by removing the offending file from the project and reinserting it again. We have now verified that the ported 'hello' design and the ported 'mta' design both function correctly on the Virtex 4 LX 160 FPGAs.

As a first step towards implementing a 64-bit floating point matrix multiplication algorithm we built an initial test design based on the 'hello' reference design which squares integer values that have been pre-loaded into a block ram. We have now verified that this code functions correctly both in simulation and using Chipscope.

Chipscope is a Xilinx tool which enables on chip verification and debugging of designs. In order to use Chipscope on the XD1 one must first add the required Chipscope cores into your design (see Xilinx documentation [40] for information on inserting Chipscope cores into a design). Once incorporated the design must be compiled and loaded onto the FPGA, ensuring that the JTAG port is connected (see Cray XD1 FPGA development guide page 68). A PC running Chipscope can then be connected to the correct JTAG port on the relevant FPGA equipped chassis. For this, a Cray JTAG I/O adapter is required. Once connected it is possible to interactively control signals on the FPGA.

Further development of a VHDL based double precision floating-point matrix multiplication implementation has been halted while DIME-C and Trident compilers are investigated.

4.4 Cray XD1 Handel-C

Handel-C is currently difficult to integrate with the Cray XD1 system. Although a Platform Support Library (PSL) exists for the system it is very much in its infancy, and we found it to cause more problems than it solved. At this stage all we have been able to run are a number of example designs in order to verify that the early version of the PSL is functioning correctly. Given that the PSL is not yet complete we initially decided to concentrate on the VHDL approach. It swiftly became apparent that we would not make significant progress with VHDL without the support of experienced hardware

engineers and this lead us to investigate C-like approaches other than that offered by Celoxica. DIME-C from Nallatech was chosen as a possible alternative to Handel-C.

4.5 DIME-C

While DIME-C is not directly compatible with the Cray XD1 system it does offer several advantages in terms of ease of use over Handel-C (See section 3.2.3) and can be used to target Nallatech's H100 series of FPGA blades (as well as all other Nallatech hardware). These blades can be slotted into an IBM Bladecenter offering a cheaper and more flexible approach than the XD1 (though not necessarily a better performance approach). Using DIME-C we were quickly able to move from C-code to code that could, theoretically, function on Nallatech's FPGA hardware. Once the code is limited to the subset of ANSI-C that makes up DIME-C the user can concentrate all of their effort on pipelining and algorithm optimization. Work with DIME-C began by implementing the integer squaring test code developed in VHDL on the Cray XD1. The coding took a matter of minutes and a bit stream was generated using DIMETalk within half an hour. Currently we do not have access to Nallatech hardware so none of our DIME-C based designs have been fully tested.

Using DIME-C we were able to quickly develop a partially pipelined double precision matrix multiplication implementation. Figure 9 shows a visualization of this implementation as generated by the DIME-C compiler. This visualization indicates pipelined loops in green and un-pipelined loops in red.

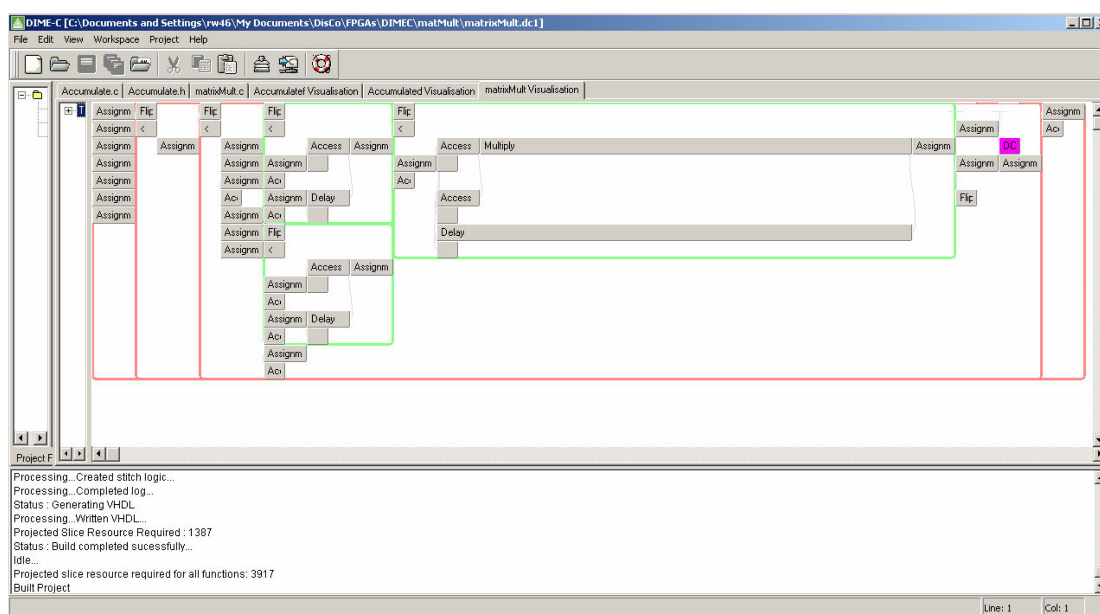

Figure 9. Visualization of DIME-C matrix multiplication. Pipelined loops are outlined in green.

It is likely that the performance of these designs is not optimal and it is not possible to assess the true value of the DIME-C approach without taking performance into account. However, purely in terms of assessing ease of use it is clear that DIME-C offers a far shorter time to solution than the other approaches detailed in this paper. This work is ongoing and in future papers we hope to report on the performance of DIME-C based implementations of a number of computational chemistry kernels.

4.6 Conclusions

There are a number of key points that summarise our initial experiences of FPGA programming.

1. If you are looking to achieve fast results then it is well worth considering purchasing software such as the Mitrion tools or hardware from vendors like SRC who provide their own high level design tools with their systems. This will save you a great deal of time and effort in getting designs running on your hardware.

2. For a software engineer to achieve ANY useful results it is absolutely necessary that the tool of choice abstracts away the hardware as far as is possible. There are several approaches that offer such abstraction from hardware (Mitrion-C, Dime-C, SRC Carte and Trident in particular) and it is down to the user to determine which is best depending on their specific requirements.

3. Even given points 1 and 2 progress is not going to be easy without ready access to an experienced hardware engineer. Even with a high-level approach you are still working with hardware.

4. Only attempt to use HDLs such as VHDL/Verilog if you are in a position to work closely with experienced hardware designers throughout the development cycle.

5. Do not expect to be able to generate easily portable code for FPGAs at this stage. There are currently no recognised standards to support FPGA programming and the current range of tools, languages and hardware are often incompatible.

Our initial experiences of FPGA programming have been largely frustrating. High performance reconfigurable computing is still very much in its infancy, with programming standards and portability between platforms still some way off. This results in a situation where a lot of time and effort can be spent writing software in a soon to be forgotten language for a soon to be forgotten machine. Unfortunately, as far as the machine is concerned we appear to have fallen into this particular trap. Cray have recently announced that they will no longer be providing software support for the FPGAs on the Cray XD1 and they have made it clear that the XD1 does not form part of their future plans. Instead, they are working with DRC to provide FPGA technology for the follow up to the XT3. The fact that a system with such potential as the XD1 can fall by the wayside (at least in terms of it's FPGA capabilities) so soon after it's emergence is an indicator of the unstable nature of HPRC. Given that SGI have had their own problems to deal with over recent months, it is relatively small companies like SRC, DRC computers and Nallatech who are increasingly having to drive the HPRC market forward. This is yet another similarity to the early use of parallel computing in the HPC arena.

Despite the current problems in HPRC there is still a great deal of untapped potential for FPGAs. It is our intention to continue our work in this area in an effort to quantify that potential. Once we have a clear picture of the performance gains that are possible in reality we will be able to make an informed judgement as to whether FPGAs are ready for HPC.

Acknowledgements

We would very much like to thank Rob Halsall of the CCLRC Instrumentation Department, Microelectronics Design Group for helping us understand some of the basics of hardware design, and showing us model implementations of some algorithms. We would also like to thank Roger Gook of Celoxica for organizing Handel-C training for two of the authors, and for helping us to use Handel-C on the Cray XD1. We would also like to acknowledge the useful feedback provided by Dave Strenski of Cray in relation to version 1.0 of this document.

References

[1] J. Dongarra, A. J. van der Steen**,** Overview of Recent Supercomputers, Oct 2004 http://www.netlib.org/utk/papers/advanced-computers/xd1.html

[2] SGI RASC RC100 Datasheet, Silicon Graphics Inc. 2006, http://www.sgi.com/pdfs/3920.pdf

[3] SRC Computers homepage, http://www.srccomp.com/default.htm

[4] DRC Computers, Product information, 2006, http://www.drccomputer.com/pages/products.html

[5] Vendor Spotlight: Cray Selects DRC FPGA Coprocessors for Supercomputers. Article from HPCWire, May 2006, http://www.hpcwire.com/hpc/644554.html

[6] Nallatech H100 Series datasheet, 2006, http://www.nallatech.com/mediaLibrary/images/english/5595.pdf

[7] Xilinx university program course notes, http://www.eece.unm.edu/vhdl/Labs2004/spring2004/lab01/lab1_intro.htm

[8] Xilinx Virtex 2 Pro overview, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_pro_fpgas/overview/index.htm

[9] Xilinx Virtex 4 overview, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm

[10] http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex5/capabilities/index.htm

[11] OpenFPGA homepage, 2006, http://www.openfpga.org/

[12] Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967

[13] B.C. Schafer, S.F. Quigley, A.H.C. Chan. Evaluation of an FPGA Implementation of the Discrete Element Method, In proceedings of the Field-Programmable Logic and Applications: 11th International Conference, FPL 2001, Belfast, Northern Ireland, UK, August 27-29, 2001.

[14] V. Kindratenko, D. Pointer, D. Raila, and C. Steffen, Comparing CPU and FPGA Application Performance, February 21, 2006.

[15] IEEE Computer Society, New York. IEEE Standard for Binary Floating-Point Arithmetic, 1985. IEEE Standard 754--1985.

[16] Xilinx Logicore Floating-point Operator v3.0 datasheet, Sept 28 2006.

[17] Y. Dou, S. Vassiliadis, G.K. Kuzmanov, G.N. Gaydadjiev. 64-bit Floating-Point FPGA Matrix Multiplication, From the proceedings of the international symposium on Field programmable gate arrays, 20-22 Feb 2005.

[18] K. D. Underwood and K. S. Hemmert. Closing the gap: CPU and FPGA Trends in sustainable floating-point BLAS performance. In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, 2004.

[19] K. S. Hemmert and K. D. Underwood. An Analysis of the Double-Precision Floating-Point FFT on FPGAs. In Proceedings of the IEEE Symposium on FPGAs forCustom Computing Machines, Napa Valley, CA 2005.

[20] M. de Lorimier and A. DeHon. Floating point sparse matrix-vector multiply for FPGAs. In Proceedings of the ACM International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 2005.

[21] G. Govindu, S. Choi, V. K. Prasanna, V. Daga, S. Gangadharpalli, and V. Sridhar. A high-performance and energy efficient architecture for floating-point based lu decomposition on FPGAs. In Proceedings of the 11th Reconfigurable Architectures Workshop (RAW), Santa Fe, NM, April 2004.

[22] L. Zhuo and V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. In 18th International Parallel and Distributed Processing Symposium (IPDPS'04), Santa Fe, NM, April 2004.

[23] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In Proceedings of the ACM International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 2005.

[24] S Williams, et al. The Potential of the Cell Processor for Scientific Computing, May 2006

[25] Clearspeed csx600 datasheet, 2006, http://www.clearspeed.com/downloads/CSX600Processor.pdf

[26] R. C. Ingham, Wikipedia article on VHDL, 12th July 2002, http://en.wikipedia.org/w/index.php?title=VHDL&oldid=115145

[27] Mitrion product brief, 2005, http://www.mitrion.com/press/Mitrion_product_brief_051017.pdf

[28] University of Florida's High Performance Computing and Simulation Research Centre web pages http://docs.hcs.ufl.edu/xd1/app_mappers

[29] S Mohl, "Unparalleled Computing Power" presentation given at RSSI 2005, http://www.ncsa.uiuc.edu/Conferences/RSSI/2005/docs/Mohl.pdf

[30] Overview of Mitrion tools, 2006, http://www.xilinx.com/products/design_tools/logic_design/advanced/esl/mitrion.htm

[31] Celoxica homepage, 2006, http://www.celoxica.com/

[32] http://www.it.iitb.ac.in/~sameerds/seminar/x111.html, 2006

[33] Joseph Fernando, Dennis Dalessandro, Ananth Devulapalli, Eric Stahlberg Kevin Wohlever, Mapping Linear System Solver Algorithms to FPGA on the XD1 Using High Level Toolsets, Presented at RSSI06, http://gladiator.ncsa.uiuc.edu/PDFs/rssi06/posters/03_Joseph_Fernando.pdf

[34] Xilinx Spartan Series Overview, 2006, http://www.xilinx.com/products/silicon_solutions/fpgas/spartan_series/index.htm

[35] ML401/ML402/ML403 Evaluation Platform Users Guide, May 2006, http://www.xilinx.com/bvdocs/userguides/ug080.pdf

[36] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. Math. Soft., 16 (1990), pp. 1--17.

[37] Linear Algebra PACKage online resources, 2006, http://www.netlib.org/lapack/

[38] Cray XD1 FPGA Development Manual, Cray, 2005

[39] Xilinx Core Generator online documentation,
http://www.xilinx.com/products/design_tools/logic_design/design_entry/ coregenerator.htm

[40] Xilinx Chipscope online documentation, http://www.xilinx.com/literature/literature-chipscope.htm