Department of Electrical and Computer Systems Engineering

Technical Report MECSE-26-2006

FPGAs vs Microprocessors

A.P.W. Bohm and G.K. Egan

MONASH UNIVERSITY

FPGAs vs Microprocessors

A.P.W. Bohm[1] and G.K. Egan[2]

Department of Computer Science
Colorado State University
Fort Collins, USA.[1]

Department of Electrical & Computer Systems Engineering
Monash University 3800
Melbourne, Australia.[2]

Abstract—**this paper presents some issues when implementing or re-implementing computational kernels on Field Programmable Gate Arrays (FPGAs). The class of kernels considered is those which use data streaming.**

Index Terms—microprocessors, FPGAs, streaming, dataflow.

I. Introduction

Microprocessors have taken a lot of the thinking about programming away from programmers. The increasing levels of abstraction we use, and the computational efficiencies these abstractions can bring, have until now been matched by an apparently unending increase in computational power offered by contemporary microprocessors. Unfortunately this growth is coming to an end. Increased complexity is offering only marginal gains in performance and this coupled with increased clock rates is leading to power consumption which is unacceptable for many important embedded applications.

A conventional microprocessor reads and executes (strictly interprets) a program using, at its core, the classic Von Neumann cycle (fetch instruction, decode instruction, fetch operands, execute, and store operands). The style of programming formulation tends to be memory centric dating from a time when memories were generally faster than CPUs. The order of magnitude difference between the CPU and memory clock speed is overcome somewhat by using multi-layered cache structures which relieves us of the effort required for explicit memory management. Unfortunately it also makes it difficult to bound execution times, a serious issue in real-time embedded applications. Managing cache coherency is becoming of serious concern in the newer hyper-threaded multi-cored microprocessors. The use of double precision floating point arithmetic which apparently relieves the programmer of the need to understand the range and precision needs of a particular computation, is in some cases misguided due to the power consumption and chip area required to implement floating point functions.

FPGAs (field programmable gate arrays) offer considerable flexibility and provide us with an opportunity to bypass some of the strictures of contemporary microprocessors. FPGAs may be used to execute a computation directly rather than to interpret some representation of the computation as is the case for Von Neumann architectures. This is accomplished by turning a program into a circuit and laying the circuit out on the configurable logic blocks of the FPGA. FPGAs have no caches, but they have on chip block RAM. These block RAMs are under the explicit control, so the time to perform the computation is deterministic. Because the program is laid out on the FPGA in the form of a dataflow graph, fine grain parallelism is exposed. In some cases we choose to program the FPGAs with a soft processor and augment the instruction set or add application specific logic to provide the overall required functionality. We can choose in this case to add explicit cache management partially overcoming the criticisms above.

II. The Von Neumann Legacy

For some time we have been obliged, or is it habit, to express significant classes of computation, originally expressed as directed graphs, in some form of textual language, usually an imperative language. What may have been simple scalar variables are collected together into arrays to better fit the machines whose basic interpretation mechanisms emphasise iteration and indexed data structures. This has in some cases led us to express computations using matrix notations where this is not at all necessary. The temporal validity of these former scalars embedded within matrices is often lost. Alternative data structures that preserve the timing semantics of the computation are streams (flexible size FIFOs) and delay queues (fixed sized buffers with simple timing characteristics).

As an example large distributed process control systems may be aggregated and described by a single large system state matrix. The resulting sparse-matrix problem, and all the complexities, principally memory addressing and data locality management, that accompany a memory based formulation may well have been avoided using the more appropriate dataflow programming model in conjunction with streams.

The conclusion one may reach is that the very way we think about and formulate problems has been and is determined by the nature of what von Neumann himself viewed as an interim computer architecture given the technology of the day. Von Neumann's own interests we know extended to large parallel architectures including self-replicating architectures.

III. Floating Point Arithmetic

Computational scientists may be prepared to program block RAMs explicitly provided there are very large performance gains, but they will not give up (e.g. IEEE standard) floating point arithmetic in general, although there are particular cases where fixed-point or special purpose floating point arithmetic can give rise to impressive speedups on FPGAs.

Some will remember that to perform computations on analog computers, which preceded digital computers, required very careful consideration of the maximum absolute value of intermediate computational variables. If the values were too large then amplifiers saturated (arithmetic overflow) rendering the computation invalid. If the variables were scaled down too aggressively to avoid overflow the resulting signal to noise ratios could deteriorate introducing errors (precision). What there is left of the literature in this domain could give some insight into appropriate arithmetic implementations for a given computation.

Microprocessors use floating point arithmetic as they are expected to perform general purpose computations and to cope with all possible applications. This is not the case when we are implementing direct execution kernels on FPGAs. Are we just lazy or have we forgotten how to manage range, precision and error propagation in our computations?

Currently floating point operations take a large amount of space on FPGAs, because they have to be implemented in the normal FPGA fabric, i.e., there is no special support for them, as there is for e.g. integer multiplication. Current FPGAs allow for about 50 (growing to 200+ in newer FPGAs) single precision floating point operations to be laid out on a chip. As FPGAs run at much lower clock rates than microprocessors, their floating point performance is not impressive. FPGAs currently often only accomplish speedups vs. microprocessors in the 3 to 5 range, (although there are exceptions [1]).

IV. Programming FPGAs

FPGAs have until recently been programmed using Hardware Description Languages such as Verilog and VHDL. This makes it hard for application experts trained in more mainstream languages, such as C and FORTRAN, to take advantage of FPGA technology.

Researchers have been working on algorithmic (imperative) programming language compilers for FPGAs. Example developments are Handel-C, Nimble, Defacto, SA-C, and Mitrion-C. SRC Computer Inc. (SRC) [2] is a very early adopter of this compiler technology. The SRC MAP compiler translates standard C and Fortran to FPGAs. SA-C and Mitrion-C are languages with single assignment semantics. These languages map naturally to dataflow graphs. Handel-C has CSP (communicating sequential processes) semantics. The rest are Von Neumann languages that require more elaborate analysis and transformation techniques (e.g. Static Single Assignment analysis) to be mapped into dataflow graphs.

Even though there is now programming language support for FPGAs, the programming practice is still hard particularly for those steeped in contemporary abstract programming styles. To exploit the fine grain direct execution model of FPGAs, programs need to be restructured and often rewritten from scratch. The reason for this is that in order to exploit the FPGA architecture, the programmer has to be aware of the amount of parallelism in the computation, the memory allocation (which relates back to the amount of parallelism unleashed), the staging of memory accesses through registers, delay queues, FIFOs, block RAMs and on board memories (OBMs) external to the FPGAs but directly coupled. FPGAs typically have very little on-chip RAM.

Most of the time there are orders of magnitude more computational parallelism than can be implemented directly on an FPGA, which means that the programmer needs to "fold" or "tile" the computation in such a way that the kernel will fit on the available chip area, the FPGAs bandwidth to the on board memories is maximized, and the block RAMs are accessed in parallel as much as possible. This is not readily expressed in the algorithmic programming languages.

V. A representative FPGA platform

Currently FPGAs run at frequencies up to ~400 MHz, usually lower than that (100 - 200 MHz). However, where a microprocessor has one port dedicated to its memory, an FPGA has many ports to many parallel on board memories (OBMs). For example, in the SRC's SRC7 two FPGAs are connected to 18 memories. This higher memory bandwidth in FPGA based machines provides for a more balanced machine architecture and can be exploited by the fine grain parallel program laid out on the FPGAs.

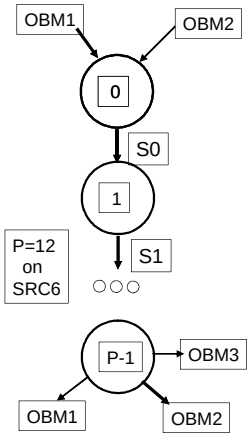
1) An example – LU decomposition

Figure 1 gives a straightforward LU decomposition kernel code fragment.

```
for k = 1 to n  {
  for i = k+1 to n
    Aik /= Akk
  for i = k+1 to n
    Aij -= Aik*Akj }
```

Figure 1 Simple LU decomposition code fragment from [3].

Restructuring the code for dataflow implementation [4] requires inverting the data dependencies to create a stream/process network: instead of taking Akk, Aik and Akj to Aij in iteration k, the whole matrix A is streamed through processes, each representing an iteration. Iteration k+1 can start after iteration k has finished row k+1. Neither row k nor column k are read or written anymore. Results from iteration/process k flow to process k+1. Process k uses row k to update A[k+1:n, k+1:n]. Actually, only a fixed number (P) of processes run in parallel. After that, data is re-circulated.

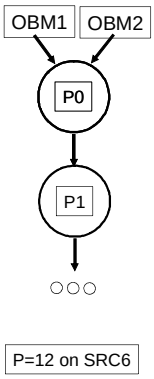


OBMS:
OBM1 -> OBM2 on even sweeps
OBM2 -> OBM1 on odd sweeps
OBM3 : siphoning off finished results

Processes are grouped in a parallel sections construct each process is a section.

Process 0
reads either from OBM1 or OBM2
writes to stream S0
Process i
reads from Si-1 writes to Si
Process P-1
reads from Sp-1
writes finished data to OBM3
writes rest to OBM1 or OBM0

Figure 2 Process pipeline




```
for(i = (s-1)*P; i < n; i++) {
 for(j = (s-1)*P; j < n; j++) {
```

```
//ping-pong
if(k&0x1) w= OBM1 [ i * n + j ];
      else w= OBM2 [ i * n + j ];
// if (i < me)  leave data unchanged
if (i ==me) {  // store my row
  if (j == i ) piv = w;
  myRow[ j ] = w;
}
else
if (i > me) {  // update this row with my row
  //  if (j < me) leave data unchanged
  if (j == me) { w /= piv; mul = w;}
  else if (j > me) w -= mul*myRow[ j ];
}
put_stream( &S0, w);
}}
```

Figure 3 Inside Process 0

All other processes are similar. The rest of the processes also read from stream (where P0 reads 'ping-pong' style from the OBMs). Last process writes ping-pong style to OBM.

The behavior of the restructured LU decomposition algorithm is not at all obvious at first glance. It requires significant understanding of the FPGA and in particular its OBM.

2) Performance

The relative performance for the LU decomposition is shown in Figure 4. Note the unpredictable behavior of the Pentium due to its memory hierarchy (cache) performance and that the FPGA implementation is about 5 times faster for n=512.

Again the limiting factor is the number of floating point units that were available at the time. The new generation of FPGAs have a significant number of embedded floating point units. The commitment of dedicated FPGA chip area to floating point units recognizes the previously noted extreme reluctance of applications programmers to explore other arithmetic paradigms [5] despite the potential benefits in speed and power consumption which are critical in some applications.

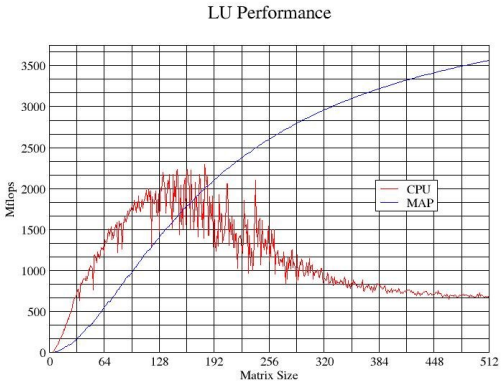


Figure 4 Pro Red (jagged peaks): 2.8Ghz Pentium. Blue: SRC MAP6 with 2 FPGAs at 100 MHz.

VI. FPGA PROGRAMMING STRATEGIES

Because of their nature, FPGAs are well suited to applications where data is streamed into a pipelined computational kernel with the data being transformed and continuously streamed out of the FPGA. Most classical signal processing applications have these characteristics.

Some of the more effective programming strategies include:

- Delay queuing to avoid re-reading data from OBMs effectively projecting data forward in time to the specific point of use. This is often done using windows in image processing and other scientific codes amenable to a stencil approach - used in Erode/Dilate pipelines in Focus of Attention codes.

- Replication of inner loops to exploit fine grain parallelism and memory bandwidth is a common compiler technique to reduce control overheads – used in a Gauss-Seidel iterative solver.

- Avoiding read/write conflicts in inner loop bodies, as these slow down the inner loop clock rate - used for inner products (matrix-vector multiply, matrix-matrix multiply) using hardware macros instantiated in the C program.

- Turning loop oriented codes into task and stream oriented codes to exploit data locality. Used in LU decomposition by inverting the data dependencies.

VII. CONCLUSIONS

There are no silver bullets which will allow us to go directly from the kernel of an existing program to a high performance FPGA implementation.

To achieve the promise of direct execution architectures will require, at least for the foreseeable future, the ability to juggle significant spatial and temporal relationships in the formulation applications. These are skills which to a significant degree have been either hidden or not required with systems built on the Von Neumann interpretive execution architecture. While current microprocessor designers clearly have these skills they are not now routinely developed within in university courses where abstraction rules. While some of these skills may be innate and not teachable there are working principles which we can codify that do not necessarily require deep insight.

It seems clear that most programming of FPGA based solutions will depend perversely on dominant existing imperative languages. These languages are not particularly well suited to describing stream based algorithms of the class we may be interested in, but there is a wealth of compiler knowledge available to translate these programs into the dataflow graphs which lie directly on the path to FPGA realization. This is certainly the case where we adhere to a single assignment discipline and avoid large flat memory structures and pointers.

Augmentation of these languages will be required to allow us to better manage the lifetime and location of data. Some of the new compilers are already addressing this in a primitive way. Importantly we must have the ability to project data forward in time (to some future clock cycle) without the need to explicitly create and manage queues or tapped queues. In many cases, stencils, which are familiar to us, can be mapped to quite specific hardware structures although with current compilers this must be explicit.

To close there may be some merit in the use of engines which directly interpret the dataflow graphs (dataflow machines [6]), when our computational kernels exceed current FPGA capacities, or when we need to support more than one application concurrently; back to the future perhaps?

ACKNOWLEDGMENT

We wish to thank SRC Computers Inc. [2] for access to their facilities and NCSA for holding the Reconfigurable Systems Summer Institute 2006 [7].

REFERENCES

[1] Kindratenko, V., Accelerating Scientific Applications with Reconfigurable Computing, NCSA Reconfigurable Systems Summer Institute 2006.

[2] SRC Computers Inc., www.srccomp.com/default.htm

[3] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., Introduction to Algorithms, MIT Press, pp 750, 2001.

[4] Bohm, A.P.W., The Power of Streams on the SRC MAP®, NCSA Reconfigurable Systems Summer Institute 2006.

[5] Constantinides, G.A., Cheung, P.Y., and Luk, W., Synthesis of Saturation Arithmetic Architectures, ACM Transactions on Design Automation of Electronic Systems (TODAES) Volume 8 , Issue 3, pp 334-354, 2003.

[6] Egan, G.K., Webb N.J. and Bohm W., 'Some Features of the CSIRAC II Dataflow Machine Architecture', in Advanced Topics in Data-Flow Computing, Prentice-Hall, pp143-173, 1991.

[7] NCSA RSSI06, www.ncsa.uiuc.edu/Conferences/RSSI