mHume for parallel FPGA

Abdallah Al Zain[1], Greg Michaelson[2], and Wim Vanderbauwhede[3]

[1] Heriot-Watt University, Edinburgh, Scotland, EH14 4AS, UK
A.D.AlZain@hw.ac.uk
[2] Heriot-Watt University, Edinburgh, Scotland, EH14 4AS, UK
G.Michaelson@hw.ac.uk
[3] University of Glasgow, Glasgow, Scotland, G12 8QQ, UK
wim@dcs.gla.ac.uk

Abstract. The formally motivated Hume language, based on the coordination of concurrent automata performing functional computations, was designed to support the development of systems requiring strong assurance that resource bounds are met. mHume is an experimental subset oriented to exploration of efficient heterogeneous multi-processors implementations. In this paper, the deployment of mHume on the FPGA MicroBlaze architecture is discussed. Preliminary results suggest very fast performance and good scalability compared with stock multi-core processors.

Keywords: Hume, FPGA.

1 Introduction

While the number of cores on stock CPUs continues to grow steadily, in accordance with Moore's Law, their internal configurations are strongly constrained by the underlying CPU architecture. In contrast, contemporary FPGAs themselves offer immediate opportunities for very large numbers of processing elements, with highly flexible connectivity and excellent scalability, albeit with poorer performance than conventional CPUs. Thus, there is considerable interest in synergistic exploitation of concurrency in *heterogeneous architectures*, typically comprising multi-core processors with SIMD acceleration augmented with an FPGA.

In the EPSRC supported Islay project, we are exploring alternative routes to deploying the functionally-flavoured, concurrent language Hume [7] on such architectures. Hume is based on autonomous *boxes*, linked point to point, and to the environment, by single buffered *wires*. Thus, we are interested in directly realising Hume boxes as loci of concurrency.

Hume's design is oriented to systems where there is need for strong assurance that bounds on resources such as time and space are met. This is enabled through Hume's rigorous formal foundations, reflected in a unified tool chain built around well-characterised Hume Abstract Machine(HAM), enabling tight articulaltion of program analysis and implementation.

We have explored the implementation of Hume on both multi-core and FPGA architectures via the HAM interpreter (hami) [10, 1], which offers consistent speedup on regular programs. However, FPGA performance is markedly poorer than on CPUs. We have also explored the direct implementation of Hume on FPGAs, via C generated from HAM through the standard tool chain, but this offers poor flexibility for multi-processor exploitation.

Thus, in our current research, we are investigating the direct generation of C from Hume itself, through a lightweight compiler for the *mHume* subset, which generates code that facilitates multi-processor implementations. In this paper, we present the first results of our experiments, which show both consistent speedup and very decent performance of multi-box programs on FPGA multi-processors.

2 mHume overview

mHume is a proper subset of Hume, itself based on an automata-like *coordination* layer, for describing boxes and wires, and a functional *expression* layer, for describing pattern matching and processing within boxes, both sharing a rich polymorphic type system and *definition* layer, for describing types, structures and functions. mHume retains an expression layer restricted to integer arithmetic but the full core coordination layer, offering considerable potential for richer expressions in future.

To get some flavour of mHume, consider the following program which generates squares of successive integers, illustrated in Figure 1.

Box `inc` generates the integers, and box `square` generates the squares by repeated adding and counting, Figure 2.

`inc` wires output `n'` (next integer) to input `n`, and output `r` to `square`'s input `i`. On each execution cycle, `inc` matches the next integer, outputs it to `square` via `r`, and also increments it and sends it back to itself via `n'`.

`square` wires outputs `s'` (sum), `c'` (count) and `v'` (current value) back to the corresponding inputs `s`, `c` and `v`. It also wires input `i` to `inc`'s output `r`, and its output `o` to the stream `output` associated with standard output `std_out`.

Note that for Hume pattern matching, there must be appropriate values on all inputs for a match to succeed, except for the pattern `*` which ignores the corresponding input. Thus, `square` is composed of three matches:

1. `(*,s,0,v) -> (s,*,*,*)`: ignoring input from `inc`, if the count is `0` then output the sum;
2. `(*,s,c,v) -> (*,s+v,c-1,v)`: ignoring input from `inc`, add the current value to the sum, decrement the count and retain the current value;
3. `(i,*,*,*) -> (*,0,i,i)`: for the next input from `inc`, set the sum to `0`, and the count and current value to that input.

The mHume syntax is summarised in Figure 3.

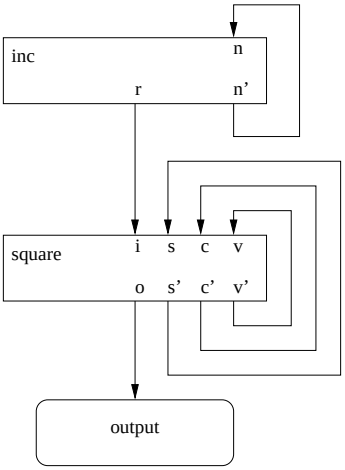

Fig. 1. Square program.

```
type integer = int 64;

box inc
in (n::integer)
out (r::integer,n'::integer)
match (n) -> (n,n+1);

box square
in (i::integer,s::integer,
      c::integer,v::integer)
out (o'::integer,s'::integer,
      c'::integer,v'::integer)
match
 (*,s,0,v) -> (s,*,*,*) |
 (*,s,c,v) -> (*,s+v,c-1,v) |
 (i,*,*,*) -> (*,0,i,i);

stream output to "std_out";

wire inc (inc.n' initially 0)
          (square.i,inc.n);

wire square
(inc.r,square.s',square.c',square.v')
(output,square.s,square.c,square.v);
```

Fig. 2. Square program Code

3 mHume execution model and compiler

mHume follows closely the Hume two stage execution model. A program executes repeatedly in cycles during which boxes are alternately in $READY$ (awaiting input) or $BLOCKED\_OUTPUT$ (output pending)states. Initially, all boxes are $READY$.

On each cycle, at the *match stage*, only a $READY$ box may attempt to match inputs. If it succeeds, it generates pending outputs and becomes $BLOCKED\_OUTPUT$. At this stage, no inputs are consumed or outputs asserted. Then, at the *super step* stage, only a $BLOCKED\_OUTPUT$ box may attempt to assert pending outputs. Provided all the previous outputs have been consumed, it consumes its inputs, asserts its outputs and becomes $READY$. Note that this input/output behaviour is very similar to AlgolW's "call by value result".

mHume is compiled to C by a very simple multi-pass, syntax directed processor, written in Haskell, and closely aligned to the execution model. We consciously generate a very restricted subset of C (declaration, assignment, condition, jump), to simplify potential cost analysis in future.

Compilation proceeds as follows:

program → [*component*`;`]$^+$
component → *box* | *wire* | *stream* | *typedef*
box → `box` *id* `in (`*links*`) out (`*links*`) match` *matches*
links → *link* [`,` *links*]*
link → *var*`::`*type*
matches → *match* [`|` *matches*]*
match → *pattern* `->` *exps*
pattern → *patt* [`,` *pattern*]*
patt → *int* | *var* | $*$
exps → *exp* [`,` *exps*]*
exp → *int* | *var* | `(` *exp* `)` | *exp op exp* | $*$
op → `+` | `-` | `*` | `/`
wire → `wire` *id* `(`*inwires*`) (`*outwires*`)`
inwires → *inwire*[`,` *inwire*]*
inwire → *id*[.*var*[`initially` *int*]]
outwires → *outwire*[`,` *outwire*]*
outwire → *id*[.*var*]
stream → `stream` *id* `{ from | to } "` *path* `"`
typedef → `type` *var* `=` *type*
type → *var* | `int` *int*

Fig. 3. mHume syntax.

1. for each box *id*, generate `int` variables:
   (a) *id*`state` for the execution state (0=$READY$; 1=$BLOCKED\_OUTPUT$);
   (b) *id*`PATT` for the number of the most recently successful *match*;
   (c) *id*`I/O`*i* for the *i*th input/output link value;
   (d) *id*`I`*i*`STATE` for the *i*th input link status flag (0=$EMPTY$; 1=$FULL$);
2. set all box states to $READY$;
3. generate wire initialisation;
4. generate match stage: for each $READY$ box; for each *match*:
   (a) for each *patt*:
      i. for non-ignore (`*`) *patt*, check if corresponding input link $FULL$;
      ii. for constant *patt*, check input has required value;
   (b) if all *patt*s satisfied:
      i. set corresponding input status flags to $EMPTY$;
      ii. remember that this *pattern* succeeded;
      iii. set appropriate output links to *exp* values, taking *var* values from corresponding input links;
      iv. set box status to $BLOCKED\_OUTPUT$.
5. generate super step stage: for each $BLOCKED\_OUT$ box, for each *match*:
   (a) if *match* succeeded, if all inputs links wired to outputs links $EMPTY$:
      i. copy non-ignore (`*`) output links to input links and set input link status flags to $FULL$;
      ii. set box status to $READY$

Note that the N *pattern*s in a *match* are numbered from 0 to $N-1$.

For example for the second match of box `square` above:

```
(*,s,c,v) -> (*,s+v,c-1,v) |
```

the C generated for intialisation, and the match and super step stages, is:

```
 1. int squareSTATE; int squarePATT;
 2. int squareI0; int squareI0STATE; ...
 3. int squareI3; int squareI3STATE;
 4. int squareO0; ... int squareO3;
    ...
 5. squareSTATE = 0;
    ...
 6. squareI0STATE = 0; ... squareI3STATE = 0;
    ...
 7. square1:
 8.    if(squareI1STATE == 0) goto square2;
 9.    if(squareI2STATE == 0) goto square2;
10.    if(squareI3STATE == 0) goto square2;
11.    squareI1STATE = 0; ... squareI3STATE = 0;
12.    squarePATT = 1;
13.    squareO1 = squareI1+squareI3;
14.    squareO2 = squareI2-1;
15.    squareO3 = squareI3;
16.    goto squareSUCC;
    ...
17. squareSUCC:
18.    squareSTATE = 1;
    ...
19.    if(squareSTATE == 0) goto squareENDSUPER;
    ...
20. squareSPATT1:
21.    if(squarePATT != 1) goto squareSPATT2;
22.    if(squareI1STATE != 0) goto squareSPATT2;
23.    if(squareI2STATE != 0) goto squareSPATT2;
24.    if(squareI3STATE != 0) goto squareSPATT2;
25.    squareI1 = squareO1; squareI1STATE = 1;
26.    squareI2 = squareO2; squareI2STATE = 1;
27.    squareI3 = squareO3; squareI3STATE = 1;
28.    squareSTATE = 0;
29.    goto squareENDSUPER;
    ...
```

Line 1 defines execution and pattern state variables. Lines 2 to 3 define variables and status flags for the input links. Line 4 defines variables for the output links. Line 5 sets the initial box execution state to $READY$. Line 6 sets the input status flags to $EMPTY$.

In the execution stage, at line 7, the first match has failed. For the second match, lines 8 to 10 check that required inputs are $FULL$. Line 11 sets the input status flags to $EMPTY$ and line 12 sets the pattern state to 1 to indicate that the second match succeeded. Lines 13 to 15 set the outputs to the corresponding expressions.

On completion of the box execution, line 18 sets the box state to $BLOCK\_OUTPUT$.

In the superstep stage, line 19 checks that the box state is $BLOCKED\_OUT$. Then, at line 20, the first match had not succeeded. Line 21 checks that the second match succeeded. Lines 22 to 24 check that inputs corresponding to outputs are $EMPTY$. Lines 25 to 27 copy outputs to inputs and set status flags to $FULL$. Line 28 sets the box execution status to $READY$.

While there is some avoidance of redundant jumps, code generation is, over all, naive. Opportunities for improvement include simplification of *self wired* matches, where a box's outputs are only to its own inputs.

4 MicroBlaze FPGA architecture

FPGAs are versatile configurable electronic devices that can be utilised as accelerators to implement tailored computational logic specific to the application being executed. Moreover, these components can be reconfigured at any time for new applications, making it possible to perform a wide range of tasks. Thanks to the continuing advances in CMOS technology, as with cores following Moore's law, FPGAs have now arrived at a stage where they form a viable alternative to Application-Specific Integrated Circuits (ASICs) for many applications.

Craven and Athanas [4] have identified major performance disadvantages of FPGAs compared to microprocessors:

- The maximum clock frequency for FPGAs is typically a few hundred MHz, while non-embedded microprocessors typically run at a few GHz.
- The FPGA's configurability comes at the cost of a large overhead (compared to equivalent-functionality ASICs).
- Floating-point arithmetic on FPGAs is very resource-intensive compared to integer arithmetic (comparable to CPUs without a FPU)

Despite these disadvantages FPGAs are still able to outperform microprocessors because:

- FPGAs are used to design specialised circuits for specific tasks.
- All the logic on the FPGA can be utilised to perform the specific task.
- FPGAs deliver a vast amount of fine-grained parallelism.
- FPGAs offer huge memory bandwidths through configurable logic, on-chip block RAMs, and local memories.

In particular, contemporary soft-core FPGA architectures enable high degrees of flexible and scalable parallelism.

The MicroBlaze soft processor core The MicroBlaze is a 32-bit RISC (reduced instruction set computer) synthesizable soft processor core developed and maintained by Xilinx Inc [16]. It is specifically designed for Xilinx FPGAs and therefore makes efficient use of their resources. The resource utilization of the

MicroBlaze is significantly smaller. Furthermore, the MicroBlaze has a solid documentation and can easily be extended by user defined IP(Intellectual Property)-blocks. Xilinx provides a complete development environment (EDK) to configure the processor and all attached IP-blocks. The EDK includes a complete GNU-tool chain for the software part.

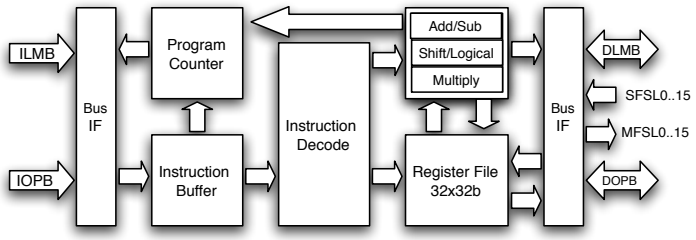

Fig. 4. MicroBlaze core structure

Figure 4 depicts the MicroBlaze building blocks which include:

- general purpose registers,
- instruction word with three operands and two addressing modes,
- instructions and data buses that comply with IBM's OPB (On-chip Peripheral Bus) and PLB (Processor Linker Bus) specification, and provide direct connection to on-chip block RAM through LMB (Local Memory Bus),
- instructions to support FSL (Fast Simplex Link),
- and, hardware multiplier.

The MicroBlaze core implements a Harvard architecture. This means that it has separate bus interface units for data and instruction access. Each bus interface unit is split further into a Local Memory Bus (LMB) and IBM's PLB and OPB buses. The LMB provides single-cycle access to on-chip dual-port block RAM. The PLB and OPB interfaces provide connection to both on and off chip peripherals and memory.

4.1 FSL: Fast Simplex Link

The FSL is a fast interface supported in the MicroBlaze architecture by dedicated low cycle count read and write operations. FSL is formed by an (asynchronous) 32-bit FIFO of configurable depth. This architecture allows high throughput with low latency and simple data handling. The FSL is always a dedicated connection to or from a single component. In the board we use for the experiment in this paper, the MicroBlaze core provides 16 input and 16 output interfaces to FSL channels. Finally, the presence of these FSL channels motivates our choice of the MicroBlaze as a soft core, as it allows us to create a true network of processors. The resulting multi-core architecture has a very high total bandwidth and is entirely free of the traditional bus bottleneck.

5 Parallel FPGA design

5.1 Hardware Apparatus

In our experiment we used the *Xilinx Univesity Program (XUP) Virtex-II Pro* Board. It provides an advanced hardware platform that consists of a high performance Virtex-II Pro FPGA surrounded by a comprehensive collection of peripheral components that can be used to create a complex system and to demonstrate the capability of the Virtex-II Pro Platform FPGA. The Virtex-II Pro contains two embedded PowerPC 405 cores and a 10/100 Ethernet PHY device. The board provides up to 2GB of double data rate SDRAM, an RS-232 DB9 serial port, an Ethernet port, up to 256MB of CompactFlash storage, four LEDs and four switches, a 100MHz system clock and a 75MHz SATA clock. It also includes support for JTAG-over-USB FPGA configuration bit-streams.

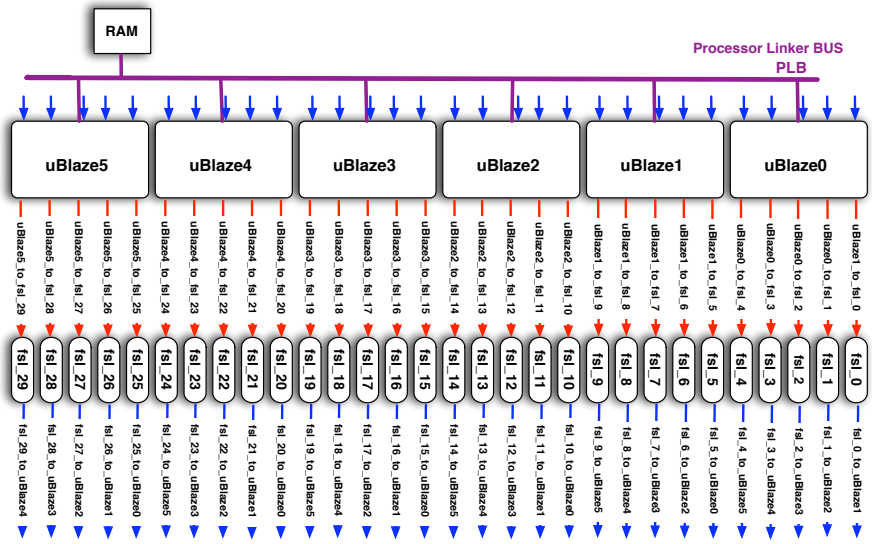

Fig. 5. Parallel Architecture Design

5.2 Parallel Design

In our parallel FPGA design we constructed six MicroBalzes which are fully connected in parallel as show in Figure 5. The communication handler in the MicroBlazes is connected to the FSL ports (subsection 4.1). To increase distribution between the MicroBlazes we design a full mesh topology. To fulfil our network topology design each MicroBlaze uses five FSL buses to communicate

with the other MicroBlazes. We, also, configure the depth of each FSL bus FIFO to one, to realise the Hume super-schedule strategy and at the same time to allow each MicroBlaze to clock at different rates. Using FSL bus for communicate between MicroBlazes provides a master-slave parallel model, i.e, when a MicroBlaze writes to the FSL bus, it will act as a master on the FSL bus and the receiver MicroBlaze from the FSL bus will be consider as a slave to the FSL bus. For instance as shown in Figure 5, MicroBlaze 0 (`uBlaze0`) uses port `uBlaze0_to_fsl_0` to write to FSL 0 (`fsl_0`) and FSL 0 uses port `fsl_0_to_uBlaze1` to write to MicroBlaze 1 (`uBlaze1`). In this scenario MicroBlaze 0 is a master and MicroBlaze 1 is a slave. In our design due to the single depth of the FSL bus, at the moment when MicroBlaze 0 (master) writes on FSL 0, FSL 0 makes the data available to MicroBlaze 1 (slave) and it prevents MicroBlaze 0 from writing again to FSL 0 until MicroBlaze 1 reads the data from FSL 0.

6 mHume on FPGA

6.1 Crafting mHume to Run on the FPGA

Sequential FPGA Design: The generated `C` code from the mHume compiler assumes the presence of a POSIX-style operating system that takes care of I/O, memory allocation and thread/process-level parallelism. However, an operating system is an unjustifiable overhead in the case of deploying mHume generated `C` code on the FPGA: the FPGA will typically be used as an accelerator on a host system, and the host will take care of I/O; as we will see further, we have no need for file system access, processes or threads either. As a consequence, it was essential to rearrange the generated `C` code to be free of OS-specific functionality. For instance, relying on the OS timer to coordinate wires had to be adjusted to use library functions which access the processor cycle count registers instead. Moreover, all memory allocations for wires and boxes had to be adapted to a static allocation instead of using *malloc*-style dynamic assignment. More importantly, embedded hardware systems like FPGA lack the file system concepts, which means mHume implementations had to be enhanced to read input files through dedicated ports on the FPGA board.

Parallel FPGA Design: The sequentially running `C` code has to be split between the designed MicroBlazes on the FPGA. Each mHume box will be assigned to a separate MicroBlaze. All communications between boxes occur through the FSL channels which achieve *completely lock-free communication* between boxes. There is no need to synchronise two communicating boxes: since any output written during one mHume scheduling cycle will never be read before the subsequent cycle, and since the subsequent cycle will not be scheduled before all output is completed, it follows that a box/MicroBlaze can never start to read data before it has completely finished its output.

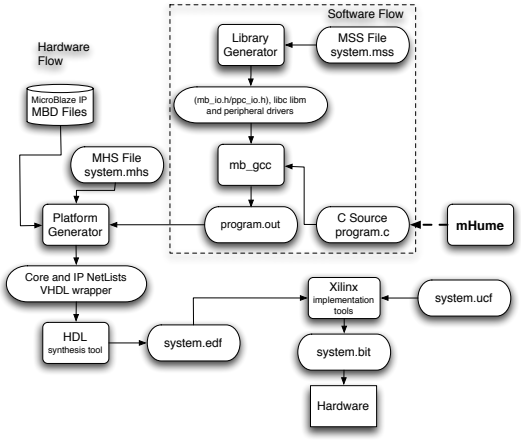

Fig. 6. From mHume to FPGA

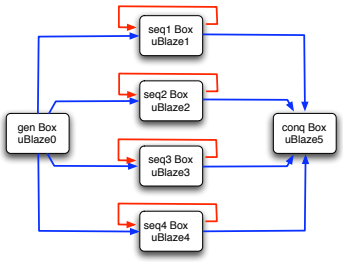

Fig. 7. multi-sq Example in Parallel FPGA

6.2 From mHume to FPGA

Figure 6 illustrates the steps of moving the generated `C` code of mHume to the FPGA. The figure is revamped from the *Xilinx*'s on-line support documentations [15]. The figure includes software and hardware flow. These flows describe the *Xilinx* development tools for MicroBlaze system building process. They include Microprocessor Hardware Specification (MHS) and Microprocessor Software Specification (MSS) files to define hardware and software systems. In our sequential approach from mHume to FPGA, those files have been generated automatically using the *Xilinx* EDK wizard system, with minor changes to simplify the memory connection, Local Access Memory (LAM), with the Multi-Port Memory Controller (MPMC), to fit with mHume's wire and box design. These hardware and software system files provide the core to build the MicroBlaze system automatically using *Xilinx* EDK tools.

For the purpose of our parallel design we had to implement manually the parallel MicroBlazes in the MHS file. Adding an extra MicroBlaze to the design is a bit more than making a replica of one generated by the *Xilinx* EDK wizard. Each MicroBlaze use the Local Memory Bus (LMB) to access the on-chip block RAM, and it uses data and instruction LMB. The on-chip block RAM has to be divided between the participant MicroBlaze processors with the consideration that each MicroBlaze can not allocate more than 8KB for itself. For the MicroBlazes to be able to communicate, FSL buses have to be implemented in the design as coprocessors. Also each MicroBlaze must be aware of the master FSLs, which transfers data to the MicroBlaze, and the slave FSLs, which receives data from the specified MicroBlaze.

Subsequently, the EDK tools integrate the MicroBlaze cores and the appropriate peripherals, and create custom-built C libraries and drivers. After this, the

EDK uses the Platform Generator and Library Generator tools, in the hardware and software flows respectively, to setup the particular hardware and software for the corresponding design. In our implementation, platform and library generator tools have to be configured to select our defined STDIN/STDOUT peripherals, map the added peripherals to the appropriate drivers, specify the correct heap and stack size, map the stack and heap to correspondent memory, and set the correct boots and debug options for the mHume.

At this stage, the generated mHume `C` file for each MicroBlaze use the Library Generator to build system-specific library C functions that map the mHume generated `C` functions with the peripherals functions and configure the C libraries. For clarification, the Library Generator uses the provided mHume configuration to setup the STDIN/STDOUT for the generated `C` files using the STDIN and STDOUT attributes in the MSS file and the *INBYTE* and *OUTBYTE* attributes in the Microprocessor Peripheral Definition (MPD) file. Moreover, the Library Generator writes a *xparameters.h* header file which must be included in the header files. The *xparameters.h* file provides essential information for driver function calls and the base addresses of the peripherals in the system. Moreover it includes the communication ID for each FSL bus which is the only identification MicroBlazes uses for communication over the FSL bus interface.

Then the EDK uses the Platform Generator to build the hardware files, which include the system netlists and HDL code and BlockRAM netlists initialised with the program code. These hardware files are then used by the synthesis toolchain to create the final hardware system (i.e. the MicroBlaze processor and its peripherals) on the FGPA.

7 Software Apparatus

To examine the implementation of our parallel FPGA design, we structured a mHume example with six boxes, *multi-sq*. The first box, *gen*, similar to `gen` above, generates an integer number and passes it to four other boxes. These boxes, *sq1*, *sq2*, *sq3* and *sq4*, all similar to `square` above, calculate the square of the integer received from the *gen* box by using iterative sum and count operations, and then each box passes its output to the *conq* box. The *conq* box sums all the values it has received and prints the output to the STDOUT. After this, the program proceeds to the next cycle and the *gen* box increases the generated value by one. The *multi-sq* program repeats this cycle N times according to the user input. The source code of the program can be obtained from [9].

To run this example in the parallel FPGA design described above, we assign each box to a different MicroBlaze as shown in Figure 7

8 Evaluation

In this section we report and analyse the performance of the *multi-sq* example, comparing the performance of the parallel FPGA with the performance of both a sequential FPGA design and an Intel PC with a 2.33 GHz CPU. We also

Input	Run-time (s)		Speedup
	6 MicroBlaze 100 MHz	Intel Xeon 2.33GHz	
10	0.0000023	0.000005	2.17
100	0.0000203	0.00026	12.80
500	0.0001003	0.006319	63.00
1000	0.0002003	0.025267	126.14
1500	0.0003003	0.053642	178.62
2000	0.0004003	0.091355	228.21
3000	0.0006003	0.199057	331.59
4000	0.0008003	0.35047	437.92
5000	0.0010003	0.545584	545.42
10000	0.0020003	2.170327	1085.00

Table 1. Run-time (s) performance

Input	Clock Cycle		Speedup
	6 MicroBlaze 100 MHz	Intel Xeon 2.33GHz	
10	230	11650	50.65
100	2030	605800	298.42
500	10030	14723270	1467.92
1000	20030	58872110	2939.19
1500	30030	124985860	4162.03
2000	40030	212857150	5317.44
3000	60030	463802810	7726.18
4000	80030	816595100	10203.61
5000	100030	1271210720	12708.29
10000	200030	5056861910	25280.51

Table 2. Clock cycle performance

compare the performance of optimised and non-optimised hand written pure `C` code for the multi-sq program in the sequential architectures and the parallel FPGA design.

8.1 Parallel FPGA Performance vs Sequential FPGA and Intel PC

In Tables 1 and 2, the first column shows the user input to the *multi-sq* program. The second column reports the run-time using the parallel FPGA design and utilising six MicroBlazes of 100MHz, in Seconds and Clock Cycle respectively. The third column, in both Tables 1 and 2, reports the run-time using a Intel Xeon PC with 2.33 GHz, in Seconds and Clock Cycle respectively. Finally, the last column, in both tables, indicates the performance improvement, speedup, between the parallel and sequential run-time.

The performance reported in Tables 1 and 2 shows a clear indication of the massive improvement in the performance under the parallel FPGA design. The advancement of the performance is more evident when clock cycles are compared due to the wide gulf here between the MicroBlaze (100MHz) and the Intel PC (2.33GHz). This immense enhancement of the performance is ascribed to a couple of reasons: the obvious one which is related to the parallel architecture and the ability to evaluate all the six boxes in the program simultaneously at the same moment.

The second reason is imputed to the realisation of the mHume super step (Section 3) in our parallel FPGA design. The super step in mHume guarantees that all boxes are at the same level of evaluation; this means they are required to wait for the slowest box regardless of whether or not its output is relevant to any of the other boxes to be in the next stage of evaluation. The super step restriction produces the quadratic behaviour of the *multi-sq* example as reported in Tables 1 and 2.

In the parallel FPGA implementation, the super step and wiring of mHume is carried out by the FSL buses. To ensure that the parallel FPGA design has

the same guaranteed level of correctness that the super step pledges by restricting the boxes to the same level of evaluation, FSL buses have been formed by a single depth of an asynchronous FIFO. For instance in the *multi-sq* example, the *gen* box generates set of data/inputs to *sq1*, *sq2*, *sq3* and *sq4* boxes, and the data is pushed to the correspond FSL buses. At this stage *gen* is not restricted to wait to coordinate with other boxes to read and evaluated the anticipate data from the buses, as expected from the mHume super step design. On the contrary, *gen* starts its next cycle and produces the next set of data. However, if the generated data is to be pushed to FSL buses where the boxes corresponding to the previous set of data had failed to pull it out, the *gen* box status becomes $BLOCKED\_OUTPUT$ until the other boxes pull out the data from the FSL buses. This will ensure the correctness of the program and avoid any possible conflicts between boxes with regards to evaluations. Moreover, this strategy enhances the performance of the program under the parallel FPGA design as shown in the linear scale of performance in regard to the size of input.

8.2 Parallel FPGA Design Performance Against Hand Written C

For a sanity check, in this subsection we compare the performance of our parallel FPGA design and an Intel Xeon PC against a hand written `C` implementation of the *multi-sq* example. In this comparison we use two different hand written `C` implementations, a fully optimised one and a completely non optimised code for the same problem. The `C` code to be run on the parallel FPGA is compiled and optimised using Xilinx `gcc` compiler.

Input	Run-time (s)				Speedup based on		
	MicroBlaze, 100 MHz		Intel Xeon, 2.33GHz				
	Seq	Par 6	Optim	NO Optim	Seq	Optim	NO Optim
10	0.00002	0.000001	0.000001	0.000003	17.89	0.83	2.50
100	0.00029	0.000009	0.000002	0.000069	31.39	0.22	7.42
500	0.00175	0.000045	0.000008	0.001550	38.67	0.18	34.22
1000	0.00376	0.000090	0.000015	0.006117	41.66	0.16	67.74
1500	0.00589	0.000135	0.000022	0.013691	43.54	0.16	101.19
2000	0.00804	0.000180	0.000029	0.023655	44.61	0.16	131.20
3000	0.01256	0.000270	0.000043	0.04901	46.47	0.16	181.32
4000	0.01712	0.000360	0.000058	0.086294	47.53	0.16	239.51
5000	0.02187	0.000450	0.000072	0.132634	48.58	0.16	294.55
10000	0.04635	0.000900	0.000143	0.518483	51.48	0.16	575.90

Table 3. Run-time performance of running C code of the multi-sq example on embedded processors versus desktop PC

In Table 3, the first column shows the input values for *multi-sq*. The second and third columns report the execution time using a single MicroBlaze and a

Input	Clock Cycle				Speedup based on		
	MicroBlaze, 100 MHz		Intel Xeon, 2.33GHz				
	Seq	Par 6	Optim	NO Optim	Seq	Optim	NO Optim
10	2147	120	2330	6990	17.89	19.42	58.25
100	29193	930	4660	160770	31.39	5.01	172.87
500	175177	4530	18640	3611500	38.67	4.11	797.24
1000	376253	9030	34950	14252610	41.67	3.87	1578.36
1500	589105	13530	51260	31900030	43.54	3.79	2357.73
2000	804405	18030	67570	55116150	44.61	3.75	3056.91
3000	1256109	27030	100190	114193300	46.47	3.70	4224.69
4000	71712709	36030	135140	201065020	47.54	3.75	5580.49
5000	2187709	45030	167760	309037220	48.58	3.73	6862.92
10000	4635317	90030	333190	1208065390	51.49	3.70	13418.48

Table 4. Clock cycle performance of running C code of the multi-sq example on embedded processors versus desktop PC

parallel design of six MicroBlazes respectively. The fourth and fifth columns outline the run-time of the optimised and non-optimised `C` code using Intel Xeon PC with 2.33 GHz CPU. The last three columns reveal the run-time improvement, speedup, of the `C` code using the parallel FPGA design of six MicroBlazes against the single MicroBlaze and the Intel PC running the optimised and non optimised `C` implementation. Table 4 discloses the clock cycle for the run-times presented in Table 3, and reports the performance based on the clock cycle.

The performance results reported in Table 3 show sustained improvement of the hand written `C` code under the parallel FPGA design, particularly in comparison with the single MicroBlaze and the non optimised code performance. A marginal improvement against the optimised `C` code is is shown in the seventh column of the table. However, Table 4 exhibits preferable capabilities for the parallel design when clock cycle is considered for measurement, even against the optimised `C` code. The FPGA board used in this experiment is about 10 years old: a more recent board will have faster a clock cycle than the current 100MHz per MicroBlaze and that will result in a smaller run-time and an improvement in the performance against the Intel PC architecture.

9 Related Work

There have been some attempts to extend functional-based programming languages to use FPGAs:

- Lava [2, 3] extends Haskell with operations that allow the high-level description of FPGA circuits.
- Intel's reFL^{ect} [6] is strongly typed and similar to ML, but with quotation and anti-quotation constructs. Its features are intended for applications in hardware design and verification.

– MetaML [13] is very similar to Intel's reFL^{ect} with more direct focus on program generation, and control and optimization of evaluation.
– Template Haskell [14] is also focused on program generation, and the control and optimization of evaluation. To support this it generates code at compile time. An example of a HDL embedded in Haskell using Template Haskell can be found in [12].
– The functional derivation approach is used for deriving FPGA circuits from Haskell specifications [8].
– The Reduceron is an FPGA-based reduction machine targeting Haskell [11]

Our work, as presented in this paper, is novel in adopting a soft processor approach and in attempting to follow a complete development path from source language to target FPGA hardware.

10 Conclusion

We have shown that a simple model of, and compilation route for, mHume can deliver very decent parallel performance on an FPGA, through direct realisation of box concurrency on MicroBlaze soft cores.

mHume is, of course, an experimental language, with data and computation constructs restricted to integers. Nonetheless, as mHume includes the key features of Hume's coordination layer, our preliminary results give us confidence that equally good performance can be achieved for increasingly large Hume subsets. Thus, we next plan to systematically extend mHume with: character, float and string types; vectors; input/output; and auxilliary definitions of functions and constructed types.

Our compiler generates sequential C and substantial hand modification is needed to parallelise it. Thus, a key priority is to fully automate the generation of C with appropriate concurrency constructs, both for MicroBlaze and wider multi-core execution. This will then support mHume execution on heterogeneous systems combining multi-core with FPGA, in the first instance by user nomination of box placement, and, in the longer term, with automatic placement driven by cost analysis.

In the longer term we also plan to further optimise the C generated for mHume, in particular by identifying self-output boxes, and by implementing the Hierarchical Hume box encapsulation[5].

Acknowledgements

This research is supported by the UK EPSRC Islay project (EP/F030592, EP/F030657, EP/F03072X, and EP/F031017) "Adaptive Hardware Systems with Novel Algorithmic Design and Guaranteed Resource Bounds". We are also grateful to *Xilinx* for support through their University Programme.

References

1. A. Al Zain, W. Vanderbauwhede, and G. Michaelson. Hume on fpga. In *Draft Proceedings of 10th International Symposium on Trends in Functional Programming (TFP10)*, University of Oklahoma, Oklahoma, USA, 2010.
2. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 1998.
3. K. Claessen and G. J. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits '02, Grenoble, France*, April 2002.
4. S. Craven and P. Athanas. Examining the viability of FPGA supercomputing. *EURASIP Journal on Embedded Systems*, 2007(1):13, 2007.
5. G. Grov and G. Michaelson. Towards a Box Calculus for Hierarchical Hume. In M. Morazon, editor, *Trends in Functional Programming*, volume 8, pages 71–88. Intellect, 2008.
6. J. Grundy, T. Melham, and J. O'leary. A reflective functional language for hardware design and theorem proving. *Journal Functional Programming*, 16(2):157–196, 2006.
7. K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. GPCE 2003: Intl. Conf. on Generative Prog. and Component Eng., Erfurt, Germany*, pages 37–56. Springer-Verlag LNCS 2830, Sep. 2003.
8. J. Hawkins and A. Abdallah. Behavioural synthesis of a parallel hardware jpeg decoder from a functitonal specification. In *Proc. EuroPar 2002*, August 2002.
9. G. Michaelson. multisq: minihume example. ONLINE, June 2010. http://www.macs.hw.ac.uk/~ceeatia/IFL10-miniHume/multisq.hume.
10. G. Michaelson, G. Grove, and A. Al Zain. Multi-core parallelisation of hume through structured transformation. In *Draft Proc. of 21st Intl. Workshop on Implementation and Application of Functional Languages (IFL '09)*, Seton-Hall University, New Jersey, USA, 2009.
11. M. Naylor and C. Runciman. The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. *Implementation and Application of Functional Languages*, pages 129–146, 2008.
12. J. Odonnell. Embedding a hardware description language in template haskell. *Domain-Specific Program Generation*, pages 195–265, 2004.
13. E. Pasalic, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *In the International Conference on Functional Programming (ICFP 02*, pages 218–229. ACM, 2002.
14. T. Sheard and S. Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, 2002.
15. Xilinx support documentation. Online White Papers. http://www.xilinx.com/support/documentation/white_papers/.
16. Xilinx. MicroBlaze Processor Reference Guide (v 4.0). Technical report, Xilinx, 2004.