TOWARDS HUME SIMD VECTORISATION

Abdallah Al Zain, Valerie Gibson, Greg Michaelson

School of Mathematical & Computer Sciences, Heriot-Watt University, Riccarton, Scotland, EH14 4AS
phone: (+44) 131 451 4197, fax: + (+44) 131 451 3732, email: {ceeatia,vg8,greg }@macs.hw.ac.uk
web: www.macs.hw.ac.uk

Kevin Hammond, Steffen Jost

School of Computer Science, University of St Andrews, Scotland, KY16 9SX
phone: (+44) 133) 463241, fax: (+44) 1334 463278 email: {kh,jost}@dcs.st-and.ac.uk
web: www.dcs.st-and.ac.uk/~kh

Hans-Wolfgang Loidl

Ludwig-Maximilians University, D-80339 Munich, Germany
phone: (+49) 89 / 2180 9864 ,fax: (+49) 89 / 2180 9338,email: hwloidl@tcs.ifi.lmu.de
http://www.tcs.informatik.uni-muenchen.de/~hwloidl/

ABSTRACT

Hume is a novel formally-motivated programming language oriented to developing software where strong assurance of resource use is paramount, in particular embedded systems. In this paper, we explore the use of Hume in a context of heterogeneous platforms where resource knowledge may guide the mapping of activities to different platform components. We present an overview of the Hume language design and methodology, and discuss its deployment in the exploitation of SIMD vectorisation of a simple low-level image processing routine.

1. INTRODUCTION

Contemporary hardware platforms offer considerable challenges beyond traditional CPUs in integrating on-chip shared memory multi-core and SIMD vector processing, with GPU and FPGA co-processing. It is often difficult to map standard sequential algorithms onto such platforms to gain optimal performance benefits from such heterogeneous opportunities.
We have recently begun to elaborate an approach based on the novel Hume programming language which has been designed to enable highly accurate analyses of software component resource use. In particular, generic models of Hume time (WCET) and space resource use may be parameterised on hardware specific properties to enable direct comparisons of likely behaviours of the same components in different hardware realisations. And this may, in principle, be achieved without actually constructing and instrumenting different concrete versions of such components.

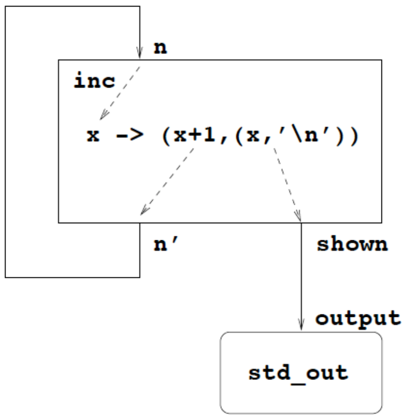
2. HUME

Hume [4] is a contemporary language in the functional tradition, oriented to applications where a high degree of confidence is required that programs meet resource constraints such as time and space needs. Its strength lies in the very tight integration both of its formal design, offering strongly coupled semantics and type-based cost models, and of its tool chain, with equally closely coupled static resource analyses and native code compilation.
Hume embodies a number of novel design choices which underpin its satisfaction of practical, applied and declarative concerns. It draws strongly on modern functional languages like SML and Haskell, providing recursive functions over rich polymorphic types. However, rather than offering a traditional monolithic language, Hume explicitly separates coordination, realised as concurrent finite state boxes communicating over wires, from control, realised as generalised transitions with boxes from patterns over inputs to expressions over outputs. Boxes are repeatedly invoked, consuming inputs and generating outputs, but each box invocation is single shot and pure declarative, with no internal state persisting between invocations. Thus, by varying the types on wires and the expressiveness of control within boxes, Hume may be treated as a family of languages from full-Hume, which is Turing complete, to FSM- (Finite State Machine) and HW-(Hardware) Humes with highly accurate cost models and analyses.
Hume is implemented via a core abstract machine which unifies cost modelling and compilation. A common front end parses and type-checks Hume programmes, and constructs an abstract syntax tree (AST). The native code compiler then traverses the tree generating C for subsequent compilation via gcc. Similarly, the cost analysers traverse ASTs calculating costs. Cost analysis may be integrated into the compiler to guide space allocation and time constraint checking.
Hume has been deployed in a wide range of applications, including real-time control and image analysis for the Pioneer P3-AT robot and the Cycab autonomous vehicle, both under Linux, and for low level, stand-alone embedded use on a Renesas M32C board with 24K of memory.
For example, the following program generates a sequence of ascending integers on the standard output, starting with 0. Note that line numbers are not part of Hume but are used in subsequent discussion.

```
1 -- counter - inc.hume
2 program
```

```
3 stream output to "std\_out";
4 box inc
5 in (n::int 64)
6 out (n'::int 64,shown::(int 64,char))
7 match
8 x -> (x+1,(x,'\n'));
9 wire inc
10 (inc.n' initially 0)
11 (inc.n,output);
```

1: Comments begin with --.
2: Programs start with `program` or `module`.
3: The stream `output` is associated with standard output, named through the string `"std out"`.
4: The box is named `inc`.
5: The box has one input called `n`, a 64 bit integer.
6: The box has two outputs. `n'` is a 64 bit integer. `shown` is a tuple of a 64 bit integer and a character.
7-8: If there is a value on input `n`, then the pattern `x` will set a new local variable called `x` to that value and evaluate the right hand side. Output `n'` will be set to `x+1`. Output shown will be set to (x+1,'\n') i.e. `x+1` followed by a newline character.
9: Box `inc` is now wired by position.
10: `inc`'s input `n` is wired implicitly to `inc`'s input `n'`. `n` is initialised to `0`.
11: `inc`'s output `n'` is wired implicitly to `inc`'s input `n`. `inc`'s output `shown` is wired to `output` and hence to `"std out"`.
The program is illustrated in the following Figure



3. HUME COST ANALYSIS

Determining the cost of executing a sequence of instructions can be easy for a given machine model, if each instruction has a constant cost. It is also possible to determine a useful upper bound on the execution costs if the variation in cost is quite small for each instruction, such as bounding the cost of adding two arbitrary integer. However, in practise the cost of many instructions varies significantly, depending on arguments and/or the overall state of the machine. As an extreme example, a function call might return after an arbitrarily large amount of time, which makes it useless to assume the worst case (never). Therefore one needs to consider all possible machine states at the start of the computation and track all possible state transformations during execution, which is a tedious and generally infeasible approach. The only solution to this problem is to abstract all possible states into a smaller, more manageable classes of equivalent states. Our approach is especially radical, since we abstract the entire state and represent it by a single, nonnegative rational number, referred to as the *potential* of the machine state. Note that we will never actually compute this number, the potential, for any actual machine state other than the initial state. Instead, we examine the effect of each operation on the overall potential and define the *amortised cost* [5,6] of an instruction as a suitable constant such that:

amortised_cost = actual_cost - potential_before + potential_after

holds for all possible states, with equality being preferred. The benefit is that determining the amortised cost for a sequence of operation is very easy, since the amortised cost is constant and does not depend on the machine. The actual cost of the entire sequence is then bounded by the sum of the amortised costs plus the potential of the initial state.
Automatically performing the analysis means that we first construct a standard typing derivation. Next, each constructor is then assigned a resource variable (ranging over non-negative rational numbers), representing the potential credited by each node of that constructor of that particular type (note that types may differ by their potential only). The analysis generates a set of constraints over those variables, according to the dataflow and the actual cost occurring in each possible path of computation.
For example, the actual worst-case execution costs for the Renesas M32C/85U processor have been determined by the aiT tool [8] for machine-code level instructions. Each instruction of the source program is examined precisely once. Loops in the source program are dealt with by identifying some resource variables contained in the constraint set. The generated constraint sets are well behaved and can easily be solved by using a standard LP-solver, such as [1]. In this way, bounds on resource consumption are associated with each expression in the program through their types. The potential annotated types then give rise to a linear closed form expression, depending on the input sizes, which represents an upper bound on the execution costs. Because we have used a type-based approach, we have already formally proven that our analysis will always give a *guaranteed upper bound* on resource consumption

4. VECTORISATION

SIMD processors are optimised for parallel operations on contiguous bit sequences, typically 128 bits, which are relatively long by comparison with CPU's 32 or 64 bits but extremely short compared with data sequences measured at least in kilobytes. Thus, opportunities for vectorisation are found deep in programs in the lowest-level bodies of loops, processing linear sequences of values which may be easily coerced to sequences of adjacent bytes.

General purpose CPUs have provided additional SIMD vectorisation for at least 10 years. However, with the notable exception of Cockshott's Vector Pascal[2], programming language implementations have been curiously slow in offering portable technologies for exploiting SIMD processing. Commonly, programmers write sub-systems themselves in processor-specific assembly language or call out to compiler-specific SIMD libraries.

Our long term objective is to use Cockshott's approach to identifying vectorising oppostunities. However, where Cockshott uses a universal abstract machine language, with translators to and from source and target languages, we intend to analyse directly Hume vector operations in the Hume Abstract Machine code. As a major step towards this, we are exploring the hand vectorisation of substantial image processing algorithms in C, identifying a core set of vectoriseable tropes in this domain, and then exploring their use in Hume, again by hand in the first instance.
There are two possible routes to such deployment of vector routines. First of all, Hume provided a foreign function interface (FFI) for accessing and automatically compiling and linking external routines written in other languages, in particular C/C++ and Java. For an FFI

call, there must be a mapping from the Hume types to the foreign types and back, and correspondences between the function calling conventions in parameter passing.
Secondly, Hume also provides a skeletal `operation` box which may be associated with potentially "unsafe" foreign function calls, that is calls whose effects on the internal states of Hume programs are unpredictable. The `operation` form has similar language correspondence requirements to FFI, but enables external activities to be more cleanly delineated at the box level.
We are currently completing a first set of experiments in using the FFI route to exploit vectorisation: these are discussed next.

5. EXAMPLE: EDGE DETECTION

To illustrate the potential for foreign function vectorisation in Hume, we consider a simple edge detection algorithm. In our experiment, the program takes a 240*240 image and runs a standard convolution method. This is done by using a Gaussian mask (a 5*5 matrix) and sliding it over every viable pixel in the image. An edge is a sharp difference in the intensity of a pixel and its surrounding elements and the Gaussian mask helps to enhance these intensities.

We assume that the reader is familiar with this technique, and focus on its expression in Hume and how vectorisation opportunities may be exploited.

First of all, we declare basic Hume types aliases, in particular for sized integers:

```
type integer = int 32;
type Int = int 32;
```

and for the vectors we will manipulate:

```
type pixel = vector 1..3 of integer;
type irow = vector 1..240 of pixel;
type iImage = vector 1..240 of irow;
type mrow = vector 1..5 of pixel;
type mImage = vector 1..5 of mrow;
type drow = vector 1..236 of pixel;
type dImage = vector 1..236 of drow;
type pixelCH = vector 1..3 of char;
type im_rowCH = vector 1..240 of pixelCH;
type ImageCH = vector 1..240 of im_rowCH;
```

The conversion from integer to character and back makes use of Hume's high level vector operation `vecmap` which applies a function to every element of a vector:

```
c2i i = vecmap i c2i_c; c2i_c c = vecmap c c2i_p3; c2i_p3
c = vecmap c c2i_p;
c2i_p g = (g::char) as Int;

numCh i = vecmap i numCh_2; numCh_2 c = vecmap c numCh_3;
numCh_3 c = vecmap c numCh_4; numCh_4 g = if g>40 then 255
else 0;

i2c i = vecmap i i2c_c; i2c_c c = vecmap c i2c_p3;
i2c_p3 c = vecmap c i2c_p; i2c_p g = (g::Int) as char;
```

We need a function to access elements of 2D images:

```
get5 v n p = ...
```

and various masks for convolutions:

```
minus1 = <<(-1),(-1),(-1)>>;

mask = <<  <<minus1, minus1, minus1,minus1, minus1 >>, ...
>>;
```

To aid vectorisation, we make explicit the base operations on pixels and construct an explicit systematic hierarchy of image manipulation functions (`rgbconv, colconv, matconv, colsum, matsum`).

We also require functions to ensure that after convolution all values are within the RGB range (`newvalue`, `newpixel`), to find the average values of pixels and fill pixels with average values (`avgpix`, `finalpix`).

The main convolution is a nested hierarchy of recursive calls to process rows by columns:

```
MAXR = 236; MAXC = 236;

col i j n v =
 if j>n   then []
 else finalpix
         (newvalue
            (avgpix
              (newpixel
                (matsum
                  (matconv
                    (get5 v i j ) mask))))):
         col i (j+1) n v;

row i m v =
 if i>m then []
 else col i 1 MAXC v++row (i+1) m v;
```

Auxilliary functions are required to pad out the image to the original size (`pad`), convert list to a linear or 2D vector (`listToVec2`) , construct a vector of zero pixels (`makeZero`), take and drop elements from lists, and construct new vectors and images of the same shape as exemplars (`makerow`).

Finally, the main program(`run`):

```
run image = numCh (listToVec2 (pad 1 240 240 3 3 236 236
(row 1 MAXR image)));
```

is embodied in a loop that converts and integer sequence to characters for manipulation (`i2c`) and calls the top-level processing function:

```
box runable
  in (imgch::iImage)
  out (out1::ImageCH)
  match
   (j)->(i2c (run j));
```

In turn, the box is connected to standard input and output:

```
wire runable (input)(output);
```

6. VECTORISATION IN C

These days, many C compilers try to output vector instructions but this is non-trivial. For one thing, vector units are often quite picky about alignment; while a CPU might only require alignment on 4-byte boundaries for loads, its vector unit could need data aligned on 16-byte boundaries. For another, the compiler has to make sure that it can make the operations happen at the same time without altering the program semantics of the code [9].
One solution to this problem is to use vector intrinsics, which look like normal C functions, but have a one-to-one mapping with vector instructions. This isn't a new concept; the square root instruction on architectures such as x86 is typically generated in the same way by compilers.
Using vector intrinsics has a significant disadvantage, however. Vector instruction sets differ between architectures. Using intrinsics, which have a one-to-one mapping with instructions, restricts code to one architecture. It also prevents code from working on older chips that don't have a vector unit.

What is really needed is a method of writing vector code that isn't tied to a specific instruction set. Sadly, the C specification does not provide this mechanism. Fortunately, however, the GNU Compiler Collection (GCC) does.
The additions to C for supporting vector operations in GCC are very simple. Vector types are defined with `typedef`, just as scalar types are. The only difference is an attribute that indicates the number of elements in the vector. The following example shows how to define a vector of four signed integers:

```
typedef int v4si __attribute__
        ((vector_size(4*sizeof(int))));
```

The only constraint is that the size must be a power of two multiple of a scalar type. Once this type is defined, it can be operated on it as if it were a scalar type.
However, everything comes with a price, and in this case the price is compiler compatibility. It is straightforward to move pure ISO C from GCC to any other standards-compliant C compiler, such as the Intel C++ Compiler (ICC) or XL C, that might give better performance on a particular architecture. But once GCC extensions are used, only GCC can be deployed subsequently. However, if every platform targeted is already supported by GCC, the code is not for distribution, or GCC extensions are already using elsewhere, this restriction may not matter.
First, the necessary vector data types are provided. This is done using an appropriate `typedef`, for instance the declaration of a pixel causes the compiler to set the mode for the `pixel_4` type to be 32-bit int which means a vector of 4 units of 4 bytes.

```
typedef                 int                 pixel_4
__attribute__((vector_size(4*sizeof(int32_t))));
typedef union pixel_4__
{
  pixel_4 v;  int32_t i[4];
}pixel_4_;

typedef                 int                 row_16
__attribute__((vector_size(16*sizeof(int32_t))));
typedef union row_16__
{
  row_16 v;  int32_t i[16];
}row_16_;

typedef                 int               image_128
__attribute__((vector_size(128*sizeof(int32_t))));
typedef union image_128__
{
  image_128 v;  int32_t i[128];
}image_128_;
```

Note that `image_128_` is a `union` of a vector `(v)` and an array of pointers to each element `(i)`. The latter is needed when an operation is not supported by vectorisation and the array must be cast to a normal array.
These types are then used to define the variables that will be mapped to SIMD registers in the host CPU. If a function is declared as inline extern, the compiler does not generate a non-inline version of it; which means it is not possible create a pointer to the function, but it prevents linker errors from multiply-defined symbols:

```
#define          INLINE          inline          extern
__attribute__((always_inline))

typedef int32_t pixel_[4];
typedef int32_t pixel_3[3];
typedef pixel_  row_[240];
typedef pixel_3 row_3[240];
```

The main function is identified for vectorisation by the generic `INLINE`. By declaring a function inline, GCC can be directed to make calls to that function faster. GCC `inline` achieves better integration of the function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the `inline` function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case.

```
INLINE HEAP * image_ (int maxRow, int maxCol, HEAP * or-
igImage)
{
  row_3 tmpImage[240], final_image[240];
  image_128_ img5x5, mask_image;
  row_16_ tmp_row;  pixel_4_ final_pixel;
  int32_t final_value;
  HEAP * newImage;
  int i,j,i1, j1, co, ro, index, j_tmp, avgPixel;

image_128 mask__ = { ...};
row_16 row_zero = {...};
pixel_4 pixel_zero = {0,0,0,0};
mask_image.v = mask__;
heapToBytes((int8*) tmpImage, origImage);

  for(ro=0; ro< maxRow; ro++){
    for(co=0; co< maxCol; co++){
      index = 0;
      for(i=ro; (i < (ro+5)) && (i < maxRow); i++)
for(j=co; (j< (co+5)) && (j< maxRow); j++){
img5x5.i[index++] = tmpImage[i][j][2];
        img5x5.i[index++] = tmpImage[i][j][1];
        img5x5.i[index++] = tmpImage[i][j][0];
        }
```

In the next fragment, in a unitary vector operation, each of the 25 elements in `img5x5.v` will be multiplied to the corresponding 25 elements in `mask_image.v` and the resulting vector will be stored in `img5x5.v`:

```
  img5x5.v = img5x5.v * mask_image.v;
```

Here, `img5x5` and `mask.image` are of type `image_128_` which includes an explicitly nominated vector so GCC can insert low-level SIMD code for these operations.
One of the advantages of vector operations is that a vector type can be specified as a return type for a function, vector types can also be used as function arguments, and vectors can be assigned to other vectors:

```
final_pixel.v = pixel_zero;
tmp_row.v = row_zero;
```

One of the disadvantages is that it is not possible to operate between vectors of different lengths or different signedness without a cast. Furthermore, as noted above, if the vector operation is not supported directly by GCC, the vector type needs to be cast to an array:

```
for(j_tmp=0; j_tmp<15; j_tmp++){
   for(j=j_tmp; j<75;j=j+15)
 tmp_row.i[j_tmp] = tmp_row.i[j_tmp]+
                    img5x5.i[j];
 }
 for(i=0; i<15; i=i+3){
   final_pixel.i[0] = final_pixel.i[0] +
                      tmp_row.i[i+0];
   final_pixel.i[1] = final_pixel.i[1] +
                      tmp_row.i[i+1];
   final_pixel.i[2] = final_pixel.i[2] +
                      tmp_row.i[i+2];
 }

newImage = bytesToHeap((int8*) final_image)       return
newImage;
}
```

Hume now calls vector code as:

```
foreign import ccall "hlib.h i2cRow" i2cR_ ::
        row_240 -> im_rowCH;
foreign import ccall "hlib.h i2cPixel" i2cP_ ::
        pixel -> pixelCH;
foreign import ccall "hlib.h  image_"  image_ ::
        Int -> Int -> image_240x240 -> image_240x240 ;
```

to replace the original calls to convert integers to characters and pixels, and for the convolution itself. Note that only one line is needed to initiate each of the external C functions.

7. DISCUSSION

The improvement for this simple vectorisation is dramatic. The original Hume time for processing a 240*240 pixel image is of the order of 2280 milliseconds, whereas the vectorisation reduces this by almost 60% to 956 millisecond.

While acknowledging that the example is simple, and that unnecessary overheads are introduced by the conversion between vectors and lists, we think that this initial experiment shows considerable potential for improvement of vector based image processing in Hume.

However, such vectorisation still remains something of a black art. Thus we will seek to:

a) systematically elaborate a library of domain specific vector operations;
b) embed such operations as Hume box operations;
c) explore the identification of opportunities for vectorisation in high-level Hume and in HAM;
d) develop automatic parallelisation that can substitute/wire appropriate FFI calls/operations for such opportunities;
e) build cost models of vector operations on well characterised SIMD processors;
f) inform automatic parallelisation using the analyses derived from the cost models.

Vectorisation seems most applicable at the Hume expression level within boxes. In the longer term, we are also exploring the exploitation of multiple cores using multi-threading technologies across Hume boxes. It will be extremely interesting to investigate how SIMD and multi-core acceleration may optimally complement each other. We anticipate that out cost models and analyses will play a central role in such optimisation in determining the relative benefits of different balances of SIMD and thread parallelism.

ACKNOWLEDGEMENTS

This research is supported by UK EPSRC project EP/F030592/1 `Adaptive Hardware Systems with Novel Algorithmic Design and Guaranteed Resource Bounds'. We would like to thank our colleagues Andy Wallace and Jing Ye for their collaboration.

REFERENCES

1. M. Berkelaar, K. Eikland, and P. Notebaert. lp solve: Open source (mixedinteger) linear programming system. GNU LGPL (Lesser General Public Licence). http://lpsolve.sourceforge.net/5.5.

2. P. Cockshott and K. Renfrew. SIMD Programming Manual for Linux and Windows. Springer, 2003. ISBN 1-85233-794.

3. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a reallife processor. In EMSOFT, pages 469–485. Springer-Verlag LNCS 2211, 2001.

4. K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In Proc. 2003 Intl. Conf. on Generative Programming and Component Engineering, – GPCE 2003, Erfurt, Germany, pages 37–56. Springer-Verlag LNCS 2830, Sep. 2003.

5. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pages 185–197. ACM, 2003.

6. M. Hofmann and S. Jost. Type-based amortised heap-space analysis (for an objectoriented language). In P. Sestoft, editor, Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems, volume 3924 of LNCS, pages 22–37. Springer, 2006.

7. C. Okasaki. Purely Functional Data Structures. Cambridge University Press, 1998.

8. R. E. Tarjan. Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods, 6(2):306–318, April 1985.

9. D. Chisnall. Vector programming with GCC. InformIT rticle is provided courtesy of Prentice Hall Professional, March 30 2007.