Rapid codesign of a soft vector processor and its compiler

Matthew Naylor and Simon W. Moore
Computer Laboratory, University of Cambridge, UK
{matthew.naylor,simon.moore}@.cl.cam.ac.uk

***Abstract*—Despite a decade of activity in the development of soft vector processors for FPGAs, high-level language support remains thin. We attribute this problem to a design method in which the high-level vector programming interface is only really considered once the processor architecture has been perfected, by which point the designer may be committed to the time-consuming development of a complicated compiler. In this paper, we present the *codesign* of a soft vector processor and a lightweight compiler, which *together* lift the level of abstraction for the programmer while allowing a rapid compiler implementation phase. We demonstrate the effectiveness of our approach on a range of applications from digital signal processing, neuroscience, and machine learning.**

I. Introduction

Motivation. A soft vector processor is a highly-reusable FPGA core allowing certain applications to be programmed largely, or entirely, in software, avoiding slow hardware development cycles [1]. Furthermore, it can provide excellent performance. Even a small vector core, fitting in around 10% of a medium-sized FPGA, can saturate off-chip DDR2 bandwidth in serious parallel applications, thus matching the run-time performance of full-custom hardware pipelines [3], but implemented with a fraction of the effort. Compared to the use of multiple scalar cores, a single vector core offers superior performance per unit area, allowing more compute capability to be packed into a single FPGA [3].

Our broad motivation is to provide support for the development of massively parallel applications using large FPGA clusters. A key attraction of using commodity FPGA boards for this purpose is thay they are already equipped with the advanced communication capabilities needed to obtain a high degree of scalability. Another attraction is that FPGAs are highly customisable, not limited to any particular model of computation, but this is a double-edged sword: application development is slow if a *full*-custom approach is taken. Therefore it is vital to provide reusable cores, such as soft vector processors, to enable rapid development with judicious use of full-custom pipelines only where necessary.

Problem. Although the use of a soft vector processor can greatly reduce development time, programming one can still involve a plethora of low-level chores – off-putting to potential users. To illustrate, suppose we wish to code the simple *SAXPY* benchmark [4], shown below, using an existing vector core.

```
for (int i = 0; i < n; i++)
  Y[i] = a*X[i] + Y[i];
```

Figure 1 shows an implementation of this benchmark using Altera's NIOS-II processor with BlueVec vector extensions [3]. Vector instructions, such as `Load`, `Commit`, `AddW`, `MulW`, and `Store` are all C macros that expand to inline assembly code.

```
void saxpy(int a, int* X, int* Y, int n) {
 Load(v0, X);      // Load 8 ints into vector
 Load(v1, Y);      // registers v0 and v1.
 for (int i = 0; i < n/8; i++) {
  Commit;          // Commit prior loads.
  Load(v0,X+8);    // Pre-fetch data for
  Load(v1,Y+8);    // next iteration.
  MulW(v0,a,v0);   // Apply the
  AddW(v1,v0,v1);  // SAXPY equation.
  Store(v0,Y);     // Update array Y
  X+=8; Y+=8;      // Move to next 8 objects.
 }
}
```

Fig. 1. The *SAXPY* benchmark using the BlueVec soft vector processor. Note the assembly-level description with explicit chunking of input arrays into vectors of size 8, software pipelining for parallel fetch and compute, and manual register allocation.

The details will be explained shortly, but for now it suffices to see that even this trivial benchmark involves a number of low-level programming chores such as array chunking (or "strip mining"), software pipelining, manual register allocation, and assembly-level coding. This kind of low-level programming is very typical of soft vector processors in general [1], [2], [5], [6]. Although the goal to simplify programming is often left as future work, this ignores the opportunity for the programming interface to influence the processor architecture, and may lead to overly-complicated implementation [4].

Contribution. Hardware/software codesign is a method that aims to exploit the synergy between hardware and software in order to optimise a system's design goals [7]. In this paper, we present the codesign of a soft vector processor – *BlueVec-II* – and its compiler, where two of the main design goals are: (1) to support a higher-level programming interface; and (2) to do so within a short development period. As a quick introduction, Figure 2 shows the BlueVec-II implementation of the *SAXPY* benchmark, which is noticeably simpler than the original BlueVec version in Figure 1. In the next section we use these two examples, and others, to illustrate the important design choices behind the architecture and compiler.

II. Design

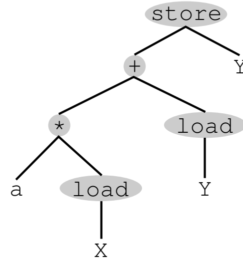
Decoupled vector processing. One of the most common properties of existing soft vector processors is that they are *tightly-coupled* to a host scalar processor. Essentially, this means that scalar and vector instructions appear together in a *single* instruction stream. Control-flow and data frequently pass back and forth between the scalar host and the vector co-processor. To illustrate, Figure 1 shows a performance-critical loop in which some instructions execute on the scalar unit, e.g. `X+=8`, some on the vector unit, e.g. `Load(v0,X)`, and some involve data transfer between the two, e.g. `Load(v1,Y+8)`

```
Kernel saxpy() {
 Param<int> a;
 Param<int*> X, Y;

 Stream<int> sX = load(X);
 Stream<int> sY = load(Y);
 store(a*sX+sY, Y);

 return kernel();
}
```

(a)



(b)

Fig. 2. The *SAXPY* benchmark using BlueVec-II. When called, function (a) constructs an abstract syntax tree (b) which is then compiled to vector code and written to instruction memory on BlueVec-II. The vector code can then be invoked when desired using the `call` function (see text body).

where the value of `Y+8` is computed on the scalar unit and passed as an argument to the vector unit.

One drawback of the tightly-coupled design is the challenge in developing a compiler which must understand both scalar code *and* vector code, as well as the interaction between the two. Given that the scalar compiler, in our case `gcc`, is already highly sophisticated, even the addition of a simple vector-code compilation pass could require a lot of work. This is especially the case for a hardware developer (not necessarily a compiler expert) who may wish to add support for a new instruction they have added to the vector unit.

To avoid this problem, BlueVec-II adopts a *decoupled* design in which the vector and scalar instruction streams are *completely separate*. Now vector code can be analysed and transformed in isolation, independently of the scalar compiler. BlueVec-II's programming interface is implemented as a lightweight C++ library that runs on the scalar host, generating vector code and writing it to BlueVec-II's instruction memory at run-time. To dynamically compile the kernel defined in Figure 2 into BlueVec-II instruction memory, we write

```
Kernel saxpyKernel = saxpy();
```

This kernel expects three parameters: `a`, `X` and `Y`. To invoke it from the scalar host, we must pass arguments to be used as values for these parameters:

```
call(n, saxpyKernel, a, X, Y);
```

where `n` is the length of all arrays processed by the kernel, in this case the lengths of `X` and `Y`. The kernel can be called as many times as required, and with different arguments each time. It can also be called asynchronously, allowing the NIOS-II and BlueVec-II cores to run in parallel, with the two threads joinable by a call to `bluevecSync()`.

Implicit chunking and double buffering. Figure 1 shows that, in the tightly-coupled approach, quite a lot of scalar code is used to keep the vector unit both busy and efficient. First, the input arrays must be split into eight-element vectors and processed in a loop since each BlueVec vector register is only eight elements wide (a process known as *strip mining*). Second, a software pipeline must be set up to allow latent memory accesses to run in parallel with compute. This is achieved by pre-fetching the first chunks of `X` and `Y` outside the loop (*priming* the pipeline) and then, in each iteration, pre-fetching the chunks that will be needed in the next iteration while processing the latest available chunks. The `Commit` instruction enables the target registers of the loads to be updated, and stalls processing until all outstanding loads have completed.

```
Kernel fir(int* filter, int ntaps) {
  Param<int*> input, output;

  Stream<int> x = load(input);
  Stream<int> sum = streamOf(0);

  for (int i = 0; i < ntaps; i++) {
    sum = sum + x*filter[i];
    x = shiftIn(0, x);
  }

  store(sum, output);
  return kernel();
}
```

Fig. 3. BlueVec-II kernel to implement an N-tap FIR filter. The same task requires around 40 lines of vector code (excluding code to implement double-buffering) using the VEGAS vector core.

In order to lift the level of abstraction for the programmer, we generalise fixed-size vectors to arbitrary-sized *streams*. Our C++ operations on streams such as

```
Stream<int> operator+(Stream<int> a,
                      Stream<int> b)
```

do not specify how many `+` operations should be done in parallel, and how many in sequence. This, along with chunking and double-buffering, is left to the vector unit hardware. This is straightforward to implement, and also efficient. It means, for example, that the explicit scalar instructions required to implement branching, pointer arithmetic, and pre-fetching in Figure 1 can be implemented as implicit background operations in the hardware. It is important, though, that BlueVec-II operates by *dataflow evaluation*, i.e. does not attempt to *fully* execute one stream instruction before considering the next – that would be inefficient in terms of the memory needed to store the intermediate results of a computation.

Vector code analysis and transformation. A soft vector processor typically contains a set of vector registers which must be explicitly managed by the programmer. As the number of vector instructions in a block of code rises, efficient manual register management becomes a real burden. This is where the benefits of a decoupled design can be applied. In Figure 2, stream functions such as `load`, `store`, `-` and `*` actually construct an *abstract syntax tree* on the NIOS-II. Only when the `kernel()` function is called are the operations compiled to vector code and transferred to BlueVec-II. Part of this process includes an automatic liveness analysis and register allocation phase. As a result, the programmer can simply declare as many streams as desired – if there is insufficient register capacity, an error will be reported. To achieve the same degree of automated code transformation in a tightly-coupled design would require a more heavyweight solution since vector instructions are intermingled with a complex scalar instruction set.

Sliding window algorithms. Existing soft vector processors are commonly applied to the implementation of sliding window algorithms [1], [2], [6]. For comparison, Figure 3 shows the BlueVec-II implementation of the classic N-tap FIR filter. Whereas the scalar implementation of an FIR filter involves sliding a N-element window (filter) across an input signal, computing a dot-product at each point, the vector version involves creating N shifted versions of the input signal such

```
Kernel ivalueAccum() {
  Param<int*> targets, weights;

  Stream<int> t = load(targets);
  Stream<int> w = load(weights);
  storeLocal(loadLocal(t)+w, t);

  return kernel();
}
```

Fig. 4. Updating I-values using lane local memories in BlueVec-II. The same task requires over 30 lines of code using the BlueVec vector core [3].

that the concatenation of ith elements of each represents the window at position i. In BlueVec-II this is achieved using the construct `shiftIn(x,s)`, which shifts stream s right by one position, inserting x at the front. We also use the construct `streamOf(x)` to obtain a stream with value x at every position. Other sliding window algorithms, such as 2D convolution and median filtering, are implemented similarly.

An interesting aspect of this example is that C++ is being used as a meta-language to generate vector code on the scalar host, e.g. the `for` statement is executed and unrolled before the kernel is compiled and passed to BlueVec-II. The example also illustrates the difference between compile-time and run-time kernel parameters: whereas values for the parameters `filter` and `ntaps` are supplied when the kernel is compiled, the values of `input` and `output` are supplied on every invocation. The `filter` could be moved to a run-time parameter too, with the overhead of having to pass the N taps on each invocation.

Lane local memories. Just as an FPGA has finite resources, BlueVec-II has a finite number of parallel execution units – referred to as *vector lanes*. As well as an ALU, each vector lane contains a low-latency on-chip *lane local memory* [1]. These memories can be accessed using the following stream-indexed load and store operations.

```
Stream<int> loadLocal(Stream<int> addr);
void storeLocal(Stream<int> data, Stream<int> addr);
```

In these operations, the ith value in the address stream `addr` is used to index the local memory in lane i **mod** *numLanes*. A common use of lane local memories is to implement lookup tables that approximate expensive functions. In this case, although each memory contains the same data table, different elements can be accessed in parallel by different lanes. The next sections explore some other uses lane local memories.

Synaptic updates in neural simulation. *I-value accumulation* is a key process in the simulation of spiking neural networks [3]. In a typical biological network, each neuron is connected to thousands of target neurons. When a neuron spikes, some amount of current passes to each target, depending on the weight of the connection. If these targets and connection weights are stored in arrays `targets` and `weights`, the input-current (I-value) of each target is updated as follows.

```
for (int i = 0; i < numTargets; i++)
  ivalues[targets[i]] += weights[i]
```

Figure 4 shows a BlueVec-II kernel to perform this computation, with I-values. It assumes that the `targets` array has been arranged so that the ith element refers to a neuron n such that i **mod** *numLanes* is equal to n **mod** *numLanes*. This means that I-values are distributed evenly over the lane local memories, and can be updated in parallel.

```
Kernel resetSum() {
  storeLocal(0, streamOf(0));
  return kernel();
}

Kernel mulAccum() {
  Param<int> factor;
  Param<int*> row;

  Stream<int> r = load(row);
  Stream<int> sum = loadLocal(0);
  sum = sum + factor*r;
  storeLocal(0, sum);
  return kernel();
}

Kernel storeSum() {
  Param<int*> output;
  store(loadLocal(0), output);
  return kernel();
}
```

Fig. 5. Three BlueVec-II kernels for large matrix multiplication.

Matrix multiplication. Figure 5 shows three BlueVec-II kernels to implement matrix multiplication, another common application of soft vector processors from the literature [2], [6]. These kernels use the following overloaded variants of the `loadLocal` and `storeLocal` functions.

```
Stream<int> loadLocal(int addr);
void storeLocal(Stream<int> data, int addr);
```

Instead of taking a stream of addresses, these variants take the first address of a contiguously aligned stream. The three kernels use these instructions to reset, accumulate, and store each row of the result matrix. Assuming the kernels have been compiled and bound to identifiers `resetKernel`, `accumKernel`, and `storeKernel` respectively, the following NIOS-II code multiplies two `N` $\times$ `N` matrices `A` and `B` to produce matrix `C`.

```
for (int y = 0; y < N; y++) {
  call(N, resetKernel);
  for (int x = 0; x < N; x++)
    call(N, accumKernel, A[y*N+x], &B[x*N]);
  call(N, storeKernel, &C[y*N]);
}
```

Motion estimation. As a final example of an application often used in the evaluation of soft vector processors, Figure 6 shows a BlueVec-II kernel implementing *motion estimation*. The task is to compute the sum-of-absolute-differences (SAD) between a given 16×16 block and every sub-block of a given search region. Although just another sliding-window algorithm, it is interesting because of the size of the window. For large windows, such as a 16×16 block, parallel lanes operate on highly-shared data and it becomes increasingly important to cache blocks of data in on-chip memory. We achieve this by using lane local memories to store the search region. The top-left co-ordinate of every sub-block of this region is passed to the kernel via the `positions` parameter, and the SAD at each position is written to memory beginning at the address pointed to by the `results` parameter. The function `abs`, to compute the magnitude $|x|$ of a given value x, is defined as follows.

```
Stream<int> abs(Stream<int> x)
  { setCond(x < 0); return cond(-x, x); }
```

The function `setCond` is used to set a condition register inside BlueVec-II, and `cond` uses this register to determine whether to return its first argument or its second.

```
Kernel motionEstimation(int width)
{
 Param<int*> positions, results;
 Param<int> block[16][16];

 Stream<int> topLeft = load(positions);
 Stream<int> sad = streamOf(0);

 for (int y = 0; y < 16; y++)
   for (int x = 0; x < 16; x++) {
    Stream<int> addr = topLeft+y*width+x;
    sad += abs(block[y][x] - loadLocal(addr));
   }

 store(sad, results);
 return kernel();
}
```

Fig. 6. A BlueVec-II kernel for motion estimation. The task is to compute the sum-of-absolute differences between a given 16×16 block and every sub-block of a search region of a given width.

III. Evaluation and Conclusion

BlueVec-II's high-level programming interface has been achieved without compromising good performance. For the DSP benchmarks presented in Section II, Table I shows that a 32-lane BlueVec-II core is competative with the existing 32-lane VEGAS core [6] on run-time, though falls slightly behind the recent 32-lane MXP core [10]. In terms of resources, a 32-lane BlueVec-II consumes 18k ALM logic blocks (20% of a Stratix IV 230) and 190 M9K BRAMs (15%) – fewer logic blocks than both VEGAS (37k ALMs) and MXP (46k ALMs).

We now apply a 32-core, 32-lane BlueVec-II arrangement (1024 lanes in total), running on a cluster of 16 DE4-230 FPGA boards, to the following applications.

1. Simulation of 64k *Izhikevich* neurons per core with 1k connections per neuron (targets spread over 3 cores) for 1s of activity with a 1ms time-step and a 10Hz firing rate.
2. A *back propagation* training run applied to 277 faces, each 120x128 pixels, for 50 epochs, with a hidden layer of one perceptron per core. Our implementation is based on [8].

Figure 7 shows the 16-FPGA system scales well. For reference, results from a 32 hyper-thread Xeon server (E5-2680) are also shown. The Xeon versions have not yet been adapted to use AVX extensions, though it is unclear if this would affect scalability. Having observed the DDR2 bandwidth utilisation at 51% and 46% for the BlueVec-II versions of these applications, we are confident we are getting good performance from our FPGAs. When data-sets are large and difficult to cache, a soft vector core can be sufficient to saturate external memory.

Related work. Recently, support for targeting the VENICE soft vector processor [9] has been added to the Microsoft Accelerator compiler, which aims to map high-level data-parallel algorithms onto a range of target platforms, including GPGPUs and multi-cores. Although the idea is certainly very attractive, the authors report that the independent developments of processor and compiler have led to a mismatch in which a number of key VENICE operations are not expressible in Accelerator and, vice-versa, some Accelerator operations are not supported by VENICE. This problem is compounded by the fact that Accelerator is a licensed commercial product with a closed-source compiler front-end, hence cannot be modified.

TABLE I. Run-time performance on some DSP benchmarks.

Application	Size of data set	BlueVec-II (μs)	VEGAS (μs)	MXP (μs)
fir (16 tap)	4096	43	47	26
matmul	1024×1024	$976,000$	$720,000$	$284,400$
motest	48×48	240	247	79
filt3x3	320×240	793	753	627

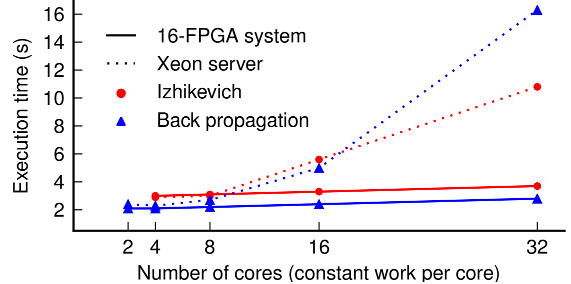


Fig. 7. BlueVec-II on a multi-FPGA versus a multi-core Xeon server.

Conclusion. The development of a high-level programming interface for a soft vector processor can be aided by *codesigning* that processor *together with* its compiler. We have seen that some programming abstractions, such as the generalisation of fixed-sized vectors to arbitrary-sized streams, can be supported directly in hardware, with an efficient and straightforward implemention. We have also seen that the effect of architectural choices, such as decoupling the vector unit from its scalar host, can have far-reaching consequences on the compiler. By separating vector code from the scalar instruction stream, we could easily implement vector code compilation passes, such as register allocation, independently of the compiler for the scalar host language. All this resulted in a custom vector programming API and compiler, implemented as a lightweight C++ library, that is easily adaptable by application developers.

The design presented in this paper allows for high-level descriptions of a range of vector processing tasks with good performance. Using it, we have demonstrated the potential of FPGA clusters to implement scalable parallel applications, and have observed that for some applications near-optimal performance can be achieved using distributed vector cores programmed in software. This leaves us with little doubt about the potential of soft vector processors to support rapid FPGA development. Techniques for developing high-level programming interfaces are a key step towards realising this potential.
(This work is sponsored by EPSRC grant EP/G015783/1.)

References

[1] J. Yu, C. Eagleston, C. H. Chou, M. Perreault and G. Lemieux, *Vector Processing As a Soft Processor Accelerator*, ACM TRETS, 2009.
[2] A. Severance and G. Lemieux, *Embedded Supercomputing in FPGAs with the VectorBlox MXP Matrix Processor*. CODES 2013.
[3] M. Naylor, P. J. Fox, A. T. Markettos, S. W. Moore, *Managing the FPGA memory wall: Custom computing or vector processing?*, FPL'13.
[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, Morgan Kaufmann.
[5] P. Yiannacouras, J. G. Steffan, and J. Rose, *Vespa: Portable, scalable, and flexible fpga-based vector processors*, In CASES'08.
[6] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. Lemieux, *VEGAS: Soft vector processor with scratchpad memory*. In FPGA'11.
[7] J. Teich, *Hardware/Software Codesign: The Past, the Present, and Predicting the Future*, Proceedings of the IEEE, 2012.
[8] T. M. Mitchell, J. Shufelt, http://www.cs.cmu.edu/~tom/faces.html.
[9] Z. Liu, A. Severance, G. Lemieux, S. Singh, *Accelerator Compiler for the VENICE Vector Processor*, FPGA'12.
[10] G. Lemieux, [private communication].