MANAGING THE FPGA MEMORY WALL: CUSTOM COMPUTING OR VECTOR PROCESSING?

Matthew Naylor, Paul J Fox, A Theodore Markettos and Simon W Moore

Computer Laboratory
University of Cambridge
Cambridge, United Kingdom
{matthew.naylor, paul.fox, theo.markettos, simon.moore}@cl.cam.ac.uk

ABSTRACT

Managing the memory wall is critical for massively parallel FPGA applications where data-sets are large and external memory must be used. We demonstrate that a soft vector processor can efficiently stream data from external memory whilst running computation in parallel. A non-trivial neural computation case study illustrates that multi-core vector processing coupled with careful layout of data structures performs similarly to an elaborate full-custom memory controller and execution pipeline. The vector processing version was far simpler to code so we encourage others to consider vector machines before contemplating a full-custom architecture on FPGA.

1. INTRODUCTION

A great deal of research on efficiently mapping algorithms onto FPGAs produces custom computation pipelines, which aim to exploit the massively parallel computation resources available on today's FPGAs. Constructing complex custom pipelines is time consuming. Suitable abstractions such as C-to-gates improve productivity, but often at the expense of performance. Vector processing can be an attractive alternative to C-to-gates [1], yielding good performance with the convenience of software programming and debugging.

Applications demanding massive compute often require large data-sets. The need to stream data from memory external to an FPGA becomes a bottleneck for a broad class of applications whose pinnacle of performance is reached when external memory bandwidth is saturated with useful data transfer, not when the FPGA compute resources are maximally used. This is known as the "memory wall", and is an increasing problem for both ASICs and FPGAs [2], where compute resources are more plentiful than external memory bandwidth. In this paper, we focus on this class of application with an in-depth case study of neural computation which has demanding data interdependencies and large data-sets that need to be held in external memory.

Our previous work on custom neural computation pipelines for FPGAs resulted in a high performance design described using Bluespec HDL [3], capable of efficiently streaming neuron and synapse parameters from DDR2 memory to achieve real-time performance with 64k neurons and 64M synapses per Altera Stratix IV 230 FPGA [4].

This custom pipeline implementation took around three man-years to complete, and resulted in us having a deep understanding of the Izhikevich spiking neuron model. While highly parameterised, this implementation is still rather inflexible (e.g. if the neuron model has to be changed) and is therefore of less utility to neuroscientists (our prospective customers) than existing software-based neural computation systems. However, this work did identify that given some modest parallel compute, external memory bandwidth becomes a performance bottleneck for this application. As a consequence, we explore vector processing for neural computation, with a particular focus on making efficient use of external memory bandwidth using burst transfers.

Section 2 presents BlueVec, a vector co-processor for an Altera NIOS II. Section 3 uses neural computation as a case study to compare and contrast custom computing, vector processing, and multi-core implementations, and the results of this case study are given in Section 4. Section 5 provides conclusions and considers their implications on future research directions.

2. BLUEVEC ARCHITECTURE

Recent work at the University of British Colombia has led to a series of soft vector processors [2, 5, 6, 7] allowing the rapid development of high-performance program accelerators on FPGAs. Of these, VIPERS [2] is perhaps the most interesting for neural computation applications: it supports *lane-local memories* that can be addressed independently and in parallel using a vector of addresses and, as we discover, this feature is ideal for parallel distribution of synaptic updates to neurons scattered throughout memory. Unfortunately VIPERS lacks an important feature for these ap-

plications: *burst memory access* for high-performance streaming of data from external memory. While the successors to VIPERS [5, 6, 7] have made great progress towards optimising external memory bandwidth efficiency, they all omit lane-local memories. Therefore we develop our own soft vector processor – *BlueVec* – to meet both requirements.

Rather than design a whole new processor from scratch, we implement BlueVec as a *custom instruction set extension* to the NIOS II, an existing 32 bit scalar processor developed by Altera which comes with a rich software development toolchain including a C/C++ compiler.

2.1. Vector width

BlueVec is a minimalist vector co-processor – written in around 1k lines of Bluespec HDL in around 2 man-months – with two external interfaces: (1) a *custom instruction slave* interface for connection to a NIOS II, and (2) a *memory mapped master* interface for connection to external memory. Assuming a NIOS II clock frequency of 200 MHz, and a DDR2 external memory transferring 64 bits of data on both edges of a 400 MHz clock, the maximum data transfer rate between processor and memory is 256 bits per NIOS II clock cycle. This motivates processing vectors of 256 bits per clock cycle, which can be treated as either:

- 8 lanes of 32 bit words (`W` instructions) *or*
- 16 lanes of 16 bit half-words (`H` instructions) *or*
- 32 lanes of 8 bit bytes (`B` instructions).

2.2. Register file and instruction set

Given that a NIOS II custom instruction is defined as containing three 5 bit register operands, the obvious design choice for BlueVec is a three-operand vector instruction set with a 32-element register file. We take this option, but there are alternatives, e.g. a large scratchpad in place of a register file with support for long vectors, which would allow a greater number of vector lanes [6].

An illustrative portion of the BlueVec instruction set is shown in Figure 1. Note the use of *v* and *s* prefixes to distinguish BlueVec vector registers and NIOS II scalar registers respectively. All BlueVec instructions are implemented as C macros which expand to inline assembly code. Hence any valid C expression or variable can be used in place of a scalar register, but vector registers must be constants in the range `v0`...`v31`. The following sections discuss parts of the BlueVec instruction set in more detail.

2.3. External memory

Vectors can be loaded from external memory using the instruction

`Load(`*vDest*`,`*sAddr*`,`*burstLength*`)`

When executed, a *burstLength*-element sequence of 256 bit vectors beginning at address *sAddr* is read into registers

$$vDest, (vDest+1), \ldots, (vDest+burstLength-1).$$

However, the register file is *not modified* until a `Commit` instruction is issued. This makes the latent `Load` instruction *non-blocking* so it can be issued well before its result is actually needed, where need is signified by a blocking `Commit`. For example, data for the next iteration of a loop can be fetched while data for the current iteration is processed. In principle, `Commit` instructions can be inferred in hardware using register scoreboarding, but we have opted for an explicit design to keep the hardware simple.

The corresponding store instruction is non-blocking and, at the time of writing, does not support bursts:

`Store(`*vSrc*`,`*sAddr*`)`.

2.4. Lane-local memories

Each half-word vector lane has its own local Block RAM giving a a vector form of scratch pad memory that is accessible by a vector of 16 addresses. The instruction

`LoadLocalH(`*vDest*`,`*vAddr*`)`

loads $LOCAL_i[vAddr[i]]$ into $vDest[i]$ for each of the lane local memories $LOCAL_i$ where $i \in \{0..15\}$. The size of each $LOCAL_i$ is a BlueVec design parameter that can be altered on a per-application basis. The corresponding store instruction is:

`StoreLocalH(`*vSrc*`,`*vAddr*`)`

Only half-word versions of these instructions are supported since 16 bit addresses are more appropriate for medium-sized Block RAMs than 8 or 32 bits.

2.5. Pipelining

In order to achieve a clock frequency above 200 MHz, i.e. not inhibit the NIOS II clock frequency, BlueVec uses a 3 stage pipeline:

- *F*: operand fetch (from register file)
- *E*: execute instruction
- *W*: writeback result (to register file)

Most instructions execute in a single cycle and provide a result that can be used immediately. This is made possible by *register forwarding*: the result and destination register of stages *E* and *W* are inspected by stage *F* and used to override, if necessary, the values fetched from the register file.

There are three instructions which do not fully complete in a single cycle:

$$
\begin{aligned}
\texttt{Mul[B|H|W]}(vDest, vSrcA, vSrcB) &= vDest[i] \leftarrow vSrcA[i] \times vSrcB[i] \\
\texttt{Cmp[B|H|W]}(vDest, vSrcA, vSrcB) &= vDest[i] \leftarrow \textbf{if } vSrcA[i] \leq vSrcB[i] \textbf{ then } 1 \textbf{ else } 0 \\
\texttt{Cond[B|H|W]}(vDest, vElse, vCond) &= vDest[i] \leftarrow \textbf{if } vCond[i] \textbf{ then } vDest[i] \textbf{ else } vElse[i] \\
\texttt{Index[B|H|W]}(sDest, vSrc, sIndex) &= sDest \leftarrow vSrc[sIndex] \\
\texttt{Set[B|H|W]}(vDest, sMask, sSrc) &= vDest[i] \leftarrow \textbf{if } sMask \,\&\, 2^i \textbf{ then } sSrc \textbf{ else } vDest[i] \\
\texttt{LoadLocalH}(vDest, vAddr) &= vDest[i] \leftarrow LOCAL_i[vAddr[i]] \\
\texttt{Load}(vDest, sAddr, burstLength) &= vDest, \ldots, (vDest + burstLength - 1) \leftarrow \\
&\qquad MEM[sAddr], \ldots, MEM[sAddr + burstLength - 1] \\
\texttt{HtoW}(vDest, vSrc, upper) &= vDest[i] \leftarrow \textbf{if } upper \textbf{ then } vSrc[2 \times i] \textbf{ else } vSrc[i]
\end{aligned}
$$

Fig. 1. An illustrative portion of the BlueVec instruction set. Register names prefixed with *v* denote 256 bit vectors and those prefixed with *s* denote 32 bit scalars. Terms of the form $v[i]$ denote the i^{th} value in vector *v*. Index *i* ranges from 0 to 31, 15, and 7 for byte (`B`), half-word (`H`), and full-word (`W`) instructions respectively. $LOCAL_i$ denotes lane local memory *i*, and *MEM* denotes external memory with a 256 bit data bus. The `HtoW` instruction converts half-word vectors to word vectors. Other BlueVec instructions include vector addition, subtraction, shifting, and word to half-word conversion.

- `LoadLocalH` completes in one cycle but the caller must wait one further cycle before reading the result: register forwarding is not possible at stage *E* since the output of Block RAM is not yet available. Waiting can be achieved using `NoOp` or any other instruction that does not read the destination register.
- `Mul` completes in two cycles but the caller must wait two further cycles before reading the result. This is due to a 3 cycle latency on FPGA multiplier blocks clocked at over 200 MHz.
- `Index` takes 3 cycles to complete since it must pass through the whole pipeline before a result can be returned to the NIOS II.

2.6. Record/Playback

We observed that the NIOS II is unable to issue custom vector instructions at the maximum possible rate of one per cycle. As a workaround, we introduced a record/playback facility which allows sequences of instructions to be written to a local instruction memory inside BlueVec, and played back at the maximum rate by issuing a single NIOS II instruction. To illustrate,

```
Record(begin);
  // Sequence of vector instructions
Record(end);
```

records the given instruction sequence which can be issued by calling `Playback(begin,end)`. A recorded instruction sequence can be viewed as a parameterless sub-routine. Support for parameters would be an obvious improvement for future work (but so far not essential for our applications).

3. NEUROCOMPUTING CASE STUDY

As a case study we compare our custom neural computation pipeline [4] for the Izhikevich spiking neuron model [8] to an implementation of the same algorithm using BlueVec. Here, we focus on the critical inner loop: *I-value accumulation*. Details of the full algorithm can be found in [9].

3.1. *I*-value accumulation

A spiking neural network consists of neurons connected by synaptic connections. In a typical biologically-plausible network, each neuron has synaptic connections to around 10^3 other neurons. Each synaptic connection has an associated *delay* and a *weight*, which signifies the strength of the connection. Collectively the combination of target neuron, delay and weight are known as a *synaptic update*. When a neuron spikes each synaptic update needs to be delayed and then summed with other synaptic updates targeted at the same neuron to produce a total input current (termed as an *I-value*) for each neuron. We refer to this process of summing synaptic updates as *I-value accumulation*.

If the *I*-value of neuron `n` is denoted `ivalues[n]`, and its target connections and associated weights are stored in arrays `targets` and `weights` respectively, then the *I*-value accumulation process required if neuron `n` spikes is:

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

In both the custom pipeline and BlueVec implementations, each array has elements which are 16 bits in size. Since the number of *I*-values is equal to the number of neurons, there is ample capacity for `ivalues` to be stored in on-FPGA Block RAM, which has a total size of 2MB on a Stratix IV

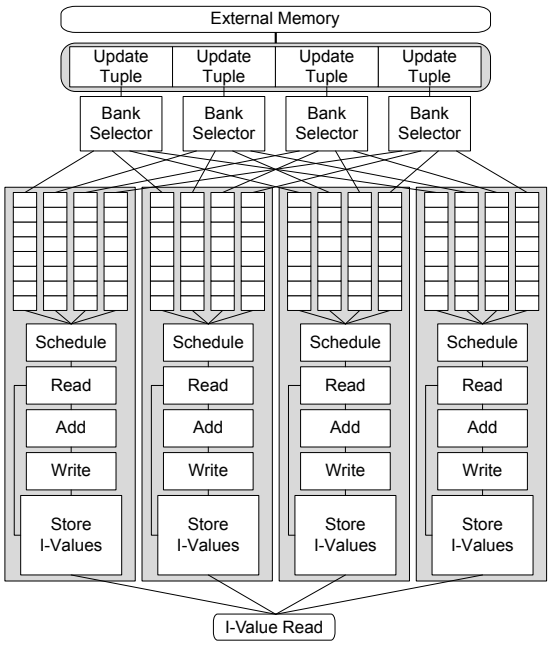

Fig. 2. Custom pipeline implementation of the I-value accumulation phase. Four banks are shown for clarity – there are actually eight banks.

230. However, as the number of synaptic connections is typically $10^3\times$ the number of neurons, the `targets` and `weights` arrays for each neuron must be stored in external memory if we are to simulate many thousands of neurons.

3.2. Implementation A: custom pipeline

Assuming the `targets` and `weights` arrays are interleaved in memory to give an array of (target, weight) pairs called *update tuples*, our custom pipeline implementation of the I-value accumulation loop (known as the accumulator block in previous work) is shown in Figure 2. The on-FPGA Block RAM used to store the I-values is partitioned into 8 banks, since 8 update tuples can be obtained (in a single 256 bit DDR2 memory transfer) per clock cycle when efficient burst reads are used. Each bank is surrounded by a pipeline which processes update tuples. Each update tuple in an incoming word is then allocated to the bank that holds the I-value for the target neuron, with FIFO queues and arbiters used to allow multiple update tuples in the same 256 bit word to target the same bank.

While the FIFO queues do provide some tolerance of uneven load between banks, in practice it was found that highest performance was achieved when update tuples were arranged in 256 bit words such that they are effectively statically scheduled, with the update tuple in position x of a 256 bit word always targeting a neuron whose identifier modulo 8 is equal to x (or else being empty, denoted by zero weight).

```
// Read 16 targets and weights into
// vector registers v8 and v16
Load(v8, targets, 1);
Load(v16, weights, 1);
for (i = 0; i < numTargets; i+=16) {
   Commit;

   // Pre-fetch values for next
   // iteration in background
   Load(v8, targets+i+16, 1);
   Load(v16, weights+i+16, 1);

   // Update 16 I-values
   LoadLocalH(v0, v8);
   NoOp;
   AddH(v0, v0, v16);
   StoreLocalH(v0, v8);
}
```

Fig. 3. BlueVec implementation of the I-value accumulation phase (without bursts and record/playback).

As a result of this static scheduling, the complex accumulator block effectively becomes a vector of independent blocks, and hence the function it performs is amenable to implementation using the BlueVec vector processor.

3.3. Implementation B: vector processing

Figure 3 shows a BlueVec implementation of I-value accumulation. While this vectorised loop gives good speed-up over the simple scalar loop, it is markedly improved by changing the *burstLength* argument of each `Load` instruction from 1 to 8. Consequently, the loop increment changes from 16 to 128 and each of the four instructions at the end of the loop is performed eight times as follows.

```
LoadLocalH(v0, v8); NoOp;
AddH(v0, v0, v16); StoreLocalH(v0, v8);
...
LoadLocalH(v0, v15); NoOp;
AddH(v0, v0, v23); StoreLocalH(v0, v15);
```

This results in a very long sequence of vector instructions that can be efficiently issued at a rate of one per clock cycle using the record/playback feature.

4. RESULTS

We now discuss the performance and productivity of our custom computing and vector processing approaches to neural computation. Each approach was implemented on a Terasic DE4 evaluation board with a Stratix IV 230 FPGA, using a single DDR2 external memory bank.

Table 1. Run time and % of total for phases of Izhikevich neuron model with and without a BlueVec co-processor.

	NIOS II		+ BlueVec	
	Time / s	%	Time / s	%
I-values	57.2	72	1.4	36
Neuron updates	17.2	22	1.7	44
Spike delay buffer	1.6	2	0.5	13
Total	79.0		3.9	

4.1. Single-core performance

Table 1 shows the times taken by our scalar (without BlueVec) and vector (with BlueVec) Izhikevich neural computation systems to compute 1 s of neural activity in a benchmark network consisting of 64k neurons with 64M synaptic connections [9]. Note that the scalar version has been optimised for performance: *I*-values are stored in a large Block RAM, and the NIOS II has a 4kB data cache with 256 bit cache lines that can be filled by single DDR2 memory transfers. Our custom pipeline implementation operates in real-time, around 80× faster than the scalar version, but only 4× faster than the vector version.

The performance profiles in Table 1 are split into three phases. The time for the *I*-value accumulation stage (discussed in Section 3) is reduced by 40× using vector processing. The neuron update and spike delay phases have not been discussed here, but details can be found in [9].

We observed that DDR2 bandwidth utilisation for the vector version is 16%. The fact that a single BlueVec is not able to saturate memory bandwidth is a consequence of an imbalance between memory access and compute. For example, while the states of 16 neurons can be fetched in 6 memory transfers, 44 instructions are required to process them. This motivates increasing vector sizes and hence the number of vector lanes in future work. But in the meantime, we explore the use of multiple cores to saturate memory bandwidth. Given that a single BlueVec core consumes 7k Altera ALMs (7% of total) and 96 multiplier blocks (7% of total), there is plenty of scope for multiple BlueVec cores.

4.2. Multi-core performance

Neural computation is a highly-parallel task, and our benchmark neural network is easily split into smaller networks that can be processed in parallel with negligible communication. Table 2 shows the performance improvement obtained with multiple NIOS II and BlueVec cores accessing shared DDR2 memory and distributed Block RAMs for stack and instruction memory. The cores are connected in a star network, with a master core connected bidirectionally to all slave cores. Notably, the quad-core BlueVec configuration

Table 2. Run time, logic and bandwidth utilisation for a multi-threaded Izhikevich neuron simulator with varying number of cores and vector co-processors.

NIOS II cores	BlueVec cores	Time (s)	Logic (%)	Bandwidth DDR2 (%)
2	0	40.2	14	1.9
4	0	20.9	19	3.8
8	0	10.8	30	7.5
16	0	6.1	53	13.1
32	0	5.0	100	17.6
2	2	2.2	26	30.5
4	4	1.3	49	51.0

Table 3. Lines of code for Izhikevich and leaky integrate-and-fire neuron models.

Model	Implementation	Lines
Izhikevich	Single-threaded	193
	Single-threaded and Vectorised	417
	Multi-threaded	265
	Multi-threaded and Vectorised	511
	Custom pipeline	2700
LIF	Single-threaded	324
	Single-threaded and Vectorised	496

gives performance that is well within a factor of 2 of our custom pipeline. Interestingly, the two have very similar logic utilisation. While it would also be possible to have multiple custom pipeline instances sharing a single DDR2 memory bank, it would not be worthwhile since a single instance already exhibits very high DDR2 bandwidth utilisation.

Both the scalar and vector software versions benefit significantly from multiple cores. However Table 2 shows poor performance scaling from 16 to 32 scalar cores: as the number of cores increases, the spatial locality of memory accesses decreases, and the performance of DDR2 memory drops. Even if performance scaled perfectly, the number of scalar cores required to match the quad-core BlueVec implementation would consume 4× more FPGA logic.

4.3. Productivity

Table 3 shows the numbers of lines of code in our neural computation systems. The almost 3k lines needed for the custom pipeline is striking, indicating the extra detail a hardware designer must express. In fact, this line count would be even higher if it included general-purpose libraries developed in-house. Hardware development cycles can be slow for other reasons too, such as long synthesis times, trial-and-error refinements needed to meet tight timing constraints, and a lack of convenient I/O mechanisms for debugging.

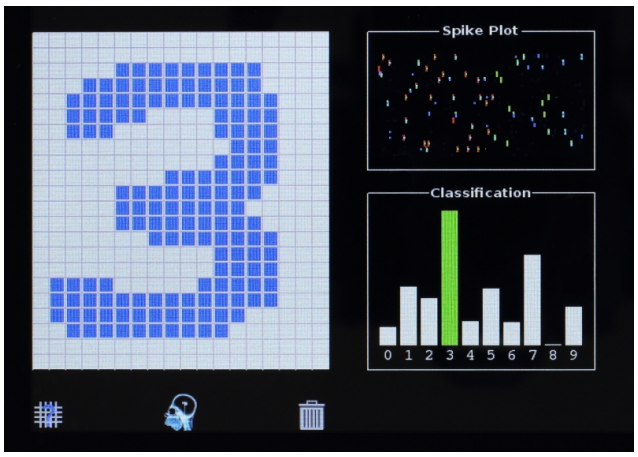

Fig. 4. An implementation of the Nengo digit-recognition model on a DE4 FPGA board with touch-screen.

The convenience of a software-based approach has allowed us to develop other efficient neural computation systems in a very short time. Figure 4 shows a screenshot of our leaky integrate-and-fire (LIF) system running the Nengo [10] model for digit recognition on a DE4 FPGA with touch-screen. Using a single vector processor, we were able to achieve a 20× speed-up over a scalar implementation with just two days work. We do not believe that implementing a custom pipeline LIF system in this time-frame is possible, even with re-use of components from the Izhikevich system.

5. CONCLUSION

Managing the memory wall is critical for many data intensive applications on FPGA. Achieving good performance with modern DDR2/DDR3 memories requires careful use of burst accesses, which is well suited to vector memory operations. While custom compute pipelines and their associated memory controllers can achieve high performance, it is still critical to perform vector/burst accesses. When memory becomes the bottleneck, there is scope to use more general purpose vector compute and still yield similar performance, possibly at the expense of area. Of course reconfigurable vector arithmetic units can cater for application specific demands, which helps to maintain the benefits of FPGA over other non-reconfigurable compute.

We have demonstrated that a quad-core BlueVec (NIOS II + vector) machine has performance that falls within a factor of 2 of the custom pipeline for our neural computation application, and has similar logic usage. The Bluespec code for BlueVec together with vectorised C code proved to be much more compact and easy to develop than the custom pipeline despite using a high-level language (Bluespec).

Based on our experiences presented in this paper, we encourage others mapping algorithms onto FPGA to consider external bandwidth requirements first. If the memory wall is near, then consider vector-style computation since it is relatively easy to design and can yield high performance. Moreover, the process of analysing the memory footprint of data structures and thinking about vectorisation will not only be useful for a vector implementation but also for a custom pipeline implementation should greater compute be needed.

6. ACKNOWLEDGEMENTS

Many thanks are due to our colleagues Prof. S.B. Furber (Uni. of Manchester), Prof. A.D. Brown (Uni. of Southampton) and Mr. S. Marsh (Uni. of Cambridge) for collaborating on neural simulation techniques. The UK EPSRC provided much of the funding through grant EP/G015783/1.

7. REFERENCES

[1] J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux, "Vector processing as a soft processor accelerator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, pp. 12:1–12:34, 2009.

[2] S. A. McKee, "Reflections on the memory wall," in *Computing Frontiers, 1st Conference on*, 2004, p. 162.

[3] R. S. Nikhil and K. R. Czeck, *BSV by Example*. CreateSpace, 2010.

[4] S. W. Moore, P. J. Fox, S. J. T. Marsh, A. T. Markettos, and A. Mujumdar, "Bluehive - a field-programable custom computing machine for extreme-scale real-time neural network simulation," in *Field-Programmable Custom Computing Machines, 20th Annual IEEE Symposium on*, 2012, pp. 133–140.

[5] P. Yiannacouras, J. G. Steffan, and J. Rose, "VESPA: Portable, scalable, and flexible FPGA-based vector processors," in *Compilers, Architectures and Synthesis for Embedded Systems, 2008 International Conference on*, pp. 61–70.

[6] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux, "VEGAS: Soft vector processor with scratchpad memory," in *Field Programmable Gate Arrays, 19th ACM/SIGDA International Symposium on*, 2011, pp. 15–24.

[7] A. Severance and G. Lemieux, "VENICE: A compact vector processor for FPGA applications," in *Field-Programmable Technology (FPT), 2012 International Conference on*, pp. 261–268.

[8] E. Izhikevich, "Simple model of spiking neurons," *Neural Networks, IEEE Transactions on*, vol. 14, no. 6, pp. 1569 – 1572, Nov. 2003.

[9] P. J. Fox, "Massively parallel neural computation," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-830, Mar. 2013. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-830.pdf

[10] C. Eliasmith, T. C. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, and D. Rasmussen, "A large-scale model of the functioning brain," *Science*, vol. 338, no. 6111, pp. 1202–1205, 2012.