Continuous Real-World Inputs Can Open Up Alternative Accelerator Designs

Bilel Belhadj[†] Antoine Joubert[†] Zheng Li[§] Rodolphe Héliot[†] Olivier Temam[§]
[†]CEA LETI, France [§]INRIA, Saclay, France

ABSTRACT

Motivated by energy constraints, future heterogeneous multi-cores may contain a variety of accelerators, each targeting a subset of the application spectrum. Beyond energy, the growing number of faults steers accelerator research towards fault-tolerant accelerators.

In this article, we investigate a fault-tolerant and energy-efficient accelerator for signal processing applications. We depart from traditional designs by introducing an accelerator which relies on *unary coding*, a concept which is well adapted to the continuous real-world inputs of signal processing applications. Unary coding enables a number of atypical micro-architecture choices which bring down area cost and energy; moreover, unary coding provides graceful output degradation as the amount of transient faults increases.

We introduce a configurable hybrid digital/analog micro-architecture capable of implementing a broad set of signal processing applications based on these concepts, together with a back-end optimizer which takes advantage of the special nature of these applications. For a set of five signal applications, we explore the different design tradeoffs and obtain an accelerator with an area cost of $1.63mm^2$. On average, this accelerator requires only 2.3% of the energy of an Atom-like core to implement similar tasks. We then evaluate the accelerator resilience to transient faults, and its ability to trade accuracy for energy savings.

1. INTRODUCTION

Due to ever stringent technology constraints, especially energy [12] and faults [6], the micro-architecture community has been contemplating increasingly varied approaches [42, 15, 37] for designing architectures which realize different tradeoffs between application flexibility, area cost, energy efficiency and fault tolerance. Finding appropriate accelerators is both an open question and fast becoming one of the key challenges of our community [36].

But in addition to technology constraints, there is a similarly important evolution in the systems these chips are used in, and the applications they are used for [7]. Our community is currently focused on general-purpose computing, but a large array of more or less specialized (potentially high-volume) devices and applications are in increasing demand for performance and sophisticated processing tasks. Examples include voice recognition which has recently become mainstream on smartphones (e.g., Siri on iPhones), an increasing number of cameras which integrate facial expression recognition (e.g., photo is taken when a smile is detected), or even self-driving cars which require fast video and navigation processing (e.g., Google cars), and a host of novel applications stemming from and based upon sensors such as the Microsoft Kinect, etc. These devices and applications may well drive the creation of less flexible but highly efficient micro-architectures in the future. At the very least, they already, or may soon correspond, to applications with high enough volume to justify the introduction of related accelerators in multi-core chips.

A notable common feature of many of these emerging applications is that they continuously process "real-world", and often noisy, low-frequency, input data (e.g., image, video, audio and some of the radio signals), and they perform more or less sophisticated tasks on them. In other words, many of these tasks can be deemed *signal processing* tasks at large, where the signal nature can vary. Digital Signal Processors (DSPs) have been specifically designed to cope with such inputs. Today, while DSPs still retain some specific features (e.g., VLIW, DMA), they have become increasingly similar to full-fledged processors. For instance the TI C6000 [1] has a clock frequency of 1.2GHz, a rich instruction set and a cache hierarchy.

The rationale for this article is that continuous real-world input data processing entails specific properties which can be leveraged to better cope with the combined evolution of technology and applications. We particularly seek low-cost solutions to enable their broad adoption in many devices, enough flexibility to accommodate a broad set of signal processing tasks, high tolerance to transient faults, and low-energy solutions.

The starting point of our approach is to rely on *unary coding* for representing input data and any data circulating within the architecture. Unary coding means that an integer value V is coded with a train of V pulses. This choice may come across as inefficient performance-wise, since most architectures would transmit value V in a single cycle using $log_2(V)$ bits in parallel. However, while real-world input data may be massive (e.g., high-resolution images/videos), its update frequency is typically very low, e.g., 50/60Hz for video, and rarely above 1MHz for most signals, except for high-frequency radio signals. Both the time margin provided by low frequency input signal and by unary coding enable a set of atypical micro-architecture design innovations, with significant benefits in energy and fault tolerance; some of these innovations are nested, one enabling another, as explained thereafter.

The first benefit of unary coding is more progressive tolerance to faults than word-based representations. Errors occurring on data represented using n bits can exhibit high value variations depend-

ISCA'13, Tel-Aviv, Israel.

ing on the magnitude of the bit which suffered the fault (low to high order). With unary coding, losing one pulse induces a unit error variation, so the amplitude of the error increases progressively with the number of faults.

The ability to lose pulses with no significant impact on application behavior brings, in turn, cost and energy benefits. Two of the main architecture benefits are: (1) the switches in the fabric routing network can operate asynchronously, without even resorting to handshaking, (2) the clock tree is smaller than in a usual fabric since fewer elements operate synchronously.

Another major benefit of unary coding is the existence of a powerful, but very cost effective operator, for realizing complex signal processing tasks. Recent research has shown that most elementary signal processing operations can not only be expressed, but also densely implemented, using a Leaky Integrate and Fire (LIF) spiking neuron [40, 29, 10]. A spiking neuron may be slower than a typical digital operator but it realizes more complex operations, such as integration, and the low input signal frequency provides ample time margin. Each operator can be built with just a few such neurons; however, to dispel any possible confusion, there is no *learning* involved, the weights of the synapses are precisely set, i.e., *programmed*, to accomplish the desired function; the neuron is only used as an elementary operator.

Using a spiking neuron enables, in turn, additional micro-architecture optimizations. A spiking neuron can be realized as a digital or analog circuit, but analog versions have been shown to be much denser and more energy efficient [20], so we select that option. However, we avoid two of the main pitfalls of past attempts at designing analog architectures: the difficulty to chain and precisely control analog operators, and the difficulty to program such architectures [17]. For the first issue, we resort to a hybrid digital-analog grid-like architecture where the routing network and control is asynchronous but digital, while only the computational operators at the grid nodes are analog. The programming issue is overcome by creating a library of signal processing operations, themselves based on the unique nature of the elementary hardware operator (spiking neuron).

A signal processing application is then expressed as a graph of such operations and mapped onto the architecture. While such graphs can theoretically require a massive number of connections, unary coding enables again a set of unique back-end compiler optimizations which can further, and drastically, reduce the overall cost of the architecture.

Unary coding and the low input signal frequency enable additional micro-architecture benefits. For instance, our fabric routing network is composed of time-multiplexed one-bit channels instead of multi-bit channels. Another benefit is that real-world input data expressed as pulses does not need to be fetched from memory, it can directly come from sensors, saving again on cost and energy. This is possible because analog real-world data can be converted into pulses more efficiently than into digital words [30], and this conversion can thus be implemented within sensors themselves [32], or as a front-end to the architecture.

In this article, we investigate the design of a micro-architecture based on these concepts, we propose a detailed implementation, we develop a back-end compiler generator to automatically map application graphs onto the architecture, and which plays a significant role in keeping the architecture cost low, and we evaluate the main characteristics of this micro-architecture (area, energy, latency, fault tolerance) on a set of signal processing applications.

This architecture is less flexible than a processor or an FPGA, but far more flexible than an ASIC, and thus capable of harnessing a broad range of applications. The computational unit (analog neuron) of this architecture has been successfully taped out, and the full architecture presented in this paper is in the process of being taped out.

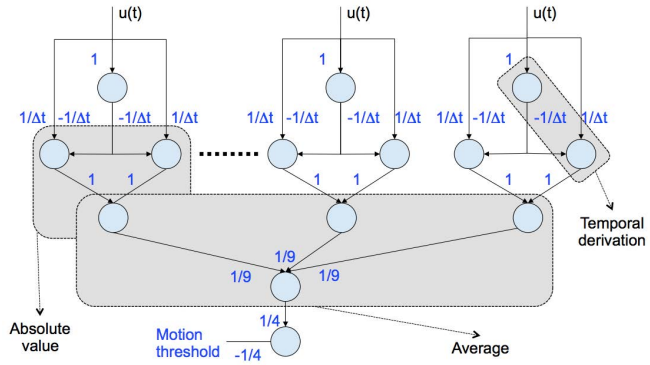


Figure 1: *Neuron-based programming of motion estimation, the main operations are highlighted, and the weights are shown on the edges.*

In Section 2, we recall the basic concepts allowing to program signal processing applications as graphs of operators, and to implement these operators using analog neurons. In Section 3, we present the micro-architecture and how real-world input data characteristics are factored in the design. In Section 4, we discuss the back-end generator, especially which mapping optimizations help keep the architecture cost low. In Section 5, we introduce the methodology and applications. In Section 6, we explore the optimal architecture configuration (area) for our target applications, and evaluate the energy, fault tolerance and bandwidth of our architecture. In Section 7, we present the related work and conclude in Section 8.

2. PROGRAMMING

In this section, we explain the main principles that allow to express a signal processing task as a combination of elementary operators, themselves based upon a unique elementary analog operator (the analog neuron). We leverage recent research on the topic [10], and we briefly recap the main principles below, because they are not yet broadly used in the micro-architecture community. While our community may be more familiar with imperative programs run on traditional cores, it has already delved into similar streaming-oriented computing approaches through programming languages such as StreamIt [39]. While the principles we present below are similar, the program representation is different (application graphs, similar to DFGs, instead of actual instruction-based programs), and the elementary operators are the building blocks of signal processing tasks instead of only elementary arithmetic operators.

2.1 Application Graphs

A broad set of signal processing tasks are actually based on a limited set of distinct and elementary operations: spatial and time derivation, integration, max/min, multiplexing, delay, plus the elementary arithmetic operations (addition, subtraction, multiplication, division) [27]. Assuming a continuous input stream of data, it is possible to build almost any signal processing application as a connected graph of these operators; some specific tasks, such as those involving recursive algorithms or division by a non-constant factor, can be more difficult to implement though (note that division

$$\dot{V}_i(t) = \frac{-V_i(t)}{\tau_i} + \sum_{j=1}^{n} W_{ij}\, s_j(t-\Delta_{ij})$$

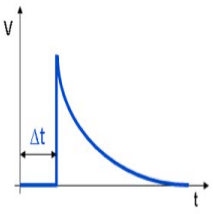


Figure 2: *LIF neuron equation and temporal impulse response.*

by a variable can be implemented as well, but it requires a larger group of neurons [11]).

Consider, for instance, the task of detecting whether motion occurred within an image and by how much, i.e., motion estimation. The graph of operations corresponding to that task is shown in Figure 1; that figure shows two levels of decomposition: an operation-level decomposition represented with grey bounding boxes (highlighting the main types of operations used), and the individual neurons composing these operations. Let us, for now, consider the operation-level decomposition. Motion estimation is a pipeline of the following operations: first, the luminance temporal derivative of each pixel is computed to determine pixel-level variations, the absolute value of each derivative is computed, then the absolute value of the derivatives of all pixels are averaged, and finally, the average is compared against a threshold to determine if a motion occurred within the image.

Similarly, most signal processing tasks can be expressed as directed acyclic graphs of a set of operations. In the next section, we explain how such operations can be implemented using a unique analog operator (the analog neuron).

2.2 Expressing Operations Using Analog Neurons

We first introduce the analog operator, i.e., the analog neuron, and then explain how the different elementary operations involved in signal processing tasks can be expressed using that operator.

We use the Leaky Integrate and Fire neuron (LIF neuron) [16], one of the most commonly used neuron model. A LIF neuron is a powerful analog signal processing operator because it performs two tasks at once: integrating the signal over time (temporal integration) and over the different inputs (spatial integration). The input of such a neuron are *spikes*, which is just another word for "pulses"; from now on, we will use "spike" instead of "pulse". An analog neuron integrates such spikes over time, and after it reaches a certain threshold, it itself produces an output spike (it fires), see Figure 2 (right).

The LIF neuron differential equation is shown in Figure 2, where $V_i(t)$ is the internal potential of neuron i, $s_j(t)$ is the output of neuron j, τ_i is a leakage time constant, W_{ij} is the synaptic weight from neuron j to neuron i, and Δ_{ij} is the synaptic delay, i.e., the time needed for a spike originating from neuron j to reach neuron i. The output $s_i(t)$ of neuron i is equal to 0, unless the voltage $V_i(t)$ becomes greater than a threshold voltage V_{th}, in which case $s_i(t) = 1$ (an output spike). Note that leakage has an important functional role, especially for multiplication. The multplier operator works as a coincidence detector: thanks to (fast) leakage, the neuron spikes only when two incoming spikes occur close enough in time. If the two input spike distributions are independent, then the probability that the multiplier neuron fires an output spike is equal to the product of the input spike probabilities over a given time window, times a constant which depends upon τ_i [35]

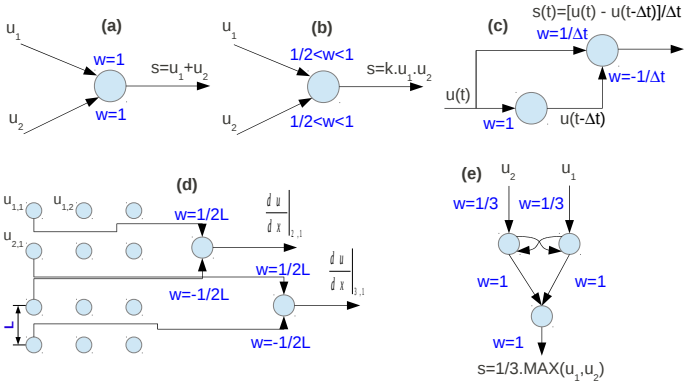


Figure 3: *Implementing some of the main signal processing operations solely using analog neurons: (a) addition, (b) multiplication, (c) temporal derivation, (d) spatial derivation, (e) maximum.*

In Figure 3, we show how to implement some of the main signal processing operations only using analog neurons. A simple operation like multi-input addition is trivially implemented because the neuron realizes a sum of its inputs, see Figure 3(a). A more complex operation like time derivation is implemented by combining two neurons, one performing the addition of two inputs, with respective weights $\frac{1}{\Delta t}$ and $-\frac{1}{\Delta t}$, and one introducing a delay of Δt for one of the inputs, thereby realizing the operation $\frac{(u(t)-u(t-\Delta t))}{\Delta t}$. Both operations are used in the motion estimation application of Figure 1, along with average and absolute value: the average of N values is implemented by summing N values using one neuron and synaptic weights set to $\frac{1}{N}$, while the absolute value of the derivative is obtained by adding the max of the derivative and 0, i.e., $max(\frac{(u(t)-u(t-\Delta t))}{\Delta t}, 0)$, and the max of the opposite of the derivative and 0, i.e., $max(\frac{(u(t)-u(t-\Delta t))}{\Delta t}, 0)$.

2.3 Readout and Spike Loss

Since the input data is coded as a set of spikes, and neurons outputs are spikes, the application output (readout) is also a set of spikes. Counting the number of spikes over a given period of time provides the readout value, which can then be transformed into a classic n-bit digital value and stored in memory or passed to other traditional hardware blocks. The readout *time window* is a key parameter: the more time is allocated to count output spikes, the more precise the output value. So the time window is an energy/accuracy/bandwidth toggle. A short time window speeds up computations and reduces energy at the expense of accuracy. Conversely, a large time window provides a form of redundancy, with no area overhead (unlike traditional circuit redundancy approaches such as ECC) but with a performance and energy overhead; one benefit of that form of redundancy is that it can be adjusted.

If some spikes are lost, either due to faults (noise) or spike collisions in the architecture (later discussed in Section 3), then the readout value is affected albeit in a progressive manner again. So the readout value is only stochastically correct, not deterministic (not always exactly the same for the same input). However, this follows a growing trend of accepting to trade application accuracy for energy or fault-tolerance benefits [8, 13].

3. MICRO-ARCHITECTURE

In this section we present and discuss the main architecture design challenges and opportunities of the proposed approach. Since the goal of the architecture is to implement a broad range of signal

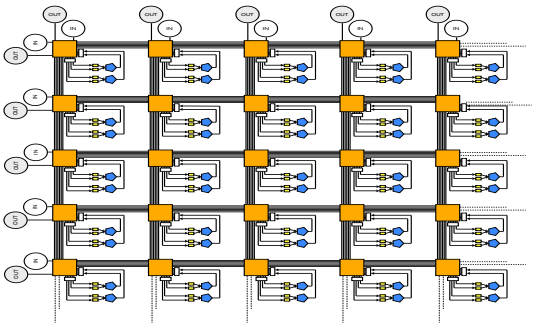


Figure 4: *Tiled hybrid digital-analog architecture.*

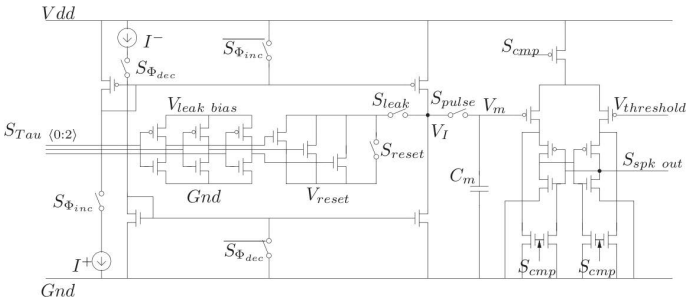


Figure 5: *Analog neuron.*

processing applications, we turn to a generic grid-like structure, see Figure 4, reminiscent of the regular structure of FPGAs, where each node actually contains several neurons and storage for synaptic weights, and nodes are connected through a routing network.

In this architecture, the synapse arrays, the FIFOs after the synapse encoders (see Figure 7), the DAC and the I/Os are clocked, while the switches, routing network, crossbars (see Figure 7) and neurons are not clocked.

The analog/digital interfaces are the DAC and the neuron itself. The DAC converts digital weights into analog inputs for the neuron. The analog neuron output is converted into a digital spike using its threshold function (and thus, it behaves like a 1-bit ADC).

3.1 Analog Operator

We introduce the analog neuron, the sole computational operator used in this architecture. Even though there exist multiple analog neuron designs, most are geared towards fast emulation of biological neural networks [41, 3]. We implemented and taped out an analog neuron at 65nm, shown in Figure 5, capable of harnessing input signals ranging from low (1kHz) to medium (1MHz) frequency, compatible with most applications processing real-world input data. The tape out contains almost all the digital logic present in the tile of the final design presented in Section 3.3, although it is scaled down (especially, the number of synapses is lower). Still, the energy per spike of the analog neuron and the surrounding logic (internal crossbar, synapses, DAC, etc) has been measured at 40pJ/spike (at 65nm). Our analog design is competitive with the recent design of the IBM Cognitive Chip [26], which exhibits 45pJ/spike at 45nm. In the IBM chip, the neuron is digital, and for instance, it contains a 16-bit adder to digitally implement the LIF neuron leakage. While the design of the analog neuron is beyond the scope of this article, we briefly explain its main characteristics and operating principles below, and why its energy is so low.

First, note that the aforementioned frequencies (1kHz to 1MHz) correspond to the rate at which the neuron and I/O circuits can process spikes per second, not the overall *update frequency* of the input data. That frequency can potentially scale to much higher values by leveraging the massive parallelism of the architecture, i.e., an fMHz input signal can be de-multiplexed and processed in parallel using f input circuits. For instance, the motion estimation application processing a black and white SVGA (800x600) image at 100Hz would correspond to a maximum spike frequency of 48MHz ($800 \times 600 \times 100$). That application requires $4 \times N_{pixels} + 3$ neurons for N_{pixels} pixels, so it must process $(4 \times 800 \times 600 + 3) \times 100$ spikes per second. Since the maximum processing rate of a neuron is 1Mspikes per second in our design, we need $\lceil \frac{(4 \times 800 \times 600 + 3) \times 100}{10^6} \rceil = 193$ neurons to process images at a speed compatible with the input rate. This is the number of neurons used for that application in the experiments of Section 6.

An analog neuron has both assets and drawbacks. The main drawback is the need for a large capacitance. The capacitance has a central role as it accumulates spikes, i.e., it performs the addition of inputs, and we have already explained that its leaky property also has a functional role, see Section 2.2. In spite of this drawback, the analog neuron can remain a low-cost device: our analog neuron has an area of $120\mu m^2$ at 65nm. This area accounts for the capacitance size and the 34 transistors. Note that, since the capacitance is implemented using only two metal layers, we can fit most of the transistor logic underneath, so that most of the area actually corresponds to the capacitance.

The analog neuron of Figure 5 operates as follows. When a spike arrives at the input of a neuron, *before* the synapse, it triggers the $S_{\Phi_{inc}}$ switch (see bottom left), which is bringing the current to the capacitance via V_I. The different transistors on the path from $S_{\Phi_{inc}}$ to V_I form a current mirror, which aims at stabilizing the current before injecting it. Now, if the synapse has value V, this spike will be converted into a train of V pulses. These pulses are emulated by switching on/off S_{pulse} V times, inducing V current injections in the capacitance. Since the synaptic weights are coded over 8 bits (1 sign bit and 7-bit absolute value), the maximum absolute value of V is 127. The clock is used to generate a pulse, and in theory, each clock edge could provide a pulse, and thus we could have two pulses per clock cycle; however, in order to avoid overlapping pulses, we use only one of the two clock edges. As a result, generating 127 pulses requires 127 clock cycles, and the minimum clock rate is 127MHz. But we use a 500MHz clock in our design because a pulse should not last more than 1ns due to the characteristics of the capacitance (recall a pulse lasts half a clock cycle, and thus 1ns for a 500MHz clock). However, only the control part of the neuron needs to be clocked at that rate, the rest of the micro-architecture can be clocked at a much lower rate since input spikes typically occur with a maximum frequency of 1MHz. We have physically measured the neuron energy at the very low value of 2pJ per spike (recall a spike is fired after V pulses; we used the maximum value of $V = 127$ for the measurement). Even though the energy consumed in the DAC increases the overall energy per spike to 40pJ, the very low energy per spike consumed by the neuron itself allows to stay competitive with the IBM digital design at 45nm (45pJ) in spite of our lesser 65nm technology node.

The group of transistors on the center left part of the figure implements the variable leakage of the capacitance (which occurs when S_{leak} is closed) necessary to tolerate signal frequencies ranging from 1kHz to 1MHz. After the pulses have been injected, the neuron potential (corresponding to the capacitance charge) is compared against the threshold by closing S_{cmp}, and an output spike is generated on S_{spkout} if the potential is higher than the threshold.

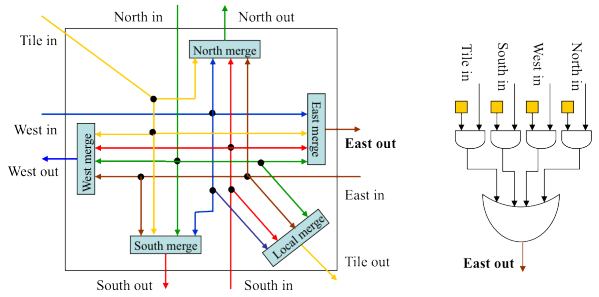


Figure 6: *Router switch for a single channel.*

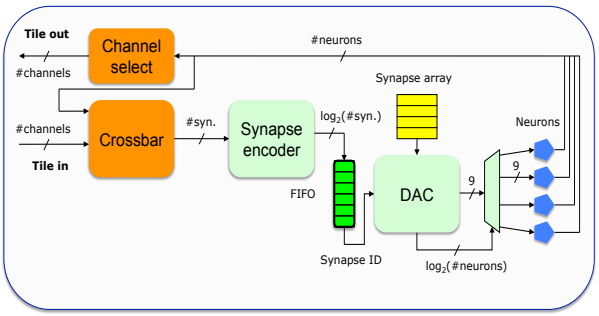


Figure 7: *Tile architecture (4 neurons).*

3.2 Routing Network

Since the information is coded as spikes and our core operator processes analog spikes, it might have seemed natural to transmit *analog* spikes through the network. However, our network routes *digital* spikes for two reasons. First, analog spikes would require frequent repeaters to maintain signal quality, and these repeaters are actually performing analog-digital-analog conversions. So it is just more cost-efficient to convert the analog inputs into digital spikes, digitally route these spikes, and then convert them back to analog spikes again at the level of the operator. Only one digital-analog and one analog-digital conversion are necessary. As a result, the tile includes a digital-analog converter as later explained in Section 3.3. The second motivation for digital spikes is that *routing* analog spikes actually requires to convert them first into digital spikes anyway, as pure analog routing is typically difficult.

Asynchronous switch without handshaking. The software edges of an application graph are statically mapped onto hardware channels, and some of these software edges can actually share the same hardware channels as later explained in Section 4.2. So spikes could potentially compete at a given switch, and in theory, a switch should have buffers and a signaling protocol to temporarily store incoming spikes until they can be processed. However, our switch implementation corresponds to the one of Figure 6: no buffer, not even latches, no handshaking logic (note also that the configuration bits are programmed and remain static during execution). The reason for this design choice lays again in the nature of the processed information. If two spikes compete for the same outgoing channel at exactly the same time, one spike will be lost. This loss of spiking information is tolerable for the reasons outlined in Section 2.3. Moreover, the update frequency of data is low enough (up to 1MHz) that such collisions will be very infrequent.

Finally, collisions are reduced in two ways. One consists in irregularly introducing spikes at the input nodes: instead of sending regularly spaced spikes, we randomly vary the delay between consecutive spikes. As a result, the probability that spikes collide in switches somewhere in the architecture is greatly reduced. The second method, which may completely replace the first method in future technology nodes, requires no action on our part. We actually transform what is normally a hindrance of analog design into an asset: process variability induces minor differences among the transistors and capacitance characteristics of analog neurons so that, even if they receive input data at exactly the same time, the probability that they output their spikes at exactly the same time, and that such spikes collide at a later switch decreases as the variability increases.

Network structure. While the routing network shares several similarities with FPGAs, i.e., sets of 1-bit channels connecting switch boxes [23], see Figure 4, there are two main differences. First, there are no long wires, only neighbor-to-neighbor connections: not only long-distance transmissions have a significant latency and energy overhead [18], but the layered/streaming structure of most signal processing graphs does not warrant such long lines. Second, there are no connect boxes (C-box in FPGA terminology): besides the 4 sets of channels connecting switch boxes, a fifth set of channels links the switch box to the tile, see Figure 6, thereby replacing C-boxes.

3.3 Clustered Tile

While it is possible to implement tiles with a single neuron, it is not the best design tradeoff for at least three reasons. First, it often happens that signal processing applications graphs exhibit *clustered* patterns, i.e., loosely connected graphs of sub-graphs, with each sub-graph being densely connected. Consider, for instance, the temporal derivation + absolute value sub-blocks of motion estimation in Figure 1, which are repeated several times.

Second, inter-tile communications are typically more costly, both in area (increased number of channels, increased switch size) and energy (longer communication distance), than short intra-tile communications. Therefore, it is preferable to create tiles which are *clusters* of neurons, allowing for these short intra-tile communications.

A third motivation for grouping neurons is the relatively large size of the Digital/Analog Converter (DAC) with respect to the small neuron size, i.e., $1300\mu m^2$ vs. $120\mu m^2$ at 65nm in our design. On the other hand, we can leverage again the moderate to low frequency of most signal processing applications to multiplex the DAC between several neurons without hurting application latency.

As a result, the tile structure is the one shown in Figure 7, where a DAC is shared by multiple neurons (12 in our final tile design). In order to leverage intra-tile communications, the crossbars to/from the tile are linked together, allowing to forward the output of neurons directly back to the input of neurons within the same tile. These crossbars are also used to select which neurons are routed to which channels.

3.4 Limited and Closely Located Storage

Currently, most hardware implementations of neural structures rely on centralized SRAM banks for storing synapses, and address them using Address Event Representation (AER) [5], where, in a nutshell, an event is a spike, and each neuron has an address. However, the goal of most of these structures is again the hardware emulation of biological structures with very high connectivity (thousands of synapses per neuron), rather than the efficient implementation of computing tasks. In our case, it is less cost-effective, but more efficient, energy-wise, to distribute the storage for synapses across the different tiles for two reasons: (1) because signal pro-

cessing application graphs show that these synapse arrays can be small (a few synapses per neuron, so just a few tens of synapses per tile) so that each access requires little energy, (2) because the storage is then located close to the computational operator (neuron), the storage-to-operator transmission occurs over very short distances, which is again effective energy-wise [18, 36].

The synaptic array is shown in Figure 7. Synapses are used as follows. The spike is routed by the crossbar connecting the switch to the tile, and it is then fed into the synapse encoder. This encoder simply converts the channel used by the spike into an index for the synapse array, since each channel is uniquely (statically) assigned to a given synapse. The DAC then converts the corresponding synaptic weight value into a spike train for the analog neuron, as explained in Section 3.1. As mentioned in Section 3.1, weights are coded over 8 bits (sign + value).

3.5 I/Os

The peripheral nodes of the architecture contain input and output logic, see Figure 4. These nodes are directly connected to the switches of the routing network, so that the maximum number of inputs and outputs is exactly equal to the number of hardware channels (twice more for corner nodes where two entries of the switch are free).

Inputs. The input logic consists in transforming n-bit data into spike trains. In production mode, this logic could potentially reside at the level of sensors, rather than in the accelerator. For now, we embed it because, if the accelerator is not directly connected to sensors, it is more efficient to implement spike train conversion within the accelerator itself, rather than transmitting spike trains through an external bus or network.

The main particularity of the input logic is to slightly vary the delays between the generated spikes so that they are not regularly spaced in time, with the intent of limiting spike collisions at the level of switches. For that purpose, the input logic includes a Gaussian random number generator. We also implicitly leverage process variability to further vary spike delays, though we have not yet extensively investigated its impact.

Outputs. Unlike the input logic, the readout logic should always reside within the accelerator in order to convert output spikes into n-bit data, and transmit them either to memory or to the rest of the architecture.

The main task of the readout logic is to count the number of output spikes over an application-specific time window, which is set at configuration time.

3.6 Integration with a Processor

While the accelerator presented in this article is tested as a standalone chip, we briefly discuss in this section potential integration paths with a processor. There are two main possible configurations: using the accelerator to process data already stored in memory, or using it between a sensor and a processor (or between a sensor and the memory).

In the first configuration, the accelerator would be used like a standard ASIC or FPGA, and it would have to be augmented with a DMA. Note that the speed of the accelerator can be increased because the operating speed of the analog neuron can be ramped up. With a maximum input spike frequency of 1MHz, a clock frequency of 500MHz, and a synaptic weight resolution of 7 bits, the maximum output spike rate is $500 \times \frac{10^6}{2^7} = 3.8$ MHz. Increasing the clock frequency or decreasing the resolution would increase the analog neuron maximum output spike rate. Moreover, the potential data/input parallelism of the accelerator could largely compensate for a lower clock frequency.

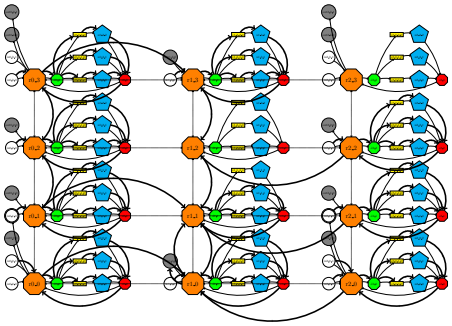


Figure 8: *Example mapping for a scaled down 3x3 Motion Estimation application (thick edges are software edges, routers are (orange) octagons, synapses are (yellow) rectangles, neurons are (blue) pentagons, inputs and outputs are (white and grey) circles, small (green and red) octagons are tile incoming/outgoing circuits).*

The second configuration is potentially the most promising. In the context of embedded systems (e.g., smartphones), the need to go from sensors to memory, and back to the processor, is a major source of energy inefficiency. At the same time, some of the key computational tasks performed in smartphones relate to sensor data (audio, image, radio). The accelerator has the potential to remove that memory energy bottleneck by introducing sophisticated processing at the level of the sensors, i.e., accelerators *between* sensors and processors. It can also significantly reduce the amount of data sent to the processor and/or the memory by acting as a preprocessor, e.g., identifying the location of faces in an image and only sending the corresponding pixels to the processor (for more elaborate processing) or the memory (for storage).

4. BACK-END COMPILER

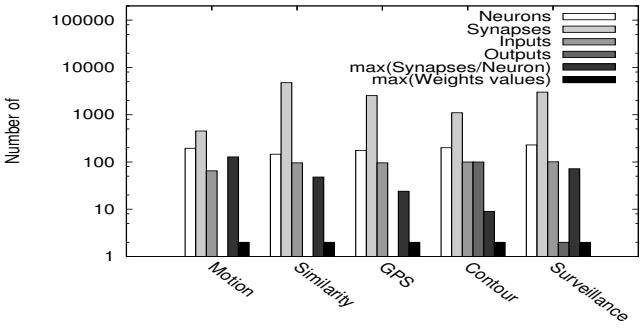
A signal processing application is provided as a textual representation of the application graph, which lists individual (software) neurons and their id, as well as edges between neurons, along with the synaptic weight carried by the edge (software synapse). The compiler tool flow is the following: after parsing the graph, the compiler uses an architecture description file (e.g., #tiles, #channels, #neurons/tile, #synapses/neuron, grid topology and # of inputs/outputs, etc) to map the corresponding application graph onto the architecture. An example mapping for a scaled down Motion Estimation application is shown in Figure 8; in this example 10 out of 12 tiles, 29 out of 36 neurons are used.

The mapping task bears some resemblance to the mapping of circuits onto FPGAs, and it is similarly broken down into *placement* (assigning software neurons to hardware neurons) and *routing* (assigning software edges to hardware paths through channels and routers). However, the fact the data passing through channels are not values, but only unary spikes, provides significant mapping optimization opportunities, which would not be possible with traditional FPGAs.

4.1 Capacity Issues of Software Graphs

Placement consists in mapping software neurons onto hardware neurons. However, the connectivity of software neurons can vary broadly. Consider in Figure 9 our different target applications: they have roughly the same number of neurons (between 146 to 200), but their number of synapses varies from 452 to 4754.

When the number of inputs of a software neuron exceeds the number of hardware synapses allocated per neuron, in theory, the mapping is not possible. A simple solution consists in splitting



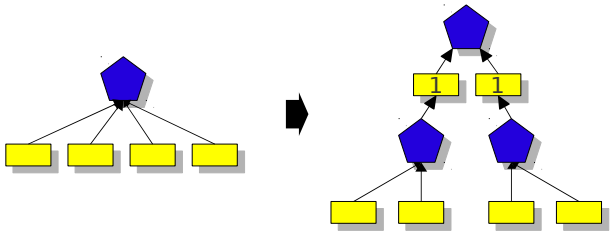


Figure 10: *Splitting neurons to reduce the number of synapses per neuron.*

neurons [38] which have more synapses than the maximum number provided by the hardware, see Figure 10. All the introduced synaptic weights are equal to 1 since the role of the added intermediate neurons is to build partial sums.

Unfortunately, for some very densely connected applications, such as GPS, see Section 5, this necessary splitting can drastically increase the requirements in number of synapses and neurons, and result in considerable architecture over-provisioning, defeating the goal of obtaining a low-cost low-energy design. In Figure 11, we have plotted the average total number of software synapses (and neurons) of all applications after applying this mandatory splitting, for different hardware configurations, i.e., different number of hardware synapses per neuron (on the x-axis). When the number of hardware synapses per neuron is small (e.g., 2), massive splitting is required, resulting in high numbers of software synapses (and software neurons), more than 7500 on average for the target applications, see *Soft. synapses (nb split)*. However, because the number of hardware neurons must be equal or greater to the number of software neurons (in order to map them all), the total number of hardware synapses (total number of hardware neurons $\times$ number of synapses per neuron) cannot drop below a certain threshold; and that threshold dramatically increases with the number of hardware synapses per neuron, see *Hard. synapses*. For 64 hardware synapses per neuron, the minimum architecture configuration which can run all applications has more than 10,000 hardware synapses, even though the maximum number of required software synapses is only 4754 for our benchmarks.

4.2 Sharing Synapses and Channels

However, by leveraging the unary (spike) nature of the data passing through channels and synapses, several properties of the hardware/software tandem allow to drastically reduce the number of hardware synapses and channels, and thus the overall hardware cost.

(1) If two different software edges with same target neuron pass

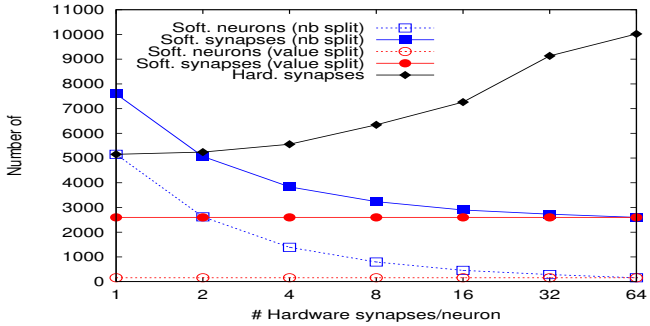


Figure 11: *Average number of software synapses (and neurons) and hardware synapses, when splitting according to number of synapses or weights values.*

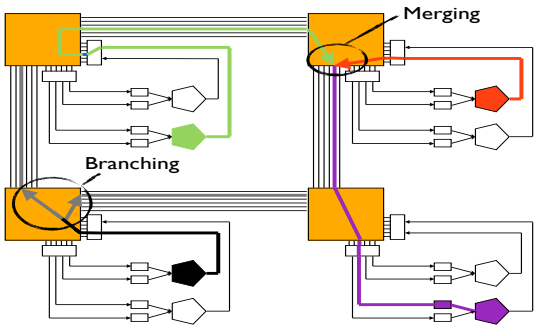


Figure 12: *Sharing after merging, or branching after sharing.*

through synapses *with same weight*, they can in fact be mapped to the same hardware synapse, instead of being mapped to two distinct synapses.

(2) Software edges with the same target neuron and same weight can also share the same channel as soon as they meet on their (hardware) path to the target neuron.

(3) Software edges with the same source neuron, but different target neurons, can share the same channel until they branch out to their respective target neuron.

Channel sharing is illustrated in Figure 12 for cases (2) and (3).

We can leverage synapse and channel sharing as follows. For synapses, instead of splitting a software neuron if its *number* of software synapses exceeds the number of hardware synapses per neuron, we can split the neuron only if the number of distinct synaptic weights *values* among its synapses exceeds the number of hardware synapses per neuron, a much more infrequent case, as illustrated with the comparison of *Max(Synapses/neuron)* and *Max(Weights values)* in Figure 9. With that new splitting criterion, the average number of software synapses remains almost the same as in the original software graph, see *Soft. synapses (value split)* in Figure 11.

We leverage channel sharing by mapping compatible software edges to the same hardware channels. In Figure 13 we show the number of hardware channels required with and without factoring channel sharing. For some of the applications, using channel sharing can decrease the number of required hardware channels by up to 50x. Branching and merging are performed by the compiler, i.e., they are *static* optimizations, not run-time ones. The only required hardware modification is in the router switch configuration, see Figure 6: implementing branching means allowing an incom-

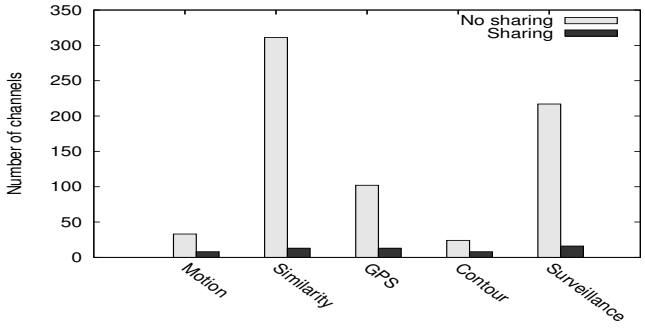


Figure 13: *Number of minimum required channels with and without sharing.*

ing channel to be routed to multiple (instead of a single) outgoing channels, and vice-versa for merging.

5. METHODOLOGY

In this section, we describe our methodology and the applications used for the evaluation.

Architecture design and evaluation. The analog neuron has been evaluated using the Cadence Virtuoso Analog Design environment. The digital part of the tile architecture and the routing network have been implemented in VHDL and evaluated using Synopsys and the STM technology at 65nm; we used Synopsys PrimeTime to measure the energy of the digital part, and the Eldo simulator of the STM Design Kit for the analog part. As mentioned before, a test chip containing several neurons and some test logic for evaluating the analog design has been recently taped out. We used this test chip to validate the neuron energy per spike.

In addition to synthesis, we have implemented a C++ simulator to emulate the operations of the architecture, the injection of spikes, spike collisions (especially at the switch level) and the quality of the readout results.

Compiler. Due to the atypical nature of the optimizations, the back-end compiler has been implemented from scratch. While called a back-end, it actually parses the source textual representation of the application graphs, loads an architecture description file with all the characteristics of the micro-architecture, and generates a routing and placement of the application. It is called a back-end rather than a full compiler, because we have focused on routing and placement optimizations for now.

Processor comparison. We compare the accelerator against a simple in-order processor architecture running at 1GHz similar to an Intel Atom embedded processor. Even though the accelerator architecture may be closer to FPGAs or systolic arrays than cores, we compare against a core because, within a heterogeneous multi-cores, tasks will be mapped either to a core or to accelerators. Moreover, this is a reference point, and the literature provides ample evidence of the relative characteristics between cores, systolic arrays, ASICs [18] and FPGAs [22]. The processor pipeline has 9 stages, the L1 instruction cache is 32KB, 8-way, the L1 data cache is 24KB, 6-way, the memory latency is set at 100ns, and its measured peak power is 3.78W. The simulator was implemented using Gem5 [4], and power was measured using McPat [24]. We wrote C versions of the benchmarks and compiled them using the GCC cross-compiler 4.5.0 and -O optimization to generate Alpha binary; we measure only the time and energy of the functional part of the program (excluding initialization, allocation, etc), equivalent to the application graph mapped onto the grid.

For the processor experiments, in order to avoid biasing the comparison towards the accelerator by factoring idle time of the processor waiting for (low-frequency) inputs, we stored all inputs in a trace, and we processed them at full speed (no idle time). We then divide the total energy by the number of inputs, and we obtain an energy per row, which is used as the basis for comparison.

Applications. We have selected and implemented the textual graph representation of five signal processing applications, which are described below. Due to paper length constraints we omit the application graphs; note however that a downscaled version of the graph of motion estimation (3x3 resolution vs. 800x600 in our benchmark) is shown in Figure 1, and that statistics on the number of nodes and synapses (edges) are shown in Figure 9. For each application, the relative error is computed with respect to the processor output. We use the error to then derive the SNR (Signal-to-Noise Ratio, i.e., $\frac{\mu}{\sigma}$ where μ is the average value and σ is the standard deviation), a common error metric for signal processing applications. The general Rose criterion states that a signal is detectable above its background noise if SNR $\geq$ 5 [28]. For each benchmark, we define below the nature of the application-specific error, used to compute the SNR.

Image processing (Contour). Several typical image processing tasks can be implemented. We chose edge detection, which is performed by repeated convolution of a filter on an incoming image. The error is defined as the average difference between the pixel luminance obtained with the accelerator and the processor. For instance, the Rose criterion for image processing means that with SNR $\geq$ 5, 100% of image features can be distinguished.

Similarity (Similarity). An input signal, e.g., image, audio, radio (we chose images for our benchmark), is compared against a reference image, and the degree of correlation between the two inputs (i.e., the sum of products of differences between the two images) is computed. This process can typically be used to pre-filter a stream of information, and later invoke a more sophisticated image recognition process. The degree of correlation is the error.

Motion estimation (Motion). This application has already been briefly described in Section 2.1. The input is an SVGA video feed (800x600@100Hz) presented as a continuously varying set of pixels luminance, and we seek to detect variations across time. The evolution of the whole image is aggregated into a single value, which is the motion estimation. The error is computed just before the threshold application as the average of the absolute value of the luminance derivatives (computing the error after threshold application would result in a very low and favorable error).

GPS (GPS). In GPS applications, one of the computing intensive tasks consists in finding the time delay between a given satellite signal and a reference satellite signal (arbitrarily chosen among the detected satellites); the satellite signal corresponds to a voltage amplitude. This time delay is then used to compute the distance between the two satellites with respect to the observer. After repeating this process on several consecutive signals (3 or more), it is possible to obtain the observer location via trilateration. The error is defined as the maximum of the correlations between the two signals, thus becoming a cross-correlation problem. The GPS signal is first sampled, and then several time-shifted autocorrelations are run in parallel, at a lower frequency than the original signal.

Surveillance. We combine several of the aforementioned applications into one larger application, implementing a more complex processing task, namely surveillance. It detects movement within an incoming video feed, filters out the image to find shapes contours when motion is detected, and then compares them again a human silhouette to detect human presence. The error is based on

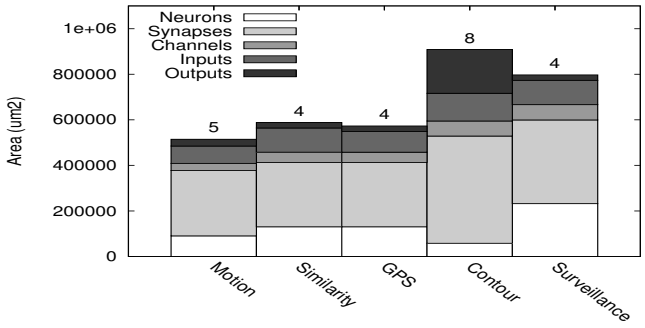


Figure 14: *Optimal architecture area, and breakdown, for each application (grid dimension on top).*

Synapses	Neurons	Routing	Inputs	Outputs
1022712	204125	155444	95000	150000

Table 1: *Area breakdown (in μm^2) of tradeoff configuration (4 synapses/neuron, 12 neurons/tile, 20 channels, 5x5 tiles).*

the output of the similarity task, which is the last one in the surveillance pipeline.

Measurements. Run-time and energy derive from the number of spikes required to achieve the desired accuracy target. They are given per input, e.g., one image for *Contour*, *Similarity* and *Surveillance*, 40ms video for *Motion*, and one frame for *GPS* (a vector of voltages).

6. EVALUATION

We first explore the best configurations with respect to area. We then evaluate the latency and energy of that configuration. We also evaluate the robustness (fault tolerance) of the architecture.

6.1 Design-Space Exploration

There are complex interplays between the four main parameters of the grid architecture: the grid dimension, the number of channels, the number of neurons per tile and the number of synapses per neuron. As the number of neurons increases, the total number of synapses increases as well, and thus, the number of connections that must be routed by the switch to the tile would increase similarly (along with the output connections from the neuron). Since the switch size, and the size of the intra-tile crossbars, increases exponentially with the number of wires to route and the number of neurons, it is not necessarily efficient to use very large switches, and thus a very high number of neurons per tile and/or synapses per neuron.

For each benchmark, we randomly explored the design space in order to find a near-optimal configuration. The area breakdown of these application-optimal configurations is shown in Figure 14, with the grid dimension on top of bars. The synapses account for the largest area; note that this figure not only includes the synaptic storage but also the DAC and the surrounding logic, the DAC being the largest logic block. In the future, we plan to investigate synapses implemented with novel memory devices such as memristors [9]; not only the DAC would no longer be necessary, but the hardware synapses would be more densely implemented.

We then sought a tradeoff configuration for all applications combined. Because each application has different routing, or computational requirements, some characteristics of the tradeoff configuration are naturally over-dimensioned for each application. As a result, the tradeoff configuration is about 80% larger than the largest optimal application-specific configuration, though it still only requires $1.63mm^2$ (including I/O logic). The area breakdown is shown in Table 1. From now on, all measurements are performed using this tradeoff configuration.

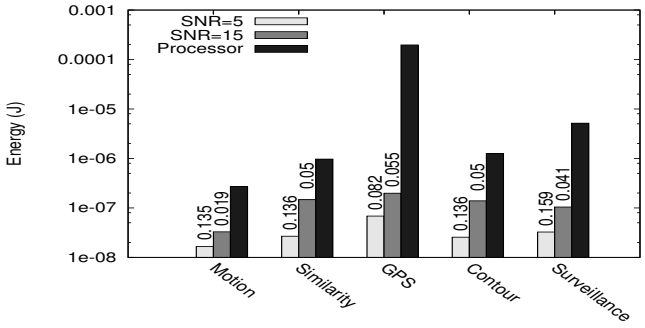


Figure 15: *Energy per input for accelerator and processor; the energy/accuracy tradeoff is illustrated with two different SNR values (the application relative error is shown on top of the accelerator bars).*

Synapses	Neurons	Routing	Inputs	Outputs
98.26%	1.63%	0.11%	0.01%	<0.01%

Table 2: *Energy breakdown (%) among synapses, neurons and routing.*

6.2 Energy

We now evaluate the energy spent to accomplish the different applications, more exactly the energy spent processing one input, since there is a potentially infinite number of inputs. As explained in Section 2.3, the key parameter is the readout time window, which, in turn, is determined by the target signal-to-noise ratio (SNR), or, in other words the energy/accuracy tradeoff.

In Figure 15, we show the energy of the accelerator for SNR=5 and SNR=15, as well as for the processor. On average, the energy consumed by the accelerator is equal to only 2.3% of the energy consumed by the processor for SNR=5, and 8.1% for SNR=15, i.e., the energy ratio with the processor varies between 7x and about two orders of magnitude, depending on the application and SNR. Considering reported ratios between ASICs and high-end processors are about two orders of magnitude [18], the accelerator performs only slightly worse than an ASIC at SNR=5.

Since the SNR has a direct impact on the relative application error (see Section 5 for the definition of each application error), we also report the relative error on top of each accelerator bar. As mentioned before, the error increases as the energy decreases. At SNR=5, the average application error is 13.0%, and 4.3% at SNR=15.

In Table 2, we show the energy breakdown among the three main components: neurons, synapses and routing. The unary coding design choices and the associated compiler optimizations enable to bring the routing energy cost down to 0.11%; as a result of analog design, the energy share of the neuron is now very low at 1.63%. So additional improvements should now come from synapse design, which is beyond the scope of this article.

6.3 Fault Tolerance

We evaluate the tolerance of the accelerator to transient faults by injecting a Gaussian noise which results in random spike loss; the injection is performed by randomly removing/adding spikes on the communication channels. We then used the relative error, defined for each application in Section 5, and computed with respect to

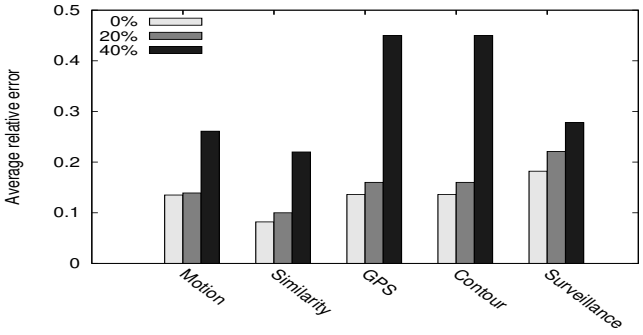


Figure 16: *Impact of spike loss on output average relative error.*

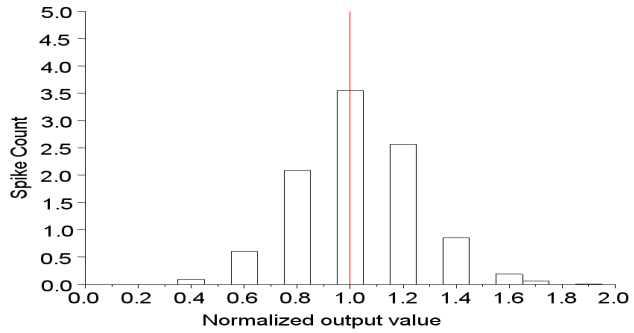


Figure 17: *Surveillance value distribution.*

the processor output, to assess the fault tolerance capability of the accelerator.

We first set the readout time window, which, as mentioned before, already has an impact on application error. We use a time window corresponding to SNR=5, a case already shown in Section 6.2 and Figure 15.

We then experiment with two noise levels, respectively corresponding to a spike loss of 20% and 40%. As shown in Figure 16, a spike loss of 20% induces almost the same average relative error as no spike loss. When the spike loss is very high though (40%), the error becomes significant; note that increasing the readout time window would bring this error down, at the expense of energy. Moreover, the readout time window is limited by the input bandwidth, i.e., the time between two readouts cannot be higher than the time between two inputs.

In Figure 17, we show the distribution of the normalized output value (with respect to the mean) for a spike loss of 20% for the *Surveillance* application; the correct expected value is marked with a (red) vertical line. Recall that the value is determined by counting spikes, hence the *Spike count* y-axis. As can be observed, values are arranged in a Gaussian-like distribution around the target value, higher errors being less probable.

6.4 Time

We now analyze the computational bandwidth of the accelerator, i.e., the time between two outputs. As mentioned in Section 2.3, the readout time is actually determined by the readout time window, i.e., the time during which output spikes are counted, and it depends on the target signal-to-noise ratio. In Figure 18, we provide the readout time for SNR=5 and SNR=15: the readout time must increase in order to achieve a higher accuracy.

Even though the goal of this architecture is not to minimize execution time (most target applications can be considered as soft

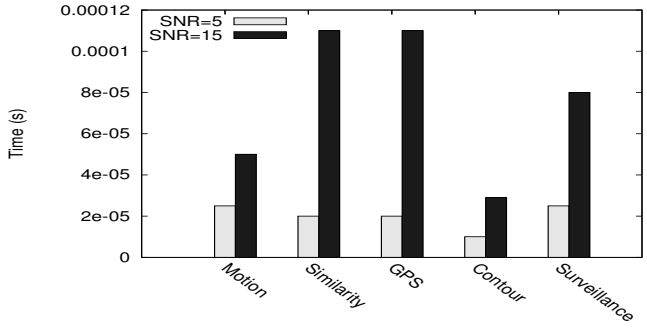


Figure 18: *Readout time per input.*

Motion	Similarity	GPS	Contour	Surveillance
0.01	0.13	0.08	23.82	0.16

Table 3: *Execution time per input of processor over accelerator (SNR=5).*

real time applications), we compare against the processor execution time in order to provide a reference point. In Table 3, we indicate the ratio of the processor execution time over the accelerator execution time. For most applications, the processor is faster, but we can note that for an application like *Contour*, which has as many outputs as the number of inputs (an image comes in, a processed image comes out), and which thus exhibits massive parallelism and bandwidth, the accelerator actually outperforms the processor. However, for any application, we can trade the *temporal* accumulation of spikes for a *spatial* accumulation by having more neurons perform similar/redundant computations in parallel.

7. RELATED WORK

Due to stringent energy constraints, such as Dark Silicon [18], there is a growing consensus that future high-performance micro-architectures will take the form of heterogeneous multi-cores, i.e., combinations of cores and accelerators. Accelerators can range from processors tuned for certain tasks, to ASIC-like circuits, such as the ones proposed by the GreenDroid project [42], or more flexible accelerators capable of targeting a broad range of, but not all, tasks [15].

The accelerator proposed in this article follows that spirit of targeting a specific, but broad, domain. We focus on signal processing tasks traditionally handled by DSPs, such as the TI C6000 [1], or FPGA circuits. However, FPGAs or DSPs usually do not exhibit tolerance to transient faults. Even though the grid-like structure of our accelerator is reminiscent of FPGAs, one or two neurons can realize the operations requiring several LUTs (e.g., addition, temporal derivation, etc), and the routing network of our accelerator is significantly less dense; finally, FPGAs cannot take advantage of the optimizations of Section 4, which play a significant role in reducing the overall cost of the routing network. Beyond DSPs and FPGAs, the property that signal processing tasks handle real-world input data has been leveraged in the past for investigating alternative processing approaches [33]. For instance, Serrano-Gotarredona et al. [32] recently demonstrated an efficient smart retina by processing real-time continuous analog inputs. Analog hardware for signal processing tasks has been considered in the past, including reconfigurable grids of analog arrays [17] which bear some similarity to the proposed accelerator. However the tile is completely different, composed of an analog array, instead of an analog neuron. Although the grid organization provides application flexibility, due to the basic nature of the operators (operational

transconductance amplifiers, capacitances, transistors), it remains almost as difficult to program as it is to design traditional analog circuits.

In order to achieve low energy, low cost and fault tolerance, we consider unary coding, i.e., spike coding, of information. A natural operator for processing spike coding are neurons. Hardware implementations of spiking neurons have been pioneered by Mead [25]. While many implementations of hardware neurons and neural networks have been investigated, the goal is usually fast modeling of biological neural networks [31]. The goal of this accelerator is to implement signal-processing tasks as efficiently as possible. While it uses analog neurons as operators, it is not a neuromorphic architecture in the sense that it doesn't try to emulate the connectivity and structure of biological neural networks; it does not rely on learning either, it directly programs the synaptic weights; the latter is not necessarily a better approach than learning for certain tasks, but it is a more familiar and controlled approach for the computing systems community at large, based on commonplace tools such as libraries and compilers. Even though the architecture is programmed, it is not deterministic either due to possible spike loss, but the spike loss is infrequent enough that it has no noticeable impact for the target (signal processing) applications. Some neuromorphic chips have been used to implement signal-processing applications beyond biological emulation [43], however, they come with a number of features that make them potentially less efficient, hardware-wise for that purpose. One of the most distinctive feature is that such chips rely on Address-Event Representation (AER) [5] in order to implement the very high connectivity (in the thousands) of biological neural networks. In AER, the connection is realized by addressing a memory, instead of direct physical links via a routing network, as in our accelerator; an example is Neurogrid [34] which also uses analog neurons and asynchronous digital routing, but the connectivity is implemented using AER. While this approach makes a lot of sense for the emulation of biological neural networks, it may not be the best choice when energy is a primary concern because of the need to address a look-up table and the fact the spike is implemented with multiple bits (address) instead of one. Moreover, many signal-processing applications do not require the high connectivity of biological neural networks, and the fact the application graphs are built out of low fan-in/fan-out operators further reduces the connectivity requirements, albeit sometimes at the cost of more resources (splitting). In an earlier work, we mentioned the notion of an accelerator based on a grid of analog spiking neurons with direct physical links [38], but no micro-architecture design had been investigated, and the programming is only described as largely similar to the place and route process of FPGAs, while we show in this article that it can be significantly more complex.

Recently, hardware neural networks have been increasingly considered again as potential accelerators, either for very dedicated functionalities within a processor, such as branch prediction [2], for their energy benefits [26], or because of their fault-tolerance properties [37, 19]. However, all the aforementioned neural networks rely on learning, and there is a large body of work on learning-based hardware neural networks of which it would not be possible to cite a representative subset here.

The growing attention to hardware neural networks is symptomatic of a trend to trade some loss of application accuracy for fault-tolerance capabilities and/or energy benefits. This notion of inexact computing has been pioneered by Chakrapani et al. [8], and it has been recently pursued by Kruijf et al. [21] and Esmaeilzadeh et al. [14].

8. CONCLUSIONS AND FUTURE WORK

We have leveraged the properties of continuous real-world input, especially low-frequency and tolerance to noise, as well as recent research in signal processing operators based on spiking neurons, to design the micro-architecture of a cost-effective low-energy accelerator exhibiting graceful output degradation to noise/faults. We have also shown that careful optimizations, specific to this architecture and type of applications, play a major role in keeping the architecture cost down.

Potentially, this architecture can exhibit massive parallelism, but 2D implementation and pin restrictions limit its input bandwidth. We plan to explore 3D implementations where sensor layers would be directly stacked on top of such an accelerator in order to achieve drastically higher input bandwidth, and truly take advantage of the potential parallelism.

9. ACKNOWLEDGMENTS

We want to thank the reviewers for their helpful suggestions. This work was supported by the ANR project Arch^{2}Neu.

10. REFERENCES

[1] "TMS320C6000 CPU and instruction set reference guide," Texas Instruments, Tech. Rep., 2006.

[2] R. S. Amant, D. A. Jimenez, and D. Burger, "Low-power, high-performance analog neural branch prediction," in *International Symposium on Microarchitecture*, Como, 2008.

[3] J. V. Arthur and K. Boahen, "Silicon-Neuron Design: A Dynamical Systems Approach," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 58, no. 99, p. 1, 2011.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[5] K. A. Boahen, "Point-to-point connectivity between neuromorphic chips using address events," *IEEE Transactions on Circuits and Systems*, vol. 47, no. 5, pp. 416–434, 2000.

[6] S. Borkar, "Design perspectives on 22nm CMOS and beyond," in *Design Automation Conference*, Jul. 2009, pp. 93–94.

[7] D. Burger, "Future Architectures will Incorporate HPUs (keynote)," in *International Symposium on Microarchitecture*, 2011.

[8] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee, "Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology," in *Design, Automation and Test in Europe Conference*, Munich, 2006, p. 1110.

[9] A. Chanthbouala, V. Garcia, R. O. Cherifi, K. Bouzehouane, S. Fusil, X. Moya, S. Xavier, H. Yamada, C. Deranlot, N. D. Mathur, M. Bibes, A. Barthélémy, and J. Grollier, "A ferroelectric memristor." *Nature materials*, vol. 11, no. 10, pp. 860–4, Oct. 2012. [Online]. Available: http://dx.doi.org/10.1038/nmat3415

[10] S. Deneve, "Bayesian Spiking Neurons I : Inference," *Neural Computation*, vol. 117, pp. 91–117, 2008.

[11] C. Eliasmith and C. H. Anderson, *Neural Engineering: Computation, Representation and Dynamics in Neurobiological Systems*. MIT Press, 2003.

[12] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, Jun. 2011.

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, T. Harris and M. L. Scott, Eds. ACM, 2012, pp. 301–312.

[14] ——, "Neural Acceleration for General-Purpose Approximate Programs," in *International Symposium on Microarchitecture*, 2012.

[15] K. Fan, M. Kudlur, G. S. Dasika, and S. A. Mahlke, "Bridging the computation gap between programmable processors and hardwired accelerators," in *HPCA*. IEEE Computer Society, 2009, pp. 313–322.

[16] W. Gerstner and W. M. Kistler, *Spiking Neuron Models*. Cambridge University Press, 2002.

[17] T. S. Hall, C. M. Twigg, J. D. Gray, P. Hasler, and D. V. Anderson, "Large-Scale Field-Programmable Analog Arrays for Analog Signal Processing," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 52, no. 11, pp. 2298–2307, 2005.

[18] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *International Symposium on Computer Architecture*. New York, New York, USA: ACM Press, 2010, p. 37.

[19] A. Hashmi, A. Nere, J. J. Thomas, and M. Lipasti, "A case for neuromorphic ISAs," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY: ACM, 2011.

[20] A. Joubert, B. Belhadj, O. Temam, and R. Heliot, "Hardware Spiking Neurons Design: Analog or Digital?" in *International Joint Conference on Neural Networks*, Brisbane, 2012.

[21] M. D. Kruijf, S. Nomura, and K. Sankaralingam, "Relax : An Architectural Framework for Software Recovery of Hardware Faults," in *International Symposium on Computer Architecture*. Saint-Malo: ACM Press, 2010.

[22] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *International Symposium on Field Programmable Gate Arrays*, ser. FPGA '06. New York, NY, USA: ACM, Feb. 2006, pp. 21–30.

[23] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and Single-Driver Wires in FPGA Interconnect," in *International Conference on Field-Programmable Technology*. IEEE, 2004, pp. 41–48.

[24] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 469–480.

[25] C. Mead, *Analog VLSI and Neural Systems*. Addison-Wesley, 1989.

[26] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm," in *IEEE Custom Integrated Circuits Conference*. IEEE, Sep. 2011, pp. 1–4.

[27] T. K. Moon and W. C. Stirling, *Mathematical Methods and Algorithms for Signal Processing*. Prentice Hall, 1999.

[28] A. Rose, *Advances in Electronics , Vol. 1*, L. N. Y. A. P. Martin, Ed., 1948.

[29] U. Rutishauser and R. J. Douglas, "State-dependent computation using coupled recurrent networks," *Neural computation*, vol. 21, no. 2, pp. 478–509, 2009.

[30] R. Sarpeshkar and M. O'Halloran, "Scalable hybrid computation with spikes." *Neural computation*, vol. 14, no. 9, pp. 2003–2038, 2002.

[31] J. Schemmel, J. Fieres, and K. Meier, "Wafer-scale integration of analog neural networks," in *International Joint Conference on Neural Networks*. Ieee, Jun. 2008, pp. 431–438.

[32] R. Serrano-Gotarredona, M. Oster, P. Lichtsteiner, A. Linares-Barranco, R. Paz-Vicente, F. Gomez-Rodriguez, L. Camunas-Mesa, R. Berner, M. Rivas-Perez, T. Delbruck, S.-C. Liu, R. Douglas, P. Hafliger, G. Jimenez-Moreno, A. Civit Ballcels, T. Serrano-Gotarredona, A. J. Acosta-Jimenez, and B. Linares-Barranco, "CAVIAR: a 45k neuron, 5M synapse, 12G connects/s AER hardware sensory-processing- learning-actuating system for high-speed visual object recognition and tracking." *IEEE transactions on neural networks*, vol. 20, no. 9, pp. 1417–38, Sep. 2009.

[33] S. Sethumadhavan, R. Roberts, and Y. Tsividis, "A Case for Hybrid Discrete-Continuous Architectures," *IEEE Computer Architecture Letters*, vol. 99, no. RapidPosts, 2011.

[34] R. Silver, K. Boahen, S. Grillner, N. Kopell, and K. L. Olsen, "Neurotech for neuroscience: unifying concepts, organizing principles, and emerging tools." *The Journal of neuroscience : the official journal of the Society for Neuroscience*, vol. 27, no. 44, pp. 11 807–19, Oct. 2007.

[35] M. V. Srinivasan and G. D. Bernard, "A proposed mechanism for multiplication of neural signals," *Biological Cybernetics*, vol. 21, no. 4, pp. 227–236, 1976.

[36] Steve Keckler, "Life After Dennard and How I Learned to Love the Picojoule (keynote)," in *International Symposium on Microarchitecture*, Sao Paolo, Dec. 2011, p. Keynote presentation.

[37] O. Temam, "A Defect-Tolerant Accelerator for Emerging High-Performance Applications," in *International Symposium on Computer Architecture*, Portland, Oregon, 2012.

[38] O. Temam and R. Heliot, "Implementation of signal processing tasks on neuromorphic hardware," in *International Joint Conference on Neural Networks*. IEEE, Jul. 2011, pp. 1120–1125.

[39] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 2304. Berlin, Heidelberg: Springer, Mar. 2002.

[40] B. P. Tripp and C. Eliasmith, "Population models of temporal differentiation." *Neural computation*, vol. 22, no. 3, pp. 621–659, 2010.

[41] A. van Schaik, "Building blocks for electronic spiking neural networks." *Neural networks*, vol. 14, no. 6-7, pp. 617–628, 2001.

[42] G. Venkatesh, J. Sampson, N. Goulding-hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "QsCORES : Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores Categories and Subject Descriptors," in *International Symposium on Microarchitecture*, 2011.

[43] R. J. Vogelstein, U. Mallik, J. T. Vogelstein, and G. Cauwenberghs, "Dynamically reconfigurable silicon array of spiking neurons with conductance-based synapses," *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp. 253–265, 2007.