BenchNN: On the Broad Potential Application Scope of Hardware Neural Network Accelerators

Tianshi Chen[◇], Yunji Chen[◇], Marc Duranton[†], Qi Guo[♭], Atif Hashmi[*],
Mikko Lipasti[*], Andrew Nere[*], Shi Qiu[♯], Michèle Sebag[‡], Olivier Temam[§],
◇ ICT, China, † CEA LIST, France, ♭ IBM Research, China, ∗ University of Wisconsin,
♯ USTC, China ‡ LRI, CNRS, France, § INRIA, France

***Abstract*—**

Recent technology trends have indicated that, although device sizes will continue to scale as they have in the past, supply voltage scaling has ended. As a result, future chips can no longer rely on simply increasing the operational core count to improve performance without surpassing a reasonable power budget. Alternatively, allocating die area towards *accelerators* targeting an application, or an application domain, appears quite promising, and this paper makes an argument for a neural network hardware accelerator. After being hyped in the 1990s, then fading away for almost two decades, there is a surge of interest in hardware neural networks because of their energy and fault-tolerance properties. At the same time, the emergence of high-performance applications like Recognition, Mining, and Synthesis (RMS) suggest that the potential application scope of a hardware neural network accelerator would be broad. In this paper, we want to highlight that a hardware neural network accelerator is indeed compatible with many of the emerging high-performance workloads, currently accepted as benchmarks for high-performance micro-architectures. For that purpose, we develop and evaluate software neural network implementations of 5 (out of 12) RMS applications from the PARSEC Benchmark Suite. Our results show that neural network implementations can achieve competitive results, with respect to application-specific quality metrics, on these 5 RMS applications.

***Keywords*-neural networks; benchmark; accelerator; PARSEC;**

I. Introduction

Due to the lack of voltage scaling, only a fraction of transistors on a chip will be used simultaneously in the future, an evolution coined Dark Silicon [1], [2]. This evolution jeopardizes many-cores and massive on-chip parallelism: if not all cores can be used simultaneously, then other paths for improving program performance must be sought. The most promising alternative is *customization*: instead of splitting each algorithm of a program into many threads, a chip can embed special hardware, called *accelerators*, which can execute each of these algorithms much more efficiently than normal cores. It is increasingly likely that, in the future, much of the computing heavy lifting will occur within such accelerators. On the other hand, the type and implementation of these accelerators remains an open and key micro-architecture research issue [3]. Accelerators can take many forms, and they range from specially configured cores, to reconfigurable circuits, to ASICs.

While technology constraints (power) determined the shift towards multi-cores and then heterogeneous multi-cores, another technology constraint, namely the increasing number of defects[4], may largely determine the nature of accelerators themselves. For instance, both cores and ASICs are very susceptible to transient or permanent faults. Reconfigurable circuits are almost as susceptible to transient faults; they can potentially cope with permanent faults thanks to their intrinsic redundancy, but they are known to be significantly less energy-efficient than ASICs [5].

One of the key challenges in designing such accelerators is finding the right balance between application scope and efficiency. Too broad of an application scope (like reconfigurable circuits), and the accelerator is energy inefficient; too small of a scope (like ASICs) and too many accelerators are necessary to cover a meaningful share of the application spectrum. As a result, some of the best candidate accelerators for heterogeneous multi-cores could be *multi-purpose* accelerators [6], [7], each covering a share, but not all, of the application spectrum, and the combination of which covers a large share of that spectrum.

In this article, we argue that one class of accelerators which can realize the multiple aforementioned constraints (energy efficiency, significant application scope, tolerance to permanent and transient faults) is a *hardware neural network*. Unlike other machine-learning algorithms, neural networks are known to be intrinsically tolerant to faults, and hardware neural networks have been recently shown to be effectively tolerant to transistor-level defects [8]. Hardware neural networks can also be designed so as to be very energy efficient [9], [10]. On the other hand, the notion that the application scope of hardware neural networks is broad is often met with skepticism. A common misconception is that neural networks are geared toward classification tasks only, and thus have a very restricted scope that is incompatible with general-purpose computing.

This perception of neural networks is a failure to acknowledge the drastic shift that high-performance applications have experienced in the past few years. Intel has attracted the attention of the community to RMS (Recog-

978-1-4673-4532-3/12/$31.00 ©2012 IEEE

Benchmark	Task	Main computational kernel	Category	ANN alternative
blackscholes	Option pricing	Differential equations	Approximation	Approximation using MLP[a]
bodytrack	Track 3D pose of body in video	Annealed particle filter	Classification	Feature extraction and recognition with CNN[b] [11]
canneal	Chip routing	Simulated annealing	Optimization	Optimization using HNN[c]
dedup	File compression	Hashing and compression	Classification	Hashing and compression using an unsupervised neural network
facesim	Modeling face movements	Image synthesis	Approximation	Interpolation using MLP (partial) [12]
ferret	Content (image) similarity	Feature extraction, indexing and hashing	Clustering/Classification	NN-based Gabor filters and SOM for comparison[d]
fluidanimate	Fluid simulation	Navier-Stokes equations	Approximation	CeNN[e] for solving Navier Stokes equation [13]
freqmine	Frequent itemset miner	Database requests	Classification	Learning features correlations [14] using MLP
streamcluster	Online clustering	Distance-based clustering	Clustering	Online clustering using SOM
swaptions	Option pricing	Simulated annealing	Approximation	Option pricing approximation using MLP
vips	Image processing library	Affine transformations and convolutions	Raw NN operation	Convolutions and filtering using CNNs as operators (no learning) [15]
x264	Video encoding	H264 algorithm	Classification	MLP to learn 2D transforms in NGVC, H265 [16]

[a]MLP stands for Multi-Layer Perceptron, a standard form of artificial neural network.
[b]CNN stands for Convolutional Neural Network.
[c]HNN stands for Hopfield Neural Network.
[d]SOM are Self-Organized Maps, another form of neural networks.
[e]CeNN stands for Cellular Neural Networks

Table I
Competitive ANN-based alternatives for PARSEC computational tasks.

nition, Mining and Synthesis) applications [17] as some of the most important emerging high-performance applications. That effort partly motivated the development of the PARSEC benchmark suite [18] by Princeton University. Many of the PARSEC benchmarks rely on four categories of algorithms: *classification*, *clustering*, *statistical optimization* and *approximation*, see Table I. While the PARSEC benchmarks rely on a varied set of techniques for each of these four kinds of algorithms, NNs (Neural Networks) can provide a competitive alternative for many of these tasks. In Table I, we briefly describe the core computational task of each PARSEC benchmark. 10 out of 12 benchmarks (especially Recognition and Mining tasks) correspond to tasks for which there are known competitive NN-based algorithms; for 2 benchmarks, part of the task could potentially be replaced using a neural network. In other words, the potential application scope of a hardware neural network accelerator is very broad.

In this article, we motivate the need for a NN accelerator by demonstrating the broad potential application scope of neural networks. We provide evidence for this claim by showing that a significant share of the popular PARSEC benchmark suite can be reimplemented using neural network algorithms. We pick at least one PARSEC benchmark corresponding to each of the four aforementioned categories, 5 tasks in total, and show that the quality of the results are usually competitive with that of the original task. Although some neural network alternatives may not achieve the same precision accuracy as a PARSEC implementations, we argue that in many cases a competitive (but slightly worse) accuracy paired with an efficient neural network accelerator is best. This is especially sensible in the context of embedded systems, where tasks such as recognition or mining need to be efficiently implemented, but need not achieve state-of-the-art accuracy.

For the sake of completeness, we also report execution times, but these *software* NN implementations are not meant to be competitive time-wise; they are usually significantly slower. Since hardware accelerators can provide speedups of several orders of magnitude compared to software versions run on a processor [19], [8], it is likely that a *hardware* version could be competitive time-wise (and naturally, energy-wise as well) in most (if not all) cases. Furthermore, the inherently parallel nature of neural network algorithms favors concurrent processing on massively parallel neural accelerators. Finally, such applications run on a hardware NN accelerator would be largely resilient to transient or permanent defects due to the ability to retrain neural networks in order to silence out faulty parts [8].

There are many different ways to implement hardware neural networks; BenchNN is designed to be independent of given hardware implementations; its main purpose is to assess the benefit of such accelerators for a broad set of general-purpose applications. We plan to distribute the reimplemented PARSEC tasks as a benchmark suite called *BenchNN*, at www.benchnn.org,[1] with the goal to stim-

[1]The web site will be open in the next few months.

PARSEC	% time spent in target task
blackscholes	99%
canneal	90%
ferret	95%
streamcluster	89%
dedup	95%

Table II
Percentage of time spent in PARSEC task which was replaced with an NN.

ulate research on hardware neural network accelerators.

II. Reimplementing PARSEC tasks as NNs

In this section, we explain how 5 PARSEC tasks can be implemented using neural networks. We briefly describe the task, explain which part of the PARSEC task can be redesigned using a neural network and how it can be done, describe the neural network algorithm used for that task, and compare the PARSEC and NN implementations using application-specific metrics.

In Table II, we list the PARSEC benchmarks we consider and the percentage of the execution time spent in the task which was replaced by the NN. We briefly comment on the remaining benchmarks in Section II-F.

A. Financial Market Prediction (blackscholes)

1) Problem Description: blackscholes is a financial application which predicts the price of options (a financial product) at a certain date in the future. For that purpose, it uses a partial differential equation introduced by Black and Scholes [20]. Due to the lack of a closed form expression, the solution must be numerically computed. Because the equation only provides a *prediction*, the solution comes with an error. This error is simply computed by comparing against the true option price once available.

2) From PARSEC Code to NN: In the PARSEC benchmark, the task is implemented as a set of direct, non-iterative, computations spread over two functions, CNDF and BlkSchlsEqEuroNoDiv. The latter function is the main one: its inputs are the parameters based on which the option price is predicted (including the date at which it is predicted), and its output is the predicted option price at that date. These parameters are listed below.

- *spot price:* the market price at the time of the prediction;
- *risk-free interest rate*
- *strike price:* the price at which the contract (prediction) is passed;
- *volatility*
- *call time:* the delay (in days or years) till the prediction target date;
- *option type*

Since all input parameters are scalar, this model can be easily expressed using an NN with 6 input parameters and one output. Since the inputs and outputs of BlkSchlsEqEuroNoDiv are exactly the ones used for the NN model, the NN algorithm can be directly substituted within the PARSEC code, the aforementioned function serving as an API call to the NN algorithm.

We use a Multi-Layer Perceptron (MLP) to implement that model; we briefly recall the main principles of an MLP in the next section.

3) NN Algorithm: The most traditional form of artificial neural network is a Multi-Layer Perceptron, which typically contains one input layer, one output layer, and one or several hidden layers. The optimal number of layers and the number of neurons per layer are typically explored during a training phase, and the selected parameters for the blackscholes problem are provided in the next section. MLPs are feed-forward networks, where information flows from the input layer ($l = 0$) to the output layer ($l = 2$). Each neuron performs the following computations. Let y_j^l the output of neuron j at layer l, $y_j^l = f\left(o_j^l\right)$ where $o_j^l = \sum_{i=0}^{N_{l-1}} w_{ji}^l y_i^{l-1}$, w_{ji} is the synaptic weight between neuron i in layer $l-1$ and neuron j in layer l, N_l is the number of neurons in layer l, and f is the activation function, often the sigmoid $f(x) = \frac{1}{1+exp^{-x}}$. We train the network using back-propagation [21], the most popular training algorithm.

4) Evaluation:

Accuracy. In PARSEC, the input data is the recorded evolution of an option, i.e., the actual option price. This input data can be broken down into training and test sets. We use 10-fold cross-validation for these experiments, which were themselves repeated 10 times.

We first conducted an exploration of the network hyper parameters and found that the following configuration performs best:

- 1 hidden layer
- 15 neurons in hidden layer
- learning rate: 0.01

We trained the network using 10,000 epochs (iterations). We define the error for one input as the difference between the predicted and real option price; the model error is defined as the average error over all test inputs. Overall, the error of the PARSEC benchmark is $1e-5$ using single precision, while the neural network error is $3e-5$, i.e., slightly higher. In Figure 1, we plot the predicted value (y-axis) against the actual value (x-axis) for both blackscholes and the neural network model.

Slowdown. The slowndown of the neural network version over the PARSEC version is 3.6x.

B. Placement Optimization (canneal)

1) Problem Description: canneal is an optimization benchmark which uses simulated annealing to minimize the

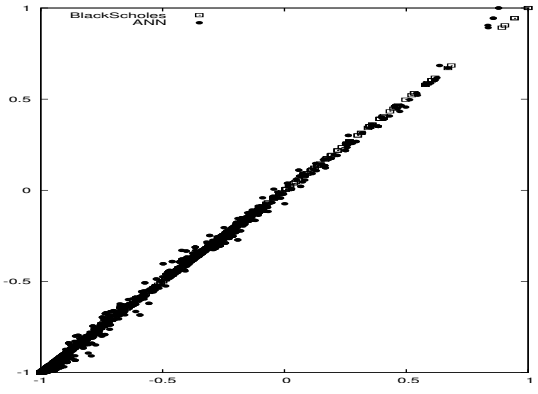

Figure 1. *PARSEC vs. NN error.*

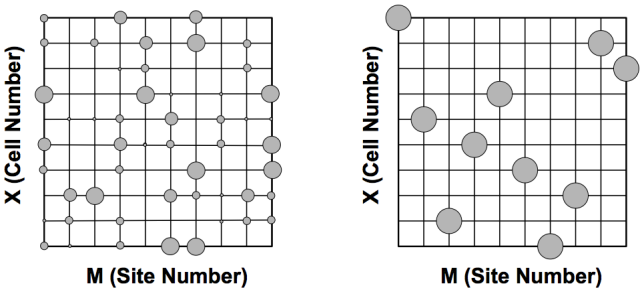

Figure 2. *Left:The neurons in the HNN are initialized to random activation levels. Right: The HNN converges so at most one neuron is active per row (cell) and per column (site)*

routing cost of a chip design.

2) From PARSEC Code to NN: The simulated annealing based cell-placement algorithm performs random swaps of cells and calculates the difference in wire length. Typically, swaps that result in shorter wire length are kept. However, simulated annealing also allows a portion of "bad swaps" to take place, helping the algorithm escape local minima solutions. The number of "bad swaps" allowed decays over time via a *temperature* variable. The input to the benchmark is a netlist of variable size (tens to thousands of nets), and the output is a chip layout where each cell is placed at a single location and the total wire length is at least a local minima.

Since the goal of this benchmark is to optimize the placement of cells, a Hopfield Neural Network (HNN) is chosen due to its success at solving various optimization problems including layout and placement problems [22], [23]. While this section describes at a very high level the use of HNNs to solve the cell placement problem, the next section briefly describes some of the important algorithmic details of HNNs.

There are several steps necessary in order to create a HNN capable of finding a minimal routing based on a given netlist. If we consider that there are X-cells in the netlist to be placed on M-sites, the HNN will contain $X \times M$ neurons. The synaptic weights between these neurons is established in such a way that the network converges based on the constraints of the optimization problem. These constraints include placing a cell only once, placing only one cell per site, and minimizing the routing overhead between cells [23]. The $X \times M$ neurons are initialized with a random activation level (Figure 2 left), and after execution, will converge on a state where only one neuron is on per row (each cell is placed once in each of the X rows) and at most there is one neuron on per column (only one cell is placed in each site), as seen in Figure 2 on the right.

There are two challenges relating to HNNs for this particular problem. The first challenge is, depending on the initial activation levels of the HNN, solutions may become stuck at local minima. For HNNs, the typical solution to this problem is to simply initialize many HNNs with different random initial values and choose the best solution after convergence, and we utilize this approach to select the best initial candidate solution. Afterwards, we perturb a small number of the neurons (typically near 10%) to find a more optimal solution based on this candidate network. While this does not guarantee that the system will reach the global minima, it does help the system converge on better solutions.

The second challenge relates to the scalability of HNNs, since typically HNNs are fully connected; that is, in a network of $X \times M$ neurons, there are a total of $(X \times M)^2$ synaptic weights. When netlists contain over 100,000 cells, it quickly becomes infeasible to use a single large network to solve this problem (at least from a software NN perspective). To address this issue, we create a hierarchy of HNNs which can each compute a partial solution to the layout problem in parallel. In this hierarchy, the first level divides the full netlist into more reasonably sized chunks (macro-cells) that will not exhaust memory resources. For example, this chunking scheme would use an HNN to solve the layout of cells 0-24, while another HNN solves 25-49, and so on. The second level, in parallel, will solve the placement of these first level macro-cells in relation to each other. That is, the "netlist" of the second level considers the number of connections between macro-cell 0 (cells 0-24), macro-cell 1 (cells 25-49), and so on. The size of the macro-cells are input parameters to the HNN. The number of levels necessary depends on the netlist size and the size of available memory to the system (i.e. larger macro-cells means less hierarchical levels, but more memory requirements).

3) NN Algorithm: HNNs are typically fully connected networks (i.e. every neuron connects to every other neuron) and often use continuous activation functions to solve optimization problems. A typical activation function is described below:

$$V = \frac{1}{2}(1 + tanh(\frac{U}{U_0})) \quad (1)$$

Net Size	Canneal Wire Length	HNN Wire Length	HNN Levels
10	132	68	1
100	4365	3011	2
100K	9.27e+07	8.77e+07	4
200K	2.49e+08	2.44e+08	4

Table III
Average wire length as calculated by PARSEC's `canneal` *and HNN.*

Net Size	Slowdown
10	1x
100	1x
100K	87x
200K	97x

Table IV
Execution time slowdown for the HNN implementation of PARSEC `canneal`.

Here, V is the activation level of the neuron, U is the input to the neuron, and U_0 is a constant. Hopfield networks evolve over time to minimize an energy function as described below:

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}V_iV_j - \sum_{i=1}^{N} I_iV_i \qquad (2)$$

Here, N is the number of neurons, w_{ij} is the synaptic weight between the output of neuron j and its input i, I_i is the external input to neuron i, and V_i is the activation levels of neurons i. By minimizing this energy function, HNNs are able to solve optimization problems [22], [23] or act as a content addressable memory capable of recalling stored patterns [24].

4) Evaluation: In PARSEC, the inputs to the canneal benchmark are netlists of various sizes, and for the HNN implementation of the placement problem, we use the same inputs. For small netlists (less than 100 cells), a single HNN can be used to solve the placement problem, while larger netlists require the hierarchical HNN scheme described above. The netlist containing 100,000 cells (or more) utilizes a four level HNN, where each of these smaller HNNs can be solved in parallel.

Accuracy. Table III shows the average wire length calculated by the default canneal implementation compared to the wire length calculated by the HNN. The final layouts determined by the HNNs are typically on par or better than the solutions provided by canneal.

Slowdown. As can be seen in Table IV, the slowdown for small problems is negligible, but it is significant for large problems. However, the hierarchical approach of Section II-B2 can be leveraged to break down the problem into smaller sizes compatible with a hardware accelerator.

C. Content Similarity (`ferret`)

1) Problem Description: Content similarity consists of finding one or several objects matching an input object. The object (content) can be of many types, e.g., image, audio, video, genomics data, etc. The Ferret content similarity toolkit [25] was developed at Princeton, and it has been used for implementing several search applications, including searching through continuously archived video [26]. In PARSEC, the benchmark is used for stationary image similarity.

One of the main difficulties of image similarity is defining *similarity*. `ferret` uses a notion of image similarity biased towards color moments (color characteristics of the image) and bounding boxes as well as segments sizes (scales of the features). As a result, two very different, but say, largely red, images could be classified as similar. The more commonly admitted notion of similarity is related to the more abstract nature of an object [27], e.g., deciding that an image contains a "cat", whatever its color, size, etc.; we use that notion for our experiments, and we apply it to the task of classifying images as being human faces or not; human faces images are a subset of the database provided with `ferret`.

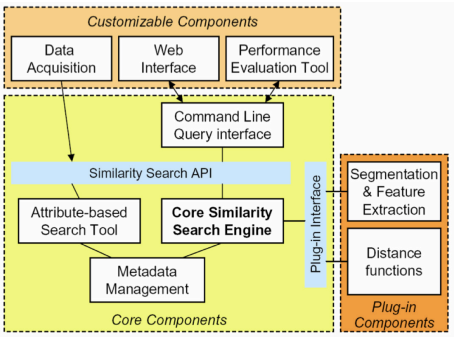

Figure 3. *Structure of the Ferret toolkit [25].*

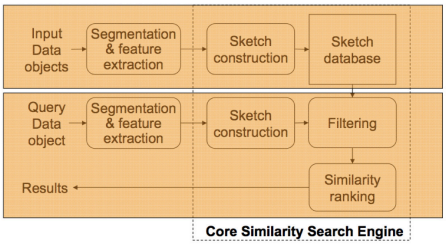

Figure 4. *Core similarity search engine.*

2) From PARSEC Code to NN: The structure of the original Ferret toolkit is shown in Figure 3. The Ferret toolkit was designed to build a modular, expandable, and data independent core search engine. For our purpose, the most important part is the *core similarity search engine* (see Figure 4), which realizes two main functions: similarity

search for high-dimensional data objects and constructing highly compact data structures (i.e., `sketches`, for feature vectors and for filtering and ranking) [25].

Much like neural networks, content similarity consists of a training (database construction) and testing (query) phases, see Figure 4. In both phases, input objects are first processed by a domain specific segmentation and feature extraction unit. Each data object is converted into a set of feature vectors. These vectors are then "compressed" into a compact bit vector (the sketch), which is then either stored in the database (construction) or compared against database elements (query), depending on the phase.

A query consists of comparing an input sketch against database sketches with a user-supplied distance function. The nature of the features and the ability to quickly estimate the distance between two sketches are key aspects of the content similarity process. We replace both steps with NNs, one with fixed weights (feature extraction for sketch construction), and the other with supervised learning (sketch distance). The training/testing structure of the benchmark is a natural fit for NNs.

3) NN Algorithm: The feature extraction is performed using a set of 2160 Gabor filters which decompose an image into elementary segments with scale and orientation sensitivity, effectively replacing sketches. Gabor filters can be implemented using a small set of neurons [28].

The similarity itself is implemented using a Multi-Layer Perceptron (MLP), where the input to the NN is the output of the Gabor filters. Images of the database are labeled with an abstract category, and the output of the MLP are the different possible categories. Since we look for similarities with only one image category ("faces"), the MLP has a single output which indicates the distance to the "faces" category; more categories and outputs could be easily implemented. The MLP has been introduced in Section II-A3, and the one used for this application only differs in its parameterization: 100 hidden neurons, 2160 inputs.

4) Evaluation:

Accuracy. We train the MLP using 256 images of various categories ("face", "lake", "misc") using back-propagation for at least 200 epochs. The training images are a selected subset of the 34,973 "large" images in the `ferret` data set. The remaining 34,205 "large" images are used for testing. Instead of expressing the results as an average distance, which is hard to interpret, we report it as a classification accuracy. The original PARSEC benchmark classifies the set of images with an accuracy of 88%, while our NN-based version classifies with an accuracy of 93%.

Compared to the approach used in `ferret`, the proposed neural network approach has the disadvantage of being less flexible when the category changes: in `ferret`, it merely consists of picking a different set of pre-computed reference sketches. Alternatively, the neural network must be trained on the new category. On the other hand, the neural network has the advantage of recognizing that a new image belongs to a known class (e.g. "`face`") directly, without comparing it with several/many elements in the database. Moreover, Le et al. [29] have recently shown that neural networks with unsupervised learning can categorize automatically large databases of images automatically. For the only task of comparing an input sketch against database sketches, a Radial basis function neural network (RBF NN) is well appropriate, with very efficient hardware implementation, as shown, for example, by the CM1K chip of CogniMem [30].

Slowdown. The ratio of the average execution time for processing one input in `ferret` (sketch construction, distance computation with sketches of the database) over the average execution time for processing one input with the neural network (Gabor filters + MLP) is 2x.

D. On-Line Clustering (`streamcluster`)

1) Problem Description: `streamcluster` is an online clustering program in the PARSEC benchmark suite. It classifies the input data into several groups so each group shares similar features. In order to tackle the large data invoked by the stream application, `streamcluster` follows a divide-and-conquer strategy, as illustrated in Figure 5. To be specific, it first divides the input data into several chunks, each of which is analyzed on one thread using an improved k-median clustering algorithm. After that, each thread returns the centers of the clusters obtained locally, and the centers obtained by all threads are put together. By clustering the centers returned by all threads, `streamcluster` presents the centers of the final clusters. Given the cluster centers, each example can be efficiently classified to the cluster whose center is the nearest to the example in the feature space.

2) From PARSEC Code to NN: `streamcluster` is used for inputs with a high number of dimensions. The role of clustering is implicitly twofold: to reduce the data dimensionality and then to cluster together the low-dimensional data. The most time-consuming task, by far, is the dimension reduction (89% of the original `streamcluster` execution time). Self-Organizing Maps (SOMs) are unsupervised artificial neural networks which can perform dimension reduction efficiently. A notable feature of SOMs is that they preserve the neighborhood information in dimension reduction, which is critical for clustering. As a result, we modify the data flow graph of `streamcluster`, as shown in Figure 5, by plugging SOMs routines before the k-means clustering step in each thread. The input to the SOM is simply the high-dimensional data already used by `streamcluster`.

3) NN Algorithm: A SOM can be viewed as a grid of neurons, as shown in Figure 6. The initial weight vectors of all neurons are determined either randomly or using some parameter tuning techniques. Then, they are iteratively updated. At each iteration, an input is fed to the SOM, and its distance to all weight vectors are computed, and the

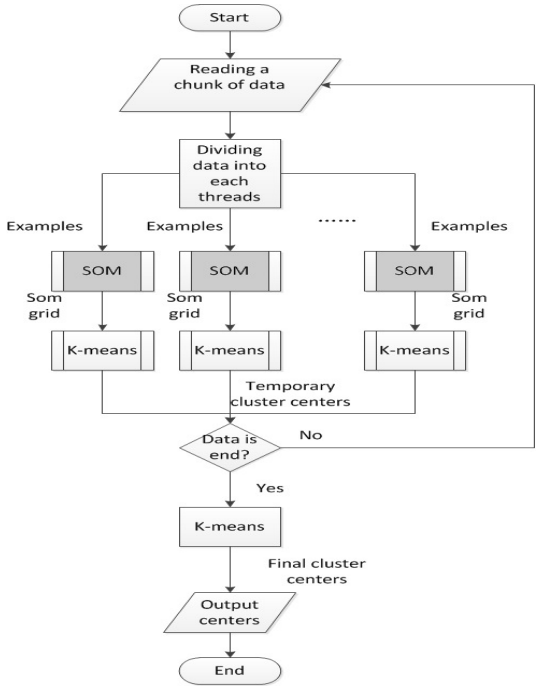

Figure 5. *Dataflow graph of streamcluster.*

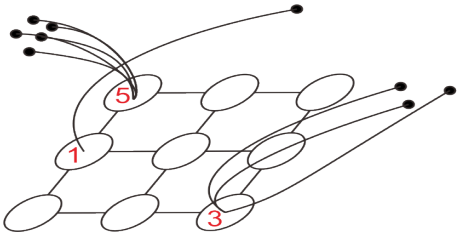

Figure 6. *Dimension reduction using a Self-Organizing Map (black dots are high-dimensional data, hollow circles are SOM neurons).*

neuron whose weight vector is the closest to the example is selected as the Best Matching Unit (BMU). The weight vector of the BMU, as well as the weight vectors of the neurons within the neighborhood of the BMU, are updated incrementally. Let $W(t)$ be the weight vector of a neuron at the t-th iteration ($t = 0, 1, 2, \ldots$). Let $I(t)$ be the input that is fed to the SOM at the t-th training iteration. The weight vector at the $(t+1)$-th iteration is given by $W(t+1) = W(t) + \theta(t)\alpha(t)(I(t) - W(t))$, where α(t) and $\theta(t)$ are the learning rate and neighborhood function at the t-th iteration, respectively.

As shown in Figure 6, a SOM maps high-dimensional data onto neurons, and neighboring data in the high-dimensional space remain neighbors on the SOM grid.

4) Evaluation:

Accuracy. Data points are labeled so the clustering accuracy can be defined as the fraction of correctly clustered data points.

The SOM parameters are as follows:

- Number of grid rows: $row = \frac{\sqrt{size}}{2}$
- Number of grid columns: $col = \frac{\sqrt{size}}{2}$
- Number of training iterations: $row \times col \times 500$
- Learning rates (two-stage SOM grid): 0.5, 0.05
- The radius of the neighbor function: $\frac{row+col}{4}$

Size is the chunk size passed to each thread (which varies between 16 and 50 in our experiments).

We use the following 3 data sets (2 real-world data sets, one randomly generated data set) to compare the accuracy of the original `streamcluster` to our NN implementation:

- **Iris**. A UCI (machine-learning repository) data set with 100 data points per step (total number of points: 150; dimension of each data point: 4), 2 threads.
- **Wine**. Another UCI data set with 50 data points per step (total number of points: 178; dimension of each data point: 13), 3 threads.
- **Random**. Points drawn using random normal distributions with 100 points per step (total number of points: 1000; dimension of each data point: 10), 2 threads.

Input \ Version	streamcluster	streamcluster + SOM
Iris	88.67%	93.33%
Wine	66.85%	93.26%
Random	95.60%	95.40%

Table V
Clustering accuracy.

Table V compares the accuracy of both implementations. For random data, both versions perform similarly, and for real data, the version with the SOM actually outperforms the PARSEC benchmark. Note that the random points are generated by several random normal distributions (PARSEC implementation). As a result, they end up forming natural clusters, corresponding to these distributions. Such trivially clustered data may not significantly exercise clustering techniques, hence the comparable accuracy results for this data set.

Input	Slowdown
Iris	0.7x
Wine	2.1x
Random	12.2x

Table VI
Execution time slowdown for the NN implementation of PARSEC `streamcluster`.

Slowdown. In Table VI, we show the slowdown of the NN implementation of `streamcluster`. The slowdown is higher (12.2x) for the largest data set (Random, largest lattice size) because the software version of SOM is sequential, and its performance decreases as the lattice size increases.

E. Hashing and Compression (`dedup`)

1) Problem Description: `dedup` is a data compression application which combines data-deduplication with Ziv-Lempel to achieve high compression ratios. To compress a

file, dedup processes it through different pipelined stages. In the first stage, the program breaks the input file into coarse-grained chunks that can be processed in parallel. In the second stage, each of the coarse-grained chunks is divided into fragments. Third, each of the unique smaller fragments is assigned a unique hash value. The fourth stage builds a global database of fragments indexed via the hash value. If a fragment has not been encountered before, it is compressed using Ziv-Lempel algorithm and is added to the database. The final stage generates the output file that consists of compressed fragments and hash values such that each of the compressed fragments occurs exactly once in the output file.

2) From PARSEC Code to NN: We have used a neural networks to replace four out of the five stages of dedup: fragmentation, hashing, building of the global database and compression.

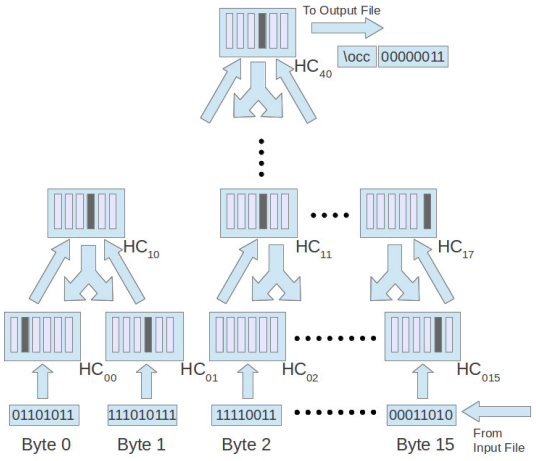

Figure 7. *NN organization for* dedup. HC_{00} = *Hypercolumn*$_0$ *in Level*$_0$

To compress a file by chunks of 16 bytes, we use an unsupervised neural network with 16 inputs and 256 outputs (see Figure 7). The 16 inputs correspond to 16 bytes of the file, while the ID of the 256 output neurons serves as signatures to be written to the output file. The input file is read 1 byte at a time and these bytes are pushed into a 16-byte long queue. If the network does not recognize the current 16-byte pattern in the queue, it is trained on the current pattern. The oldest byte in the queue is popped out of the queue is placed in the compression buffer. Finally, a new byte is read and pushed into the queue. If a pattern is recognized, there is no need to write it to the output file. In such a case, one of the output neurons reaches a threshold value and fires. The ID of the firing neuron preceded by an *occurrence* flag is marked to be placed into the compression buffer. The queue is then flushed and the next byte is read from the input file.

Each of the output neurons keeps track of the number of times it signaled a recognized pattern. This allows the network to keep track of the signatures that occur frequently in the input file. The network also includes a mechanism for forgetting infrequently occurring patterns, which ensures that network capacity is not wasted on retaining infrequently occurring signatures. If a pattern is deemed to be forgotten, but had been encountered more than once in the input file, then a special *training* byte followed by the 16-byte pattern is added to the appropriate location in the compression buffer, so that, during decompression, that pattern can be relearned when appropriate.

It should be noted that in the compressed buffer (the input to the compression stage), each of the 16-byte signatures that occurred more than once in the input file are preceded by the one byte *training* flag and the index of the output neuron that learned that signature. If a 16-byte signature occurs only once in the input file, it is not preceded by any such flag. Furthermore, all the multiple occurrences of a 16-byte signature learned by the network are written as a one byte *occurrence* flag followed by the index of the corresponding output neuron.

Once the entire input file is processed, the contents in the compression buffer are compressed using a neural network based algorithm. This neural network based compression algorithm replaces the Ziv-Lempel compression stage of the PARSEC dedup benchmark.

Using similar principles, we have also implemented the decompression algorithm using a neural networks.

3) NN Algorithms: Following is a brief description of the two neural network algorithms that we use to replace different stage of PARSEC dedup benchmark.

Hashing and Building Global Database. We use a hierarchical competitive hypercolumn/minicolumn network with unsupervised learning [31].

A hypercolumn contains multiple minicolumns which share the same receptive field (inputs). These minicolumns are connected to each other via inhibitory links. A minicolumn is said to fire if the correlation between its weights and the current input pattern in its receptive field exceeds a threshold. If the weights of a minicolumn are small, that minicolumn may also fire even when its inputs do no justify that firing (spontaneous activations). However, if a spontaneously active minicolumn is not inhibited by the neighboring minicolumns, it strengthens its weights to increase its correlation with the current input and eventually learns to recognize that input. Finally, the competitive learning behavior of the minicolumns ensures that every minicolumn within a hypercolumn learns a unique input signature.

We create a 5-level (Levels 0-4) hierarchy with 16 hypercolumns at Level 0. This network is organized in the form of a binary tree. The only hypercolumn at Level 4 is initialized with 256 minicolumns while the rest of the hypercolumns are initialized with 1024 minicolumns each. Presently, the receptive field of each of the hypercolumns is set to one byte of data and the threshold of the minicolumns

File Size	PARSEC CR	NN CR
5KB	2.13	1.77
10KB	2.27	1.94
100KB	2.33	2.51
1MB	1.74	2.00
5MB	1.2	1.26
11MB	1.08	1.11

Table VII
Compression Ratios (CR) of PARSEC `dedup` *vs. cortical column* `dedup`.

File Size	Slowdown
5KB	67x
10KB	115x
100KB	975x
1MB	10022x
5MB	38080x
11MB	65000x

Table VIII
Execution time slowdown of `dedup` *vs. cortical column version.*

is set such that a minicolumn fires only if there is a perfect match between its weights and the input. This imposes a constraint in terms of the maximum number of data bytes that a hierarchical hypercolumn network can simultaneously process, i.e., for the 5-level hierarchical network with 16 hypercolumns at Level 0, the maximum number of bytes simultaneously processed is 16. As a result, this network can identify 16-byte long signatures only. This shortcoming is overcome using multiple hierarchical hypercolumn networks with different numbers of hypercolumns at Level 0.

Compression. Schmidhuber et al. proposal for compression combines predictive neural networks with Huffman Coding to achieve compression rates better than Ziv-Lempel algorithm [32]. In their algorithm n characters become the input to the predictor neural network which emits an output P_{n+1}, an estimate for the $n+1$ character. Finally, the code for the predicted value is generated by feeding P_{n+1} into the Huffman coding algorithm.

4) Evaluation:

Accuracy. In Table VII, we compare the compression ratio of the original PARSEC benchmark against the version with cortical columns. Except for small (5KB) files, the compression ratio of the NN version is always better.

Slowdown. We report the slowdowns of the software NN version in Table VIII. As can be observed, the slowdowns are significant, so that a hardware NN accelerator may not be able to provide a competitive implementation in this case. While `dedup` is the only of our 5 benchmarks with such slowdowns, we plan to investigate alternative and more classic unsupervised neural networks with better execution time characteristics; so far, our criterion for selecting the NN algorithms was to achieve the best possible application-specific metric performance.

F. Other PARSEC Benchmarks

Beyond the 5 aforementioned benchmarks, we plan to progressively augment BenchNN with additional NN implementations of the PARSEC benchmarks.

- `bodytrack`. There are known state-of-the-art techniques for performing body tracking using neural networks [11].
- `swaptions`. Swaptions solves a problem very similar to `blackscholes` (option price prediction), except it uses a different method (Monte Carlo).
- `freqmine`. Frequent itemset miners are deterministic (not tolerant to errors) database operations. However, they are used to solve statistical problems which lend well to neural network implementations [14].
- `vips`. An image processing library based on fundamental transformations, such as convolutions and affine transformations, which can be implemented using neural networks as operators [15].
- `facesim`. One of the state-of-the-art techniques for emulating human movements (body motion) is based on neural networks [12], where they are used to efficiently interpolate between positions. The same technique can be extended to facial movements.
- `x264`. In future video encoding formats, e.g., NGVC, H265, adaptable transformations are expected to significantly improve the compression ratio, but they are also time-consuming [16]. Neural networks could provide an efficient method for learning these complex transformations.
- `fluidanimate`. It is possible to solve the Navier Stokes equation with a cellular neural network with sufficient accuracy for visual reconstruction [13].

III. Conclusions and Future Work

We have shown that for 5 PARSEC benchmarks, a suite considered to be representative of emerging high-performance applications, it is possible to substitute the core computational task with a neural network algorithm. For each benchmark, we have examined an application-specific metric characterizing the quality (usually accuracy) of the neural network alternative implementation. We have found that neural networks can achieve either slightly worse, comparable or even better solutions, which is in line with the objectives of a hardware neural network accelerator, especially for embedded systems applications, i.e., achieving very good but not necessarily always state-of-the-art accuracy.

Because the neural networks are implemented in software, their execution is usually not competitive time-wise, by one, and in a few cases, several orders of magnitude, though the massive parallelism of a hardware neural network implementation will likely make it competitive in most cases. The different neural network algorithms tested require a few tens to a few thousand neurons, numbers compatible in area with the silicon implementation of an accelerator [10]. Because several different neural network algorithms were used, a future challenge will be designing an accelerator capable of efficiently executing these different algorithms.

REFERENCES

[1] M. Muller, "Dark Silicon and the Internet," in *EE Times "Designing with ARM" virtual conference*, 2010. [Online]. Available: http://eetimes.com/virtualshows/ArmProgramSchedule

[2] H. Esmaeilzadeh, E. Blem, R. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceeding of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 365–376.

[3] S. Keckler, "Keynote: Life after dennard and how i learned to love the picojoule," in *International Symposium on Microarchitecture (MICRO)*, Sao Paolo, Brazil, 2011.

[4] S. Borkar, "Design perspectives on 22nm cmos and beyond," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, july 2009, pp. 93 –94.

[5] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, ser. FPGA '06. New York, NY, USA: ACM, 2006, pp. 21–30.

[6] S. Yehia, S. Girbal, H. Berry, and O. Temam, "Reconciling specialization and flexibility through compound circuits." in *HPCA*. IEEE Computer Society, 2009, pp. 277–288.

[7] 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.

[8] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *ACM/IEEE, International Symposium on Computer Architecture (ISCA)*, Portland, Oregon, June 2012.

[9] 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 *Proceeding of the IEEE Custom Integrated Circuits Conference*, 2011.

[10] J.-s. Seo, B. Brezzo, Y. Liu, B. D. Parker, S. K. Esser, R. K. Montoye, B. Rajendran, J. A. Tierno, L. Chang, D. S. Modha, and D. J. Friedman, "A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons," in *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, sept. 2011, pp. 1 –4.

[11] J. F. J. Fan, W. X. W. Xu, Y. W. Y. Wu, and Y. G. Y. Gong, "Human tracking using convolutional neural networks." *IEEE Transactions on Neural Networks*, vol. 21, no. 10, pp. 1610–1623, 2010.

[12] C. H. Ek, P. H. S. Torr, and N. D. Lawrence, "Gaussian process latent variable models for human pose estimation," in *MLMI*, ser. Lecture Notes in Computer Science, A. Popescu-Belis, S. Renals, and H. Bourlard, Eds., vol. 4892. Springer, 2007, pp. 132–143.

[13] B. Zineddin, Z. Wang, and X. Liu, "Cellular neural networks, the navier-stokes equation, and microarray image reconstruction," *IEEE Transactions on Image Processing*, vol. 20, no. 11, pp. 3296–3301, 2011.

[14] N. Ye, *The Handbook of Data Mining*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 2004.

[15] F. Yang and M. Paindavoine, "A new image filtering technique combining a wavelet transform with a linear neural network : Application to face recognition," *Optical Engineering*, vol. 39, pp. 2894–2899, 2000.

[16] O. Sezer, R. Cohen, and A. Vetro, "Robust learning of 2-d separable transforms for next-generation video coding," in *Data Compression Conference (DCC), 2011*, march 2011, pp. 63 –72.

[17] P. Dubey, "Recognition, mining and synthesis moves computers to the era of tera," *Technology@Intel Magazine*, Feb 2005.

[18] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81.

[19] 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.

[20] F. Black and M. Scholes, "The Pricing of Options and Corporate Liabilities," *The Journal of Political Economy*, vol. 81, no. 3, pp. 637–654, 1973.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Neurocomputing: foundations of research," J. A. Anderson and E. Rosenfeld, Eds. Cambridge, MA, USA: MIT Press, 1988, ch. Learning representations by back-propagating errors, pp. 696–699.

[22] J. J. Hopfield and D. W. Tank, "neural computation of decisions in optimization problems," *Biological Cybernetics*, vol. 52, pp. 141–152, 1985, 10.1007/BF00339943.

[23] A. Kos and Z. Nagorny, "Fpga placement by using hopfield neural network." in *Microelectronics International*. Emerald Group Publishing Limited, 2009, pp. 22–32.

[24] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, Apr. 1982.

[25] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Ferret: a toolkit for content-based similarity search of feature-rich data," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 317–330, Apr. 2006.

[26] Z. Wang, M. D. Hoffman, P. R. Cook, and K. Li, "Vferret: Content-based similarity search tool for continuous archived video," in *in CARPE Third ACM workshop on Capture, Archival and Retrieval of Personal Experiences*, 2006.

[27] A. Mojsilovic and B. Rogowitz, "Capturing image semantics with low-level descriptors," in *International Conference on Image Processing*, vol. 1, 2001, pp. 18 –21 vol.1.

[28] D. Ringach, "Haphazard wiring of simple receptive fields and orientation columns in visual cortex." *J. Neurophysiol.*, vol. 92, no. 1, pp. 468–476, Jul 2004.

[29] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng, "Building high-level features using large scale unsupervised learning," in *International Conference in Machine Learning*, 2012.

[30] I. Cognimem Technologies, "Cognimem cm1k chip," http://www.cognimem.com/products/chips-and-modules/CM1K-Chip/index.html, 2012, retrieved: May, 2012.

[31] A. Hashmi and M. Lipasti, "Discovering cortical algorithms," in *Proceedings of the 14th International Conference on Cognitive and Neural Systems (ICCNS-14)*, 2010.

[32] J. Schmidhuber and S. Heil, "Sequential neural text compression," *IEEE Transactions on Neural Network*, vol. 7, no. 1, pp. 142–146, 1996.