DaDianNao: A Machine-Learning Supercomputer

Yunji Chen\textsuperscript{1}, Tao Luo\textsuperscript{1,3}, Shaoli Liu\textsuperscript{1}, Shijin Zhang\textsuperscript{1}, Liqiang He\textsuperscript{2,4}, Jia Wang\textsuperscript{1}, Ling Li\textsuperscript{1}, Tianshi Chen\textsuperscript{1}, Zhiwei Xu\textsuperscript{1}, Ninghui Sun\textsuperscript{1}, Olivier Temam\textsuperscript{2}

\textsuperscript{1} SKL of Computer Architecture, ICT, CAS, China
\textsuperscript{2} Inria, Scalay, France
\textsuperscript{3} University of CAS, China
\textsuperscript{4} Inner Mongolia University, China

Abstract—Many companies are deploying services, either for consumers or industry, which are largely based on machine-learning algorithms for sophisticated processing of large amounts of data. The state-of-the-art and most popular such machine-learning algorithms are Convolutional and Deep Neural Networks (CNNs and DNNs), which are known to be both computationally and memory intensive. A number of neural network accelerators have been recently proposed which can offer high computational capacity/area ratio, but which remain hampered by memory accesses.

However, unlike the memory wall faced by processors on general-purpose workloads, the CNNs and DNNs memory footprint, while large, is not beyond the capability of the on-chip storage of a multi-chip system. This property, combined with the CNN/DNN algorithmic characteristics, can lead to high internal bandwidth and low external communications, which can in turn enable high-degree parallelism at a reasonable area cost. In this article, we introduce a custom multi-chip machine-learning architecture along those lines. We show that, on a subset of the largest known neural network layers, it is possible to achieve a speedup of 450.65x over a GPU, and reduce the energy by 150.31x on average for a 64-chip system. We implement the node down to the place and route at 28nm, containing a combination of custom storage and computational units, with industry-grade interconnects.

I. INTRODUCTION

Machine-Learning algorithms have become ubiquitous in a very broad range of applications and cloud services; examples include speech recognition, e.g., Siri or Google Now, click-through prediction for placing ads [27], face identification in Apple iPhoto or Google Picasa, robotics [20], pharmaceutical research [9] and so on. It is probably not exaggerated to say that machine-learning applications are in the process of displacing scientific computing as the major driver for high-performance computing. Early symptoms of this transformation are Intel calling for a refocus on Recognition, Mining and Synthesis applications in 2005 [14] (which later led to the PARSEC benchmark suite [3]), with Recognition and Mining largely corresponding to machine-learning tasks, or IBM developing the Watson supercomputer, illustrated with the Jeopardy game in 2011 [19].

Remarkably enough, at the same time this profound shift in applications is occurring, two simultaneous, albeit apparently unrelated, transformations are occurring in the machine-learning and in the hardware domains. Our community is well aware of the trend towards heterogeneous computing where architecture specialization is seen as a promising path to achieve high performance at low energy [21], provided we can find ways to reconcile architecture specialization and flexibility. At the same time, the machine-learning domain has profoundly evolved since 2006, where a category of algorithms, called Deep Learning (Convolutional and Deep Neural Networks), has emerged as state-of-the-art across a broad range of applications [33], [28], [32], [34]. In other words, at the time where architects need to find a good tradeoff between flexibility and efficiency, it turns out that just one category of algorithms can be used to implement a broad range of applications. In other words, there is a fairly unique opportunity to design highly specialized, and thus highly efficient, hardware which will benefit many of these emerging high-performance applications.

A few research groups have started to take advantage of this special context to design accelerators meant to be integrated into heterogeneous multi-cores. Temam [47] proposed a neural network accelerator for multi-layer perceptrons, though it is not a deep learning neural network, Esmaeilzadeh et al. [16] propose to use a hardware neural network called NPU for approximating any program function, though not specifically for machine-learning applications, Chen et al. [5] proposed an accelerator for Deep Learning (CNNs and DNNs). However, all these accelerators have significant neural network size limitations: either small neural networks of a few tens of neurons can be executed, or the neurons and synapses (i.e., weights of connections between neurons) intermediate values have to be stored in main memory. These two limitations are severe, respectively from a machine-learning or a hardware perspective.

From a machine-learning perspective, there is a significant trend towards increasingly large neural networks. The recent work of Krizhevsky et al. [32] achieved state-of-the-art accuracy on the ImageNet database [13] with “only” 60
million parameters. There are recent examples of a 1-billion parameter neural network [34], and some of the same authors even investigated a 10-billion neural network the following year [8]. However, these networks are for now considered extreme experiments in unsupervised learning (the first one on 16,000 CPUs, the second one on 64 GPUs), and they are outperformed by smaller but more classic neural networks such as the one by Krizhevsky et al. [32]. Still, while the neural network size progression is unlikely to be monotonic, there is a definite trend towards larger neural networks. Moreover, increasingly large inputs (e.g., HD instead of SD images) will further inflate the neural networks sizes. From a hardware perspective, the aforementioned accelerators are limited because if most synaptic weights have to reside in main memory, and if neurons intermediate values have to be frequently written back and read from memory, the memory accesses become the performance bottleneck, just like in processors, partly voiding the benefit of using custom architectures. Chen et al. [5] acknowledge this issue by observing that their neural network accelerator loses at least an order of magnitude in performance due to memory accesses.

However, while 1 billion parameters or more may come across as a large number from a machine-learning perspective, it is important to realize that, in fact, it is not from a hardware perspective: if each parameter requires 64 bits, that only corresponds to 8 GB (and there are clear indications that fewer bits are sufficient). While 8 GB is still too large for a single chip, it is possible to imagine a dedicated machine-learning computer composed of multiple chips, each chip containing specialized logic together with enough RAM that the sum of the RAM of all chips can contain the whole neural network, requiring no main memory. By tightly interconnecting these different chips through a dedicated mesh, one could implement the largest existing DNNs, achieve high performance at a fraction of the energy and area of the many CPUs or GPUs used so far. Due to its low energy and area costs, such a machine, a kind of compact machine-learning supercomputer, could help spread the use of high-accuracy machine-learning applications, or conversely to use even larger DNNs/CNNs by simply scaling up RAM storage at each node and/or the number of nodes.

In this article, we present such an architecture, composed of interconnected nodes, each containing computational logic, eDRAM, and the router fabric; the node is implemented down to the place and route at 28nm, and we evaluate an architecture with up to 64 nodes. On a sample of the largest existing neural network layers, we show that it is possible to achieve a speedup of 450,65x over a GPU and to reduce energy by 150.31x on average.

In Section II, we introduce CNNs and DNNs, in Section III, we evaluate such NNs on GPU, in Section IV we compare GPU and a recently proposed accelerator for CNNs and DNNs, in Section V, we introduce the machine-learning supercomputer, we present the methodology in Section VI, the experimental results in Section VII and the related work in Section VIII.

II. STATE-OF-THE-ART MACHINE-LEARNING TECHNIQUES

The state-of-the-art and most popular machine-learning algorithms are Convolutional Neural Networks (CNNs) [35] and Deep Neural Networks (DNNs) [9]. Beyond early differences in training, the two types of networks are also distinguished by their implementation of convolutional layers detailed thereafter. CNNs are particularly efficient for image applications and any application which can benefit from the implicit translation invariance properties of their convolutional layers. DNNs are more complex neural networks but they have an even broader application span such as speech recognition [9], web search [27], etc.

A. Main Layer Types

A CNN or a DNN is a sequence of multiple instances of four types of layers: pooling layers (POOL), convolutional layers (CONV), classifier layers (CLASS), and local response normalization layers (LRN), see Figure 1. Usually, groups of convolutional, local response normalization and pooling layers alternate, while classifier layers are found at the end of the sequence, i.e., at the top of the neural network hierarchy. We present a simple hierarchy in Figure 1; we illustrate the intuitive task performed at the top, and we provide the formal computations performed by the layer at the bottom.

Convolutional layers (CONV). Intuitively, a convolutional layer implements a set of filters to identify characteristic elements of the input data, e.g., an image, see Figure 1. For visual data, a filter is defined by \( K_x \times K_y \) coefficients forming a kernel; these kernel coefficients are learned and form the layer synaptic weights. Each convolutional layer slides \( N_{of} \) such filters through the whole input layer (by steps of \( s_x \) and \( s_y \)), resulting in as many \( (N_{of}) \) output feature maps.

The concrete formula for calculating an output neuron \( a(x, y)^{f_o} \) at position \( (x, y) \) of output feature map \( f_o \) is

\[
out(x, y)^{f_o} = \sum_{i=0}^{N_{if}} \sum_{k_x=0}^{K_x} \sum_{k_y=0}^{K_y} w_{f_i, f_o}(k_x, k_y) \cdot in(x+k_x, y+k_y)^i,
\]

where \( in(x, y)^i \) (resp. \( out() \)) represents the input (resp. output) neuron activity at position \( (x, y) \) in feature map \( f_i \) and \( w_{f_i, f_o}(k_x, k_y) \) is the synaptic weight at kernel position \( (k_x, k_y) \) in input feature map \( f_i \) for filter (output feature map) \( f_o \). Since the input layer itself may contain multiple feature maps \( N_{if} \) input feature maps), the kernel is usually three-dimensional, i.e., \( K_x \times K_y \times N_{if} \).

In DNNs, the kernels usually have different synaptic values for each output neuron (at each \( (x, y) \) position), while...
in CNNs, the kernels are shared across all neurons of the same output feature map. Convolutional layers with private (non-shared) kernels have drastically more synaptic weights (i.e., parameters) than the ones with shared kernels ($K \times K \times N_{if} \times N_{of} \times N_x \times N_y$ vs. $K \times K \times N_{if} \times N_{of}$, where $N_x$ and $N_y$ are the input layer dimensions).

Pooling layers (POOL). A pooling layer computes the max or average over a number of neighbor points, e.g.,

$$\text{out}(x,y)^f = \max_{0 \leq k_x \leq K_x, 0 \leq k_y \leq K_y} \text{in}(x + k_x, y + k_y)^f$$

Its effect is to reduce the input layer dimensionality, which allows coarse-grain (larger scale) features to emerge, see Figure 1, and be later identified by filters in the next convolutional layers. Unlike a convolutional or a classifier layer, a pooling layer has no learned parameter (no synaptic weight).

Local response normalization layers (LRN). Local response normalization implements competition between neurons at the same location, but in different (neighbor) feature maps. Krizhevsky et al. [32] postulate that their effect is similar to the lateral inhibition found in biological neurons. The computations are as follows

$$\text{out}(x,y)^f = \text{in}(x,y)^f \left( 1 + \sum_{g=\max(0,f-k/2)}^{\min(N_f-1,f+k/2)} (a(x,y)^g)^2 \right)^{\beta}$$

where $k$ determines the number of adjacent feature maps considered, and $c, \alpha$ and $\beta$ are constants.

Classifier layers (CLASS). The result of the sequence of CONV, POOL and LRN layers is then fed to one or multiple classifier layers. This layer is typically fully connected to its $N_i$ inputs (and it has $N_o$ outputs), see Figure 1, and each connection carries a learned synaptic weight. While the number of inputs may be much lower than for other layers (due to the dimensionality reduction of pooling layers), they can account for a large share of all synaptic weights in the neural network due to their full connectivity. Multi-Layer perceptrons are frequently used as classifier layers, though other types of classifiers are used as well (e.g., multinomial logistic regression). The goal of these layers is naturally to correlate the different features extracted from the filtering, normalization and pooling steps and the output categories.

$$\text{out}(j) = t \left( \sum_{i=0}^{N_i} w_{ij} + \text{in}(i) \right)$$

where $t()$ is a transfer function, e.g., $\frac{1}{1+x^2}$, $\text{tanh}(x)$, $\text{max}(0,x)$ for ReLU [32], etc.

B. Benchmarks

Throughout this article, we use as benchmarks a sample of 10 of the largest known layers of each type, described in Table I, as well as a full neural network (CNN), winner of the ImageNet 2012 competition [32]. The full NN benchmark contains the following 12 layers (the format is $N_x, N_y, K_x, K_y, N_i$ or $N_{if}, N_o$ or $N_{of}$ as in the table): CONV (224,224,11,11,3,96), LRN (55,55,5,5,96,96), POOL (55,55,3,3,96,96), CONV (27,27,5,5,96,256), LRN (27,27,5,5,256,256), POOL (27,27,3,3,256,256), CONV (13,13,3,3,256,384), CONV (13,13,3,3,384,384), CONV (13,13,3,3,384,256), CLASS (-,-,-,9216,4096), CLASS (-,-,-,4096,4096), CLASS (-,-,-,4096,1000). For all convolutional layers, the sliding window strides $s_x$, $s_y$ are 1, except for the first convolutional layer of the full NN, where they are 4. For all pooling layers, their sliding window strides equal to their kernel dimension, i.e. $s_x = K_x$, $s_y = K_y$. Note also that for LRN layers, $k = 5$. Finally, since we consider both inference and training for each layer, see Section II-C, we have also considered the most popular pre-training method, i.e., the method used to initialize the synaptic weights, which is often time-consuming. This method is based on Restricted Boltzmann Machines (RBM) [45], and we applied it to CLASS1 and CLASS2 layers, leading to the RBM1 (2560×2560) and RBM2 (4096×4096) benchmarks.

C. Inference vs. Training

A frequent and important misconception about neural networks is that on-line learning (a.k.a. training or backward...
phase) is necessary for many applications. On the contrary, for many industrial applications off-line learning is sufficient, where the neural network is first trained on a set of data, and then only used in inference (a.k.a. testing or feed-forward phase) mode by the end user. Note that even machine-learning researchers acknowledge this choice, as one of the few examples of hardware designs coming from that community is dedicated to inference [18]. While we put more emphasis in design and experiments on the much broader market of users of machine-learning algorithms, we have also designed the architecture to support the most common learning algorithms in order to also serve as an accelerator for machine-learning researchers and we also present experiments for that usage.

### III. THE GPU OPTION

Currently, the most favored approach for implementing CNNs and DNNs are GPUs [6] due to the fairly regular nature of these algorithms. We have implemented in CUDA the different layer types of Table I. We have also implemented a C++ version in order to obtain a CPU (SIMD) baseline. We have evaluated these versions on respectively a modern GPU card (NVIDIA K20M, 5GB GDDR5, 208 GB/s memory bandwidth, 3.52 TFlops peak, 28nm technology), and a 256-bit SIMD CPU (Intel Xeon E5-4620 Sandy Bridge-EP, 2.2GHz, 1TB memory); we report the speedups of GPU over CPU (for inference) in Figure 2. The GPU can provide a speedup of 58.8x over an Intel Core-i7 920 on CNNs. One can also observe that the GPU is particularly efficient on LRN layers because of the presence of a dedicated exponential instruction, a computation which accounts for most the LRN execution time on SIMD.

While these speedups are high, GPUs have a number of limitations. First, their (area) cost is high because of both the number of hardware operators and the need to remain reasonably general-purpose (memory hierarchy, all PEs are connected to some elements of the memory hierarchy, etc). Second, the total execution time remains large (up to 18.03 seconds for the largest layer CLASS1); this may not be compatible with the milliseconds response time required by web services or other industrial applications. Third, the GPU energy efficiency is moderate, with an average power of over 74.93W for the NVIDIA K20M GPU. That figure is actually optimistic because the NVIDIA K20M only contains 1.5MB of on-chip RAM, forcing frequent high-energy accesses to the off-chip GDDR5 memory leading to a thermal design power of 225W for the entire GPU board [43].

### IV. THE ACCELERATOR OPTION

Recently, Chen et al. [5] have proposed the DianNao accelerator for the fast and low-energy execution of the inference of large CNNs and DNNs in a small form factor (3mm² at 65nm, 0.98GHz). We reproduce the block diagram of DianNao in Figure 3. The architecture contains buffers for caching input/output neurons and synapses, and a Neural Functional Unit (NFU) which is largely a pipelined version of the typical computations required to evaluate a neuron output: the multiplication of synaptic values by input neurons values in the first stage, additions of all these products in the second stage (adder trees), and application of a transfer function in the third stage (realized through linear interpolation). Depending on the layer type (classifier, convolution, pooling), different computational operators are invoked in each stage.

In order to compare their architecture against GPU, we reimplement a cycle-level bit-level version of DianNao, and we use the memory latency parameters mentioned in their article. For the sake of comparison, we use at least some (4) of the same layers (CONV2, CONV4*, POOL1 and POOL2 respectively correspond to their CONV1, CONV5*, POOL1, POOL3; the layer numbers are different but the notations are different.

<table>
<thead>
<tr>
<th>Layer</th>
<th>$N_x$</th>
<th>$N_y$</th>
<th>$K_x$</th>
<th>$K_y$</th>
<th>$N_i$</th>
<th>$N_o$</th>
<th>Synapses Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2560</td>
<td>2560</td>
<td>12.5MB Object recognition and speech recognition tasks (DNN) [11].</td>
</tr>
<tr>
<td>CONV1</td>
<td>256</td>
<td>256</td>
<td>11</td>
<td>11</td>
<td>256</td>
<td>384</td>
<td>0.096MB Multi-Object recognition in natural images (DNN).</td>
</tr>
<tr>
<td>POOL1</td>
<td>256</td>
<td>256</td>
<td>2</td>
<td>2</td>
<td>256</td>
<td>256</td>
<td>0.086MB winner 2012 ImageNet</td>
</tr>
<tr>
<td>LRN1</td>
<td>55</td>
<td>55</td>
<td>-</td>
<td>-</td>
<td>96</td>
<td>96</td>
<td>0.006MB competition [32].</td>
</tr>
<tr>
<td>LRN2</td>
<td>27</td>
<td>27</td>
<td>-</td>
<td>-</td>
<td>256</td>
<td>256</td>
<td>0.006MB</td>
</tr>
<tr>
<td>CONV2</td>
<td>500</td>
<td>375</td>
<td>9</td>
<td>9</td>
<td>32</td>
<td>48</td>
<td>0.42MB Street scene parsing (CNN) (e.g., identifying building, vehicle, etc) [18].</td>
</tr>
<tr>
<td>POOL2</td>
<td>492</td>
<td>367</td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>12</td>
<td>-</td>
</tr>
<tr>
<td>CONV3*</td>
<td>200</td>
<td>200</td>
<td>18</td>
<td>18</td>
<td>8</td>
<td>8</td>
<td>1.29GB Face Detection in YouTube videos (DNN), (Google) [34].</td>
</tr>
<tr>
<td>CONV4*</td>
<td>200</td>
<td>200</td>
<td>20</td>
<td>20</td>
<td>3</td>
<td>18</td>
<td>1.32GB YouTube video object recognition, largest NN to date [8].</td>
</tr>
</tbody>
</table>

Table I: Some of the largest known CNN or DNN layers (CONV*, indicates convolutional layers with private kernels).
We use the classifier layer as a driving example; note that for the sake of completeness, we explain in Section V-B3 how all layers are implemented on the architecture. As explained in Section II, in a classifier layer, the $N_o$ outputs are typically connected to all the $N_i$ inputs, with one synaptic weight per connection. In terms of locality, it means that each input is reused $N_i$ times, and that the synaptic weights are not reused within one classifier layer execution.

B. Node

In this section, we present the architecture node and explain the rationale for its design.

1) Synapses Close to Neurons: One of the fundamental design characteristic of the proposed architecture is to locate the storage for synapses close to neurons and to make it massive. This design choice is motivated by the decision to
move only neurons and to keep synapses in a fixed storage location. This serves two purposes.

First, the architecture is targeted for both inference and training. In inference, the neurons of the previous layer are the inputs of the computation; in training, the neurons are forward-propagated (so neurons of the previous layer are the inputs) and then backward-propagated (so neurons of the next layer are now the inputs). As a result, depending on how data (neurons and synapses) are allocated to nodes, they need to be moved between the forward and backward phases. Since there are many more synapses than neurons (e.g., $O(N^2)$ vs. $O(N)$ for classifier layers, $K \times K \times N_{if} \times N_{oj} \times N_x \times N_y$ vs. $N_{if} \times N_x \times N_y$ for convolutional layers with private kernels, see Section II), it is only logical to move neuron outputs instead of synapses. Second, having all synapses (most of the computation inputs) next to computational operators provides low-energy/low-latency data (synapses) transfers and high internal bandwidth.

As shown in Table I, layer sizes can range from less than 1MB to about 1GB, most of them ranging in the tens of MB. While SRAMs are appropriate for caching purposes, they are not dense enough for such large-scale storage. However, eDRAMs are known to have a higher storage density. For instance, a 10MB SRAM memory requires 20.73mm$^2$ at 28nm [36], while an eDRAM memory of the same size and at the same technology node requires 7.27mm$^2$ [50], i.e., a 2.85x higher storage density.

Moreover, providing sufficient eDRAM capacity to hold all synapses on the combined eDRAM of all chips will save on off-chip DRAM accesses, which are particularly costly energy-wise. For instance, a read access to a 256-bit wide eDRAM array at 28nm consumes 0.0192nJ at 28nm [25], while a 256-bit read access to a Micron DDR3 DRAM consumes 6.18nJ at 32nm [40], i.e., an energy ratio of 321x. The ratio is largely due to the memory controller, the DDR3 physical-level interface, on-chip bus access, page activation, etc.

If the NFU is no longer limited by the memory bandwidth, it is possible to scale up its size in order to process more output neurons ($N_o$) and more inputs per output neuron ($N_i$) simultaneously, and thus, to improve the overall node throughput. For instance, to scale up by 16x the number of operations performed every cycle compared to the accelerator mentioned in Section IV, we need to have $N_i = 64$ (instead of 16) and $N_o = 64$ (instead of 16). In order to achieve maximal throughput, we must fetch $N_i \times N_o$ 16-bit values from the eDRAM to the NFU every cycle, i.e., $64 \times 64 = 65536$ bits in this case.

However eDRAM has three well-known drawbacks: higher latency than SRAM, destructive reads and periodic refresh [38], as in traditional DRAMs. In order to compensate for the eDRAM drawbacks and still feed the NFU every cycle, we split the eDRAM into four banks (65536-bit wide in the above example), and we interleave the synapses rows among the four banks.

We placed and routed this design at 28nm (ST technology, LP), and we obtained the floorplan of Figure 4. The NFU footprint is very small at 0.78mm$^2$ (0.88mm $\times$ 0.88mm), but the process imposes an average spacing of 0.2 $\mu$m between wires, and provides only 4 horizontal metal layers. As a result, the 65536 wires connecting the NFU to the eDRAM require a width of $\frac{65536 \times 0.2}{4} = 3.2768mm$, see Figure 4. Consequently, wires occupy $4 \times 3.2768 \times 3.2768 - 0.88 \times 0.88 = 42.18mm^2$, which is almost equal to the combined area of all eDRAM banks, all NFUs and the I/O.

2) **High Internal Bandwidth:** In order to avoid this congestion, we adopt a tile-based design, as shown in Figure 5. The output neurons are spread out in the different tiles, so that each NFU can simultaneously process 16 input neurons

![Figure 4: Simplified floorplan with a single central NFU showing wire congestion.](image)

![Figure 5: Tile-based organization of a node (left) and tile architecture (right). A node contains 16 tiles, two central eDRAM banks and fat tree interconnect; a tile has an NFU, four eDRAM banks and input/output interfaces to/from the central eDRAM banks.](image)

![Figure 6: The different (parallel) operators of an NFU: multipliers, adders, max, transfer function.](image)
of 16 output neurons (256 parallel operations), see Figure 6. As a result, the NFU in each tile is significantly smaller, and only $16 \times 16 = 4096$ bits must be extracted each cycle from the eDRAM. We keep the 4-bank (4096-bit wide banks) organization to compensate for the aforementioned eDRAM weaknesses, and we obtain the tile design of Figure 5. We placed and routed one such tile, and obtained an area of 1.89 mm$^2$, i.e., a 28.5% area reduction over the previous design, because the routing network now only accounts for 8.97% of the overall area.

All the tiles are connected through a fat tree which serves to broadcast the input neurons values to each tile, and to collect the output neurons values from each tile. At the center of the chip, there are two special eDRAM banks, one for input neurons, the other for output neurons. It is important to understand that, even with a large number of tiles and chips, the total number of hardware output neurons of all NFUs, can still be small compared to the actual number of neurons found in large layers. As a result, for each set of input neurons broadcasted to all tiles, multiple different output neurons are being computed on the same hardware neuron. The intermediate values of these neurons are saved back locally in the tile eDRAM. When the computation of an output neuron is finished (all input neurons have been factored in), the value is sent through the fat tree to the center of the chip to the corresponding (output neurons) central eDRAM bank.

3) **Configurability (Layers, Inference vs. Training):** We can adapt the tile, and the NFU pipeline in particular, to the different layers and the execution mode (inference or training). The NFU is decomposed into a number of hardware blocks: adder block (which can be configured either as a 256-input, 16-output adder tree, or 256 parallel adders), multiplier block (256 parallel multipliers), max block (16 parallel max operations), and transfer block (two independent sub-blocks performing 16 piecewise linear interpolations; the $a, b$ linear interpolation coefficients, i.e., $y = a \times x + b$, for each block are stored in two 16-entry SRAMs and can be configured to implement any transfer function and its derivative). In Figure 7, we show the different pipeline configurations for CONV, LRN, POOL and CLASS layers in the forward and backward phases.

![Figure 7: Different pipeline configurations for CONV, LRN, POOL and CLASS layers.](image)

<table>
<thead>
<tr>
<th>Inference</th>
<th>Training</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-Point</td>
<td>Floating-Point</td>
<td>0.82%</td>
</tr>
<tr>
<td>Fixed-Point (16 bits)</td>
<td>Floating-Point</td>
<td>0.83%</td>
</tr>
<tr>
<td>Fixed-Point (32 bits)</td>
<td>Floating-Point</td>
<td>0.83%</td>
</tr>
<tr>
<td>Fixed-Point (16 bits)</td>
<td>Fixed-Point (16 bits)</td>
<td>(no convergence)</td>
</tr>
<tr>
<td>Fixed-Point (16 bits)</td>
<td>Fixed-Point (32 bits)</td>
<td>0.91%</td>
</tr>
</tbody>
</table>

Table II: Impact of fixed-point computations on error.

Each hardware block is designed to allow the aggregation of 16-bit operators (adders, multipliers, max, and the adder-s/multipliers used for linear interpolation) into fewer 32-bit operators (two 16-bit adders into one 32-bit adder, four 16-bit multipliers into 32-bit multiplier, two 16-bit max into one 32-bit max); the overhead cost of aggregable operators is very low [26]. While 16-bit operators are largely sufficient for the inference usage, they may either reduce the accuracy and/or increase (or even prevent) the convergence of training. As an example, consider a CNN trained on MNIST [35] using various combinations of fixed and floating-point representations. There is almost no impact on error if 16-bit fixed-point is used in inference only, but there is no convergence if it is used also for training. On the other hand, there is only a small impact on error if 32-bit fixed-point is used: 0.91% instead of 0.83%; moreover, in further tests, we note that the error obtained for 28 bits is 1.72%, so it decreases rapidly to 0.91% by adding 4 more bits, and further aggregating operators allows to further decrease the fixed-point error.

By default, we use 32-bit operators in training mode.

Beyond pipeline and block configurations, the tile must be configured for different data movement cases. For instance, a classifier layer input can come from the node central eDRAM (possibly after transfer from another node), or it can come from the two SRAM storages (16KB) which are used to buffer input and output neuron values, or even temporary values (such as neurons partial sums to enable reuse of input neurons values) as proposed by Chen et al. [5]. In the backward phase, the NFU must also write to the tile eDRAM after the weights update step, see Figure 7. During the gradient computations step, the input and output gradients use the data paths of input and output neurons in the forward phase, see Figure 7 again.

C. **Interconnect**

Because neurons are the only values transferred, and because these values are heavily reused within each node, the amount of communications, while significant, is not a bottleneck except for a few layers and many-node systems, as later discussed in Section VII. As a result, we did not develop a custom high-speed interconnect for our purpose,
we turned to commercially available high-performance interfaces, and we used a HyperTransport (HT) 2.0 IP block. The HT2.0 physical layer interface (PHY) we used for the 28nm process is a long thin strip of 5.635mm × 0.5575mm (with a protrusion) due to its usual location at the periphery of the die.

We use a simple 2D mesh topology; that choice may be later revisited in favor of a more efficient 3D mesh topology though. Because of the mesh topology of the architecture, each chip must connect to four neighbors via four HT2.0 IP blocks (see Figure 9), each with 16x HT links, i.e., 16 pairs of differential outgoing signals, and 16 pairs of differential incoming signals, at a frequency of 1.6GHz (we connect the HT to the central eDRAM through a 128-bit, 4-entry, asynchronous FIFO). Each HT block provides a bandwidth of 6.4GB/s in each direction. The HT2.0 latency between two neighbor nodes is about 80ns.

**Router.** Next to the central block of the tile, we implement the router, see Figure 5. We use wormhole routing, the router has five input/output ports (4 directions and injection/ejection port). Each input port contains 8 virtual channels (5 fit slots per VC). A 5x5 crossbar is equipped to connect all input/output ports. The router has four pipeline stages: routing computation (RC), VC allocation (VA), switch allocation (SA) and switch traversal (ST).

### D. Overall Characteristics

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>606 MHz</td>
</tr>
<tr>
<td># of tiles</td>
<td>16</td>
</tr>
<tr>
<td># of 16-bit multipliers/tile</td>
<td>256+32</td>
</tr>
<tr>
<td># of 16-bit adders/tile</td>
<td>256+32</td>
</tr>
<tr>
<td>tile eDRAM size/tile</td>
<td>2 MB</td>
</tr>
<tr>
<td>central eDRAM size</td>
<td>4 MB</td>
</tr>
<tr>
<td>central eDRAM latency</td>
<td>~3 cycles</td>
</tr>
<tr>
<td>Link bandwidth</td>
<td>6.4x4GB/s</td>
</tr>
<tr>
<td>Link latency</td>
<td>80 ns</td>
</tr>
</tbody>
</table>

**Table III: Architecture characteristics.**

The architecture characteristics are summarized in Table III. We have implemented 16 tiles per node. In each tile, each of the 4 eDRAM banks contains 1024 rows of 4096 bits. The total eDRAM capacity in one tile is thus $4 \times 1024 = 4 \times 1024 = 2$ MB. The central eDRAM in each node has a size of 4 MB. The total node eDRAM capacity is thus $16 \times 2 + 4 = 36$ MB.

In order to avoid the circuit and time overhead of asynchronous transfers, we decided to clock the NFU at the same frequency as the eDRAM available in the 28nm technology we used, i.e., 606 MHz. Note that the NFU implemented by Chen et al. [5] was clocked at 0.98 GHz at 65 nm, so our decision is very conservative considering we use a 28 nm technology. We leave the implementation of a faster NFU and asynchronous communications with eDRAM for future work. Nonetheless, a node still has a peak performance of $16 \times (256 + 258) \times 606 = 5.58$ TeraOps/s for 16-bit operation. For 32-bit operation, the peak performance of a node is $16 \times (144 + 72) \times 606 = 2.09$ TeraOps/s due to operator aggregation, see Section V-B3.

**E. Programming, Code Generation and Multi-Node Mapping**

1) Programming, Control and Code Generation:

This architecture can be viewed as a system ASIC, so the programming requirements are low, the architecture essentially has to be configured and the input data is fed in. The input data (values of the input layer) is initially partitioned across nodes and stored in a central eDRAM bank. The neural network configuration is implemented in the form of a sequence of *node* instructions, one sequence per node, produced by a code generator. An example output of the code generator for the inference phase of the CLASS2 layer is shown in Table V.

In this example, output neurons are partitioned into multiple 256-bit data blocks, where each block contains 256/16 = 16 neurons. Each node is allocated 4096/16/4 = 64 output data blocks (and it stores a quarter of all input neurons, i.e., 4096/4 = 1024), and each tile is allocated 64/16 = 4 output data blocks, resulting in 4 instructions per node. An instruction will load 128 input data blocks from the central eDRAM to the tiles. In the first three instructions, all the tiles will get the same input neurons, and read synaptic weights from their local (tile) eDRAM, then write back the partial sums (of output neurons) to their local NBout SRAM. In the last instruction, the NFU in each tile will finalize the sums, apply the transfer function, and store the output values back to the central eDRAM.

These node instructions themselves drive the control of each tile; the control circuit of each node generates tile instructions and sends them to each tile. The spirit of a node or tile instruction is to perform the same layer computations (e.g., multiply-add-transfer for classifier layers) on a set of contiguous input data (input neurons in the forward phase, output neurons, gradients or synapses in the backward phase). The fact the data of one instruction is contiguous allows to characterize it with only three operands: start address, step and number of iterations.

The control provides two modes of operations: processing one row at a time or *batch learning* [48], where multiple
rows are processed at the same time, i.e., multiple instances of the same layer are evaluated simultaneously, albeit for different input data. This method is commonly used in machine-learning for a more stable gradient descent, and it also has the benefit of improving synapses reuse, at the cost of slower convergence and a larger memory capacity (since multiple instances of inputs/outputs must be stored).

2) Multi-Node Mapping: At the end of a layer, each node contains a set of output neurons values, which have been stored back in the central eDRAM, see Figure 5. These output neurons form the input neurons of the next layer; so, implicitly, at the beginning of a layer, the input neurons are distributed across all nodes, in the form of 3D rectangles corresponding to all feature maps of a subset of a layer, see Figure 8. These input neurons will be first distributed to all node tiles through the (fat tree) internal network, see Figure 5. Simultaneously, the node control starts to send the block of input neurons to the rest of the nodes through the mesh.

With respect to communications, there are three main layer cases to consider. First, convolutional and pooling layers are characterized by local connectivity defined by the small window (convolutional or pooling kernel) used to sample the input neurons. Due to the local connectivity, the amount of inter-node communications is very low (most communications are intra-node), mostly occurring at the border of the layer rectangle mapped to each node, see Figure 8.

For local response normalization layers, since all feature maps at a given location are always mapped to the same node, there is no inter-node communication.

Finally, communications can be high for classifier layers because each output neuron uses all input neurons, see Figure 8. At the same time, the communication pattern is simple, equivalent to a broadcast. Since each node performs roughly the same amount of computations at the same speed, and since each node must simultaneously broadcast its set of input neurons to all other nodes, we adopt a computing-and-forwarding communication scheme [24], which is equivalent to arranging the nodes communications according to a regular ring pattern. A node can start processing the newly arrived block of input neurons as soon as it has finished its own computations, and has sent the previous block of input neurons; so the decision is made locally, there is no global synchronization or barrier.

VI. METHODOLOGY

A. Measurements

Our experiments use the following three tools.

CAD tools. We implemented a Verilog version of the node, then synthesized it, and did the layout. The area, energy and critical path delays are obtained after layout using the ST 28nm Low Power (LP) technology (0.9V). We used the Synopsys Design Compiler for the synthesis, ICC Compiler for the layout, and the power consumption was estimated using Synopsys PrimeTime PX.

Time, eDRAM and inter-node measurements. We use VCS to simulate the node RTL, an eDRAM model which includes destructive reads, and periodic refresh of a banked eDRAM running at 606MHz (the eDRAM energy was collected using CACTI5.3 [1] after integrating the 1T1C cell characteristics at 28nm [25]), and inter-node communications were simulated using the cycle-level Booksim2.0 interconnection network simulator [10] (Orion2.0 [29] for the network energy model).

GPU. We use the NVIDIA K20M GPU of Section III as a baseline. The GPU can also report its power usage. We use CUDA SDK 5.5 to compile the CUDA version of neural network codes.

B. Baseline

In order to maximize the quality of our baseline, we extracted the CUDA versions from a tuned open-source version, CUDA Convnet [31]. In order to assess the quality of this baseline, we have compared it against the C++ version run on the Intel SIMD CPU, see Section III. For the C++ version, we have first compared the SIMD version against a non-SIMD version (SIMD compilation deactivated), and we have observed an average speedup of the SIMD version of 4.07x, confirming that the compiler was effectively taking advantage of the SIMD unit. As mentioned in Section III, the CUDA/GPU over the C++/CPU (SIMD) speedups reported in Figure 2 are in line with some of the best reported results so far, by Ciresan et al. [7] (10x to 60x).

VII. EXPERIMENTAL RESULTS

We first present the main characteristics of the node layout, then present the performance and energy results of the multi-chip system.

A. Main Characteristics

The cell-based layout of the chip is shown in Figure 9, and the area breakdown in Table VI. 44.53% of the chip area is used by the 16 tiles, 26.02% by the four HT IPs, 11.66% by the central block (including 4MB eDRAM, router and control logic). The wires between the central block and the tiles occupy 8.97% of the area. Overall, about a half (47.55%) of the chip is consumed by memory cells (mostly

Figure 8: Mapping of (left) a convolutional (or pooling) layer with 4 feature maps; the red section indicates the input neurons used by node 0; (right) a classifier layer.
CONV3* and CONV4*, need a 36-node system because their size is respectively 1.29 GB and 1.32 GB. The full NN contains 59.48M synapses, i.e., 118.96MB (16-bit data), requiring at least 4 nodes.

On average, the 1-node, 4-node, 16-node and 64-node architectures are respectively 21.38x, 79.81x, 216.72x, and 450.65x faster than the GPU baseline. The first reason for the higher performance is the large number of operators: in each node, there are 9216 operators (mostly multipliers and adders), compared to the 2496 MACs of the GPU. The second reason is that the on-chip eDRAM provides the necessary bandwidth and low-latency access to feed these many operators.

Nevertheless, the scalability of the different layers varies a lot. LRN layers scale the best (no inter-node communication) with a speedup of up to 1340.77x for 64 nodes (LRN2), CONV and POOL layers scale almost as well because they only have inter-node communications on border elements, e.g., CONV1 achieves a speedup of 2595.23x for 64 nodes, but the actual speedup of LRN and POOL layers is lower than CONV layers because they are less computationally intensive. On the other hand, CLASS layers scale less well because of the high amount of inter-node communications, since each output neuron uses all input neurons from different nodes, see Section V-E2, e.g., CLASS1 has a speedup of 72.96x for 64 nodes. This is further illustrated in the time breakdown of Figure 11. Note that each bar is normalized to the total execution time, but due to the overlap of computation and communication, the cumulated bars can exceed 100%. This communication issue is mostly due to our relatively simple 2D mesh topology where the larger the number of nodes, the longer the time required to send each block of inputs to all nodes. It is likely that a more sophisticated multi-dimensional torus topology [4] can largely reduce the total broadcast time as the number of nodes increases, but we leave this optimization for future work.

1Considering that the area of K20M GPU is about 550 mm², and our node is only 67.7 mm², our design also has a high area-normalized speedup with respect to GPU (21.38 \times 550 / 67.7 = 173.69x for 1-node and 450.65 \times 550 / (64 \times 67.7) = 57.20x for 64-node).
Finally, we note that the full NN scales similarly to CLASS layers (63.35x, 116.85x, 164.80x for 4-node, 16-node, 64-node respectively). While it may suggest that CLASS layers dominate the full NN execution time, a breakdown by layer type, see Table VII, shows that it is not the case, on the contrary, CONV layers largely dominate. The reason is simply that this full NN contains layers which are a bit small for a large 64-node machine. For instance, there are three CONV layers with a dimension of 13x13 (though 256 to 384 feature maps), so, even though all feature maps are mapped to the same node, we need to attribute an $X \times Y$ area of size $2 \times 2$ or $3 \times 3$ at most per node ($\frac{13 \times 13}{64} = 2.69$) which means that either we do not use all nodes, or inter-node communications are required for every kernel computation.

![Figure 11: Time breakdown (left) for 4, 16 and 64 nodes, (right) breakdown for 1, 4, 16, 64 nodes; CLASS, CONV, POOL, LRN stand for the geometric means of all layers of the corresponding type, Gmean for the global geometric mean.](image1)

![Figure 12: Speedup w.r.t. the GPU baseline (training).](image2)

![Figure 13: Energy reduction w.r.t. the GPU baseline (inference).](image3)

![Figure 14: Energy reduction w.r.t. the GPU baseline (training).](image4)

Table VII: Full NN execution time breakdown per layer type.

<table>
<thead>
<tr>
<th></th>
<th>CONV</th>
<th>LRN</th>
<th>POOL</th>
<th>CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-node</td>
<td>96.63%</td>
<td>0.60%</td>
<td>0.47%</td>
<td>2.31%</td>
</tr>
<tr>
<td>16-node</td>
<td>96.87%</td>
<td>0.28%</td>
<td>0.22%</td>
<td>2.63%</td>
</tr>
<tr>
<td>64-node</td>
<td>93.25%</td>
<td>0.10%</td>
<td>0.08%</td>
<td>7.57%</td>
</tr>
</tbody>
</table>

**Training and initialization.** We carry out similar experiments for back propagation, and the weights pre-training phase (RBM) using 32-bit fixed point operators (while inference uses 16-bit fixed-point operators), see Section V-B3. On average, the 1-node, 4-node, 16-node and 64-node architectures are respectively 12.62x, 43.23x, 126.66x and 300.04x faster than the GPU baseline. The minimum energy improvement is 47.66x for CLASS1 with 64 nodes, while the best energy improvement is 896.58x, achieved with CONV2 on a single node. For convolutional, pooling, and LRN layers, we observe that the energy benefit remains relatively stable as the number of nodes is scaled up, though it degrades for classifier layers. The latter is expected as the average communication (and thus overall execution) time increases; again, a multi-dimensional torus should help reduce this energy loss.

In the energy breakdown of Figure 11, we can observe that, for the 1-node architecture, about 83.89% of the energy is consumed by the NFU. As we scale up the number of nodes, the trend largely corroborates previous observations: the ratio of energy spent in HT progressively increases to 29.32% on average for a 64-node system, especially due to the larger number of communications in classifier layers (48.11%).

**C. Energy Consumption**

As shown in Figure 13, the 1-node, 4-node, 16-node and 64-node architectures can reduce the energy by 330.56x, 323.74x, 276.04x, and 150.31x respectively compared to the GPU baseline. The minimum energy improvement is 47.66x for CLASS1 with 64 nodes, while the best energy improvement is 896.58x, achieved with CONV2 on a single node. For convolutional, pooling, and LRN layers, we observe that the energy benefit remains relatively stable as the number of nodes is scaled up, though it degrades for classifier layers. The latter is expected as the average communication (and thus overall execution) time increases; again, a multi-dimensional torus should help reduce this energy loss.
142.59x, and 66.94x for the 1-node, 4-node, 16-node and 64-node architectures, see Figure 14. The scalability behavior is similar to that of the inference phase.

VIII. RELATED WORK

Machine-Learning. Convolutional and Deep Neural Networks have become popular algorithms in data center based services: they are used for web search [27], image analysis [41], speech recognition [9], etc. Such services are computationally intensive, and considering the energy and operating costs of data centers, custom architectures could help from both a performance and energy perspective. But web services are only the most visible applications. CNNs are being used for handwritten digits recognition [28] with applications in banking and post-offices, Dahl et al. [9] have recently won a pharmaceutical contest using DNN algorithms. More generally, such algorithms might take a central role in the so-called big data application domain. While CNNs and DNNs keep evolving, we note for instance that the first CNN design [35] still achieves very good results compared to the latest instantiations on benchmarks such as the MNIST handwritten digits [35], with a recognition accuracy of 1.7% (1998) versus 0.23% for the best algorithm by Ciresan et al. [6] (2012).

So, even though there is an inherent risk in freezing algorithms in hardware, (1) hardware can rapidly evolve with machine-learning progress, much like it currently (and rapidly) evolves with technology progress, (2) what machine-learning researchers rightfully view as significant accuracy progress from their perspective (e.g., improving accuracy by 1 or 2% can be very difficult) may not be so significant from an end-user perspective, so that hardware needs not implement and follow each and every evolution, (3) end users are already accustomed to the notion of software libraries, and they can always choose between a rigid but very fast “hardware library” and a slow but more flexible CPU/GPU implementation.

Custom accelerators. Due to the end of Dennard scaling and the notion of Dark Silicon [42], [15], architecture customization is increasingly viewed as one of the most promising paths forward. So far, the emphasis has been especially on custom accelerators, i.e., custom tiles within a heterogeneous multi-cores. Accelerators for video compression [21], image convolutions [44], libraries or specific workloads [49], [17] have been proposed.

Closer to the target algorithms of this paper, a number of studies have recently advocated the notion of neural network accelerators, either to approximate any function of a program [16], for signal-processing tasks [2] or specifically for machine-learning tasks [22], [23], [47], [5].

Large-Scale custom architectures. There are few examples of custom architectures targeting large-scale neural networks. The closest examples are the following. Schemmel et al. [46] propose a wafer-scale design capable of implementing thousands of neurons and millions of synapses. Khan et al. [30] propose the SpiNNaker system, which is a multi-chip supercomputer where each node contains 20+ ARM9 cores linked by an asynchronous network; the planned target is a million-core machine capable of modeling a billion neurons. Finally, the IBM Cognitive Chip [39] is a functional chip capable of implementing 256 neurons and 256K synapses in 4mm² at 45nm. However, the common point between these different architectures is that their goal is the emulation of biological neurons, not machine-learning tasks, even though some of them have demonstrated machine-learning capabilities on simple tasks. Moreover, the neurons they implement are inspired from biology, i.e., spiking neurons, they do not implement the CNNs and DNNs which are the focus of our architecture. Majumdar et al. [37] investigate a parallel architecture for various machine-learning algorithms, including, but not only, neural networks; unlike our architecture, they have an off-chip banked memory, and they introduce memory banks close to PEs (similar to those found in GPUs) for caching purposes. Finally, beyond neural networks and machine-learning tasks, other large-scale custom architectures have been proposed, such as the recently proposed Anton [12], for molecular dynamics simulation.

IX. CONCLUSIONS AND FUTURE WORK

In this article, we investigate a custom multi-chip architecture for state-of-the-art machine-learning algorithms (CNNs and DNNs), motivated by the increasingly central role of such algorithms in large-scale deployments of sophisticated services for consumers and industry. On both GPUs and recently proposed accelerators, such algorithms exhibit good speedups and area savings respectively, but they remain largely bandwidth-limited. We show that it is possible to design a multi-chip architecture which can outperform a single GPU by up to 450.65x and reduce energy by up to 150.31x using 64 nodes. Each node has an area of 67.73mm² at 28nm.

We plan to improve this architecture along several directions: increasing the clock frequency of the NFU, multi-dimensional torus interconnects to improve the scalability of large classifier layers, investigating more flexible control in the form of a simple VLIW core per node and the associated toolchain. A tape-out of a node chip is planned soon, followed by a multi-node prototype.

ACKNOWLEDGMENTS

This work is partially supported by the NSF of China (under Grants 61100163, 61133004, 61222204, 61221062, 61303158, 61432016, 61472396, 61473275), the 863 Program of China (under Grant 2012AA012202), the 973 Program of China (under Grant 2011CB302500), the Strategic Priority Research Program of the CAS (under Grant
XDA06010403), the International Collaboration Key Program of the CAS (under Grant 171111KYSB20130002), a Google Faculty Research Award, the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), the 10,000 talent program, and the 1,000 talent program.

REFERENCES


