A Small-Footprint Accelerator for Large-Scale Neural Networks

TIANSHI CHEN, SHIJIN ZHANG, SHAOLI LIU, ZIDONG DU, and TAO LUO,
SKLCA, ICT, CAS
YUAN GAO, JUNJIE LIU, and DONGSHENG WANG, TNLIST, Tsinghua University
CHENGYONG WU, NINGHUI SUN, and YUNJI CHEN,
SKLCA, ICT, CAS
OLIVIER TEMAM, Inria, Saclay, France

6

Machine-learning tasks are becoming pervasive in a broad range of domains, and in a broad range of systems (from embedded systems to data centers). At the same time, a small set of machine-learning algorithms (especially Convolutional and Deep Neural Networks, i.e., CNNs and DNNs) are proving to be state-of-the-art across many applications. As architectures evolve toward heterogeneous multicores composed of a mix of cores and accelerators, a machine-learning accelerator can achieve the rare combination of efficiency (due to the small number of target algorithms) and broad application scope.

Until now, most machine-learning accelerator designs have been focusing on efficiently implementing the computational part of the algorithms. However, recent state-of-the-art CNNs and DNNs are characterized by their large size. In this study, we design an accelerator for large-scale CNNs and DNNs, with a special emphasis on the impact of memory on accelerator design, performance, and energy.

We show that it is possible to design an accelerator with a high throughput, capable of performing 452 GOP/s (key NN operations such as synaptic weight multiplications and neurons outputs additions) in a small footprint of 3.02mm^2 and 485mW; compared to a 128-bit 2GHz SIMD processor, the accelerator is 117.87× faster, and it can reduce the total energy by 21.08×. The accelerator characteristics are obtained after layout at 65nm. Such a high throughput in a small footprint can open up the usage of state-of-the-art machine-learning algorithms in a broad set of systems and for a broad set of applications.

Categories and Subject Descriptors: C.1.3 [**Other Architecture Styles**]: Neural Nets

General Terms: Architecture, Processor, Hardware

A preliminary version of this work appeared in the Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14): T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning."

This work is partially supported by the NSF of China (under Grants 61100163, 61133004, 61222204, 61221062, 61303158, 61432016, 61472396, 61473275), the 973 Program of China (under Grant 2015CB358800), 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 French ANR MHANN and NEMESIS grants, the 10,000 and 1,000 talent programs.

Y. Chen (cyj@ict.ac.cn) is the corresponding author of the article. Y. Chen is also with CAS Center for Excellence in Brain Science, China.

Authors' addresses: T. Chen, S. Zhang, S. Liu, Z. Du, T. Luo, C. Wu, N. Sun, and Y. Chen (corresponding author) are with State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China; emails:{chentianshi, zhangshijin, liushaoli, duzidong, luotao, cwu, snh, cyj}@ict.ac.cn; O. Temam is with Inria Saclay, France; email: olivier.temam@inria.fr; Y. Gao, J. Liu, and D. Wang are with Tsinghua University, Beijing 100084, China; email: {g-y12, liujunjie12}@mails.tsinghua.edu.cn, wds@tsinghua.edu.cn.

DOI: http://dx.doi.org/10.1145/2701417

Additional Key Words and Phrases: Hardware accelerator, deep learning, deep neural network, convolutional neural network

ACM Reference Format:
Tianshi Chen, Shijin Zhang, Shaoli Liu, Zidong Du, Tao Luo, Yuan Gao, Junjie Liu, Dongsheng Wang, Chengyong Wu, Ninghui Sun, Yunji Chen, and Olivier Temam. 2015. A small-footprint accelerator for large-scale neural networks. ACM Trans. Comput. Syst. 33, 2, Article 6 (May 2015), 27 pages.
DOI: http://dx.doi.org/10.1145/2701417

1. INTRODUCTION

As architectures evolve toward heterogeneous multicores composed of a mixture between cores and accelerators, designing accelerators that realize the best possible trade-off between flexibility and efficiency is becoming a prominent issue.

The first question is which category of applications should one primarily design accelerators for? Together with the architecture trend toward accelerators, a second simultaneous and significant trend in high-performance and embedded applications is developing: Many of the emerging high-performance and embedded applications, from image/video/audio recognition to automatic translation, business analytics, and all forms of robotics rely on *machine-learning techniques*. This trend even starts to percolate in our community where it turns out that about half of the benchmarks of PARSEC [Bienia et al. 2008], a suite partly introduced to highlight the emergence of new types of applications, can be implemented using machine-learning algorithms [Chen et al. 2012]. This trend in application comes together with a third and equally remarkable trend in machine learning where a small number of techniques, based on neural networks (especially Convolutional Neural Networks (CNNs) [Lecun et al. 1998] and Deep Neural Networks (DNNs) [Hinton and Srivastava 2012]), have been proved in the past few years to be state-of-the-art across a broad range of applications [Larochelle et al. 2007]. As a result, there is a unique opportunity to design accelerators that can realize the best of both worlds: significant application scope together with high performance and efficiency due to the limited number of target algorithms.

Currently, these workloads are mostly executed on multicores using Single Instruction Multiple Data (SIMD) [Vanhoucke et al. 2011], on Graphics Processing Units (GPUs) [Coates et al. 2013], or on Field-Programmable Gate Arrays (FPGAs) [Chakradhar et al. 2010]. However, the aforementioned trends have already been identified by a number of researchers who have proposed accelerators implementing CNNs [Chakradhar et al. 2010] or Multilayer Perceptrons [Temam 2012]; accelerators focusing on other domains, such as image processing, also propose efficient implementations of some of the primitives used by machine-learning algorithms, such as convolutions [Qadeer et al. 2013]. Others have proposed ASIC implementations of CNNs [Farabet et al. 2011], or of other custom neural network algorithms [Kim et al. 2010]. However, all these works have first, and successfully, focused on efficiently implementing the computational primitives, but they either voluntarily ignore memory transfers for the sake of simplicity [Qadeer et al. 2013; Temam 2012], or they directly plug their computational accelerator to memory via a more or less sophisticated Direct Memory Access (DMA) [Chakradhar et al. 2010; Farabet et al. 2011; Kim et al. 2010].

While efficient implementation of computational primitives is a first and important step with promising results, inefficient memory transfers can potentially void the throughput, energy, or cost advantages of accelerators, that is, an Amdahl's law effect, and thus, they should become a first-order concern, just like in processors, rather than an element factored in accelerator design on a second step. Unlike in processors though, one can factor in the specific nature of memory transfers in target algorithms, just like is done for accelerating computations. This is especially important in the domain of machine learning where there is a clear trend toward scaling up the size of neural

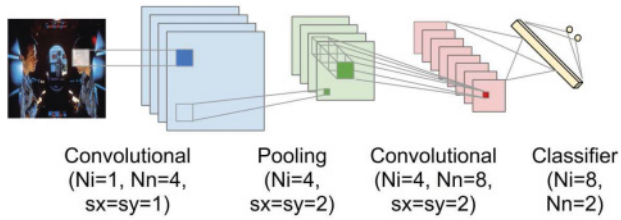

Fig. 1. Neural network hierarchy containing convolutional, pooling, and classifier layers.

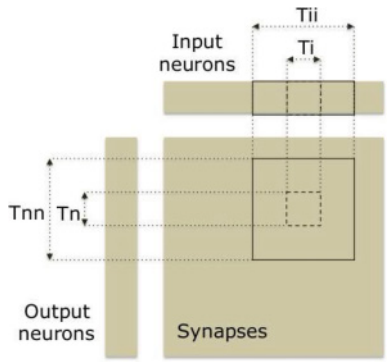

Fig. 2. Classifier layer tiling.

networks in order to achieve better accuracy and more functionality [Le et al. 2012; Hinton and Srivastava 2012].

In this study, we investigate an accelerator design that can accommodate the most popular state-of-the-art algorithms, that is, CNNs and DNNs. We focus the design of the accelerator on memory usage, and we investigate an accelerator architecture and control both to minimize memory transfers and to perform them as efficiently as possible. We present a design at 65nm that can perform 496 16-bit fixed-point operations in parallel every 1.02ns, that is, 452 GOP/s, in a 3.02-mm^2, 485-mW footprint (excluding main memory accesses). On 10 of the largest layers found in recent CNNs and DNNs, this accelerator is 117.87× faster and 21.08× more energy efficient (including main memory accesses) on average than an 128-bit SIMD core clocked at 2GHz.

In summary, our main contributions are the following:

—A synthesized (place and route) accelerator design for large-scale CNNs and DNNs, the state-of-the-art machine-learning algorithms.
—The accelerator achieves high throughput in a small area, power and energy footprint.
—The accelerator design focuses on memory behavior, and measurements are not circumscribed to computational tasks; they factor in the performance and energy impact of memory transfers.

The article is organized as follows. In Section 2, we first provide a primer on recent machine-learning techniques and introduce the main layers composing CNNs and DNNs. In Section 3, we analyze and optimize the memory behavior of these layers, in preparation for both the baseline and the accelerator design. In Section 4, we explain why an ASIC implementation of large-scale CNNs or DNNs cannot be the same as the straightforward ASIC implementation of small NNs. We introduce our accelerator

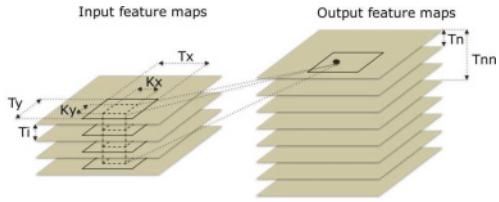

Fig. 3. Convolutional layer tiling.

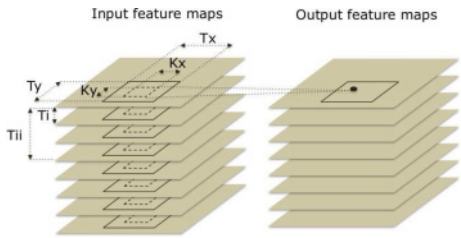

Fig. 4. Pooling layer tiling.

design in Section 5. The methodology is presented in Section 6, the experimental results in Section 7, and related work in Section 8.

2. PRIMER ON RECENT NEURAL NETWORK TECHNIQUES

Even though the role of neural networks in the machine-learning domain has been rocky, that is, initially hyped in the 1980s/1990s, then fading into oblivion with the advent of Support Vector Machines [Cortes and Vapnik 1995]. Since 2006, a subset of neural networks have emerged as achieving state-of-the-art machine-learning accuracy across a broad set of applications, partly inspired by progress in neuroscience models of computer vision, such as HMAX [Serre et al. 2007]. This subset of neural networks includes both DNNs [Larochelle et al. 2007] and CNNs [Lecun et al. 1998]. DNNs and CNNs are strongly related, yet differ in the presence and/or nature of convolutional layers, see later.

Processing vs. training. For now, we have implemented the fast processing of inputs (feed-forward) rather than training (backward path) on our accelerator. This derives from technical and market considerations. Technically, there is a frequent and important misconception that *online* learning is necessary for many applications. On the contrary, for many industrial applications *offline* learning is sufficient, where the neural network is first trained on a set of data, and then shipped to the customer, for example, trained on handwritten digits, license plate numbers, a number of faces or objects to recognize, etc.; the network can be periodically taken off line and retrained. While, today, machine-learning researchers and engineers would especially want an architecture that speeds up training, this represents a small market, and for now, we focus on the much larger market of *end users*, who need fast/efficient feed-forward networks. Interestingly, machine-learning researchers who have recently dipped into hardware accelerators [Farabet et al. 2011] have made the same choice. Still, because the nature of computations and access patterns used in training (especially back propagation) is fairly similar to that of the forward path, we plan to later augment the accelerator with the necessary features to support training.

General structure. Even though DNNs and CNNs come in various forms, they share enough properties that a generic formulation can be defined. In general, these algorithms are made of a (possibly large) number of *layers*; these layers are executed *in sequence* so they can be considered (and optimized) independently. Each layer usually

contains several sublayers called *feature maps*; we then use the terms *input feature maps* and *output feature maps* to respectively refer to feature maps of the last and current layers. Overall, there are three main kinds of layers: most of the hierarchy is composed of convolutional and pooling (also called subsampling) layers, and there is a classifier at the top of the network made of one or a few layers.

Convolutional layers. The role of convolutional layers is to apply one or several *local* filters to data from the input (previous) layer. Thus, the connectivity between the input and output feature map is local instead of full. Consider the case where the input is an image; the convolution is a 2D transform between a $K_x \times K_y$ subset (window) of the input layer and a kernel of the same dimensions (see Figure 1). The kernel values are the synaptic weights between an input layer and an output (convolutional) layer. Since an input layer usually contains several input feature maps, and since an output feature map point is usually obtained by applying a convolution to the same window of *all* input feature maps (see Figure 1), the kernel is 3D, that is, $K_x \times K_y \times N_i$, where N_i is the number of input feature maps. Note that in some cases, the connectivity is sparse, that is, not all input feature maps are used for each output feature map. The typical code of a convolutional layer is shown in Figure 7 (see *Original code*). A nonlinear function is applied to the convolution output, for instance $f(x) = tanh(x)$. Convolutional layers are also characterized by the overlap between two consecutive windows (in one or two dimensions); see steps s_x, s_y for loops x, y.

In some cases, the same kernel is applied to all $K_x \times K_y$ windows of the input layer, that is, weights are implicitly *shared* across the whole input feature map. This is characteristic of CNNs, while kernels can be specific to each point of the output feature map in DNNs [Le et al. 2012]; we then use the term *private* kernels.

Pooling layers. The role of pooling layers is to aggregate information among a set of neighbor input data. In the case of images again, it serves to retain only the salient features of an image within a given window and/or to do so at different scales (see Figure 1). An important side effect of pooling layers is to reduce the feature map dimensions. An example code of a pooling layer is shown in Figure 8 (see *Original code*). Note that each feature map is pooled separately, that is, 2D pooling, not 3D pooling. Pooling can be done in various ways. Some of the preferred techniques are the *average* and *max* operations; pooling may or may not be followed by a nonlinear function.

Classifier layers. Convolution and pooling layers are interleaved within deep hierarchies, and the top of the hierarchies is usually a *classifier*. This classifier can be linear or a multilayer (often two-layer) perceptron (see Figure 1). An example perceptron layer is shown in Figure 5 (see *Original code*). Like convolutional layers, a nonlinear function is applied to the neurons output, often a sigmoid, for example, $f(x) = \frac{1}{1+e^{-x}}$; unlike convolutional or pooling layers, classifiers usually aggregate (flatten) all feature maps, so there is no notion of feature maps in classifier layers.

3. PROCESSOR-BASED IMPLEMENTATION OF (LARGE) NEURAL NETWORKS

The distinctive aspect of accelerating large-scale neural networks is the potentially high memory traffic. In this section, we analyze in details the locality properties of the different layers mentioned in Section 2, we tune processor-based implementations of these layers in preparation for both our baseline, and the design and utilization of the accelerator. We apply the locality analysis/optimization to all layers, and we illustrate the bandwidth impact of these transformations with four of our benchmark layers (CLASS1, CONV3, CONV5, POOL3); their characteristics are later detailed in Section 6.

For the memory bandwidth measurements of this section, we use a cache simulator plugged to a virtual computational structure on which we make no assumption except that it is capable of processing T_n neurons with T_i synapses each every cycle. The cache

```
for (int n = 0; n < Nn; n++)
    sum[n] = 0;
for (int n = 0; n < Nn; n++) // output neurons
    for (int i = 0; i < Ni; i++) // input neurons
        sum[n] += synapse[n][i] * neuron[i];
for (int n = 0; n < Nn; n++)
    neuron[n] = sigmoid(sum[n]);
```

```
for (int nnn = 0; nnn < Nn; nnn += Tnn){ // tiling for output neurons
    for (int iii = 0; iii < Ni; iii += Tii){ // tiling for input neurons
        for (int nn = nnn; nn < nnn + Tnn; nn += Tn){
            for (int n = nn; n < nn + Tn; n++)
                sum[n] = 0;
            for (int ii = iii; ii < iii + Tii; ii += Ti)
                for (int n = nn; n < nn + Tn; n++)
                    for (int i = ii; i < ii + Ti; i++)
                        sum[n] += synapse[n][i] * neuron[i];
            for (int n = nn; n < nn + Tn; n++)
                neuron[n] = sigmoid(sum[n]);
} } }
```

Fig. 5. Pseudocode for a classifier (here, perceptron) layer (top: original loop nest; bottom: locality optimization).

hierarchy is inspired by Intel Core i7: L1 is 32KB, 64-byte line, eight-way; the optional L2 is 2MB, 64-byte, eight-way. Unlike the Core i7, we assume the caches have enough banks/ports to serve $T_n \times 4$bytes for input neurons, and $T_n \times T_i \times 4$bytes for synapses. For large T_n, T_i, the cost of such caches can be prohibitive, but it is only used for our limit study of locality and bandwidth; in our experiments, we use $T_n = T_i = 16$.

3.1. Classifier Layers

We consider the perceptron classifier layer (see Figures 2 and 5); the tiling loops *ii* and *nn* simply reflect that the computational structure can process T_n neurons with T_i synapses simultaneously. The total number of memory transfers is (inputs loaded + synapses loaded + outputs written): $N_i \times N_n + N_i \times N_n + N_n$. For the example layer CLASS1, the corresponding memory bandwidth is high at 120 GB/s, see *CLASS1 - Original* in Figure 6. We explain in the following how it is possible to reduce this bandwidth, sometimes drastically.

Input/Output neurons. Consider Figure 2 and the code of Figure 5 again. Input neurons are reused for each output neuron, but since the number of input neurons can range anywhere between a few tens to hundreds of thousands, they will often not fit in an L1 cache. Therefore, we tile loop *ii* (input neurons) with tile factor T_{ii}. A typical trade-off of tiling is that improving one reference (here *neuron*[*i*] for input neurons) increases the reuse distance of another reference (*sum*[*n*] for partial sums of output neurons), so we need to tile for the second reference as well, hence loop *nnn* and the tile factor T_{nn} for output neurons partial sums. As expected, tiling drastically reduces the memory bandwidth requirements of input neurons, and those of output neurons increase, albeit marginally. The layer memory behavior is now dominated by synapses.

Synapses. In a perceptron layer, all synapses are usually unique, and thus there is no reuse *within the layer*. On the other hand, the synapses are reused *across network invocations*, that is, for each new input data (also called "input row") presented to the neural network. So a sufficiently large L2 could store all network synapses and take advantage of that locality. For DNNs with private kernels, this is not possible as the total number of synapses are in the tens or hundreds of millions (the largest network to date has a billion synapses [Le et al. 2012]). However, for both CNNs and DNNs with shared kernels, the total number of synapses range in the millions, which is within the reach of an L2 cache. In Figure 6 (see *CLASS1 - Tiled+L2)*, we emulate the case where

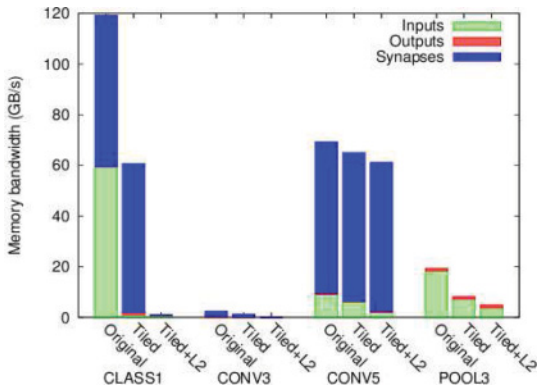

Fig. 6. Memory bandwidth requirements for each layer type (CONV3 has shared kernels, CONV5 has private kernels).

reuse across network invocations is possible by considering only the perceptron layer; as a result, the total bandwidth requirements are now drastically reduced.

3.2. Convolutional Layers

We consider two-dimensional convolutional layers (see Figures 3 and 7). The two distinctive features of convolutional layers with respect to classifier layers are the presence of input and output feature maps (loops i and n) and kernels (loops k_x, k_y).

Inputs/Outputs. There are two types of reuse opportunities for inputs and outputs: the *sliding window* used to scan the (two-dimensional (x, y)) input layer, and the reuse across the N_n output feature maps (see Figure 3). The former corresponds to $\frac{K_x \times K_y}{s_x \times s_y}$ reuses at most, and the latter to N_n reuses. We tile for the former in Figure 7 (tiles T_x, T_y), but we often do not need to tile for the latter because the data to be reused, that is, one kernel of $K_x \times K_y \times N_i$, fits in the L1 data cache since K_x, K_y are usually of the order of 10 and N_i can vary between less than 10 to a few hundreds; naturally, when this is not the case, we can tile input feature maps (ii) and introduce a second-level tiling loop iii again.

Synapses. For convolutional layers with *shared* kernels (see Section 2), the same kernel parameters (synaptic weights) are reused across all *xout*, *yout* output feature maps locations. As a result, the total bandwidth is already low, as shown for layer CONV3 in Figure 6. However, since the total shared kernels capacity is $K_x \times K_y \times N_i \times N_o$, it can exceed the L1 cache capacity, so we tile again output feature maps (tile T_{nn}) to bring it down to $K_x \times K_y \times N_i \times T_{nn}$. As a result, the overall memory bandwidth can be further reduced, as shown in Figure 6.

For convolutional layers with *private* kernels, the synapses are all unique and there is no reuse, as for classifier layers, hence the similar synapses bandwidth of CONV5 in Figure 6. As for classifier layers, reuse is still possible across network invocations if the L2 capacity is sufficient. Even though step coefficients (s_x, s_y) and sparse input to output feature maps (see Section 2) can drastically reduce the number of private kernels synaptic weights, for very large layers such as CONV5, they still range in the hundreds of megabytes and thus will largely exceed L2 capacity, implying a high memory bandwidth (see Figure 6).

It is important to note that there is an ongoing debate within the machine-learning community about shared versus private kernels [Le et al. 2012; Sermanet et al. 2012], and the machine-learning importance of having private instead of shared kernels remains unclear. Since they can result in significantly different architecture performance,

```
int yout = 0;
for (int y = 0; y < Nyin - Ky + 1; y += sy){
    int xout = 0;
    for (int x = 0; x < Nxin - Kx + 1; x += sx){
        for (int n = 0; n < Nn; n++)
            sum[n] = 0;
     // sliding window
        for (int ky = 0; ky < Ky; ky++)
            for (int kx = 0; kx < Kx; kx++)
                for (int n = 0; n < Nn; n++)
                    for (int i = 0; i < Ni; i++)
     // version with shared kernels
                        sum[n] += synapse[ky][kx][n][i]
                                * neuron[ky + y][kx + x][i];
     // version with private kernels
                        sum[n] += synapse[yout][xout][ky][kx][n][i]
                                * neuron[ky + y][kx + x][i];
        for (int n = 0; n < Nn; n++)
            neuron[yout][xout][n] = non_linear_transform(sum[n]);
        xout++;
    }
    yout++;}
```

```
for (int yy = 0; yy < Nyin - Ky + 1; yy += Ty){
    for (int xx = 0; xx < Nxin - Kx + 1; xx += Tx){
        for (int nnn = 0; nnn < Nn; nnn += Tnn){
            int yout = 0;
            for (int y = yy; y < yy + Ty; y += sy){ // tiling for y
                int xout = 0;
                for (int x = xx; x < xx + Tx; x += sx){ // tiling for x
                    for (int nn = nnn; nn < nnn + Tnn; nn += Tn){
                        for (int n = nn; n < nn + Tn; n++)
                            sum[n] = 0;
         // sliding window
                        for (int ky = 0; ky < Ky; ky++)
                            for (int kx = 0; kx < Kx; kx++)
                                for (int ii = 0; ii < Ni; ii += Ti)
                                    for (int n = nn; n < nn + Tn; n++)
                                        for (int i = ii; i < ii + Ti; i++)
         // version with shared kernels
                                            sum[n] += synapse[ky][kx][n][i]
                                                    * neuron[ky + y][kx + x][i];
         // version with private kernels
                                            sum[n] += synapse[yout][xout][ky][kx][n][i]
                                                    * neuron[ky + y][kx + x][i];
                        for (int n = nn; n < nn + Tn; n++)
                            neuron[yout][xout][n] = non_linear_transform(sum[n]);
                    } xout++;
                } yout++;
  } } } }
```

Fig. 7. Pseudocode for convolutional layer (top: original loop nest; bottom: locality optimization), both shared and private kernels versions.

this may be a case where the architecture/performance community could weigh in on the machine-learning debate.

3.3. Pooling Layers

We now consider pooling layers (see Figures 4 and 8). Unlike convolutional layers, the number of input and output feature maps is the same, and more importantly, there is no kernel, that is, no synaptic weight to store, and an output feature map element is determined only by $K_x \times K_y$ input feature map elements, that is, a 2D window (instead of a 3D window for convolutional layers). As a result, the only source of reuse comes

```
int yout = 0;
for (int y = 0; y < Nyin - Ky + 1; y += sy){
    int xout = 0;
    for (int x = 0; x < Nxin - Kx + 1; x += sx){
        for (int i = 0; i < Ni; i++)
            value[i] = 0;
        for (int ky = 0; ky < Ky; ky++)
            for (int kx = 0; kx < Kx; kx++)
                for (int i = 0; i < Ni; i++)
     // version with average pooling
                    value[i] += neuron[ky + y][kx + x][i];
     // version with max pooling
                    value[i] = max(value[i], neuron[ky + y][kx + x][i]);
        for (int i = 0; i < Ni; i++)
     // for average pooling
            neuron[xout][yout][i] = value[i] / (Kx * Ky);
     // for max pooling
            neuron[xout][yout][i] = value[i];
        xout++;
    }
    yout++;}
```

```
for (int yy = 0; yy < Nyin - Ky + 1; yy += Ty){
    for (int xx = 0; xx < Nxin - Kx + 1; xx += Tx){
        for (int iii = 0; iii < Ni; iii += Tii){
            int yout = yy / sy;
            for (int y = yy; y < yy + Ty; y += sy){
                int xout = xx / sx;
                for (int x = xx; x < xx + Tx; x += sx){
                    for (int ii = iii; ii < iii + Tii; ii += Ti){
                        for (int i = ii; i < ii + Ti; i++)
                            value[i] = 0;
                        for (int ky = 0; ky < Ky; ky++)
                            for (int kx = 0; kx < Kx; kx++)
                                for (int i = ii; i < ii + Ti; i++)
     // version with average pooling
                                    value[i] += neuron[ky + y][kx + x][i];
     // version with max pooling
                                    value[i] = max(value[i], neuron[ky + y][kx + x][i]);
                        for (int i = ii; i < ii + Ti; i++)
     // for average pooling
                            neuron[xout][yout][i] = value[i] / (Kx * Ky);
     // for max pooling
                            neuron[xout][yout][i] = value[i];
                    }xout++;
                }yout++;
  } } } }
```

Fig. 8. Pseudocode for pooling layer (top: original loop nest; bottom: locality optimization).

from the sliding window (instead of the combined effect of sliding window and output feature maps). Since there are less reuse opportunities, the memory bandwidths of input neurons are higher than for convolutional layers, and tiling (T_x, T_y) brings less dramatic improvements (see Figure 6).

4. ACCELERATOR FOR SMALL NEURAL NETWORKS

In this section, we first evaluate a "naive" and greedy approach for implementing a hardware neural network accelerator where all neurons and synapses are laid out in hardware; memory is only used for input rows and storing results. While these neural networks can potentially achieve the best energy efficiency, we show that they are not scalable. Still, we use such networks to investigate the maximum number of neurons that can be reasonably implemented in hardware.

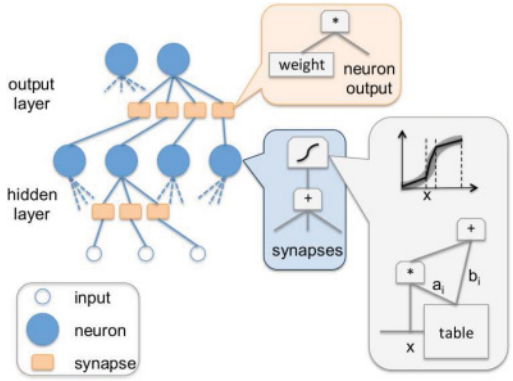

Fig. 9. Full hardware implementation of neural networks.

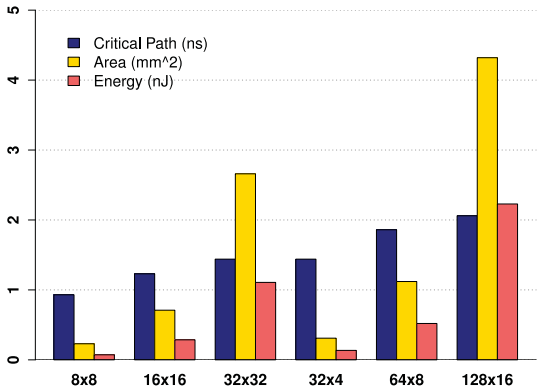

Fig. 10. Energy, critical path, and area of full-hardware layers.

4.1. Hardware Neural Networks

The most natural way to map a neural network onto silicon is simply to fully lay out the neurons and synapses, so that the hardware implementation matches the conceptual representation of neural networks (see Figure 9). The neurons are each implemented as logic circuits, and the synapses are implemented as latches or RAMs. This approach has been recently used for perceptron or spike-based hardware neural networks [Temam 2012; Merolla et al. 2011]. It is compatible with some embedded applications where the number of neurons and synapses can be small, and it can provide both high speed and low energy because the distance traveled by data is very small: from one neuron to a neuron of the next layer, and from one synaptic latch to the associated neuron. For instance, an execution time of 15ns and an energy reduction of 974× over a core has been reported for a 90-10-10 (90 inputs, 10 hidden, 10 outputs) perceptron [Temam 2012].

4.2. Maximum Number of Hardware Neurons?

However, the area, energy, and delay grow quadratically with the number of neurons. We have synthesized the ASIC versions of neural network layers of various dimensions, and we report their area, critical path, and energy in Figure 10. We have used Synopsys ICC for the place and route, and the TSMC 65nm GP library, standard VT. A hardware neuron performs the following operations: multiplication of inputs and

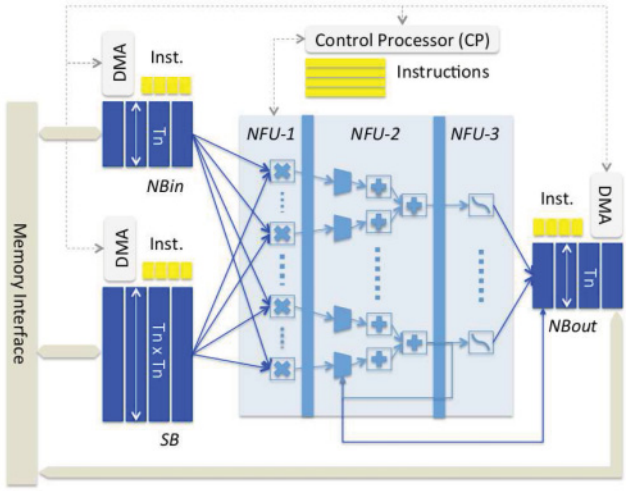

Fig. 11. Accelerator.

Table I. 32-bit Floating-Point vs. 16-bit Fixed-Point Accuracy for MNIST (Metric: Error Rate)

Type	Error Rate
32-bit floating-point	0.0311
16-bit fixed-point	0.0337

synapses, addition of all such multiplications, followed by a sigmoid (see Figure 9). A $T_n \times T_i$ layer is a layer of T_n neurons with T_i synapses each. A 16×16 layer requires less than 0.71 mm^2, but a 32×32 layer already costs 2.66mm^2. Considering the neurons are in the thousands for large-scale neural networks, a full hardware layout of just one layer would range in the hundreds or thousands of mm^2, and thus, this approach is not realistic for large-scale neural networks.

For such neural networks, only a fraction of neurons and synapses can be implemented in hardware. Paradoxically, this was already the case for old neural network designs such as the Intel ETANN [Holler et al. 1990] at the beginning of the 1990s, not because neural networks were already large at the time, but because hardware resources (number of transistors) were naturally much more scarce. The principle was to time-share the physical neurons and use the on-chip RAM to store synapses and intermediate neurons values of hidden layers. However, at that time, many neural networks were small enough that all synapses and intermediate neurons values could fit in the neural network RAM. Since this is no longer the case, one of the main challenges for large-scale neural network accelerator design has become the interplay between the computational and the memory hierarchy.

5. ACCELERATOR FOR LARGE NEURAL NETWORKS

In this section, we draw from the analysis of Sections 3 and 4 to design an accelerator for large-scale neural networks.

The main components of the accelerator are the following: an input buffer for input neurons (NBin), an output buffer for output neurons (NBout), and a third buffer for synaptic weights (SB), connected to a computational block (performing both synapse and neuron computations), which we call the Neural Functional Unit (NFU), and the

control logic (CP) (see Figure 11). Next we describe the NFU, and then we focus on and explain the rationale for the storage elements of the accelerator.

5.1. Computations: NFU

The spirit of the NFU is to reflect the decomposition of a layer into computational blocks of T_i inputs/synapses and T_n output neurons. This corresponds to loops i and n for both classifier and convolutional layers (see Figures 5 and Figure 7), and loop i for pooling layers (see Figure 8).

Arithmetic operators. The computations of each layer type can be decomposed in either two or three stages. For classifier layers: multiplication of synapses $\times$ inputs, additions of all multiplications, sigmoid. For convolutional layers, the stages are the same; the nature of the last stage (sigmoid or another nonlinear function) can vary. For pooling layers, there is no multiplication (no synapse), and the pooling operations can be average or max. Note that the adders have multiple inputs; they are in fact *adder trees* (see Figure 11); the second stage also contains shifters and max operators for pooling layers.

Staggered pipeline. We can pipeline two or three operations, but the pipeline must be staggered: the first and/or second stages (for pooling, and for classifier and convolutional layers, respectively) operate as normal pipeline stages, but the third stage is only active after all additions have been performed (for classifier and convolutional layers; for pooling layers, there is no operation in the third stage). From now on, we refer to stage n of the NFU pipeline as NFU-n.

NFU-3 function implementation. As previously proposed in the literature [Larkin et al. 2006b; Temam 2012], the sigmoid of NFU-3 (for classifier and convolutional layers) can be efficiently implemented using piecewise linear interpolation ($f(x) = a_i \times x + b_i, x \in [x_i; x_{i+1}]$) with negligible loss of accuracy (16 segments are sufficient) [Larkin et al. 2006a] (see Figure 9). In terms of operators, it corresponds to two 16×1 16-bit multiplexers (for segment boundaries selection, i.e., x_i, x_{i+1}), one 16-bit multiplier (16-bit output), and one 16-bit adder to perform the interpolation. The 16-segment coefficients (a_i, b_i) are stored in a small RAM; this allows one to implement *any* function, not just a sigmoid (e.g., hyperbolic tangent, linear functions, etc.), by changing the RAM segment coefficients a_i, b_i; the segment boundaries (x_i, x_{i+1}) are hardwired.

16-bit fixed-point arithmetic operators. We use 16-bit fixed-point arithmetic operators instead of word-size (e.g., 32-bit) floating-point operators. While it may seem surprising, there is ample evidence in the literature that even smaller operators (e.g., 8 bits or even less) have almost no impact on the accuracy of neural networks [Larkin et al. 2006a; Draghici 2002; Holi and Hwang 1993]. To illustrate and further confirm that notion, we trained and tested multilayer perceptrons on datasets from the UC Irvine Machine-Learning repository (see Figure 12) and on the standard MNIST machine-learning benchmark (handwritten digits) [Lecun et al. 1998] (see Table I), using both 16-bit fixed-point and 32-bit floating-point operators; we used 10-fold cross-validation for testing. For the fixed-point operators, we use 6 bits for the integer part and 10 bits for the fractional part (we use this fixed-point configuration throughout the article). The results are shown in Figure 12 and confirm the very small accuracy impact of that tradeoff. We also empirically evaluate the impact of small operators on the accuracy of CNN. We train a CNN (two convolutional layers, two pooling layers, and one classifier layer) with 32-bit floating-point operators, and then test it with different arithmetic operators (see Table II).When employing fixed-point operators, using 5 or 6 bits for the integer part, 10 bits for the fractional part has been sufficient to achieve the best accuracy among all 240 fixed-point settings evaluated. When employing floating-point operators, surprisingly, using 4 bits for the exponent part and 5 bits for the mantissa part achieves the best accuracy among all 256 floating-point settings

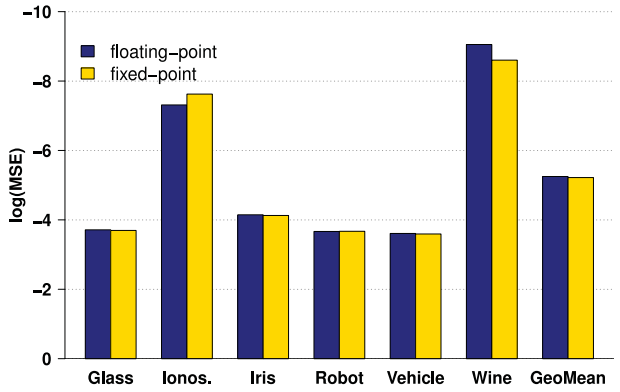

Fig. 12. 32-bit floating-point vs. 16-bit fixed-point accuracy for UCI datasets (metric: *log*(*Mean Squared Error*)).

evaluated. The CNN results show that using small operators can be a promising choice in testing (feed-forward). We conservatively use 16-bit fixed-point for now, but we will explore smaller, or variable-size, operators in the future. Note that the arithmetic operators are *truncated*, that is, their output is 16 bits; we use a standard n-bit truncated multiplier with correction constant [King and Swartzlander Jr 1997]. As shown in Table III, its area is 6.10× smaller and its power is 7.33× lower than a 32-bit floating-point multiplier at 65nm (see Section 6 for the CAD tools and methodology).

5.2. Storage: NBin, NBout, SB, and NFU-2 Registers

The different storage structures of the accelerator can be construed as modified buffers of scratchpads. While a cache is an excellent storage structure for a general-purpose processor, it is a suboptimal way to exploit reuse because of the cache access overhead (tag check, associativity, line size, speculative read, etc.) and cache conflicts [Temam and Drach 1995]. The efficient alternative, scratchpad, is used in Very Long Instruction Word (VLIW) processors but it is known to be very difficult to compile for. However, a scratchpad in a dedicated accelerator realizes the best of both worlds: efficient storage, and both efficient and easy exploitation of locality because only a few algorithms have to be manually adapted. In this case, we can almost directly translate the locality transformations introduced in Section 3 into mapping commands for the buffers, mostly modulating the tile factors. A code mapping example is provided in Section 5.3.2.

Next, we explain how the storage part of the accelerator is organized, and which limitations of cache architectures it overcomes.

5.2.1. Split Buffers. As explained before, we have split storage into three structures: an input buffer (NBin), an output buffer (NBout), and a synapse buffer (SB).

Width. The first benefit of splitting structures is to tailor the SRAMs to the appropriate read/write width. The width of both NBin and NBout is $T_n \times 2$ bytes, while the width of SB is $T_n \times T_n \times 2$ bytes. A single read width size, for example, as with a cache line size, would be a poor trade-off. If it is adjusted to synapses, that is, if the line size is $T_n \times T_n \times 2$, then there is a significant energy penalty for reading $T_n \times 2$ bytes out of a $T_n \times T_n \times 2$-wide data bank (see Figure 13, which indicates the SRAM read energy as a function of bank width for the TSMC process at 65nm). If the line size is adjusted to neurons, that is, if the line size is $T_n \times 2$, there is a significant time penalty for reading $T_n \times T_n \times 2$ bytes out. Splitting storage into dedicated structures allows one to achieve the best time and energy for each read request.

Table II. Error Rate of CNN Under Different Fixed-Point and Floating-Point Operators

Fraction \ Integer	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	90.45%	90.12%	89.73%	90.28%	89.37%	90.26%	89.67%	90.26%	89.67%	89.42%	89.95%	89.42%	89.94%	90.42%	89.83%
2	90.42%	90.42%	90.42%	90.42%	90.42%	90.42%	90.42%	90.42%	90.42%	90.42%	90.42%	90.42%	90.42%	90.42%	90.42%
3	90.33%	90.33%	90.33%	90.33%	90.33%	90.33%	90.33%	90.33%	90.33%	90.33%	90.33%	90.33%	90.33%	90.33%	90.33%
4	90.42%	90.27%	90.20%	90.20%	90.20%	90.20%	90.20%	90.20%	90.20%	90.20%	90.20%	90.20%	90.20%	90.20%	90.20%
5	88.33%	67.39%	56.03%	55.74%	55.74%	55.74%	55.74%	55.74%	55.74%	55.74%	55.74%	55.74%	55.74%	55.74%	55.74%
6	82.07%	40.94%	10.88%	6.17%	6.17%	6.17%	6.17%	6.17%	6.17%	6.17%	6.17%	6.17%	6.17%	6.17%	6.17%
7	81.81%	36.31%	3.84%	1.25%	1.25%	1.25%	1.25%	1.25%	1.25%	1.25%	1.25%	1.25%	1.25%	1.25%	1.25%
8	81.20%	33.96%	1.89%	0.93%	0.93%	0.93%	0.93%	0.93%	0.93%	0.93%	0.93%	0.93%	0.93%	0.93%	0.93%
9	81.25%	32.27%	1.52%	0.86%	0.86%	0.86%	0.86%	0.86%	0.86%	0.86%	0.86%	0.86%	0.86%	0.86%	0.86%
10	81.36%	31.4%	1.39%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%
11	81.07%	31.15%	1.33%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%
12	81.14%	30.99%	1.27%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%
13	81.12%	31.04%	1.26%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%
14	81.17%	30.99%	1.25%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%	0.85%
15	81.16%	30.97%	1.24%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%
16	81.13%	30.99%	1.25%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%

Mantissa \ Exponent	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	90.22%	86.42%	58.55%	58.83%	58.83%	58.83%	58.83%	58.83%	58.83%	58.83%	58.83%	58.83%	58.83%	58.83%	58.83%	58.83%
2	91.45%	89.29%	17.40%	18.83%	18.83%	18.83%	18.83%	18.83%	18.83%	18.83%	18.83%	18.83%	18.83%	18.83%	18.83%	18.83%
3	89.82%	89.56%	1.68%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%
4	89.73%	89.59%	1.05%	0.92%	0.94%	0.94%	0.94%	0.94%	0.94%	0.94%	0.94%	0.94%	0.94%	0.94%	0.94%	0.94%
5	89.48%	90.08%	1.17%	0.81%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%
6	89.37%	89.89%	1.19%	0.82%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%
7	89.98%	89.52%	1.25%	0.83%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%
8	89.67%	89.41%	1.42%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%
9	90.28%	89.29%	1.50%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%	0.83%
10	89.76%	89.50%	1.54%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%
11	89.42%	89.36%	1.58%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%
12	89.95%	89.15%	1.59%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%
13	89.92%	89.17%	1.59%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%
14	89.94%	89.71%	1.60%	0.84%	0.85%	0.85%	0.85%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%
15	90.12%	90.07%	1.59%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%
16	90.26%	89.90%	1.60%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%	0.84%

Table III. Characteristics of Multipliers

Type	Area (μm^2)	Power (μW)
16-bit truncated fixed-point multiplier	1,309.32	576.90
32-bit floating-point multiplier	7,997.76	4,229.60

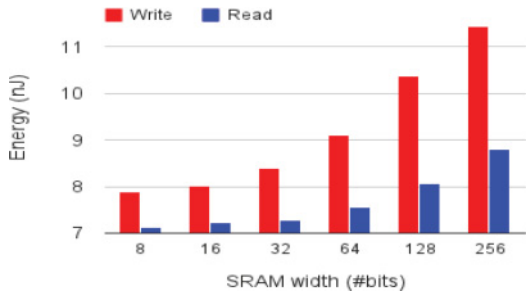

Fig. 13. Read energy vs. SRAM width.

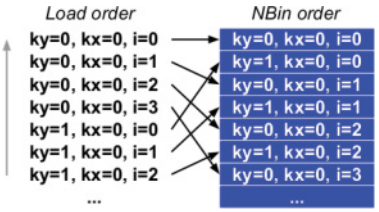

Fig. 14. Local transpose ($K_y = 2, K_x = 1, N_i = 4$).

Conflicts. The second benefit of splitting storage structures is to avoid conflicts, as would occur in a cache. It is especially important as we want to keep the size of the storage structures small for cost and energy (leakage) reasons. The alternative solution is to use a highly associative cache. Consider the constraints: the cache line (or the number of ports) needs to be large ($T_n \times T_n \times 2$) in order to serve the synapses at a high rate; since we would want to keep the cache size small, the only alternative to tolerate such a long cache line is high associativity. However, in an n-way cache, a fast read is implemented by speculatively reading all n ways/banks in parallel; as a result, the energy cost of an associative cache increases quickly. Even a 64-byte read from an eight-way associative 32KB cache costs 3.15× more energy than a 32-byte read from a direct mapped cache, at 65nm; measurements were done using CACTI [Thoziyoor et al. 2008]. And even with a 64-byte line only, the first-level 32KB data cache of Core i7 is already eight-way associative, so we need an even larger associativity with a very large line (for $T_n = 16$, the line size would be 512-byte long). In other words, a highly associative cache would be a costly energy solution in our case. Splitting storage and precise knowledge of locality behavior allows one to entirely remove data conflicts.

5.2.2. Exploiting the Locality of Inputs and Synapses (DMAs). For spatial locality exploitation, we implement three DMAs, one for each buffer (two load DMAs for inputs, one store DMA for outputs). DMA requests are issued to NBin in the form of instructions, later described in Section 5.3.2. These requests are buffered in a separate FIFO associated with each buffer (see Figure 11), and they are issued as soon as the DMA has sent all memory requests for the previous instruction. These FIFOs of DMA requests enable one to *decouple* the requests issued to all buffers and the NFU from the current buffer and NFU operations. As a result, DMA requests can be preloaded far in advance for

tolerating long latencies, as long as there is enough buffer capacity; this *preloading* is akin to prefetching, albeit without speculation. Due to the combined role of NBin (and SB) as both scratchpads for reuse and preload buffers, we use a dual-port SRAM; the TSMC 65nm library rates the read energy overhead of dual port SRAMs for a 64-entry NB at 24%.

Rotating NBin buffer for temporal reuse of input neurons. The inputs of all layers are split into chunks that fit in NBin, and they are reused by implementing NBin as a *circular buffer*. In practice, the rotation is naturally implemented by changing a register index; much like in a software implementation, there is no physical (and costly) movement of buffer entries.

Local transpose in NBin for pooling layers. There is a tension between convolutional and pooling layers for the data structure organization of (input) neurons. As mentioned before, K_x and K_y are usually small (often less than 10), and N_i is about an order of magnitude larger. So memory fetches are more efficient (long stride-1 accesses) with the input feature maps as the innermost index of the three-dimensional neurons data structure. However, this is inconvenient for pooling layers because one output is computed *per input feature map*, that is, using only $K_x \times K_y$ data (while in convolutional layers, all $K_x \times K_y \times N_i$ data are required to compute one output data). As a result, for pooling layers, the logical data structure organization is to have k_x, k_y as the innermost dimensions so that all inputs required to compute one output are consecutively stored in the NBin buffer. We resolve this tension by introducing a mapping function in NBin, which has the effect of *locally transposing* loops k_y, k_x and loop i so that data is loaded along loop i, but it is stored in NBin and thus sent to NFU along loops k_y, k_x first; this is accomplished by interleaving the data in NBin as it is loaded (see Figure 14).

For synapses and SB, as mentioned in Section 3, there is either no reuse (classifier layers, convolutional layers with private kernels, and pooling layers), or reuse of shared kernels in convolutional layers. For outputs and NBout, we need to reuse the partial sums (i.e., see reference *sum*[*n*] in Figure 5). This reuse requires additional hardware modifications, which will be explained in the next section.

5.2.3. Exploiting the Locality of Outputs. In both classifier and convolutional layers, the partial output sum of T_n output neurons is computed for a chunk of input neurons contained in NBin. Then, the input neurons are used for another chunk of T_n output neurons, etc. This creates two issues.

Dedicated registers. First, while the chunk of input neurons is loaded from NBin and used to compute the partial sums, it would be inefficient to let the partial sum exit the NFU pipeline and then reload it into the pipeline for each entry of the NBin buffer, since data transfers are a major source of energy expense [Hameed et al. 2010]. So we introduce *dedicated registers* in NFU-2, which store the partial sums.

Circular buffer. Second, a more complicated issue is what to do with the T_n partial sums when the input neurons in NBin are reused for a new set of T_n output neurons. Instead of sending these T_n partial sums back to memory (and to later reload them when the next chunk of input neurons is loaded into NBin), we temporarily rotate them out to NBout. A priori, this is a conflicting role for NBout, which is also used to store the final output neurons to be written back to memory (write buffer). In practice though, as long as all input neurons have not been integrated in the partial sums, NBout is idle. So we can use it as a temporary storage buffer by rotating the T_n partial sums out to NBout (see Figure 11). Naturally, the loop iterating over output neurons must be tiled so that no more output neurons are computing their partial sums simultaneously than the capacity of NBout, but that is implemented through a second-level tiling similar to loop *nnn* in Figures 5 and 7. As a result, NBout is not only connected to NFU-3 and memory, but also to NFU-2: one entry of NBout can be loaded into the dedicated registers of NFU-2, and these registers can be stored in NBout.

Table IV. Control Instruction Format

CP	SB				NBin							NBout				NFU						
END	READ OP	REUSE	ADDRESS	SIZE	READ OP	REUSE	STRIDE	STRIDE BEGIN	STRIDE END	ADDRESS	SIZE	READ OP	WRITE OP	ADDRESS	SIZE	NFU-1 OP	NFU-2 OP	NFU-2 IN	NFU-2 OUT	NFU-3 OP	OUTPUT BEGIN	OUTPUT END

Table V. Subset of Classifier/Perceptron Code ($N_i = 8{,}192$, $N_o = 256$, $T_n = 16$, 64-Entry Buffers)

CP	SB				NBin							NBout				NFU						
NOP	LOAD	0	0	32768	LOAD	1	0	0	0	4194304	2048	NOP	WRITE	0	0	MULT	ADD	RESET	NBOUT	SIGMOID	1	0
NOP	LOAD	0	32768	32768	READ	1	0	0	0	0	0	NOP	WRITE	0	0	MULT	ADD	RESET	NBOUT	SIGMOID	0	0
.....																						
NOP	LOAD	0	7864320	32768	LOAD	1	0	0	0	4225024	2048	READ	STORE	8388608	512	MULT	ADD	NBOUT	NFU3	SIGMOID	1	0
.....																						

5.3. Control and Code

5.3.1. CP. In this section, we describe the control of the accelerator. One approach to control would be to hardwire the three target layers. While this remains an option for the future, for now, we have decided to use *control instructions* in order to explore different implementations (e.g., partitioning and scheduling) of layers, and to provide machine-learning researchers with the flexibility to try out different layer implementations.

A layer execution is broken down into a set of instructions. Roughly, one instruction corresponds to the loops ii, i, n for classifier and convolutional layers (see Figures 5 and 7), and to the loops ii, i in pooling layers (using the interleaving mechanism described in Section 5.2.3) (see Figure 8). The instructions are stored in an SRAM associated with the *Control Processor* (CP) (see Figure 11). The CP drives the execution of the DMAs of the three buffers and the NFU. The term "processor" only relates to the aforementioned "instructions," later described in Section 5.3.2, but it has very few of the traditional features of a processor (mostly a PC and an adder for loop index and address computations); from a hardware perspective, it is more like a configurable Finite State Machine (FSM).

5.3.2. Layer Code. Every instruction has five slots, corresponding to the CP itself, the three buffers, and the NFU (see Table IV).

Because of the CP instructions, there is a need for code generation, but a compiler would be overkill in our case as only three main types of codes must be generated. So we have implemented three dedicated code generators for the three layers. In Table V, we give an example of the code generated for a classifier/perceptron layer. Since $T_n = 16$ ($16 \times$ 16-bit data per buffer row) and NBin has 64 rows, its capacity is 2KB, so it cannot contain all the input neurons ($N_i = 8192$, so 16KB). As a result, the code is broken down to operate on chunks of 2KB; note that the first instruction of NBin is a LOAD (data fetched from memory), and that it is marked as *reused* (flag immediately after load); the next instruction is a *read*, because these input neurons are rotated

in the buffer for the next chunk of T_n neurons, and the read is also marked as *reused* because there are eight such rotations ($\frac{16KB}{2KB}$); at the same time, notice that the output of NFU-2 for the first (and next) instruction is NBout, that is, the partial output neurons sums are rotated to NBout, as explained in Section 5.2.3, which is why the NBout instruction is WRITE; notice also that the input of NFU-2 is RESET (first chunk of input neurons, registers reset). Finally, when the last chunk of input neurons are sent (last instruction in table), the (store) DMA of NBout is set for writing 512bytes (256 outputs), and the NBout instruction is STORE; the NBout write operation for the next instructions will be NOP (DMA set at first chunk and automatically storing data back to memory until DMA elapses).

Note that the architecture can implement either per-image or batch processing [Vanhoucke et al. 2011]; only the generated layer control code would change.

6. EXPERIMENTAL METHODOLOGY

Measurements. We use three different tools for performance/energy measurements.

Accelerator simulator. We implemented a custom cycle-accurate, bit-accurate C++ simulator of the accelerator fabric, which was initially used for architecture exploration, and which later served as the specification for the Verilog implementation. This simulator is also used to measure time in number of cycles. It is plugged to a main memory model allowing a bandwidth of up to 250GB/s.

CAD tools. For area, energy, and critical path delay (cycle time) measurements, we implemented a Verilog version of the accelerator, which we first synthesized using the Synopsys Design Compiler using the TSMC 65nm GP standard VT library, and which we then placed and routed using the Synopsys ICC compiler. We then simulated the design using Synopsys VCS, and we estimated the power using PrimeTime PX.

SIMD. For the SIMD baseline, we use the GEM5+McPAT [Li et al. 2009] combination. We use a four-issue superscalar $\times$86 core with a 128-bit (8 $\times$ 16-bit) SIMD unit (SSE/SSE2), clocked at 2GHz. The core has a 192-entry ROB, and a 64-entry load/store queue. The L1 data (and instruction) cache is 32KB and the L2 cache is 2MB; both caches are eight-way associative and use a 64-byte line; these cache characteristics correspond to those of the Intel Core i7. The L1 miss latency to the L2 is 10 cycles, and the L2 miss latency to memory is 250 cycles; the memory bus width is 256bits. We have aligned the energy cost of main memory accesses of our accelerator and the simulator by using those provided by McPAT (e.g., 17.6nJ for a 256-bit read memory access).

We implemented a SIMD version of the different layer codes, which we manually tuned for locality as explained in Section 3 (for each layer, we perform a stochastic exploration to find good tile factors); we compiled these programs using the default `-O` optimization level but the inner loops were written in assembly to make the best possible use of the SIMD unit. In order to assess the performance of the SIMD core, we also implemented a standard C++ version of the different benchmark layers presented in the following, and on average (geometric mean), we observed that the SIMD core provides a 3.92$\times$ improvement in execution time and 3.74$\times$ in energy over the $\times$86 core.

Benchmarks. For benchmarks, we have selected the largest convolutional, pooling and/or classifier layers of several recent and large neural network structures. The characteristics of these 10 layers plus a description of the associated neural network and task are shown in Table VI.

7. EXPERIMENTAL RESULTS

7.1. Accelerator Characteristics after Layout

The current version uses $T_n = 16$ (16 hardware neurons with 16 synapses each), so that the design contains 256 16-bit truncated multipliers in NFU-1 (for classifier

Table VI. Benchmark Layers (CONV=convolutional, POOL=pooling, CLASS=classifier; CONVx* Indicates Private Kernels)

Layer	N_x	N_y	K_x	K_y	N_i	N_o	Description
CONV1	500	375	9	9	32	48	Street scene parsing (CNN) [Farabet et al. 2011], (e.g., identifying "building," "vehicle.', etc.)
POOL1	492	367	2	2	12	-	
CLASS1	-	-	-	-	960	20	
CONV2*	200	200	18	18	8	8	Detection of faces in YouTube videos (DNN) [Le et al. 2012], largest NN to date (Google)
CONV3	32	32	4	4	108	200	Traffic sign identification for car navigation (CNN) [Sermanet and LeCun 2011]
POOL3	32	32	4	4	100	-	
CLASS3	-	-	-	-	200	100	
CONV4	32	32	7	7	16	512	Google Street View house numbers (CNN) [Sermanet et al. 2012]
CONV5*	256	256	11	11	256	384	Multiobject recognition in natural images (DNN) [Hinton and Srivastava 2012], winner 2012 ImageNet competition
POOL5	256	256	2	2	256	-	

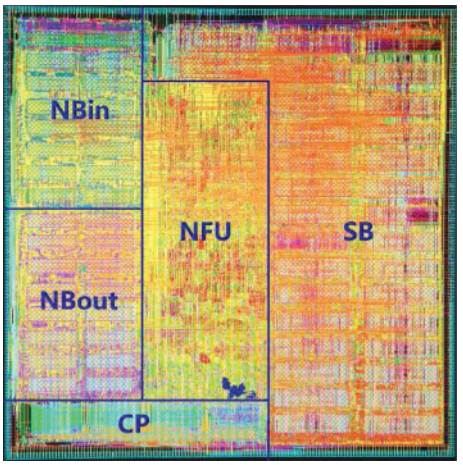

Fig. 15. Layout (65nm).

and convolutional layers), 16 adder trees of 15 adders each in NFU-2 (for the same layers, plus pooling layer if average is used), as well as a 16-input shifter and max in NFU-2 (for pooling layers), and 16 16-bit truncated multipliers plus 16 adders in NFU-3 (for classifier and convolutional layers, and optionally for pooling layers). For classifier and convolutional layers, NFU-1 and NFU-2 are active every cycle, achieving $256 + 16 \times 15 = 496$ fixed-point operations every cycle; at 0.98GHz, this amounts to 452GOP/s (Giga fixed-point OPerations per second). At the end of a layer, NFU-3 would be active as well, while NFU-1 and NFU-2 process the remaining data, reaching a peak activity of $496 + 2 \times 16 = 528$ operations per cycle (482GOP/s) for a short period.

We have done the synthesis and layout of the accelerator with $T_n = 16$ and 64-entry buffers at 65nm using Synopsys tools (see Figure 15). The main characteristics and power/area breakdown by component type and functional block are shown in Table VII. We brought the critical path delay down to 1.02ns by introducing three pipeline stages in NFU-1 (multipliers), two stages in NFU-2 (adder trees), and three stages in NFU-3 (piecewise linear function approximation) for a total of eight pipeline stages. Currently, the critical path is in the issue logic, which is in charge of reading data out of NBin/NBout; next versions will focus on how to reduce or pipeline this criti-

Table VII. Characteristics of Accelerator and Breakdown by Component Type (First 5 Lines), and Functional Block (Last 7 Lines)

Component or Block	Area in μm^2	(%)	Power in mW	(%)	Critical path in ns
ACCELERATOR	3,023,077		485		1.02
Combinational	608,842	(20.14%)	89	(18.41%)	
Memory	1,158,000	(38.31%)	177	(36.59%)	
Registers	375,882	(12.43%)	86	(17.84%)	
Clock network	68,721	(2.27%)	132	(27.16%)	
Filler cell	811,632	(26.85%)			
SB	1,153,814	(38.17%)	105	(22.65%)	
NBin	427,992	(14.16%)	91	(19.76%)	
NBout	433,906	(14.35%)	92	(19.97%)	
NFU	846,563	(28.00%)	132	(27.22%)	
CP	141,809	(5.69%)	31	(6.39%)	
AXIMUX	9,767	(0.32%)	8	(2.65%)	
Other	9,226	(0.31%)	26	(5.36%)	

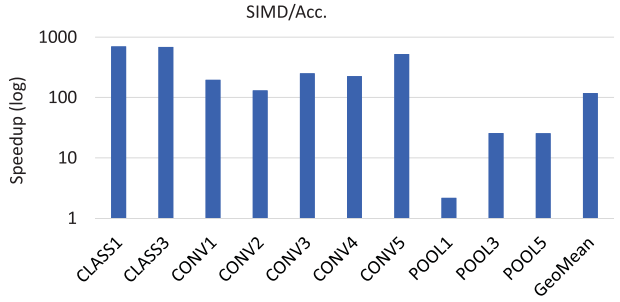

Fig. 16. Speedup of accelerator over SIMD.

cal path. The total RAM capacity (NBin + NBout + SB + CP instructions) is 44KB (8KB for the CP RAM). The area and power are dominated by the buffers (NBin/NBout/SB) at 56% and 60%, respectively, with the NFU being a close second at 28% and 27%. The percentage of the total cell power is 59.47%, but the routing network (included in the different components of the table breakdown) accounts for a significant share of the total power at 38.77%. At 65nm, due to the high toggle rate of the accelerator, the leakage power is almost negligible at 1.73%.

Finally, we have also evaluated a design with $T_n = 8$, and thus 64 multipliers in NFU-1. The total area for that design is 0.85mm^2, that is, 3.59× smaller than for $T_n = 16$ due to the reduced buffer width and the fewer number of arithmetic operators. We plan to investigate larger designs with $T_n = 32$ or 64 in the near future.

7.2. Time and Throughput

In Figure 16, we report the speedup of the accelerator over SIMD (see *SIMD/Acc*). Recall that we use a 128-bit SIMD processor, so capable of performing up to eight 16-bit operations every cycle (we naturally use 16-bit fixed-point operations in the SIMD as well). As mentioned in Section 7.1, the accelerator performs 496 16-bit operations every cycle for both classifier and convolutional layers, that is, 62× more ($\frac{496}{8}$) than the SIMD core. We empirically observe that on these two types of layers, the accelerator is on average 117.87× faster than the SIMD core, so about 2× above the ratio of computational operators (62×). We measured that, for classifier and convolutional

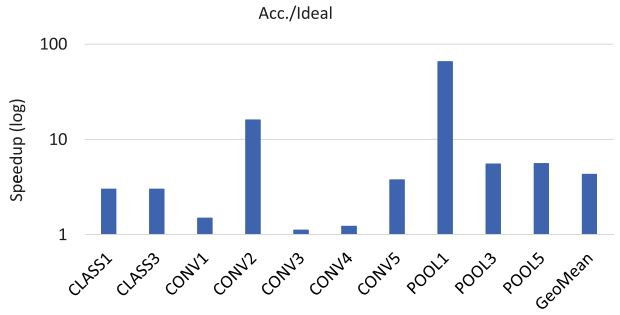

Fig. 17. Speedup of ideal accelerator over accelerator.

layers, the SIMD core performs 2.01 16-bit operations per cycle on average, instead of the upper bound of eight operations per cycle. We traced this back to two major reasons.

First, the accelerator has better latency tolerance, due to an appropriate combination of preloading and reuse in NBin and SB buffers; note that we did not implement a prefetcher in the SIMD core, which would partly bridge that gap. This explains the high performance gap for layers CLASS1, CLASS3, and CONV5, which have the largest feature map sizes, thus the most spatial locality, and which then benefit most from preloading, giving them a performance boost, that is, 629.92× on average, about 3× more than other convolutional layers; we expect that a prefetcher in the SIMD core would cancel that performance boost. The spatial locality in NBin is exploited along the input feature map dimension by the DMA, and with a small N_i, the DMA has to issue many short memory requests, which is less efficient. The rest of the convolutional layers (CONV1 to CONV4) have an average speedup of 195.15×; CONV2 has less performance (130.64×) due to private kernels and less spatial locality. Pooling layers have less performance overall because only the adder tree in NFU-2 is used (240 operators out of 496 operators): 25.73× for POOL3 and 25.52× for POOL5.

In order to further analyze the relatively poor behavior of POOL1 (only 2.17× over SIMD), we have tested a configuration of the accelerator where all operands (inputs and synapses) are ready for the NFU (i.e., ideal behavior of NBin, SB and NBout); we call this version "Ideal" (see Figure 17). We see that the accelerator is significantly slower on POOL1 and CONV2 than the ideal configuration (66.00× and 16.14×, respectively). This is due to the small size of their input/output feature maps (e.g., $N_i = 12$ for POOL1), combined with the fewer operators used for POOL1. So far, the accelerator has been geared toward large layers, but we can address this weakness by implementing a 2D or 3D DMA (DMA requests over i, k_x, k_y loops); we leave this optimization for future work.

The second reason for the speedup over SIMD beyond 62× lays in control and scheduling overhead. In the accelerator, we have tried to minimize lost cycles. For instance, when output neurons partial sums are rotated to NBout (before being sent back to NFU-2), the oldest buffer row (T_n partial sums) is eagerly rotated out to the NBout/NFU-2 input latch, and a multiplexer in NFU-2 ensures that either this latch or the NFU-2 registers are used as input for the NFU-2 stage computations; this allows a rotation without any pipeline stall. Several such design optimizations help achieve a slowdown of only 4.36× over the ideal accelerator (see Figure 17), and in fact, 2.64× only if we exclude CONV2 and POOL1.

We also compare the accelerator with a modern GPU (NVIDIA K20M, 28nm technology, 5GB GDDR5, 3.52 TFlops peak, CUDA6.0) in Figure 18. Averaged over all 10 evaluated layers, the accelerator is 0.51× slower than the GPU. We empirically observe that on two classifier layers, the accelerator is on average 1.91× faster than the

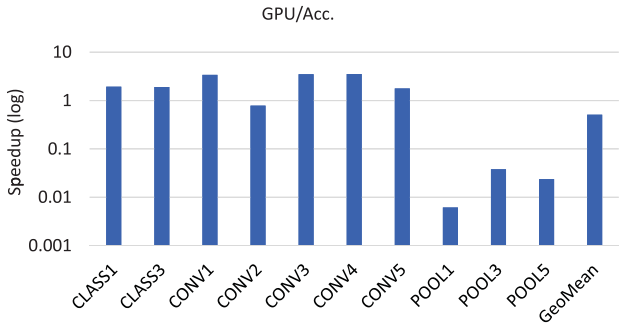

Fig. 18. Speedup of accelerator over GPU.

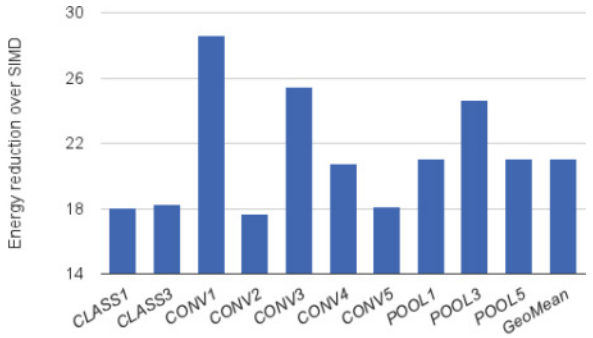

Fig. 19. Energy reduction of accelerator over SIMD.

GPU. On five convolutional layers, the accelerator is on average 2.26× faster than the GPU. The better performance of the accelerator is mainly due to reuse of NBin and SB buffers, as well as dedicated hardware support to nonlinear activation functions. On three pooling layers, we observe that the accelerator is on average 56.61× slower than the GPU. The reasons are twofold. First, there is no data reuse in three pooling layers, thus the accelerator cannot exploit any spatial locality with dedicated on-chip buffers. Second, pooling layers mainly include addition operations, but the accelerator provides much fewer adders than a GPU. Fortunately, pooling layers usually account for $< 1\%$ execution time of a CNN [Chen et al. 2014].

7.3. Energy

In Figure 19, we provide the energy ratio between the SIMD core and the accelerator. While high at 21.08×, the average energy ratio is actually more than an order of magnitude smaller than previously reported energy ratios between processors and accelerators; for instance, Hameed et al. [2010] report an energy ratio of about 500×, and 974× has been reported for a small Multilayer Perceptron [Temam 2012]. The smaller ratio is largely due to the energy spent in memory accesses, which was voluntarily not factored in the two aforementioned studies. Like in these two accelerators and others, the energy cost of computations has been considerably reduced by a combination of more efficient computational operators (especially a massive number of small 16-bit fixed-point truncated multipliers in our case), and small custom storage located close to the operators (64-entry NBin, NBout, SB, and the NFU-2 registers). As a result, there is now an Amdahl's law effect for energy, where any further improvement can only be achieved by bringing down the energy cost of main memory accesses. We tried

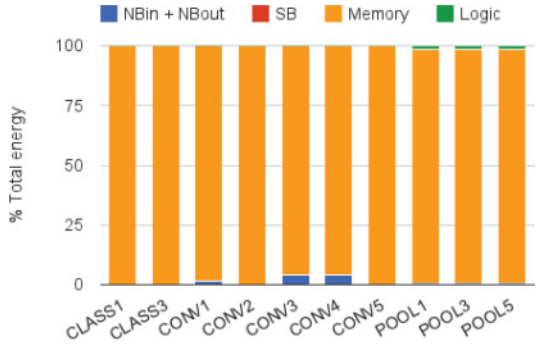

Fig. 20. Breakdown of accelerator energy.

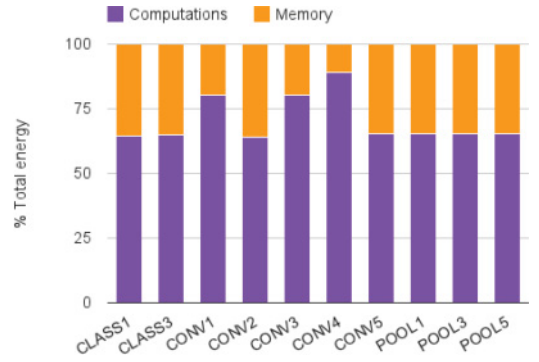

Fig. 21. Breakdown of SIMD energy.

to artificially set the energy cost of the main memory accesses in both the SIMD and accelerator to 0, and we observed that the average energy reduction of the accelerator increases by more than one order of magnitude, in line with previous results.

This is further illustrated by the breakdown of the energy consumed by the accelerator in Figure 20 where the energy of main memory accesses obviously dominates. A distant second is the energy of NBin/NBout for the convolutional layers with shared kernels (CONV1, CONV3, CONV4). In this case, a set of shared kernels are kept in SB so the memory traffic due to synapses becomes very low, as explained in Section 3 (shared kernels + tiling), but the input neurons must still be reloaded for each new set of shared kernels, hence the still noticeable energy expense. The energy of the computational logic in pooling layers (POOL1, POOL3, POOL5) is similarly a distant second expense, this time because there is no synapse to load. The slightly higher energy reduction of pooling layers (22.17× on average) (see Figure 19) is due to the fact the SB buffer is not used (no synapse), and the accesses to NBin alone are relatively cheap due to its small width (see Figure 13).

The SIMD energy breakdown is in sharp contrast, as shown in Figure 21, with about two-thirds of the energy spent in computations, and only one-third in memory accesses. While finding a computationally more efficient approach to SIMD made sense, future work for the accelerator should focus on reducing the energy spent in memory accesses.

8. RELATED WORK

Due to stringent energy constraints, such as Dark Silicon [Muller 2010; Esmaeilzadeh et al. 2011], there is a growing consensus that future high-performance

microarchitectures will take the form of heterogeneous multicores, that is, combinations of cores and accelerators. Accelerators can range from processors tuned for certain tasks, to ASIC-like circuits such as H264 [Hameed et al. 2010], or more flexible accelerators capable of targeting a broad range of (but not all) tasks [Fan et al. 2009; Yehia et al. 2009] such as QsCores [Venkatesh et al. 2011], or accelerators for image processing [Qadeer et al. 2013].

The accelerator proposed in this article follows this spirit of targeting a specific, but broad, domain, that is, machine-learning tasks here. Due to recent progress in machine learning, certain types of neural networks, especially DNNs [Larochelle et al. 2007] and CNNs [Lecun et al. 1998] have become state-of-the-art machine-learning techniques [Le et al. 2012] across a broad range of applications such as web search [Huang et al. 2013], image analysis [Mnih and Hinton 2012] or speech recognition [Dahl et al. 2013].

While many implementations of hardware neurons and neural networks have been investigated in the past two decades [Holler et al. 1990], the main purpose of hardware neural networks has been fast modeling of biological neural networks [Schemmel et al. 2008; Khan et al. 2008] for implementing neurons with thousands of connections. While several of these neuromorphic architectures have been applied to computational tasks [Merolla et al. 2011; Vogelstein et al. 2007], the specific bioinspired information representation (spiking neural networks) they rely on may not be competitive with state-of-the-art neural networks, though this remains an open debate at the threshold between neuroscience and machine learning.

However, recently, due to simultaneous trends in applications, machine-learning and technology constraints, hardware neural networks have been increasingly considered as potential accelerators, either for very dedicated functionalities within a processor, such as branch prediction [Amant et al. 2008], or for their fault-tolerance properties [Hashmi et al. 2011; Temam 2012]. The latter property has also been leveraged to trade application accuracy for energy efficiency through hardware neural processing units [Esmaeilzadeh et al. 2012; Du et al. 2014].

The focus of our accelerator is on large-scale machine-learning tasks, with layers of thousands of neurons and millions of synapses, and for that reason, there is a special emphasis on interactions with memory. Our study not only confirms previous observations that dedicated storage is key for achieving good performance and power [Hameed et al. 2010], but it also highlights that, beyond exploiting locality at the level of registers located close to computational operators [Qadeer et al. 2013; Temam 2012], considering memory as a prime-order concern can profoundly affect accelerator design.

Many of the aforementioned studies stem from the architecture community. A symmetric effort has started in the machine-learning community where a few researchers have been investigating hardware designs for speeding up neural network processing, especially for real-time applications. Neuflow [Farabet et al. 2011] is an accelerator for fast and low-power implementation of the feed-forward paths of CNNs for vision systems. It organizes computations and register-level storage according to the sliding window property of convolutional and pooling layers; but in that respect, it also ignores much of the first-order locality coming from input and output feature maps. Its interplay with memory remains limited to a Direct Memory Access (DMA), there is no significant on-chip storage, though the DMA is capable of performing complex access patterns. A more complex architecture, albeit with similar performance as Neuflow, has been proposed by Kim et al. [2010] and consists of 128 SIMD processors of 16 Processing Elements (PEs) each; the architecture is significantly larger and implements a specific neural vision model (neither CNNs nor DNNs), but it can achieve 60 frame/s (real-time) multiobject recognition for up to 10 different objects. Maashri et al. [2012] have also investigated the implementation of another neural network model, the bioinspired HMAX for vision processing, using a set of custom accelerators arranged around

a switch fabric; in the article, the authors allude to locality optimizations across different orientations, which are roughly the HMAX equivalent of feature maps. Closer to our community again, but solely focusing on CNNs, Chakradhar et al. [2010] have also investigated the implementation of CNNs on reconfigurable circuits; though there is little emphasis on locality exploitation, they pay special attention to properly mapping a CNN in order to improve bandwidth utilization.

9. CONCLUSIONS

In this article, we focus on accelerators for machine learning, because the broad set of applications and the few key state-of-the-art algorithms offer the rare opportunity to combine high efficiency and broad application scope. Since state-of-the-art CNNs and DNNs mean very large networks, we specifically focus on the implementation of large-scale layers. By carefully exploiting the locality properties of such layers, and by introducing storage structures custom designed to take advantage of these properties, we show that it is possible to design a machine-learning accelerator capable of high performance in a very small area footprint. Our measurements are not circumscribed to the accelerator fabric, they factor in the performance and energy overhead of main memory transfers; still, we show that it is possible to achieve a speedup of 117.87× and an energy reduction of 21.08× over a 128-bit 2GHz SIMD core with a normal cache hierarchy. We have obtained a layout of the design at 65nm.

Besides a planned tape-out, future work includes improving the accelerator behavior for short layers, slightly altering the NFU to include some of the latest algorithmic improvements such as Local Response Normalization, further reducing the impact of main memory transfers, investigating scalability (especially increasing T_n), and implementing training in hardware.

REFERENCES

Renee St. Amant, Daniel A. Jimenez, and Doug Burger. 2008. Low-power, high-performance analog neural branch prediction. In *International Symposium on Microarchitecture*. Como.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques*. ACM Press, New York, New York.

Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *International Symposium on Computer Architecture*. ACM Press, New York, NY, 247. DOI:http://dx.doi.org/10.1145/1815961.1815993

Tianshi Chen, Yunji Chen, Marc Duranton, Qi Guo, Atif Hashmi, Mikko Lipasti, Andrew Nere, Shi Qiu, Michele Sebag, and Olivier Temam. 2012. BenchNN: On the broad potential application scope of hardware neural network accelerators. In *International Symposium on Workload Characterization*.

Yunji Chen, Tao Luo, Shijin Zhang, Shaoli Liu, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A machine-learning supercomputer. In *International Symposium on Microarchitecture*.

Adam Coates, Brody Huval, Tao Wang, David J. Wu, and Andrew Y. Ng. 2013. Deep learning with cots hpc systems. In *International Conference on Machine Learning*. http://jmlr.org/proceedings/papers/v28/coates13.html.

Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. In *Machine Learning*. 273–297.

George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. 2013. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *International Conference on Acoustics, Speech and Signal Processing*. http://www.cs.toronto.edu/~gdahl/papers/reluDropoutBN_icassp2013.pdf.

Sorin Draghici. 2002. On the capabilities of neural networks using limited precision weights. *Neural Netw.* 15, 3 (2002), 395–414. DOI:http://dx.doi.org/10.1016/S0893-6080(02)00032-1

Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna V. Palem, Olivier Temam, and Chengyong Wu. 2014. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *Asia and South Pacific Design Automation Conference*.

Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *International Symposium on Microarchitecture*. 1–6.

Kevin Fan, Manjunath Kudlur, Ganesh S. Dasika, and Scott A. Mahlke. 2009. Bridging the computation gap between programmable processors and hardwired accelerators. In *HPCA*. IEEE Computer Society, 313–322.

Clement Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *CVPR Workshop*. IEEE, 109–116. DOI: http://dx.doi.org/10.1109/CVPRW.2011.5981829

Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *International Symposium on Computer Architecture*. ACM Press, New York, New York, 37. DOI: http://dx.doi.org/10.1145/1815961.1815968

Atif Hashmi, Andrew Nere, James Jamal Thomas, and Mikko Lipasti. 2011. A case for neuromorphic ISAs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/1950365.1950385

Geoffrey E. Hinton and N. Srivastava. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv* (2012), 1–18. http://arxiv.org/abs/1207.0580

Jordan L. Holi and Jenq-Neng Hwang. 1993. Finite precision error analysis of neural network hardware implementations. *IEEE Trans. Comput.* 42, 3 (1993), 281–290. DOI: http://dx.doi.org/10.1109/12.210171

Mark Holler, Simon Tam, Hernan Castro, and Ronald Benson. 1990. An electrically trainable artificial neural network (ETANN) with 10240 "floating gate" synapses. In *Artificial Neural Networks*. IEEE Press, Piscataway, NJ, 50–55. DOI: http://dx.doi.org/10.1109/IJCNN.1989.118698

Po-Sen Huang, Xiaodong He, Jianfeng Gao, and Li Deng. 2013. Learning deep structured semantic models for web search using clickthrough data. In *International Conference on Information and Knowledge Management*. http://dl.acm.org/citation.cfm?id=2505665

Muhammad Mukaram Khan, David R. Lester, Luis A. Plana, Alexander D. Rast, Xin Jin, Eustace Painkras, and Stephen B. Furber. 2008. SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *IEEE International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2849–2856. DOI: http://dx.doi.org/10.1109/IJCNN.2008.4634199

Joo-young Kim, Minsu Kim, Seungjin Lee, Jinwook Oh, Kwanho Kim, and Hoi-jun Yoo. 2010. A 201.4 GOPS 496 mW real-time multi-object recognition processor with bio-inspired neural perception engine. *IEEE Journal of Solid-State Circuits* 45, 1 (Jan. 2010), 32–45. DOI: http://dx.doi.org/10.1109/JSSC.2009.2031768

Eric J. King and Earl E. Swartzlander Jr. 1997. Data-dependent truncation scheme for parallel multipliers. In *Conference Record of the 31st Asilomar Conference on Signals, Systems & Computers*, Vol. 2. IEEE, 1178–1182.

Daniel Larkin, Andrew Kinane, Valentin Muresan, and Noel E O'Connor. 2006b. An efficient hardware architecture for a neural network activation function generator. In *ISNN (2)*, Jun Wang, Zhang Yi, Jacek M. Zurada, Bao-Liang Lu, and Hujun Yin (Eds.), Lecture Notes in Computer Science, Vol. 3973. Springer, 1319–1327.

Daniel Larkin, Andrew Kinane, and Noel E. O'Connor. 2006a. Towards hardware acceleration of neuroevolution for multimedia processing applications on mobile devices. In *ICONIP (3)*. 1178–1188.

Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. 2007. An empirical evaluation of deep architectures on problems with many factors of variation. In *International Conference on Machine Learning*. ACM Press, New York, New York, 473–480. DOI: http://dx.doi.org/10.1145/1273496.1273556

Quoc V. Le, MarcAurelio Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg S. Corrado, Jeffrey Dean, and Andrew Y. Ng. 2012. Building high-level features using large scale unsupervised learning. In *International Conference on Machine Learning*.

Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86 (1998). DOI: http://dx.doi.org/10.1109/5.726791

Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, 469–480. DOI: http://dx.doi.org/10.1145/1669112.1669172

Ahmed Al Maashri, Michael Debole, Matthew Cotter, Nandhini Chandramoorthy, Yang Xiao, Vijaykrishnan Narayanan, and Chaitali Chakrabarti. 2012. Accelerating neuromorphic vision algorithms for recognition. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)* (2012), 579. DOI:http://dx.doi.org/10.1145/2228360.2228465

Paul Merolla, John Arthur, Filipp Akopyan, Nabil Imam, Rajit Manohar, and D. S. Modha. 2011. A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm. In *IEEE Custom Integrated Circuits Conference*. IEEE, 1–4.

Volodymyr Mnih and Geoffrey Hinton. 2012. Learning to label aerial images from noisy data. In *Proceedings of the 29th International Conference on Machine Learning (ICML'12)*. 567–574.

Mike Muller. 2010. Dark silicon and the internet. In *EE Times "Designing with ARM" Virtual Conference*.

Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. 2013. Convolution engine: Balancing efficiency and flexibility in specialized computing. In *International Symposium on Computer Architecture*.

Johannes Schemmel, Johannes Fieres, and Karlheinz Meier. 2008. Wafer-scale integration of analog neural networks. In *International Joint Conference on Neural Networks*. IEEE, 431–438. DOI:http://dx.doi.org/10.1109/IJCNN.2008.4633828

Pierre Sermanet, Soumith Chintala, and Y. LeCun. 2012. Convolutional neural networks applied to house numbers digit classification. In *International Conference on Pattern Recognition*. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6460867.

Pierre Sermanet and Yann LeCun. 2011. Traffic sign recognition with multi-scale convolutional networks. In *International Joint Conference on Neural Networks*. IEEE, 2809–2813. DOI:http://dx.doi.org/10.1109/IJCNN.2011.6033589

Thomas Serre, Lior Wolf, Stanley Bileschi, Maximilian Riesenhuber, and Tomaso Poggio. 2007. Robust object recognition with cortex-like mechanisms. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29, 3 (March 2007), 411–26. DOI:http://dx.doi.org/10.1109/TPAMI.2007.56

Olivier Temam. 2012. A defect-tolerant accelerator for emerging high-performance applications. In *International Symposium on Computer Architecture*. Portland, Oregon.

Olivier Temam and Nathalie Drach. 1995. Software assistance for data caches. *Future Generation Computer Systems* 11, 6 (1995), 519–536. DOI:http://dx.doi.org/10.1016/0167-739X(95)00022-K

Shyamkumar Thoziyoor, Naveen Muralimanohar, and JH Ahn. 2008. CACTI 5.1. HP Labs, Palo Alto, Tech (2008). http://www.hpl.hp.com/techreports/2008/HPL-2008-20.pdf?q=cacti.

Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*.

Ganesh Venkatesh, Jack Sampson, Nathan Goulding-hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. 2011. QsCORES: Trading dark silicon for scalable energy efficiency with quasi-specific cores categories and subject descriptors. In *International Symposium on Microarchitecture*.

R. Jacob Vogelstein, Udayan Mallik, Joshua T. Vogelstein, and Gert Cauwenberghs. 2007. Dynamically reconfigurable silicon array of spiking neurons with conductance-based synapses. *IEEE Transactions on Neural Networks* 18, 1 (2007), 253–265.

Sami Yehia, Sylvain Girbal, Hugues Berry, and Olivier Temam. 2009. Reconciling specialization and flexibility through compound circuits. In *International Symposium on High Performance Computer Architecture*. IEEE, 277–288. DOI:http://dx.doi.org/10.1109/HPCA.2009.4798263

Received September 2014; revised November 2014; accepted December 2014