ECE 598: Machine Learning in Silicon Fall 2017

Case Study – DianNao

Naresh Shanbhag
Professor
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
DianNao Family: Energy-Efficient Hardware Accelerators for Machine Learning

By Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam
DianNao Family

Table 1. Accelerators in the DianNao family.

<table>
<thead>
<tr>
<th>Name</th>
<th>Process (nm)</th>
<th>Peak performance (GOP/s)</th>
<th>Peak power (W)</th>
<th>Area (mm²)</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>DianNao</td>
<td>65</td>
<td>452</td>
<td>0.485</td>
<td>3.02</td>
<td>Neural networks</td>
</tr>
<tr>
<td>DaDianNao</td>
<td>28</td>
<td>5585</td>
<td>15.97</td>
<td>67.73</td>
<td>Neural networks</td>
</tr>
<tr>
<td>ShiDianNao</td>
<td>65</td>
<td>194</td>
<td>0.32</td>
<td>4.86</td>
<td>Convolutional neural networks</td>
</tr>
<tr>
<td>PuDianNao</td>
<td>65</td>
<td>1056</td>
<td>0.596</td>
<td>3.51</td>
<td>Seven representative machine learning techniques</td>
</tr>
</tbody>
</table>

- DianNao [ASPLOS’14] → data reuse problem in DNN & CNN
- DaDianNao [MICRO’14] → eDRAM + support for training & inference
- ShiDianNao [ISCA’15] → CNN-based vision processing on mobile
- PuDianNao [ASPLOS’15] → extension to 7 ML applications (k-NN, k-means, DNN, logistic regression, SVM, naive Bayes, classification tree)
DianNao

- CNN/DNNs are mostly implemented on SIMD, GPU, or CPUs
- ASICs for CNN (LeCun) and multi-layer perceptrons (Yoo)
- previous focus → make computations efficient only
- DianNao focus → optimizes memory usage for CNN & DNNs
- DianNao → ‘computer’ in Mandarin
- 65nm design
- 496 (16-b) fixed-point ops every 1.02ns → 452 GOP/s
- $3.02\ mm^2$, 485 $mW$ (excluding main memory access)
- 118× faster and 21× more energy-efficient (incl. MM access) than a 128-bit SIMD clocked at 2GHz → based on simulations
• training costs not included → infrequent training argument
• speed-up and energy benefits for 10 of the largest layers in recent CNNs and DNNs
• sequential, hence, independent processing of layers
• sub-layers or feature maps (FMs) → input and output FMs
• three layer types: convolutional, pooling (subsampling), classifier
• $N_i$: number of input FMs; $N_n$: number of output FMs
• C-layer: $K_x \times K_y \times N_i$ kernels ($K_x \times K_y$: receptor field size)
• $s_x, s_y$: overlap between two consecutive windows
• kernel values → synaptic weights
• P-layer → subsampling layer → data aggregation, reduces dimensionality, average and max operations
• classification layer → linear or multi-layer (2-layer) perceptron
• no notion of FMs in classifier → FMs are flattened
Memory Bandwidth Requirements

• analyzes the data locality property of different layers
• assumes cache simulator feeding into a computational engine
• cache hierarchy → L1 (32KB, 64-byte line, 8-way), optional L2 (2MB, 64-byte, 8-way) → assume multiport caches that serve $4T_n$ (inputs); $4T_n T_i$ (synapses) bytes
• computational engine → sufficiently powerful to process $T_n$ neurons with $T_i$ synapses each every cycle
Original Classifier Code

```java
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]);
```

- processes $N_n$ neurons with $N_i$ synapses each every cycle
- total memory transfers $\rightarrow N_i N_n + N_i N_n + N_n$ $\rightarrow$ inputs loaded + synapses loaded + outputs written

- input neurons **reused** $\rightarrow$ too many (100s of Ks) to fit in local buffers $\rightarrow$ solution: use tiling
Tiled Code

- assume architecture can process \( T_n < N_n \) neurons with \( T_i < N_i \) synapses simultaneously
- \( T_{ii} \): input tile factor
- \( T_{nn} \): output tile factor
- trade-off between input and output reuse distance

```c
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)
            sum[n] = 0;
        for (int ii = iii; ii < iii + Tii; ii += Ti)
            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]);
    }
}
```
Impact of Tiling on Memory BW

- reduces input BW dramatically → slight increase in output BW
- synapses now dominate memory behavior → use L2 cache

**Tiled+L2**: using an L2 cache to store synapses

*Figure 6. Memory bandwidth requirements for each layer type (CONV3 has shared kernels, CONV5 has private kernels).*
• no synapse reuse within perceptron layer → all weights unique
• synapses reused for each new input data
• L2 cache can store millions of synapses → CLASS 1- Tiled+L2
Convolutional Layer Pseudo Code

• two reuse opportunities

1) sliding window → input reuse → $\frac{K_x K_y}{s_x s_y}$ reuses

2) reuse across $N_n$ output FMs → $N_n$ reuses

• tile 1) but not 2) → $K_x K_y N_i$

fits in L1 $(10 \times 10 \times 1000)$
• no kernel (no synapses); identical number of input & output FMs
• only sliding window reuse possible → less dramatic improvements as compared to convolutional layers
Realizing Small NNs

- direct mapped architecture is not scalable → area, energy, delay increases quadratically with the number of neurons
- TSMC 65nm GP library, standard $V_t$, Synopsys ICC for P&R
- recommends time-shared (folded) architecture with proper memory management

synthesis results
DianNao Accelerator Architecture

- staggered pipelining $\rightarrow$ 3rd stage active after all additions are completed

- local memory buffers for $\rightarrow$ inputs (NBin), outputs (NBout), synapses (SB)
- 3-stage (staggered) pipelined Neural Functional Unit (NFUs)
- C-layer and CLASS use all three stages; P-layer uses NFU-2
- NFU-2 $\rightarrow$ also has shifters and max for P-layer
• non-linearity $\rightarrow$ piece-wise linear interpolation

$$f(x) = a_i x + b_i; \quad x \in [x_i, x_{i+1}]$$

• two 16:1, 16-b multiplexers $\rightarrow$ boundary $[x_i, x_{i+1}]$ selection
• one 16-b (output) multiplier and one 16-b adder for interpolation
• 16 segment coefficients $(a_i, b_i)$ stored in a small RAM
• can implement $\tanh()$, ReLU and other functions too
Precision Requirements

- accuracy metric → MSE (SQNR)
- 16-b numbers → 6-b (integer part) + 10-b (fractional part)
  “ample evidence that even smaller operators have almost no impact on the accuracy…”
- truncation for arithmetic operations

Figure 12. 32-bit floating-point vs. 16-bit fixed-point accuracy for UCI data sets (metric: log(Mean Squared Error)).

Table 1. 32-bit floating-point vs. 16-bit fixed-point accuracy for MNIST (metric: error rate).

<table>
<thead>
<tr>
<th>Type</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit floating-point</td>
<td>0.0311</td>
</tr>
<tr>
<td>16-bit fixed-point</td>
<td>0.0337</td>
</tr>
</tbody>
</table>

Table 2. Characteristics of multipliers.

<table>
<thead>
<tr>
<th>Type</th>
<th>Area (μm²)</th>
<th>Power (μW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit truncated fixed-point multiplier</td>
<td>1309.32</td>
<td>576.90</td>
</tr>
<tr>
<td>32-bit floating-point multiplier</td>
<td>7997.76</td>
<td>4229.60</td>
</tr>
</tbody>
</table>
Storage

- NBin, NBout, SB → use SRAMs
- each has variable width requirements → $2T_n$ bytes (NBin, NBout; 2 bytes = 16 bits); $2T_n^2$ (SB)
- use dedicated buffers with custom buffer sizes for best time & energy per read → also helps with conflicts
- another option → highly associative cache (too expensive)
• 3 DMAs to exploit spatial locality → 2 load DMAs (NBin, SB), 1 store DMA (NBout)
• DMA requests sent to buffers in form of instructions → FIFO
• instructions issued as soon as DMA memory requests for previous instruction is complete
• decouples requests from current operations → preloading
• NBin and SB used for preloading and reuse → dual-port SRAM
Experimental Methodology

- **accelerator simulator**: custom cycle-accurate, bit-accurate C++ simulator → also measures time in number of cycles, plugged to a main memory model allowing a bandwidth of up to 250 GB/s.

- **CAD tools**: Verilog model synthesized via Synopsys Design Compiler in TSMC 65nm GP standard VT library, P&R via Synopsys ICC compiler, simulated via Synopsys VCS, power estimation via Prime-Time PX → area, energy, critical path delay estimates

- **SIMD baseline**: use GEM5+McPAT combination → 4-issue superscalar x86 core with a 128-bit (816-bit) SIMD unit (SSE/SSE2), clocked at 2GHz.
- **benchmarks**: largest convolutional, pooling and/or classifier layers of several recent and large neural network structures.

<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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV1</td>
<td>500</td>
<td>375</td>
<td>9</td>
<td>9</td>
<td>32</td>
<td>48</td>
<td>Street scene parsing (CNN) [13], (e.g., identifying “building”, “vehicle”, etc)</td>
</tr>
<tr>
<td>POOL1</td>
<td>492</td>
<td>367</td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>-</td>
<td>Detection of faces in YouTube videos (DNN) [26], largest NN to date (Google)</td>
</tr>
<tr>
<td>CLASS1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>960</td>
<td>20</td>
<td>Google Street View house numbers (CNN) [35]</td>
</tr>
<tr>
<td>CONV2*</td>
<td>200</td>
<td>200</td>
<td>18</td>
<td>18</td>
<td>8</td>
<td>8</td>
<td>Detection of faces in YouTube videos (DNN) [26], largest NN to date (Google)</td>
</tr>
<tr>
<td>CONV3</td>
<td>32</td>
<td>32</td>
<td>4</td>
<td>4</td>
<td>108</td>
<td>200</td>
<td>Traffic sign identification for car navigation (CNN) [36]</td>
</tr>
<tr>
<td>POOL3</td>
<td>32</td>
<td>32</td>
<td>4</td>
<td>4</td>
<td>100</td>
<td>-</td>
<td>Google Street View house numbers (CNN) [35]</td>
</tr>
<tr>
<td>CLASS3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>200</td>
<td>100</td>
<td>Google Street View house numbers (CNN) [35]</td>
</tr>
<tr>
<td>CONV4</td>
<td>32</td>
<td>32</td>
<td>7</td>
<td>7</td>
<td>16</td>
<td>512</td>
<td>Multi-Object recognition in natural images (DNN) [16], winner 2012 ImageNet competition</td>
</tr>
<tr>
<td>CONV5*</td>
<td>256</td>
<td>256</td>
<td>11</td>
<td>11</td>
<td>256</td>
<td>384</td>
<td>Multi-Object recognition in natural images (DNN) [16], winner 2012 ImageNet competition</td>
</tr>
<tr>
<td>POOL5</td>
<td>256</td>
<td>256</td>
<td>2</td>
<td>2</td>
<td>256</td>
<td>-</td>
<td>Multi-Object recognition in natural images (DNN) [16], winner 2012 ImageNet competition</td>
</tr>
</tbody>
</table>

Table 5. Benchmark layers (CONV=convolutional, POOL=pooling, CLASS=classifier; CONVx* indicates private kernels).
**Post Layout Numbers**

<table>
<thead>
<tr>
<th>Component or Block</th>
<th>Area in $\mu m^2$ (%)</th>
<th>Power in $mW$ (%)</th>
<th>Critical path in $ns$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCELERATOR</td>
<td>3,023,077 (20.14%)</td>
<td>485 (18.41%)</td>
<td>1.02</td>
</tr>
<tr>
<td>Combinational</td>
<td>608,842 (38.31%)</td>
<td>89 (36.59%)</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>1,158,000 (38.31%)</td>
<td>177 (27.16%)</td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td>375,882 (12.43%)</td>
<td>86 (17.84%)</td>
<td></td>
</tr>
<tr>
<td>Clock network</td>
<td>68,721 (2.27%)</td>
<td>132 (27.22%)</td>
<td></td>
</tr>
<tr>
<td>Filler cell</td>
<td>811,632 (26.85%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SB</td>
<td>1,153,814 (38.17%)</td>
<td>105 (22.65%)</td>
<td></td>
</tr>
<tr>
<td>NBin</td>
<td>427,992 (14.16%)</td>
<td>91 (19.76%)</td>
<td></td>
</tr>
<tr>
<td>NBout</td>
<td>432,966 (14.35%)</td>
<td>92 (19.97%)</td>
<td></td>
</tr>
<tr>
<td>NFU</td>
<td>846,563 (28.00%)</td>
<td>132 (27.22%)</td>
<td></td>
</tr>
<tr>
<td>CP</td>
<td>141,809 (5.69%)</td>
<td>31 (6.39%)</td>
<td></td>
</tr>
<tr>
<td>AXIMUX</td>
<td>9,767 (0.32%)</td>
<td>8 (2.65%)</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>9,226 (0.31%)</td>
<td>26 (5.36%)</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.** Characteristics of accelerator and breakdown by component type (first 5 lines), and functional block (last 7 lines).

- **NFU-1:** $T_n = 16; T_i = 16 \rightarrow 256$ 16-b multipliers
- **NFU-2:** 16 adder trees (15 adders each), 16-input shifter & max
- **NFU-3:** 16, 16-b truncated multipliers+16 adders
Speed-Up

- 118X for C and CLASS layers
- P-layers’ poor speed-up → only adder tree in NFU-2 is used
- analysis of speed-up provided

Figure 16. Speedup of accelerator over SIMD, and of ideal accelerator over accelerator.
• average energy reduction → 21×
• smaller (by 10×) than previously reported number comparing acc and proc. → energy costs of memory accesses were not reported
• got similar reduction if memory energy cost is set to 0 → consistent
• Accelerator: memory access dominate energy

Figure 18. Breakdown of accelerator energy.

• SIMD: 2/3 energy in computation and 1/3 in memory access

Figure 19. Breakdown of SIMD energy.
https://courses.engr.illinois.edu/ece598ns/fa2017

http://shanbhag.ece.uiuc.edu