Neural Acceleration for General-Purpose Approximate Programs

Hadi Esmaeilzadeh
Adrian Sampson
Luis Ceze
Doug Burger

University of Washington
Microsoft Research
computer vision
machine learning
sensory data
physical simulation
information retrieval
augmented reality
image rendering
Approximate computing

Probabilistic CMOS designs
[Rice, NTU, Georgia Tech...]

Stochastic processors
[Illinois]

Code perforation transformations
[MIT]

Relax software fault recovery
[de Kruijf et al., ISCA 2010]

Green runtime system
[Baek and Chilimbi, PLDI 2010]

Flikker approximate DRAM
[Liu et al., ASPLOS 2011]

EnerJ programming language
[PLDI 2011]

Truffle dual-voltage architecture
[ASPLOS 2012]

computer vision
machine learning
sensory data
physical simulation
information retrieval
augmented reality
image rendering
Accelerators

- Conservation
- Cores
- UCSD
- DySER
- Wisconsin
- CPU
- GPU
- BERET
- Michigan
- Vector Unit
- FPGA
- UCSD
Accelerators

- Conservation Cores
- UCSD
- DySER
  Wisconsin
- Vector Unit

Approximate computing

- computer vision
- machine learning
- sensory data
- physical simulation
- information retrieval
- augmented reality
- image rendering
An accelerator for approximate computations

- Mimics functions written in traditional languages!
- Runs more efficiently than a CPU or a precise accelerator!
- May introduce small errors!
Neural networks are function approximators

Trainable: implements many functions

Very efficient hardware implementations

Highly parallel

Fault tolerant

[Temam, ISCA 2012]
Neural acceleration

Program
Neural acceleration

Annotate an approximate program component
Neural acceleration

- Annotate an approximate program component
- Compile the program and train a neural network
Neural acceleration

1. **Annotate** an approximate program component
2. **Compile** the program and train a neural network
3. **Execute** on a fast Neural Processing Unit (NPU)
Neural acceleration

1. **Annotate** an approximate program component
2. **Compile** the program and train a neural network
3. **Execute** on a fast Neural Processing Unit (NPU)
4. **Improve** performance 2.3x and energy 3.0x on average
Programming model

```c
[[transform]]
float grad(float[3][3] p) {
    ...
}

void edgeDetection(Image &src, Image &dst) {
    for (int y = ...) {
        for (int x = ...) {
            dst[x][y] =
                grad(window(src, x, y));
        }
    }
}
```
Code region criteria

- ✔️ Hot code
- ✔️ Approximable
- ✔️ Well-defined inputs and outputs

\[ \text{grad}( ) \]

- run on every 3x3 pixel window
- small errors do not corrupt output
- takes 9 pixel values; returns a scalar
Empirically selecting target functions

Program

Accelerated Program
Compiling and transforming

1. Code Observation
2. Training
3. Code Generation

- Annotated Source Code
- Training Inputs
- Trained Neural Network
- Augmented Binary
Code observation

```c
[[NPU]]
float grad(float[3][3] p) {
...
}

void edgeDetection(Image &src,
                    Image &dst) {
    for (int y = ...) {
        for (int x = ...) {
            dst[x][y] = grad(window(src, x, y));
        }
    }
}
```

= record(p); record(result);

- **Test cases**
- **Instrumented program**
- **Sample arguments & outputs**

<table>
<thead>
<tr>
<th>p</th>
<th>grad(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>323, 231, 122, 93, 321, 49</td>
<td>53.2</td>
</tr>
<tr>
<td>49, 423, 293, 293, 23, 2</td>
<td>94.2</td>
</tr>
<tr>
<td>34, 129, 493, 49, 31, 11</td>
<td>1.2</td>
</tr>
<tr>
<td>21, 85, 47, 62, 21, 577</td>
<td>64.2</td>
</tr>
<tr>
<td>7, 55, 28, 96, 552, 921</td>
<td>18.1</td>
</tr>
<tr>
<td>5, 129, 493, 49, 31, 11</td>
<td>92.2</td>
</tr>
<tr>
<td>49, 423, 293, 293, 23, 2</td>
<td>6.5</td>
</tr>
<tr>
<td>34, 129, 72, 49, 5, 2</td>
<td>120</td>
</tr>
<tr>
<td>323, 231, 122, 93, 321, 49</td>
<td>53.2</td>
</tr>
<tr>
<td>6, 423, 293, 293, 23, 2</td>
<td>49.7</td>
</tr>
</tbody>
</table>
Training

Backpropagation

Training

Inputs
Training

- Faster
- Less robust

- Slower
- More accurate

70%  98%  99%
void edgeDetection(Image &src, Image &dst) {
    for (int y = ...) {
        for (int x = ...) {
            p = window(src, x, y);
            NPU_SEND(p[0][0]);
            NPU_SEND(p[0][1]);
            NPU_SEND(p[0][2]);
            ...
            dst[x][y] = NPU_RECEIVE();
        }
    }
}
Neural Processing Unit (NPU)
Software interface:
ISA extensions

Core

- \( \text{enq.} \) \( \text{d} \) \( \rightarrow \)

- \( \text{deq.} \) \( \text{d} \) \( \leftarrow \)

- \( \text{enq.} \) \( \text{c} \) \( \rightarrow \)

- \( \text{deq.} \) \( \text{c} \) \( \leftarrow \)

input

output

configuration

NPU
Microarchitectural interface

- Fetch
- Decode
- Issue
- Execute
- Memory
- Commit

_NPU configuration_

```plaintext
enq.d
S
NS
deq.d
configuration
enq.c
deq.c
```

NPU
A digital NPU

Bus Scheduler

Processing Engines

input

output
A digital NPU

- Multiply-add unit
- Neuron weights
- Accumulator
- Sigmoid LUT

Processing Engines

Input

Output
Experiments

Several benchmarks; annotated one hot function each
FFT, inverse kinematics, triangle intersection, JPEG, K-means, Sobel

Simulated full programs on MARSSx86
Energy modeled with McPAT and CACTI
Microarchitecture like Intel Penryn: 4-wide, 6-issue
45 nm, 2080 MHz, 0.9 V
Two benchmarks

- **edge detection**
  - 88 static instructions (56% of dynamic instructions)
  - 18 neurons

- **triangle intersection**
  - 1,079 static x86-64 instructions (97% of dynamic instructions)
  - 60 neurons, 2 hidden layers
Speedup with NPU acceleration

2.3x average speedup
Ranges from 0.8x to 11.1x
Energy savings with NPU acceleration

- 3.0x average energy reduction
- All benchmarks benefit

The diagram shows the energy reduction over all-CPU execution for various benchmarks:
- fft
- inversek2j
- jmeint
- jpeg
- kmeans
- sobel
- Geometric mean

The highest energy reduction is 21.1x for inversek2j.
Application quality loss

Quality loss below 10% in all cases
Based on application-specific quality metrics
Edge detection with gradient calculation on NPU
Also in the paper

Sensitivity to communication latency
Sensitivity to NN evaluation efficiency
Sensitivity to PE count
Benchmark statistics
All-software NN slowdown
Program
Neural networks can efficiently approximate functions from programs written in conventional languages.
CPU

flexible

low power

parallel

regular

fault-tolerant

analog

analogue
Normalized dynamic instructions

Dynamic instruction count normalized to original

fft
inversek2j
jmeint
jpeg
kmeans
sobel
geometric mean

NPU queue instructions
other instructions
Slowdown with software NN

20x average slowdown
Using off-the-shelf FANN library