Neural Acceleration for General-Purpose Approximate Programs

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, Doug Burger*
University of Washington, *Microsoft Research

Presented By: Peter Lindes, Bryan Stearns
Outline

- Overview
- Motivation
- Background
- Contributions
- Approximation through Neural Acceleration
- Architectural Implementation
- Evaluation
- Limitations
- Conclusion
- Discussion Questions
Overview

- Parrot transformation
- Neural Processing Unit (NPU)
- Offload approximate code to NPU
- Speedup: 2.3x
- Energy savings: 3.0x
- Loss of accuracy: \( \leq 9.6\% \)
Motivation

- Architectural innovation crucial for energy and performance gains
  - Cessation of Dennard Scaling
- Tension between efficiency and programmability
  - ASICs: 1000x improvement, design very expensive
  - Programmable Accelerators (GPUs and FPGAs): programming costly
  - Achieve efficiency at the cost of generality
- Tolerance to approximation
  - Many applications do not need high precision
  - Can be leveraged for substantial performance and energy gains
Related Work

- Emerging Accelerators:
  - BERET
  - Conservation Cores
  - Qs-Cores
  - DySER

- Approximate Computing
- General-Purpose Configurable Acceleration
- Neural Networks
Neural Networks

- Considerable Parallelism
Convolutional Neural Networks

Sobel Transform

\[
G_x = \begin{bmatrix}
-1 & 0 & +1 \\
-2 & 0 & +2 \\
-1 & 0 & +1
\end{bmatrix} \ast A \quad \text{and} \quad G_y = \begin{bmatrix}
-1 & -2 & -1 \\
0 & 0 & 0 \\
+1 & +2 & +1
\end{bmatrix} \ast A
\]
Contributions

- A new class of programmable accelerators
  - Approximation for better efficiency AND performance
- Can approximate small or large code blocks
  - From mainstream programming languages
- Leverages parallelism in neural networks
Proposed Approach

- Effective Trainable Accelerator
  - Neural Network approximation of imperative code
    - Parrot Transformation
  - Offload to Neural Processing Unit (NPU)
    - General speedup from parallelization
    - Possible ASIC implementation for energy efficiency
Proposed Approach

- Has Three Key Phases:
  - Programming
  - Compilation
  - Execution

- Automates almost all of the effort
  - Requires only annotations and trials from the developer
Explicitly annotate candidate functions
- Function calls will be replaced by compiler through IO redirection
- Compiler can determine actual efficacy of approximating

Candidate Criteria:
- Frequently Used (“Hot Code”)
- Tolerates Imprecision
- Well-Defined Input/Output
- “Pure”
  - Reads no data other than input
  - Affects no state other than output

```c
void edgeDetection(Image& srcImg, Image& dstImg){
    float [3][3] p; float pixel;
    for(int y = 0; y < srcImg.height; ++y)
        for(int x = 0; x < srcImg.width; ++x)
            srcImg.toGrayScale(x, y);
    for(int y = 0; y < srcImg.height; ++y)
        for(int x = 0; x < srcImg.width; ++x){
            p = srcImg.build3x3Window(x, y);
            pixel = sobel(p);
            dstImg.setPixel(x, y, pixel);
        }
}
```
Parrot Transformation

Converts imperative code into a Neural Network

Three Steps:
- Observation
- Training
- Binary Generation
The Compiler:
- Produces binary containing probes
- Collects input-output data pairs
- Uses data for training in next stage

Similar to profile-guided compilation

Often very few trials needed
Builds a 3-4 layer *Multilayer Perceptron* (MLP) neural network on training data
- Chooses best of 30 possible network structures
  - 1 or 2 hidden layers
  - From 2 to 32 (powers of 2), nodes per hidden layer
- Weight selection through backpropagation
- Iterative gradient-descent weight-adjustment
Compiler replaces function calls with NPU invocation
- One SEND per network input
- One RECEIVE per network output
- Compiler can choose not to replace code if can’t approximate well
- Count of approximated functions are limited
- One neural configuration per program per NPU

```c
void edgeDetection(Image& srcImg, Image& dstImg){
    float[3][3] p; float pixel;
    for(int y = 0; y < srcImg.height; ++y)
        for(int x = 0; x < srcImg.width; ++x)
            srcImg.toGrayScale(x, y);
    for(int y = 0; y < srcImg.height; ++y)
        for(int x = 0; x < srcImg.width; ++x){
            p = srcImg.build3x3Window(x, y);
            pixel = sobel(p);
            dstImg.setPixel(x, y, pixel);
        }
}
```
Architectural Implementation

- ISA Interface
  - Configuration I/O – *Set up Neural Net*
  - Data I/O – *Feed and Read*

- NPU (Neural Processing Unit)
  - Hardware choice (ASIC)
  - Internal Organization
  - Processing Engines (PEs)
Hardware Choice

- Digital ASIC
  \textit{(Application Specific Integrated Circuit)}
  - Low-latency
  - Power-efficiency

Other options

- Further efficiency though sub-critical voltage
- Analog (ASIC, FPAA) – \textit{Future Work}
ISA Interface

- Four new ISA instructions:
  - `enq.c %r` -- Enqueue from `r` to Config FIFO
  - `deq.c %r` -- Dequeue from Config FIFO to `r`
  - `enq.d %r` -- Enqueue from `r` to Input FIFO
  - `deq.d %r` -- Dequeue from Output FIFO to `r`

- Config Data:
  - Topology
  - PE Scheduling
  - Scaling
  - Weights
Internal Organization

- 8 Processing Elements (PEs)
  - Each neuron assigned a PE
- Bus Scheduler
  - Precise I/O ordering among neurons and queues
  - Defined in configuration
- Circular Schedule Buffer
  - Input -> PE
  - PE -> PE
  - PE -> Output
- Scaling Unit
  - Scale network input/output as necessary
Processing Elements

- Circular Weight Buffer
  - Input aligned with weights
  - Completion bit with each weight entry
- When Sum complete, pass to Sigmoid Unit
- Lookup table
- Output Register File
  - Takes final result
  - Index stored in Weight Buffer
Evaluation

- Number of PEs
- Approximation Error
- Speedup
- Sensitivity to Latency
- Energy Reduction
Number of Processing Elements

- Relative speedup per doubling
- Beyond 8 yields < 5% gain

EVALUATION
Errors measured via app-specific metrics

Range of 3-10% commensurate with other work

- Manual Video Encoder (0.5-10%)
- Green (1-20%)
- Truffle (3-10%, >80%)

### EVALUATION

#### Approximation Error

<table>
<thead>
<tr>
<th></th>
<th># of Function Calls</th>
<th># of Loops</th>
<th># of ifs/els</th>
<th># of x86-64 Instructions</th>
<th>Training Input Set</th>
<th>Neural Network Topology</th>
<th>NN MSE</th>
<th>Error Metric</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>34</td>
<td>32768 Random Floating Point Numbers</td>
<td>1 -&gt; 4 -&gt; 4 -&gt; 2</td>
<td>0.00002</td>
<td>Average Relative Error</td>
<td>7.22%</td>
</tr>
<tr>
<td>inverse2j</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>10000 (x,y) Random Coordinates</td>
<td>2 -&gt; 8 -&gt; 2</td>
<td>0.00563</td>
<td>Average Relative Error</td>
<td>7.50%</td>
</tr>
<tr>
<td>jmeint</td>
<td>32</td>
<td>0</td>
<td>23</td>
<td>1,079</td>
<td>10000 Random Pairs of 3D Triangle Coordinates</td>
<td>18 -&gt; 32 -&gt; 8 -&gt; 2</td>
<td>0.00530</td>
<td>Miss Rate</td>
<td>7.32%</td>
</tr>
<tr>
<td>jpeg</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1,257</td>
<td>Three 512x512-Pixel Color Images</td>
<td>64 -&gt; 16 -&gt; 64</td>
<td>0.00890</td>
<td>Image Diff</td>
<td>9.56%</td>
</tr>
<tr>
<td>kmeans</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>26</td>
<td>50000 Pairs of Random (r, g, b) Values</td>
<td>6 -&gt; 8 -&gt; 4 -&gt; 1</td>
<td>0.00169</td>
<td>Image Diff</td>
<td>6.18%</td>
</tr>
<tr>
<td>sobel</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>88</td>
<td>One 512x512-Pixel Color Image</td>
<td>9 -&gt; 8 -&gt; 1</td>
<td>0.00234</td>
<td>Image Diff</td>
<td>3.44%</td>
</tr>
</tbody>
</table>
“80% to 100% of each transformed application’s output elements have error less than 10%”
- Ideal Case:
  - Most instructions in substitution
  - Few I/O messages needed per dynamic substitution

EVALUATION

Speedup

94% fewer instructions
Less than ideal: Local focus
  "Hot" code only a few operations
  Communication overhead high
- Percentage speedup using described NPU
- Percentage speedup with “ideal NPU”
  - Approached using better chips
Sensitivity to Latency

Computation vs Communication

Flat => Computation Dominates
How does the example ASIC NPU compare to zero energy cost for replaced code?
Limitations

- **Applicability**
  - Candidate code requirements are imperfect

- **Programmer Effort**
  - Interest in automated training data generation

- **Quality and Error Control**
  - No guarantee about worst-case accuracy
Neural Accelerators can mimic a wide range of approximable imperative code regions
- The Parrot Transformation was successful for all tested candidate code regions

The presented methods achieve
- 2.3x speedup
- 3.0x energy savings
- Greater than 90% accuracy in all cases

Future work is directed toward better NPU hardware designs
Questions?
Discussion Questions

- Will this approach apply widely enough to justify the cost of the 1\textsuperscript{st} transistor?
- Would you as a programmer want to use the Parrot transformation?
- Is there unexploited parallelism in many relevant applications?
- How expensive is this – does it really compete with “vanilla” parallel designs?