Neural Acceleration for General-Purpose Approximate Programs

Paper authors: Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, Doug Burger

Presentation authors: Helen Liu, Rebecca Dettmar, Ricky Cui

Author Background

- Hadi Esmaeilzadeh
  - UC San Diego
- Adrian Sampson
  - Cornell University
- Luis Ceze
  - University of Washington
- Doug Burger
  - Managing director and corporate vice president of Microsoft Research’s core research labs worldwide (IEEE Xplore bio).




Introduction

This paper presents a new technique for harnessing NPUs for general-purpose computations

- Uses NPU to replace functions
- Parrot transformation
- Achieved:
  - 2.3x average speedup
  - 3.0x average energy savings for experimental benchmarks
  - Average accuracy greater than 90%

Three Main Challenges

1. Learning algorithm that accurately mimics imperative code
    a. Found neural network
    b. Propose Parrot Transformation
2. Language and compilation framework
    a. Programming model + compilation workflow
    b. Collect training dataset, train NNs, and replace code
3. Architectural interface for NPU Acceleration
    a. Tightly coupled NPU + out-of-order core enables pipeline processing
    b. ISA extensions to communicate neural configuration and runtime invocations

Overview - Parrot Transformation

3 Stages of Parrot Transformation:

1. Programming Stage
2. Compilation Stage
3. Execution Stage

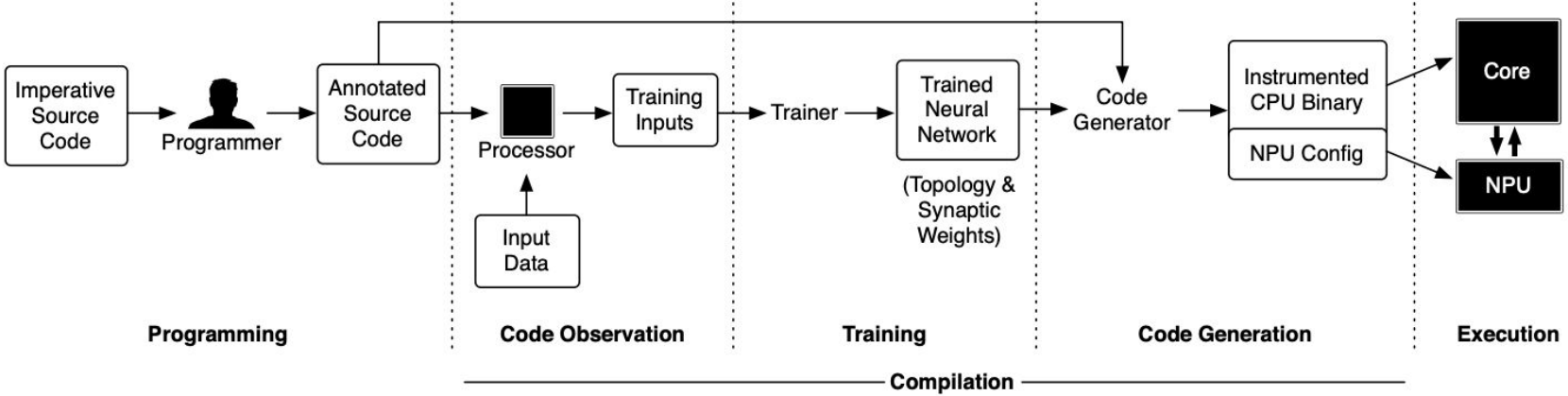


Figure 1: The Parrot transformation at a glance: from annotated code to accelerated execution on an NPU-augmented core.

Programming Model

Criterion for candidate function:

1. Frequently executed. Overhead of NPU invocation must be less than transformation profit

2. Must tolerate imprecision in its computation

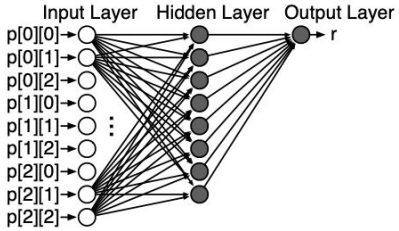
3. Must have well-defined inputs and outputs
    a. Fixed size inputs and outputs at compile time
    b. Pure function: Reads only its inputs, and writes only its return values (no side effects)

Code Example - Original vs. Parrot Transformed

```
1  float sobel [[PARROT]] (float[3][3] p){
     float x, y, r;
3    x = (p[0][0] + 2 * p[0][1] + p[0][2]);
     x = (p[2][0] + 2 * p[2][1] + p[2][2]);
5    y = (p[0][2] + 2 * p[1][2] + p[2][2]);
     y = (p[0][0] + 2 * p[1][1] + p[2][0]);
7    r = sqrt(x * x + y * y);
     if (r >= 0.7071) r = 0.7070;
9    return r;
   }
```

```
   void edgeDetection(Image& srcImg, Image& dstImg){
2    float[3][3] p; float pixel;
     for(int y = 0; y < srcImg.height; ++y)
4      for(int x = 0; x < srcImg.width; ++x)
         srcImg.toGrayeScale(x,y);
6    for(int y = 0; y < srcImg.height; ++y)
       for(int x = 0; x < scrImg.width; ++x){
8        p = srcImg.build3x3Window(x, y);
         pixel = sobel(p);
10       dstImg.setPixel(x, y, pixel);
       }
12 }
```

(a) Original implementation of the Sobel filter



(b) The sobel function transformed to a $9 \rightarrow 8 \rightarrow 1$ NN

```
   void edgeDetection(Image& srcImg, Image& dstImg){
2    float[3][3] p; float pixel;
     for(int y = 0; y < srcImg.height; ++y)
4      for(int x = 0; x < srcImg.width; ++x)
         srcImg.toGrayeScale(x,y);
6    for(int y = 0; y < srcImg.height; ++y)
       for(int x = 0; x < scrImg.width; ++x){
8        p = srcImg.build3x3Window(x, y);
         NPU_SEND(p[0][0]); NPU_SEND(p[0][1]); NPU_SEND(p[0][2]);
10       NPU_SEND(p[1][0]); NPU_SEND(p[1][1]); NPU_SEND(p[1][2]);
         NPU_SEND(p[2][0]); NPU_SEND(p[2][1]); NPU_SEND(p[2][2]);
12       NPU_RECEIVE(pixel);
         dstImg.setPixel(x, y, pixel);
14     }
   }
```

(c) parrot transformed code; an NPU invocation replaces the function call

Implementation - Compilation Workflow

Code Observation

1. Collect training dataset by logging inputs and corresponding outputs
2. Representative inputs must be provided by the programmer
    a. May be from a test suite or randomly generated

Training

1. Selects from a fixed set of neuron configurations (30 possible)
2. Trains all of them at once
3. Use backpropagation and gradient descent with learning rate 0.01
4. Test for 5000 epochs

Implementation - Compilation Workflow

Neural network topology search

1. Search neural network architecture(#hidden layers, #neurons).
   - Selection criterion: accuracy, NPU latency.
   - Cross validation.
   - 70% training, 30% testing.
   - Off-site training (Online training would increase overhead)
2. To reduce search space: ≤ 2 hidden layers, neurons = powers of 2 up to 32
3. Results: trained neural network + weights

Code Generation

1. Generate NPU configuration

2. Replace function with NPU instructions

Architecture Design for NPU Acceleration

Overhead of NPU invocation < acceleration profits -> tightly coupled NPU accelerator

ISA Support for NPU Acceleration: how CPU invokes NPU
FIFO-based interface
3 FIFO queues:

1. Config FIFO - send NPU configurations
2. Input FIFO - send inputs
3. Output FIFO - read outputs

Steps:
1. configure NPU
2. send inputs
3. NPU compute
4. read outputs

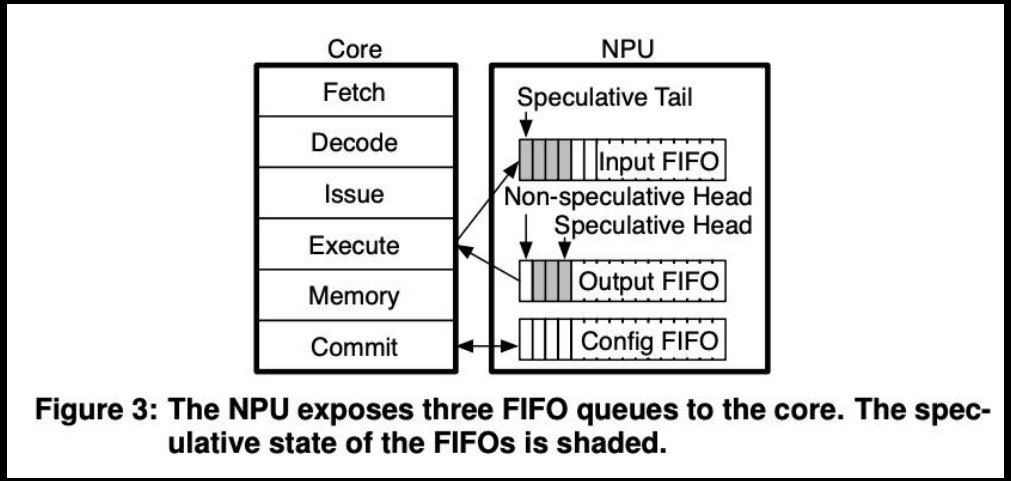


Figure 3: The NPU exposes three FIFO queues to the core. The speculative state of the FIFOs is shaded.

Architecture Design for NPU Acceleration

Commands:

- enq.c %r: enqueues the value of the register r into the config FIFO.
- deq.c %r: dequeues a configuration value from the config FIFO to the register r.
- enq.d %r: enqueues the value of the register r into the input FIFO.
- deq.d %r: dequeues the head of the output FIFO to the register r.

Process:

CPU: enq.c(configure NN), enq.d(send inputs)

NPU: compute neural network

Output FIFO: deq.d(CPU reads the output)

Architecture Design for NPU Acceleration

Modern CPU: out-of-order, speculative execution.
NPU must support this, or the pipeline will stall.

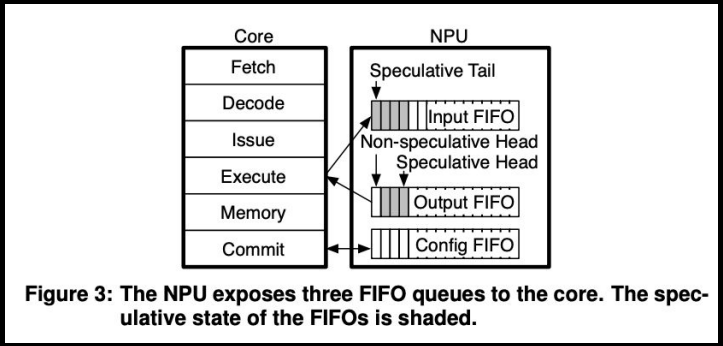


Figure 3: The NPU exposes three FIFO queues to the core. The speculative state of the FIFOs is shaded.

Speculative execution support:

- CPU speculatively enqueue inputs.
- Input FIFO header pointer only updates when instruction commits.
- Output FIFO handles both speculative head and non-speculative head pointers.

- If the branch mispredict -> rollback.
- If misprediction recovery, CPU tells NPU which instructions to squash, and NPU invalidates the output.
- If interrupted, NPU flush. Recommend to disable interrupt when NPU invoke.
- Enq.c and enq.d are not executed speculatively.
- If the context switches, NPU must save and restore state.

Neural Processing Unit

Reconfigurable NPU tightly coupled to the processor

- Each neuron is assigned to a PE – network topology determines PE scheduling
- Bus scheduling information (Input->PE, PE->PE, PE->Output) stored in the circular scheduling buffer
- During configuration neuron weights are placed into the weight buffer – compiler-directed schedule ensures that weights & inputs arrive in corresponding order

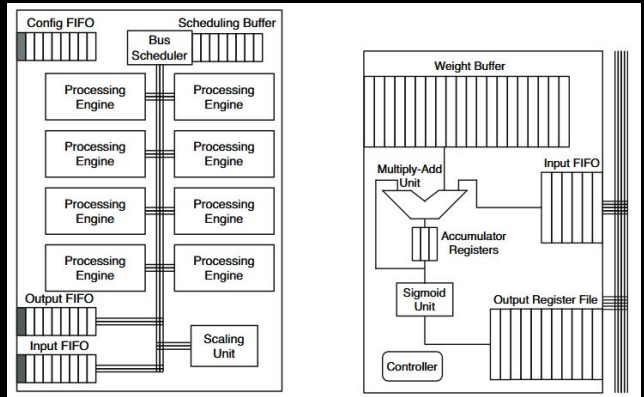


Figure 5: Reconfigurable 8-PE NPU.

Evaluation

Six benchmarks from domains useful to general applications and tolerance for imprecision, and which do not typically use neural networks

A single function with fixed inputs and outputs and no side effects is selected from each for Parrot transformation

Each network is trained on either typical program data (e.g., sample images) or some amount of random input within a permissible range and evaluated with different data of the same type

Evaluation

Table 1: The benchmarks evaluated, characterization of each transformed function, 0 data, and the result of the Parrot transformation.

	Description	Type	Evaluation Input Set	# of Function Calls	# of Loops	# of ifs/ elses	# of x86-64 Instructions	Training Input Set	Neural Network Topology	NN MSE	Error Metric	Error
fft	Radix-2 Cooley-Tukey fast Fourier	Signal Processing	2048 Random Floating Point Numbers	2	0	0	34	32768 Random Floating Point Numbers	1 -> 4 -> 4 -> 2	0.00002	Average Relative Error	7.22%
inversek2j	Inverse kinematics for 2-joint arm	Robotics	10000 (x,y) Random Coordinates	4	0	0	100	10000 (x,y) Random Coordinates	2 -> 8 -> 2	0.00563	Average Relative Error	7.50%
jmeint	Triangle intersection detection	3D Gaming	10000 Random Pairs of 3D Triangle Coordinates	32	0	23	1,079	100000 Random Pairs of 3D Triangle Coordinates	18 -> 32 -> 8 -> 2	0.00530	Miss Rate	7.32%
jpeg	JPEG encoding	Compression	220x200-Pixel Color Image	3	4	0	1,257	Three 512x512-Pixel Color Images	64 -> 16 -> 64	0.00890	Image Diff	9.56%
kmeans	K-means clustering	Machine Learning	220x200-Pixel Color Image	1	0	0	26	50000 Pairs of Random (r, g, b) Values	6 -> 8 -> 4 -> 1	0.00169	Image Diff	6.18%
sobel	Sobel edge detector	Image Processing	220x200-Pixel Color Image	3	2	1	88	One 512x512-Pixel Color Image	9 -> 8 -> 1	0.00234	Image Diff	3.44%

Results

Whole-application average error rates range between 3% and 10%, which is comparable to other hardware approximation techniques.

NPU acceleration can elide much CPU work, but the cost of evaluation and the time for queuing instructions limit the actual benefit – algorithms where significant chunks of code can be replaced with simple neural networks see the most benefit, while those where the “hot” code is more local and requires more complex models see less.

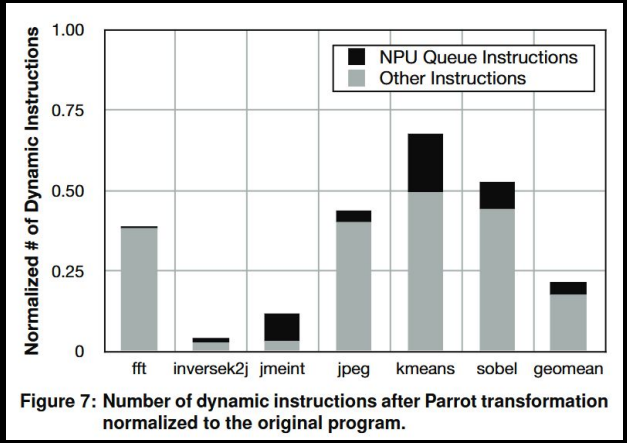
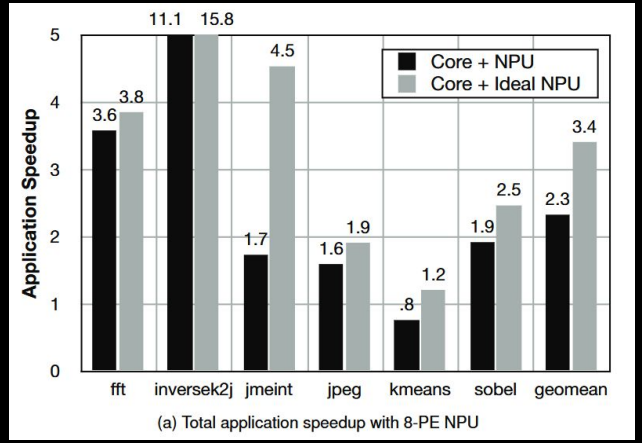


Figure 7: Number of dynamic instructions after Parrot transformation normalized to the original program.



(a) Total application speedup with 8-PE NPU

Results

It is also possible to run transformed programs on the CPU with a software model.

All benchmarks show a significant slowdown when the Parrot transformation is used as such.

The Parrot transformation requires efficient neural network execution such as hardware acceleration in order to be useful.

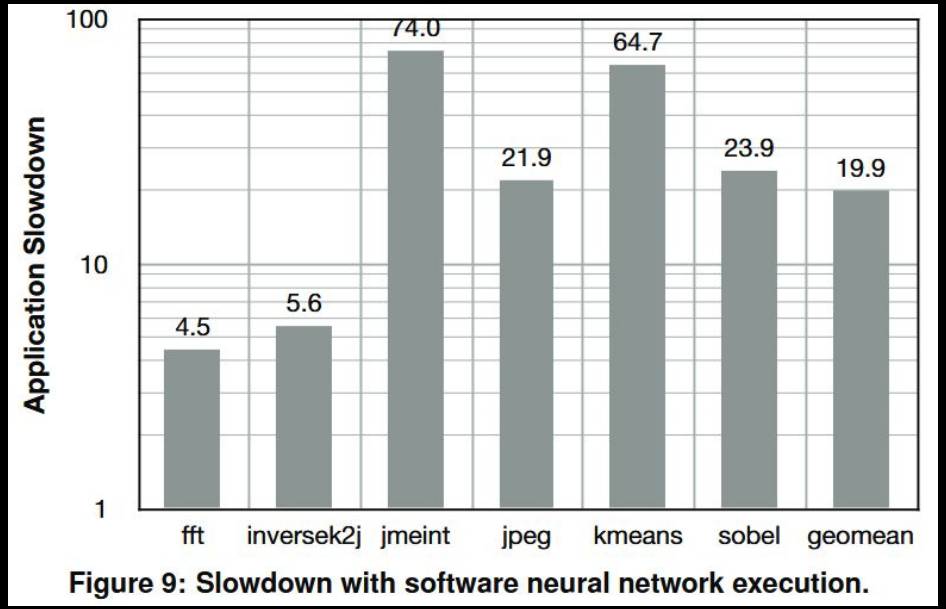


Figure 9: Slowdown with software neural network execution.

Results

A single NPU contains eight processing engines.

Performance across these benchmarks sees diminished returns as the number of PEs increases.

Less than 5% geometric mean improvement per doubling after eight PEs, does not justify complexity increase in adding more.

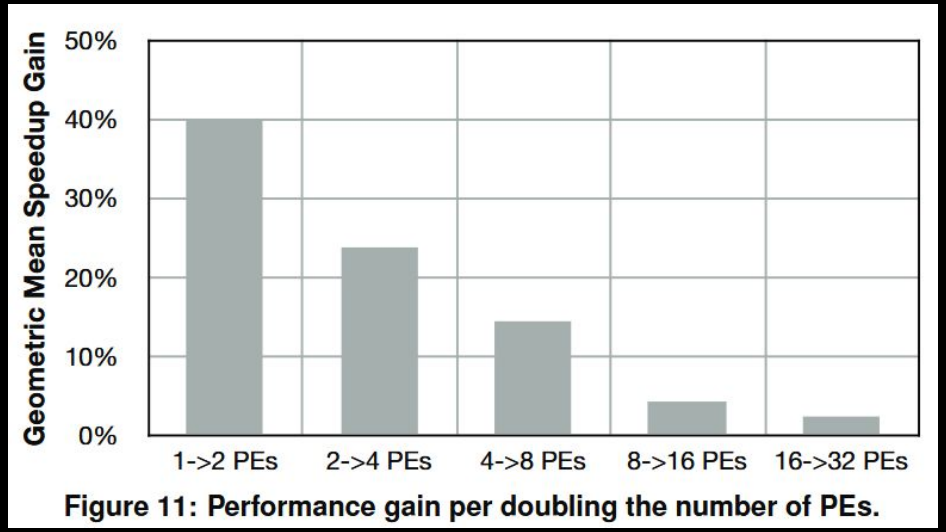


Figure 11: Performance gain per doubling the number of PEs.

Limitations

- Limited applicability to:
  - Hot regions of code
  - Approximable
  - Bounded number of inputs and outputs
- Requires programmer intervention
  - Analysis tools can help
  - Cannot be completely automatic
- Unpredictability
  - Non-deterministic compilation
  - In vivo training makes runtime non-deterministic
  - Errors up to 10%

Questions?