A Configurable Cloud-Scale DNN Processor for Real-Time AI

Requirements for Real-Time AI

DNNs are challenging to serve in interactive services

Convolutional Neural Networks (CNNs)  Recurrent Neural Networks (RNNs)

Neural processing units (NPUs) are promising for real-time AI

Must satisfy 3 requirements:
1. Low-latency
2. Flexible for long shelf life
3. Programmable and easy to use
Design Tenets for a Real-Time NPU

Extract parallelism from a single thread of execution

Achieve high utilization without batching

Scale to $O(100k)$ spatial units

Synthesis specialization
Project Brainwave NPU

High throughput, no batching without sacrificing latency, flexibility

Achieves 48 TFLOPS (96,000 MACs) on Intel Stratix 10 FPGAs

Hardware utilization at batch=1 up to 75%

DeepBench RNNs served in under 4 ms, ResNet-50 in under 2 ms

Specialized and deployed on FPGAs at cloud scale
System Architecture

Scalar processor loosely coupled to M*V processor
BW-NPU is network attached for scale out
Hardware microservices can be elastically allocated for real-time AI
Scalable on-chip model pinning to alleviate memory bottlenecks
Design Tenets for a Real-Time NPU

Extract parallelism from a single thread of execution

Achieve high utilization without batching

Scale to $O(100k)$ spatial units

Synthesis specialization
Critical Path Methodology

Model:
Long short-term memory (LSTM)

Applications:
NLP, speech, etc.

Type:
Recurrent neural network (RNN)
Design Tenets for a Real-Time NPU

Extract parallelism from a single thread of execution

**Achieve high utilization without batching**

Scale to O(100k) spatial units

Synthesis specialization
# Matrix * Vector

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Maps to</th>
<th>Data Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector * Vector</td>
<td>CNNs, RNNs</td>
<td>$O(0)$</td>
</tr>
<tr>
<td>Matrix * Matrix</td>
<td>CNNs, Batched RNNs</td>
<td>$O(N^2)$</td>
</tr>
</tbody>
</table>
# Matrix * Vector

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Maps to</th>
<th>Data Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector * Vector</td>
<td>CNNs, RNNs</td>
<td>$O(0)$</td>
</tr>
<tr>
<td>Matrix * Vector</td>
<td>CNNs, RNNs</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Matrix * Matrix</td>
<td>CNNs, Batched RNNs</td>
<td>$O(N^2)$</td>
</tr>
</tbody>
</table>
Design Tenets for a Real-Time NPU

Extract parallelism from a single thread of execution

Achieve high utilization without batching

Scale to $O(100k)$ spatial units

Synthesis specialization
ISA Design Tenets

Objectives

Leverage massive parallelism with a single thread of control
Reduce implementation complexity

Balance instruction granularity and flexibility

All instructions operate on multiples of a native dimension N
Abstracted to M-V and V-V operations
Composable into diverse DNNs: RNNs, 1-D CNNs, 2-D CNNs, MLPs, etc.

Instruction chaining

Instruction results are forwarded to the next instruction; stores are explicit
Enables long chains without dependency analysis or multi-ported register files
### Common Instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>IN</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_rd</td>
<td>Vector read</td>
<td>-</td>
<td>MemID</td>
<td>Memory index</td>
<td>V</td>
</tr>
<tr>
<td>v_wr</td>
<td>Vector write</td>
<td>V</td>
<td>MemID</td>
<td>Memory index</td>
<td>-</td>
</tr>
<tr>
<td>m_rd</td>
<td>Matrix read</td>
<td>-</td>
<td>MemID (NetQ or DRAM only)</td>
<td>Memory index</td>
<td>M</td>
</tr>
<tr>
<td>m_wr</td>
<td>Matrix write</td>
<td>M</td>
<td>MemID (MatrixRf or DRAM only)</td>
<td>Memory index</td>
<td>-</td>
</tr>
<tr>
<td>mv_mul</td>
<td>Matrix-vector multiply</td>
<td>V</td>
<td>MatrixRf index</td>
<td>-</td>
<td>V</td>
</tr>
<tr>
<td>vv_add</td>
<td>PWV addition</td>
<td>V</td>
<td>AddSubVrf index</td>
<td>-</td>
<td>V</td>
</tr>
<tr>
<td>vv_a_sub_b</td>
<td>PWV subtraction, IN is minuend</td>
<td>V</td>
<td>AddSubVrf index</td>
<td>-</td>
<td>V</td>
</tr>
<tr>
<td>vv_b_sub_a</td>
<td>PWV subtraction, IN is subtrahend</td>
<td>V</td>
<td>AddSubVrf index</td>
<td>-</td>
<td>V</td>
</tr>
<tr>
<td>vv_max</td>
<td>PWV max</td>
<td>V</td>
<td>AddSubVrf index</td>
<td>-</td>
<td>V</td>
</tr>
<tr>
<td>vv_mul</td>
<td>Hadamard product</td>
<td>V</td>
<td>MultiplyVrf index</td>
<td>-</td>
<td>V</td>
</tr>
<tr>
<td>v_relu</td>
<td>PWV ReLU</td>
<td>V</td>
<td>-</td>
<td>-</td>
<td>V</td>
</tr>
<tr>
<td>v_sigm</td>
<td>PWV sigmoid</td>
<td>V</td>
<td>-</td>
<td>-</td>
<td>V</td>
</tr>
<tr>
<td>v_tanh</td>
<td>PWV hyperbolic tangent</td>
<td>V</td>
<td>-</td>
<td>-</td>
<td>V</td>
</tr>
<tr>
<td>s_wr</td>
<td>Write scalar control register</td>
<td>-</td>
<td>Scalar reg index</td>
<td>Scalar value</td>
<td>-</td>
</tr>
<tr>
<td>end_chain</td>
<td>End instruction chain</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

PWV = point-wise vector operation. IN = implicit input (V: vector, M: matrix, -: none). OUT = implicit output.
Example LSTM Program

```c
void LSTM(int steps) {
    for (int t = 0; t < steps; t++) {
        v_rd(NetQ);
        v_wr(InitialVrf, xt);
        v_rd(InitialVrf, xt);
        mv_mul(Wf);
        vv_add(bf);
        v_wr(AddSubVrf, xWf);
        v_rd(InitialVrf, h_prev);
        mv_mul(Uf);
        vv_add(xWf);
        v_sigm();
        vv_mul(c_prev);
        v_wr(AddSubVrf, ft_mod);
        v_rd(InitialVrf, h_prev);
        mv_mul(Uc);
        vv_add(xWc);
        v_tanh();
        vv_mul(it);
        vv_add(ft_mod);
        v_wr(MultiplyVrf, c_prev);
        v_wr(InitialVrf, ct);
        v_rd(InitialVrf, ct);
        v_tanh();
        vv_mul(ot);
        v_wr(InitialVrf, ct);
        v_tanh();
        vv_mul(it);
        v_wr(InitialVrf, h_prev);
        v_wr(NetQ);
    }
}
```
Design Tenets for a Real-Time NPU

Extract parallelism from a single thread of execution

Achieve high utilization without batching

Scale to $O(100k)$ spatial units

Synthesis specialization
Microarchitecture

Executes a continuous stream of instruction chains

Function units arranged to mirror chain structure

Chaining reduces latency of critical path
Microarchitecture

Executes a continuous stream of instruction chains
Function units arranged to mirror chain structure
Chaining reduces latency of critical path

Scaling M*V
Memory bandwidth
Tile engines
Data types
Scheduling

1. Read Vector x
2. M*V by W
3. Add Vector by b
4. Vector Tanh
5. Multiply Vector by i
6. Add Vector by f
7. Vector Tanh
8. Multiply by o
9. Write Vector h
Scaling M*V: Single Spatial Unit

Start with a primitive 1 MAC for M*V
Vector and matrix register files (VRF, MRF) read 1 word/cycle
Scaling M*V: Multi-Lane Vector Spatial Unit

Compute lanes x8 = 8 MACs
Column parallelism
Scaling limit: more lanes = bigger RFs
Scaling M*V: Vector Spatial Unit Replication

Replicate DPE x4 = 32 MACs
Matrix row parallelism, distributed MRF
Reuse VRF data across dot products
Scaling limit: rows = DPEs
Scaling M*V: Scalable Replication

Tile engine: native size M*V tile

High precision accumulator (HP ACC)
Registered input fan-out tree
Result fan-in tree (muxes)
Scaling M*V: Tiling

Large matrices: tile parallelism
Add-reduction unit
Scaling limit: area, matrix columns
Scaling M*V: Narrow Precision Data Types

FP8 - FP11 sufficient
e.g., 1-bit sign, 5-bit exponent, 2-bit mantissa

Block floating point
Shared exponent for native vectors
Integer arithmetic in tile engines
Scaling M*V: Narrow Precision

Fully scaled Block FP8 on Stratix 10

- 6 Tile Engines
- 400 dot products each
- 40 lanes each
- **Total** = 6 * 400 * 40 = **96,000 MACs**

MRF is 96,000 words/cycle
Scaling M*V: Scheduling

Dynamic parallelism from single thread of execution
Avoid global cycle-by-cycle scheduling
Inter-module is asynchronous (dataflow control)
Hierarchical dispatch and decode

From MVM

Dispatcher

VRF

Hazard checking
Addresses generation

D

Acc signals
Matrix read address
Leverages fanout tree

D

Mux selects

D

HP
ACC

D

Accumulate vs.
passthrough vs.
clear
Scaling M*V: Scheduling

Scalar Processor

Top Level Scheduler

MVM Scheduler

Instructions

MFU0
MFU1
etc

Vector Arbitration
Design Tenets for a Real-Time NPU

Extract parallelism from a single thread of execution

Achieve high utilization without batching

Scale to $O(100k)$ spatial units

Synthesis specialization
Real-Time AI Evaluation

DeepBench RNN: GRU-2816, batch=1, 71B OPs/serve

<table>
<thead>
<tr>
<th>Device</th>
<th>Node</th>
<th>Latency</th>
<th>Effective TFLOPS</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>BW-NPU (Stratix 10, FP8)</td>
<td>Intel 14nm</td>
<td>2 ms</td>
<td>35.9</td>
<td>74.8%</td>
</tr>
</tbody>
</table>

CNN: ResNet-50, batch=1, 7.7B OPs/serve

<table>
<thead>
<tr>
<th>Device</th>
<th>Node</th>
<th>Latency</th>
<th>Effective TFLOPS</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>BW-NPU (Arria 10, FP11)</td>
<td>TSMC 20nm</td>
<td>1.64 ms</td>
<td>4.7</td>
<td>66%</td>
</tr>
</tbody>
</table>
Conclusions

BW-NPU is in scale production at Microsoft
Powering Microsoft services such as Bing search
Serving real-time CNNs to Azure customers

200M Images, 20TB
Land cover mapping for the whole of US in
10+ minutes
Try the Project Brainwave NPU on Azure
https://github.com/Azure/aml-real-time-ai