Hardware Specialization in Deep Learning

CSE590W 18Sp, Thursday April 19th 2018
Thierry Moreau
Deep Learning Explosion
Deep Learning Revolutionizing Computer Vision

Source: NVIDIA blogpost, June 2016
Compute Requirements is Steadily Growing

Hardware Specialization

• Idea: tailor your chip architecture to the characteristics of a stable workload

Google Cloud TPU: 180 Tflops  
NVIDIA Volta: 100 Tflops  
Apple Bionic A11: 0.6 Tflops
Hardware Specialization

• Idea: tailor your chip architecture to the characteristics of a **stable** workload

- Google Cloud TPU: 180 Tflops
- NVIDIA Volta: 100 Tflops
- Apple Bionic A11: 0.6 Tflops
# Evolution of Deep Learning

## Matrix Multiplication: $\text{fp32} \ (A, B) \times (B, C)$

<table>
<thead>
<tr>
<th>Optimization</th>
<th>New problem specification</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>quantization</td>
<td>$\text{int}4 \ (A, B) \times \text{bool} \ (B, C)$</td>
<td>Binary Connect, NIPS 15</td>
</tr>
<tr>
<td>knowledge distillation</td>
<td>$\text{fp32} \ (a, b) \times (b, c)$</td>
<td>Fitnets, ICLR 2015</td>
</tr>
<tr>
<td>compression, pruning, tensor</td>
<td>$\text{sparse int}16 \ (A, B) \times (B, C)$</td>
<td>Deep Compression, ICLR 2016</td>
</tr>
<tr>
<td>decomposition</td>
<td>$\text{fp32} \ (A,r) \times (r,B) \times (B,C)$</td>
<td>Compression of deep convolutional neural networks, ICLR 2016</td>
</tr>
</tbody>
</table>

## 2D Convolution: $\text{fp32} \ (H, W, Ci) \otimes (K, K, Co, Ci)$

<table>
<thead>
<tr>
<th>winograd</th>
<th>$\text{fp32} \ \text{FFT}^{-1}(\text{FFT}((H, W, Ci)) \cdot \text{FFT}((K, K, Co, Ci))$</th>
<th>Fast Convolutional Nets with fbfft, arXiv:1412.7580 2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>depth wise convolution</td>
<td>$\text{fp32} \ (H, W, Ci) \otimes (K, K, 1, Ci) \otimes (1, 1, Co, Ci)$</td>
<td>MobileNets, arXiv:1704.04861 2017</td>
</tr>
</tbody>
</table>
Specialization Challenge

Tape-out costs for ASICs is exorbitant
10x cost gap between 16nm and 65nm

Risky bet to design hardware accelerators for ever-changing applications
Flexibility vs. Efficiency Tradeoffs

Source: Bob Broderson, Berkeley Wireless group
Discussion Break

- Does deep learning constitute a stable workload to justify ASIC-based hardware acceleration?
TPU: Google’s Entry in the Deep Learning Acceleration Race

Highlights:

• Custom ASIC deployed in datacenters since 2015

• 65k 8-bit matrix multiply that offers peak throughput of 92 TOPS

• Targets mainstream NN applications (MLPs, CNNs, and LSTMs)

• Shows 30-80x improved TOPS/Watt over K80

Jouppi et al., In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017
What make TPUs Efficient?

• Integer inference (saves 6-30x energy over 16bit FP)
• Large amount of MACs (25x over K80)
• Large amount of on-chip memory (3.5x over K80)
TPU Block Diagram Overview
Systolic Data Flow
Hardware-Software Interface

- CISC-like instruction set
  - Read_Host_Memory
  - Read_Weights
  - MatrixMultiply/Convolve
  - Activate
  - Write_Host_Memory
TPU Floor Plan

Unified Buffer for Local Activations
(96Kx256x8b = 24 MiB)
29% of chip

Matrix Multiply Unit
(256x256x8b = 64K MAC)
24%

Host Interf. 2%
Accumulators
(4Kx256x32b = 4 MiB) 6%

Control 2%
Activation Pipeline 6%

PCle Interface 3%
Misc. I/O 1%

3% DRAM port ddr3
3% DRAM port ddr3
Roofline Model
TPU Roofline

- 1350 Operations per byte of weight memory fetched
Arithmetic Intensity in Convolutional Workloads

**Convolutional Reuse**
CONV layers only (sliding window)
- Reuse: Activations, Filter weights

**Fmap Reuse**
CONV and FC layers
- Reuse: Activations

**Filter Reuse**
CONV and FC layers (batch size > 1)
- Reuse: Filter weights

*Fig. 23. Data reuse opportunities in DNNs [82].*
What does the roofline tell us about ways to improve the TPU?

- What benchmarks would benefit from improvements on clock frequency?
- What benchmarks would benefit from higher memory bandwidth?
# NVIDIA’s Rebuttal to the TPU

<table>
<thead>
<tr>
<th></th>
<th>K80 2012</th>
<th>TPU 2015</th>
<th>P40 2016</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inferences/Sec &lt;10ms latency</strong></td>
<td>$\frac{1}{13}X$</td>
<td>1X</td>
<td>2X</td>
</tr>
<tr>
<td><strong>Training TOPS</strong></td>
<td>6 FP32</td>
<td>NA</td>
<td>12 FP32</td>
</tr>
<tr>
<td><strong>Inference TOPS</strong></td>
<td>6 FP32</td>
<td>90 INT8</td>
<td>48 INT8</td>
</tr>
<tr>
<td><strong>On-chip Memory</strong></td>
<td>16 MB</td>
<td>24 MB</td>
<td>11 MB</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>300W</td>
<td>75W</td>
<td>250W</td>
</tr>
<tr>
<td><strong>Bandwidth</strong></td>
<td>320 GB/S</td>
<td>34 GB/S</td>
<td>350 GB/S</td>
</tr>
</tbody>
</table>

Discussion Break

• What makes a specialized accelerator different from a CPU or GPU?
Deep Learning Accelerator Characteristics

Memory subsystem:
- **CPU**: L3 implicitly managed
- **GPU**: L2 mixed
- **TPU**: Unified Buffer explicitly managed

Compute primitives:
- **CPU**: Scalar
- **GPU**: Vector
- **TPU**: Tensor

Data type:
- **CPU**: fp32
- **GPU**: fp16
- **TPU**: int8
HW/SW Co-Design - #1 Tensorization

Matrix-Vector computation \((4x8x1)\)

- \([4x8]\) → \([8x1]\) → \([4x1]\)

Matrix-Matrix computation \((4x4x2)\)

- \([4x4]\) → \([4x1] \times [4x1]\)
HW/SW Co-Design - #2 Memory Architecting

Convolution-Optimized, no batching
- Accumulator Register File
- Wgt FIFO
- Activation Buffer

Large activation buffer for spatial reuse
Accumulator-local scheduling
Weight FIFO for single-use weights

GEMM-Optimized, batching
- Weight Buffer
- Accumulator Register File
- Activation FIFO

Activation FIFO for single-use activations
Large accumulator storage for GEMM blocking
Weight buffer for batched execution
Reducing type width can result in a quadratic increase of compute resources, and linear increase of storage/bandwidth

*But it also affects classification accuracy!*
VTA: Versatile Tensor Accelerator

• **VTA**: a versatile and extendable deep learning accelerator for software codesign research and the development of next architectures
Addressing the Specialization Challenge

- Targets FPGAs on low-cost edge devices (PYNQ), and high-end datacenter (in progress), allowing for fast prototyping and deployment
- Leverages HLS-C, for code compactness and easy maintainability (<1000 LoC for IP)
- Built for customization, and modularity (extensible pipeline)
- Community driven (open-sourcing in progress)
VTA Features

- Customizable tensor core, memory subsystem and data types based on bandwidth, storage and accuracy needs
- Flexible CISC/RISC ISA for expressive and compact code
- Access-execute decoupling for memory latency hiding
Customization

Tensor Intrinsic

8 x 8 vs. 1 x 16

<16 x i8> vs. <32 x i4>

Memory Subsystem

Operator Support

{ADD, MUL, SHL, MAX} vs. {ADD, SHL, MAX}
CISC/RISC ISA

• Goal: Provide the right tradeoff between expressiveness and code compactness
  • Use CISC-ness to describe high-level operation (LD, ST, GEMM, ALU)
  • Use RISC-ness to describe low-level memory access patterns
• Micro-op kernels are stored in a local micro op cache to implement different operators

micro-op cache

- 3x3 conv stride = 1
- 7x7 conv stride = 3
- Fully connected
- 2x2 max pool stride = 2
- Batch norm
- Relu

ALU Micro kernel
GEMM Micro kernel
Latency Hiding

- How do keep computation resources (GEMM) busy:
  - Without latency hiding, we are wasting compute/memory resources
  - By exploiting pipeline parallelism, we can hide memory latency
Latency Hiding

• Pipeline parallelism requirements:
  • Concurrent tasks need to access non-overlapping regions of memory
Latency Hiding

- Pipeline parallelism requirements:
  - Concurrent tasks need to access non-overlapping regions of memory
Latency Hiding

• Pipeline parallelism requirements:
  • Concurrent tasks need to access non-overlapping regions of memory
  • Data dependences need to be explicit!
Latency Hiding

• We want to enforce read-after-write (RAW) dependences

Without RAW dependence tracking, operations execute as soon as the stage is idle.
Latency Hiding

• We want to enforce read-after-write (RAW) dependences
Latency Hiding

• We want to enforce read-after-write (RAW) dependences

• AND we want to enforce write-after-read (WAR) dependences

Legend:
RAW dependence:

We are overwriting data in partition 0 before GEMM has finished consuming data from the first LD!
Latency Hiding

• We want to enforce read-after-write (RAW) dependences

• AND we want to enforce write-after-read (WAR) dependences

Legend:
- RAW dependence:
- WAR dependence:
Takeaway: work partitioning and explicit dependence graph execution (EDGE) unlocks pipeline parallelism to hide the latency of memory accesses
VTA Design Overview

MEMORY LOAD UNIT
- LOAD Q
- LOAD → EXE Q
- EXE → LOAD Q
- LOAD BUFFER
- MICRO-OP SRAM
- ACTIVATION SRAM
- KERNEL SRAM

COMPUTE
- COMPUTE Q
- COMPUTE
- REGISTER FILE
- V_ALU
- GEMM
- EXE → LOAD Q
- STORE → EXE Q
- STORE BUFFER

MEMORY STORE UNIT
- STORE Q
- STORE → EXE Q
Instruction fetch stage fetches high-level instructions from DRAM, decodes them, and pushes commands to the relevant queue (LD, EX, ST)
The load stage executes load commands to populate activation & kernel memories, the micro-op cache, and a load buffer for loading in values for the register file.
VTA Design

Compute stage executes compute commands to perform vector ALU operations or GEMM operations to update the register file according to micro-coded kernels.
Memory store stage executes store commands to store flushed register file values back to DRAM from the store buffer.
VTA Design

Stages communicate via dependence token queues to indicate that they may proceed to execute the command they’re about to work on.
Memories that connect pipeline stages follow a strict single producer, single consumer rule (fan-in=1, fan-out=1). This enables data flow execution, and makes this design modular.
VTA Microprogramming

// Pseudo-code for convolution program for the VLA accelerator
// Virtual Thread 0
0x00: LOAD(PARAM[0-71]) // LDW100
0x01: LOAD(Activ[0-24]) // LDW100
0x02: LOAD(LDBUF[0-31]) // LDW100
0x03: PUSH(LD->EX) // EXW100
0x04: POP(ST->EX) // EXW100
0x05: POP(ST->EX) // EXW100
0x06: EXE(Activ[0-24],PARAM[0-71],LDBUF[0-31],STBUF[0-7]) // EXW100
0x07: PUSH(EX->ST) // EXW100
0x08: POP(EX->ST) // EXW100
0x09: STORE(STBUF[0-7]) // STW100
0x0A: PUSH(ST->EX) // STW100
// Virtual Thread 1
0x0B: LOAD(Activ[25-50]) // LDW101
0x0C: LOAD(LDBUF[32-63]) // LDW101
0x0D: PUSH(LD->EX) // EXW101
0x0E: POP(ST->EX) // EXW101
0x0F: EXE(Activ[25-50],PARAM[0-71],LDBUF[32-63],STBUF[32-39]) // EXW101
0x10: PUSH(EX->ST) // EXW101
0x11: PUSH(EX->ST) // EXW101
0x12: PUSH(EX->ST) // EXW101
0x13: STORE(STBUF[32-39]) // STW101
0x14: PUSH(ST->EX) // STW101
// Virtual Thread 2
0x15: POP(EX->ST) // EXW102
0x16: LOAD(PARAM[0-71]) // LDW102
0x17: LOAD(Activ[0-24]) // LDW102
0x18: LOAD(LDBUF[0-31]) // LDW102
0x19: PUSH(LD->EX) // EXW102
0x1A: POP(LD->EX) // EXW102
0x1B: POP(ST->EX) // EXW102
0x1C: EXE(Activ[0-24],PARAM[0-71],LDBUF[0-31],STBUF[0-7]) // EXW102
0x1D: PUSH(EX->ST) // EXW102
0x1E: POP(EX->ST) // EXW102
0x1F: STORE(STBUF[0-7]) // STW102
// Virtual Thread 3
0x20: POP(EX->ST) // EXW103
0x21: LOAD(Activ[25-50]) // LDW103
0x22: LOAD(LDBUF[32-63]) // LDW103
0x23: PUSH(LD->EX) // EXW103
0x24: POP(LD->EX) // EXW103
0x25: POP(ST->EX) // EXW103
0x26: EXE(Activ[25-50],PARAM[0-71],LDBUF[32-63],STBUF[32-39]) // EXW103
0x27: PUSH(EX->ST) // EXW103
0x28: POP(EX->ST) // EXW103
0x29: STORE(STBUF[32-39]) // STW103

// Convolution access pattern dictated by micro-coded program.
// Each register index is derived as a 2-D affine function.
// e.g. idx_x = a_x*b_y+c_z, where c_z is specified by
// micro op 0 fields.
for y in [0...i)
for x in [0...j)
rf[idx_x] += GEVM(act[idx_y], par[idx_z])
rf[idx_x] += GEVM(act[idx_y], par[idx_z])
rf[idx_x] += GEVM(act[idx_y], par[idx_z])

// Max-pool, batch normalization and activation function
// access pattern dictated by micro-coded program.
// Each register index is derived as a 2D affine function.
// e.g. idx_x = a_x*b_y+c_z, where c_z is specified by
// micro op 0 fields.
for y in [0...i)
for x in [0...j)
// max pooling
rf[idx_x] = MAX(rf[idx_x], rf[idx_x])
rf[idx_x] = MAX(rf[idx_x], rf[idx_x])

// batch norm
rf[idx_x] = HUL(rf[idx_x], rf[idx_x])
rf[idx_x] = ADD(rf[idx_x], rf[idx_x])
rf[idx_x] = ADD(rf[idx_x], rf[idx_x])
rf[idx_x] = ADD(rf[idx_x], rf[idx_x])

// activation
rf[idx_x] = RELU(rf[idx_x], rf[idx_x])
rf[idx_x] = RELU(rf[idx_x], rf[idx_x])

(a) Blocked convolution program with multiple thread contexts

(b) Convolution micro-coded program

(c) Max pool, batch norm and activation micro-coded program
VTA Microprogramming

(a) Blocked convolution program with multiple thread contexts

```
// Virtual Thread 0
0x00: LOAD(PARAM[0-71])     // LDH100
0x01: LOAD(activ[0-24])     // L12H101
0x02: LOAD(ldbbuf[0-31])    // L12H102
0x03: PUSH(LD->EX)          // STH103
0x04: POP(LO->EX)           // STH104
0x05: EXE(activ[0-24],param[0-71]) // EXH105
0x06: PUSH(EX->LD)          // LDH106
0x07: PUSH(EX->ST)          // STH107
0x08: POP(ST->EX)           // EXH108
0x09: STOR(ldbbuf[0-31])    // STH109
// Virtual Thread 1
0x0A: LOAD(PARAM[0-71])     // LDH110
0x0B: LOAD(activ[0-24])     // L12H111
0x0C: LOAD(ldbbuf[32-63])   // L12H112
0x0D: PUSH(LD->EX)          // STH113
0x0E: POP(LO->EX)           // STH114
0x0F: EXE(activ[0-24],param[0-71]) // EXH115
0x10: PUSH(EX->LD)          // LDH116
0x11: PUSH(EX->ST)          // STH117
0x12: POP(ST->EX)           // EXH118
0x13: STOR(ldbbuf[32-63])   // STH119
0x14: PUSH(ldbbuf[32-63])   // STH120
0x15: POP()                 // EXH121
0x16: LOAD(PARAM[0-71])     // LDH122
0x17: LOAD(activ[0-24])     // L12H123
0x18: LOAD(ldbbuf[0-31])    // L12H124
0x19: PUSH(LD->EX)          // STH125
0x1A: POP(LO->EX)           // STH126
0x1B: EXE(activ[0-24],param[0-71]) // EXH127
0x1C: STOR(ldbbuf[0-31])    // STH128
0x1D: PUSH(ldbbuf[32-63])   // STH129
0x1E: POP()                 // EXH130
0x1F: LOAD(PARAM[0-71])     // LDH131
0x20: LOAD(activ[0-24])     // L12H132
0x21: LOAD(ldbbuf[32-63])   // L12H133
0x22: PUSH(LD->EX)          // STH134
0x23: PUSH(EX->ST)          // STH135
0x24: POP(ST->EX)           // EXH136
0x25: EXE(activ[0-24],param[0-71],ldbbuf[32-63],ldbbuf[32-63]) // EXH137
0x26: STOR(ldbbuf[32-63])   // STH138
0x27: PUSH(EX->ST)          // STH139
0x28: POP(EX->ST)           // STH140
0x29: STOR(ldbbuf[32-63])   // STH141
```
```
(b) Convolution micro-coded program

```
// Pseudo-code for convolution program for the VIA accelerator
// Virtual Thread 0
// LDH100: Load parameter
// L12H101: Load activate
// L12H102: Load LDBUF
// STH103: Push EX to ST
// STH104: Pop LO to EX
// EXH105: Execute (activate, parameter, LDBUF, STBUF)
// STH106: Push EX to LD
// STH107: Push EX to ST
// EXH108: Pop ST to EX
// STH109: Store LDBUF
// LDH110: Load parameter
// L12H111: Load activate
// L12H112: Load LDBUF
// STH113: Push EX to LD
// STH114: Push EX to ST
// EXH115: Execute (activate, parameter)
// STH116: Push EX to LD
// STH117: Push EX to ST
// EXH118: Pop ST to EX
// STH119: Store LDBUF
// STH120: Push LDBUF
// EXH121: Exit
// LDH122: Load parameter
// L12H123: Load activate
// L12H124: Load LDBUF
// STH125: Push EX to ST
// STH126: Pop LO to EX
// EXH127: Execute (activate, parameter)
// STH128: Push EX to LD
// STH129: Push EX to ST
// EXH128: Pop ST to EX
// STH130: Push LDBUF
// EXH129: Exit
// LDH131: Load parameter
// L12H132: Load activate
// L12H133: Load LDBUF
// STH134: Push EX to ST
// STH135: Push EX to ST
// EXH136: Pop ST to EX
// STH137: Execute (activate, parameter, LDBUF, STBUF)
// STH138: Store LDBUF
// STH139: Push LDBUF
// EXH140: Exit
```
```
(c) Max pool, batch norm and activation micro-coded program
```
```
// Microprogramming is a pain!
```

Microprogramming is a pain!
Building a deep learning accelerator compiler stack in TVM

- **TVM**: An end-to-end compiler & optimization framework for diverse hardware
Addressing the Programmability Challenge

High-Level Deep Learning Framework

NNVM Graph

TVM Compiler

RPC Layer

Runtime & JIT Compiler

FPGA/SoC Drivers

VTA FPGA Design

TVM DSL allows for separation of schedule and algorithm
Designing Scheduling Primitives for VTA

- What does an optimization stack for deep learning accelerators look like?

<table>
<thead>
<tr>
<th>Hardware Feature</th>
<th>Scheduling Primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense Linear Algebra Core</td>
<td>Tensorization</td>
</tr>
<tr>
<td>Explicitly Managed Memory Subsystem</td>
<td>Scoped cache reads/writes</td>
</tr>
<tr>
<td>Low-Level Code Generation</td>
<td>JIT compilation in runtime</td>
</tr>
<tr>
<td>Access-Execute Decoupling for Latency Hiding</td>
<td>Virtual threading</td>
</tr>
</tbody>
</table>
Virtual Threading

• How do we take advantage of pipeline parallelism with virtual threading?

Hardware-centric view: pipeline execution

Load Stage
GEMM Stage
Store Stage

Execution Phase 1

Execution Phase 2
Virtual Threading

• How do we take advantage of pipeline parallelism with virtual threading?

Software-centric view: threaded execution

Execution Phase 1

Execution Phase 2
Virtual Threading

- How do we take advantage of pipeline parallelism with virtual threading?

**Software-centric view: threaded execution**

**vthread 0**

- Execution Phase 1
  - LD
  - EX
  - LD
  - EX
  - ST

**vthread 1**

- Execution Phase 2
  - LD
  - EX
  - LD
  - EX
  - ST
Virtual Threading

Software-centric view: threaded execution

vthread 0

vthread 1

Execution Phase 1

Execution Phase 2

• Benefit #1: dependences are automatically inserted between successive stages within each virtual thread

Legend:

RAW dependence:

WAR dependence:
Virtual Threading

Software-centric view: threaded execution

• Benefit #1: dependences are automatically inserted between successive stages within each virtual thread

• Benefit #2: barriers insert dependences between execution stages to guarantee sequential consistency

Legend:
- RAW dependence: 
- WAR dependence:
Virtual Threading

Final step: virtual thread lowering into a single instruction stream

Push and pop commands dictate how to interact with the hardware dependence queues

Legend
- push dependence to consumer stage
- push dependence to producer stage
- pop dependence from producer stage
- pop dependence from consumer stage
Programming for VTA in TVM

1. How do we partition work and explicitly manage on-chip memories?

2. How do we take advantage of tensorization?

3. How do we take advantage of virtual threading?
TVM Scheduling Primitives

1. How do we partition work and explicitly manage on-chip memories?

    // Tile
    yo, xo, yi, xi = s[OUT].tile(y, x, 4, 4)

    // Cache read
    INP_L = s.cache_read(INP, vta.act, [OUT])
    s[INP_L].compute_at(s[OUT], xo)

2. How do we take advantage of tensorization?

    // Tensorize
    s[OUT_L].tensorize(ni)

3. How do we take advantage of virtual threading?

    // Virtual Threading
    tx, co = s[OUT_L].split(co, factor=2)
    s[OUT_L].bind(tx, thread_axis("cthread"))
Full Stack Evaluation (TVM)

- Full evaluation on PYNQ FPGA board

TVM can offload most convolution operations to the FPGA (40x speedup on off-loadable layers). Utilization improves from at best 52% to 74%.

TVM can exploit latency hiding mechanisms to improve throughput.
Resources

• Build your own simple VTA accelerator: https://gitlab.cs.washington.edu/cse599s/lab1

• TVM Tutorial for VTA to be released

• Looking for alpha users of the full VTA open source design

moreau@uw.edu
Thank you

moreau@uw.edu