Software Co-design for the First Wafer-Scale Processor (and Beyond)

Cerebras Systems
Cerebras Wafer Scale Engine (WSE)

The Most Powerful Processor for AI

400,000 AI-optimized cores
46,225 mm² silicon
1.2 trillion transistors
18 Gigabytes of On-chip Memory
9 PByte/s memory bandwidth
100 Pbit/s fabric bandwidth
TSMC 16nm process
Cerebras CS-1: Cluster-Scale DL Performance in a Single System

Powered by the WSE

Programming accessibility of a single node, using TensorFlow, PyTorch, and other frameworks

Deploys into standard datacenter infrastructure

Multiple units delivered, installed, and in-use today across multiple verticals

Built from the ground up for AI acceleration
Architecture Designed for Deep Learning

Each component optimized for AI compute

Compute
- Fully-programmable core, ML-optimized extensions
- Dataflow architecture for sparse, dynamic workloads

Memory
- Distributed, high performance, on-chip memory

Communication
- High bandwidth, low latency fabric
- Cluster-scale networking on chip
- Fully-configurable to user-specified topology

Together, orders of magnitude performance and efficiency gain

Linear cluster-scale performance on a single chip
SW Co-designed for Scale from the Beginning

This time, we’ll cover how the codesigned software leverages the power and flexibility of the WSE for deep learning acceleration.

And what’s next.
Users **program the WSE using standard ML frameworks**, e.g. TensorFlow, PyTorch

Cerebras Graph Compiler automatically compiles the DNN graph
- Extracts from Framework, converts to Cerebras IR, performs matching to Cerebras kernels
- Place & Route allocates compute and memory, configures on-chip network

Enables **straightforward programming, flexible execution, high performance**
Matching to Kernel Library

Graph matching from FW ops to **Kernels:**

- Primitives to be sized and placed by rest of CGC
- Expressed as nested for-loops for generality

**2 Kernel Types:**

1. Auto-generated
   - General and supports various operations
   - Polyhedral techniques
   - Unrolling loop dimensions across fabric

2. Hand-optimized
   - High-performance common kernels
   - Hand-tuned ucode and fabric programming

MATMUL Op

```
for i = 0...783:
    for j = 0...255:
        out0[j] += lhs[i]*rhs[i][j]
```

MATMUL Kernel
Choosing the Optimal Mapping Strategy

• Choose mapping strategy for each kernel
  • Model parallel – size and allocation of each kernel
  • Data parallel – replication factor

• Strategy determines
  • Allocation of compute cores to kernel
  • Amount of memory to kernel
  • Optimal communication pattern

Neural Network Kernels
Automatically Exploring the Optimization Search Space

Neural Network Kernels

One possible allocation of the compute, memory, and fabric to each kernel

A different allocation of compute, memory, and fabric to each kernel
Automatically Exploring the Optimization Search Space

Option 3x3: 4X slower ¼ area
Option 3x6: 2X slower ½ area
Option 6x6: 36 cores
Option 6x12: 2X faster 2X area
Option 12x12: 4X faster 4X area

Fine-tuned Area vs Performance trade-off per kernel
Mapping Compute Kernels on the CS-1
Co-Designed for Training Flexibility and Performance

Compiler stack and hardware architecture co-designed

Result: Flexibility and Performance
1. Model parallel and data parallel execution
2. Sparsity harvesting and inducing
3. Dynamic neural network execution
1) Flexible Parallelism

• Optimization search enables **spectrum** of parallel execution strategies on WSE

• **Single algorithm** uses both model and data parallelism in optimization

• Execution strategies **optimized** for different **network characteristics**

---

**Data Parallel**

- Device 1
- Device 2
- Sample 1
- Sample N

Running multiple **samples** at same time

**Model Parallel**

- Device 1
- Device 2
- Model Part 1
- Model Part 2

Running multiple **parts of network** at same time

---

**Layer-sequential**

- WSE
- GPU

**Data Parallel**

- (Batch Size)
Using Model Parallelism

• **Run all layers on fabric sections**
  • Layers in parallel = more performance
  • Execute network as pipeline
  • Enabled by high bandwidth interconnect

• **Small batch size**
  • No need to replicate network

• **No weight sync overhead**
  • Weights updated in place

Result: linear performance scaling with **small** batch size
Using Data Parallelism

- Run layer replicas on fabric sections
  - Replicas in parallel = more performance
  - Applies to smaller layers/networks

- Not forced to large batch size
  - Small batch size per replica
  - Single sample execution enabled by memory performance at low batch
  - Larger batch by running multi-samples

- Low weight sync overhead
  - Enabled by low latency and high bandwidth interconnect

Result: linear performance scaling with medium batch size
2) Translating Sparsity into Training Performance

Large number of zeros in neural network
- Nonlinears create activation sparsity
- Mul-Add by zero does not change the result

Kernels designed for sparsity
- Harvest natural sparsity in neural network
- Induce sparsity when not naturally occurring
Core Designed for Sparsity

Enabled by dataflow scheduling in hardware
- Fabric data triggers instruction lookup
- State machine schedules datapath cycles

Intrinsic sparsity harvesting
- Sender filters out sparse zero data
- Receiver skips unnecessary processing

Fine-grained execution datapaths
- Small cores with independent instructions
- High utilization for dynamic non-uniform work
Natural Sparsity in Transformer

• Transformer uses ReLU and Dropout non-linears
  • ReLU is 90% naturally sparse
  • Dropout is 30% naturally sparse
• **1.2x perf gain** vs. dense non-linear and no dropout
Natural Sparsity in Transformer

• Transformer uses ReLU and Dropout non-linears
  • ReLU is 90% naturally sparse
  • Dropout is 30% naturally sparse
• 1.2x perf gain vs. dense non-linear and no dropout
Inducing Sparsity

- Sparsity can be **induced** by
  - Adding sparse non-linear (e.g. ReLU)
  - Dropping relatively small values
- Inducing sparsity on 34-Layer dense FC model
- **1.7x perf gain** with ReLU
- **2.4x perf gain** with ReLU+50% sparsity

![34-Layer FC Performance vs. Induced Sparsity](chart.png)
Inducing Sparsity in BERT

- BERT has no natural sparsity
- But sparsity can be induced on most layers in both fwd and bwd pass
- Up to **1.5x perf gain** with 50% sparsity and minimal accuracy loss
- ML user has control
3) Designed to Unlock Smarter Techniques and Scale

WSE has a data flow architecture
- Flexibility to stream *token by token*
- Inherent sparsity harvesting

WSE is a MIMD architecture
- Can program each core independently
- Perform different operations on different data
Flexibility Enables Dynamic ML Methods

Fine-grained dynamic execution enables new ML techniques

1. Variable sequence length
   • Stop at end of sequence, no padding

2. Irregular/NAS models
   • High utilization for non-square matrices

3. Recursive dynamic depth
   • Run enough layers to meet objective

4. Dynamic (and long) sequence length
   • Process only relevant part of sequence

Universal Transformer with Dynamic Depth.
CS-1 HW/SW Co-design Enables Next Gen DL models

The community wants smarter **and** larger models

- **CS-1** is the most *powerful single node*
  - Automatic scaling through model and data parallelism
  - *Accessible* cluster-scale performance on a single chip
- **CS-1** is **flexible and dynamic**
  - Fine-grained sparsity harvesting and induction
  - Novel adaptive & dynamic novel ML techniques

This combination of flexibility and performance **enables the next generation of models and techniques** otherwise challenged today.
Wafer Scale Engine – Generation 2

850,000 AI-optimized cores

2.6 Trillion Transistors

TSMC 7nm Process