Cerebras AI Training Workshop

May 7- 8, 2024

Agenda

Time	Topic
Day 1: Tuesday 6 May 1:00pm-4:30pm CDT (11:00am-2:30pm PDT)	
1:00 - 1:20pm	Introduction
1:20 - 1:35pm	Hardware and systems
1:35 - 1:50pm	Software and programming
1:50 - 2:00pm	Break
2:00 - 2:30pm	How-to: Model porting, layer API, data loaders
2:30 - 2:45pm	HuggingFace to CS-2 overview
2:45 - 3:05 pm	How-to: Monitoring and profiling
3:05 - 3:15pm	Break
3:15 - 4:00pm	Hands-on session for training
4:00 - 4:30pm	Release 2.2.1 highlights
Day 2: Wednesday 7 May 1:00pm-4:30pm CDT (11:00am-2:30pm PDT)	
1:00 - 1:45pm	Efficient training with Cerebras, scaling laws, how to train LLMs
1:45 - 2:45pm	User training: hands-on LLM model
2:45 - 3:00pm	Break
3:00 - 4:00pm	HPC: CS for HPC: SDK, CSL and past examples
4:00 - 4:20pm	Roadmap presentation
4:20 – 4:30pm	Closing, final Q&A

Cerebras Systems

Building and deploying a new class of computer system
Designed for the purpose of accelerating AI and changing the future of AI work

Founded in 2016

350+ Engineers

Offices
Silicon Valley | San Diego | Toronto | Tokyo

Customers
North America | Asia | Europe | Middle East

MAYO CLINIC GSK AstraZeneca TotalEnergies Argonne NATIONAL LABORATORY

Large-scale AI+HPC has **transformative potential** for science and industry

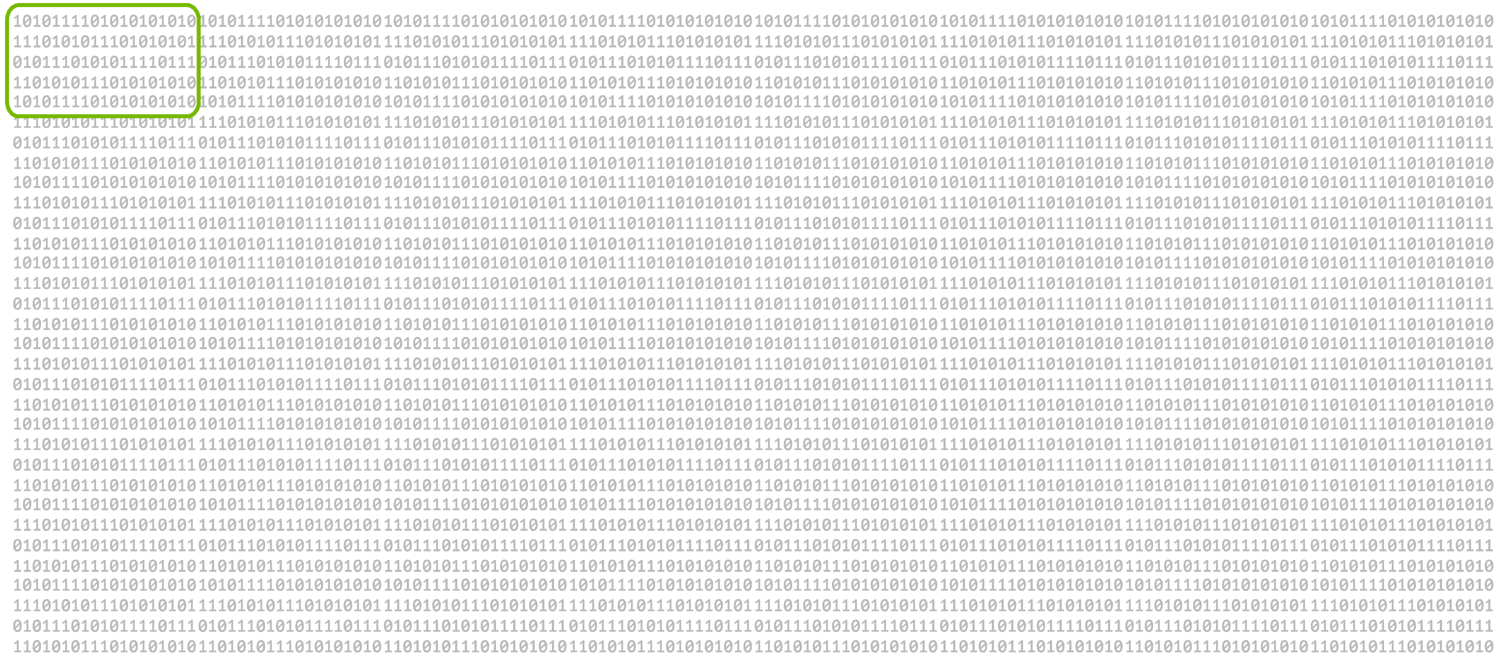
However, these compute workloads are **complex and time-intensive** to implement on clusters of legacy, general purpose processors

Performance and programming at scale are constraints on our ability to “go big”

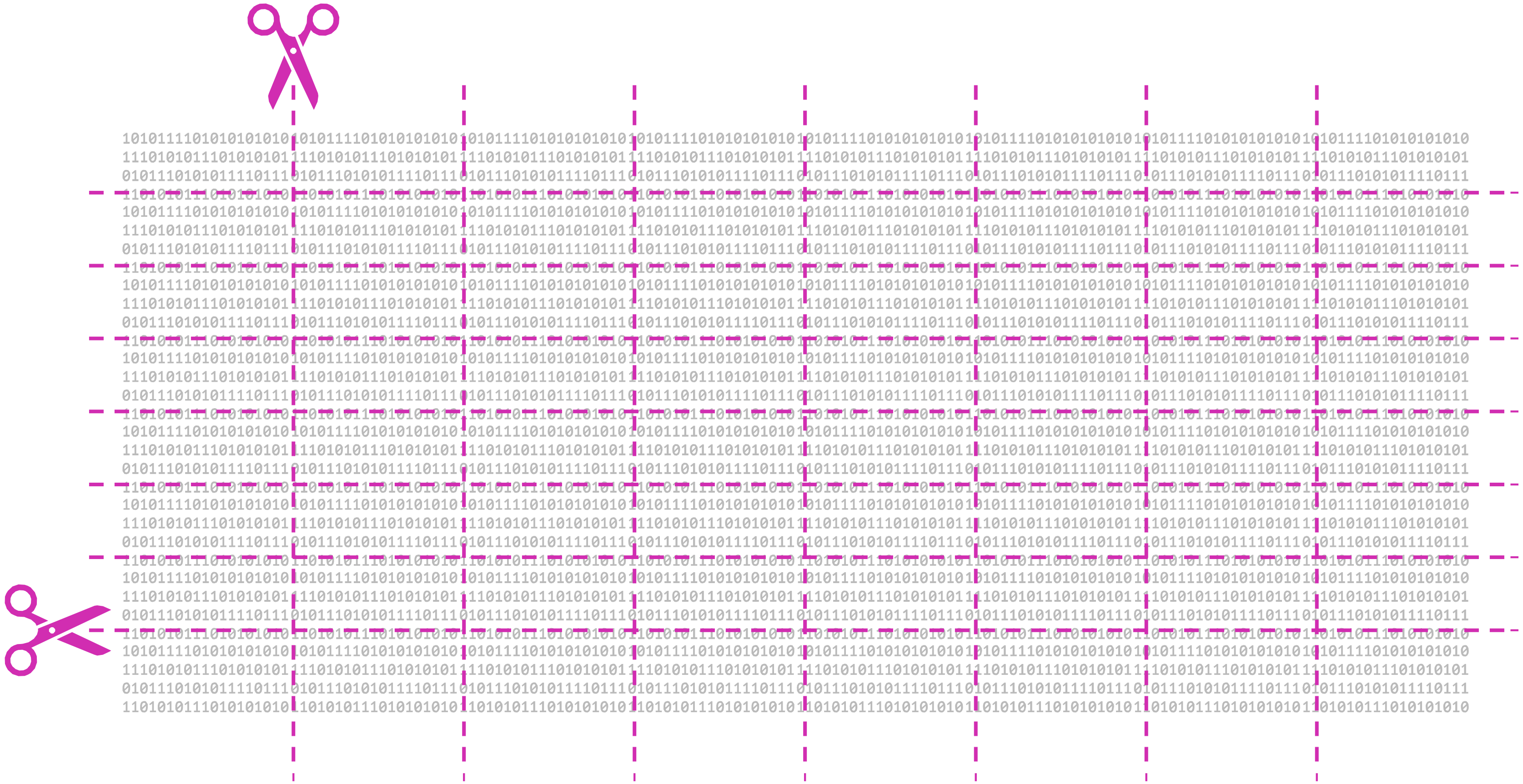
Large Models Don’t Fit on GPUs

ChatGPT (28TB)

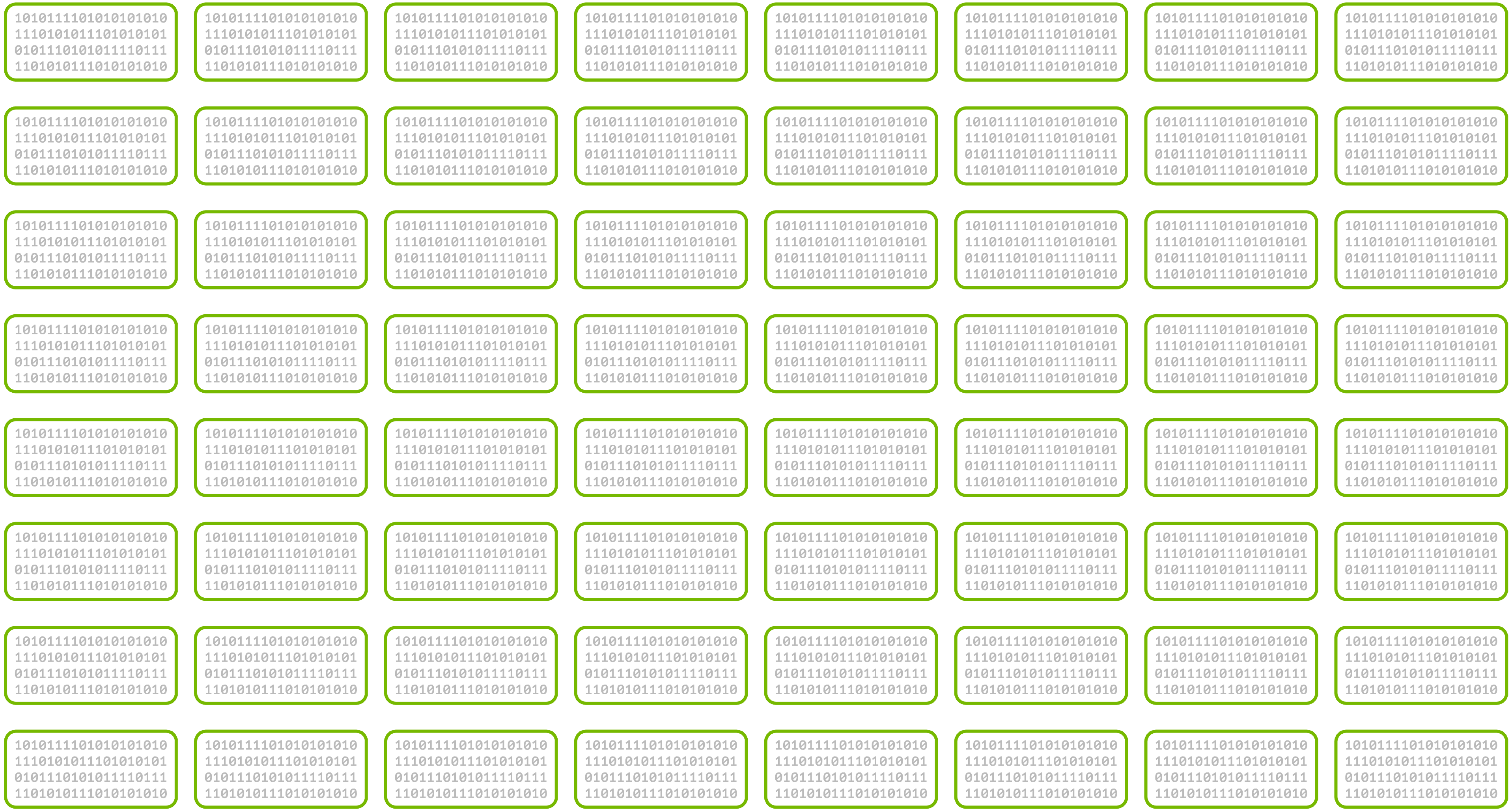
H100 (80GB)



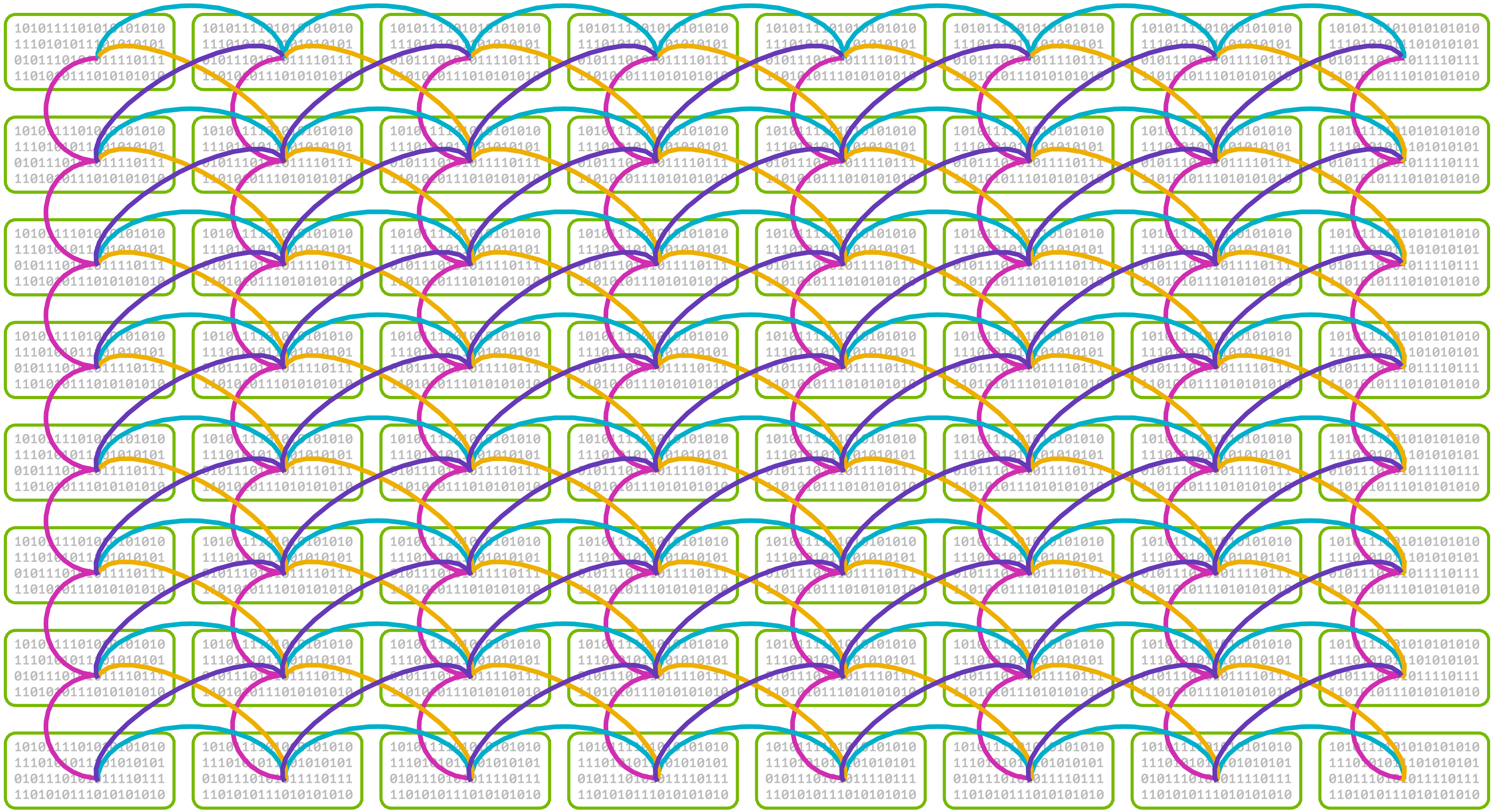
Developers must cut the model into many pieces..



And spread them on hundreds of GPUs



Then re-write the model to work across a cluster



An ML problem just turned into a parallel programming problem.

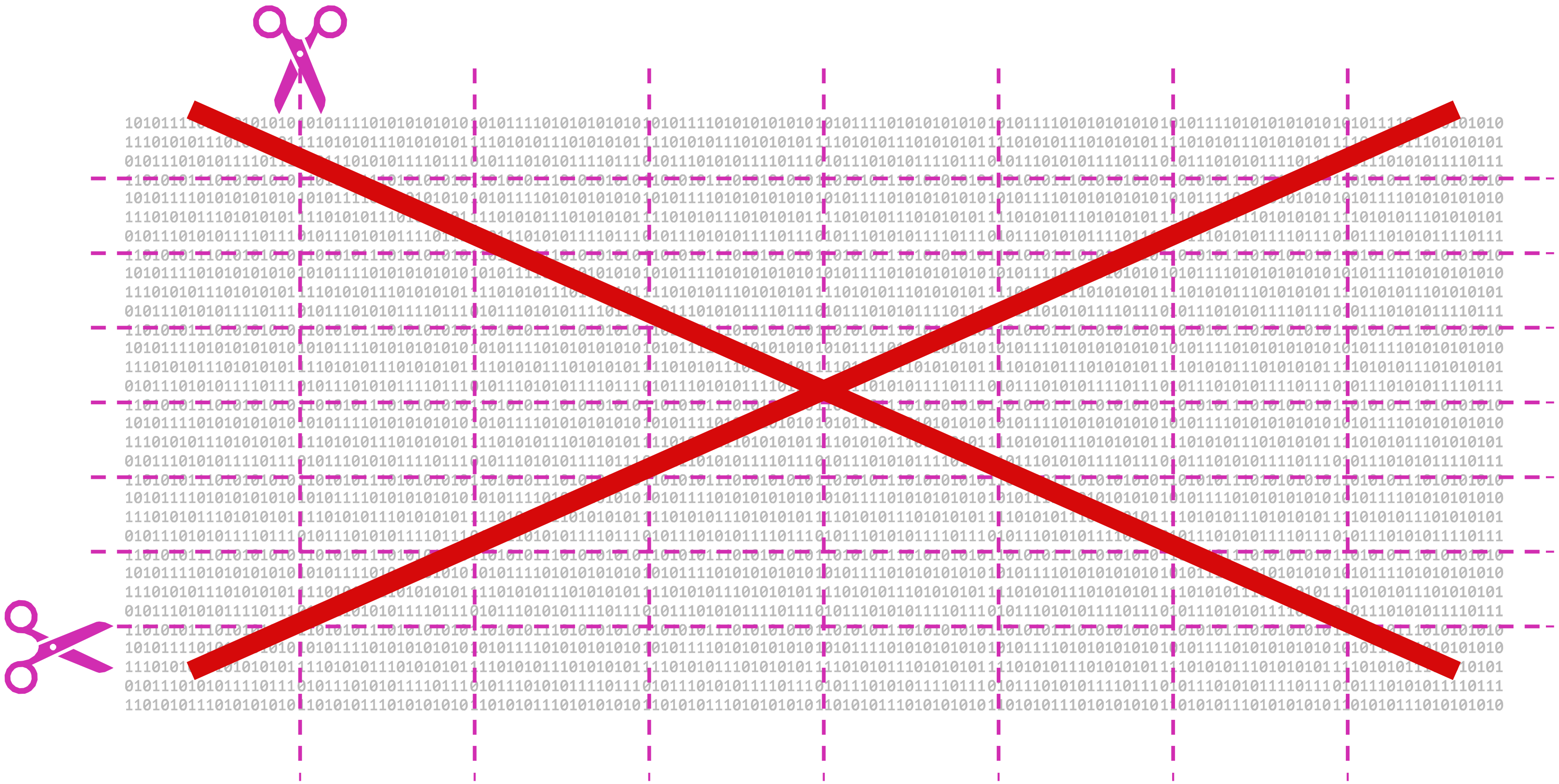
A hardware problem just became a supercomputer problem.

This causes a code explosion

nanoGPT
1B Parameters
639 lines of code

Megatron
100B Parameters
20,507 lines of code

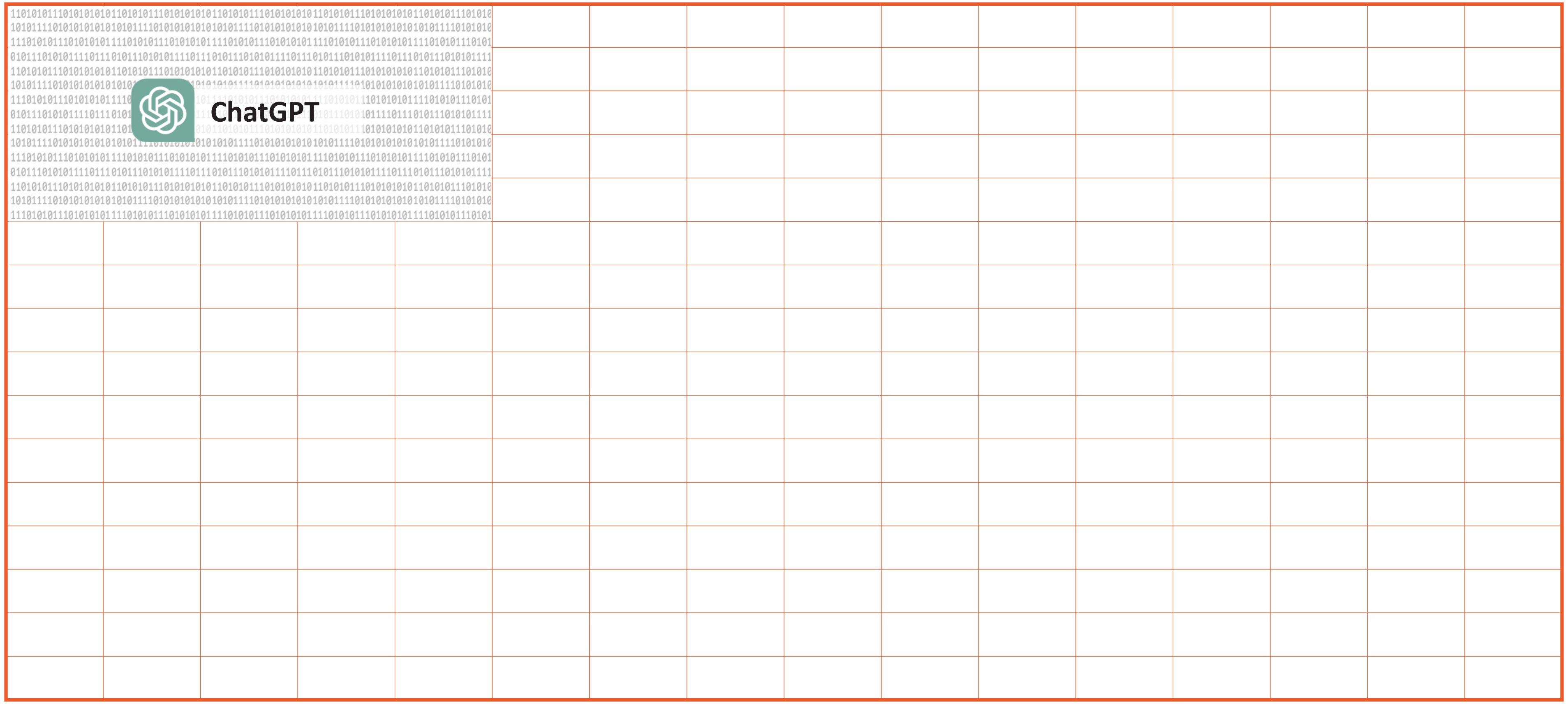
You never have to do this on Cerebras



The Cerebras Way

Build a compute & memory system that's vastly larger than the model

Cerebras Wafer Scale Cluster up to 1,200 TB



The Cerebras Way

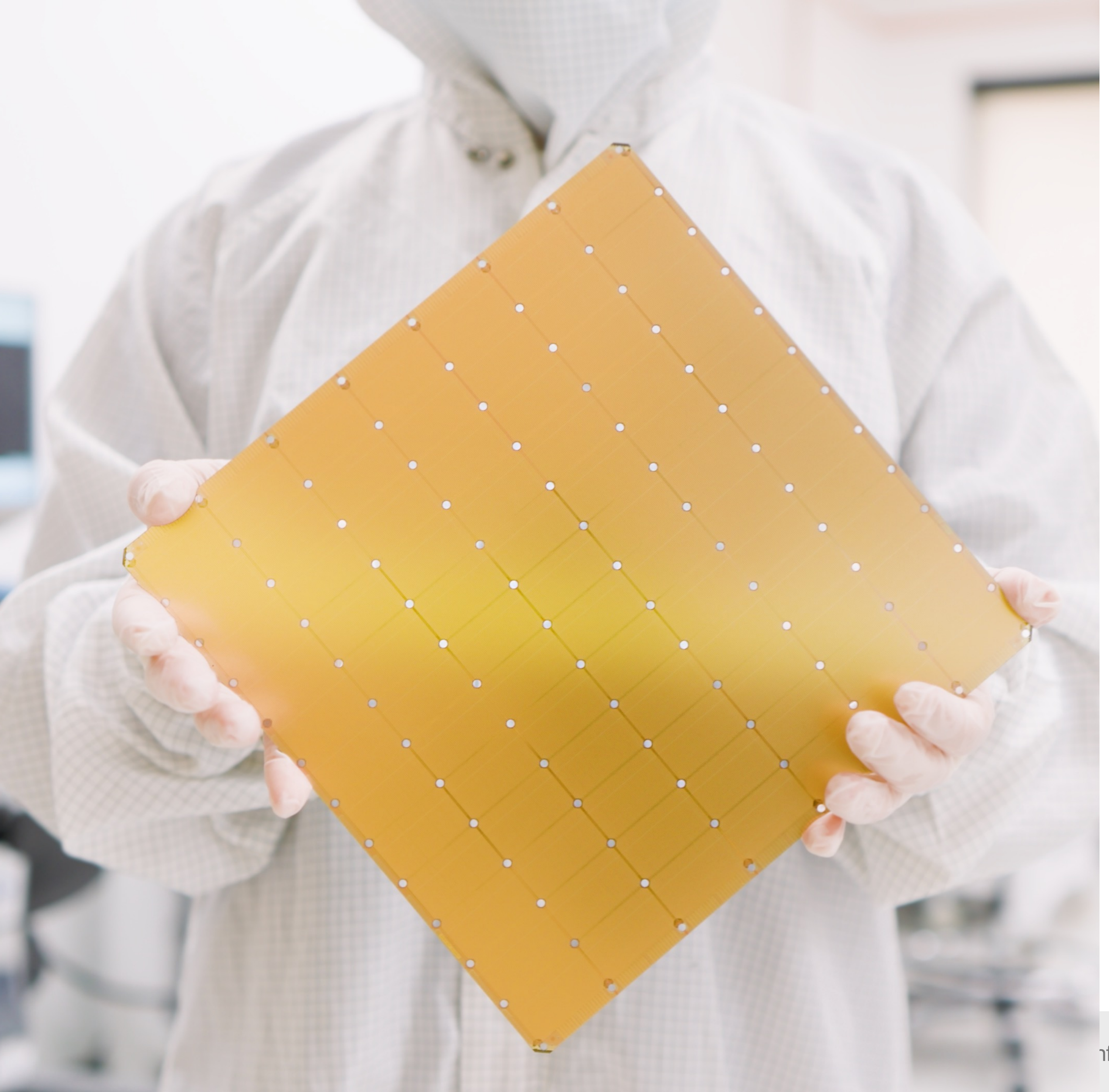
Make GenAI models easy

Build the fastest AI accelerators

Connect into easy to use and quick to deploy AI supercomputers

Train models for the open source community and enterprise customers

Provide extensive in-house ML expertise



Cerebras Wafer-Scale Engine

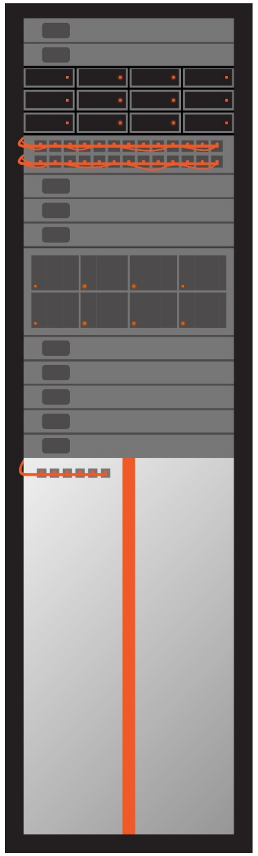
- We built the largest chip in the world;
  56x larger than a GPU;
  tailor-made for large Generative AI workloads.

- Outperforms state-of-the-art chips across all key dimensions.

- It is faster, easier to use, and requires less power and space than alternative hardware.

Cerebras CS-2, CS-3

Wafer Scale Cluster: Scalable AI Supercomputer



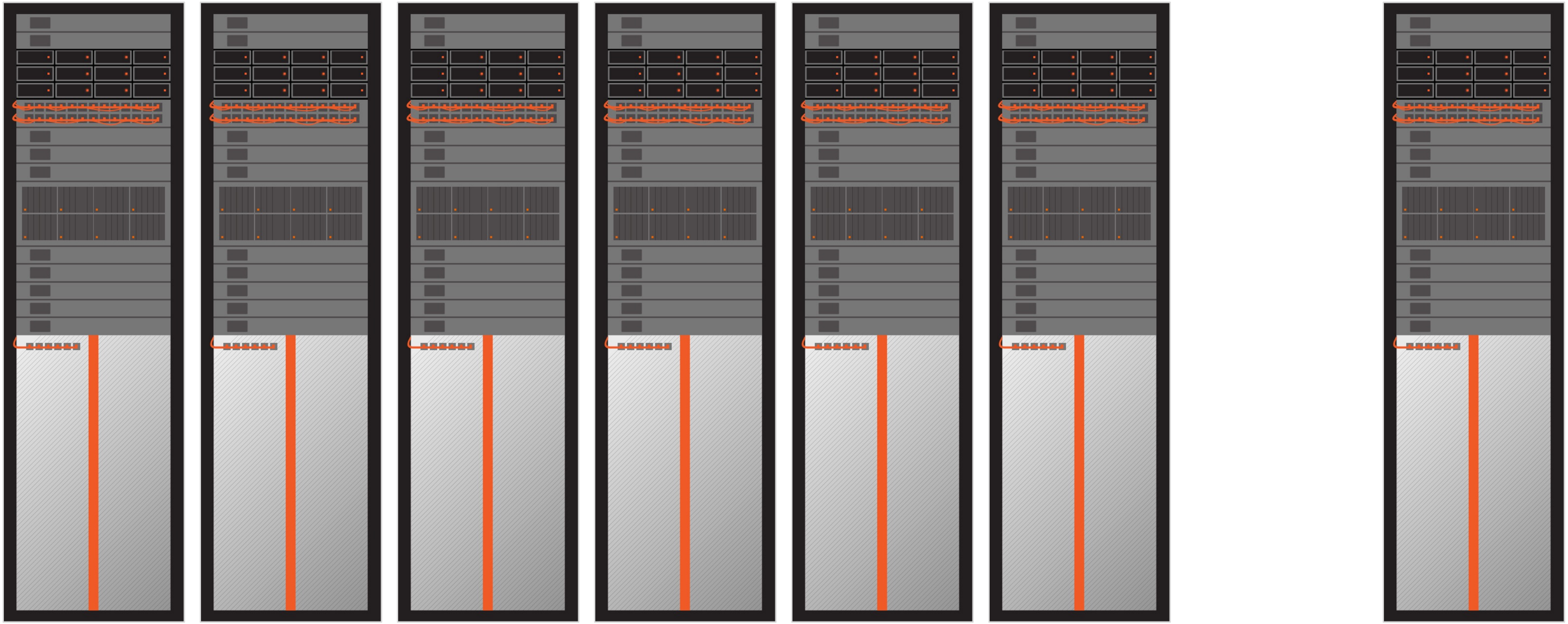
MemoryX: 128 gigabyte → 1 petabyte

SwarmX: 1 CS-2, 1 CS-3 → 192 CS-2, 2048 CS-3s

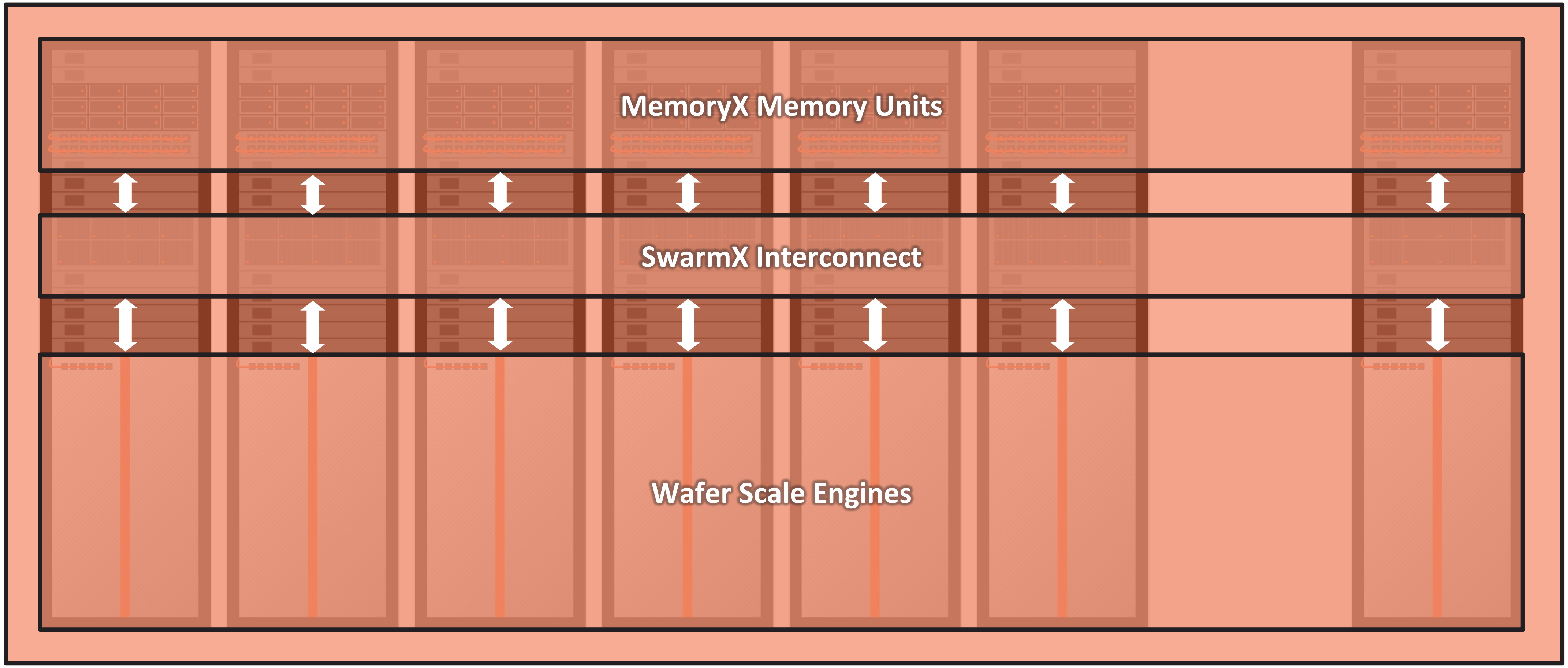
CS-3: 62 petaflops → 256 exaflops

1 billion parameters → 24 trillion parameters

Exa-scale Performance

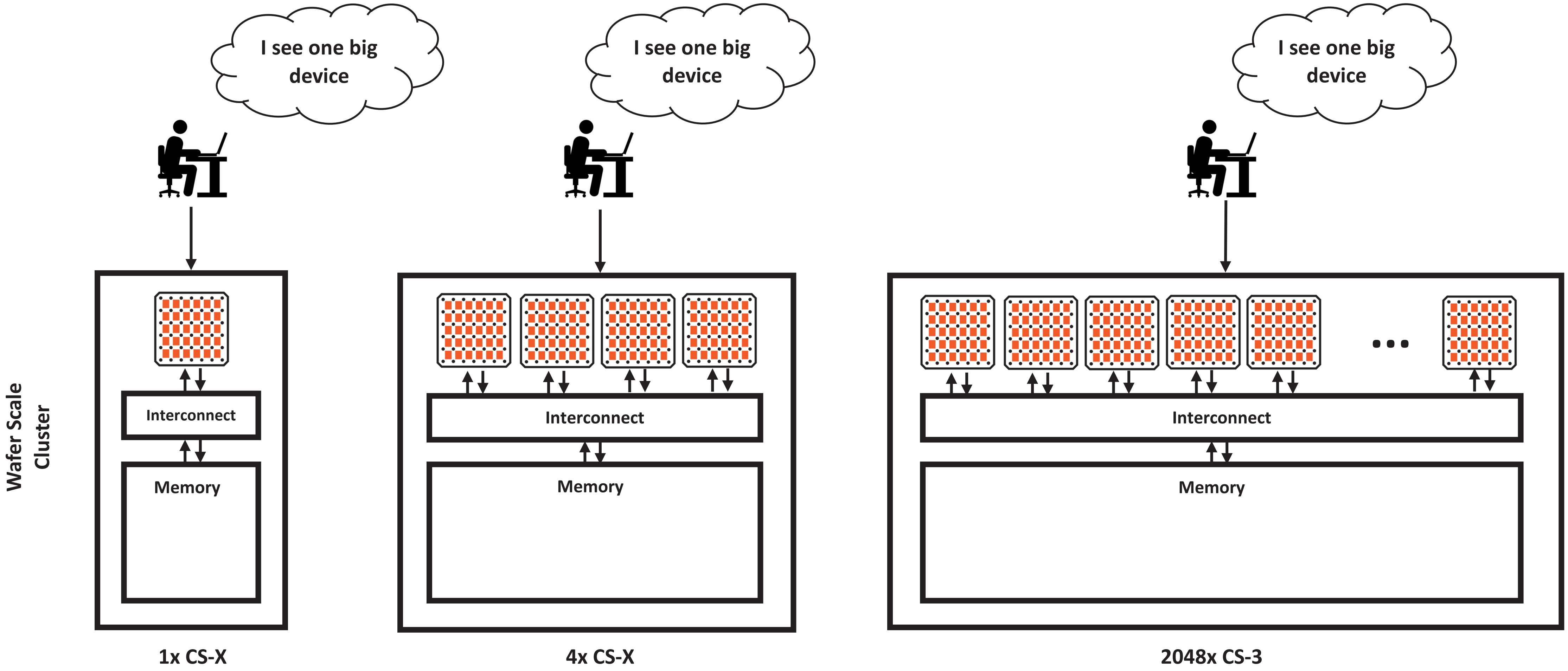


Single Device Simplicity



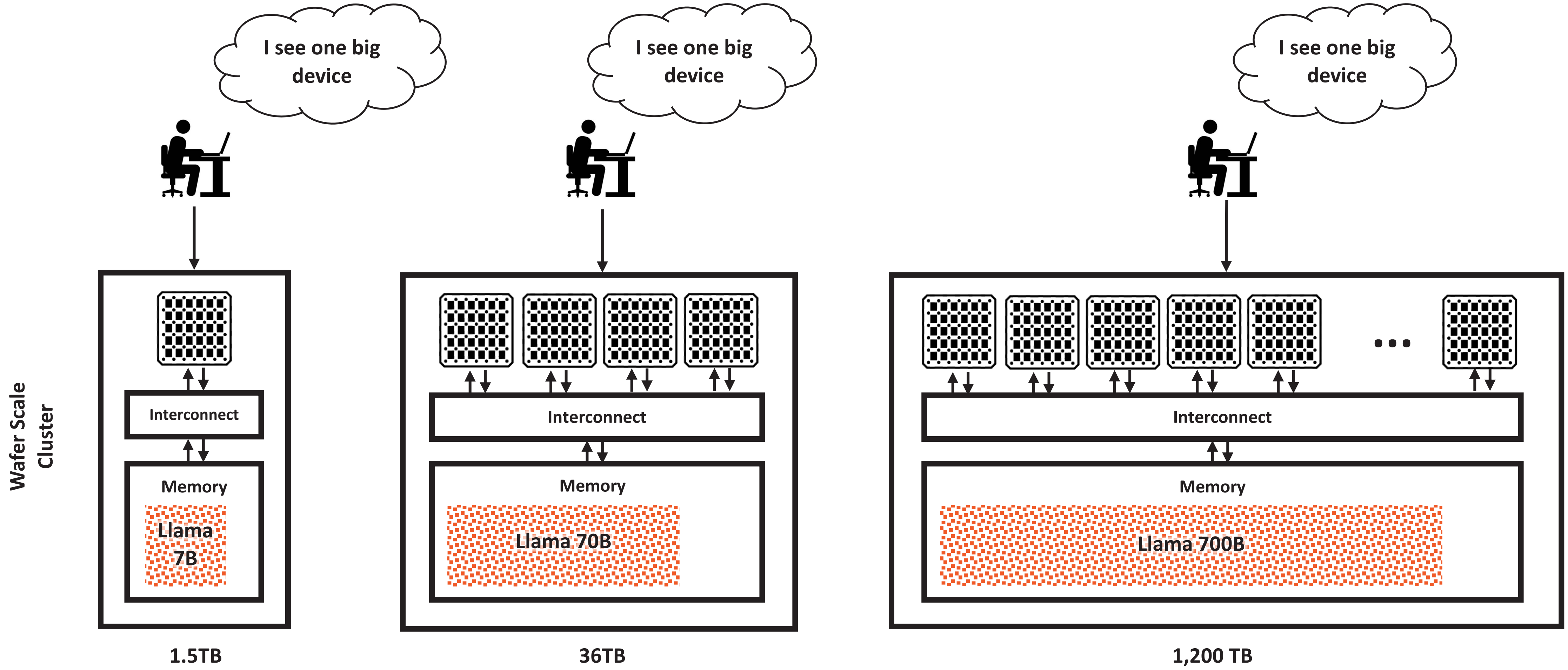
The Cluster Look and Program Like a Single Device

You Program It Like A Single Device
No Matter The Cluster Size

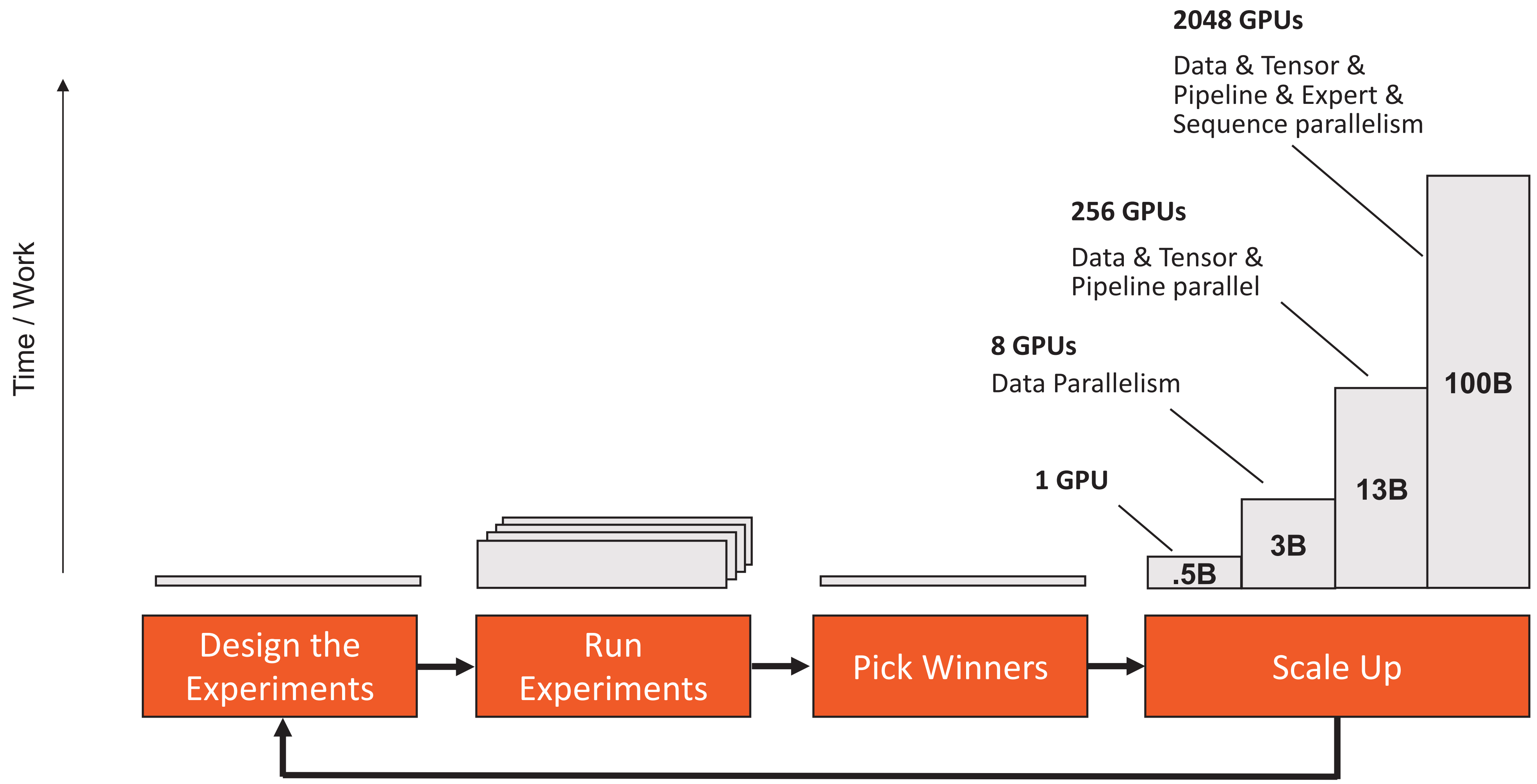


And Your Model Always Fits

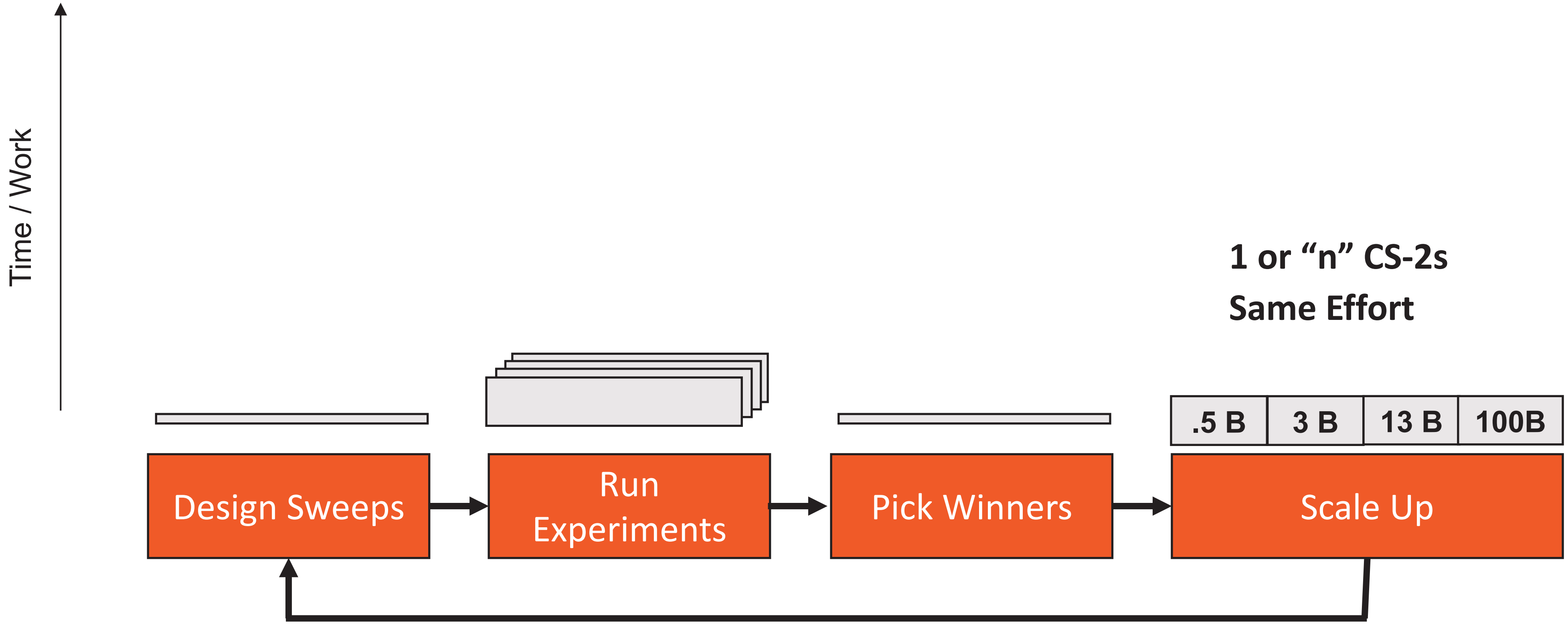
1B or 1T Parameters



How to scale on a GPU?



How to scale on a CS-2?



On GPUs, **small models are the default**;
large models take large engineering effort.

On CS-Xs, **large models are the default**;
small models come for free.

Models on Cerebras

From multi-lingual LLMs to healthcare chatbots to code models

BTLM-3B-8K
3B PARAMETERS • 8K CONTEXT

7B Performance in a 3B Model

Open Source. Trained on Cerebras

CrystalCoder
7B PARAMETERS • 1.3T TOKENS

Coding + English. The most open source & reproducible model in the world.

Open Source. Trained on Cerebras

cerebras

Jais
13B & 30B

State of the art Arabic + English models

Open Weights. Trained on Cerebras

Med42
FINED-TUNED LLAMA2-70B

Medical Q&A LLM Scores 72% on USMLE

Trained on Cerebras

gigaGPT
GPT-3 in 565 LINES OF CODE

Cerebras implementation of nanoGPT

Open Source. Trained on Cerebras

SlimPajama
627BTOKEN DATASET

Extensively deduplicated dataset with twice the perf/token

Open Source

Cerebras-GPT
111M-13B PARAMETERS

First family of GPT models released under Apache 2.0

Open Source. Trained on Cerebras

All the Latest ML Techniques & Recipes

RAG

LoRA

Variable Seq Training
DPO

LL360 – Open data, models, scripts

Sparse Models

Multi Modal

Multi-lingual
Pre-training & IFT

Llama70B fine tuning
Domain Adaptation

GPT-3 in 565 lines of code

Most FLOP efficient LLM dataset

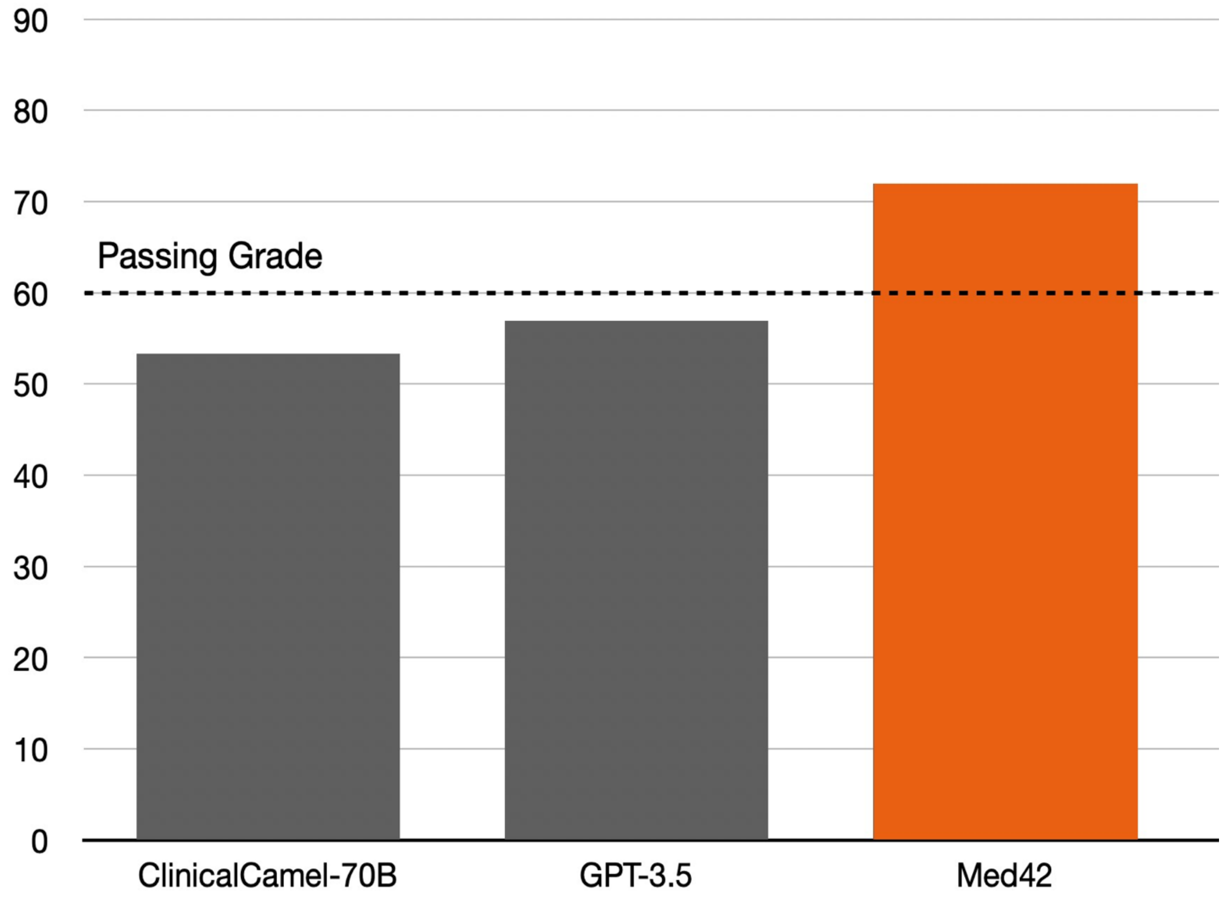
First family of open GPT models and OSS use of muP

MoE

Med42: Llama-70B Fine-tuned in <1 Week to Pass the US Medical License Exam

MED42 By M42

- Scored **72% on USMLE**, beating GPT-3.5
- With M42: global healthcare company with over 450 hospitals and clinics
- Custom curated healthcare dataset of peer-reviewed papers, medical textbooks, international health agency datasets.
- Run finished in 1 weekend

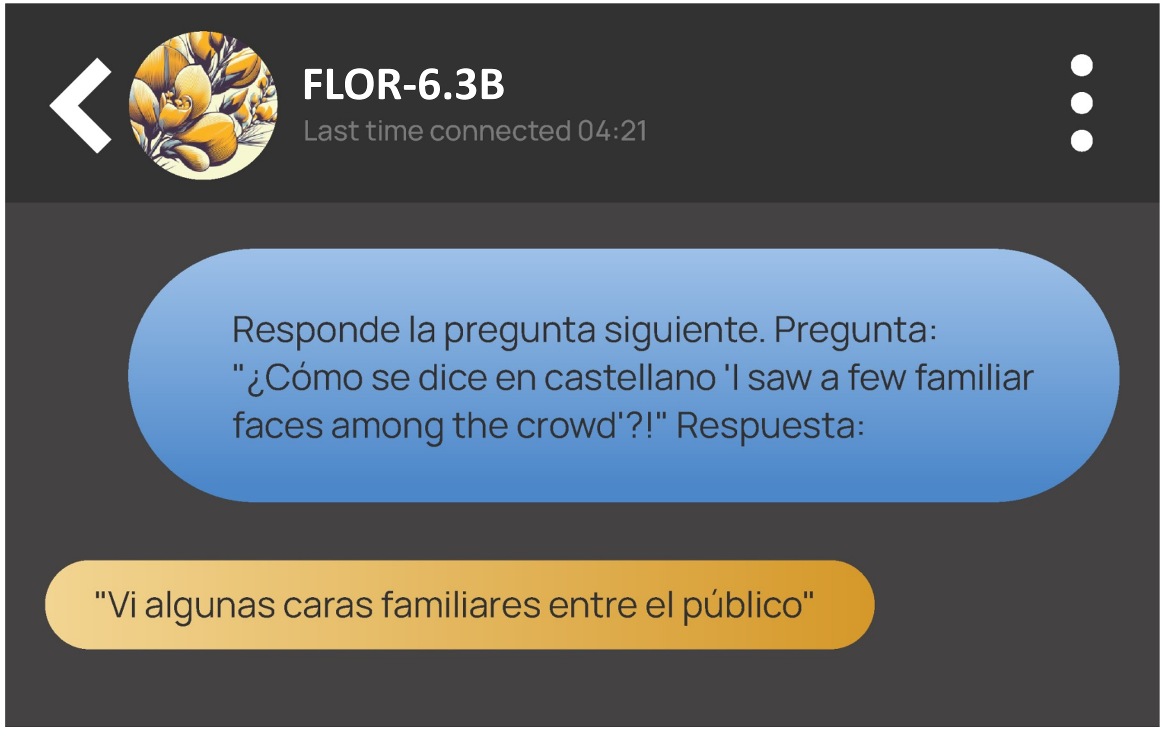


FLOR-6.3B State-of-the-Art Catalan, Spanish, and English LLM

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

- **Best Catalan model**, beating BLOOM-7.3B
- **Used latest language adaptation techniques** for languages with less training data
- **Reduced inference cost by 10%** vs. BLOOM, incorporating a new, more efficient tokenizer
- **Used to build RAG systems** for specialized domains

- Trained on 140B Tokens and in 2.5 days.
- **Open Source:** Downloaded over 3000 times



JAIS-30B: State-of-the-Art Arabic-English Bilingual LLM

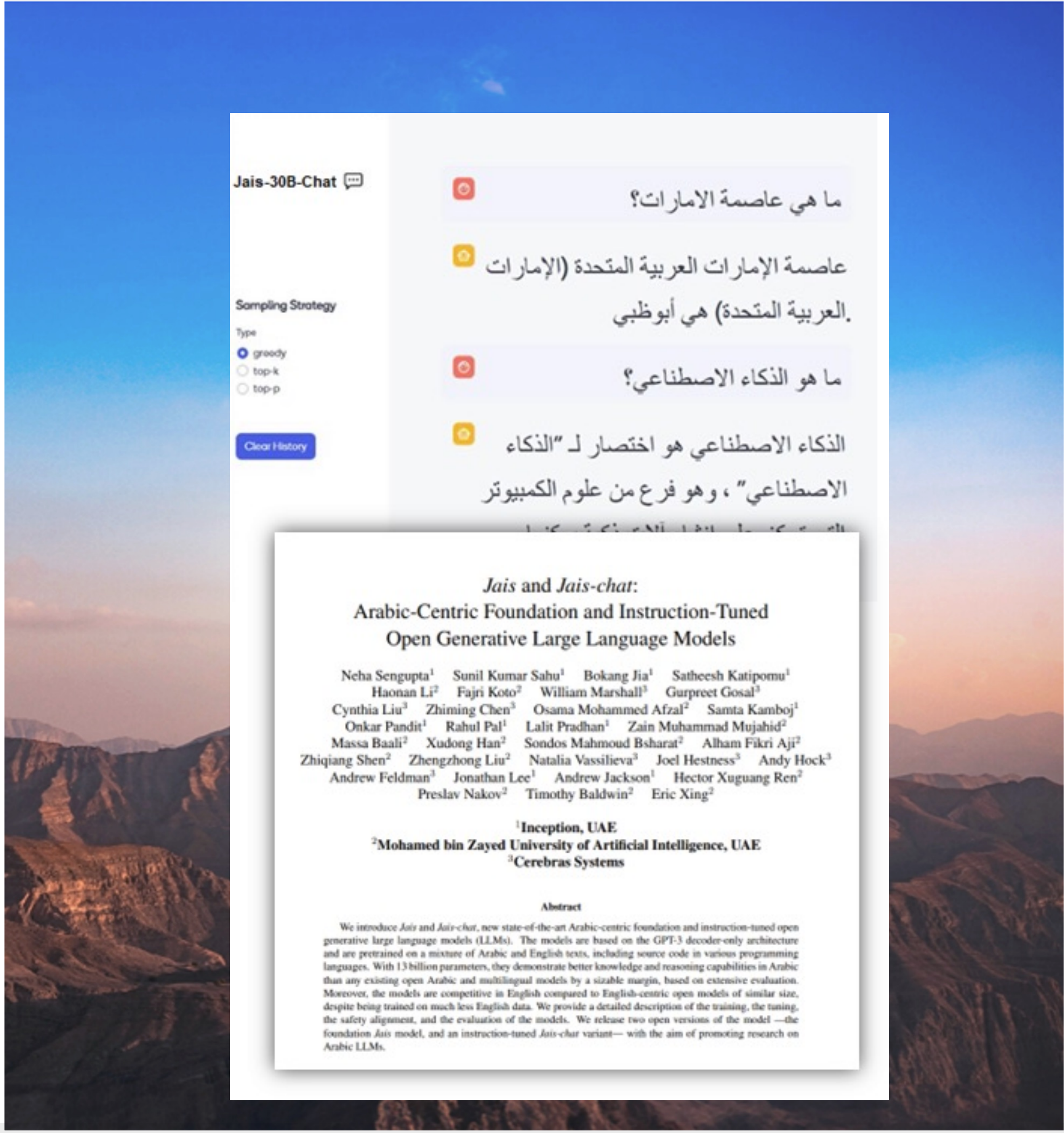
- **SoTA Arabic:** Outperforms all other Arabic models
- **English:** Llama-30B quality in English

- **Co-developed** with G42's Core42 and MBZUAI

- **Now on Azure AI Cloud** as the foundation of their Model-as-a-Service in the Middle East

Checkpoints on HuggingFace

Paper available on Arxiv



Cerebras & GlaxoSmithKline

"On a Cerebras system we pre-trained our EBERT model for 1.75 epochs of 127 epigenomes in ~2.5 days with batch size 8192, which we estimate would have taken ~24 days of training on a GPU cluster with 16 nodes."

"The training speedup afforded by the Cerebras system enabled us to explore architecture variations, tokenization schemes and hyperparameter settings in a way that would have been prohibitively time and resource intensive on a typical GPU cluster."

24 days reduced to 2.5 days with Cerebras

Paper: https://arxiv.org/abs/2112.07571

Epigenomic language models powered by Cerebras

Meredith V. Trotter[1*], Cuong Q. Nguyen[1], Stephen Young[1*], Rob T. Woodruff[1*], Kim M. Branson[1]

[1]Artificial Intelligence and Machine Learning, GlaxoSmithKline

*{meredith.v.trotter, stephen.r.young, rob.x.woodruff}@gsk.com

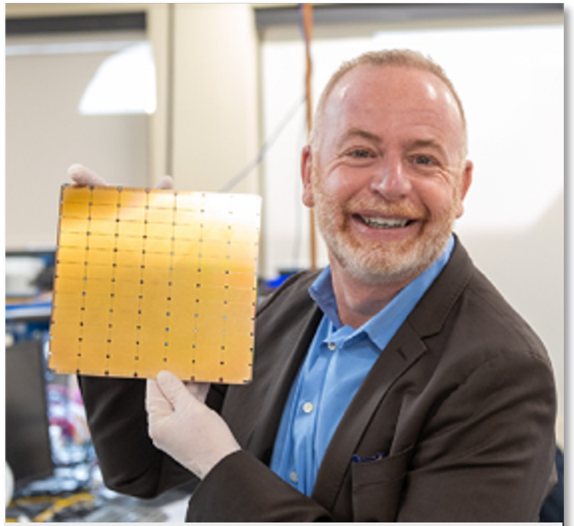
Abstract

Large scale self-supervised pre-training of Transformer language models has advanced the field of Natural Language Processing and shown promise in cross-application to the biological 'languages' of proteins and DNA. Learning effective representations of DNA sequences using large genomic sequence corpuses may accelerate the development of models of gene regulation and function through transfer learning. However, to accurately model cell type-specific gene regulation and function, it is necessary to consider not only the information contained in DNA nucleotide sequences, which is mostly invariant between cell types, but also how the local chemical and structural 'epigenetic state' of chromosomes varies between cell types. Here, we introduce a Bidirectional Encoder Representations from Transformers (BERT) model that learns representations based on both DNA sequence and paired epigenetic state inputs, which we call Epigenomic BERT (or EBERT). We pre-train EBERT with a masked language model objective across the entire human genome and across 127 cell types. **Training this complex model with a previously prohibitively large dataset was made possible for the first time by a partnership with Cerebras Systems, whose CS-1 system powered all pre-training experiments.** We show EBERT's transfer learning potential by demonstrating strong performance on a cell type-specific transcription factor binding prediction task. Our fine-tuned model exceeds state of the art performance on 4 of 13 evaluation datasets from ENCODE-DREAM benchmarks and earns an overall rank of 3rd on the challenge leaderboard. We explore how the inclusion of epigenetic data and task-specific feature augmentation impact transfer learning performance.

TotalEnergies achieves 228x speedup vs. A100 on seismic imaging algorithm

*“As can be seen, when the largest problem is solved, a speedup of 228x is achieved... **Moreover…it is unlikely that such a performance gap can be closed**… given the strong scalability issues encountered by this kind of algorithm when using a large number of multi-GPU nodes in HPC clusters.”*

Speedup of 228x achieved with Cerebras



Diego Klahr VP
VP of Engineering at TotalEnergies

Paper: https://arxiv.org/abs/2204.03775

Massively scalable stencil algorithm

Mathias Jacquelin[§]
Cerebras Systems Inc.
Sunnyvale, California, USA
mathias.jacquelin@cerebras.net

Mauricio Araya-Polo[§] and Jie Meng
TotalEnergies EP Research & Technology US, LLC.
Houston, Texas, USA
mauricio.araya@totalenergies.com

arXiv:2204.03775v1 [cs.MS] 7 Apr 2022

***Abstract*—Stencil computations lie at the heart of many scientific and industrial applications. Unfortunately, stencil algorithms perform poorly on machines with cache based memory hierarchy, due to low re-use of memory accesses. This work shows that for stencil computation a novel algorithm that leverages a localized communication strategy effectively exploits the Cerebras WSE-2, which has no cache hierarchy. This study focuses on a 25-point stencil finite-difference method for the 3D wave equation, a kernel frequently used in earth modeling as numerical simulation. In essence, the algorithm trades memory accesses for data communication and takes advantage of the fast communication fabric provided by the architecture. The algorithm —historically memory bound— becomes compute bound. This allows the implementation to achieve near perfect weak scaling, reaching up to 503 TFLOPs on WSE-2, a figure that only full clusters can eventually yield.**

***Index Terms*—Stencil computation, high performance computing, energy, wafer-scale, distributed memory, multi-processor architecture and micro-architecture**

I. Introduction

Stencil computations are central to many scientific problems and industrial applications, from weather forecast ([32]) to earthquake modeling ([19]). The memory access pattern of this kind of algorithm, in which all values in memory are accessed but used in only very few arithmetic operations, is particularly unfriendly to hierarchical memory systems of traditional architectures. Optimizing these memory operations is the main focus of performance improvement research on the topic.

Subsurface characterization is another area where stencils are widely used. The objective is to identify major structures in the subsurface that can either hold hydrocarbon or be used for CO_2 sequestration. One step towards that end is called seismic modeling, where artificial perturbations of the subsurface are modeled solving the wave equation for given initial and boundary conditions. Solving seismic modeling efficiently is crucial for subsurface characterization, since many perturbation sources need to be modeled as the subsurface model iteratively improves. The numerical simulations required by seismic algorithms for field data are extremely demanding, falling naturally in the HPC category and requiring practical evaluation

[§]Equal contribution.

Traditional architecture	WSE
L1	Memory
L2 & L3	∅
DRAM	∅
Off-node interconnect	Fabric & routers

TABLE I: Equivalences between traditional architectures and the WSE

of technologies and advanced hardware architectures to speed up computations.

Advances in hardware architectures have motivated algorithmic changes and optimizations to stencil applications for at least 20 years ([23]). Unfortunately, the hierarchical memory systems of most current architectures is not well-suited to stencil applications, therefore limiting performance. This applies to multi-core machines, clusters of multi-cores, and accelerator-based platforms such as GPGPUs, FPGAs, etc. ([2], [5]). Alternatively, non-hierarchical architectures were explored in this context, such as the IBM Cell BE ([3]), yielding high computational efficiency but with limited impact.

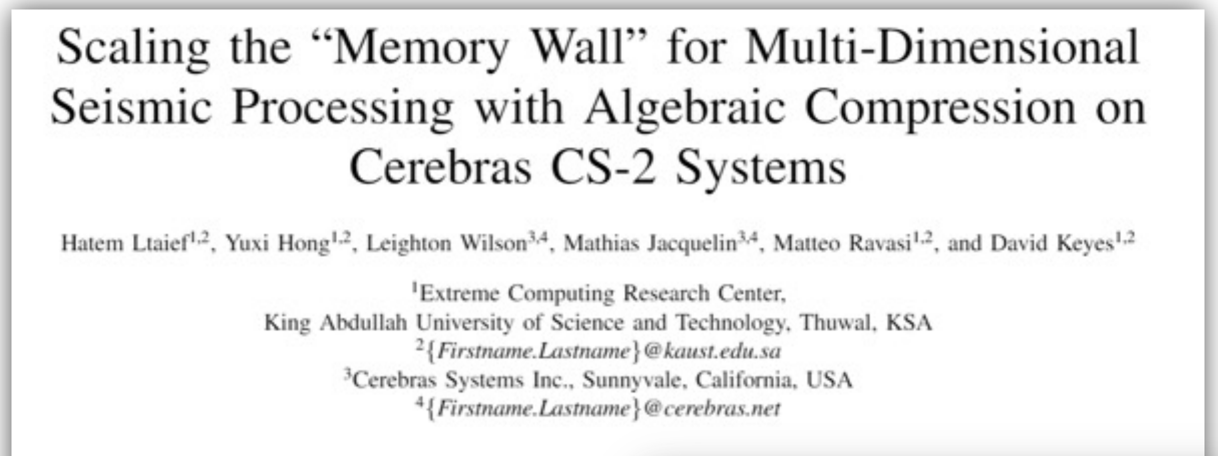
A key element for large scale simulations is the potential of deploying substantial number of processing units connected by an efficient fabric. The Cell BE lacked the former and it had limited connectivity. Another example of non-hierarchical memory system is the Connection Machine ([12]), which excelled on scaling but at the cost of a very complex connectivity. In this work, a novel stencil algorithm based on localized communications that does not depend on memory hierarchy optimizations is introduced. This algorithm can take advantage of architectures such as the WSE from Cerebras ([4]) and potentially Anton 3-like systems ([28]). These are examples of architectures addressing both limitations described above.

Another angle to be considered is the availability of hardware-based solutions in the market. Literature review yields no generally available hardware architecture addressing the specific bottlenecks of stencil applications. Only a few custom designs examples are available ([10], [14]).

In this work, an implementation of such seismic modeling method on a novel architecture is presented. The proposed mapping requires a complete redesign of the basic stencil algorithm. The contribution of this work is multi-fold:

KAUST uses Cerebras CS-2 cluster to achieve performance of the world's #1 supercomputer at 1/10th the cost

"We report ***92.58PB/s*** *sustained throughput, more than 3X faster than the aggregated theoretical bandwidth of Leonardo or Summit...* ***Our bandwidth score thus outperforms the fastest supercomputer Frontier and is comparable to Fugaku, at a much lower acquisition and operational cost****."*



Scaling the "Memory Wall" for Multi-Dimensional Seismic Processing with Algebraic Compression on Cerebras CS-2 Systems

Hatem Ltaief[1,2], Yuxi Hong[1,2], Leighton Wilson[3,4], Mathias Jacquelin[3,4], Matteo Ravasi[1,2], and David Keyes[1,2]

[1]Extreme Computing Research Center,
King Abdullah University of Science and Technology, Thuwal, KSA
[2]{Firstname.Lastname}@kaust.edu.sa
[3]Cerebras Systems Inc., Sunnyvale, California, USA
[4]{Firstname.Lastname}@cerebras.net

Abstract— We exploit the high memory bandwidth of AI-customized Cerebras CS-2 systems for seismic processing. By leveraging low-rank matrix approximation, we fit memory-hungry seismic applications onto memory-austere SRAM wafer-scale hardware, thus addressing a challenge arising in many wave-equation-based algorithms that rely on Multi-Dimensional Convolution (MDC) operators. Exploiting sparsity inherent in seismic data in the frequency domain, we implement embarrassingly parallel tile low-rank matrix-vector multiplications (TLR-MVM), which account for most of the elapsed time in MDC operations, to successfully solve the Multi-Dimensional Deconvolution (MDD) inverse problem. By reducing memory footprint along with arithmetic complexity, we fit a standard seismic benchmark dataset into the small local memories of Cerebras processing elements. Deploying TLR-MVM execution onto 48

Tony Chan
President, KAUST

Paper: https://dl.acm.org/doi/10.1145/3581784.3627042

Argonne National Labs Uses CS-2 to Accelerate Monte Carto Particle Transport by **130x** Over A100

“The WSE is found to run ***130 times faster*** *than a highly optimized CUDA version of the kernel run on an NVIDIA A100 GPU – significantly outpacing the expected performance increase given the relative number of transistors each architecture has”*

Upcoming PHYSOR publication demonstrates **180x** over A100.

Paper: https://arxiv.org/abs/2311.01739

Efficient Algorithms for Monte Carlo Particle Transport on AI Accelerator Hardware

John Tramm[a,*], Bryce Allen[a,b], Kazutomo Yoshii[a], Andrew Siegel[a], Leighton Wilson[c]

[a] *Argonne National Laboratory, 9700 S Cass Ave., Lemont, 60439, IL, USA*
[b] *University of Chicago, 5801 S. Ellis Ave., Chicago, 60637, IL, USA*
[c] *Cerebras Systems Inc., 1237 E Arques Ave, Sunnyvale, 94085, CA, USA*

Abstract

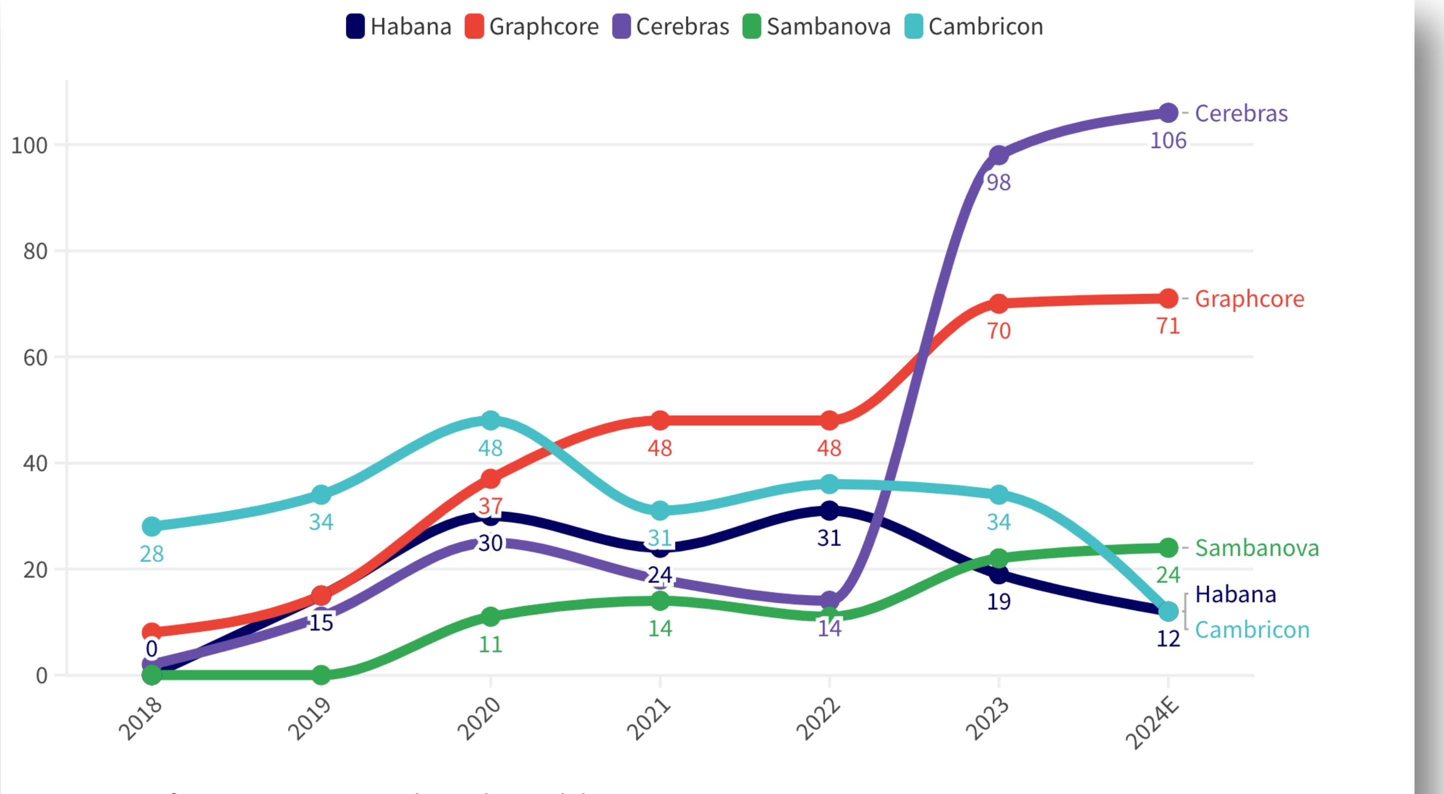
The recent trend in computing towards deep learning has resulted in the development of a variety of highly innovative AI accelerator architectures. One such architecture, the Cerebras Wafer-Scale Engine 2 (WSE2), features 40 GB of on-chip SRAM making it an attractive platform for latency- or bandwidth-bound HPC simulation workloads. In this study, we examine the feasibility of performing continuous energy Monte Carlo (MC) particle transport by porting a key kernel from the MC transport algorithm to Cerebras' CSL programming model. We then optimize the kernel and experiment with several novel algorithms for decomposing data structures across the WSE2's 2D network grid of approximately 750,000 user-programmable distributed memory compute cores and for flowing particles (tasks) through the WSE2's network for processing. New algorithms for minimizing communication costs and for handling load balancing are developed and tested. The WSE2 is found to run 130 times faster than a highly optimized CUDA version of the kernel run on an NVIDIA A100 GPU — significantly outpacing the expected performance increase given the relative number of transistors each architecture has.

Cerebras is the #1 AI Semiconductor Startup

Cerebras is the leader in Generative AI and High-Performance Computing publications

Committed to accelerating research through open-source, including:

- State-of-the-art models (BTLM, Jais-30B)
- Datasets and scripts (SlimPajama)
- Model training frameworks (GigaGPT)



Link to report: https://press.airstreet.com/p/state-of-ai-report-compute-index-v3

We appreciate this opportunity to present you our system,

and importantly,

to discuss how we can help you accelerate research
And explore new scientific frontiers.

Hardware and Systems

Cerebras Wafer-Scale Engine (WSE-2)

Still the Largest Chip Ever Made

850,000 cores optimized for sparse linear algebra

46,225 mm² silicon

2.6 trillion transistors

40 gigabytes of on-chip memory

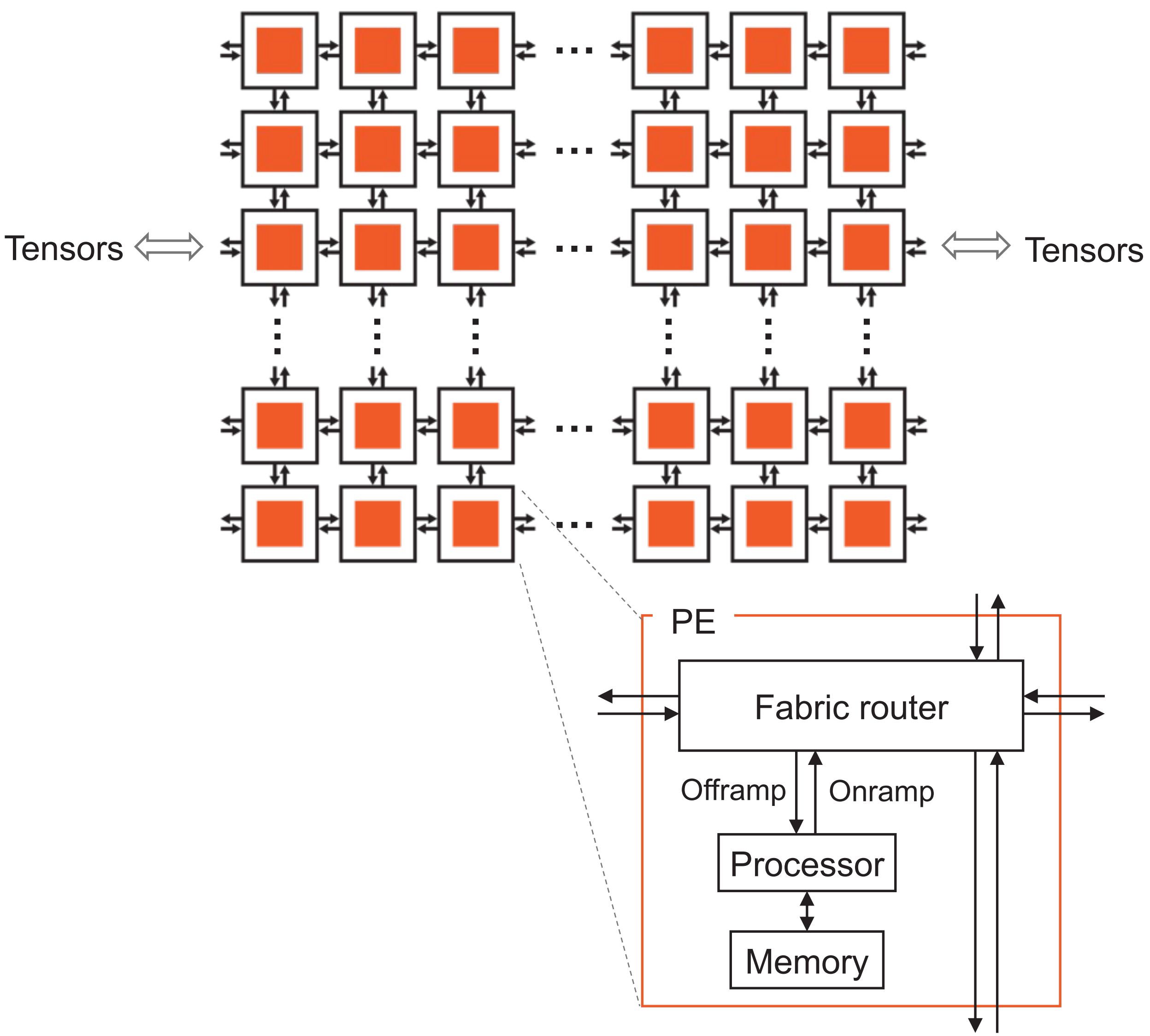
20 PByte/s memory bandwidth

220 Pbit/s fabric bandwidth

7nm process technology

Cluster-scale performance in a single chip

WSE Architecture Basics



The WSE appears as a logical 2D array of individually programmable Processing Elements

Flexible compute

- 850,000 general purpose CPUs
- 16- and 32-bit native FP and integer data types
- **Dataflow programming**: Tasks are activated or triggered by the arrival of data packets

Flexible communication

- Programmable router
- Static or dynamic routes (**colors**)
- Data packets (**wavelets**) passed between PEs
- 1 cycle for PE-to-PE communication

Fast memory

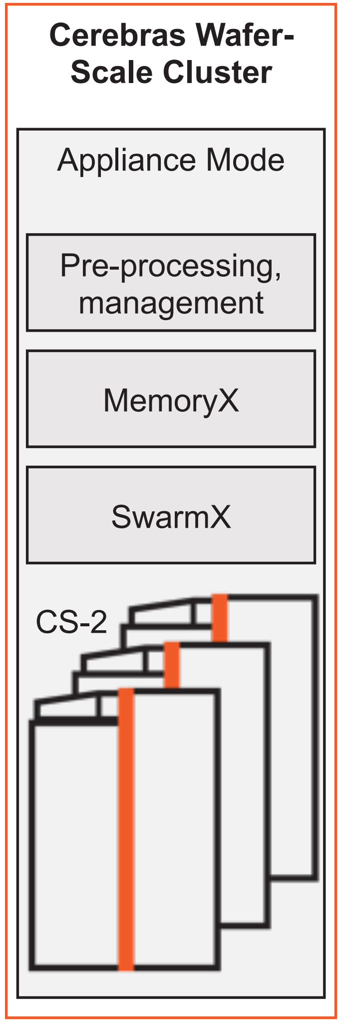
- 40GB on-chip SRAM
- Data and instructions
- 1 cycle read/write

Wafer Scale Cluster

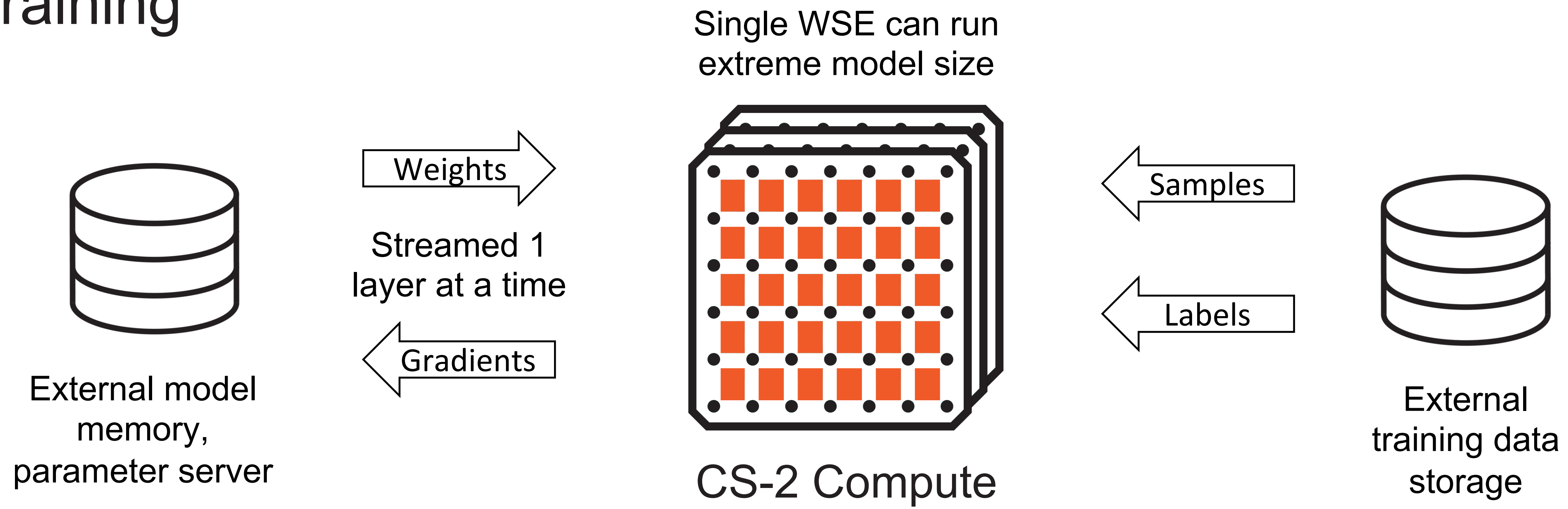
- Purpose-built high performance, scalable appliance
  - Complete hardware + software solution for large-scale AI
  - One to many CS-2s

- Datacenter-scale AI compute in a single row or lab
  - CS-2 accelerator(s)
  - Disaggregated, independently scalable parameter storage
  - High performance smart interconnect fabric
  - Standards-based input and management workers

- Benefits
  - Run the largest models today on a single machine
  - Scale up model size with a single line code change
  - Scale out to go faster with near-linear performance
  - One or many machines programmable as a single node
  - Simple data-parallel scaling; no need for complex model- / tensor-parallel distribution



Cerebras Weight Streaming technology disaggregates storage and compute to enable trillion parameter model training



Scale model size and training speed independently

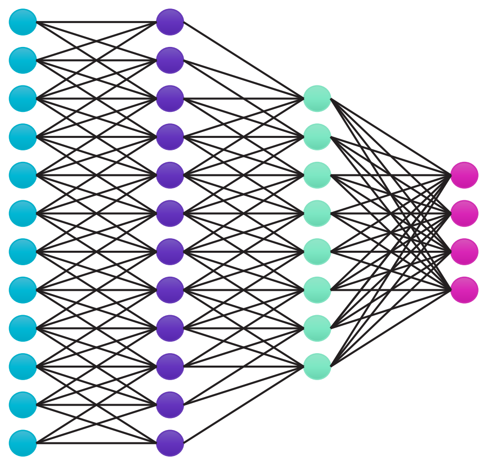
Weight Streaming Execution Model

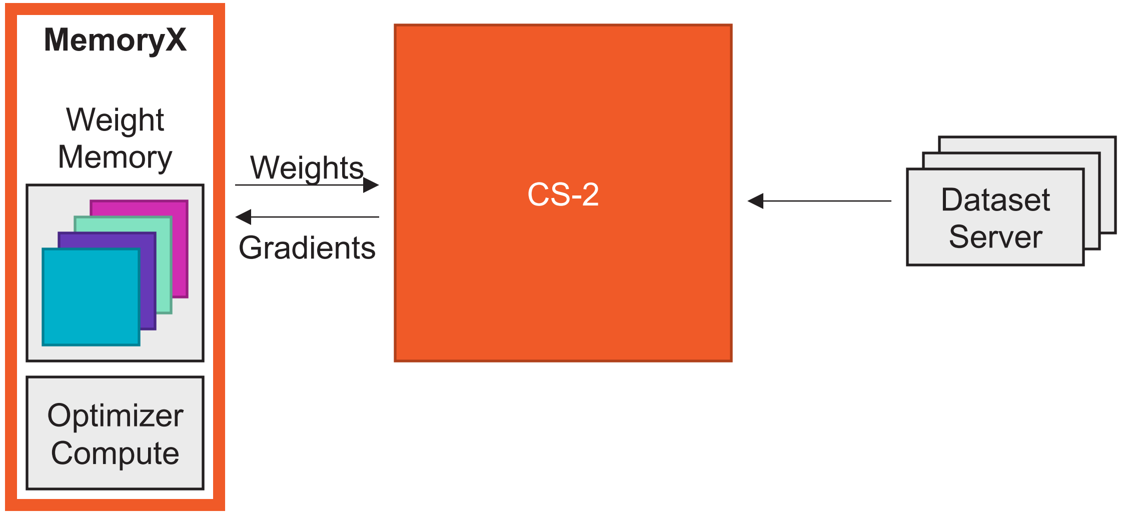
Built for extreme-scale neural networks:

- Weights stored externally off-wafer
- Weights streamed onto wafer to compute layer
- Activations only are resident on wafer
- Execute one layer at a time

Decoupling weight optimizer compute

- Gradients streamed out of wafer
- Weight update occurs in MemoryX

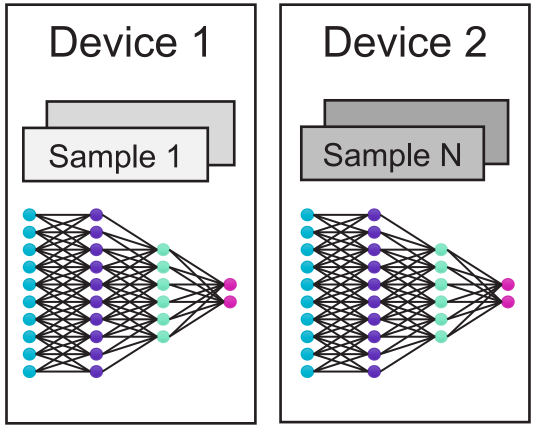




Challenges to Scaling

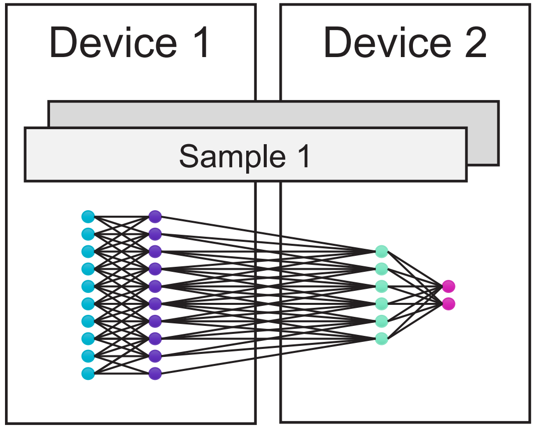
Hybrid parallelism on traditional devices

Data Parallel



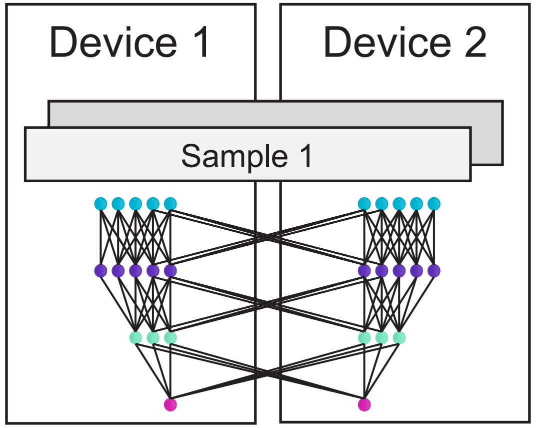
Multiple samples at a time
Parameter memory limits

Pipelined Model Parallel



Multiple layers at a time
Communication overhead
N^2 activation memory

Tensor Model Parallel



Multiple splits at a time
Communication overhead
Complex partitioning

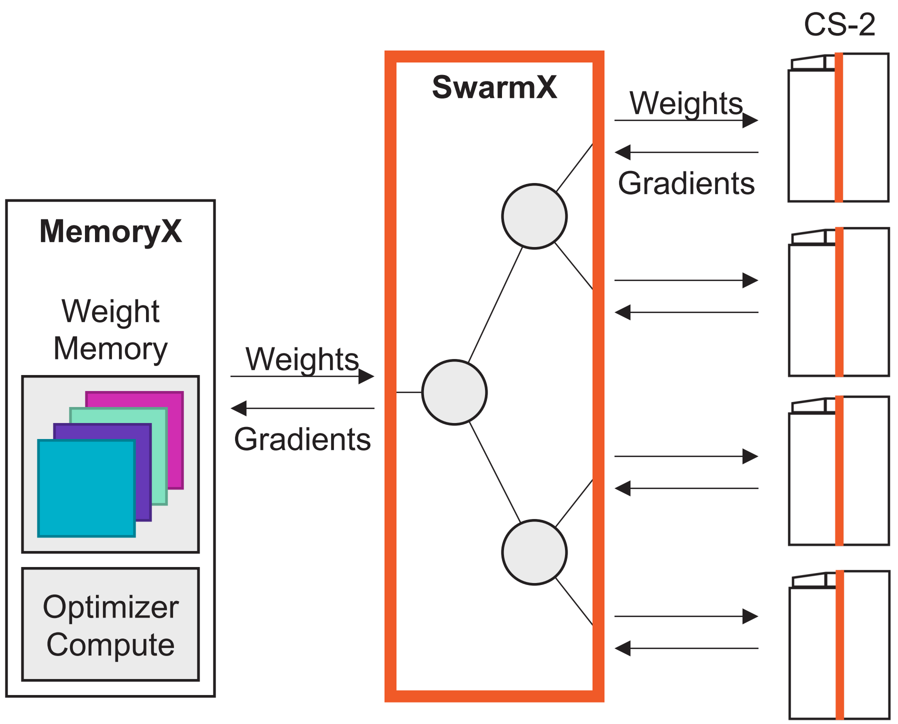
Distribution complexity scales dramatically with cluster size

cerebras

Near-Linear Data Parallel Only Scaling

Specialized interconnect for scale-out

- Data parallel distribution through SwarmX interconnect
- Weights are **broadcast** to all CS-2s
- Gradients are **reduced** on way back



Multi-system scaling with the same execution as single system

- Same system architecture
- Same network execution flow
- Same software user interface

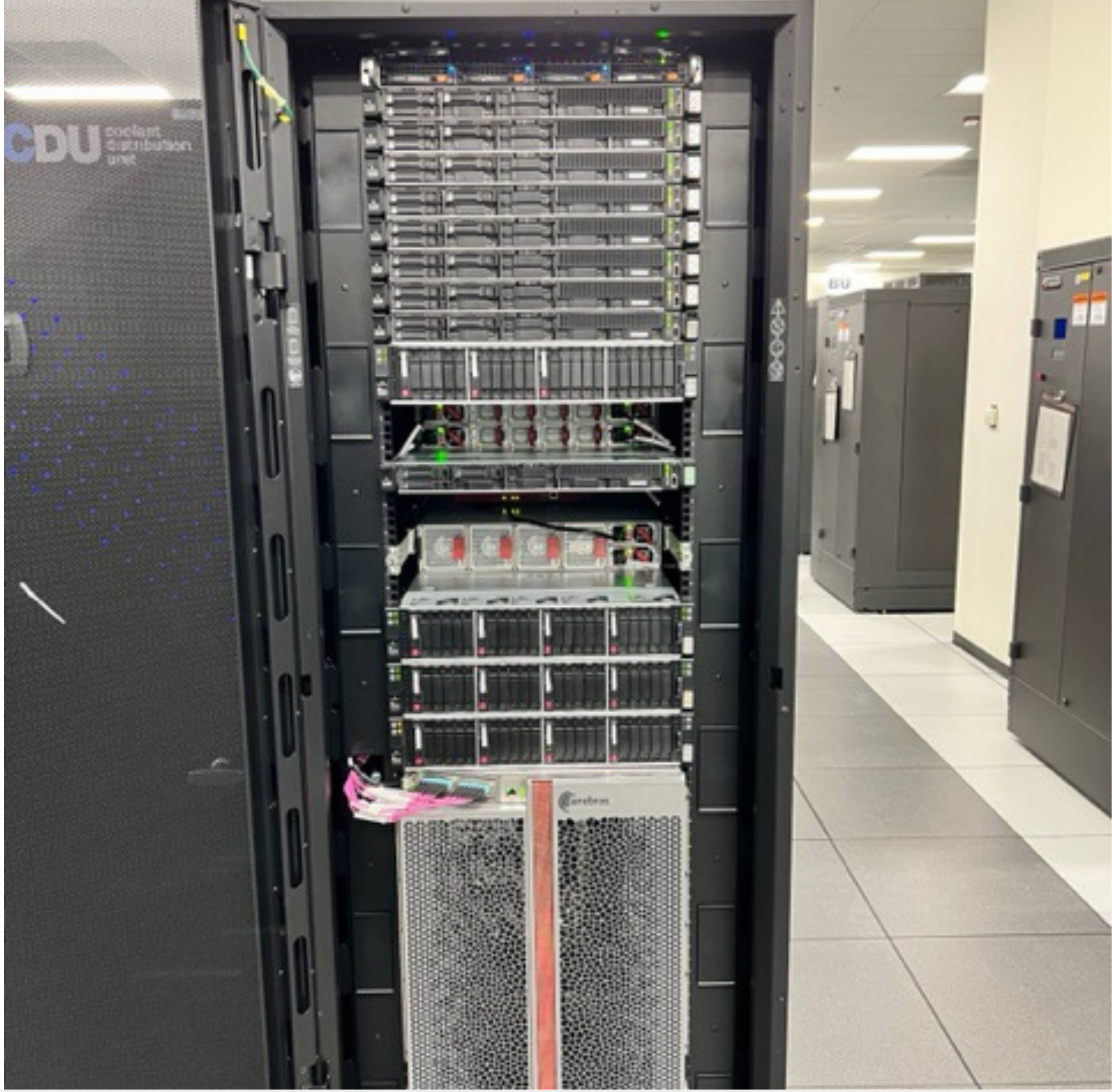
Cerebras WS Cluster Differentiators

- Many independent small cores
  - 850,000 processor cores
  - Each core has its own program code and HW scheduler
- Large on-chip memory near compute
  - Distributed architecture, all cores have dedicated memory
  - Single clock cycle memory access
- Sparsity acceleration
  - Enabled by fine-grained dataflow and high memory bandwidth
  - Speed up structured and unstructured sparsity
- Disaggregated compute and parameter memory
  - Scaling to multiple chips with only data parallelism
- Simple programming and linear performance scaling

Want to Dive Deeper? Check out our Hot Chips 34 Presentation: https://hc34.hotchips.org/

Cerebras systems at ALCF

- 2-node Wafer-Scale Cluster
  - Supporting up to 30B parameter models
  - GenAI-optimized:
    - NLP (LLMs)
    - Multimodal VQA
  - 2x CS-2s, with:
    - 850k cores each
    - 40GB on chip memory each
  - Can distribute jobs across one or both CS-2s, with data parallel scaling when using both machines

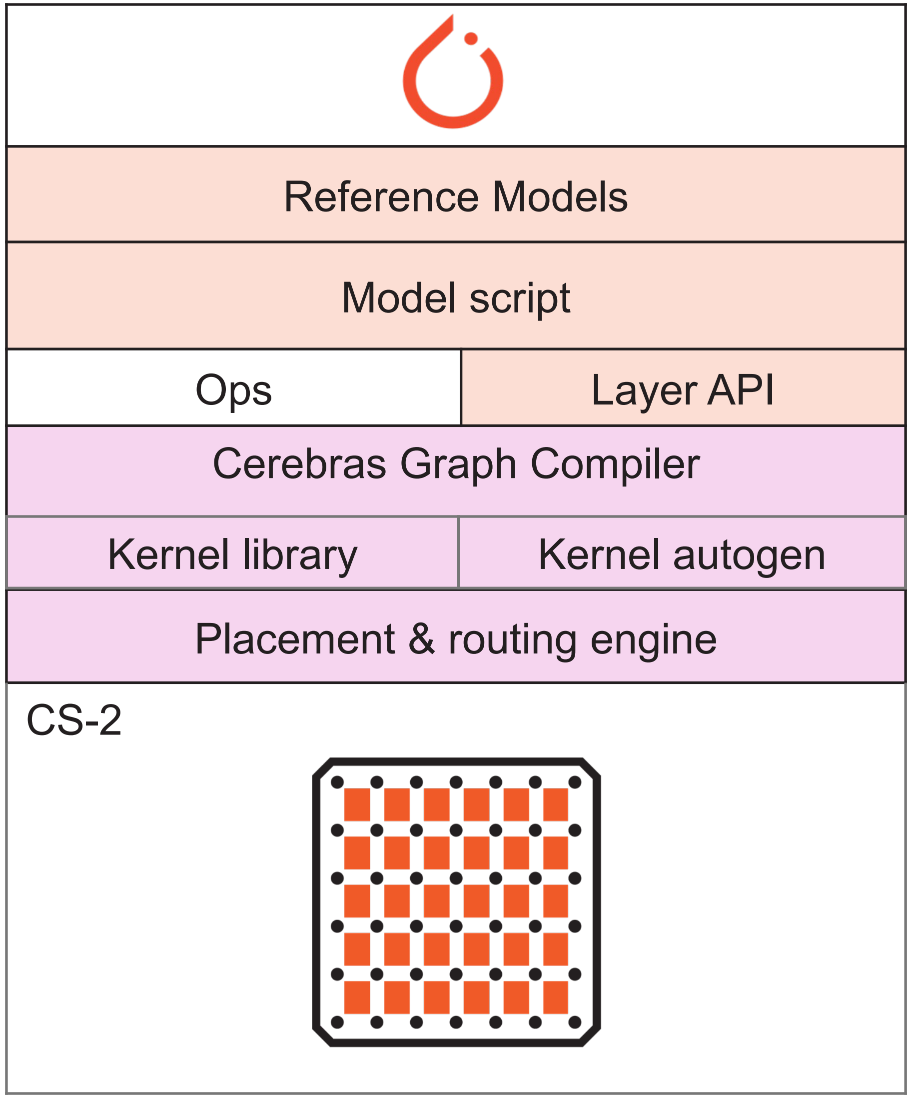


Software and Programming

Lowering from Model to Wafer

Integration with PyTorch

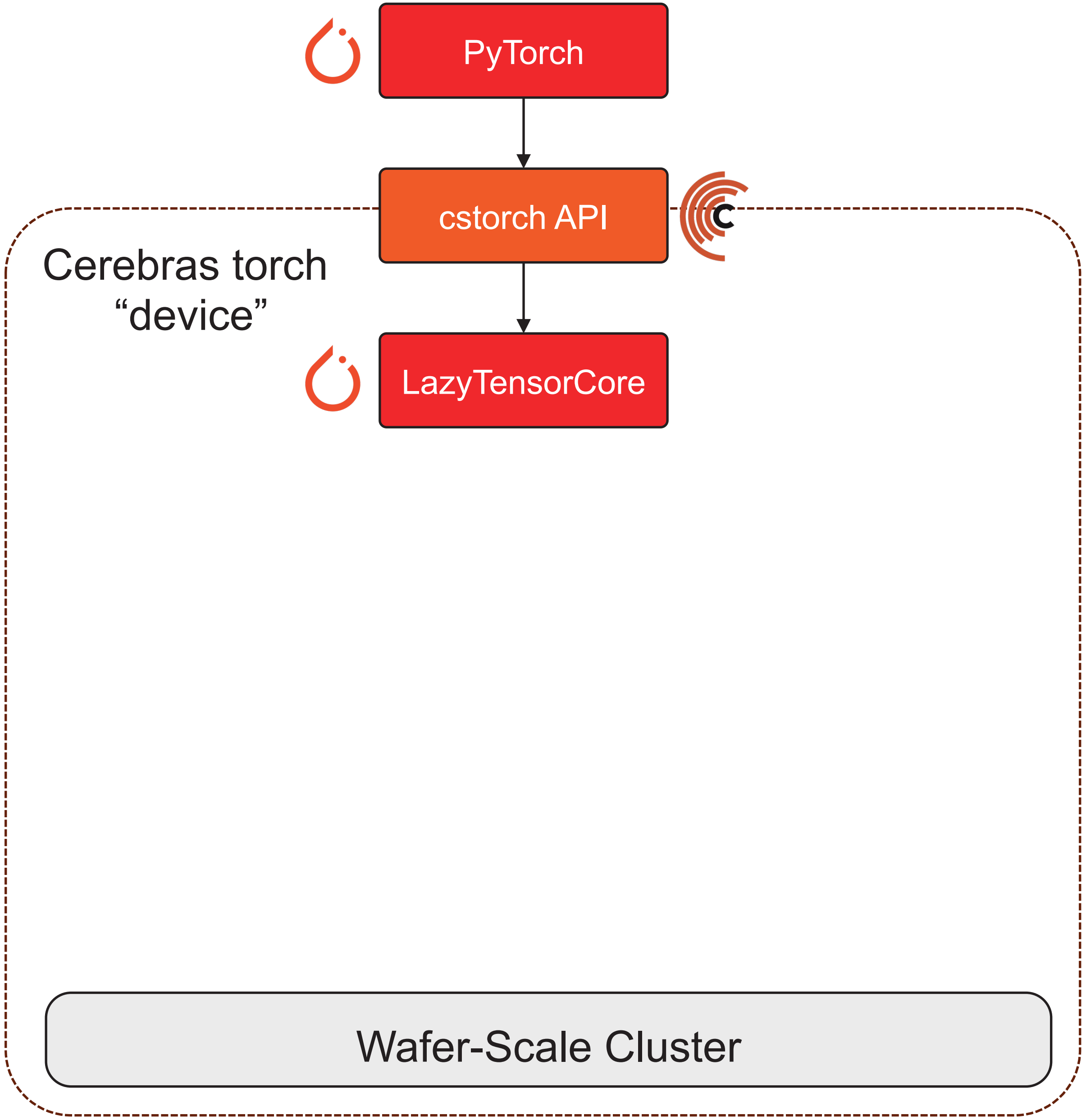
- Models defined in framework + Cerebras API
- Optimally maps from PyTorch to high performance kernels
  - Uses polyhedral code-generation or hand-written kernels
- Compiler using industry standard MLIR framework
  - Cerebras is an active contributor to the MLIR open- source community
- User does not worry about distributed compute or parallelism



cstorch Software Stack

Frontend API

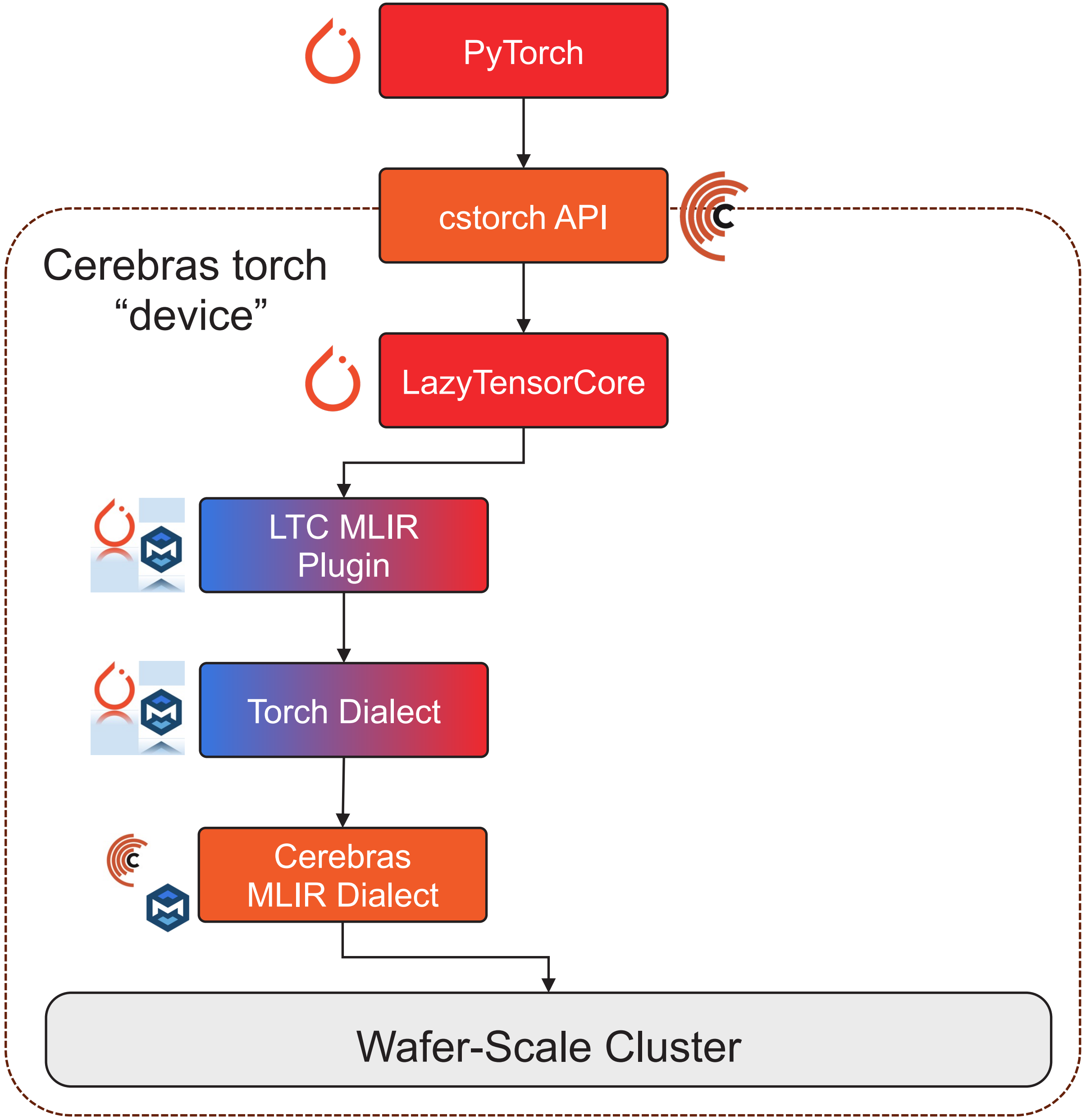
- `cstorch` API mirrors `torch` API
  - Helps with single device abstraction
- Tensor Ops traced through LazyTensorCore (LTC)
  - Graph-by-execution with lazy evaluation
  - Also drives Google's xla/tpu device



cstorch Software Stack

Compilation

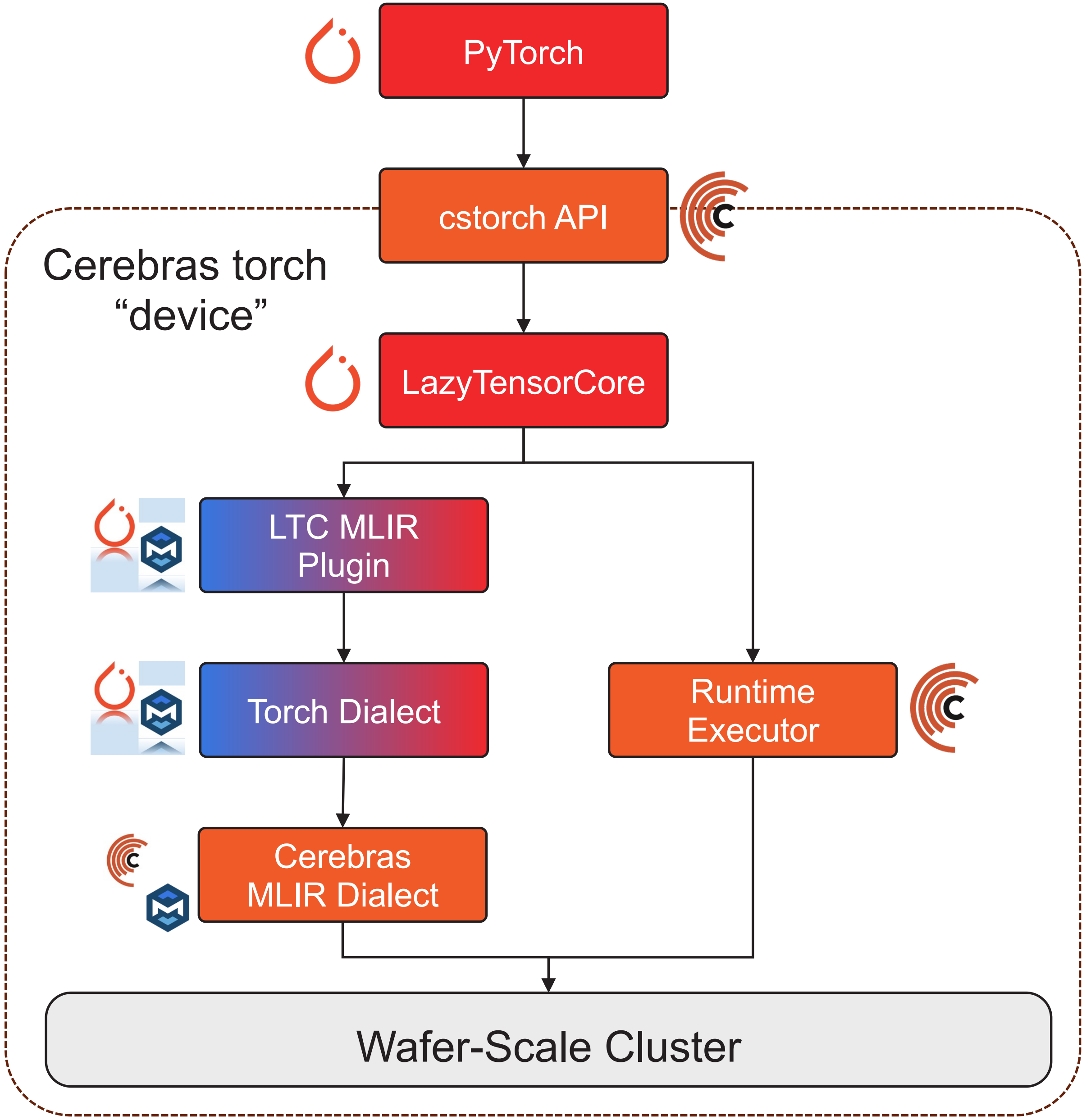
- `cstorch` API mirrors `torch` API
  - Helps with single device abstraction
- Tensor Ops traced through LazyTensorCore
  - Graph-by-execution with lazy evaluation
  - Also powers Google's xla/tpu device
- MLIR translation from LTC provided by torch-mlir
  - Hardware focused compiler ecosystem for torch
- Cerebras MLIR stack handles cluster optimizations



cstorch Software Stack

Runtime Executor

- `cstorch` API mirrors `torch` API
  - Helps with single device abstraction
- Tensor Ops traced through LazyTensorCore
  - Graph-by-execution with lazy evaluation
  - Also powers Google's xla/tpu device
- MLIR translation from LTC provided by torch-mlir
  - Hardware focused compiler ecosystem for torch
- Cerebras MLIR stack handles cluster optimizations
- Tensors get transferred to cluster as needed
  - Initial weights sent before first step
  - Inputs sent each step from custom data executor
- Execution driven asynchronously by cluster



Running on Cerebras with Cerebras ModelZoo

https://github.com/Cerebras/modelzoo

- Cerebras ModelZoo supports a wide range of decoder-only (GPT-style), encoder-only (BERT-style) and encoder-decoder (T5-style) models
  - Support for various **positional encodings**: learned (GPT), fixed, RoPE (GPT-J, Llama), ALiBi (Bloom)
  - Support for various **activation functions**: relu, gelu (GPT), swiglu (Llama)
  - Support for sequential (GPT, Llama) and parallel (GPT-J, GPT-NeoX) **attention and feed-forward blocks**
  - Support for different **attention types**: vanilla multi-head (GPT), MQA (Llama 7B, 13B), GQA (Llama-2 70B)
- We provide **checkpoint converters** to and from HuggingFace format for many popular models
  - Llama, Llama-2, Falcon, Bloom, CodeGen, Starcoder, and others
- These models can be **trained and fine-tuned** on Cerebras hardware
- **Even the largest models** can run on 1xCS-2
  - Llama 70B requires > 1TB of memory for weights and optimizer states only
  - Full fine-tuning is feasible on 1xCS-2

How to scale from 1B to 70B on Cerebras

gpt3_1b_params.yaml

```
### GPT-3 XL 1.3B

hidden_size: 2048
num_hidden_layers: 24
num_heads: 16
```

llama2_70b_params.yaml

```
### Llama-2 70B

hidden_size: 8192
num_hidden_layers: 80
num_heads: 64
```

Training:

```
python run.py \
--params gpt3_1b_params.yaml \
--num_steps=100 \
--model_dir=model_dir \
```

Training:

```
python run.py \
--params llama2_70B_params.yaml \
--num_steps=100 \
--model_dir=model_dir \
```

Programming / training with the cluster is simple

Define the model

- Write in PyTorch
- Parameterize based on yaml file
- Write *logical* model for *single* device

Train the model

- Point to the model parameters
- Specify the number of CS-2s
- Specify the number of steps
- Run!

params_gpt3xl.yaml

```
### GPT-3 XL 1.3B

hidden_size: 2048
num_hidden_layers: 24
num_heads: 16
```

training:

```
python run.py \
--params params_gpt3xl.yaml \
--num_csx 1 \
--num_steps 100 \
--model_dir model_dir \
--mode train
```

cerebras

Scaling to larger models is simple

Scaling the model

- Change the model parameters in yaml
  - Let's run GPT-NeoX 20B on 4x CS-2s
- Fully data-parallel training
- Run!

params_gptneox.yaml

```
### GPT-NeoX 20B

hidden_size: 6144
num_hidden_layers: 44
num_heads: 64
```

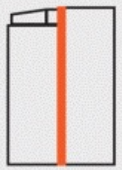
training:

```
python run.py \
--params params_gptneox.yaml \
--num_csx 4 \
--num_steps 100 \
--model_dir model_dir \
--mode train
```

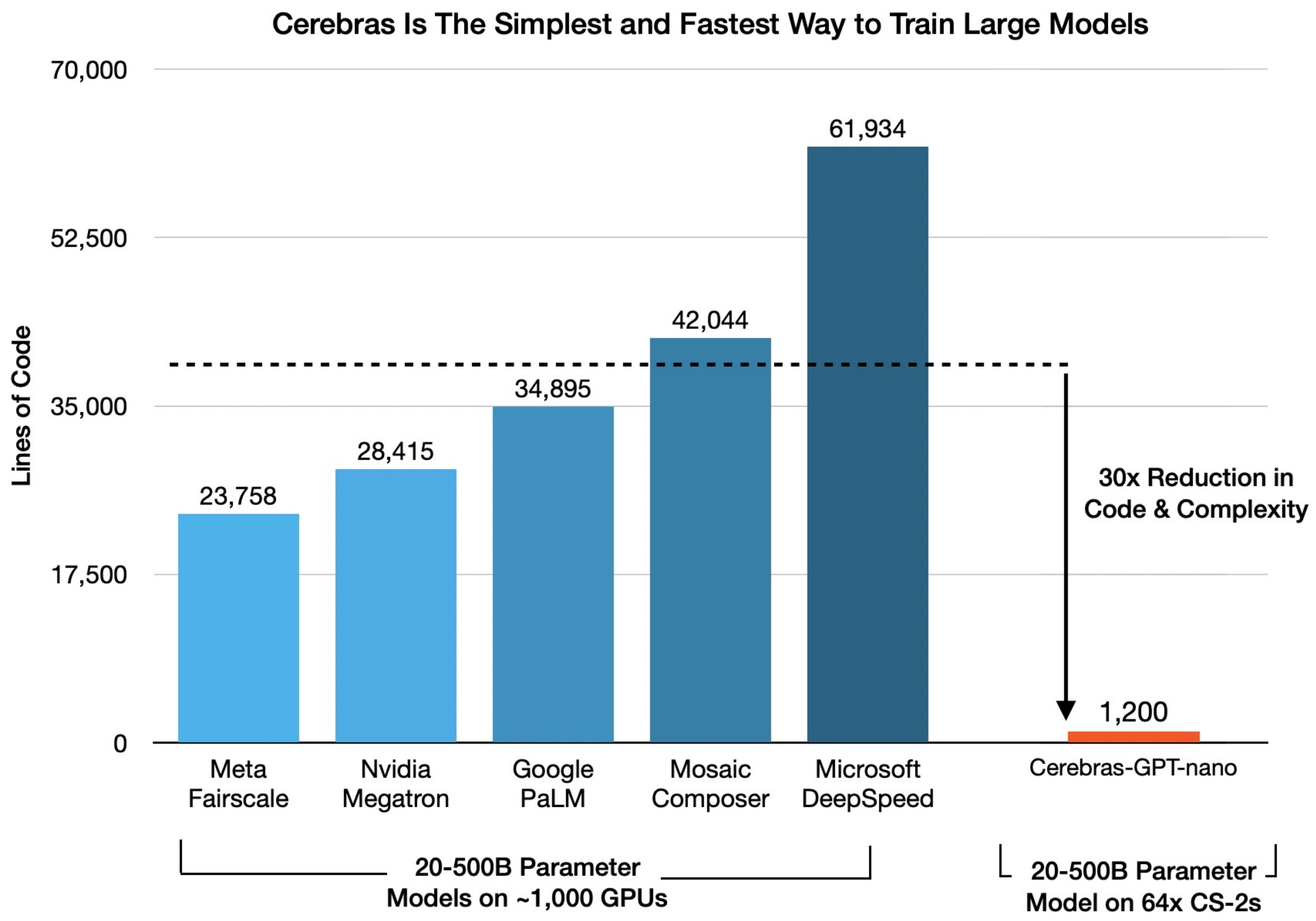
Scaling from one CS-2 to a cluster is a 1-line change

```
python run.py
--params params.yaml
--num_csx = 1
--model_dir = model_dir
--num_steps = 1000
--mode=train
```

How many nodes?



Weight Streaming Simplifies Large Model Training by 30x



Cerebras CS-2 trains 100B parameter models with the ease and simplicity of a GPU training a 1B parameter model.

We made our compute and memory extremely large so that our software can be extremely simple.

The result:

- 30x speed up in implementation
- A fraction the # of ML engineers
- Dramatically faster iteration and experimentation
- Get to market first with far larger and more accurate models.

Data Parallel Models Enables Near Linear Scaling

- Even the largest state-of-the-art models can train on a single CS-2
- Near-linear time to solution scaling across multiple CS-2s in a wafer-scale cluster

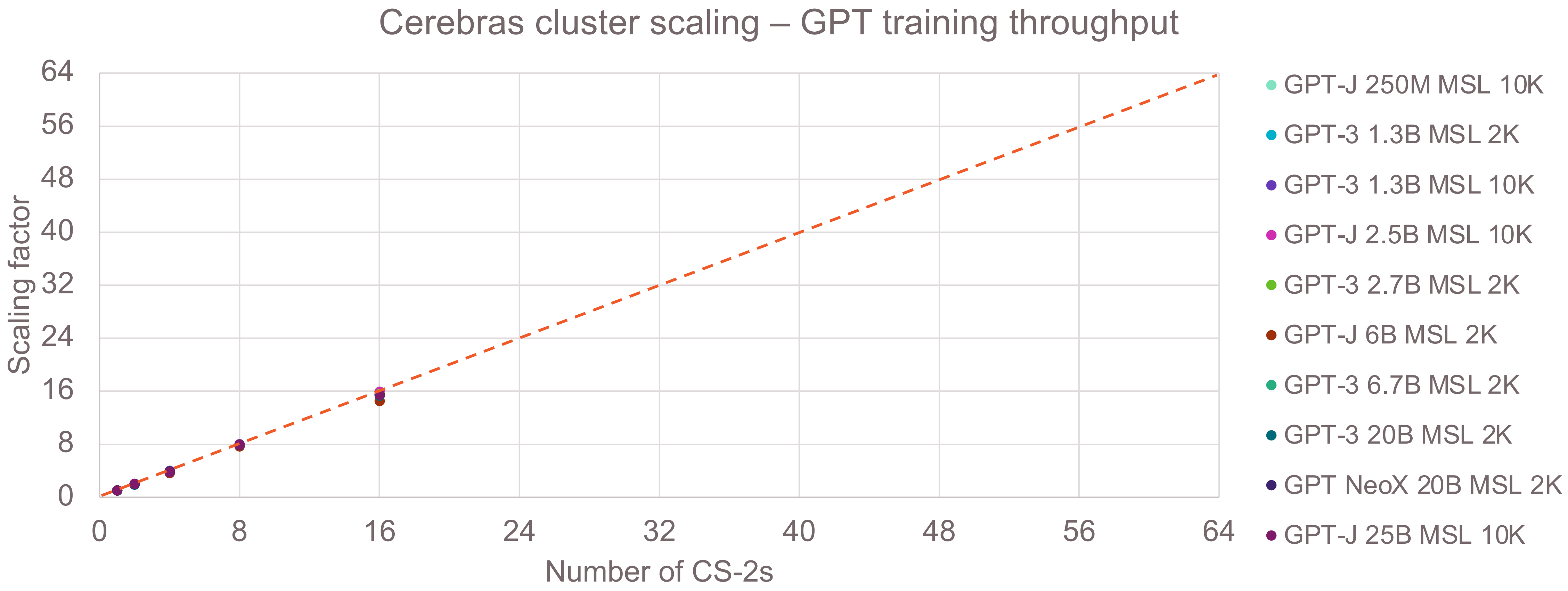


Figure. Measured training throughput scaling for 250M-20B GPT models over 1-16 CS-2 systems; projected scaling to 64 systems.

Break

Resume at 2:00pm CT

Software APIs

Model Porting, Layers API, and Dataloaders

Model Porting

Model Porting Options

Stage	Data Processing and Dataloaders	Define model architecture
(1) Getting started with Cerebras Ecosystem	Use data preprocessing from Cerebras Model Zoo	Use model implementation in Cerebras Model Zoo and customize hyperparameters in the params yaml file
(2) Use your own data and hyperparameters	Implement your own data preprocessing	
(3) Define your own model using Cerebras Model Zoo tools		Port your PyTorch or code using run function in Cerebras Model Zoo and Cerebras Model Zoo supported operations API
(4) Define your model using Cerebras PyTorch API		Have more flexibility porting your code with Cerebras PyTorch API

cerebras

Modify reference models in Cerebras Model Zoo

- If your primary goal is to use one of the Model Zoo models with minimal changes, we recommend start from the Cerebras Model Zoo and add changes you need.

- Hypothetical scenario:
  - We work with the PyTorch implementation of FC_MNIST in the Cerebras Model Zoo. We create a synthetic dataloader to evaluate performance of the network with respect to different input sizes and number of classes.

- To achieve this goal:
  - In data.py, we create a function called get_random_dataloader that creates random images and labels. We instrument the function to specify in the params.yaml file the number of examples, the batch size, the seed, the image_size and the num_classes of this dataset.

Modify reference models in Cerebras Model Zoo

- In data.py, we create a function called get_random_dataloader that creates random images and labels.

```
import torch
import numpy as np

def get_random_dataloader(input_params,shuffle,num_classes):
    num_examples = input_params.get("num_examples")
    batch_size = input_params.get("batch_size")
    seed = input_params.get("seed",1)
    image_size = input_params.get("image_size",[1,28,28])
    # Note: please cast the tensor to be of dtype `np.int32` when running on CS-2 sys
    np.random.seed(seed)
    image = np.random.random(size = [num_examples,]+image_size).astype(np.float32)
    label = np.random.randint(low =0, high = num_classes, size = num_examples).astype

    dataset = torch.utils.data.TensorDataset(
        torch.from_numpy(image),
        torch.from_numpy(label)
    )

    return torch.utils.data.DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        num_workers=input_params.get("num_workers", 0),
    )

def get_train_dataloader(params):
    return get_random_dataloader(
        params["train_input"],
        params["train_input"].get("shuffle"),
        params["model"].get("num_classes")
    )

def get_eval_dataloader(params):
    return get_random_dataloader(
        params["eval_input"],
        False,
        params["model"].get("num_classes")
    )
```

Modify reference models in Cerebras Model Zoo

- In model.py, we change the number of classes to a parameter in the params.yaml file.

```
class MNIST(nn.Module):
    def __init__(self, model_params):
        super().__init__()
        self.loss_fn = nn.NLLLoss()
        self.fc_layers = []
        input_size = model_params.get("input_size",784)
        num_classes = model_params.get("num_classes",10)
        ...
        self.last_layer = nn.Linear(input_size, num_classes)
        ...
```

Modify reference models in Cerebras Model Zoo

- In configs/params.yaml, we add the additional fields used in the dataloader and model definition.

```yaml
train_input:
    batch_size: 128
    drop_last_batch: True
    num_examples: 1000
    seed: 123
    image_size: [1,28,28]
    shuffle: True

eval_input:
    data_dir: "./data/mnist/val"
    batch_size: 128
    num_examples: 1000
    drop_last_batch: True
    seed: 1234
    image_size: [1,28,28]

model:
    name: "fc_mnist"
    mixed_precision: True
    input_size: 784 #1*28*28
    num_classes: 10
    ...
```

Create new models leveraging Cerebras run function

- If your primary goal is to develop new model and data preprocessing scripts, we suggest to start by leveraging the common backbone in Cerebras Model Zoo, the run function and file structure.

- The run function modularizes the model implementation, the data loaders, the hyperparameters and the execution. To use the run function you need:

  - Implementation that includes the following:
    - Model definition
    - Data loaders for training and evaluation

  - Params YAML file. This file will be used at runtime.

Create new models leveraging Cerebras run function

- Your code skeleton will approximately look like this.

Import

```
import os
import sys

import torch

#Append path to parent directory of Cerebras Model Zoo Repository
sys.path.append(os.path.join(os.path.dirname(__file__), ".."))
from Cerebras/modelzoo/tree/master/modelzoo/common/pytorch.run_utils import run
from Cerebras/modelzoo/tree/master/modelzoo/common/pytorch.PyTorchBaseModel import P
```

Define Model

1. Define the model architecture with torch.nn.Module
2. Then, wrap it by defining a PyTorchBaseModel.

```
#Step 1: Define Model
#Step 1.1 Define Module
class Model(torch.nn.Module):
    def __init__(self, params):
        ...
    def forward(inputs):
        ...
        return outputs
    ...

#Step 1.2 Define PyTorchBaseModel
class BaseModel(PyTorchBaseModel):
    def __init__(self, params, device = None)
        self.model = Model(params)
        self.loss_fn = ...
        ...
        super().__init__(params=params, model=self.model, device=device)
    def __call__(self, data):
        ...
        inputs, targets = data
        outputs = self.model(inputs)
        loss = self.loss_fn(outputs, targets)
        return loss
```

Define Dataloader

- requires a callable (class or function) that takes as input a dictionary of params returns a torch.utils.data.DataLoader.

```
#Step 2: Define dataloaders
def get_train_dataloader(params):
    ...
    loader = torch.utils.data.DataLoader(...)
    return loader

def get_eval_dataloader(params):
    ...
    loader = torch.utils.data.DataLoader(...)
    return loader
```

Execute script with run function

```
#Step 3: Setup run function
def main():
    run(BaseModel, get_train_dataloader, get_eval_dataloader)

if __name__ == '__main__':
    main()
```

cerebras

Create new models leveraging Cerebras run function

- Create params YAML file. The paremeters skeleton looks like this.

Section	Required	Notes
`runconfig`	Yes	Used by run to set up logging and execution. It expects fields: `max_steps`, `checkpoint_steps`, `log_steps`, `save_losses`.
`optimizer`	Yes	Used by `PyTorchBaseModel` to set up optimizer. It expects fields: `optimizer_type`, `learning_rate`, `loss_scaling_factor`.
`model`	No	By convention, it is used to customize the model architecture in `nn.Module`. Fields are tailored to needs inside the model.
`train_input`	No	By convention, it is used to customize train_data_fn. Fields are tailored to needs inside train_data_fn.
`eval_input`	No	By convention, it is used to customize eval_data_fn. Fields are tailored to needs inside eval_data_fn.

```yaml
train_input:
    ...

eval_input:
    ...

model:
    ...

optimizer:
    optimizer_type: ...
    learning_rate: ...
    loss_scaling_factor: ...

runconfig:
    max_steps: ...
    checkpoint_steps: ...
    log_steps: ...
    seed: ...
    save_losses: ...
```

Cerebras PyTorch API

- Historically, we had a number of PyTorch runners in ModelZoo that dictated the full run

- Pros & Cons:
  - Easy configuration via Model Zoo params.yaml
  - Tied to Model Zoo to run any PyTorch models on a Cerebras system
  - Limited generalizability and customizability

- New PyTorch API:
  - Leverages PyTorch 2.0
  - Make things as transparent as possible
  - Give users the flexibility to write their own training loops
  - Provide a more robust API that is less prone to errors when changes are made

Cerebras PyTorch API

- A simple skeleton of a full training script.

```python
import torch
import cerebras_pytorch.experimental as cstorch

backend = cstorch.backend("CSX", ...)

with backend.device:
    # user defined model
    model: torch.nn.Module = ...

compiled_model = cstorch.compile(model, backend)

loss_fn: torch.nn.Module = ...

optimizer: cstorch.optim.Optimizer = cstorch.optim.configure_optimizer(
    optimizer_type="...",
    params=model.parameters(),
    ...
)
lr_scheduler: cstorch.optim.lr_scheduler.LRScheduler = cstorch.optim.configure_lr_sch
    optimizer, learning_rate=...,
)

grad_scaler = None
if loss_scale != 0.0:
    grad_scaler = cstorch.amp.GradScaler(...)

@cstorch.checkpoint_closure
def save_checkpoint(step):
    checkpoint_file = f"checkpoint_{step}.mdl"

    state_dict = {
        "model": model.state_dict(),
        "optimizer": optimizer.state_dict(),
    }
    if lr_scheduler:
        state_dict["lr_scheduler"] = lr_scheduler.state_dict()
    if grad_scaler:
        state_dict["grad_scaler"] = grad_scaler.state_dict()

    state_dict["global_step"] = step

    cstorch.save(state_dict, checkpoint_file)
```

```python
global_step = 0

# Load checkpoint if provided
if checkpoint_path is not None:
    state_dict = cstorch.load(checkpoint_path)

    model.load_state_dict(state_dict["model"])
    optimizer.load_state_dict(state_dict["optimizer"])
    if lr_scheduler:
        lr_scheduler.load_state_dict(state_dict["lr_scheduler"])
    if grad_scaler:
        grad_scaler.load_state_dict(state_dict["grad_scaler"])

    global_step = state_dict.get("global_step", 0)

@cstorch.compile_step
def training_step(batch):
    inputs, targets = batch
    outputs = compiled_model(inputs)
    loss = loss_fn(outputs, targets)

    cstorch.amp.optimizer_step(
        loss, optimizer, grad_scaler, max_gradient_norm=1.0
    )

    return loss
```

```python
@cstorch.step_closure
def post_training_step(loss: torch.Tensor):
    print("Loss: ", loss.item())

dataloader = cstorch.utils.data.DataLoader(
    train_dataloader_fn,
    ...
)
executor = cstorch.utils.data.DataExecutor(
    dataloader=dataloader,
    num_steps=1000,
    chekpoint_steps=100,
    cs_config=cstorch.utils.CSConfig(...),
)

for i, batch in enumerate(executor):
    loss = training_step(dataloader)

    post_training_step(loss)

    # Always call save_checkpoint, but is only truly
    # run every 100 steps
    save_checkpoint(i)
```

CSTorch Layers API

Running on Cerebras Wafer-Scale Cluster using cstorch API

1. Import cstorch package

Import

```python
import torch
import cerebras_pytorch as cstorch

model: torch.nn.Module = Model()
model = cstorch.compile(model, "CSX")
loss_fn = torch.nn.NLLLoss()
optimizer = cstorch.optim.SGD(model.parameters(), lr=0.01)
dataloader = cstorch.utils.data.DataLoader(get_train_dataloader)
executor = cstorch.utils.data.DataExecutor(
    dataloader
)

@cstorch.trace
def training_step(inputs, targets):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_fn(outputs, targets)
    loss.backward()
    optimizer.step()
    return loss

@cstorch.step_closure
def print_loss(loss: torch.Tensor):
    print(f"Loss: {loss.item()}")

for inputs, targets in executor:
    loss = training_step(inputs, targets)
    print_loss(loss)
```

Running on Cerebras Wafer-Scale Cluster using cstorch API

1. Import cstorch package
2. Define the model
   - Model is defined as if running on a single device
   - Use familiar torch API with some drop-in replacements
   - Wrap dataloader in a cstorch data executor

Define Model

```python
import torch
import cerebras_pytorch as cstorch

model: torch.nn.Module = Model()
model = cstorch.compile(model, "CSX")
loss_fn = torch.nn.NLLLoss()
optimizer = cstorch.optim.SGD(model.parameters(), lr=0.01)
dataloader = cstorch.utils.data.DataLoader(get_train_dataloader)
executor = cstorch.utils.data.DataExecutor(
    dataloader
)

@cstorch.trace
def training_step(inputs, targets):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_fn(outputs, targets)
    loss.backward()
    optimizer.step()
    return loss

@cstorch.step_closure
def print_loss(loss: torch.Tensor):
    print(f"Loss: {loss.item()}")

for inputs, targets in executor:
    loss = training_step(inputs, targets)
    print_loss(loss)
```

Running on Cerebras Wafer-Scale Cluster using cstorch API

1. Import cstorch package
2. Define the model
   - Model is defined as if running on a single device
   - Use familiar torch API with some drop-in replacements
   - Wrap dataloader in a cstorch data executor
3. Create the training loop method
   - Nothing novel here, except the decorator

Define Training Loop

```python
import torch
import cerebras_pytorch as cstorch

model: torch.nn.Module = Model()
model = cstorch.compile(model, "CSX")
loss_fn = torch.nn.NLLLoss()
optimizer = cstorch.optim.SGD(model.parameters(), lr=0.01)
dataloader = cstorch.utils.data.DataLoader(get_train_dataloader)
executor = cstorch.utils.data.DataExecutor(
    dataloader
)

@cstorch.trace
def training_step(inputs, targets):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_fn(outputs, targets)
    loss.backward()
    optimizer.step()
    return loss

@cstorch.step_closure
def print_loss(loss: torch.Tensor):
    print(f"Loss: {loss.item()}")

for inputs, targets in executor:
    loss = training_step(inputs, targets)
    print_loss(loss)
```

Running on Cerebras Wafer-Scale Cluster using cstorch API

1. Import cstorch package
2. Define the model
   - Model is defined as if running on a single device
   - Use familiar torch API with some drop-in replacements
   - Wrap dataloader in a cstorch data executor
3. Create the training loop method
   - Nothing novel here, except the decorator
4. Run the training loop
   - Under the hood, compiles the model on the first step and starts asynchronous execution
   - Outputs (losses) are retrieved as available

Train

```python
import torch
import cerebras_pytorch as cstorch

model: torch.nn.Module = Model()
model = cstorch.compile(model, "CSX")
loss_fn = torch.nn.NLLLoss()
optimizer = cstorch.optim.SGD(model.parameters(), lr=0.01)
dataloader = cstorch.utils.data.DataLoader(get_train_dataloader)
executor = cstorch.utils.data.DataExecutor(
    dataloader
)

@cstorch.trace
def training_step(inputs, targets):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_fn(outputs, targets)
    loss.backward()
    optimizer.step()
    return loss

@cstorch.step_closure
def print_loss(loss: torch.Tensor):
    print(f"Loss: {loss.item()}")

for inputs, targets in executor:
    loss = training_step(inputs, targets)
    print_loss(loss)
```

Running on Cerebras Wafer-Scale Cluster using cstorch API

- Scale out to multiple CS-2s with a single configuration change
- Near-linear scaling is achieved automatically
- No model change
- No change to the training loop
- No change to effective batch size

Scale out

```
import torch
import cerebras_pytorch as cstorch

model: torch.nn.Module = Model()
model = cstorch.compile(model, "CSX")
loss_fn = torch.nn.NLLLoss()
optimizer = cstorch.optim.SGD(model.parameters(), lr=0.01)
dataloader = cstorch.utils.data.DataLoader(get_train_dataloader)
executor = cstorch.utils.data.DataExecutor(
    dataloader, cs_config=cstorch.utils.CSConfig(num_csx=16)
)

@cstorch.trace
def training_step(inputs, targets):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_fn(outputs, targets)
    loss.backward()
    optimizer.step()
    return loss

@cstorch.step_closure
def print_loss(loss: torch.Tensor):
    print(f"Loss: {loss.item()}")

for inputs, targets in executor:
    loss = training_step(inputs, targets)
    print_loss(loss)
```

Sparsity Code Example

- Dynamic sparsity motivates an “optimizer”
  - Updates the sparsity pattern on a cadence
  - Aligns sparsity of params, gradients, and optionally optimizer state
- Static sparsity is a special case of not updating
- Similar to `torch.nn.prune`, but fully traced for AoT compile
- The torch level representation uses masks
  - Compiler automatically transforms to Compressed Sparse Row (CSR)

Setup

Apply

Update

```python
import torch
import cerebras_pytorch as cstorch

model: torch.nn.Module = Model()
model = cstorch.compile(model, "CSX")
loss_fn = torch.nn.NLLLoss()
optimizer = cstorch.optim.SGD(model.parameters(), lr=0.01)
sparsity_optimizer = cstorch.sparse.RigLSparsityOptimizer(
    model.named_parameters(), sparsity=0.9, schedule=1000
)
dataloader = cstorch.utils.data.DataLoader(get_train_dataloader)
executor = cstorch.utils.data.DataExecutor(dataloader)

@cstorch.trace
def training_step(inputs, targets):
    optimizer.zero_grad()
    sparsity_optimizer.apply_sparsity()
    outputs = model(inputs)
    loss = loss_fn(outputs, targets)
    loss.backward()
    optimizer.step()
    sparsity_optimizer.step()
    return loss

@cstorch.step_closure
def print_loss(loss: torch.Tensor):
    print(f"Loss: {loss.item()}")

for inputs, targets in executor:
    loss = training_step(inputs, targets)
    print_loss(loss)
```

Dataloading

Offline Huggingface Data Conversion

- If you have a functioning Huggingface-style dataset, it is most efficient to convert it into HDF5 format in advance.

- Modelzoo leverages a utility function, *convert_dataset_to_HDF5(),* for this.

- After your dataset is in HDF5 form, simply specify an HDF5DataProcessor to leverage in your model config.

```
dataset, data_collator = HuggingFace_BookCorpus(
    split="train", num_workers=8, sequence_length=2048
)
convert_dataset_to_HDF5(
    dataset=dataset,
    data_collator=data_collator,
    output_dir="./bookcorpus_hdf5_dataset/",
)
```

```
train_input:
    data_dir: <path to samples saved into h5 files>
    data_processor: "GptHDF5DataProcessor"
    ...
eval_input:
    data_dir: <path to samples saved into h5 files>
    data_processor: "GptHDF5DataProcessor"
    ...
```

Implementing Custom Dataloaders

- Because all data loading occurs on CPU devices in the Cerebras appliance, **we only need to make a couple of tweaks to existing Pytorch dataloaders**.
- First, we use the modelzoo helper getters *num_tasks()* and *task_id*() for efficient sharding.
- Second, we set *drop_last*=*True* to ensure batch sizes are consistent during training.

```python
import torch
import numpy as np

from tokenizers import Tokenizer
from modelzoo.transformers.pytorch.input_utils import num_tasks, task_id

class ShardedTextDataset(torch.utils.data.Dataset):
    def __init__(self, input_file, sequence_length):
       self.sequence_length = sequence_length
        with open(input_file, "r") as f:
            text = f.read()
        tokenizer = Tokenizer.from_pretrained("gpt2")
        self.data = np.array(tokenizer.encode(text).ids, dtype=np.int32)
        self.data = [
            self.data[i : i + self.sequence_length + 1]
            for i in range(
                0, len(self.data) - self.sequence_length - 1, self.sequence_length
            )
        ]
        self.data = self.data[task_id()::num_tasks()]

    def __getitem__(self, i):
        x = self.data[i]
        return {
            "input_ids": x[:-1],
            "attention_mask": np.ones(self.sequence_length, dtype=np.int32),
            "labels": x[1:],
        }

    def __len__(self):
        return (len(self.data) - 1) // self.sequence_length


dataloader = torch.utils.data.DataLoader(
    ShardedTextDataset("/path/to/data.txt", 128),
    batch_size=16,
    shuffle=True,
    drop_last=True,
)
```

Huggingface - CS-2 Porting

Framework Conversion Options

Custom or Non-Modelzoo HF Model

- Need to use the cstorch Layers API to re-implement the model.

- If it's *similar* to a model in the Modelzoo, we can tweak an existing model implementation.
  - *gpt_model.py, bert_model.py,* etc

- Otherwise, use supported ops and existing models as references to modify your Pytorch implementation.

Framework Conversion Options

Custom or Non-Modelzoo HF Model

- Need to use the cstorch Layers API to re-implement the model.
- If it's *similar* to a model in the Modelzoo, we can tweak an existing model implementation.
  - *gpt_model.py, bert_model.py,* etc
- Otherwise, use supported ops and existing models as references to modify your Pytorch implementation.

Modelzoo-Supported HF Model

- Life is easy!
- Use Cerebras' checkpoint conversion utility to convert from HF to CS-2 format…or between Modelzoo versions!
- Then fine-tune or eval like any Modelzoo model.
- Convert back to HF for evaluation or portability as needed!

Supported Modelzoo Implementations

Bert	Bert-sequence-classifier	Bert-token-classifier	Bert-summarization	Bert-q&a
Bloom	Bloom-headless	**Btlm**	Btlm-headless	**codegen**
Codegen-headless	**Code-llama**	Code-llama-headless	**Dpr**	**Falcon**
Falcon-headless	**Flan-ul2**	**Gpt2**	Gpt2-headless	Gpt2 w/ muP
Gptj	Gptj-headless	**Gpt-neox**	Gpt-neox-headless	**Jais**
Llama	Llama-headless	**LlamaV2**	LlamaV2-headless	**Llava**
Mpt	Mpt-headless	**Mistral**	Mistral-headless	**Octocoder**
Octocoder-headless	**Roberta**	**Santacoder**	Santacoder-headless	**Sqlcoder**
Sqlcoder-headless	**T5**	**Transformer**	**Ul2**	**Wizardcoder**
Wizardcoder-headless	**Wizardlm**	Wizardlm-headless		

Checkpoint Conversion: GPT-J 6B

- Start by downloading the huggingface checkpoint of interest (if needed).

```
$ mkdir ~/my_checkpoints

$ wget -P opensource_checkpoints https://huggingface.co/EleutherAI/gpt-j-6B/raw/main/config.json
~/my_checkpoints

$ wget -P opensource_checkpoints https://huggingface.co/EleutherAI/gpt-j-6B/resolve/main/pytorch_model.bin
~/my_checkpoints
```

Checkpoint Conversion: GPT-J 6B

- Start by downloading the huggingface checkpoint of interest (if needed).

```
$ mkdir ~/my_checkpoints

$ wget -P opensource_checkpoints https://huggingface.co/EleutherAI/gpt-j-6B/raw/main/config.json
~/my_checkpoints

$ wget -P opensource_checkpoints https://huggingface.co/EleutherAI/gpt-j-6B/resolve/main/pytorch_model.bin
~/my_checkpoints
```

- Specify the model type, source and target frameworks, then convert!

```
$ python ~/modelzoo/src/cerebras/modelzoo/tools/convert_checkpoint.py \
        convert \
        --model gptj \
        --src-fmt hf \
        --tgt-fmt cs-2.2 \
        --output-dir ~/my_checkpoints/ \
        --config ~/my_checkpoints/config.json \
        ~/my_checkpoints/pytorch_model.bin
```

Job monitoring and profiling

How to monitor the results with TensorBoard

1. Activate Python environment (if not already activated)

```
$ source /venv/venv_cerebras_r2.0.2/bin/activate
```

2. Launch TensorBoard choosing the model directory of the run

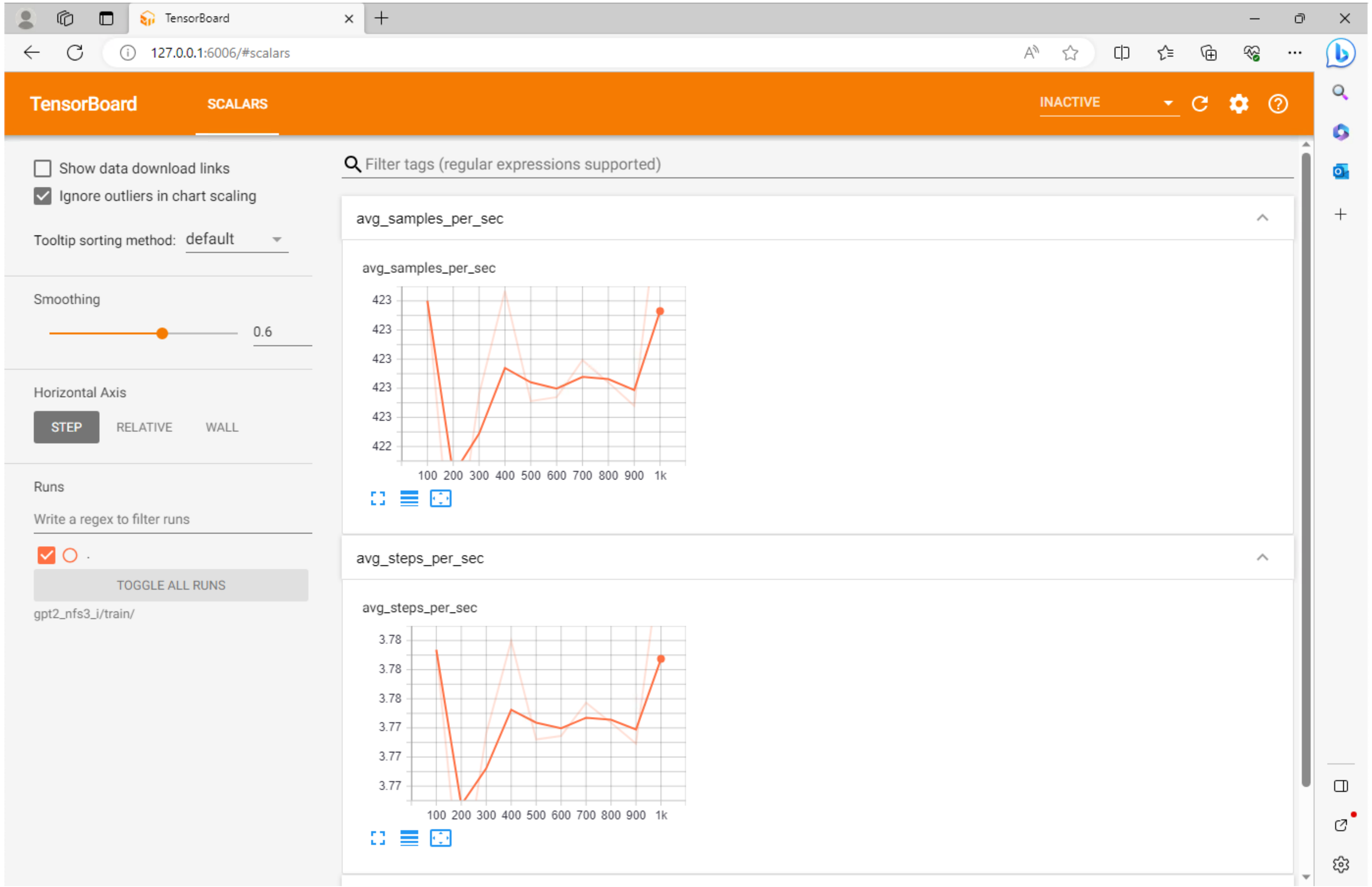
```
$ tensorboard --logdir_spec={your_modeldir}/train/ --bind_all --port=6006
```

3. ssh into the user node with port binding from your local machine

```
$ ssh -N -L localhost:6006:localhost:6006  {your_username}@10.72.0.27
```

4. Open `127.0.0.1:6006` from your local browser

Example output in TensorBoard



How to monitor the queue

1. Use the Cerebras tool `csctl` to query the status of the queue. The job phase is one of QUEUED, RUNNING, SUCCEDED, FAILED.

```
$ csctl get jobs

NAME                 AGE  PHASE      SYSTEMS                    USER    LABELS
wsjob-000000000001   18h  RUNNING    CS2-01-01                  user2   custom_label_2
```

2. Every job is recorded using a jobID and it is printed in the training output.
3. To only display all the jobs running including historical ones, use

```
$ csctl get jobs -a

NAME                 AGE  PHASE      SYSTEMS                    USER    LABELS
wsjob-000000000000   43h  SUCCEEDED  CS2-01-01                  user1   custom_label_1
wsjob-000000000001   18h  RUNNING    CS2-01-01                  user2   custom_label_2
```

4. To cancel jobs

```
$ csctl cancel job wsjob-000000000001
```

5. Detailed documentation
   https://docs.cerebras.net/en/latest/wsc/getting-started/csctl.html

How to profile your code with CSTorch Profiler

Capabilities

1. Highlights 10 most time-consuming PyTorch modules
2. Outputs a JSON file format compatible with Google Chrome's tracing tool.

Limitations

1. Currently, does not display details of PyTorch modules that get executed on the host servers (only works on wafer ops).
2. Currently, only profiles `train` mode.

We will share detailed documentation after the presentation!

How to profile your code with CSTorch Profiler

1. Clone the [Cerebras Model Zoo repository]()
2. Navigate to the Cerebras Model Zoo model config that you want to run.

```
cd modelzoo/src/cerebras/Cerebras Model Zoo/models/nlp/gpt2/config
```

3. In the “runconfig” , do the following to specify the range of steps which needs to be profiled:

```
runconfig:
    .....
    op_profiler_config:
        start_step: 1
        end_step: 3
    .....
```

4. As you can see for the above example, step number 1, 2 and 3 would be profiled.
5. Start the training as usual.

Example output in console

```
+----+----------------------------------------------+--------------------+--------------+
|    | PyTorch MODULE NAME                          |   CSX TIME (in ms) |   % CSX time |
|----+----------------------------------------------+--------------------+--------------|
|  0 | loss_fn.fwd                                  |             164816 |    70.8502   |
|  1 | model.transformer_decoder.layers.0.self_attn.fwd |           5279 |     2.26931  |
|  2 | model.embedding_layer.word_embeddings.fwd    |               5241 |     2.25297  |
|  3 | model.fwd                                    |               4897 |     2.1051   |
|  4 | model.lm_head.fwd                            |                953 |     0.40967  |
|  5 | model.transformer_decoder.layers.18.self_attn.fwd |           870 |     0.373991 |
|  6 | model.transformer_decoder.layers.19.self_attn.fwd |           868 |     0.373131 |
|  7 | CrossEntropyLoss_1.fwd                       |                792 |     0.340461 |
|  8 | model.transformer_decoder.layers.5.self_attn.fwd |            776 |     0.333583 |
|  9 | model.transformer_decoder.layers.8.self_attn.fwd |            709 |     0.304781 |
+----+----------------------------------------------+--------------------+--------------+
```

Break

Resume at 3:15 pm CT

Hands-on session for training @ ALCF

Bill Arnold
Argonne Leadership Computing Facility
arnoldw@anl.gov

How to contact Cerebras?

- Email us at developer@cerebras.net
- Sign up for our monthly newsletter at info.cerebras.net/subscribe
- Join our Discord at discord.gg/hZp5MUyw
- Join our Discourse at discourse.cerebras.net/
- LinkedIn - linkedin.com/company/cerebras-systems/
- Twitter - twitter.com/CerebrasSystems

Talk to researchers and our ML/SDK Engineers here!