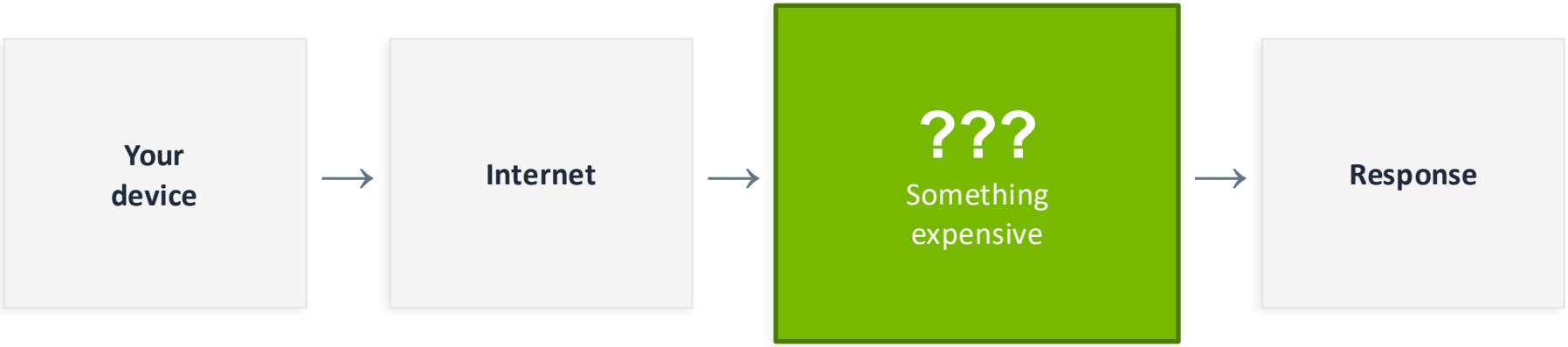
What Really Happens When You Call an LLM API?

Model Parallelism: Building and Deploying Large Neural Networks

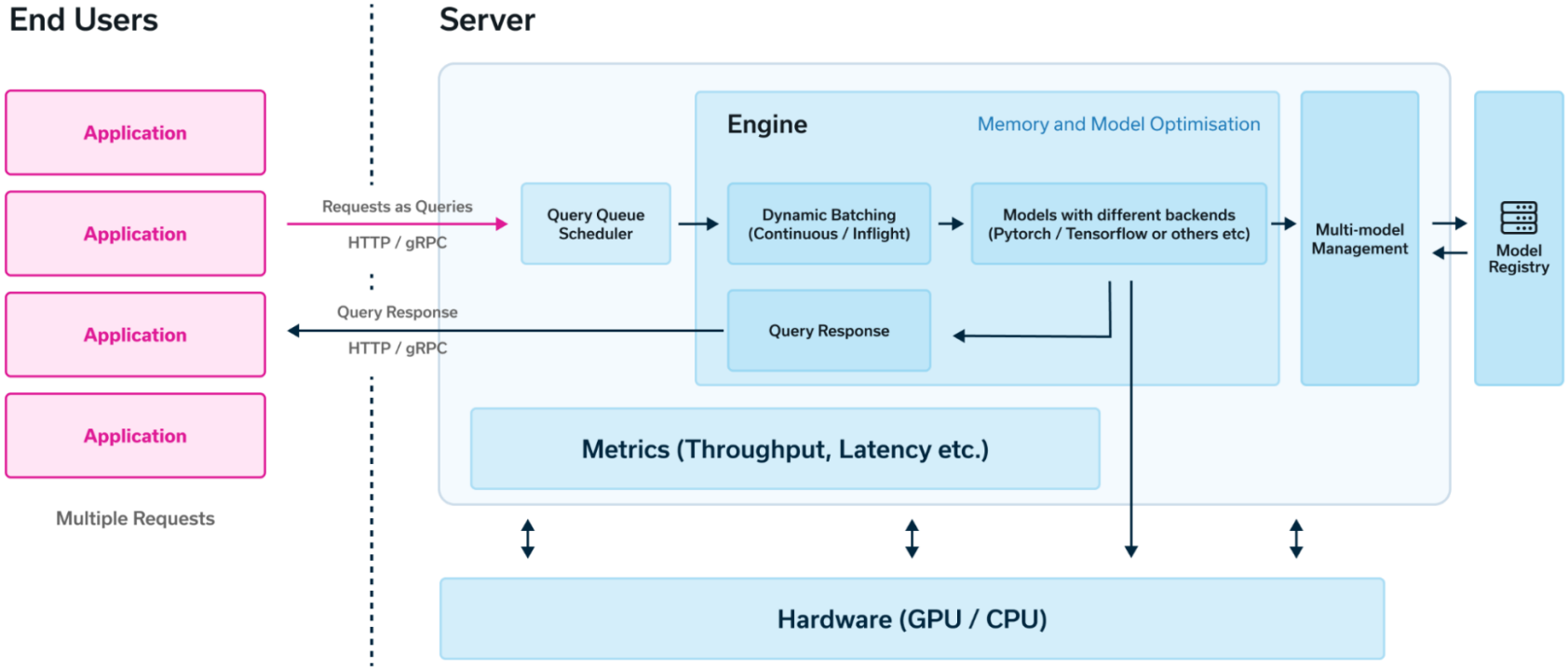
Pamekitti Puktalae, MTS @ AltoTech Global
Postgraduate Researcher in Semantic Technology in Smart Buildings, UCL

Agentic Coding Fitness 2026, Week 3

You type a message. You get a response. What's in between?



You type a message. You get a response. What's in between?



https://www.analyticsvidhya.com/blog/2024/07/efficient-llm-deployment/

2

What Is an LLM, Physically?

From a single multiply-add to 175 billion learned numbers stored in a file

Transformer

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

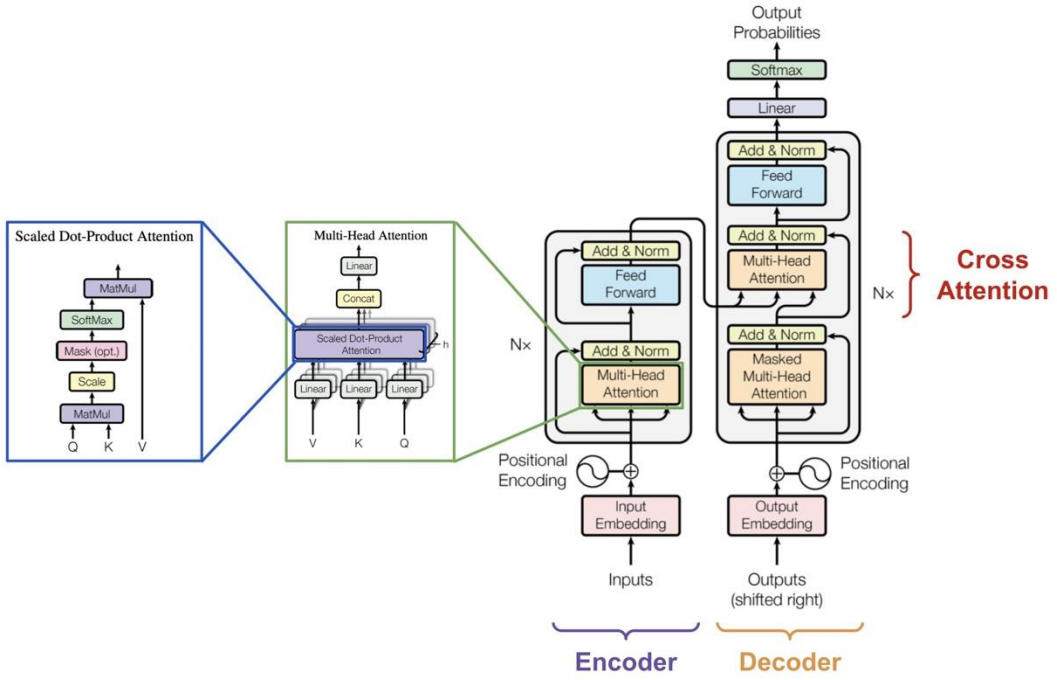
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

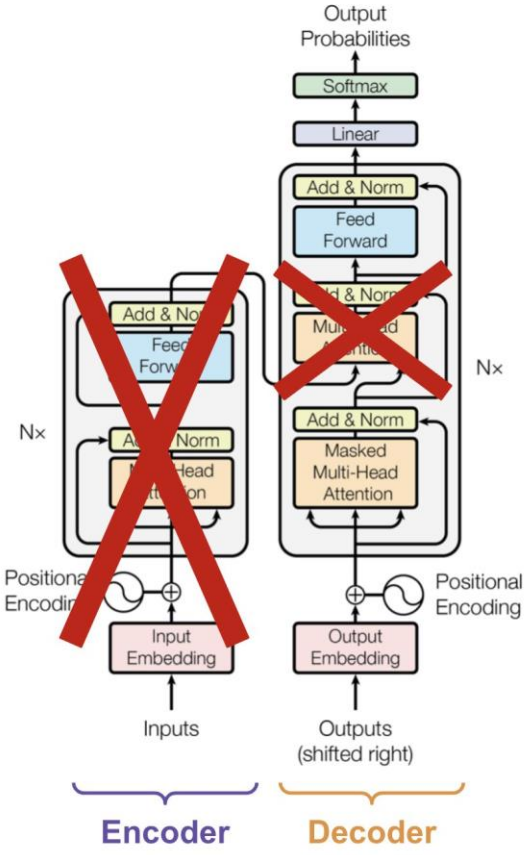
Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

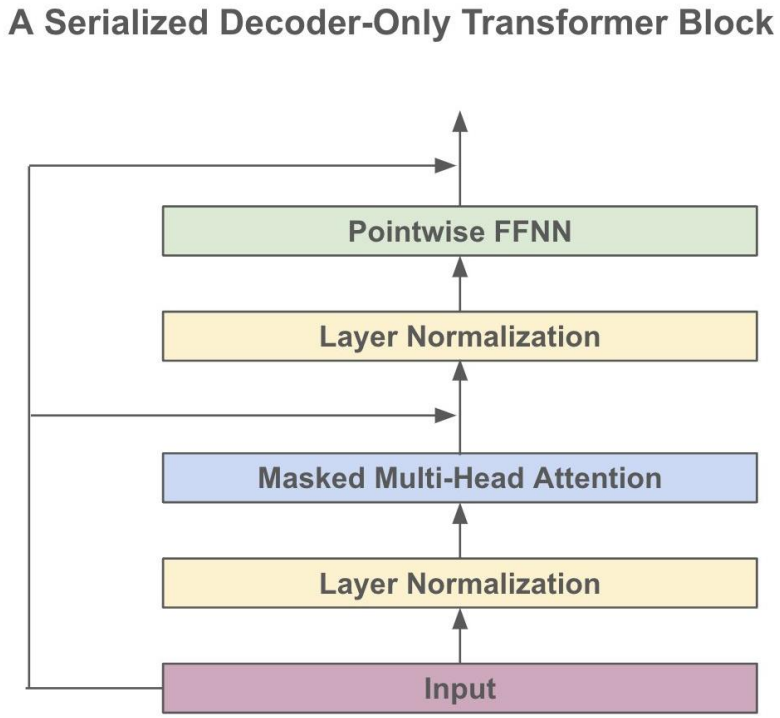


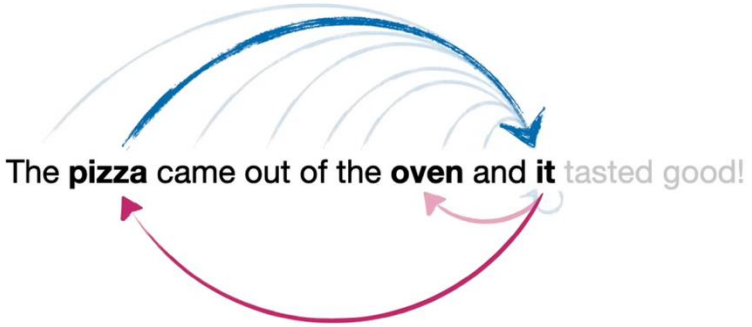
Decoder-only Transformer

Everything related to the encoder is removed!

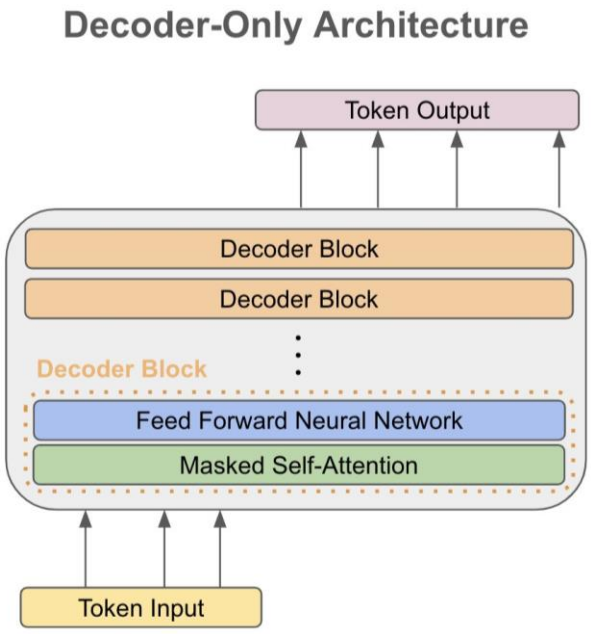


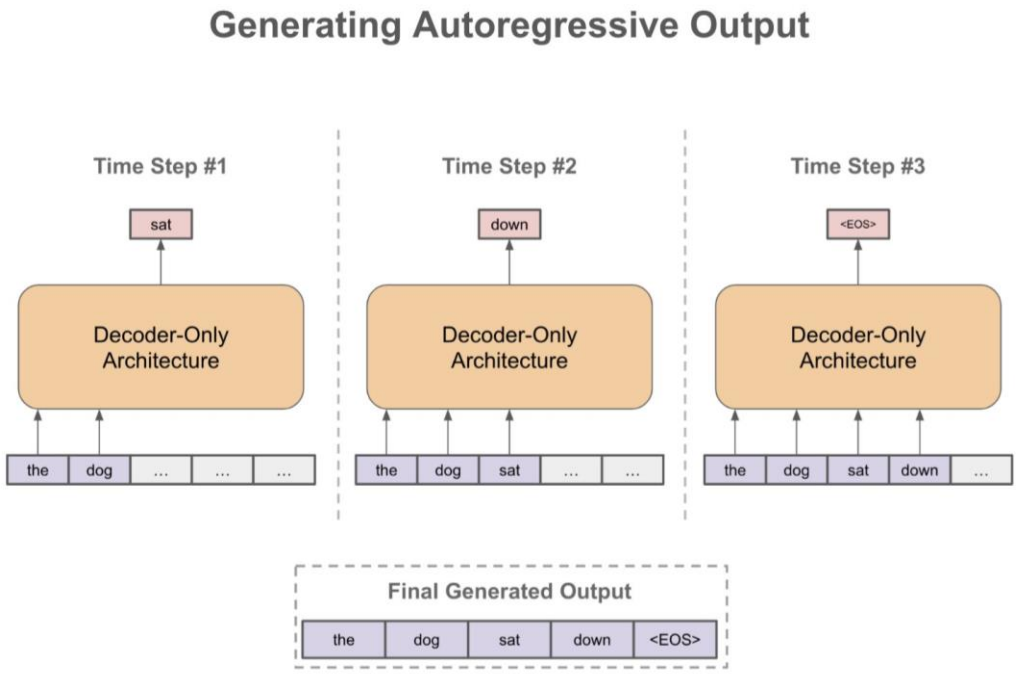
Decoder-only Transformer



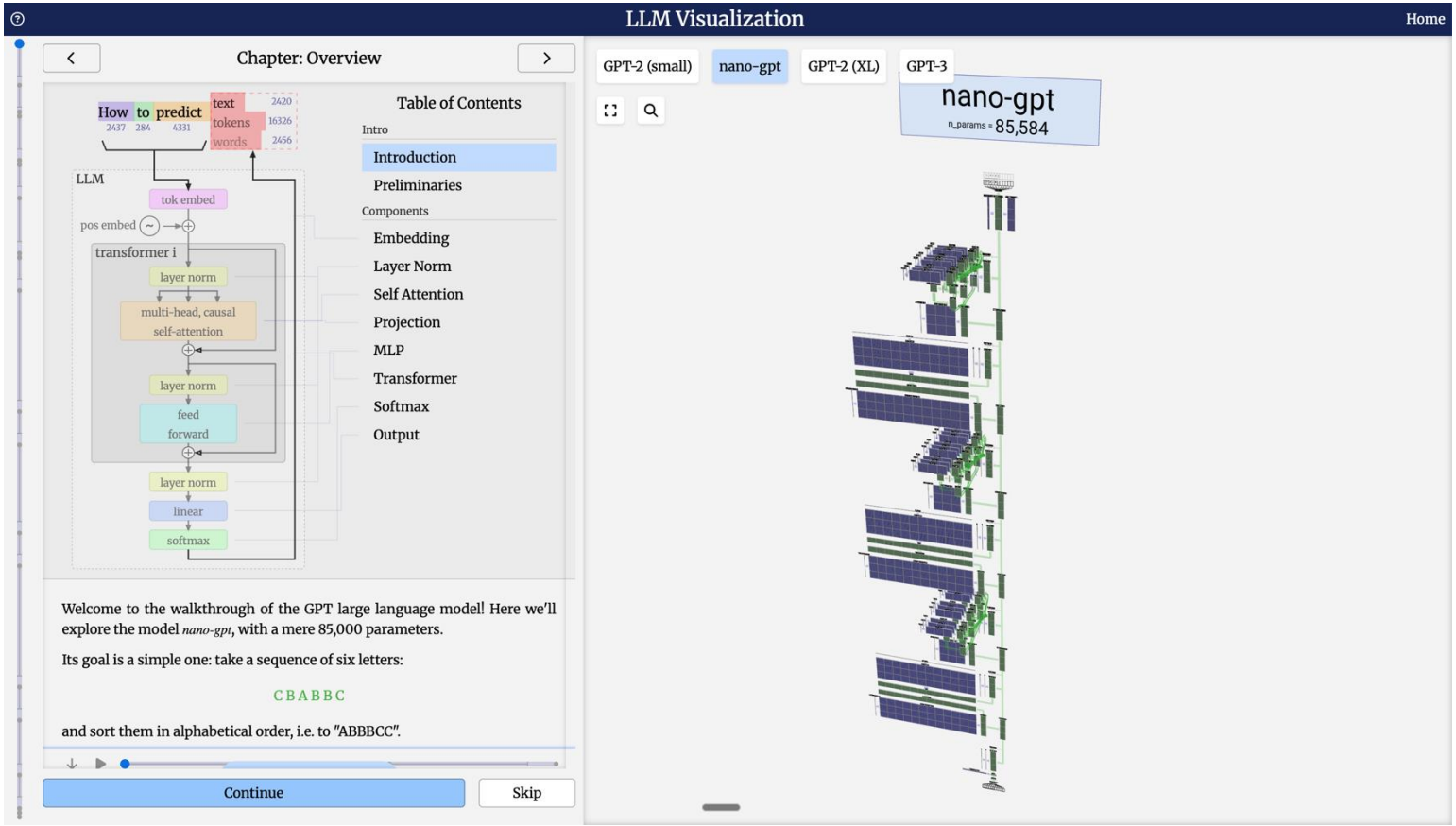


Decoder-only Transformer

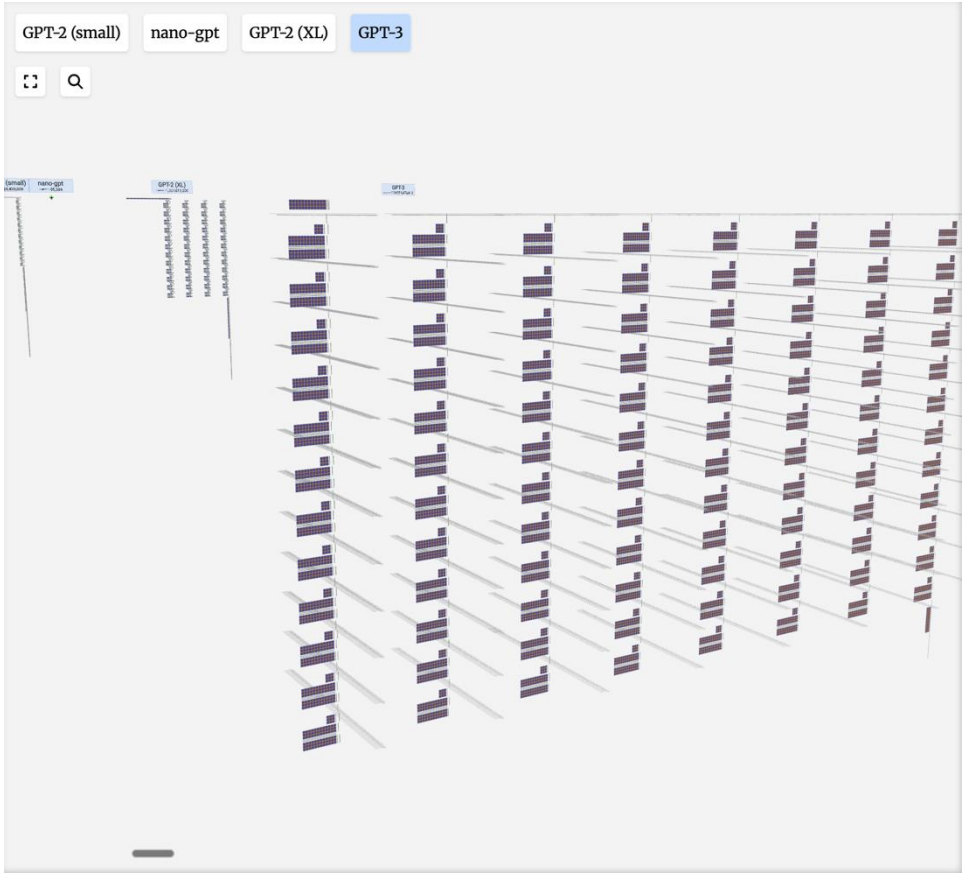


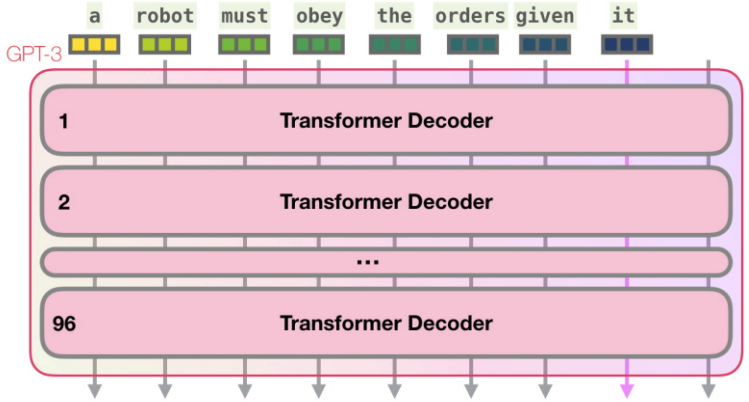


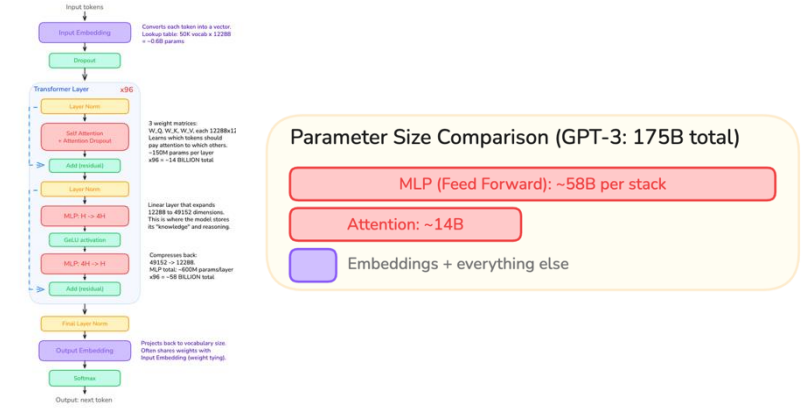
LLM Visualization: https://bbycroft.net/llm



GPT-3: https://bbycroft.net/llm







Before 175 billion parameters... let's start with 2.

The simplest possible neural network:

- 1 inputs (x1, x2)
- 1 weights (w1, w2)
- 1 bias (b)
- 1 output (y)

2 total weights. That is the entire model.

The Formula

$$y = (x \times w) + b$$

Worked example:

```
x=4.0 w=0.5 b=0.1

y = (4.0×0.5) + 0.1
y = 2.0 + 0.1
y = 2.1
```

This is the atom of every neural network. Everything else is doing this millions of times.

A weight is just a number. But the RIGHT number.

0.342

This is a weight. A number stored in a file.
175B parameters = 175 billion of these numbers.

Before training (random):
`0.891, -0.234, 0.557...`
"The cat jumped purple banana twelve"

After training (learned):
`0.342, -0.118, 0.773...`
"The cat jumped onto the table"

Training = adjusting each weight, tiny bit at a time, to reduce errors:

1. Start with random weights (model is useless)
2. Feed training example: "The cat sat on the ___" → correct answer: "mat"
3. Model predicts "banana" -- measure error (loss function)
4. Backpropagation: figure out which weights caused the error
5. Gradient descent: adjust each weight slightly to reduce error
6. Repeat billions of times. Weights converge to useful values.

One neuron does one multiply-add. A layer does thousands.

Single Neuron (from 2.1)

2 inputs → 3 weights → 1 output

```
y = (x1×w1) + (x2×w2) + b
```

A Layer

2 inputs → 12 weights → 4 outputs

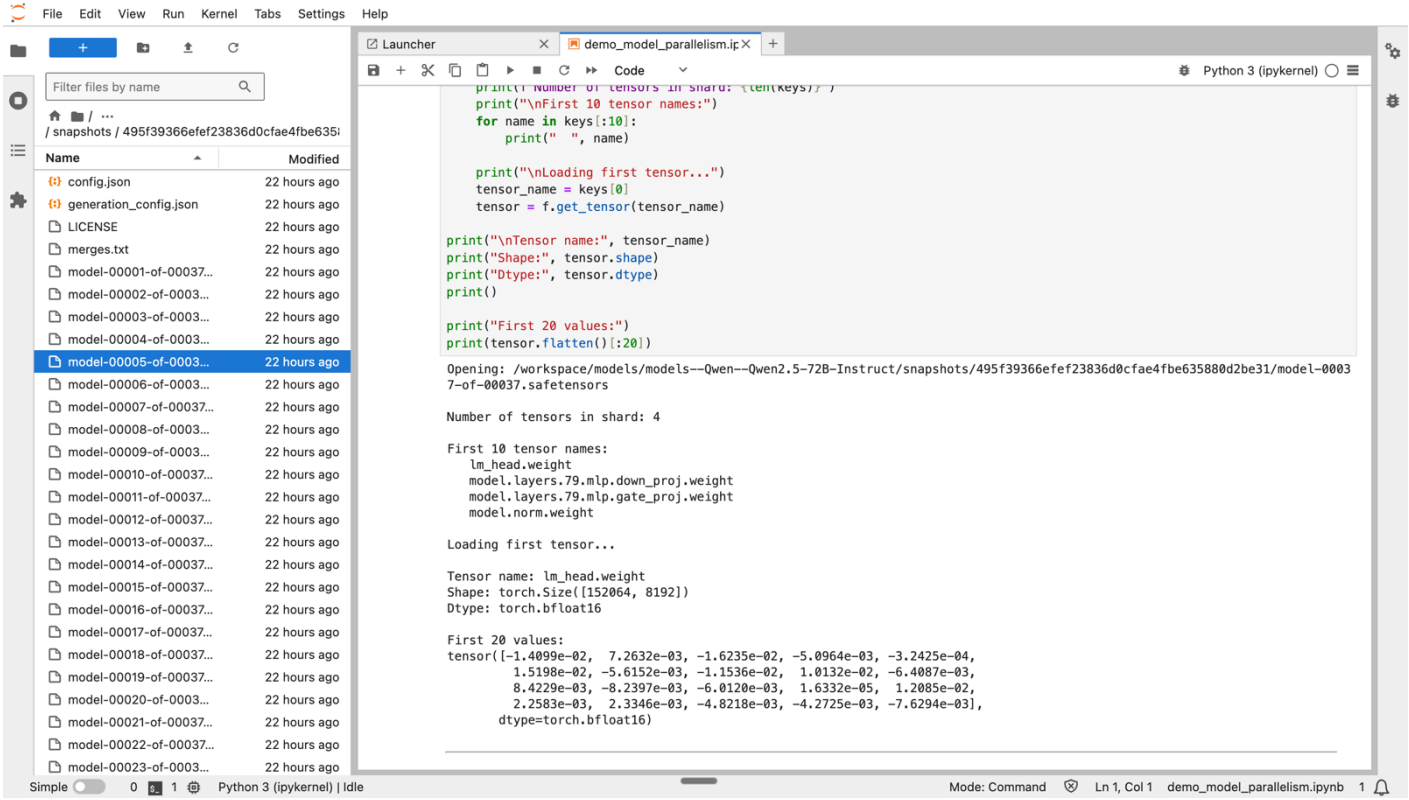
```
output = input_vector × weight_matrix
```

Matrix form:

```
Input:  [4.0, 2.0]
Weight: [0.5  0.1  -0.2  0.8]    Bias: [0.1, -0.2, 0.3, 0.0]
        [-0.3 0.7   0.4 -0.1]
Output: [1.5, 1.6, 0.3, 3.0]
```

[INSERT: Diagram -- two input circles connected to four output circles with a grid of arrows, showing a matrix multiply]

An LLM is just files containing numbers arranged in special way!



Same model, four sizes

Precision	Bytes/weight	175B Size	Fits on...	Quality
FP32	4 bytes	700 GB	10x H100 (80 GB each)	Perfect fidelity
FP16 / BF16	2 bytes	350 GB	5x H100	Negligible loss
INT8	1 byte	175 GB	3x H100	Very slight loss
INT4	0.5 bytes	**88 GB**	**2x RTX 4090 (24 GB + offload)**	Noticeable on reasoning

Inference (per token): ~350B FLOPs/token **Training (per token):** ~1+ T FLOPs/token

Photo analogy: FP32 = RAW photo (huge, perfect) | FP16 = High JPEG (half size, can't tell) | INT8 = Medium JPEG | INT4 = Thumbnail (fine for most uses)

3

Model Sizes & GPU Math

Why did models get so big, and what does it physically mean to fit a model on a GPU?

DRAMATIC INCREASE IN MODEL SIZES

The Trend Continues

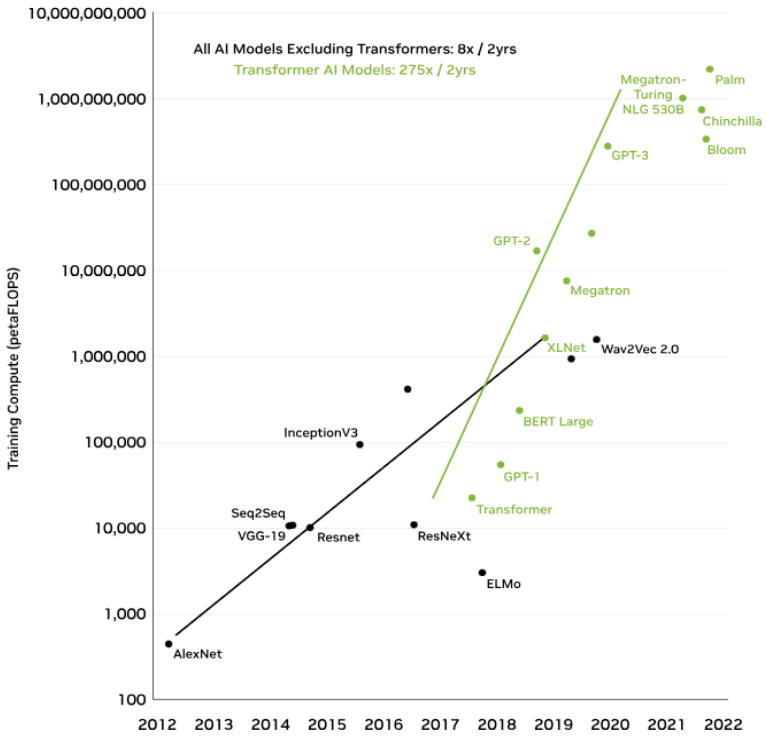


Figure 3. Compute required for training transformer models.

- 2017 paper – Attention is All you Need introduced the transformer architecture
  - Removed the sequential processing dependency of RNNs
  - Enabled language models to be trained with parallelism
  - Beginning of the Large Model Revolution
- BERT from Google in 2019
  - Trained on large corpus of text data and applied embeddings to other tasks
  - Popularized finetuning of models to get task specific embeddings
- Megatron-LM (NVIDIA) 2021 – showed how to scale training efficiently to thousands of GPUs with model parallelism which led to the explosion of larger models
- GPT-3 was the precursor to the popular chat-gpt which made LLMs commercially viable

Why did we go from 117 million to 744 billion parameters?

The Discovery of Scaling Laws

Kaplan et al. (2020) discovered a law of nature:

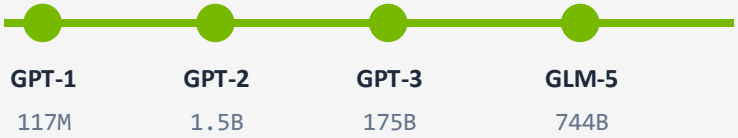
If you make the model bigger, the errors go down on a predictable curve.

Power law relationship:

L(N) ~ N^(-0.076)

Every 10x increase in parameters gives the same proportional improvement in error.

[INSERT: Simplified Kaplan et al. scaling curve -- log-log plot of parameters vs. test loss with GPT-1, GPT-2, GPT-3, GLM-5 as labeled dots on a downward line]



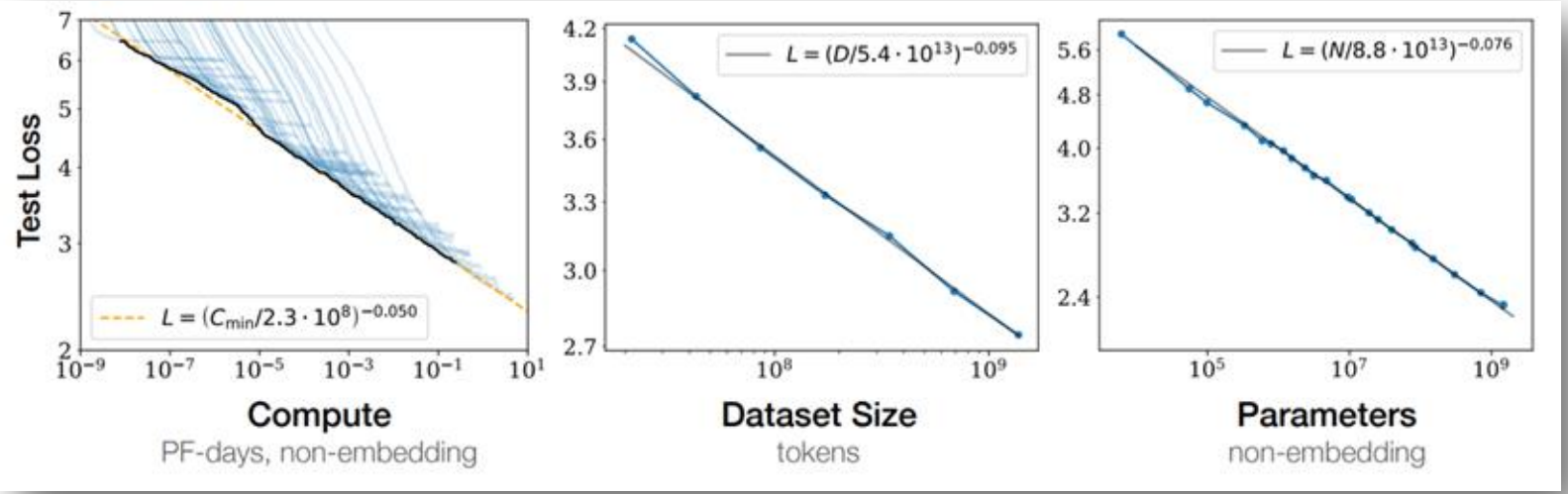
Three Scaling Knobs:

More Parameters + More Data + More Compute = Lower Error

Once you know the curve, the business decision is simple: **spend 10x more, get a predictable improvement.**

EMPIRICAL EVIDENCE

The Scaling Laws in NLP

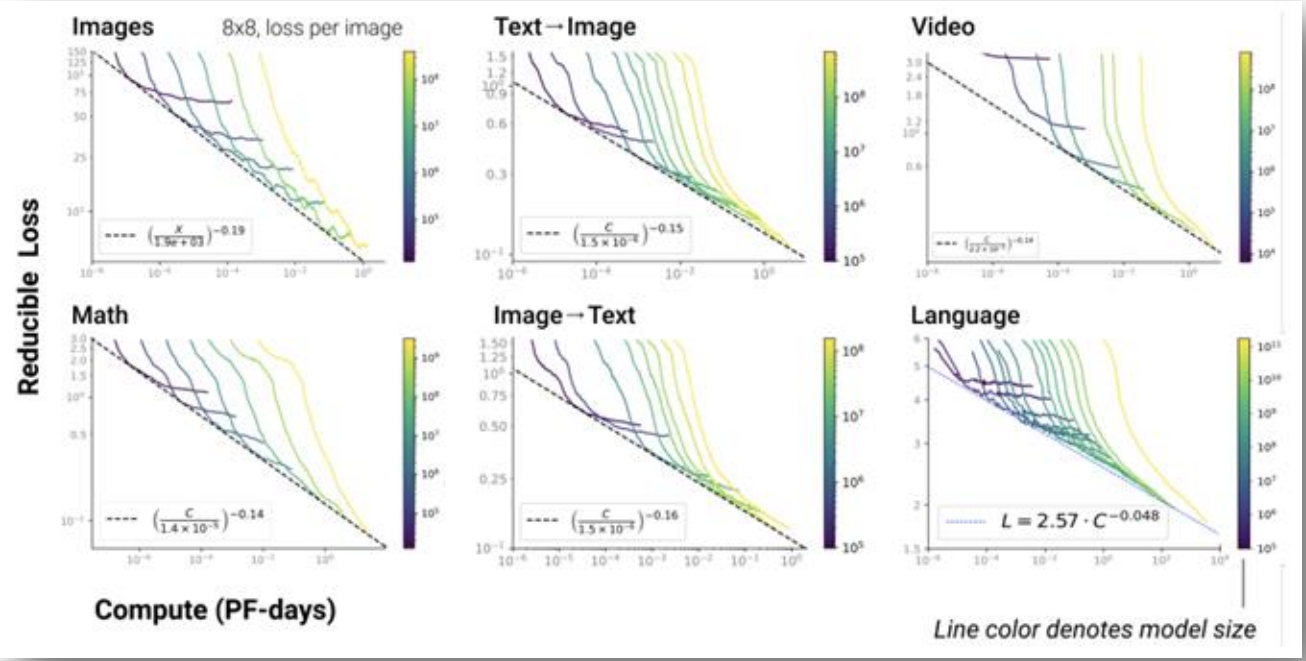


Error rate decreases with a **simultaneous** increase in compute, dataset size and model size

Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B. Brown, Prafulla Dhariwal, Scott Gray, Chris Hallacy, Benjamin Mann, Alec Radford, Aditya Ramesh, Nick Ryder, Daniel M. Ziegler, John Schulman, Dario Amodei, Sam McCandlish. **Scaling Laws for Autoregressive Generative Modeling**.2020

EMPIRICAL EVIDENCE

The Scaling Laws for generative models

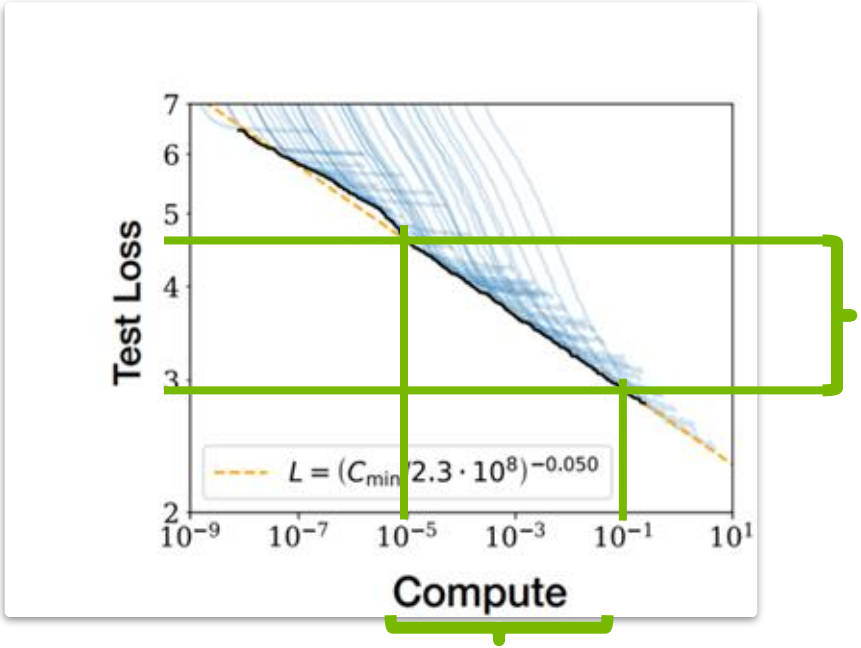


Error rate decreases with a **simultaneous** increase in compute, dataset size and model size

Henighan, Tom, et al. **"Scaling laws for autoregressive generative modeling."** *arXiv preprint arXiv:2010.14701* (2020).

ARE LARGE LANGUAGE MODELS WORTH IT?

The cost of incremental improvement



10,000x Increase

- Plots shows test set loss on y-axis and increasing compute scale on the x-axis
- As compute grows you get a decrease in test loss error
- The reduction in test loss error is not directly proportional to the increase in compute
- Why are AI companies interested in using large compute if accuracy doesn't improve that much?

Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B. Brown, Prafulla Dhariwal, Scott Gray, Chris Hallacy, Benjamin Mann, Alec Radford, Aditya Ramesh, Nick Ryder, Daniel M. Ziegler, John Schulman, Dario Amodei, Sam McCandlish. **Scaling Laws for Autoregressive Generative Modeling**.2020

FEW SHOT LEARNING

Learning from far fewer examples

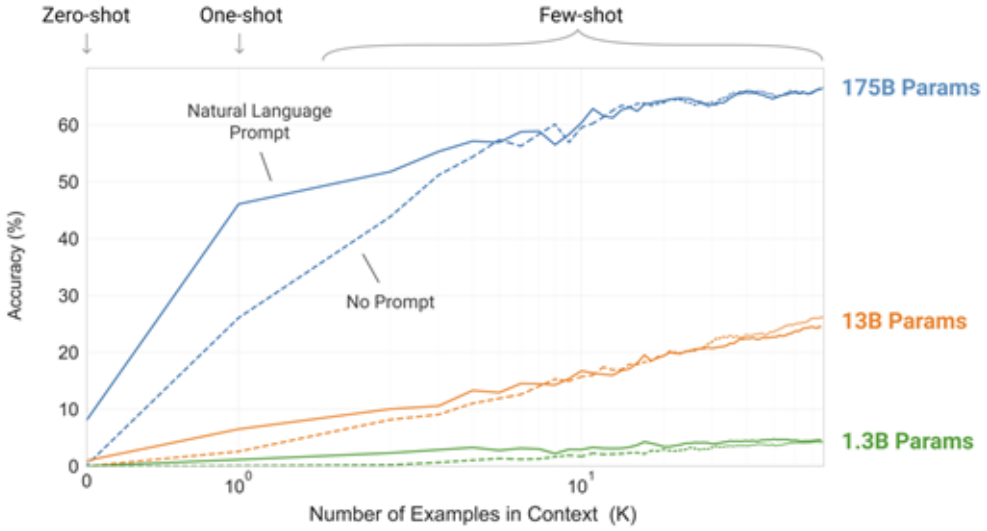
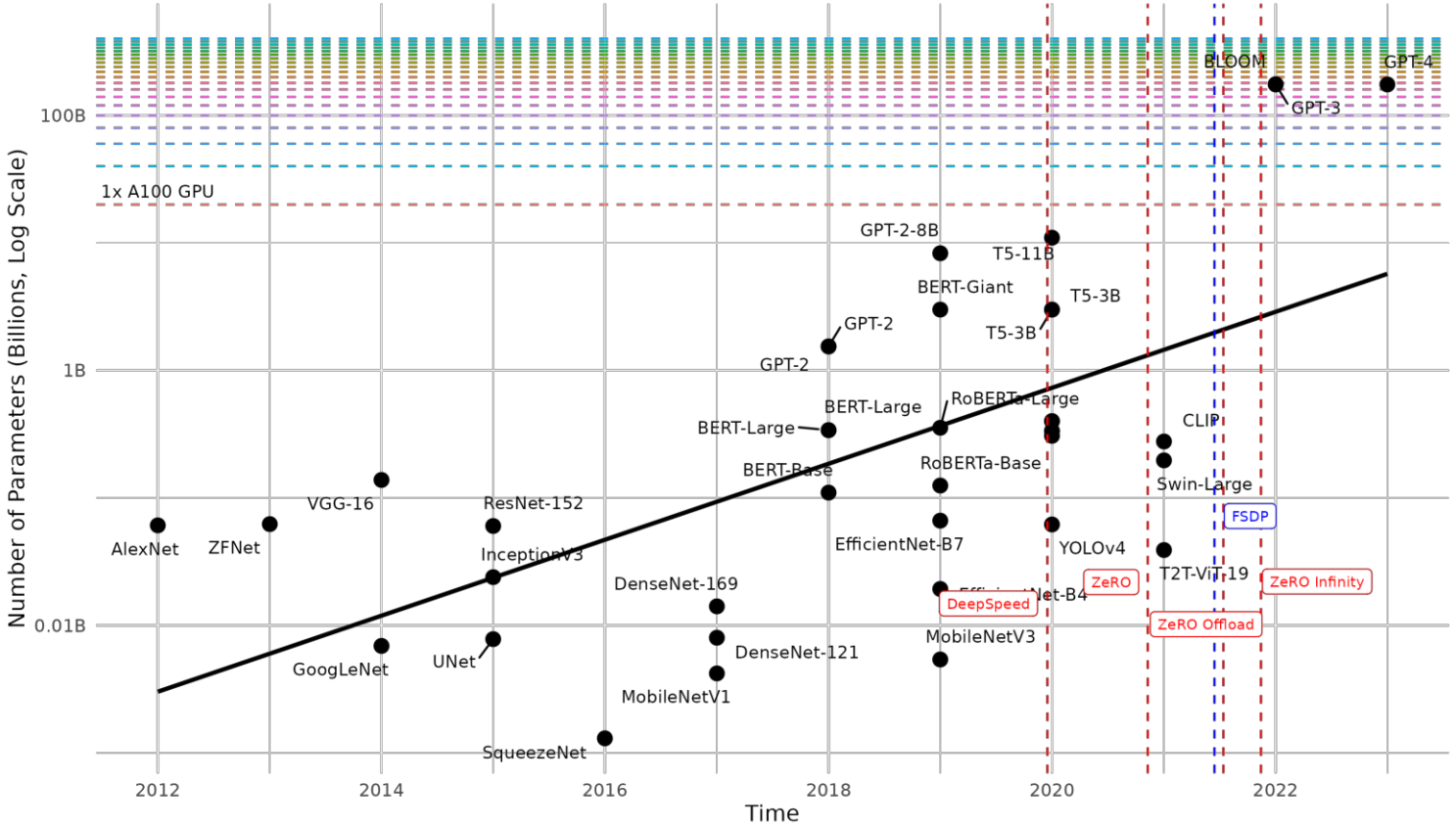


Figure 1.2: Larger models make increasingly efficient use of in-context information. We show in-context learning performance on a simple task requiring the model to remove random symbols from a word, both with and without a natural language task description (see Sec. 3.9.2). The steeper "in-context learning curves" for large models demonstrate improved ability to learn a task from contextual information. We see qualitatively similar behavior across a wide range of tasks.

Large models are more sample efficient needing less epochs to train and fewer examples learn new tasks

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Agarwal, S. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*..

Model Size



https://www.ddn.com/resources/whitepapers/large-language-models/

Model Size

Model	Parameters (B)	Size (FP16, GB)	Size (INT8, GB)	Typical Use Case
BERT Base	0.11	0.4	0.2	Text classification, embeddings
Mistral 7B	7.3	14	7	General NLP tasks, chat, summarization
Gemma 2 9B	9	18	9	Balanced reasoning and efficiency
LLaMA 3 70B	70	140	70	Complex reasoning, multilingual tasks
Mixtral 8×7B	46 (active 12.9)	26	13	Mixture-of-experts performance with lower cost
GPT-3	175	350	175	Broad NLP, text generation
Claude 2	~100	200	100	Document analysis, enterprise tasks
PaLM 2	540	1080	540	Multimodal and multilingual applications
GPT-4 (est.)	>1,000	>2,000	>1,000	Advanced reasoning, multimodal AI

https://labelyourdata.com/articles/llm-fine-tuning/llm-model-size

SCALE OF COMPUTE

Within reach of most companies

Model size	Attention heads	Hidden size	Number of layers	Number of parameters (billion)	Model-parallel size	Number of GPUs	Microbatch size	Batch size	Achieved teraFlOP/s per GPU	Percentage of theoretical peak FLOP/s	Achieved aggregate petaFLOP/s
1.7B	24	2304	24	1.7	1	32	16	512	137	44%	4.4
3.6B	32	3072	30	3.6	2	64	16	512	138	44%	8.8
7.5B	32	4096	36	7.5	4	128	16	512	142	46%	18.2
18B	48	6144	40	18.4	8	256	8	1024	135	43%	34.6
39B	64	8192	48	39.1	16	512	4	1536	138	44%	70.8
76B	80	10240	60	76.1	32	1024	2	1792	140	45%	143.8
145B	96	12288	80	145.6	64	1536	2	2304	148	47%	227.1
310B	128	16384	96	310.1	128	1920	1	2160	155	50%	297.4
530B	128	20480	105	529.6	280	2520	1	2520	163	52%	410.2
1T	160	25600	128	1008.0	512	3072	1	3072	163	52%	502.0

Weak scaling throughput for GPT models ranging from 1 billion to 1 trillion parameters.

- ~6 weeks on 1 x DGX A100
- ~2 weeks on 4 x DGX A100

- ~65 weeks on 1 x DGX A100
- ~16 weeks on 4 x DGX A100

- ~5 years on 1 x DGX A100
- ~1 year on 4 x DGX A100

- ~69 years on 1 x DGX A100
- ~17 year on 4 x DGX A100

Scaling up of training is a form of time compression enabling faster time to convergence

Narayanan, Deepak, et al. "**Efficient large-scale language model training on GPU clusters using megatron-LM.**" *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021.

PREDICTABILITY OF LARGE MODELS

Optimal Model size given a compute budget

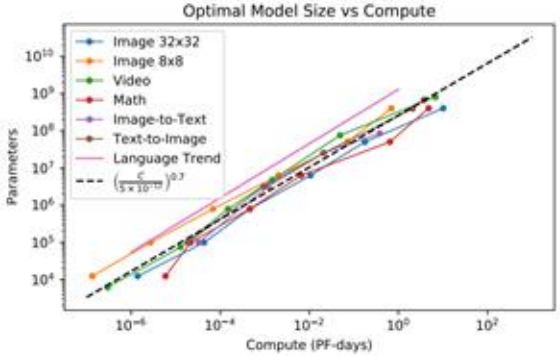


Figure 2 Optimal model size is consistent across domains— We display the optimal model size N_{opt} as a function of the training compute budget C. Not only does $N_{opt}(C)$ behave as a power-law, but the behavior is remarkably similar for all data modalities.

For a given compute budget you can estimate to optimal model size to train across different domains

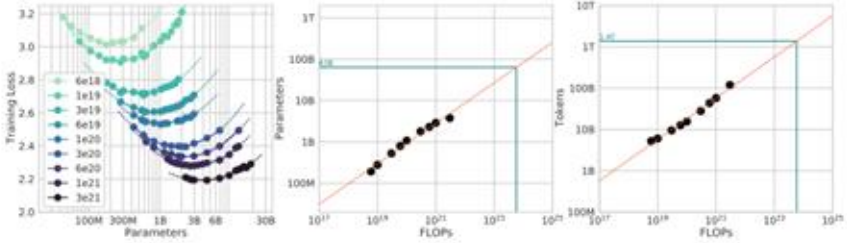


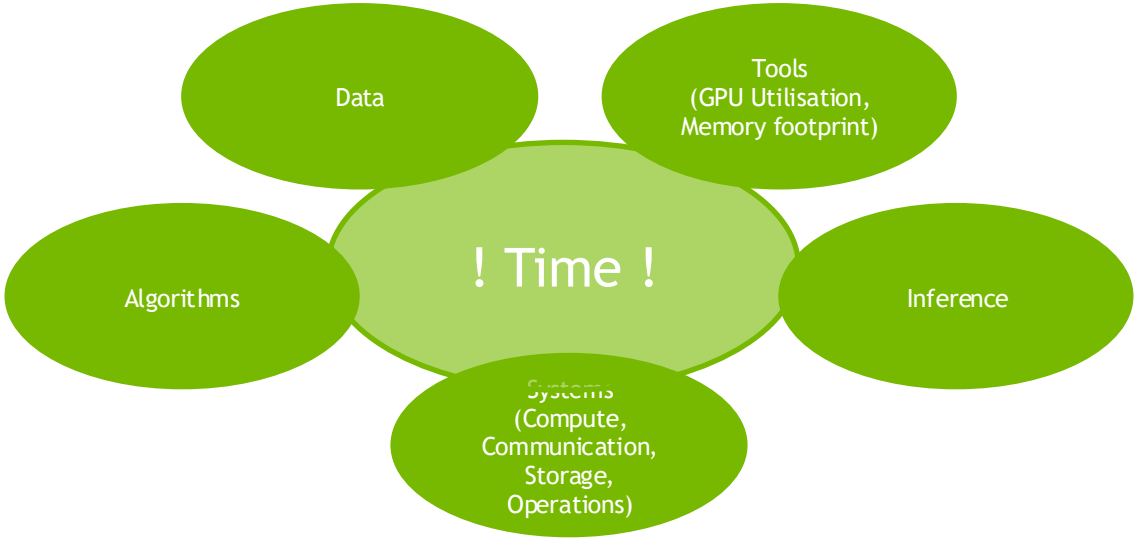
Figure 3 | **IsoFLOP curves.** For various model sizes, we choose the number of training tokens such that the final FLOPs is a constant. The cosine cycle length is set to match the target FLOP count. We find a clear valley in loss, meaning that for a given FLOP budget there is an optimal model to train (**left**). Using the location of these valleys, we project optimal model size and number of tokens for larger models (**center** and **right**). In green, we show the estimated number of parameters and tokens for an *optimal* model trained with the compute budget of *Gopher*.

For a given compute budget you can estimate to optimal model size and number of tokens needed to achieve a specific loss value

Henighan, Tom, et al. "**Scaling laws for autoregressive generative modeling.**" *arXiv preprint arXiv:2010.14701* (2020).
Hoffmann, Jordan, et al. "**Training Compute-Optimal Large Language Models.**" *arXiv preprint arXiv:2203.15556* (2022).

KEY CHALLENGES

Large models require large execution time – catastrophic impact of bottlenecks



4

GPU and VRAM

What is a GPU physically, and why does it need its own memory?

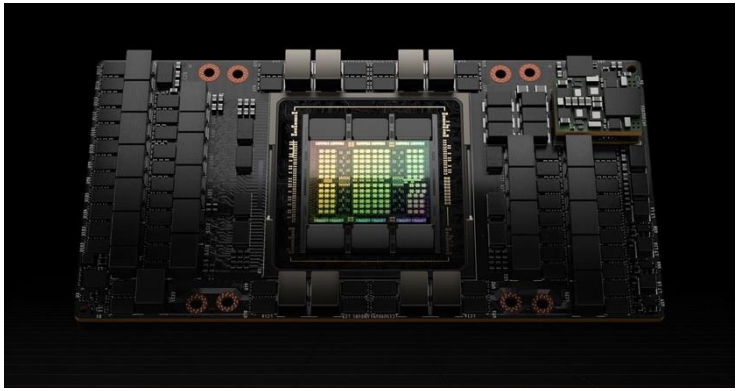
What physically IS a GPU?

A GPU is a silicon chip the size of a stamp

Everything else on the card exists to feed it:

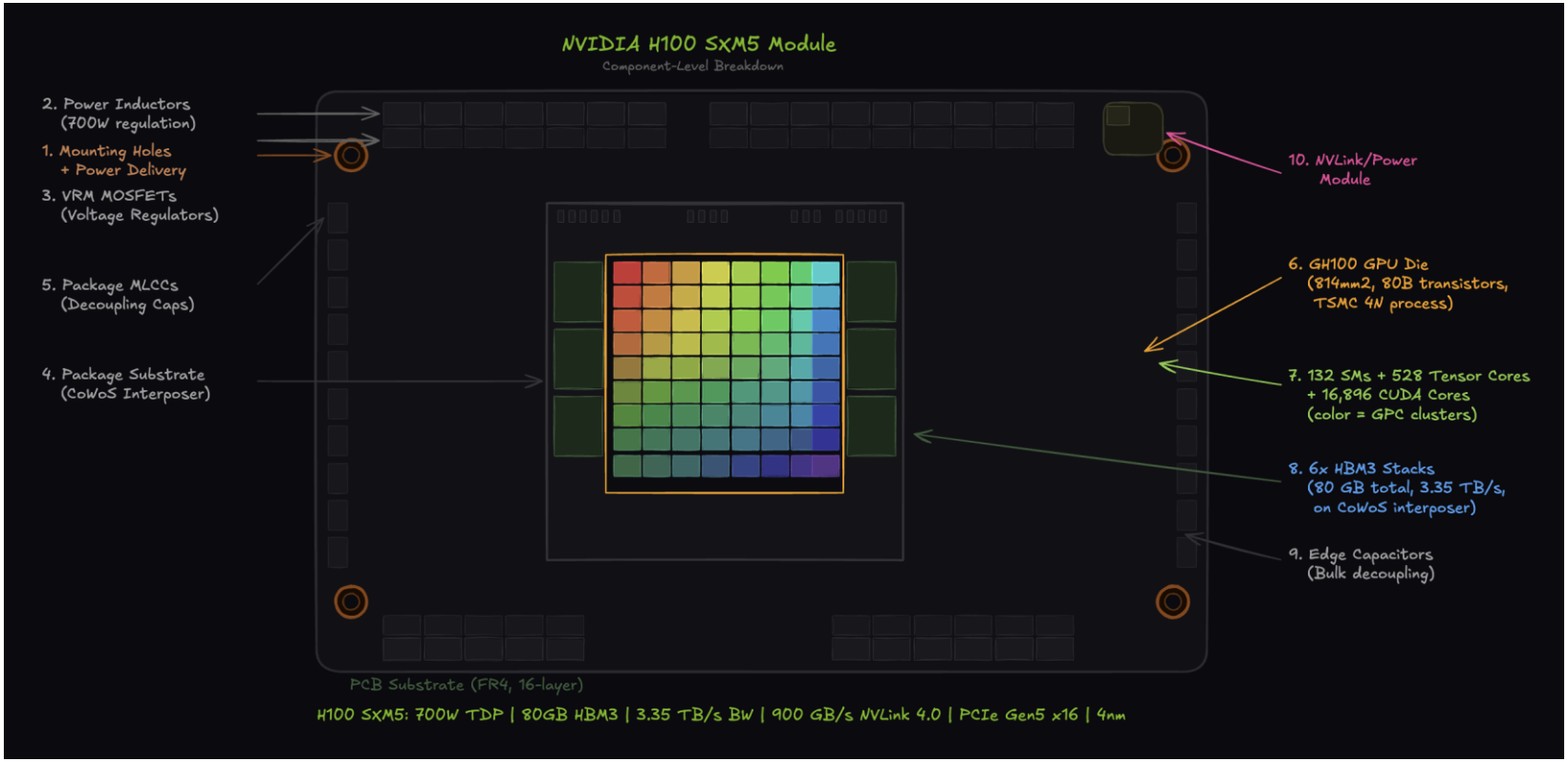
- Power delivery (700W)
- HBM3 memory stacks (80-192 GB)
- NVLink connectors (GPU-to-GPU)
- PCIe connector (to the server)

NVIDIA H100 SXM



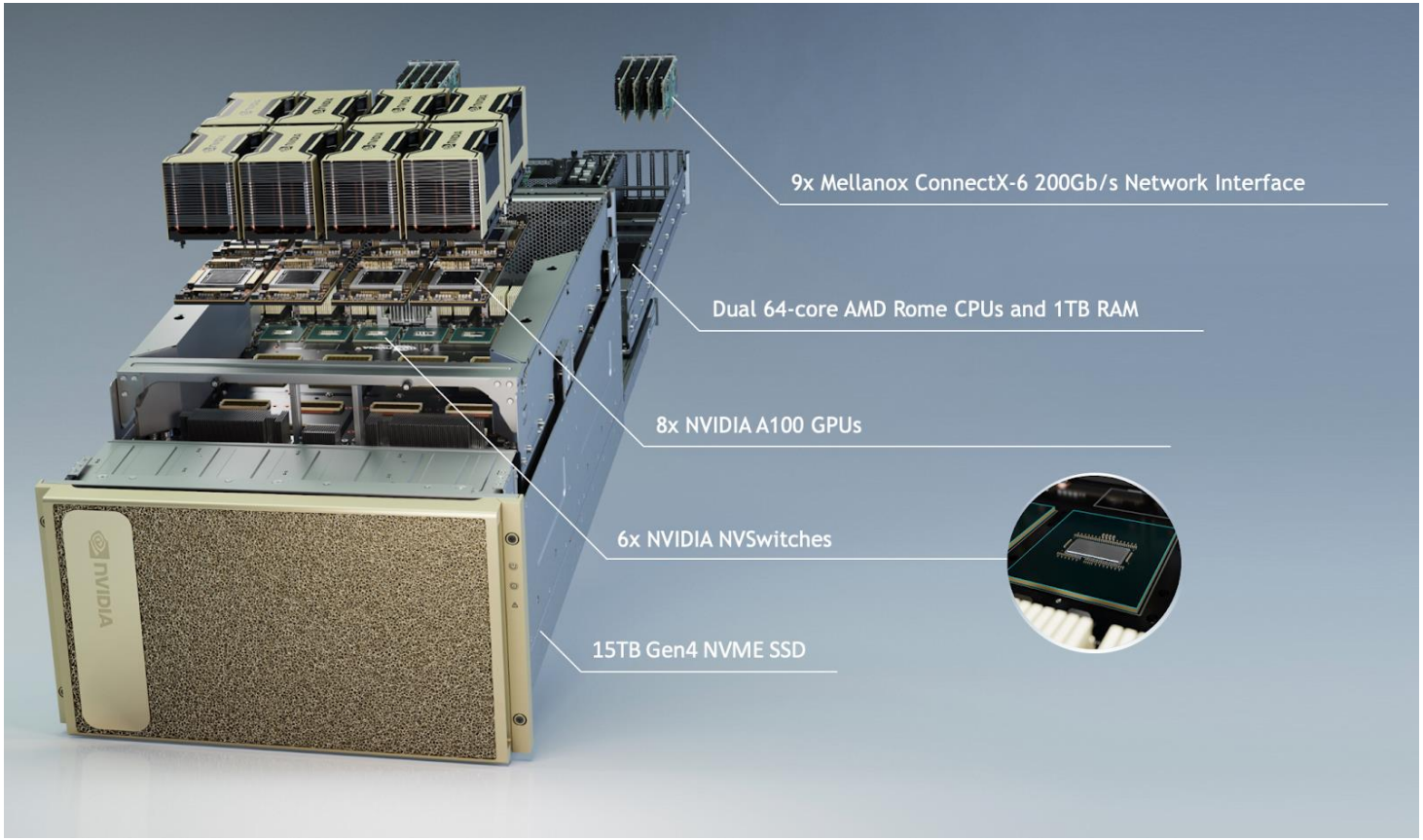
- 80 billion transistors
- 814 mm² die (postage stamp size)
- TSMC 4N process (~4 nanometer)
- 16,896 CUDA cores + 528 Tensor Cores
- 80 GB HBM3 @ 3,350 GB/s
- 700W power draw
- ~$30,000

NVIDIA H100



https://www.nvidia.com/en-us/data-center/h100/

NVIDIA DGX H100



https://mcomputers.cz/2020/05/14/nvidia-ampere/dgx-a100-exploded-view/

NVIDIA Blackwell Platform

NIM
CUDA-Accelerated
Agentic AI Libraries

Omniverse
CUDA-Accelerated
Physical AI Libraries

CUDA-X Libraries

CUDA • DOCA • NCCL
Cluster-Scale Software
System Software
Chip Software

Accelerated
Software Stack

GB200 NVL72 SuperPOD

Grace Blackwell
MGX Node

NVLink Switch

Quantum Switch

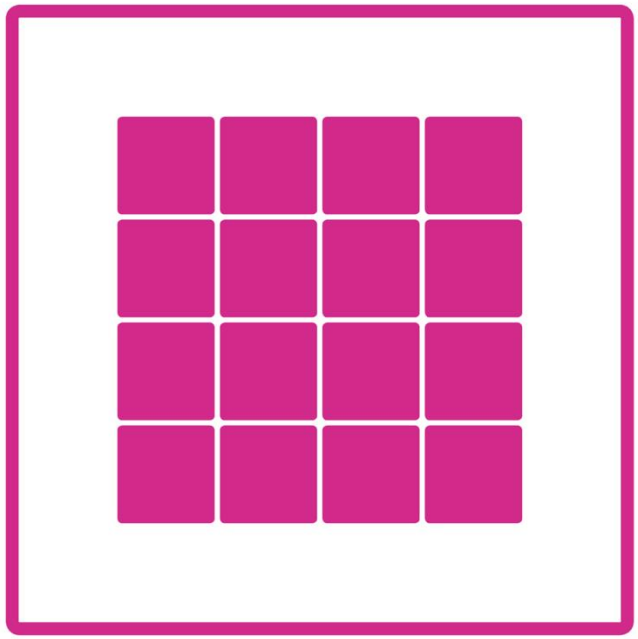
Spectrum-X Switch

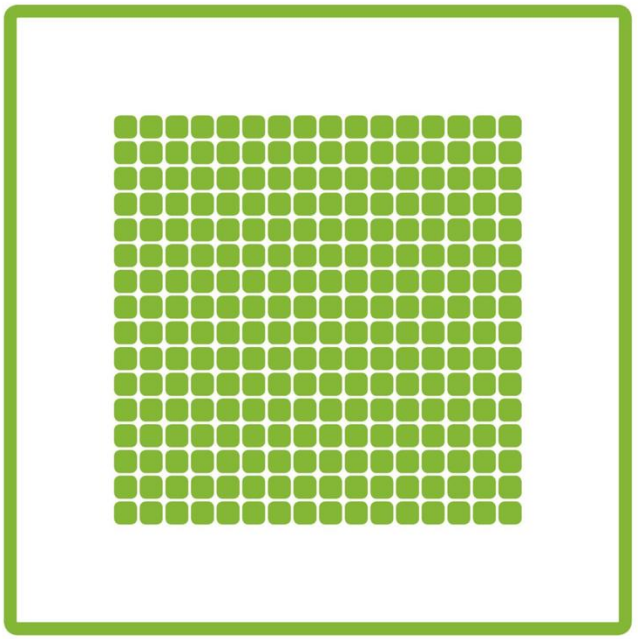
Chips Purpose-Built for AI Supercomputing
GPU | CPU | DPU | NIC | NVLink Switch | IB Switch | Enet Switch

NVIDIA DGX Spark

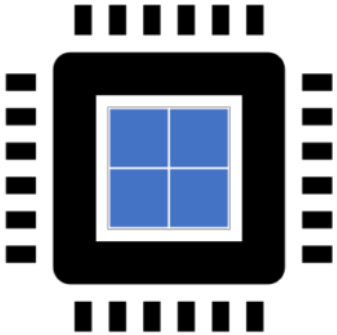
https://www.nvidia.com/en-us/data-center/h100/

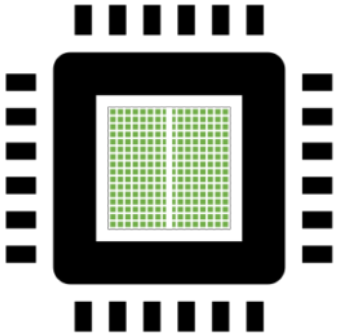
CPUs versus GPUs





Model Size





CPU	GPU
Central Processing Unit	Graphics Processing Unit
4-8 Cores	100s or 1000s of Cores
Low Latency	High Throughput
Good for Serial Processing	Good for Parallel Processing
Quickly Process Tasks That Require Interactivity	Breaks Jobs Into Separate Tasks To Process Simultaneously
Traditional Programming Are Written For CPU Sequential Execution	Requires Additional Software To Convert CPU Functions to GPU Functions for Parallel Execution

https://www.layerstack.com/blog/cpu-vs-gpu-which-processor-is-right-for-you/?srsltid=AfmBOoov85RZoMYxD8LkUEwsVr9zHn1CaNvhQn23kERtzGlDs55tvsPC

What does a single GPU core actually do?

One CUDA core does ONE operation per clock:

```
result = a × b + c
```

Fused Multiply-Add (FMA)

H100 Scale

16,896

CUDA cores all multiplying at once

GPU Performance:

H100 runs at ~1.8 GHz → 1.8 billion cycles per second.
Each CUDA core does 1 FMA per cycle = 2 FLOPs.
So one core ≈ 3.6 GFLOPs/sec.

With 16,896 cores:
≈ 60 TFLOPs (FP32) total 60 trillion operations per second

Same model, four sizes

Precision	Bytes/weight	175B Size	Fits on...	Quality
FP32	4 bytes	700 GB	10x H100 (80 GB each)	Perfect fidelity
FP16 / BF16	2 bytes	350 GB	5x H100	Negligible loss
INT8	1 byte	175 GB	3x H100	Very slight loss
INT4	0.5 bytes	**88 GB**	**2x RTX 4090 (24 GB + offload)**	Noticeable on reasoning

Inference (per token): ~350B FLOPs/token **Training (per token):** ~1+ T FLOPs/token

Photo analogy: FP32 = RAW photo (huge, perfect) | FP16 = High JPEG (half size, can't tell) | INT8 = Medium JPEG | INT4 = Thumbnail (fine for most uses)

More cores = need more bandwidth

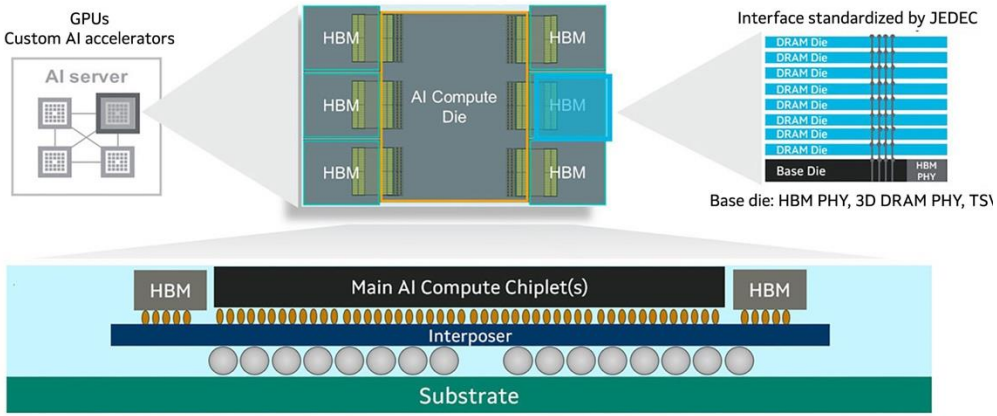
GPU	Cores	VRAM	Bandwidth	Price
RTX 4090	16,384	24 GB	1,008 GB/s	$1,600
A100	6,912	80 GB	2,039 GB/s	$10,000
H100	16,896	80 GB	**3,350 GB/s**	$30,000
B200	18,432	**192 GB**	**8,000 GB/s**	~$40,000

B200 has only ~10% more cores than H100

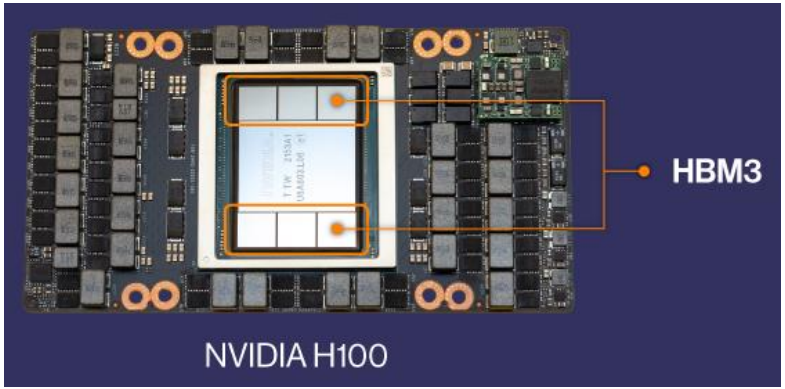
but 2.4x the VRAM and 2.4x the bandwidth

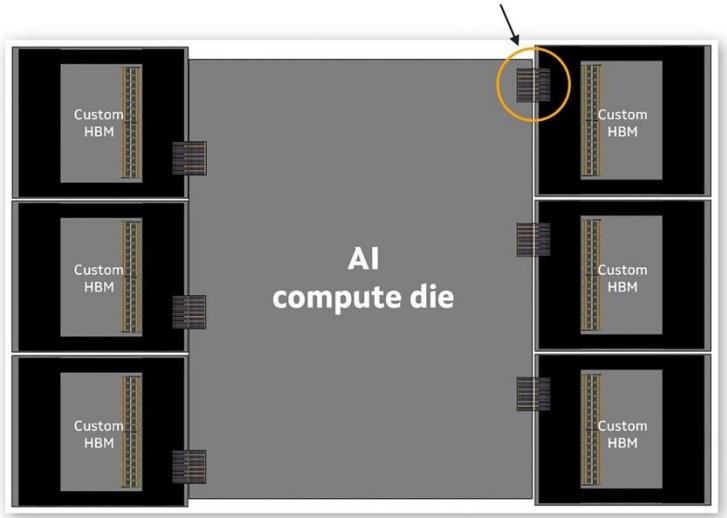
NVIDIA is investing in memory, not cores -- because memory is the bottleneck for AI

VRAM: High Bandwidth Memory (HBM)



Standard HBM integration with interposer





XPU with Marvell custom HBM architecture

https://www.marvell.com/products/custom-asic/custom-hbm-compute-architecture.html

https://www.fibermall.com/blog/gddr-hbm.htm?srsltid=AfmBOopoJjmjClbLJ2hylqEgFe-NvXU7wUjEuvg7qKOaecdckHcmW0Jt

VRAM vs your laptop's RAM

	System RAM (DDR5)	GPU VRAM (HBM3e)
Physical form	Long stick on motherboard	Tiny stacked chips on GPU
Distance from chip	~10-30 cm from CPU	~2 cm from GPU die
Bandwidth	**~50 GB/s**	**3,350 GB/s**
Capacity	64-512 GB	24-192 GB
Bus width	64-bit	5,120-bit

VRAM is 67x faster but much smaller. Many times VRAM is the bottleneck

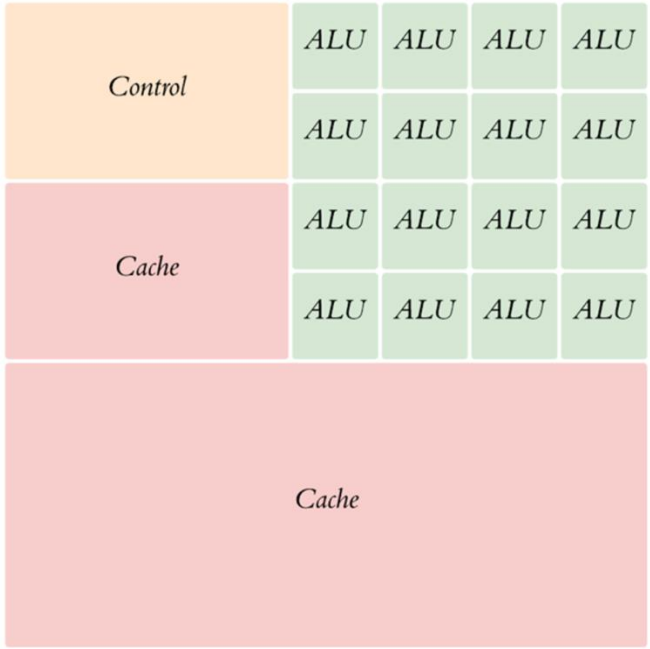
Why can't the GPU just use system RAM?

The bandwidth bottleneck:

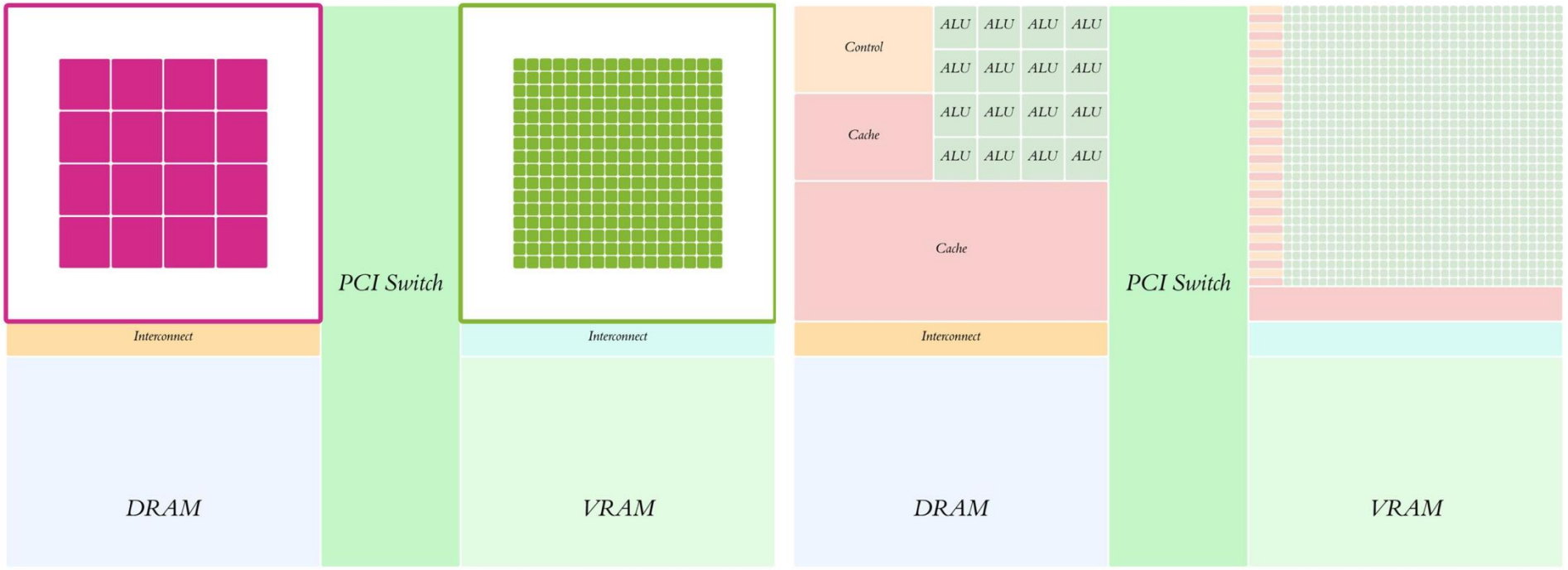
LLaMA 70B at FP16 (140 GB weights):

Via PCIe: 140 / 64 = 2.2 sec/token vs **From VRAM: 140 / 3,350 = 0.04 sec/token**

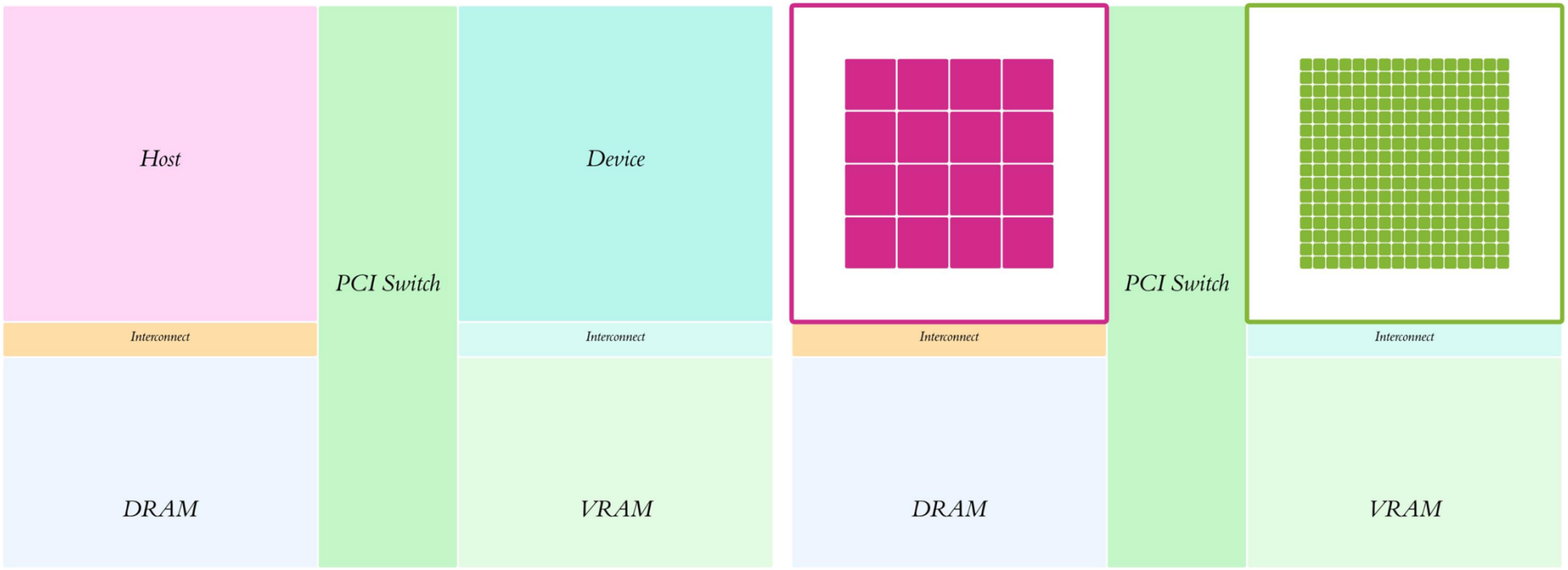
Arithmetic Logic Units (ALUs) - Controls - Caches



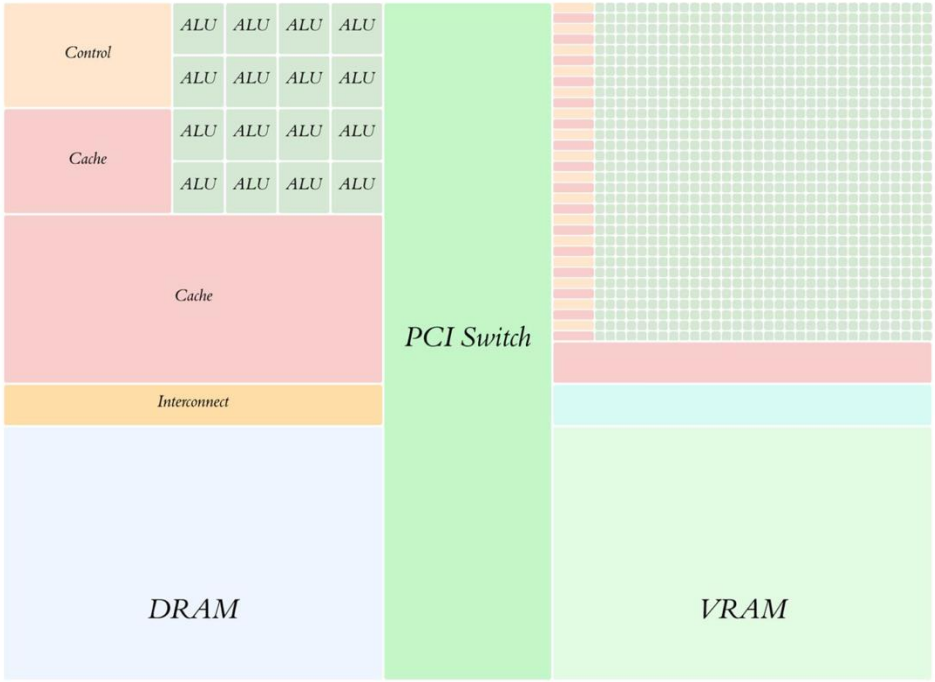
Heterogeneous Computing



Host - Device Terminologies

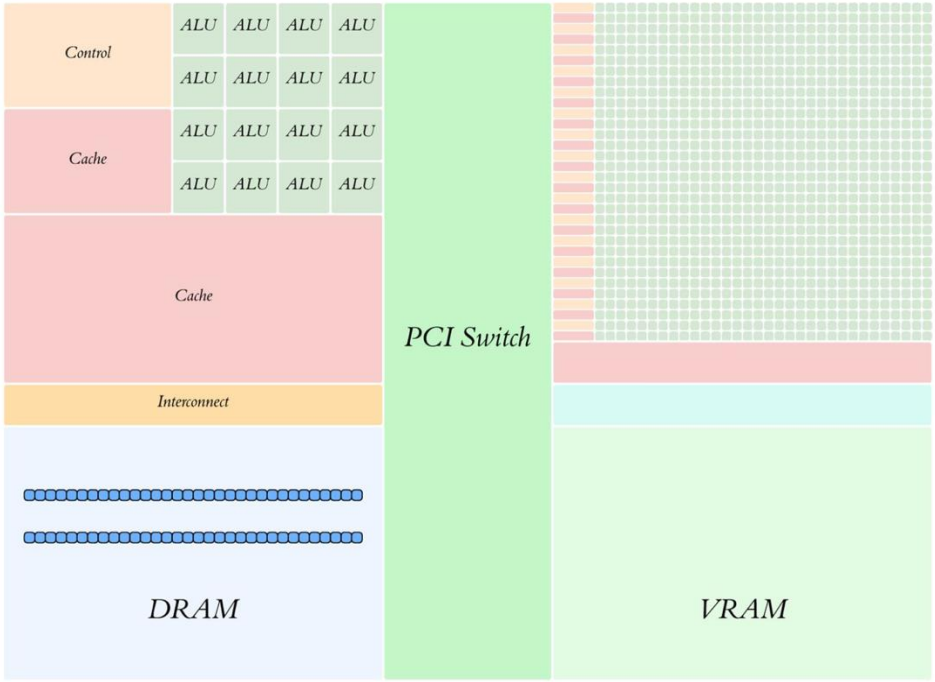


CUDA C/C++ program



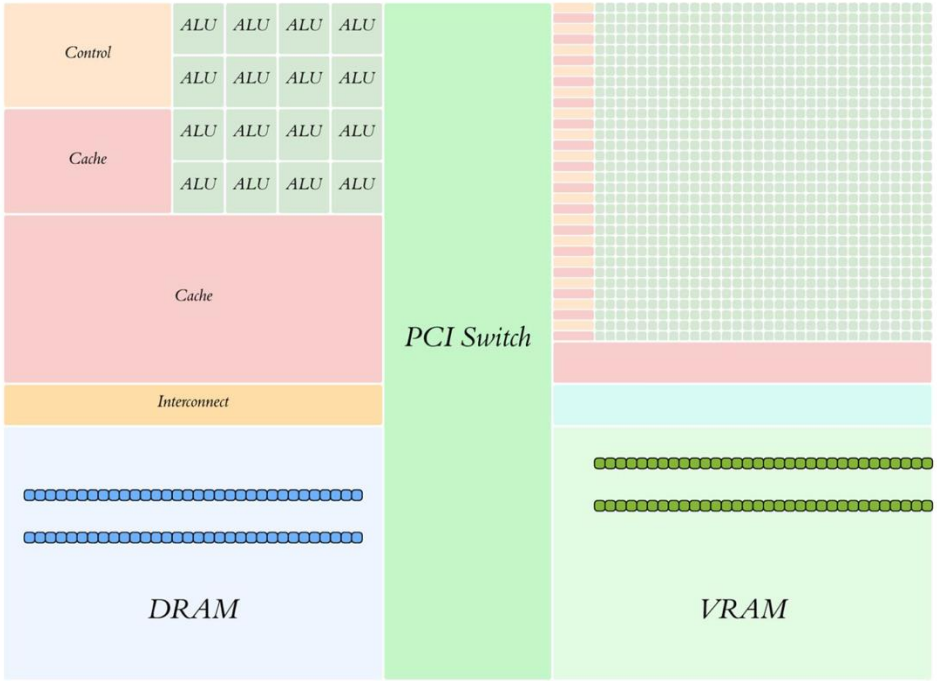

```
int main(int argc, char **argv) {
   int size = n * sizeof(float);
   float *h_src, *h_dst, *d_src, *d_dst;

   malloc((void **) &h_src, size); memset(...);
   malloc((void **) &h_dst, size); memset(...);
   cudaMalloc((void **) &d_src, size);
   cudaMalloc((void **) &d_dst, size);

   cudaMemcpy(d_src, h_src, size, cudaMemcpyHostToDevice);
   mathKernel<<<..., ...>>>(d_src, d_dst, ...);
   cudaMemcpy(h_dst, d_dst, size, cudaMemcpyDeviceToHost);

   cudaFree(d_src);
   cudaFree(d_dst);
   free(h_src);
   free(h_dst);
   return 0;
}
```

malloc(...);



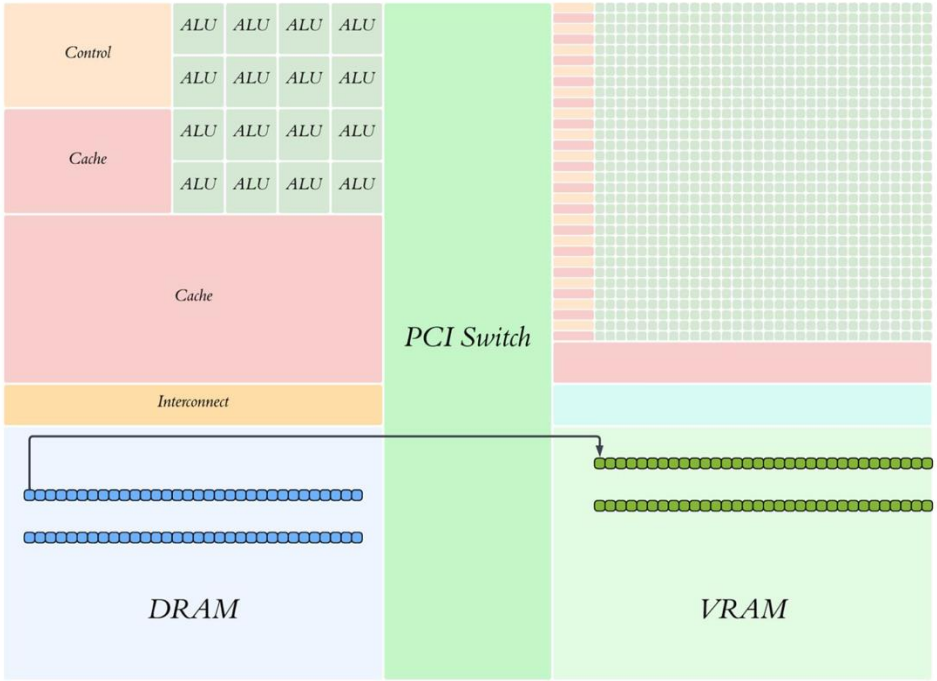

```
int main(int argc, char **argv) {
   int size = n * sizeof(float);
   float *h_src, *h_dst, *d_src, *d_dst;

   malloc((void **) &h_src, size); memset(...);
   malloc((void **) &h_dst, size); memset(...);
   cudaMalloc((void **) &d_src, size);
   cudaMalloc((void **) &d_dst, size);

   cudaMemcpy(d_src, h_src, size, cudaMemcpyHostToDevice);
   mathKernel<<<..., ...>>>(d_src, d_dst, ...);
   cudaMemcpy(h_dst, d_dst, size, cudaMemcpyDeviceToHost);

   cudaFree(d_src);
   cudaFree(d_dst);
   free(h_src);
   free(h_dst);
   return 0;
}
```

cudaMalloc(...);



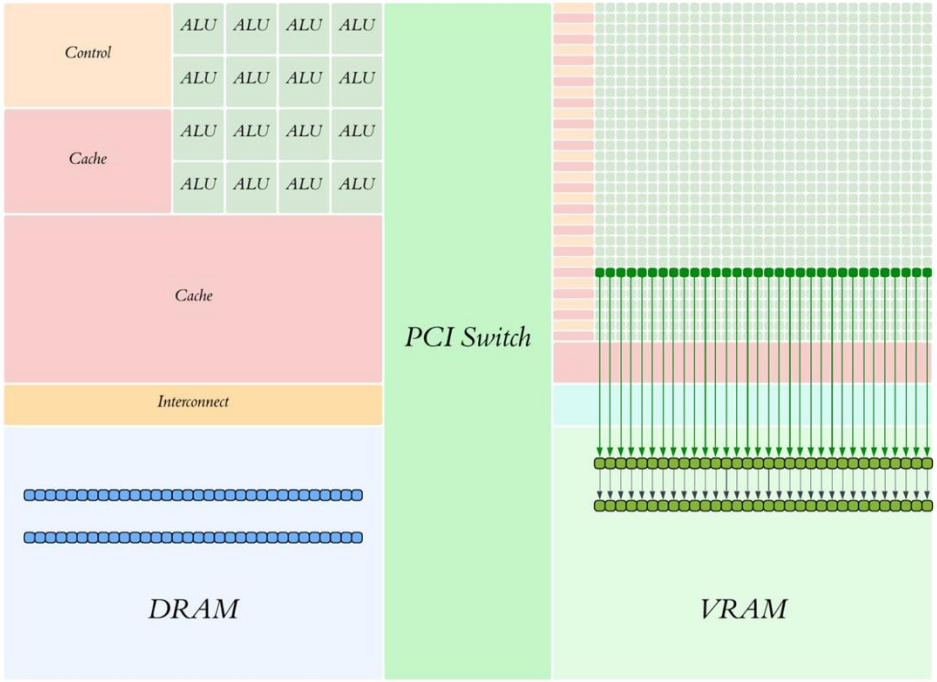

```
int main(int argc, char **argv) {
    int size = n * sizeof(float);
    float *h_src, *h_dst, *d_src, *d_dst;

    malloc((void **) &h_src, size); memset(...);
    malloc((void **) &h_dst, size); memset(...);
    cudaMalloc((void **) &d_src, size);
    cudaMalloc((void **) &d_dst, size);

    cudaMemcpy(d_src, h_src, size, cudaMemcpyHostToDevice);
    mathKernel<<<..., ...>>>(d_src, d_dst, ...);
    cudaMemcpy(h_dst, d_dst, size, cudaMemcpyDeviceToHost);

    cudaFree(d_src);
    cudaFree(d_dst);
    free(h_src);
    free(h_dst);
    return 0;
}
```

cudaMemcopy(dst, src, size, cudaMemcpyHostToDevice);



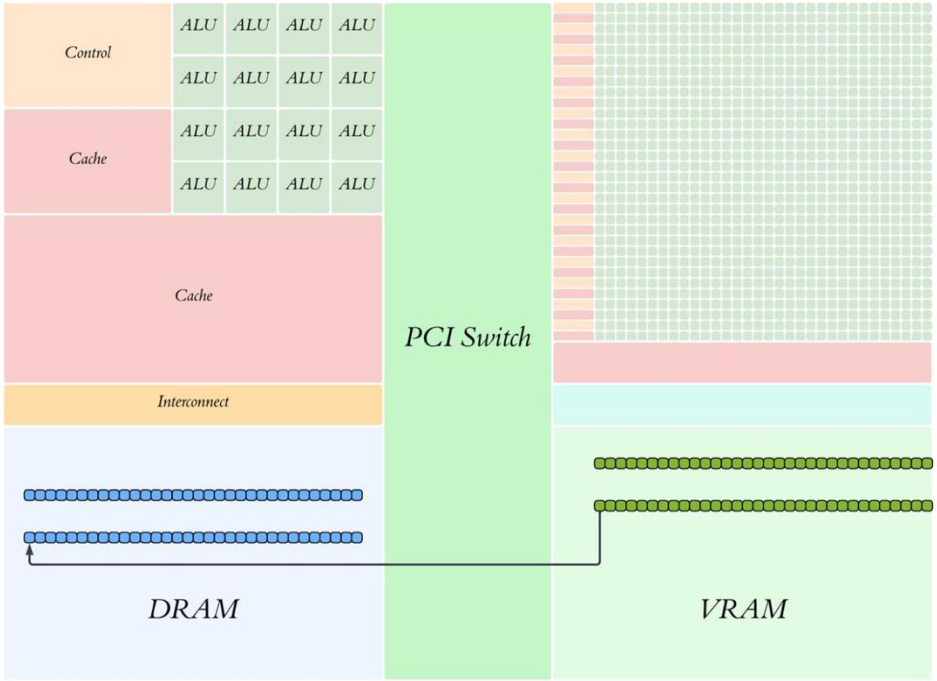

```
int main(int argc, char **argv) {
   int size = n * sizeof(float);
   float *h_src, *h_dst, *d_src, *d_dst;

   malloc((void **) &h_src, size); memset(...);
   malloc((void **) &h_dst, size); memset(...);
   cudaMalloc((void **) &d_src, size);
   cudaMalloc((void **) &d_dst, size);

   cudaMemcpy(d_src, h_src, size, cudaMemcpyHostToDevice);
   mathKernel<<<..., ...>>>(d_src, d_dst, ...);
   cudaMemcpy(h_dst, d_dst, size, cudaMemcpyDeviceToHost);

   cudaFree(d_src);
   cudaFree(d_dst);
   free(h_src);
   free(h_dst);
   return 0;
}
```

mathKernel<<<..., ...>>>(...);



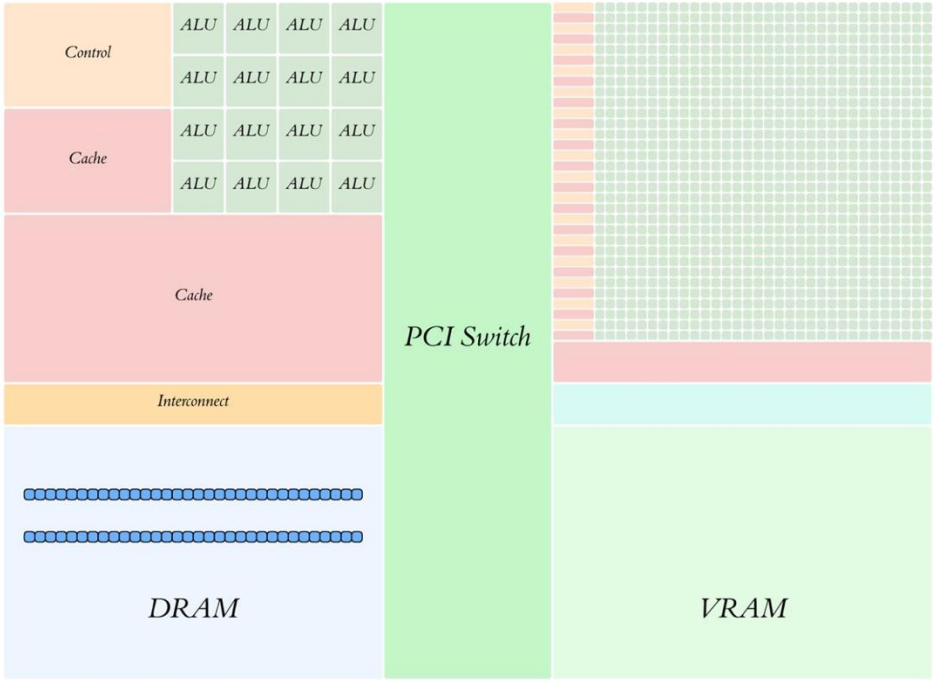

```
int main(int argc, char **argv) {
   int size = n * sizeof(float);
   float *h_src, *h_dst, *d_src, *d_dst;

   malloc((void **) &h_src, size); memset(...);
   malloc((void **) &h_dst, size); memset(...);
   cudaMalloc((void **) &d_src, size);
   cudaMalloc((void **) &d_dst, size);

   cudaMemcpy(d_src, h_src, size, cudaMemcpyHostToDevice);
   mathKernel<<<..., ...>>>(d_src, d_dst, ...);
   cudaMemcpy(h_dst, d_dst, size, cudaMemcpyDeviceToHost);

   cudaFree(d_src);
   cudaFree(d_dst);
   free(h_src);
   free(h_dst);
   return 0;
}
```

cudaMemcopy(dst, src, size, cudaMemcpyDeviceToHost);



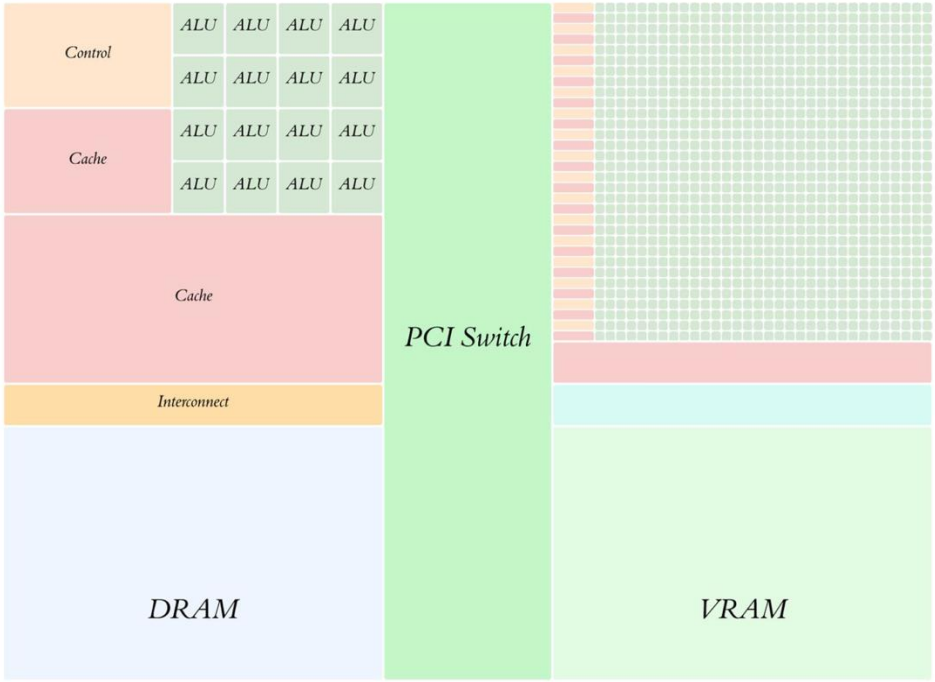

```
int main(int argc, char **argv) {
    int size = n * sizeof(float);
    float *h_src, *h_dst, *d_src, *d_dst;

    malloc((void **) &h_src, size); memset(...);
    malloc((void **) &h_dst, size); memset(...);
    cudaMalloc((void **) &d_src, size);
    cudaMalloc((void **) &d_dst, size);

    cudaMemcpy(d_src, h_src, size, cudaMemcpyHostToDevice);
    mathKernel<<<..., ...>>>(d_src, d_dst, ...);
    cudaMemcpy(h_dst, d_dst, size, cudaMemcpyDeviceToHost);

    cudaFree(d_src);
    cudaFree(d_dst);
    free(h_src);
    free(h_dst);
    return 0;
}
```

cudaFree(...);



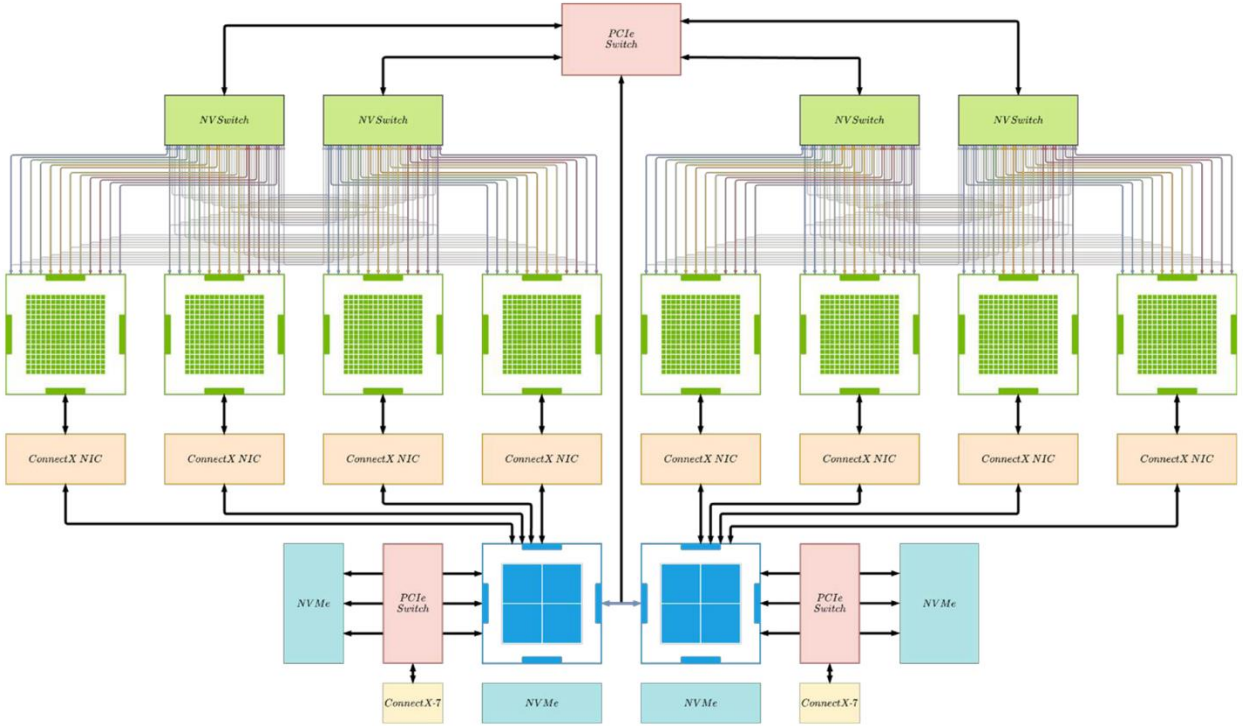

```
int main(int argc, char **argv) {
   int size = n * sizeof(float);
   float *h_src, *h_dst, *d_src, *d_dst;

   malloc((void **) &h_src, size); memset(...);
   malloc((void **) &h_dst, size); memset(...);
   cudaMalloc((void **) &d_src, size);
   cudaMalloc((void **) &d_dst, size);

   cudaMemcpy(d_src, h_src, size, cudaMemcpyHostToDevice);
   mathKernel<<<..., ...>>>(d_src, d_dst, ...);
   cudaMemcpy(h_dst, d_dst, size, cudaMemcpyDeviceToHost);

   cudaFree(d_src);
   cudaFree(d_dst);
   free(h_src);
   free(h_dst);
   return 0;
```

free(...);

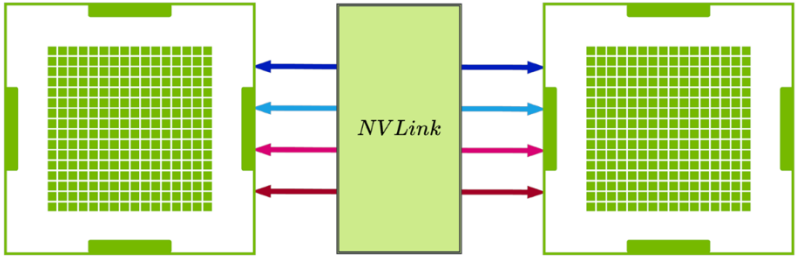


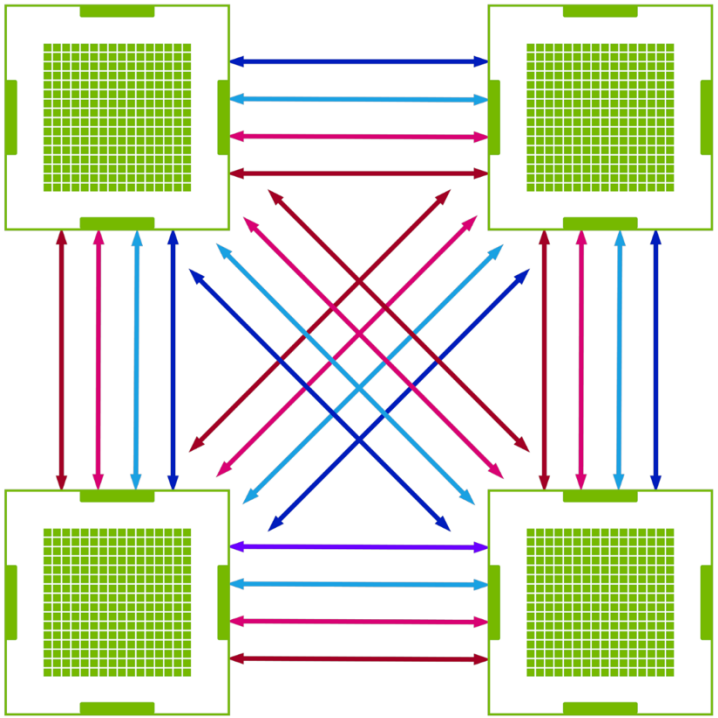

```
int main(int argc, char **argv) {
    int size = n * sizeof(float);
    float *h_src, *h_dst, *d_src, *d_dst;

    malloc((void **) &h_src, size); memset(...);
    malloc((void **) &h_dst, size); memset(...);
    cudaMalloc((void **) &d_src, size);
    cudaMalloc((void **) &d_dst, size);

    cudaMemcpy(d_src, h_src, size, cudaMemcpyHostToDevice);
    mathKernel<<<..., ...>>>(d_src, d_dst, ...);
    cudaMemcpy(h_dst, d_dst, size, cudaMemcpyDeviceToHost);

    cudaFree(d_src);
    cudaFree(d_dst);
    free(h_src);
    free(h_dst);
    return 0;
}
```

DGX H100 Systems

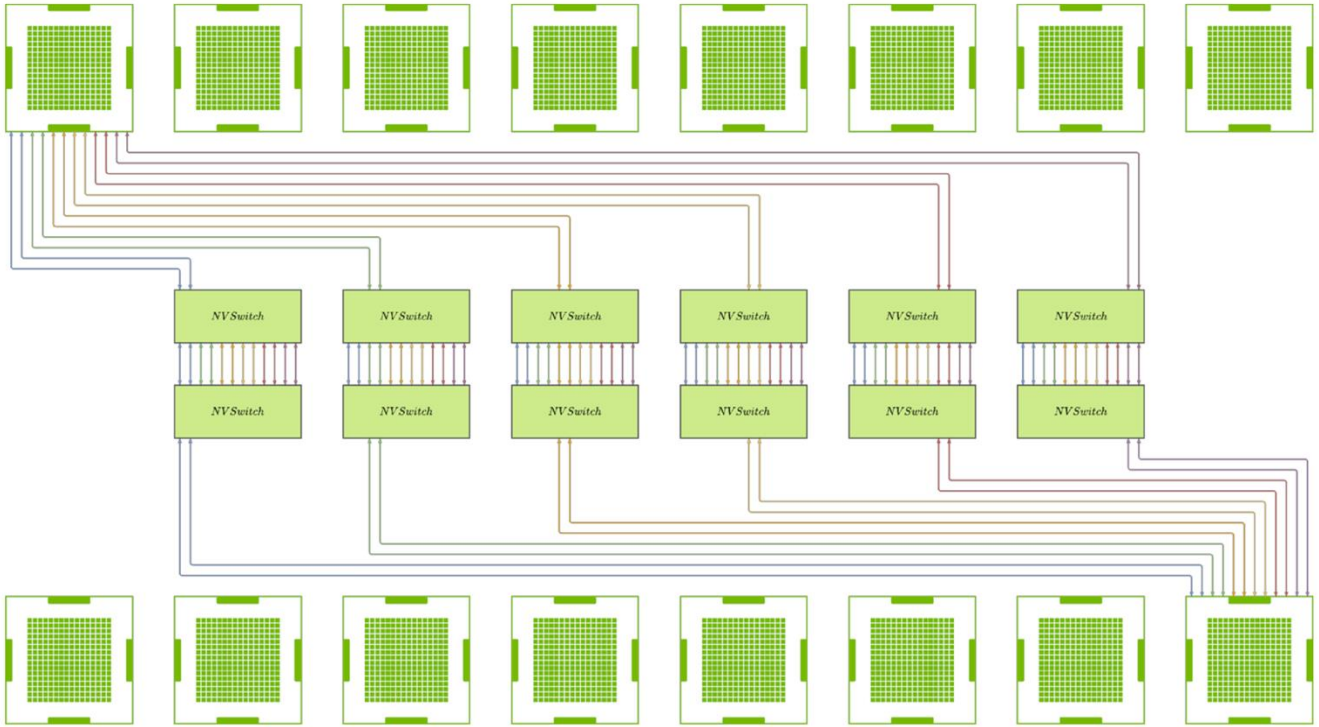


NVLink





NVSwitch

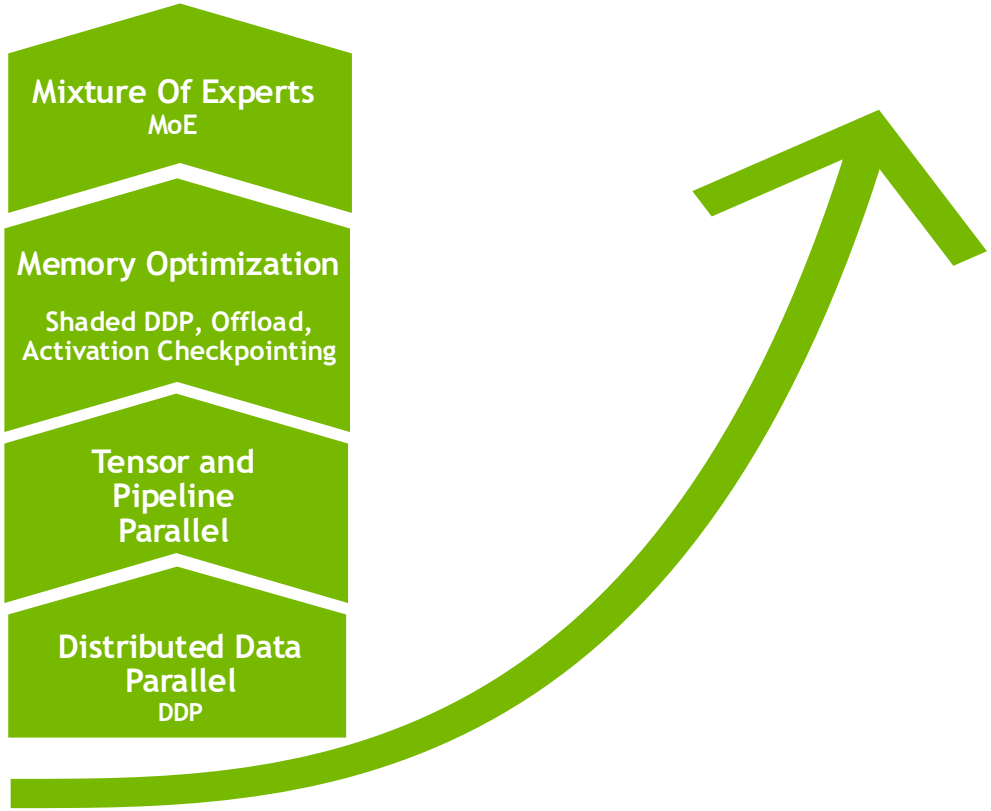


5

Advanced concepts of Large Distributed Training

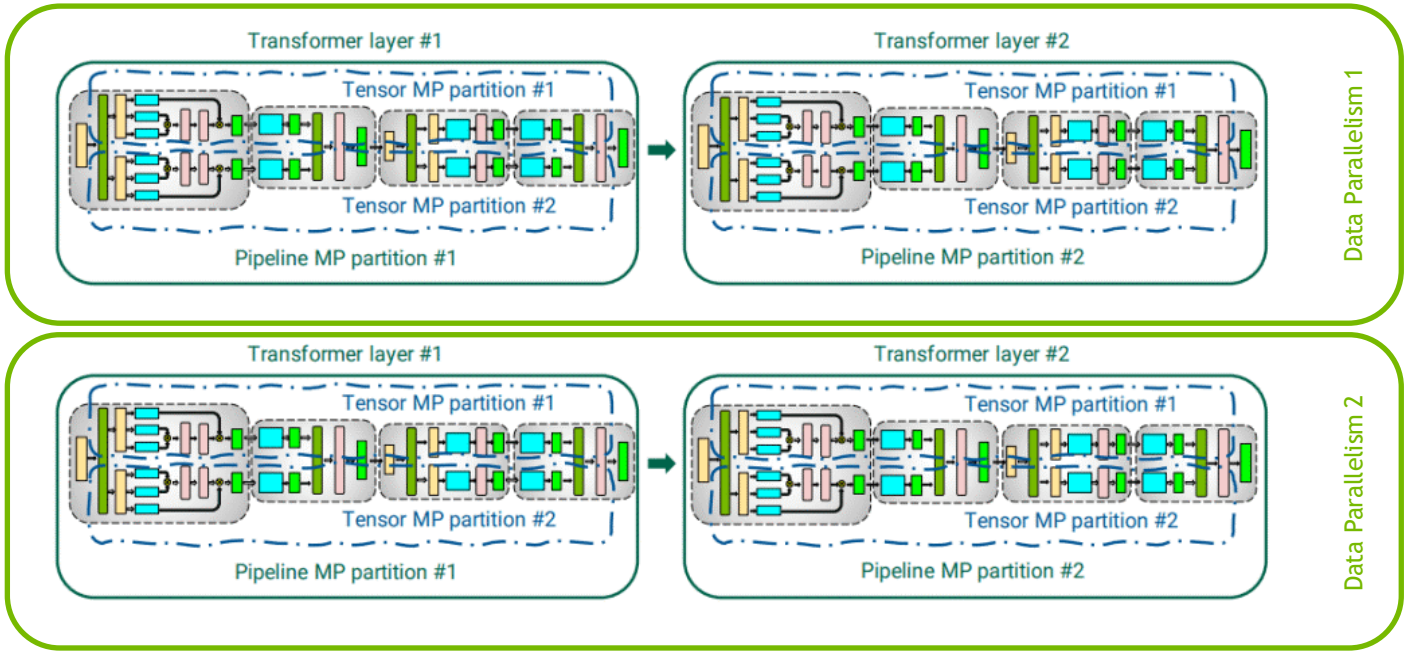
SCALING CHALLENGES

Distributed Training



DEALING WITH MEMORY CONSTRAINTS

Various forms of parallelism



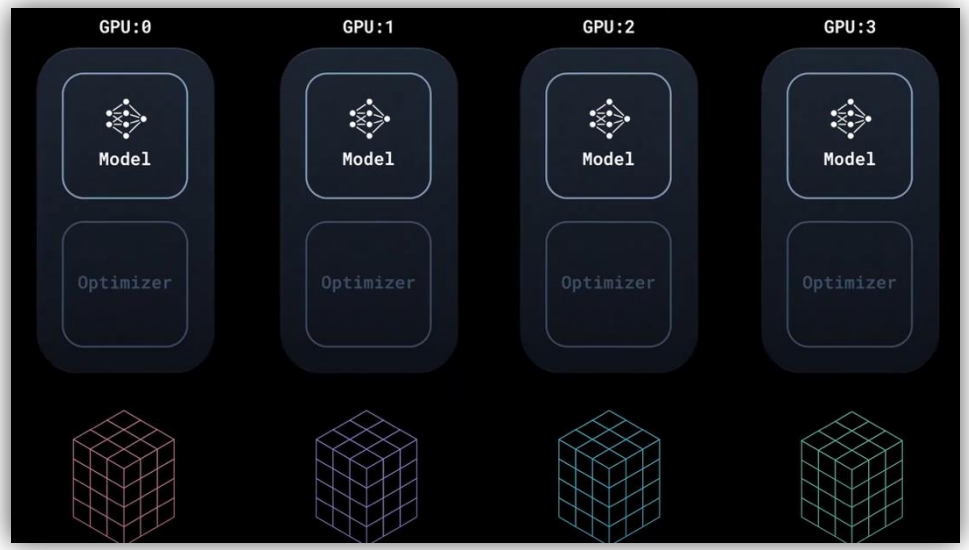
4 main types of parallelism are data, tensor, pipeline, and sequence

DISTRIBUTED DATA PARALLEL – DDP

PROCESS MORE DATA IN THE SAME TIME PERIOD

DATA PARALLELISM

PyTorch Distributed Data Parallel



Process more data by scaling model replicas

Each GPU has a copy of the model

Optimizer states and random seeds are the same

Each GPU gets a different batch of data

Concurrently processing 4x the data of a single GPU

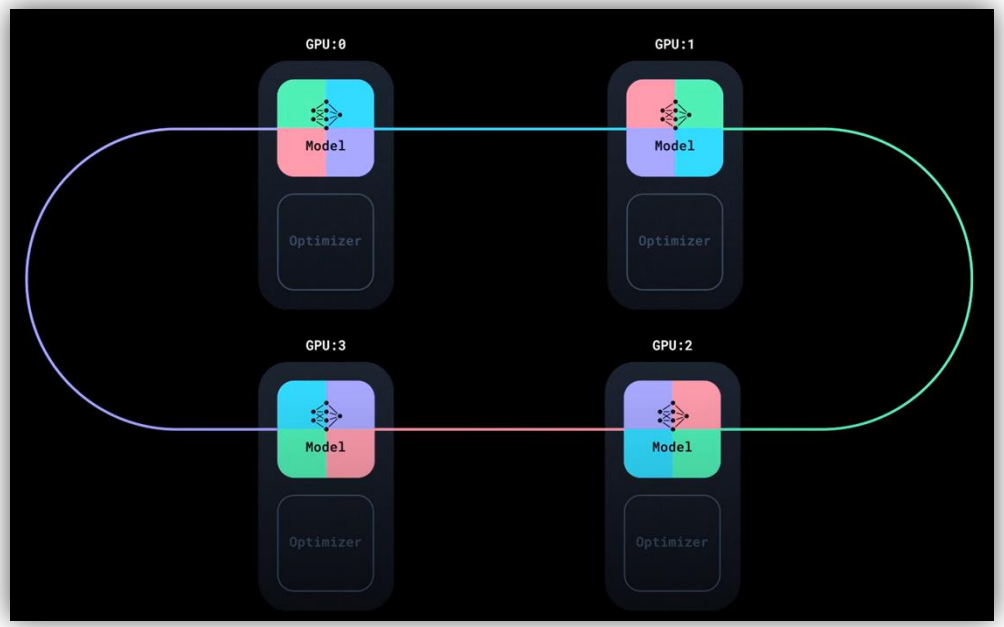
Data sampling is handled by a Distributed Sampler

Data Parallel training at large scale may affect model quality

PyTorch Distributed Data Parallel
Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model

DATA PARALLEL

PyTorch Data Parallel Training



Each model is different because it trained on different batch of data

Gradients from each GPU need to be synchronized, represented by the curved lines

Synchronization is done with bucketed Ring-AllReduce algorithm

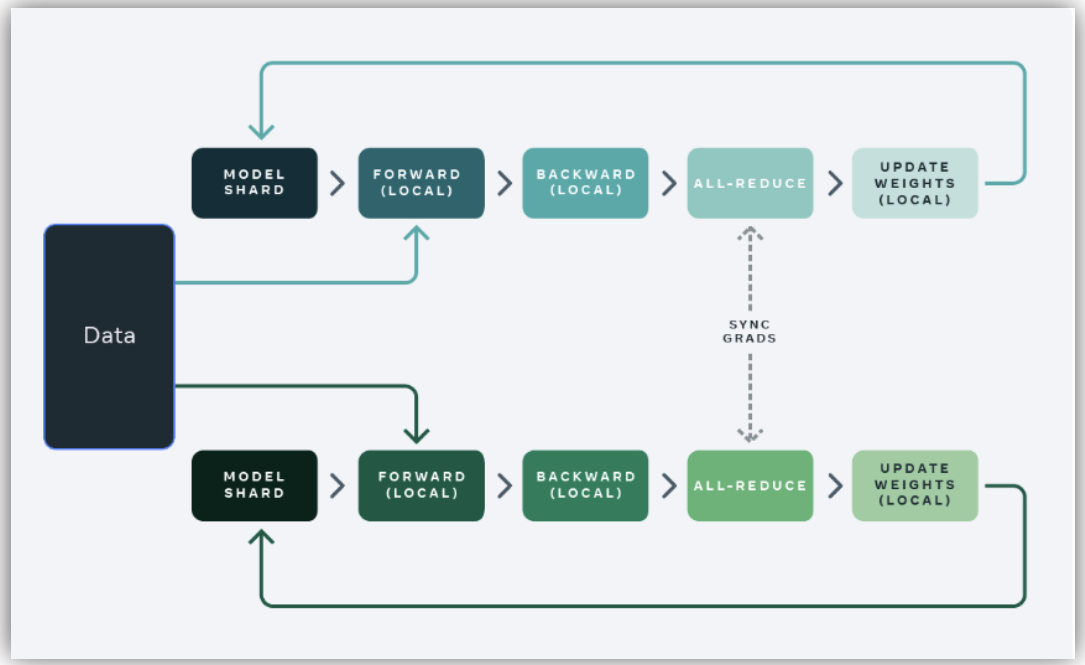
Algorithm overlaps gradient computation with communication so GPU is always utilized

Scaling with data parallel introduces communication overhead when syncing gradients

PyTorch Distributed Data Parallel

STANDARD DISTRIBUTED DATA PARALLEL - DDP

PyTorch Data Parallel Training



Gradients are synchronized before the update step, ensured model replicas stay consistent across iterations

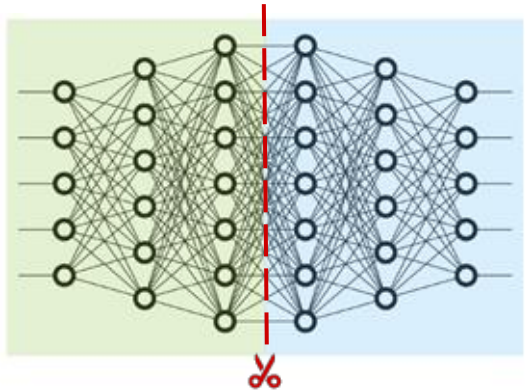
PyTorch Distributed Data Parallel

TENSOR / PIPELINE PARALLELISM

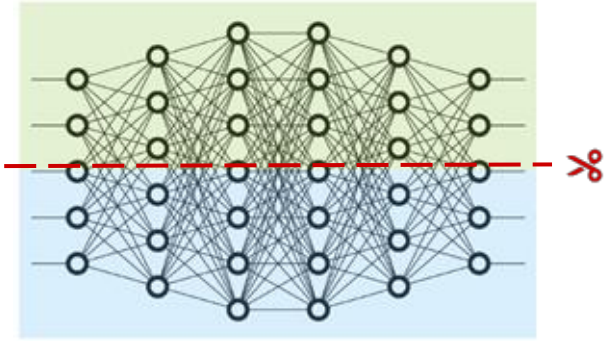
TECHNOLOGIES THAT ENABLE SCALING LARGE MODELS

Complementary Types of Parallelism

- Pipeline (Inter-Layer) Parallelism
  - Split contiguous sets of layers across multiple GPUs
  - Layers 0,1,2 and layers 3,4,5 are on different GPUs
  - *Maximizes GPU utilization in single-node*



- Tensor (Intra-Layer) Parallelism
  - Split individual layers across multiple GPUs
  - Both devices compute different parts of Layers 0,1,2,3,4,5
  - *Minimizes Latency in single-node*

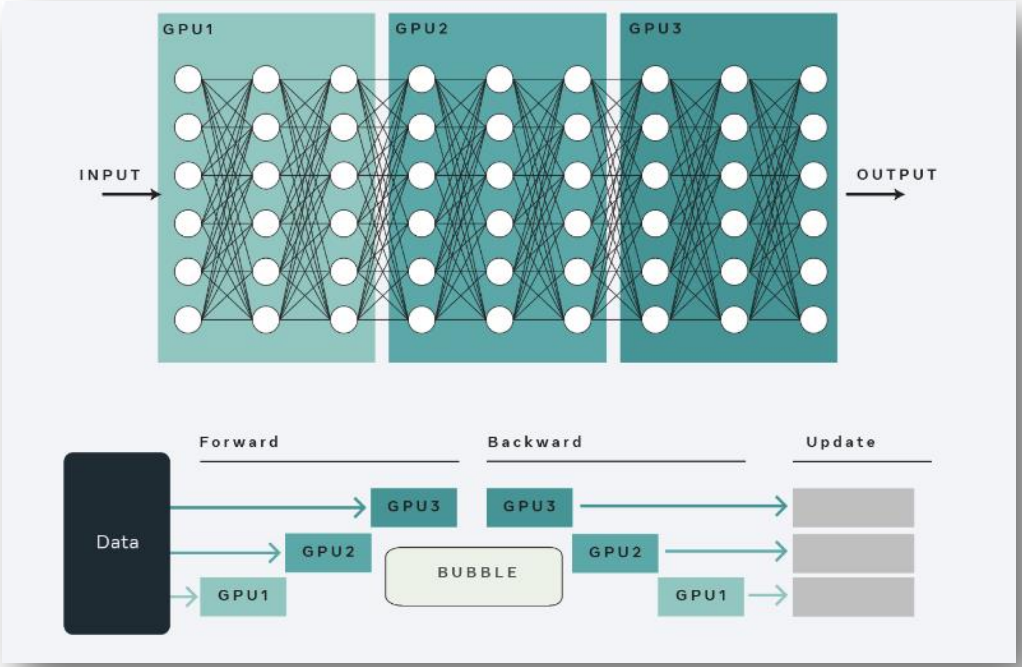


https://docs.nvidia.com/deeplearning/nemo/user-guide/docs/en/main/nlp/nemo_megatron/parallelisms.html

https://developer.nvidia.com/blog/scaling-language-model-training-to-a-trillion-parameters-using-megatron/

PIPELINE PARALLELISM

PIPELINE PARALLELISM

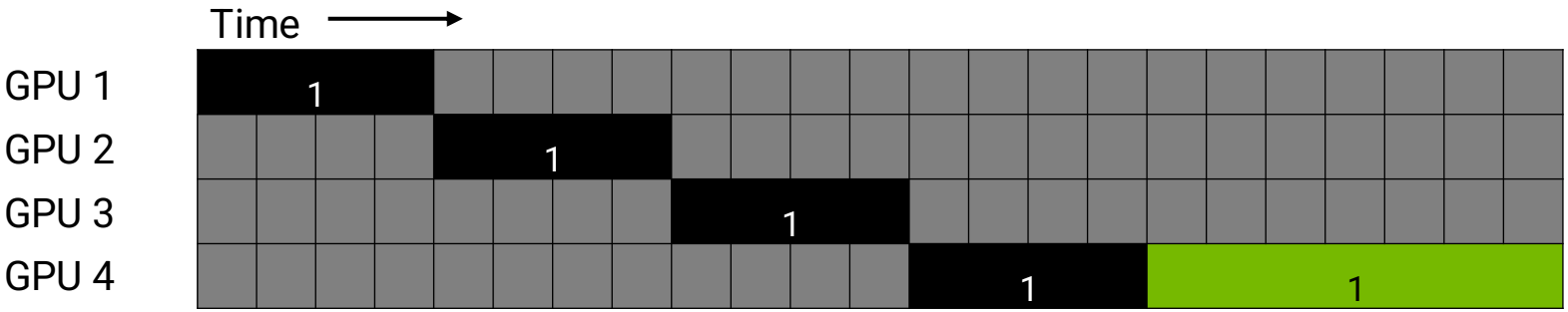


Pipeline parallel is sequentially processed as groups of continuous model layers, leading to GPU underutilization (bubbles)

For more details see: Fairscale Pipeline Parallelism

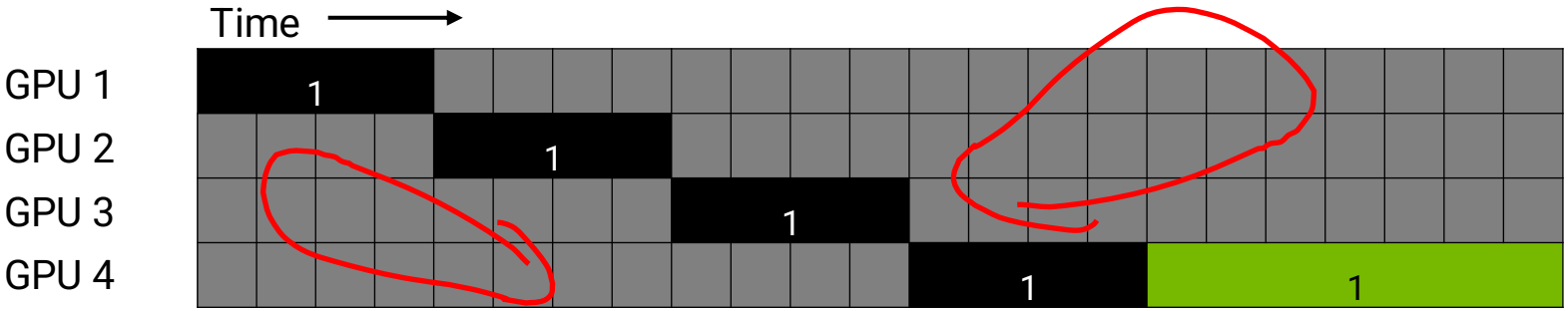
PIPELINE PARALLELISM

Challenges



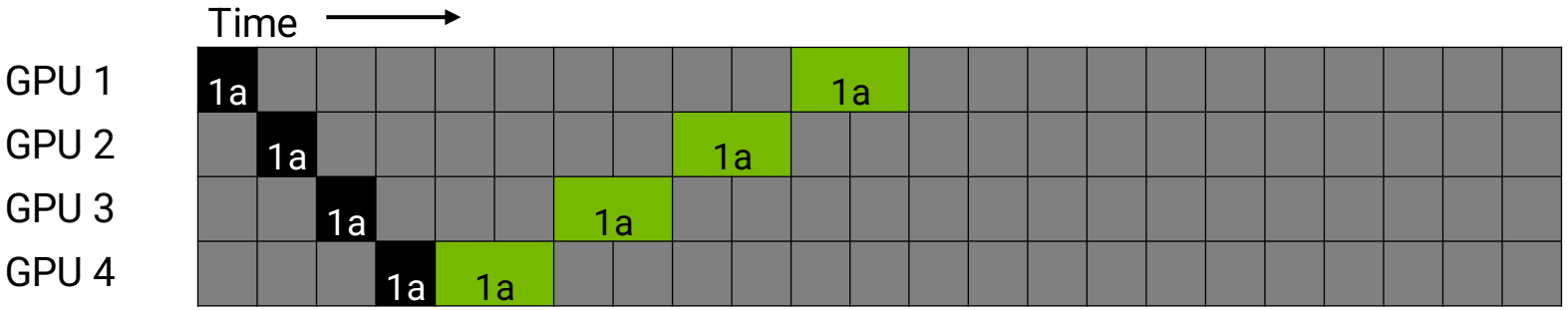
PIPELINE PARALLELISM

Challenges – Idle Workers



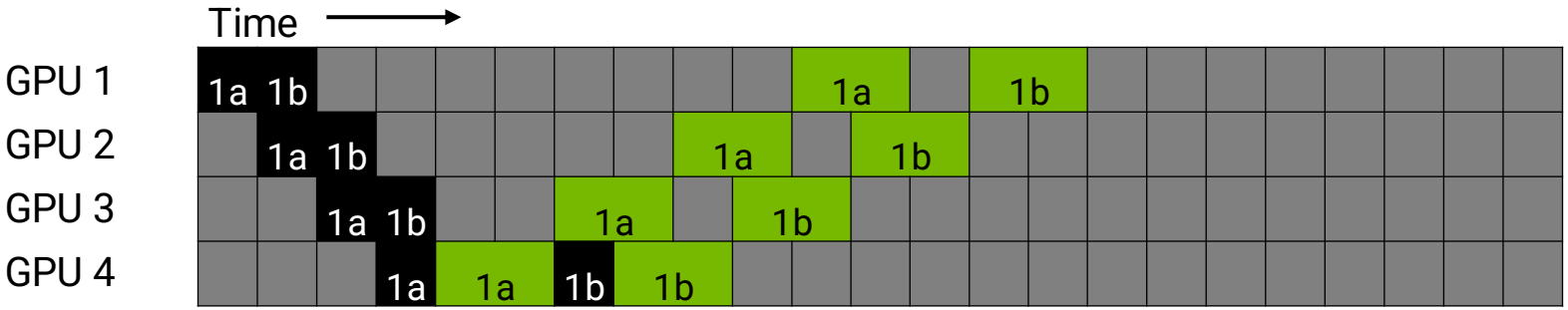
PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution



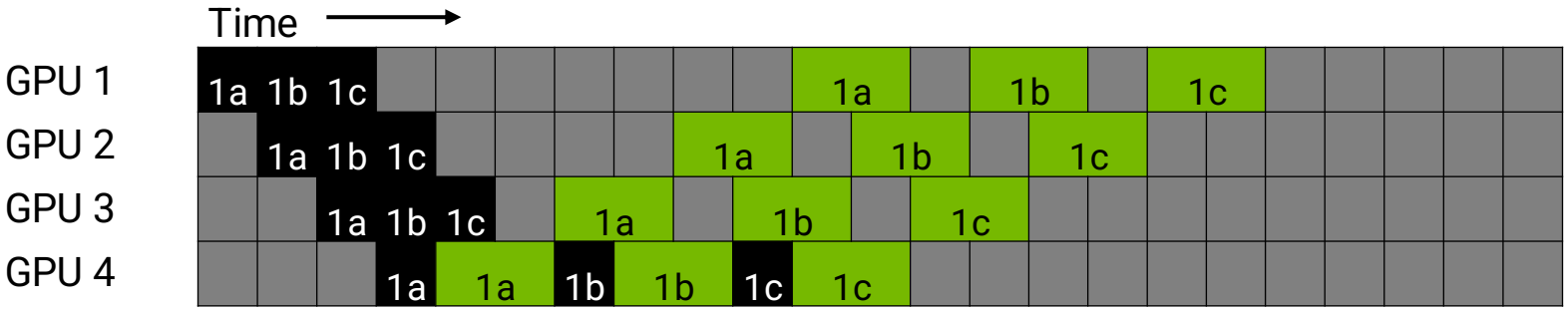
PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution



PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution

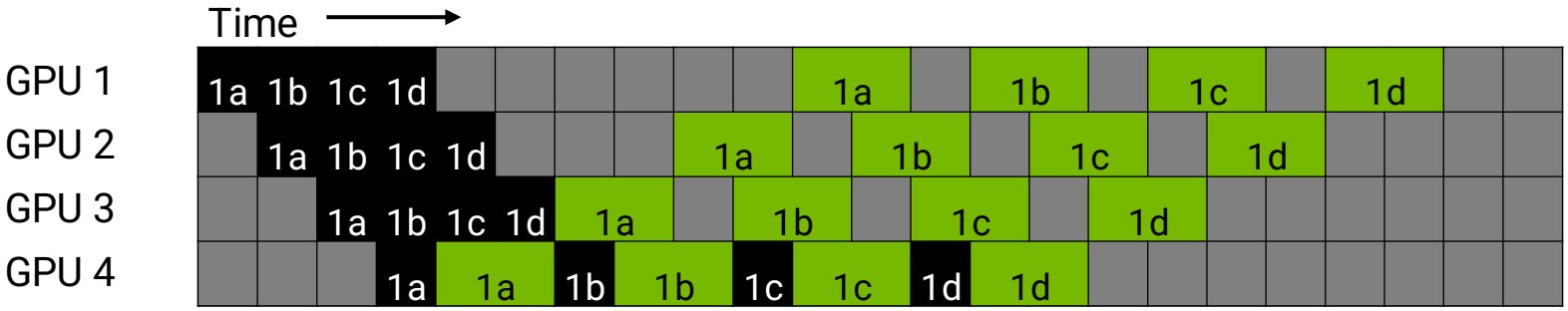


Forward pass

Backward pass

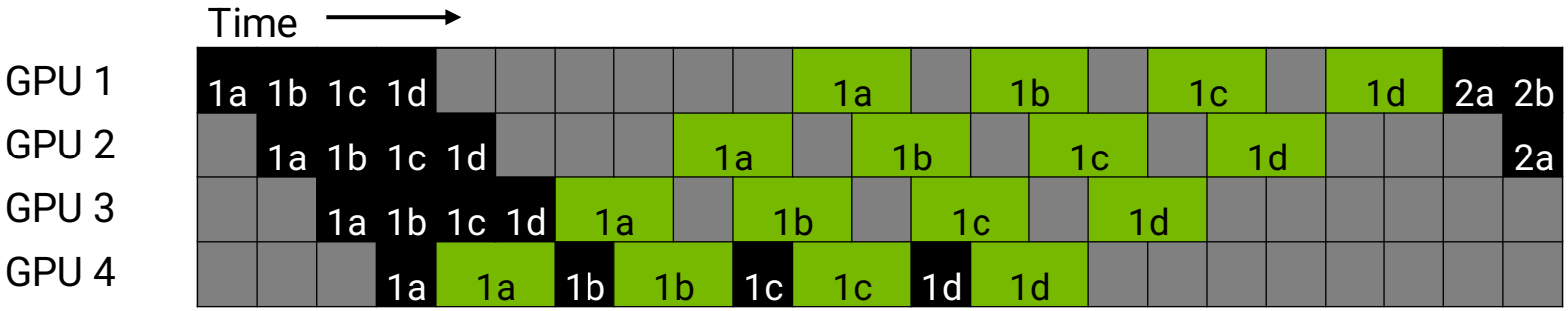
PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution



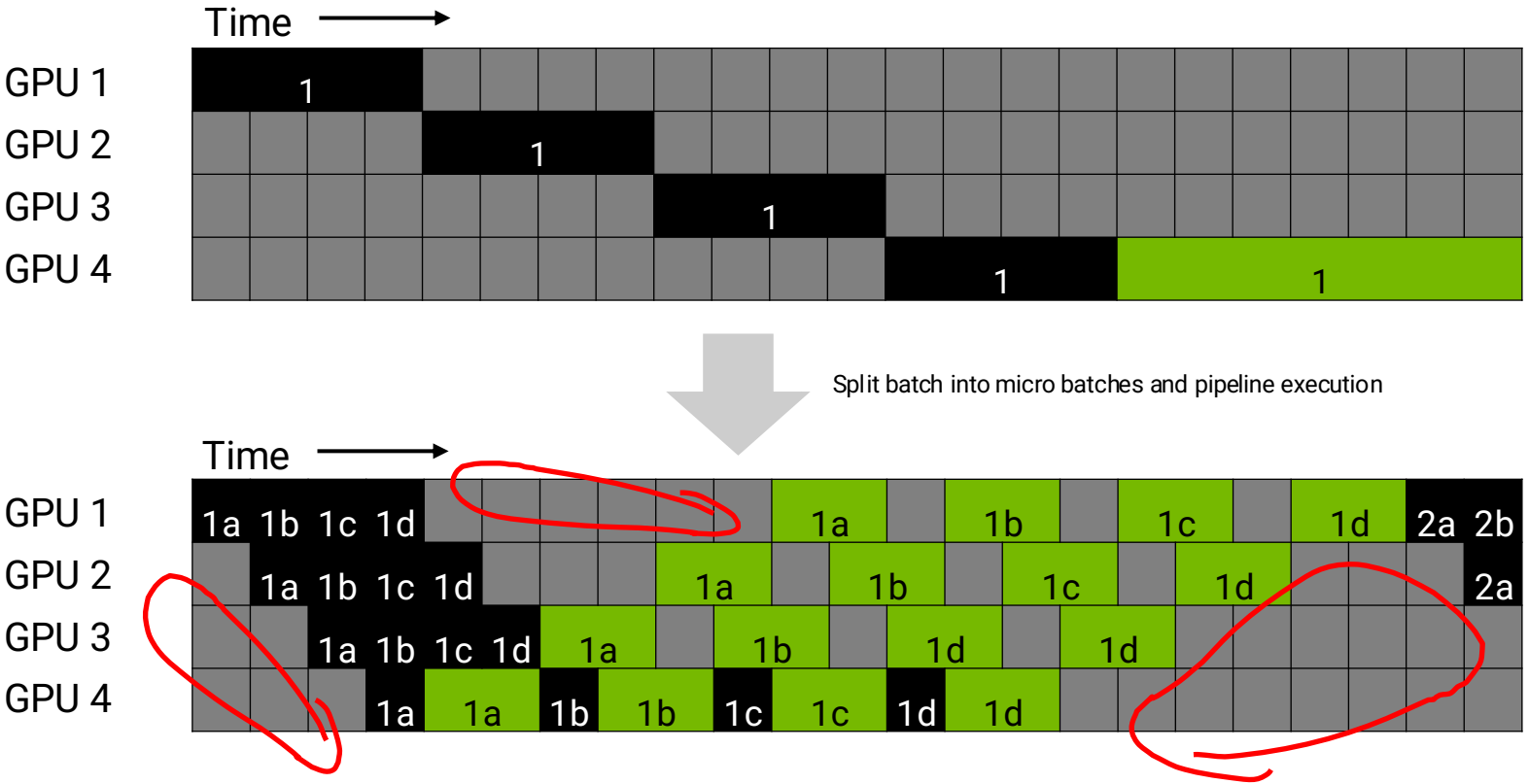
PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution



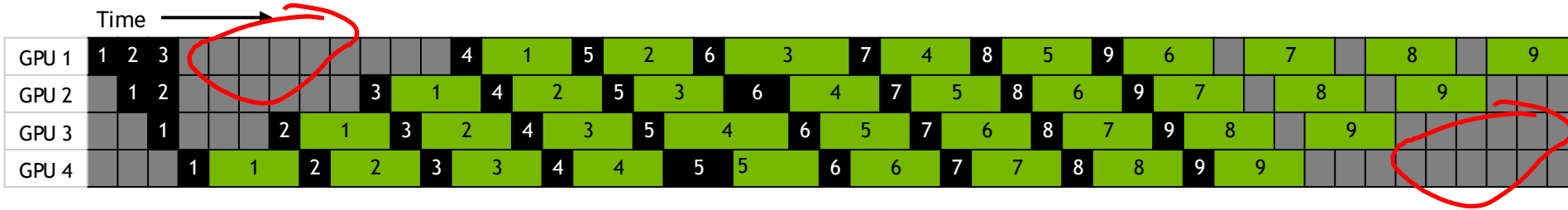
PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution



PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution



$$\text{total time} = (m + p - 1) \times (t_f + t_b)$$
$$\text{ideal time} = m \times (t_f + t_b)$$
$$\text{bubble time} = (p - 1) \times (t_f + t_b)$$

$$\text{bubble time overhead} = \frac{\text{bubble time}}{\text{ideal time}} = \frac{p - 1}{m}$$

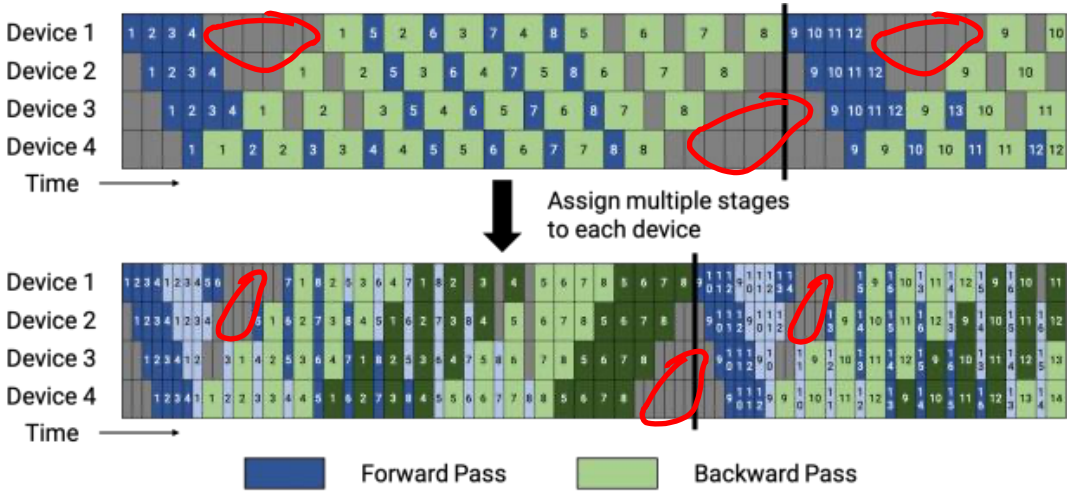
p : number of pipeline stages
m : number of micro batches
t_f : forward step time
t_b : backward step time

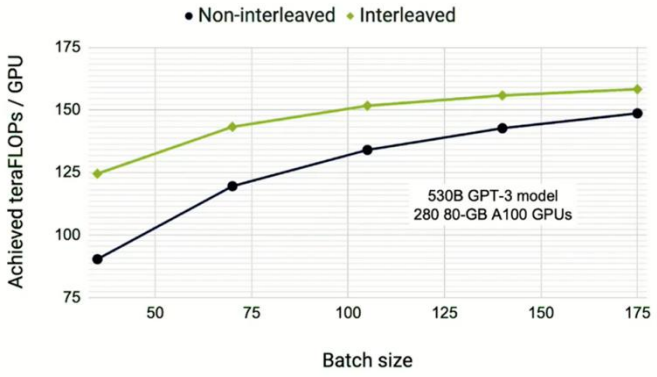
Forward pass

Backward pass

PIPELINE PARALLELISM

Interleaved Pipeline





Each gpu has groups of layers that are not continuous across groups, reducing bubbles

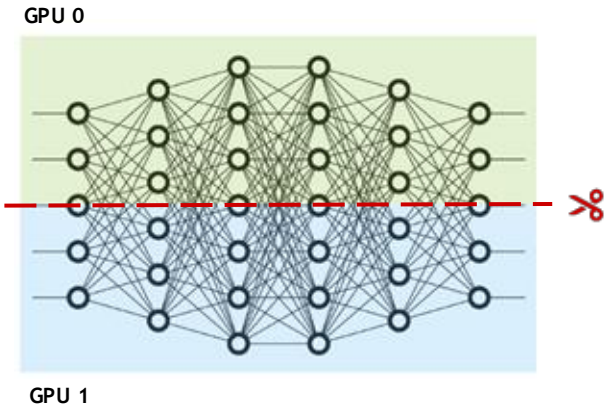
ous set of layers. For example, if each device had 4 layers before (i.e., device 1 had layers 1 – 4, device 2 had layers 5 – 8, and so on), we could have each device perform computation for two model chunks (each with 2 layers), i.e., device 1 has layers 1, 2, 9, 10; device 2 has layers 3, 4, 11, 12; and so on. With this scheme, each device in the pipeline is assigned multiple pipeline stages (each pipeline stage has less computation compared to before).

Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, Matei Zaharia. 2021

TENSOR PARALLELISM

TENSOR PARALLELISM

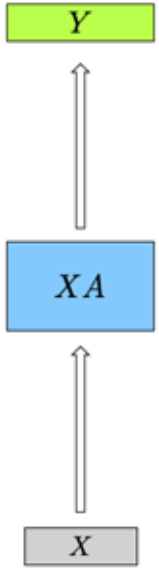
Why?

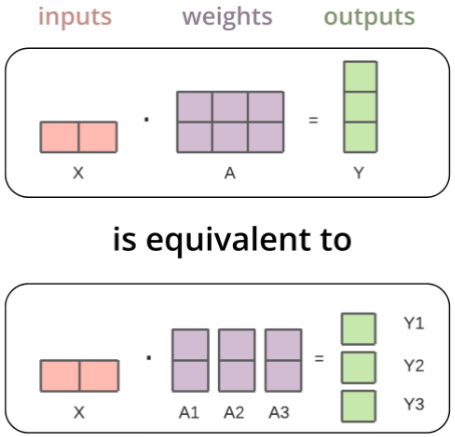


- Use to scale beyond data parallelism
- Typically used when model can't fit on a single GPU
- Reduces memory proportional to the number of workers (model dependent)
- Requires high compute-bandwidth to overcome communication overhead
  - NVIDIA DGX servers with NVSwitch (600GB/sec on DGX A100)
- Easier to load-balance (used to reduce latency for inference)
- Less restrictive on the batch-size (avoids bubble issue in pipelining)
- Tensor parallelism works well for large matrices (e.g. Transformers)
- Doesn't scale well beyond node boundary

Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model

SIMPLE EXAMPLE OF TENSOR PARALLELISM



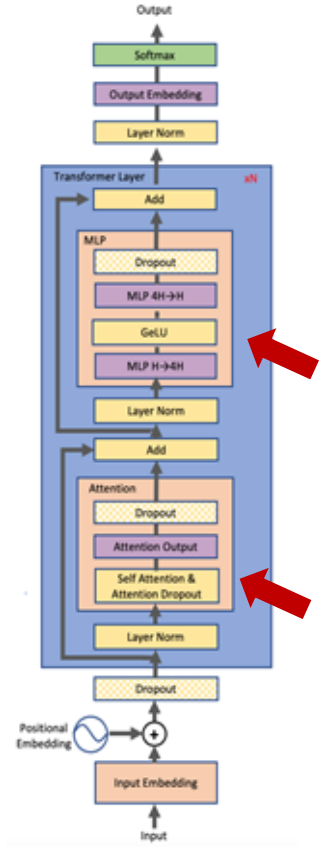


Split the XA matrix multiplications along the columns of A, column parallel split

[Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model]

TRANSFORMERS CELL EXAMPLE

TRANSFORMERS CELL



MLP TENSOR PARTITIONING

Focus on the GeLU operation:

- **Approach 1:** Split X column-wise and A row-wise:

$$X = [X_1, X_2] \quad A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \Rightarrow Y = \mathrm{GeLU}(X_1 A_1 + X_2 A_2)$$

  - Before GeLU we will need a communication point

- **Approach 2:** Split A column-wise:

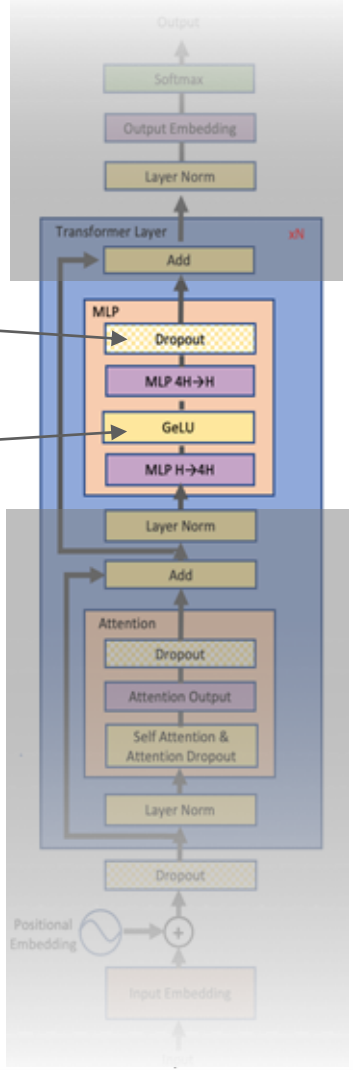
$$A = [A_1, A_2] \Rightarrow [Y_1, Y_2] = [\mathrm{GeLU}(XA_1), \mathrm{GeLU}(XA_2)]$$

  - No communication is required

Chosen approach

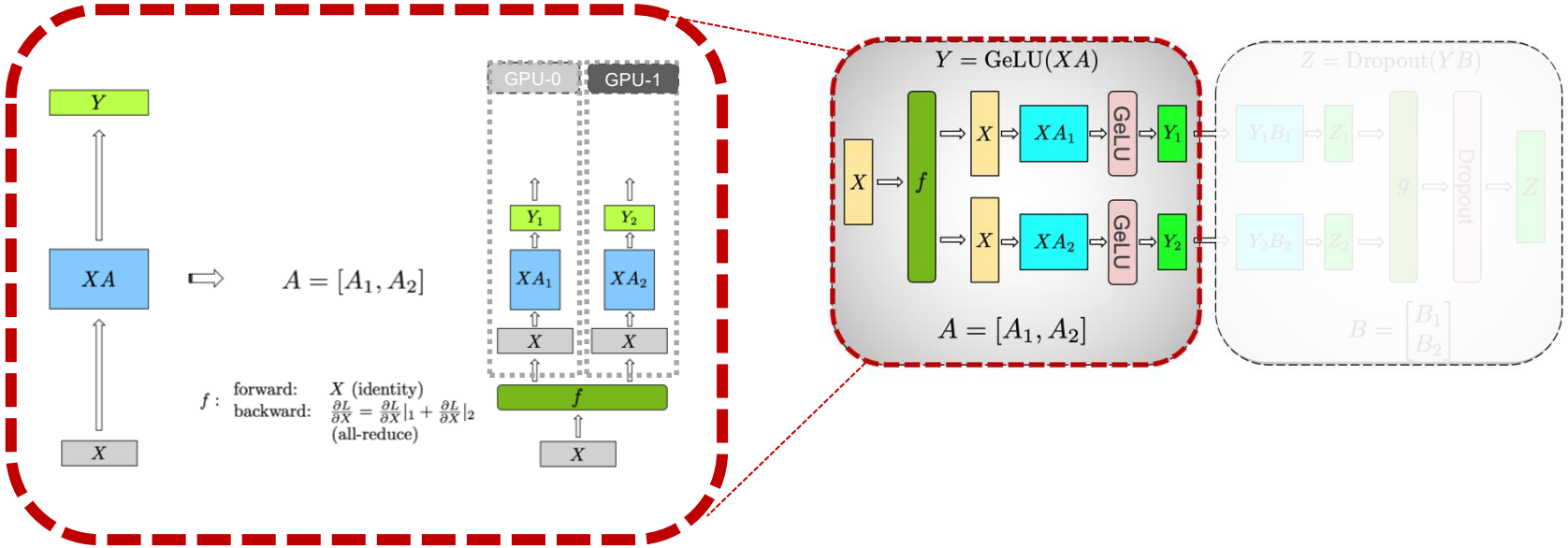
$$Z = \mathrm{Dropout}(YB)$$

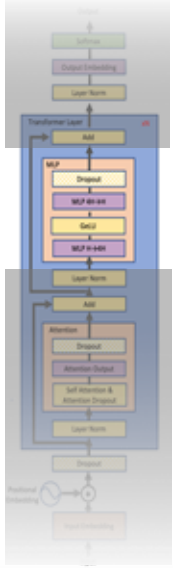
$$Y = \mathrm{GeLU}(XA)$$



MLP TENSOR PARTITIONING

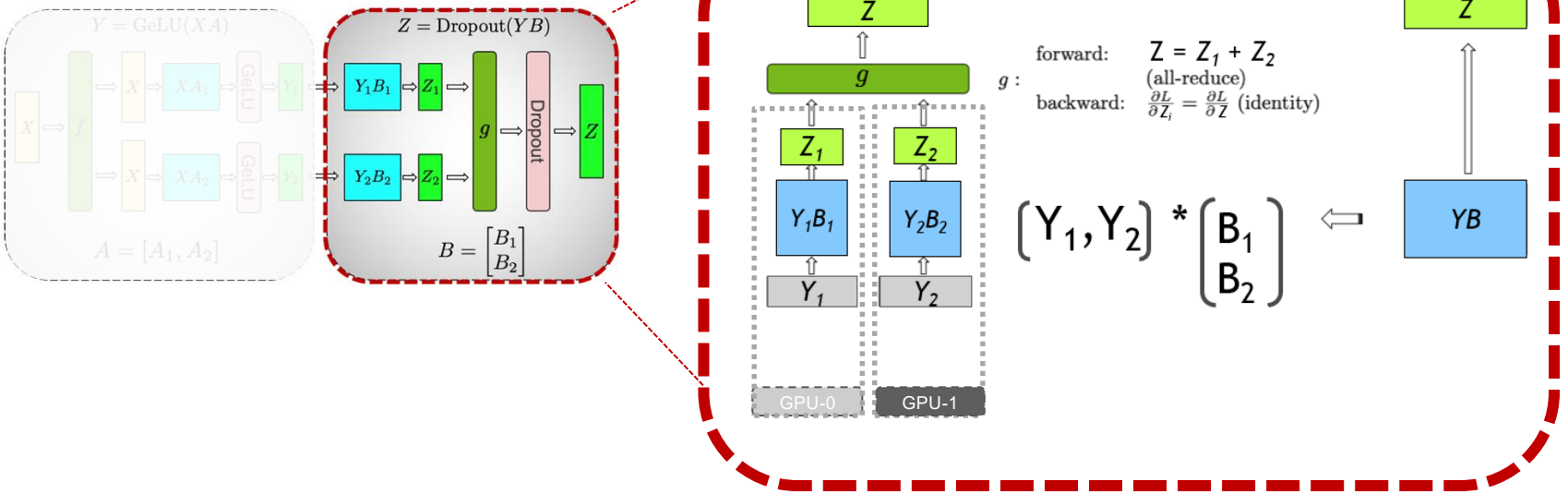
GeLU Column Parallel





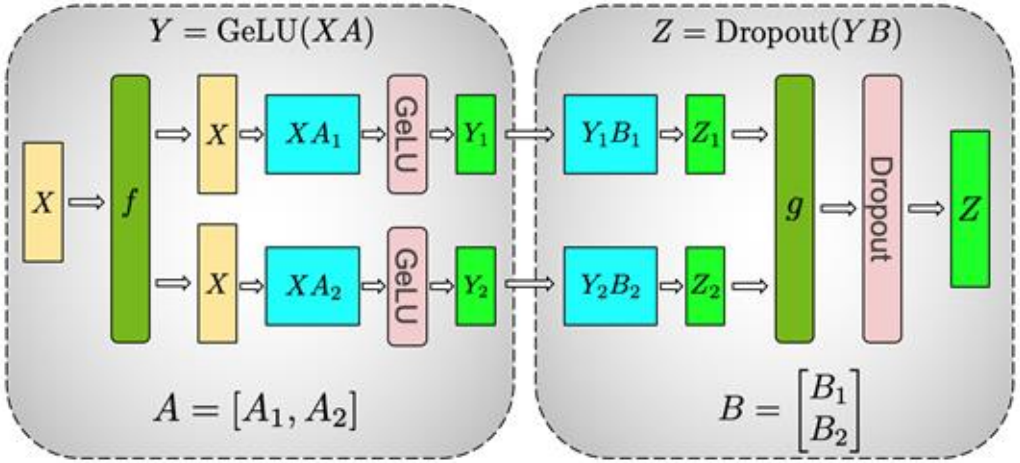
MLP TENSOR PARTITIONING

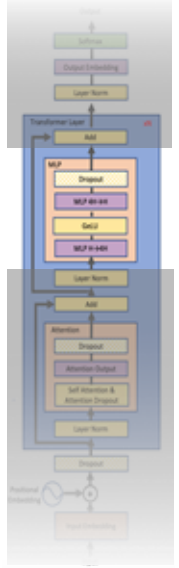
Dropout Row Parallel



MLP TENSOR PARTITIONING

GeLU Column Parallel

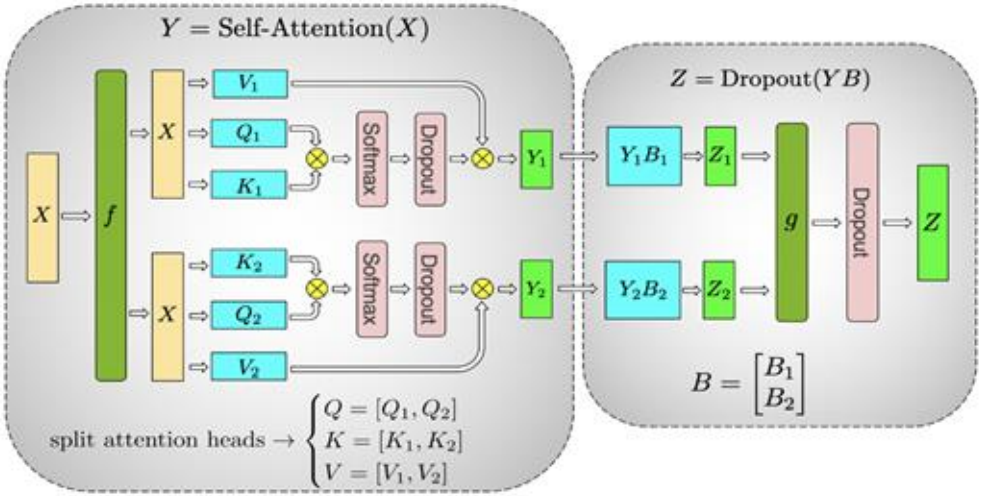




f and ***g*** are **conjugate**, ***f*** is **identity** operator in the forward pass and **all-reduce** in the backward pass while ***g*** is **all-reduce** in forward and **identity** in backward.

SELF-ATTENTION TENSOR PARTITIONING

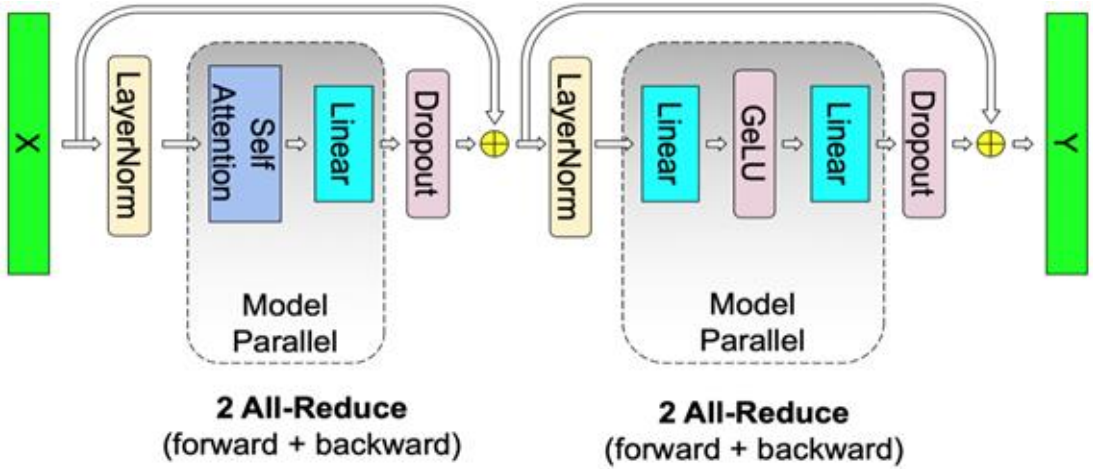
Same mechanism applies



f and ***g*** are **conjugate**, ***f*** is **identity** operator in the forward pass and **all-reduce** in the backward pass while ***g*** is **all-reduce** in forward and **identity** in backward.

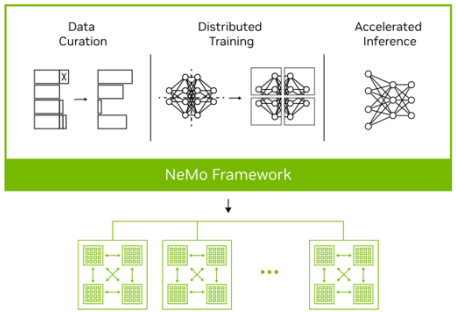
TENSOR PARALLEL TRANSFORMER LAYER

All Together



TENSOR PARALLEL IMPLEMENTATIONS

Libraries examples



deepspeed

Mesh TensorFlow

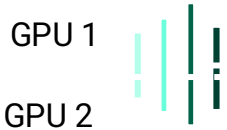
FairScale

Colossal-AI

PIPELINE AND TENSOR PARALLELISM

Recap

Tensor Parallelism



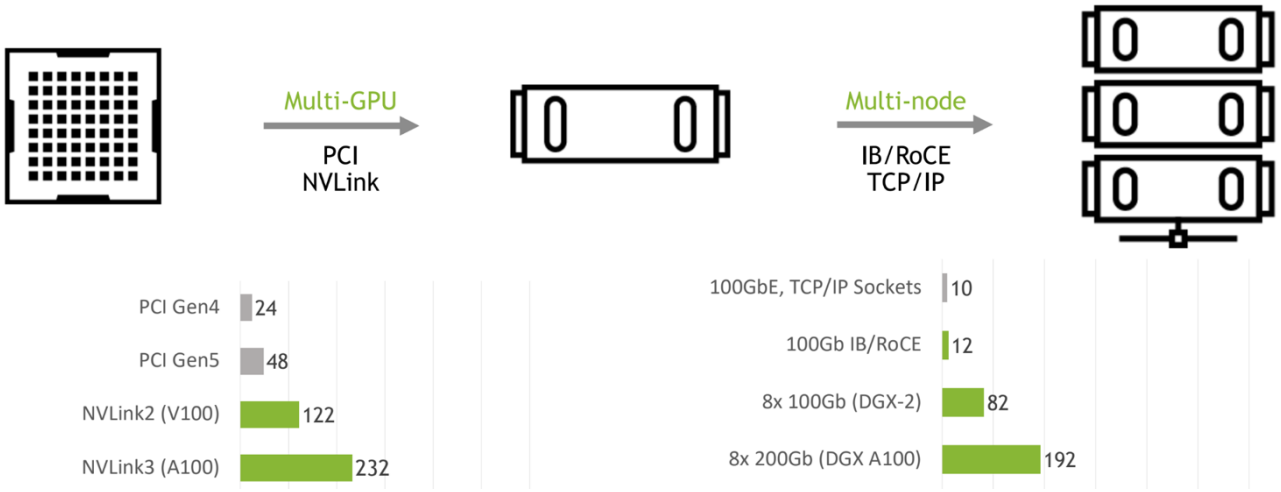
- Split individual layers across multiple GPUs where all devices compute different parts of Layers
- Challenge: Communication expensive
- Great performance within a server using NVSwitch
- Limitations: Limited number of Model Architectures | GPT-3 & T5

Pipeline Parallelism

- Split contiguous groups of layers across multiple GPUs so that Layers 0,1,2 and layers 3,4,5 are on different GPUs ...
- Communication cheap, maximizes GPU utilization over InfiniBand
- Good performance at larger batch sizes (pipeline stall amortized)
- Exceptions/Limitations: No Interleave Scheduling for Pipeline parallelism

TENSOR PARALLEL TRANSFORMER LAYER

Communication Expensive

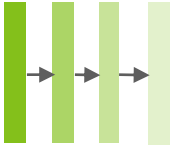


NCCL tests Allreduce Bus Bandwidth in GB/s.

HYBRID MODEL PARALLELISM

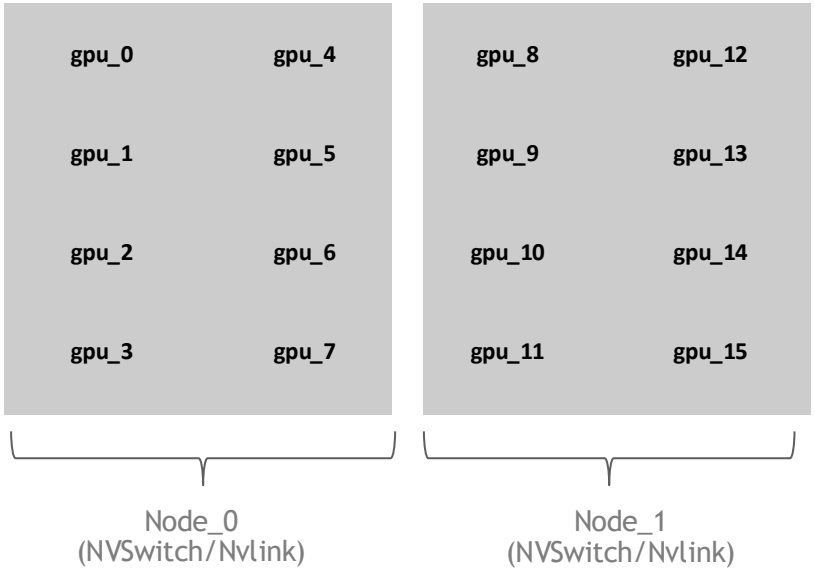
NEMO MODEL PARALLELISM

GPU Affinity Grouping Example



Neural Network: 4 layers
Hardware: 2 nodes , 8 GPUs per node

- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2

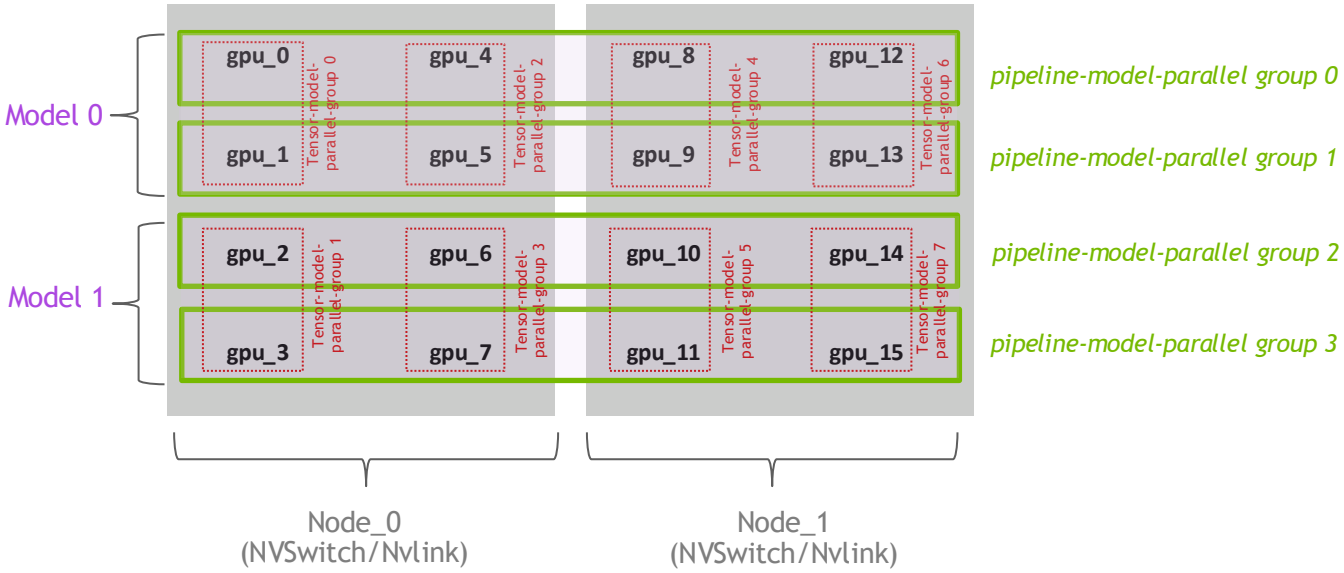


https://docs.nvidia.com/deeplearning/nemo/user-guide/docs/en/main/nlp/nemo_megatron/parallelisms.html

NEMO MODEL PARALLELISM

GPU Affinity Grouping Example

2 nodes , 8 GPUs per node
- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2

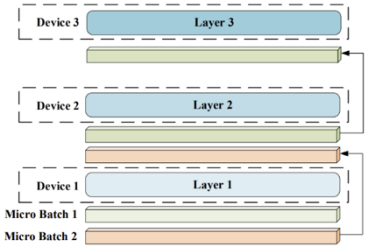


https://docs.nvidia.com/deeplearning/nemo/user-guide/docs/en/main/nlp/nemo_megatron/parallelisms.html

SEQUENCE PARALLELISM

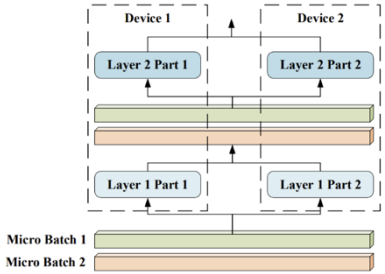
DEALING WITH MEMORY CONSTRAINTS

Sequence Parallelism – Train with longer sequences



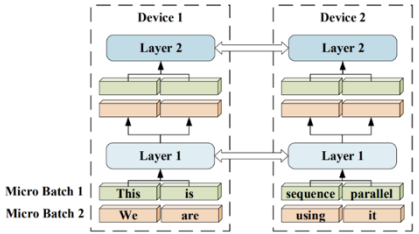
(a) Pipeline parallelism

- In Pipeline parallelism, groups of layers are place on GPUs (sequential execution)
- Whole sequence is needed for training



(b) Tensor parallelism

- In Tensor Parallelism, individual layers are split amongst GPUs
- Whole sequence is needed for training



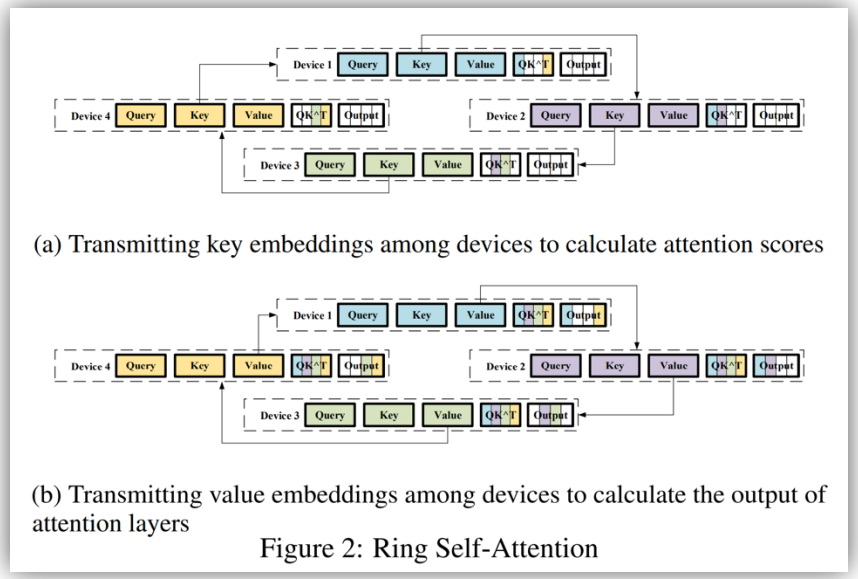
(c) Sequence parallelism (Ours)

- In Sequence Parallelism we distribute the sequence across gpus
- Devices share the same parameters

Sequence Parallelism: Long Sequence Training from System Perspective. Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, Yang You. 2022

RING SELF ATTENTION

Sequence Parallelism – Train with longer sequences



(a) Transmitting key embeddings among devices to calculate attention scores

(b) Transmitting value embeddings among devices to calculate the output of attention layers

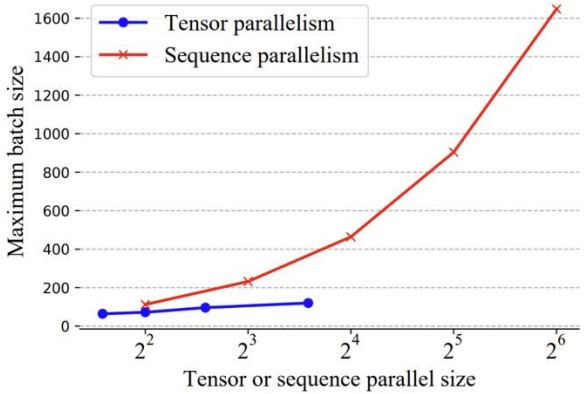
Figure 2: Ring Self-Attention

- Main challenge in sequence parallelism is calculating attention scores across devices
- Query, Key and Value (Q,K,V) matrices are split amongst the gpus
- Key embeddings are transferred among the GPUs
  - Query embeddings of each sub-sequence can multiply all key embeddings
- All query embeddings collect their attention scores on each device
- After attention scoring, Value embeddings are transferred among the GPUs to compute-self attention output.
- No communication in the MLP layer just self-attention module

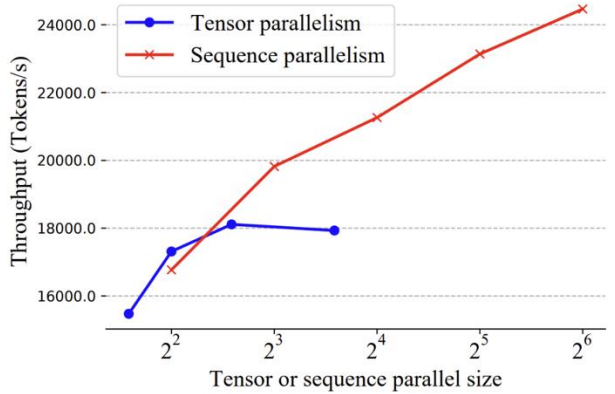
Sequence Parallelism: Long Sequence Training from System Perspective. Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, Yang You. 2022

DEALING WITH MEMORY CONSTRAINTS

Sequence Parallelism



(a) Maximum batch size of BERT Base scaling along tensor or sequence parallel size



(b) Throughput of BERT Base scaling along tensor or sequence parallel size

Figure 3: Scaling with sequence/tensor parallelism

Sequence parallelism allows you to get higher batch sizes and throughput during training by increasing the parallel size and same communication overhead as tensor parallelism

Sequence Parallelism: Long Sequence Training from System Perspective. Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, Yang You. 2022

DEALING WITH MEMORY CONSTRAINTS

Sequence parallelism

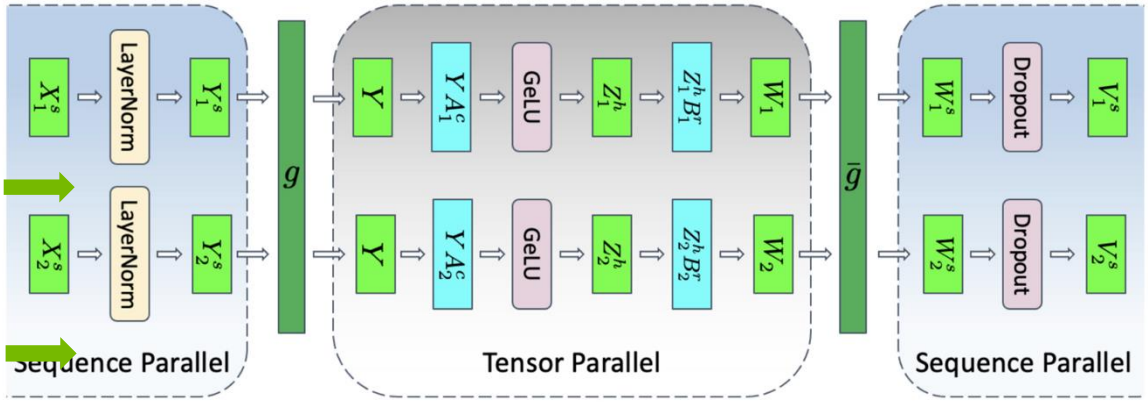


Figure 6: MLP layer with tensor and sequence parallelism. g and $\bar{g}$ are conjugate. g is all-gather in forward pass and reduce-scatter in backward pass. $\bar{g}$ is reduce-scatter in forward pass and all-gather in backward pass.

Tensor + Sequence Parallel has the same communication overhead as Tensor Parallel

Reducing Activation Recomputation in Large Transformer Models. Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, Bryan Catanzaro. 2022

ACTIVATIONS CHECKPOINTING

ACTIVATION RECOMPUTE CHALLENGES

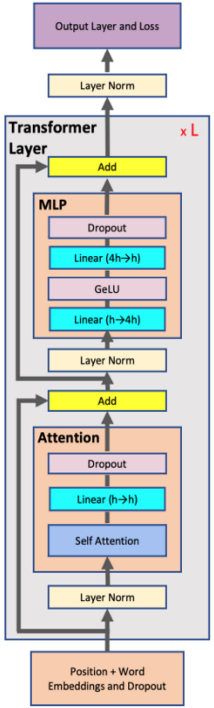
Activation Definition

Activation: Any tensor that is created in the forward pass and is necessary for gradient computation in the backward pass

Reducing Activation Recomputation in Large Transformer Models. Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, Bryan Catanzaro. 2022

ACTIVATION RECOMPUTE CHALLENGES

Activation Memory Per Transformer Layer



4.1 Activations Memory Per Transformer Layer

As shown in Figure 2, each transformer layer consists of an attention and an MLP block connected with two layer-norms. Below, we derive the memory required to store activations for each of these elements:

Attention block: which includes self attention followed by a linear projection and an attention dropout. The linear projection stores its input activations with size $2sbh$ and the attention dropout requires a mask with size sbh. The self attention shown in Figure 3 consists of several elements:

- **Query (Q), Key (K), and Value (V) matrix multiplies:** We only need to store their shared input with size $2sbh$.
- QK^T **matrix multiply:** It requires storage of both Q and K with total size $4sbh$.
- **Softmax:** Softmax output with size $2as^2b$ is required for back-propagation.
- **Softmax dropout:** Only a mask with size as^2b is needed.
- **Attention over Values (V):** We need to store the dropout output ($2as^2b$) and the Values ($2sbh$) and therefore need $2as^2b + 2sbh$ of storage.

Summing the above values, in total, the attention block requires $11sbh + 5as^2b$ bytes of storage.

MLP: The two linear layers store their inputs with size $2sbh$ and $8sbh$. The GeLU non-linearity also needs its input with size $8sbh$ for back-propagation. Finally, dropout stores its mask with size sbh. In total, MLP block requires $19sbh$ bytes of storage.

Layer norm: Each layer norm stores its input with size $2sbh$ and therefore in total, we will need $4sbh$ of storage.

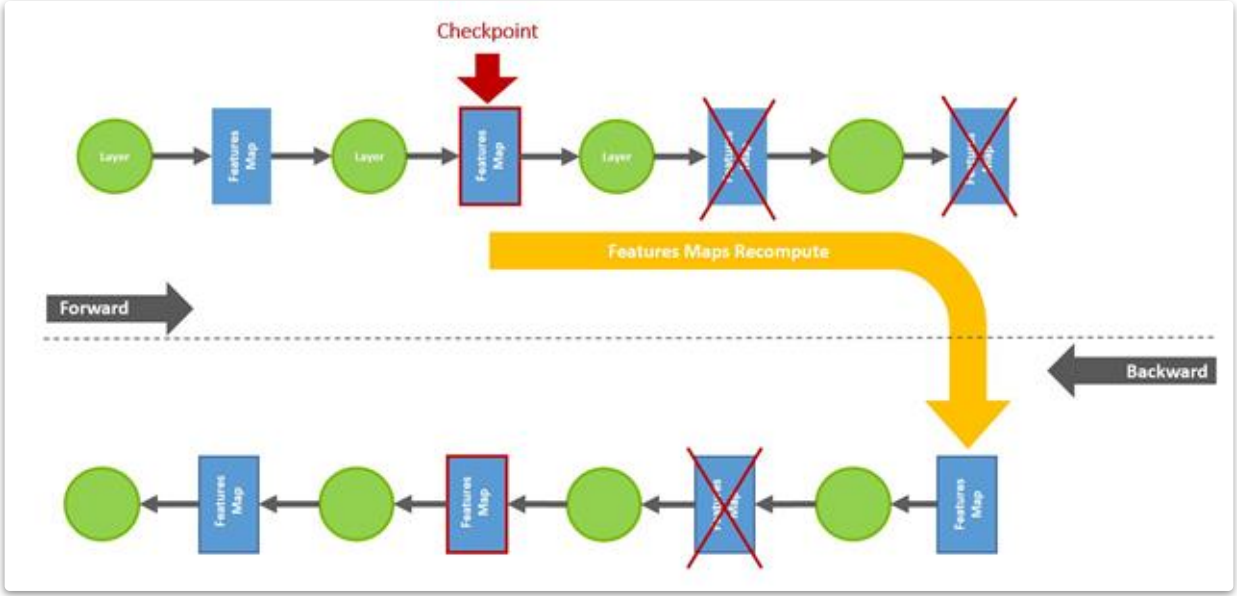
Summing the memory required for attention, MLP, and the layer-norms, the memory required to store the activations for a single layer of a transformer network is:

$$\text{Activations memory per layer} = sbh\left(34 + 5\frac{as}{h}\right). \quad (1)$$

Reducing Activation Recomputation in Large Transformer Models. Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, Bryan Catanzaro. 2022

ACTIVATION RECOMPUTE CHALLENGES

Activation Recompute



Balance the memory savings and computational overhead?

ACTIVATION RECOMPUTE CHALLENGES

Selective Activation Recompute with Megatron-LM

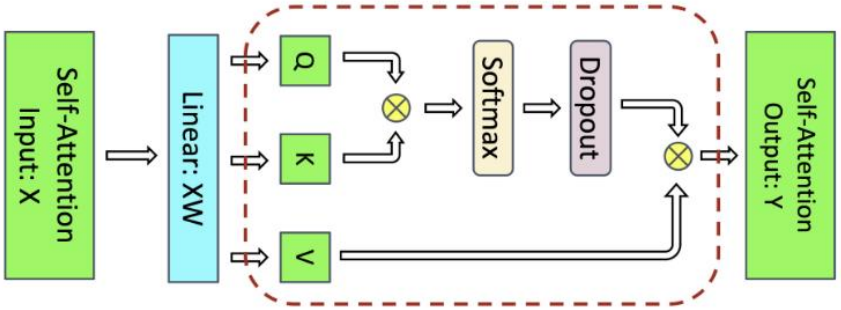


Figure 3: Self-attention block. The red dashed line shows the regions to which selective activation recomputation is applied (see Section 5 for more details on selective activation recomputation).

- Saves the activations that take less space and are expensive to recompute
- Recompute activations that take a lot of space but are relatively cheap to recompute.

Reducing Activation Recomputation in Large Transformer Models. Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, Bryan Catanzaro. 2022

DEALING WITH MEMORY CONSTRAINTS

Megatron Sequence parallelism + Selective Checkpointing

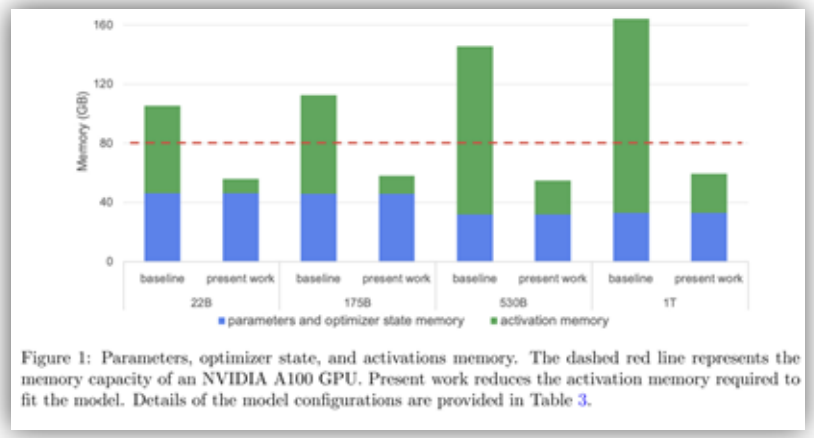


Figure 1: Parameters, optimizer state, and activations memory. The dashed red line represents the memory capacity of an NVIDIA A100 GPU. Present work reduces the activation memory required to fit the model. Details of the model configurations are provided in Table 3.

Model Size	Iteration Time (seconds)		Throughput Increase	Model FLOPs Utilization	Hardware FLOPs Utilization
	Full Recompute	Present Work			
22B	1.42	1.10	29.0%	41.5%	43.7%
175B	18.13	13.75	31.8%	51.4%	52.8%
530B	49.05	37.83	29.7%	56.0%	57.0%
1T	94.42	71.49	32.1%	56.3%	57.0%

Table 5: End-to-end iteration time. Our approach results in throughput increase of around 30%.

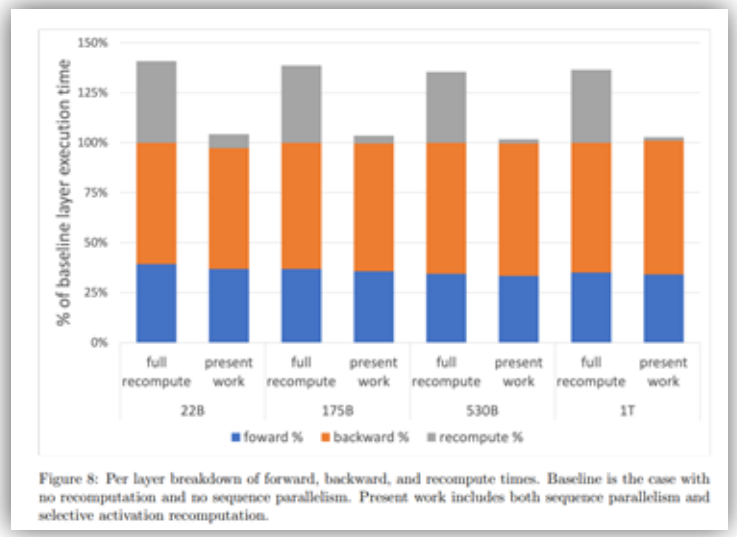


Figure 8: Per layer breakdown of forward, backward, and recompute times. Baseline is the case with no recomputation and no sequence parallelism. Present work includes both sequence parallelism and selective activation recomputation.

Reducing Activation Recomputation in Large Transformer Models. Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, Bryan Catanzaro. 2022

MIXTURE OF EXPERTS

MIXTURE OF EXPERTS

Not a new Idea - 1991

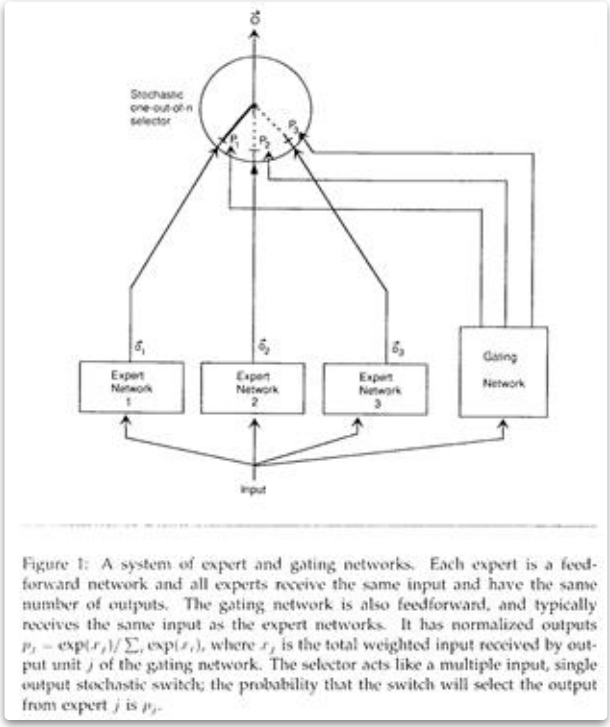


Figure 1: A system of expert and gating networks. Each expert is a feed-forward network and all experts receive the same input and have the same number of outputs. The gating network is also feedforward, and typically receives the same input as the expert networks. It has normalized outputs $p_i = \exp(x_i)/\sum_i \exp(x_i)$, where x_j is the total weighted input received by output unit j of the gating network. The selector acts like a multiple input, single output stochastic switch; the probability that the switch will select the output from expert j is p_j.

- Training a single, multilayer network to perform different subtasks is difficult
- Interaction effects slow learning and lead to poor generalization
- Prior knowledge of task separation is evidence for an expert system
- System is composed of a series of experts with a router (gating network)
- Weights of experts are decoupled
- Choice of objective function is critical – how to combine experts

Adaptive mixtures of local experts. R. A. Jacobs, M. I. Jordan, S. J. Nowlan, G. E. Hinton. Neural Computation, 3(1):79–87, 1991.

MIXTURE OF EXPERTS

MoE in Neural Networks (2017)

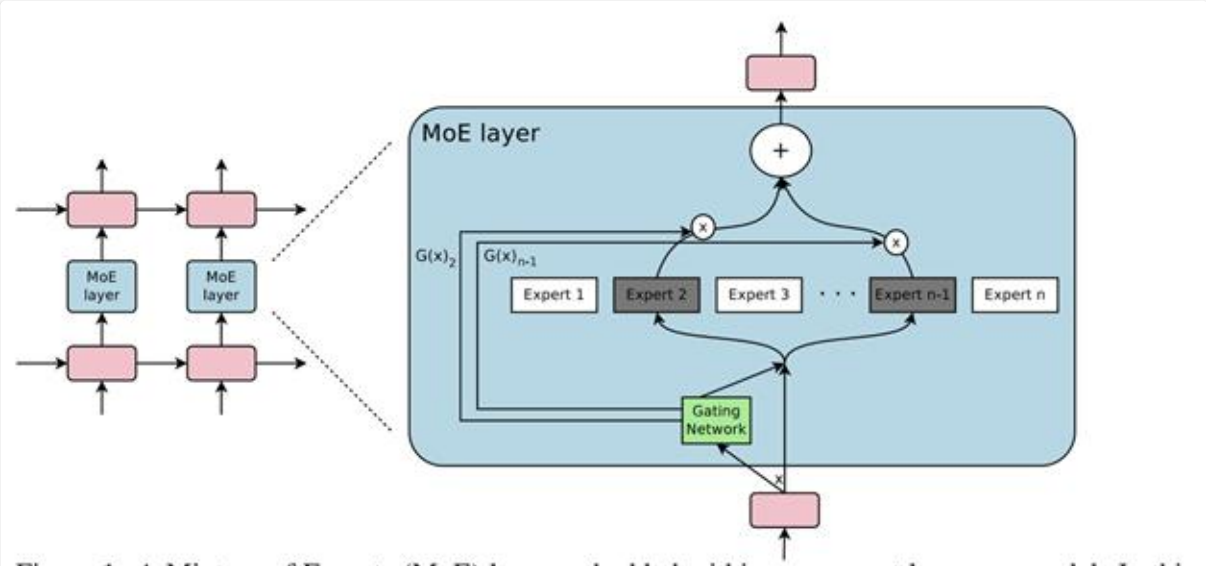


Figure 1: A Mixture of Experts (MoE) layer embedded within a recurrent language model. In this case, the sparse gating function selects two experts to perform computations. Their outputs are modulated by the outputs of the gating network.

Applied thousands of mixture of experts to language modelling inside of an RNN

Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, Jeff Dean. ICLR 2017

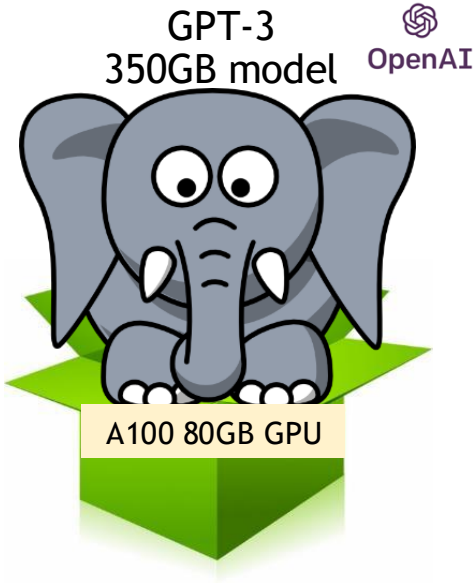
6

Inference Of Large Neural Networks

INFERENCE OF HUGE MODELS

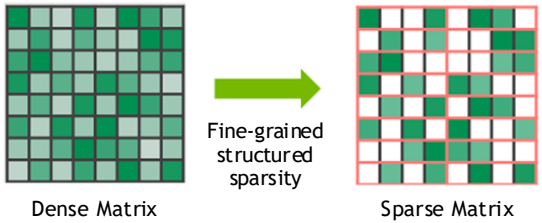
Goals and Challenges

- **Goal: To infer huge models in an efficient and convenient way, including**
  - Maximizing Utilization of GPUs
  - A unified and simple inference solution for many models in production
  - Easier deployments, scaling and support
  - Maximizing Throughput, Minimizing Latency
- **Challenges:**
  - Huge model requires more memory than available on 1 GPU
  - Model needs to be optimized before the inference
  - Frameworks used for training Huge Models are quite complex and inadequate for inference



PRODUCTION DEPLOYMENT

Pruning - 2:4 Structured Sparsity



175B parameters

2 bytes per weight
(FP16)

$$\frac{175 * 10^9 * 2}{1024 * 1024 * 1024 * 2} = 162.98GB$$

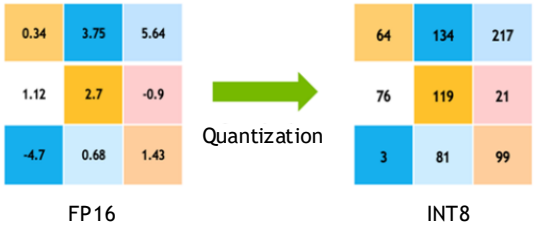
We want GPU memory consumption in GB

Remove half of the weights

https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/

PRODUCTION DEPLOYMENT

Quantization



175B parameters

1 byte per weight (INT8)

$$\frac{175 * 10^9 * 1}{1024 * 1024 * 1024 * 2} = 81.49GB$$

We want GPU memory consumption in GB

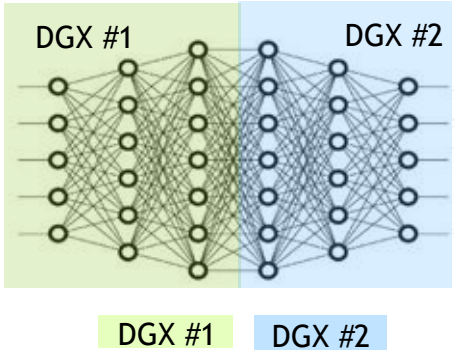
Remove half of the weights

https://developer.nvidia.com/blog/achieving-fp32-accuracy-for-int8-inference-using-quantization-aware-training-with-tensorrt/

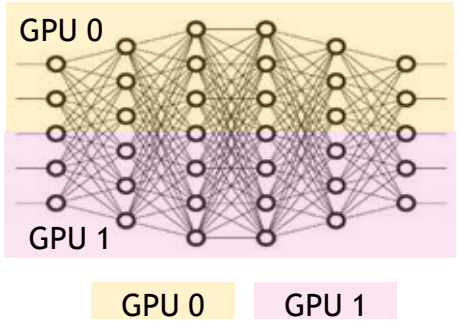
MODEL PARALLELISM

Complementary Types of Model Parallelism

- Inter-Layer (Pipeline) Parallelism
  - Split sets of layers across multiple devices
  - *Inference:*
    - *Maximizes GPU utilization and Throughput*
    - *Can be used easily with TRITON*

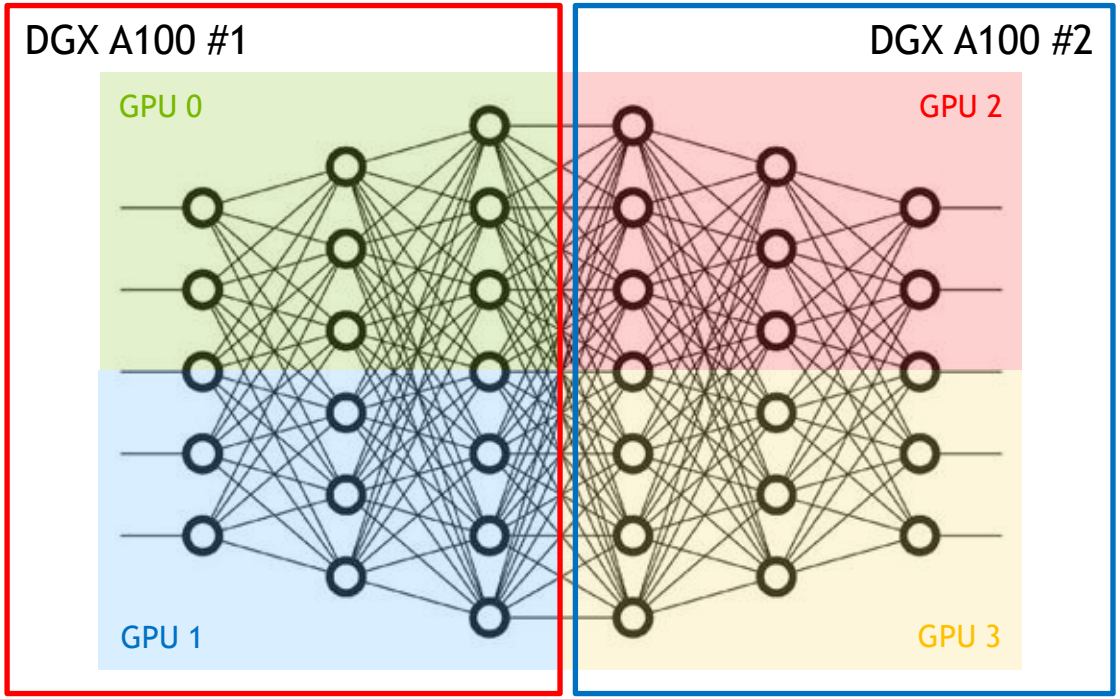


- Intra-Layer (Tensor) Parallelism
  - Split individual layers across multiple devices
  - *Inference:*
    - *Minimizes latency*



MODEL PARALLELISM

Combined Model Parallelism. Multiple GPUs in Multiple DGXs.



Inter + Intra Parallelism

LIVE DEMO:

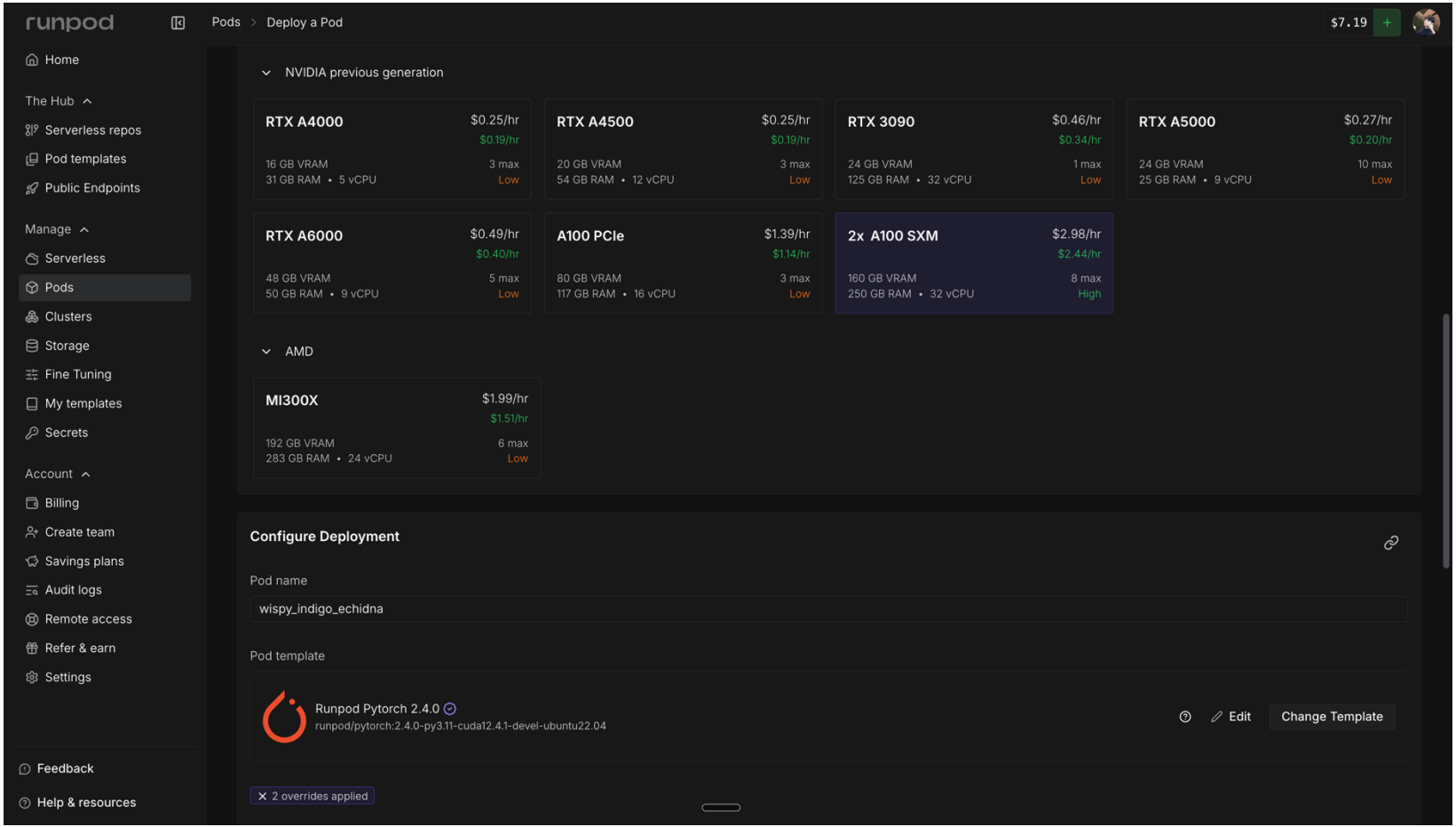
HOW DO YOU RUN A MODEL THAT DOESN'T FIT ON A GPU?

- MODEL: QWEN2.5 72B INSTRUCT (72 BILLION PARAMETERS)
- HARDWARE: 2X NVIDIA A100 80GB (SXM)

WE HAVE A PROBLEM: THIS MODEL IS 144GB, BUT EACH GPU ONLY HAS 80GB OF MEMORY.

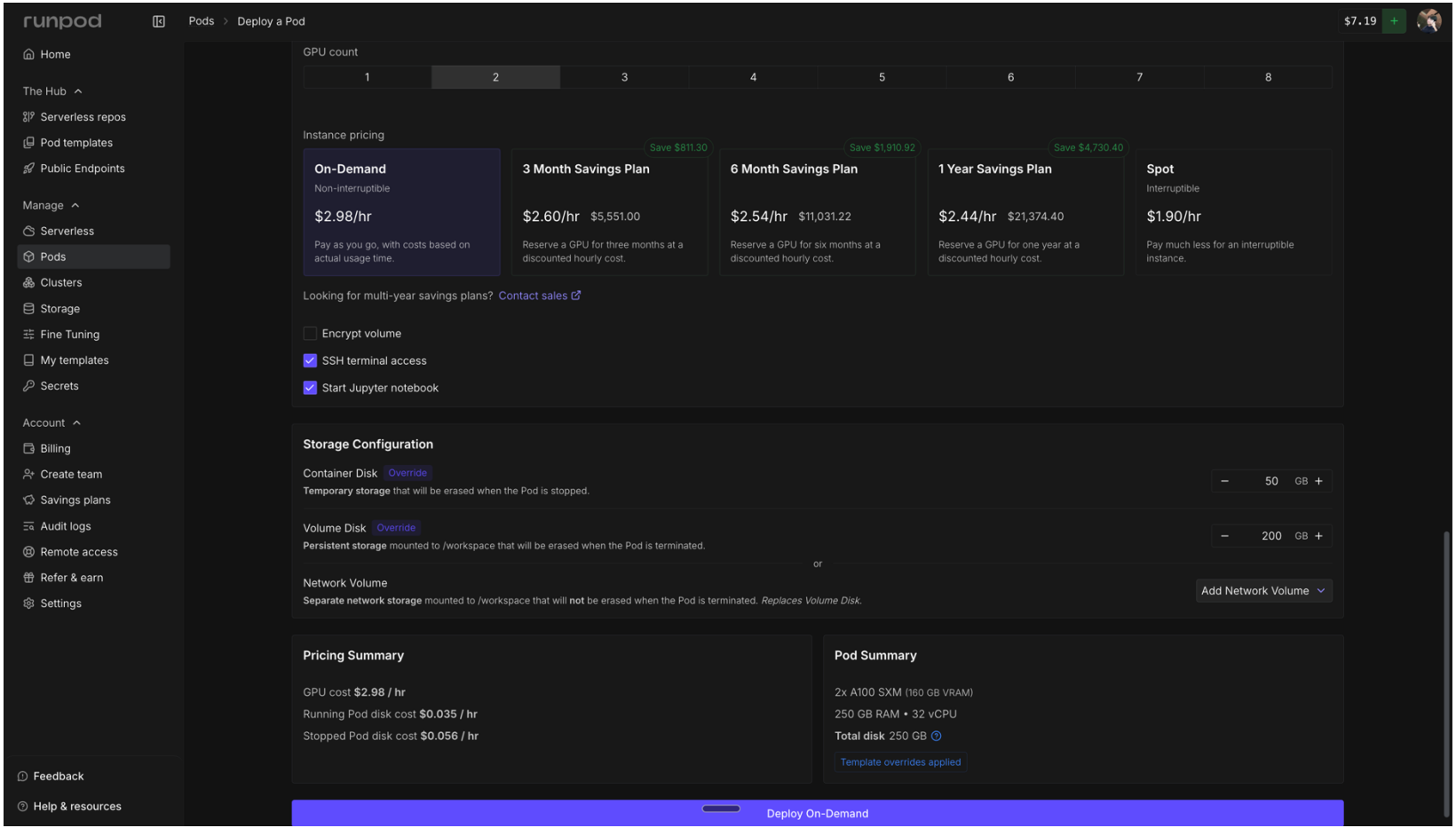
Setting Up GPUs

https://www.runpod.io/



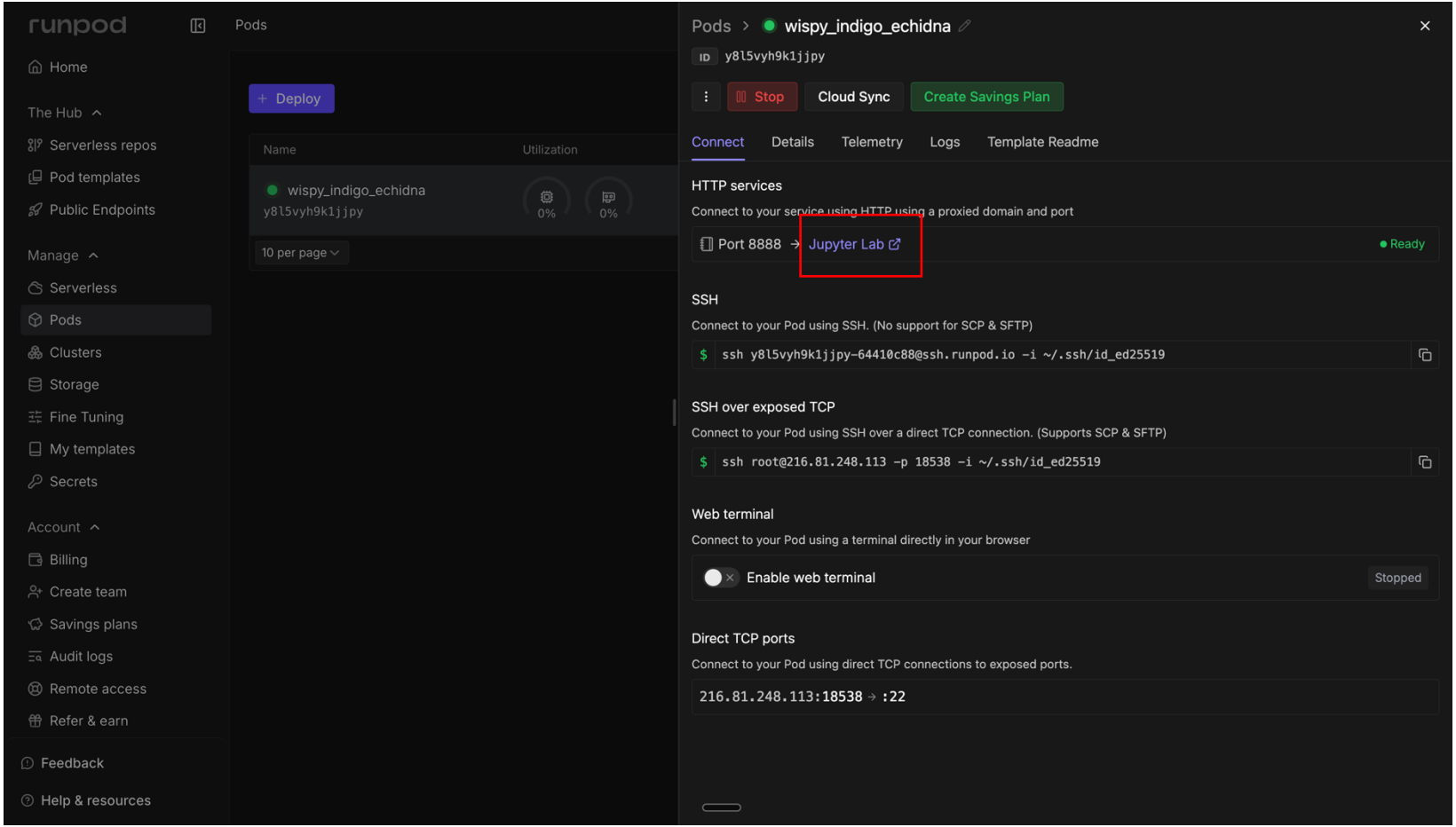
Setting Up GPUs

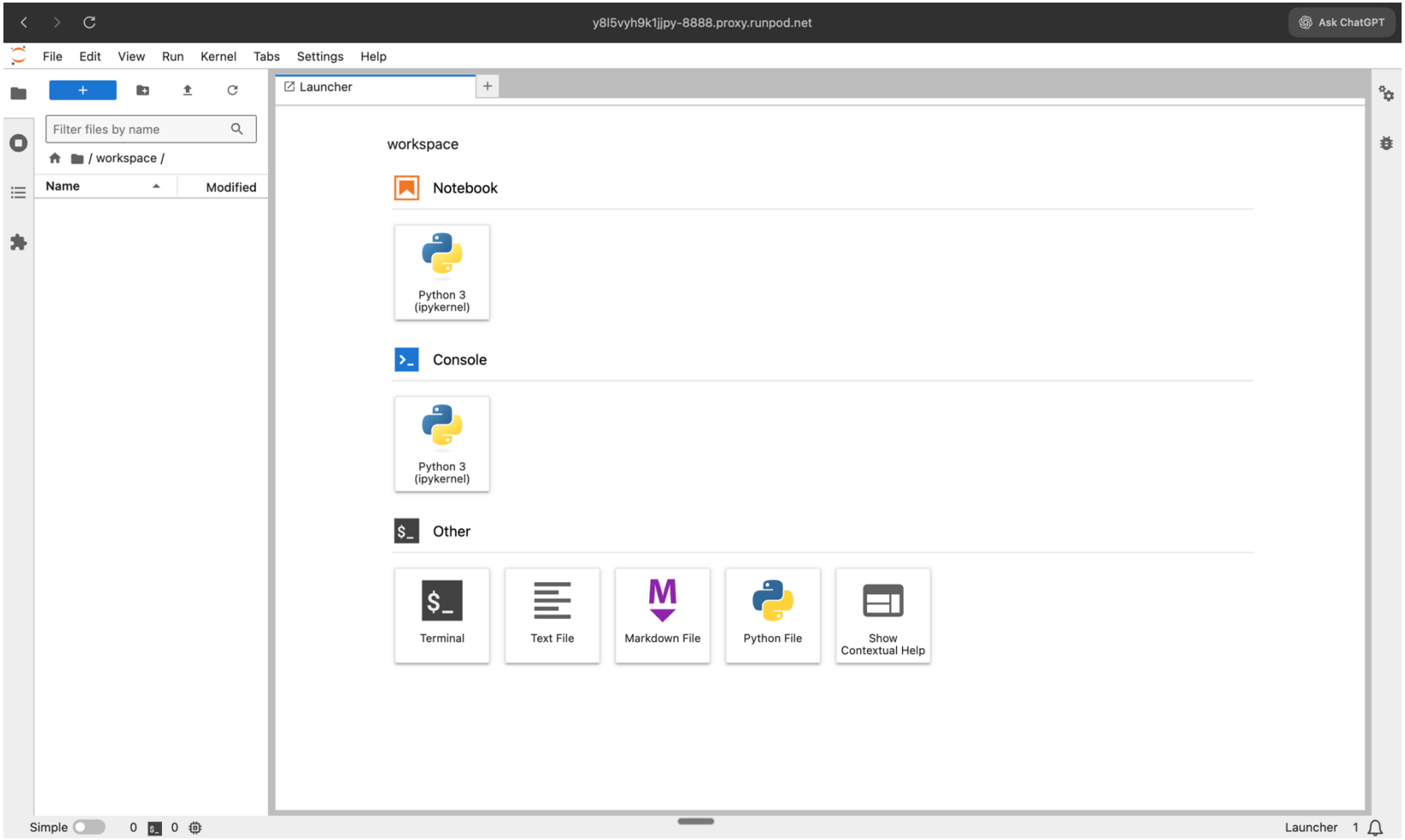
https://www.runpod.io/



Setting Up GPUs

https://www.runpod.io/

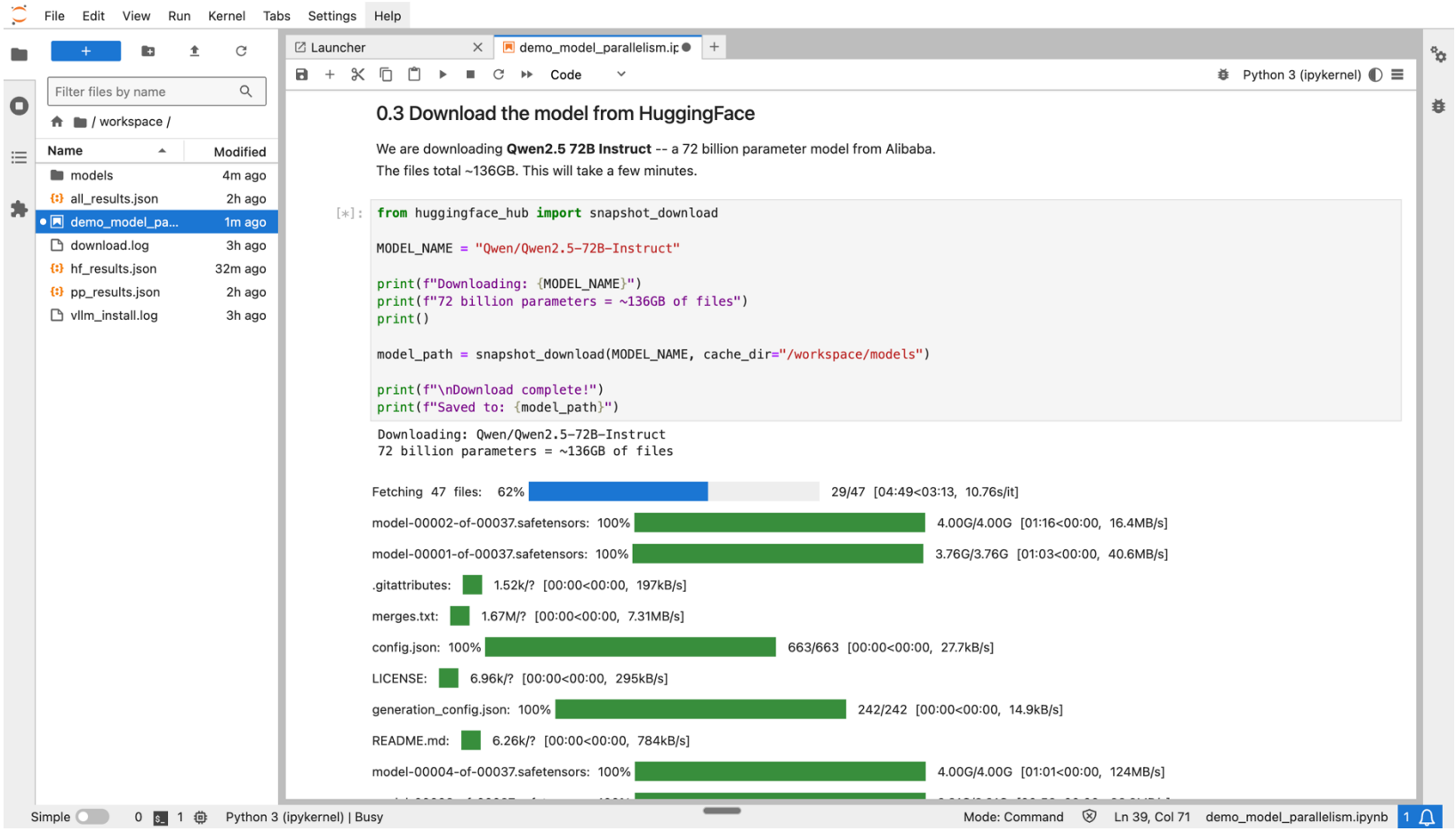


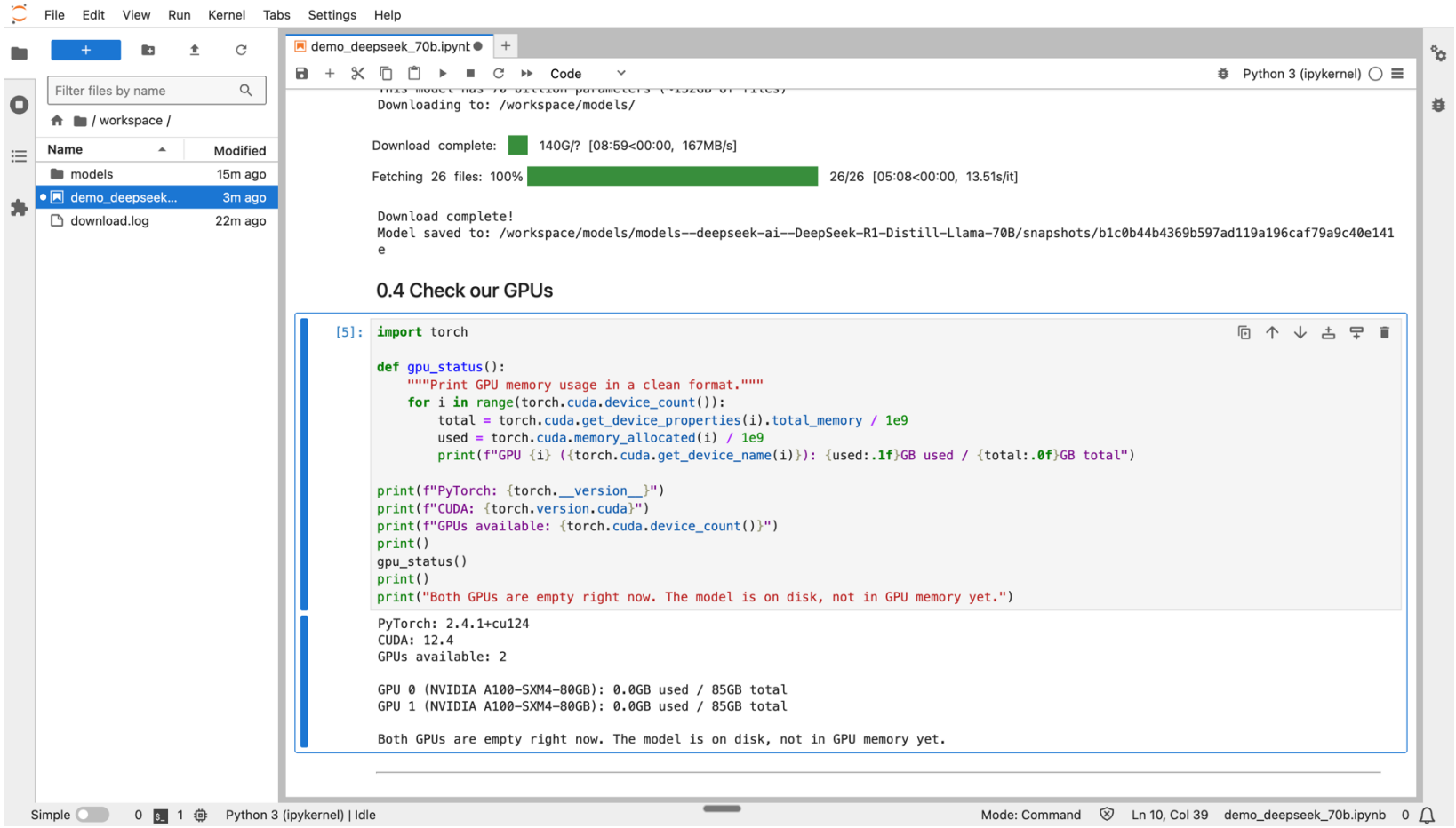

y8l5vyh9k1jjpy-8888.proxy.runpod.net
Ask ChatGPT
File Edit View Run Kernel Tabs Settings Help
Filter files by name
/ workspace /
Name
Modified
Launcher
workspace
Notebook
Python 3 (ipykernel)
Console
Python 3 (ipykernel)
Other
Terminal
Text File
Markdown File
Python File
Show Contextual Help
Simple
0 0
Launcher 1

PART 1:

DOWNLOAD THE MODEL FROM HUGGINGFACE

WE ARE DOWNLOADING QWEN2.5 72B INSTRUCT
A 72 BILLION PARAMETER MODEL FROM ALIBABA.


File Edit View Run Kernel Tabs Settings Help
Launcher
demo_model_parallelism.ip
Code
Python 3 (ipykernel)
Filter files by name
/ workspace /
Name
Modified
models 4m ago
all_results.json 2h ago
demo_model_pa... 1m ago
download.log 3h ago
hf_results.json 32m ago
pp_results.json 2h ago
vllm_install.log 3h ago
0.3 Download the model from HuggingFace
We are downloading Qwen2.5 72B Instruct -- a 72 billion parameter model from Alibaba.
The files total ~136GB. This will take a few minutes.
[*]:
from huggingface_hub import snapshot_download
MODEL_NAME = "Qwen/Qwen2.5-72B-Instruct"
print(f"Downloading: {MODEL_NAME}")
print(f"72 billion parameters = ~136GB of files")
print()
model_path = snapshot_download(MODEL_NAME, cache_dir="/workspace/models")
print(f"\nDownload complete!")
print(f"Saved to: {model_path}")
Downloading: Qwen/Qwen2.5-72B-Instruct
72 billion parameters = ~136GB of files
Fetching 47 files: 62% 29/47 [04:49<03:13, 10.76s/it]
model-00002-of-00037.safetensors: 100% 4.00G/4.00G [01:16<00:00, 16.4MB/s]
model-00001-of-00037.safetensors: 100% 3.76G/3.76G [01:03<00:00, 40.6MB/s]
.gitattributes: 1.52k/? [00:00<00:00, 197kB/s]
merges.txt: 1.67M/? [00:00<00:00, 7.31MB/s]
config.json: 100% 663/663 [00:00<00:00, 27.7kB/s]
LICENSE: 6.96k/? [00:00<00:00, 295kB/s]
generation_config.json: 100% 242/242 [00:00<00:00, 14.9kB/s]
README.md: 6.26k/? [00:00<00:00, 784kB/s]
model-00004-of-00037.safetensors: 100% 4.00G/4.00G [01:01<00:00, 124MB/s]
Simple 0 1 Python 3 (ipykernel) | Busy
Mode: Command Ln 39, Col 71 demo_model_parallelism.ipynb 1


File Edit View Run Kernel Tabs Settings Help
demo_deepseek_70b.ipynb
Code
Python 3 (ipykernel)
Filter files by name
/ workspace /
Name
Modified
models 15m ago
demo_deepseek... 3m ago
download.log 22m ago
Downloading to: /workspace/models/
Download complete: 140G/? [08:59<00:00, 167MB/s]
Fetching 26 files: 100% 26/26 [05:08<00:00, 13.51s/it]
Download complete!
Model saved to: /workspace/models/models--deepseek-ai--DeepSeek-R1-Distill-Llama-70B/snapshots/b1c0b44b4369b597ad119a196caf79a9c40e141e
0.4 Check our GPUs
[5]:
import torch
def gpu_status():
"""Print GPU memory usage in a clean format."""
for i in range(torch.cuda.device_count()):
total = torch.cuda.get_device_properties(i).total_memory / 1e9
used = torch.cuda.memory_allocated(i) / 1e9
print(f"GPU {i} ({torch.cuda.get_device_name(i)}): {used:.1f}GB used / {total:.0f}GB total")
print(f"PyTorch: {torch.__version__}")
print(f"CUDA: {torch.version.cuda}")
print(f"GPUs available: {torch.cuda.device_count()}")
print()
gpu_status()
print()
print("Both GPUs are empty right now. The model is on disk, not in GPU memory yet.")
PyTorch: 2.4.1+cu124
CUDA: 12.4
GPUs available: 2
GPU 0 (NVIDIA A100-SXM4-80GB): 0.0GB used / 85GB total
GPU 1 (NVIDIA A100-SXM4-80GB): 0.0GB used / 85GB total
Both GPUs are empty right now. The model is on disk, not in GPU memory yet.
Simple 0 1 Python 3 (ipykernel) | Idle
Mode: Command Ln 10, Col 39 demo_deepseek_70b.ipynb 0

1.3 Try to load on 1 GPU -- watch it crash

[ ]:

```
from transformers import AutoModelForCausalLM
import gc

print("Attempting to load 144GB model onto 1 GPU (80GB)...")
print("This WILL fail.")
print()

try:
    model = AutoModelForCausalLM.from_pretrained(
        model_path,
        torch_dtype=torch.float16,
        device_map={"":  "cuda:0"},  # Force everything onto GPU 0
    )
except Exception as e:
    # CRITICAL: Clean up the partial allocation from the failed load!
    # Without this, device_map="auto" will see less free VRAM and
    # offload layers to CPU, making inference ~20x slower.
    try:
        del model
    except NameError:
        pass
    gc.collect()
    torch.cuda.empty_cache()

    print(f"CRASHED: {type(e).__name__}")
    print(f"\n{e}")
    print("\n" + "="*60)
    print("The model is 144GB. The GPU has 80GB. It doesn't fit.")
    print("We need a solution.")
    print()
    gpu_status()  # Verify GPUs are clean after cleanup
```

This 144 GB Model Doesn't Fit in 1 80 GB GPU

Parameters × 2 = Gigabytes

Memory (GB) = Parameters (billions) × Bytes per param FP16 = 2 bytes/param

Model	Parameters	× 2 bytes	= Memory	H100s (80 GB)
GPT-2	1.5B	1.5 × 2	**3 GB**	1
LLaMA 2 7B	7B	7 × 2	**14 GB**	1
LLaMA 2 70B	70B	70 × 2	**140 GB**	2
GPT-3	175B	175 × 2	**350 GB**	5
LLaMA 3.1 405B	405B	405 × 2	**810 GB**	11
GLM-5	**744B**	744 × 2	**1,488 GB**	**19**

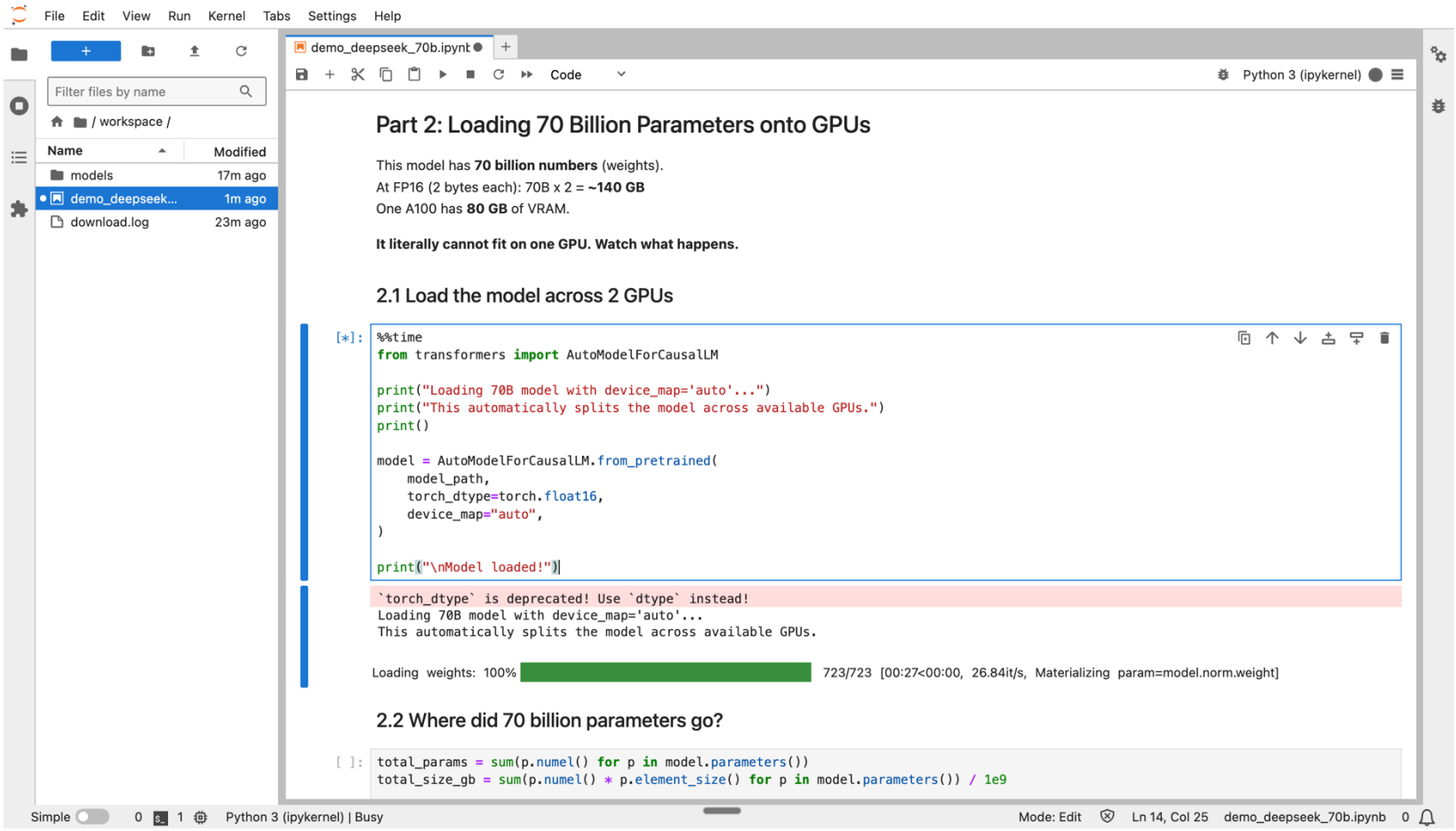
And this is JUST the weights. During actual use, you need even more memory for KV cache, activations, and overhead.

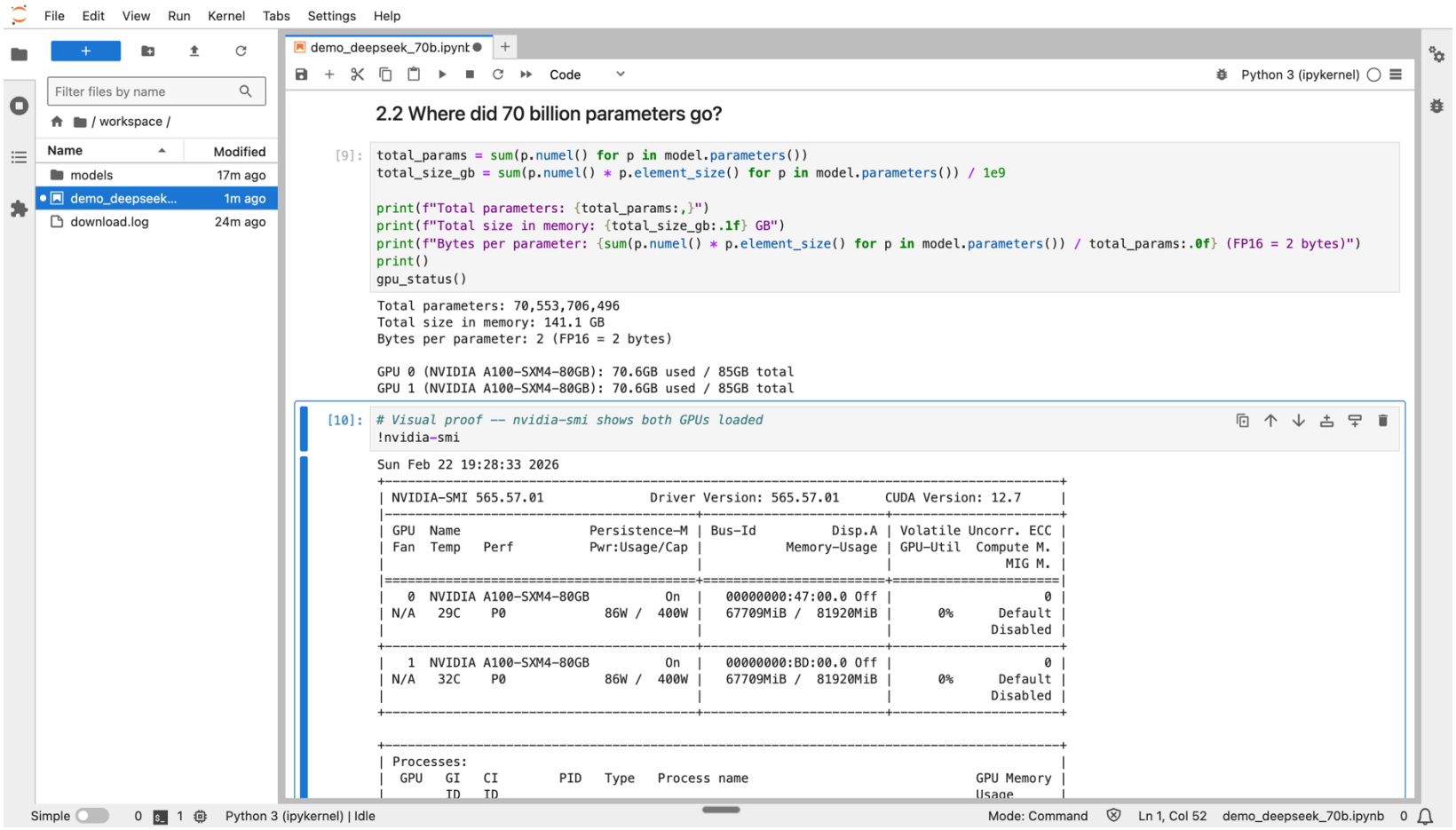
PART 2:

PIPELINE PARALLELISM WITH HUGGINGFACE (2 GPUS)

IDEA: PUT SOME LAYERS ON GPU 0 AND OTHER LAYERS ON GPU 1.

DATA FLOWS FROM GPU 0 TO GPU 1, LIKE AN ASSEMBLY LINE.


File Edit View Run Kernel Tabs Settings Help
demo_deepseek_70b.ipynb
Code
Python 3 (ipykernel)
Filter files by name
/ workspace /
Name
Modified
models 17m ago
demo_deepseek... 1m ago
download.log 23m ago
Part 2: Loading 70 Billion Parameters onto GPUs
This model has 70 billion numbers (weights).
At FP16 (2 bytes each): 70B x 2 = ~140 GB
One A100 has 80 GB of VRAM.
It literally cannot fit on one GPU. Watch what happens.
2.1 Load the model across 2 GPUs
[*]:
%%time
from transformers import AutoModelForCausalLM
print("Loading 70B model with device_map='auto'...")
print("This automatically splits the model across available GPUs.")
print()
model = AutoModelForCausalLM.from_pretrained(
model_path,
torch_dtype=torch.float16,
device_map="auto",
)
print("\nModel loaded!")
`torch_dtype` is deprecated! Use `dtype` instead!
Loading 70B model with device_map='auto'...
This automatically splits the model across available GPUs.
Loading weights: 100% 723/723 [00:27<00:00, 26.84it/s, Materializing param=model.norm.weight]
2.2 Where did 70 billion parameters go?
[ ]:
total_params = sum(p.numel() for p in model.parameters())
total_size_gb = sum(p.numel() * p.element_size() for p in model.parameters()) / 1e9
Simple 0 1 Python 3 (ipykernel) | Busy
Mode: Edit Ln 14, Col 25 demo_deepseek_70b.ipynb 0


File Edit View Run Kernel Tabs Settings Help
demo_deepseek_70b.ipynb
Filter files by name
/ workspace /
Name
Modified
models
17m ago
demo_deepseek...
1m ago
download.log
24m ago
Code
Python 3 (ipykernel)
2.2 Where did 70 billion parameters go?
[9]:
total_params = sum(p.numel() for p in model.parameters())
total_size_gb = sum(p.numel() * p.element_size() for p in model.parameters()) / 1e9
print(f"Total parameters: {total_params:,}")
print(f"Total size in memory: {total_size_gb:.1f} GB")
print(f"Bytes per parameter: {sum(p.numel() * p.element_size() for p in model.parameters()) / total_params:.0f} (FP16 = 2 bytes)")
print()
gpu_status()
Total parameters: 70,553,706,496
Total size in memory: 141.1 GB
Bytes per parameter: 2 (FP16 = 2 bytes)
GPU 0 (NVIDIA A100-SXM4-80GB): 70.6GB used / 85GB total
GPU 1 (NVIDIA A100-SXM4-80GB): 70.6GB used / 85GB total
[10]:
Visual proof -- nvidia-smi shows both GPUs loaded
!nvidia-smi
Sun Feb 22 19:28:33 2026
| NVIDIA-SMI 565.57.01 Driver Version: 565.57.01 CUDA Version: 12.7 |
| GPU Name Persistence-M | Bus-Id Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap | Memory-Usage | GPU-Util Compute M. |
| MIG M. |
| 0 NVIDIA A100-SXM4-80GB On | 00000000:47:00.0 Off | 0 |
| N/A 29C P0 86W / 400W | 67709MiB / 81920MiB | 0% Default |
| Disabled |
| 1 NVIDIA A100-SXM4-80GB On | 00000000:BD:00.0 Off | 0 |
| N/A 32C P0 86W / 400W | 67709MiB / 81920MiB | 0% Default |
| Disabled |
| Processes: |
| GPU GI CI PID Type Process name GPU Memory |
| ID ID Usage |
Simple
Python 3 (ipykernel) | Idle
Mode: Command
Ln 1, Col 52
demo_deepseek_70b.ipynb

Both GPUs are loaded! The model is split across 2 GPUs.

This is **pipeline parallelism** in action -- different layers live on different GPUs.

2.3 Which layers are on which GPU?

[11]:
```python
from collections import Counter

device_counts = Counter()
device_sizes = {}

for name, param in model.named_parameters():
    device = str(param.device)
    size_mb = param.numel() * param.element_size() / 1e6
    device_counts[device] += 1
    device_sizes[device] = device_sizes.get(device, 0) + size_mb

print("Model layer distribution across GPUs:")
print("=" * 50)
for device in sorted(device_sizes.keys()):
    print(f"  {device}: {device_counts[device]} tensors, {device_sizes[device]/1000:.1f} GB")

print(f"\nTotal: {sum(device_counts.values())} tensors across {len(device_sizes)} GPUs")
```

```
Model layer distribution across GPUs:
==================================================
  cuda:0: 361 tensors, 70.6 GB
  cuda:1: 362 tensors, 70.6 GB

Total: 723 tensors across 2 GPUs
```

2.3 Which layers are on which GPU?

This is pipeline parallelism: the first half of layers on GPU 0, second half on GPU 1.

[10]:
```python
# Show the layer-to-GPU mapping
gpu0_layers = []
gpu1_layers = []

for name, param in model.named_parameters():
    if "layers." in name:
        layer_num = int(name.split("layers.")[1].split(".")[0])
        device = str(param.device)
        if "cuda:0" in device:
            if layer_num not in gpu0_layers:
                gpu0_layers.append(layer_num)
        elif "cuda:1" in device:
            if layer_num not in gpu1_layers:
                gpu1_layers.append(layer_num)

gpu0_layers.sort()
gpu1_layers.sort()

print("PIPELINE PARALLELISM: Layer-to-GPU Mapping")
print("=" * 50)
print(f"GPU 0: Layers {gpu0_layers[0]}-{gpu0_layers[-1]} ({len(gpu0_layers)} layers)")
print(f"GPU 1: Layers {gpu1_layers[0]}-{gpu1_layers[-1]} ({len(gpu1_layers)} layers)")
print()
print("Data flows: Input -> GPU 0 -> GPU 1 -> Output")
print("This is an assembly line: each GPU handles a different stage.")
```

```
PIPELINE PARALLELISM: Layer-to-GPU Mapping
==================================================
GPU 0: Layers 0-38 (39 layers)
GPU 1: Layers 39-79 (41 layers)

Data flows: Input -> GPU 0 -> GPU 1 -> Output
This is an assembly line: each GPU handles a different stage.
```

2.4 Generate text and measure speed

[9]:
```python
import time
from transformers import TextStreamer

# A reasoning question with a clear, concise answer
prompt = "A farmer has 3 fields. Field A produces twice as much wheat as Field

inputs = tokenizer(prompt, return_tensors="pt").to(model.device)
num_input_tokens = inputs["input_ids"].shape[1]

print(f"Prompt: '{prompt}'")
print(f"\nGenerating with Pipeline Parallelism...\n")

# Custom streamer that also tracks time-to-first-token
class TimedStreamer(TextStreamer):
    def __init__(self, tokenizer, **kwargs):
        super().__init__(tokenizer, **kwargs)
        self.start_time = time.time()
        self.first_token_time = None

    def on_finalized_text(self, text, stream_end=False):
        if self.first_token_time is None and text:
            self.first_token_time = time.time()
        print(text, end="", flush=True)

streamer = TimedStreamer(tokenizer, skip_special_tokens=True)

start = time.time()
with torch.no_grad():
    outputs = model.generate(
        **inputs,
        max_new_tokens=2048,
        temperature=0.7,
        do_sample=True,
        streamer=streamer,
    )
pp_total_time = time.time() - start
```

```
Subtract 50 from both sides:

\[
5x = 500
\]

Divide both sides by 5:

\[
x = 100
\]

Now, we can find the production for each field:
- Field B produces \( x = 100 \) kg.
- Field A produces \( 2x = 2 \times 100 = 200 \) kg.
- Field C produces \( 2x + 50 = 200 + 50 = 250 \) kg.

So, the production from each field is:
- Field A: 200 kg
- Field B: 100 kg
- Field C: 250 kg

To verify, we add the amounts together:

\[
100 + 200 + 250 = 550 \text{ kg}
\]

The solution is correct. Each field produces:
- Field A: 200 kg
- Field B: 100 kg
- Field C: 250 kg

============================================================
PIPELINE PARALLELISM RESULTS (HuggingFace, 2 GPUs):
  Total time:                 44.7s
  Tokens generated:           414
  Speed:                      9.3 tokens/sec

Correct answer: Field B = 100kg, Field A = 200kg, Field C = 250kg
```

PART 3:
QUANTIZATION WITH HUGGINGFACE (1 GPU)

EACH WEIGHT IS STORED AS A 16-BIT FLOAT (FP16 = 2 BYTES).
WHAT IF WE USED 4-BIT INTEGERS INSTEAD (INT4 = 0.5 BYTES)?

FP16: 70B X 2 BYTES = 140 GB **NEEDS 2 GPUS**
INT4: 70B X 0.5 BYTES = 35 GB **FITS ON 1 GPU!**

THE TRADEOFF: LESS PRECISION MEANS SLIGHTLY LOWER QUALITY.

LIKE COMPRESSING A PHOTO FROM RAW TO JPEG
SMALLER FILE, ALMOST THE SAME PICTURE.

3.1 First, unload the FP16 model

We need to free GPU memory before loading the quantized version.

[11]:
```
import gc

del model
gc.collect()
torch.cuda.empty_cache()

print("FP16 model unloaded. GPU memory freed.")
gpu_status()
```

```
FP16 model unloaded. GPU memory freed.
  GPU 0 (NVIDIA A100-SXM4-80GB): 0.0 GB used / 85 GB total
  GPU 1 (NVIDIA A100-SXM4-80GB): 2.5 GB used / 85 GB total
```

3.2 Load the same model in 4-bit

[12]:
```
%%time
from transformers import BitsAndBytesConfig

quant_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_quant_type="nf4",
)

print("Loading the SAME 70B model in 4-bit quantization...")
print("70B x 0.5 bytes = ~35GB. This should fit on a SINGLE GPU.")
print()

model_4bit = AutoModelForCausalLM.from_pretrained(
    model_path,
    quantization_config=quant_config,
    device_map="auto",
)

print("\n4-bit model loaded!")
```

```
Loading the SAME 70B model in 4-bit quantization...
70B x 0.5 bytes = ~35GB. This should fit on a SINGLE GPU.
```

3.3 Proof: it fits on 1 GPU!

[13]:
```
!nvidia-smi
```

```
Sun Feb 22 23:07:49 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 565.57.01              Driver Version: 565.57.01      CUDA Version: 12.7     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA A100-SXM4-80GB          On  |   00000000:47:00.0 Off |                    0 |
| N/A   29C    P0             87W /  400W |   29377MiB /  81920MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA A100-SXM4-80GB          On  |   00000000:BD:00.0 Off |                    0 |
| N/A   34C    P0            115W /  400W |   41465MiB /  81920MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
+-----------------------------------------------------------------------------------------+
```

Compare to Part 2 where we needed **both** GPUs.

Now the same 70B model fits on **one** GPU thanks to quantization.

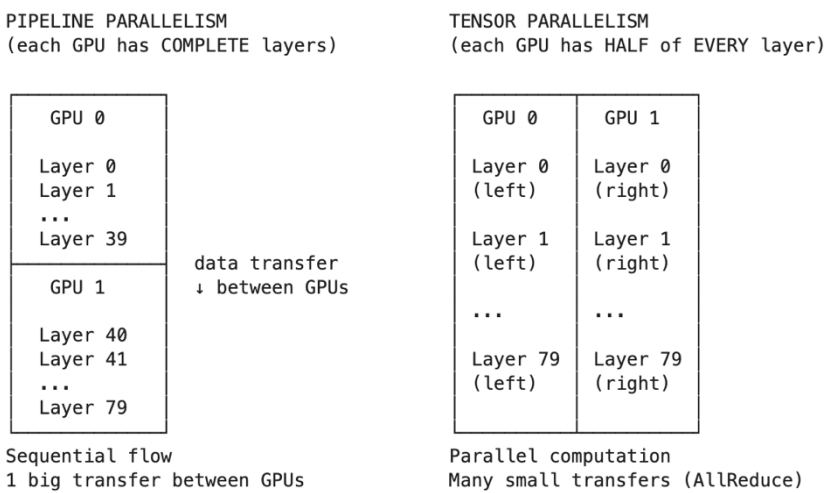
PART 4:
VLLM TENSOR PARALLEL (2 GPUS)

TENSOR PARALLELISM: SPLIT EVERY WEIGHT MATRIX ACROSS GPUS.
BOTH GPUS COMPUTE IN PARALLEL ON EVERY LAYER.

Part 4: Understanding Tensor Parallelism (Concept)

Pipeline parallelism splits **layers** across GPUs. But there's another approach:
Tensor parallelism splits individual weight matrices.

Instead of giving each GPU complete layers, we give each GPU **half of every matrix**.
Each GPU computes a partial result, then they combine.



Why is TP faster? Pipeline parallelism has **pipeline bubbles**: GPU 1 sits idle while GPU 0 processes, then vice versa. Tensor parallelism keeps **both GPUs busy simultaneously** on every layer.

The tradeoff: TP requires **fast GPU-to-GPU communication** (NVLink at 600+ GB/s). PP only needs occasional transfers. This is why:

- **TP is used within a server** (GPUs connected by NVLink)
- **PP is used across servers** (connected by slower InfiniBand)

We'll see the speed difference in Part 6 using vLLM.

6.2 vLLM Tensor Parallel (2 GPUs)

Tensor Parallelism: split every weight matrix across GPUs. Both GPUs compute in parallel on every layer.

[2]:
```
%%time
from vllm import LLM, SamplingParams

print("Loading 70B model with vLLM Tensor Parallelism...")
print("tensor_parallel_size=2")
print("This splits MATRICES across GPUs -- both GPUs work on every layer.")
print()

llm_tp = LLM(
    model=model_path,
    tensor_parallel_size=2,
    dtype="float16",
    gpu_memory_utilization=0.92,
    max_model_len=2048,
    enforce_eager=True,
    disable_custom_all_reduce=True,
)

print("\nvLLM loaded with Tensor Parallelism!")
```

```
Loading 70B model with vLLM Tensor Parallelism...
tensor_parallel_size=2
This splits MATRICES across GPUs -- both GPUs work on every layer.

INFO 02-22 23:15:04 [utils.py:261] non-default args: {'dtype': 'float16', 'max_model_len': 2048, 'tensor_parallel_size': 2, 'gpu_memory_utilization': 0.92,
'disable_log_stats': True, 'enforce_eager': True, 'disable_custom_all_reduce': True, 'model': '/workspace/models/models--Qwen--Qwen2.5-72B-Instruct/snapsho
s/495f39366efef23836d0cfae4fbe635880d2be31'}
INFO 02-22 23:15:14 [model.py:541] Resolved architecture: Qwen2ForCausalLM
WARNING 02-22 23:15:14 [model.py:1885] Casting torch.bfloat16 to torch.float16.
INFO 02-22 23:15:14 [model.py:1561] Using max model len 2048
INFO 02-22 23:15:14 [scheduler.py:226] Chunked prefill is enabled with max_num_batched_tokens=8192.
INFO 02-22 23:15:14 [vllm.py:624] Asynchronous scheduling is enabled.
WARNING 02-22 23:15:14 [vllm.py:662] Enforce eager set, overriding optimization level to -O0
INFO 02-22 23:15:14 [vllm.py:762] Cudagraph is disabled under eager mode
WARNING 02-22 23:15:14 [system_utils.py:140] We must use the `spawn` multiprocessing start method. Overriding VLLM_WORKER_MULTIPROC_METHOD to 'spawn'. See
ttps://docs.vllm.ai/en/latest/usage/troubleshooting.html#python-multiprocessing for more information. Reasons: CUDA is initialized
(EngineCore_DP0 pid=13285) INFO 02-22 23:15:20 [core.py:96] Initializing a V1 LLM engine (v0.15.1) with config: model='/workspace/models/models--Qwen--Qwen
2.5-72B-Instruct/snapshots/495f39366efef23836d0cfae4fbe635880d2be31', speculative_config=None, tokenizer='/workspace/models/models--Qwen--Qwen2.5-72B-Instr
ct/snapshots/495f39366efef23836d0cfae4fbe635880d2be31', skip_tokenizer_init=False, tokenizer_mode=auto, revision=None, tokenizer_revision=None, trust_remot
_code=False, dtype=torch.float16, max_seq_len=2048, download_dir=None, load_format=auto, tensor_parallel_size=2, pipeline_parallel_size=1, data_parallel_si
e=1, disable_custom_all_reduce=True, quantization=None, enforce_eager=True, enable_return_routed_experts=False, kv_cache_dtype=auto, device_config=cuda, st
```

```
vLLM loaded with Tensor Parallelism!
CPU times: user 5.18 s, sys: 5.08 s, total: 10.3 s
Wall time: 1min 45s
```

[3]:
```python
sampling_params = SamplingParams(
    temperature=0.7,
    max_tokens=2048,
)

print(f"Generating with vLLM Tensor Parallelism...\n")

start = time.time()
outputs = llm_tp.generate([prompt], sampling_params)
vllm_tp_total_time = time.time() - start

generated = outputs[0].outputs[0].text
vllm_tp_tokens = len(outputs[0].outputs[0].token_ids)
vllm_tp_speed = vllm_tp_tokens / vllm_tp_total_time

print(prompt + generated)
print(f"\n{'='*60}")
print(f"vLLM TENSOR PARALLEL RESULTS (2 GPUs):")
print(f"  Total time:                {vllm_tp_total_time:.1f}s")
print(f"  Tokens generated:          {vllm_tp_tokens}")
print(f"  Speed:                     {vllm_tp_speed:.1f} tokens/sec")
print(f"\nCorrect answer: Field B = 100kg, Field A = 200kg, Field C = 250kg")
```

```
The solution is correct. Each field produces:

- Field A: 200 kg
- Field B: 100 kg
- Field C: 250 kg

============================================================
vLLM PIPELINE PARALLEL RESULTS (2 GPUs):
  Total time:                43.0s
  Tokens generated:          462
  Speed:                     10.8 tokens/sec

Correct answer: Field B = 100kg, Field A = 200kg, Field C = 250kg

FAIR COMPARISON (same engine -- vLLM):
  vLLM Tensor Parallel:     19.6 tok/s  (411 tokens in 20.9s)
  vLLM Pipeline Parallel:   10.8 tok/s  (462 tokens in 43.0s)
  TP speedup over PP:     1.83x

Same model, same engine, same prompt -- the ONLY difference is the parallelism strategy.
```

```
Generating with vLLM Tensor Parallelism...
```

Adding requests: 100%  1/1 [00:00<00:00, 49.75it/s]

Processed prompts: 100%  1/1 [00:20<0

```
A farmer has 3 fields. Field A produces twice as much wheat as Field B. Field C produces 50kg more
w much does each field produce? Let's denote the amount of wheat produced by Field B as \( x \) kg.

According to the problem:
- Field A produces twice as much wheat as Field B, so Field A produces \( 2x \) kg.
- Field C produces 50 kg more than Field A, so Field C produces \( 2x + 50 \) kg.
- The total production from all three fields is 550 kg.

We can set up the following equation to represent the total production:
```

SUMMARY

COMPLETE COMPARISON: Four ways to run a 70B model

	HF Pipeline	HF INT4	vLLM TP	vLLM PP
GPUs used	2	1	2	2
Parallelism	Pipeline	None	Tensor	Pipeline
Framework	HuggingFace	HuggingFace	vLLM	vLLM
Precision	FP16	INT4	FP16	FP16
Speed (tok/s)	9.3	8.5	19.6	10.8
Tokens generated	414	414	411	462
Total time	44.7s	49.0s	20.9s	43.0s
Quality	Full	Slight loss	Full	Full
Memory per GPU	~70 GB	~35 GB	~66 GB	~70 GB

FAIR COMPARISON (same engine, same model, same prompt):
  vLLM Tensor Parallel vs vLLM Pipeline Parallel: 1.83x speedup

KEY TAKEAWAYS:
  1. Tensor Parallelism is faster than Pipeline Parallelism (both GPUs work on every layer)
  2. The vLLM TP vs PP comparison is FAIR (same engine) -- proves TP's advantage
  3. HF vs vLLM comparison shows engine optimization matters too (PagedAttention, fused kernels)
  4. Quantization trades quality for efficiency -- fits 70B on 1 GPU
  5. Real systems combine all 3: TP within server, PP across servers, DP for throughput

3D PARALLELISM in production:
  TP(8 GPUs/server) x PP(4 servers) x DP(33 copies) = 1,056 GPUs
  This is how models like Claude and GPT are trained and served.

7

AI at AltoTech

AI Chief Engineer: Enhancing HVAC Operations With Domain-Specialized AI Agents That Leverage Building Ontology and Real-Time IoT Data

Pamekitti Puktalae (pamekitti.p@altotech.ai), Andaman Lekawat, Boon Hawaree, Jirayut Chatphet, Warodom Khamphanchai, Pisitchai Panyapalangkun

Motivation

Our goal is to address **facility management inefficiencies**:

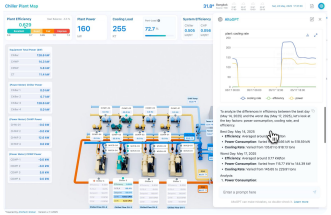
- Frequent ad-hoc analyses require repetitive manual workflows for data extraction and interpretation
- Critical engineering expertise is lost when senior engineers depart
- Facility managers lack contextual understanding of building data to identify risks and inefficiencies proactively
- Failures, energy waste, and complaints often go unnoticed

Challenges

- General **LLM lacks** Heating, Ventilation, and Air Conditioning **expertise**
- Data lacks semantic relationships between building's sensors/spaces
- Operational data distributed across **heterogeneous sources**: Equipment data, sensor readings, documentation, maintenance logs
- **Facility security compliance** requires on-premises processing

Contributions & Innovation

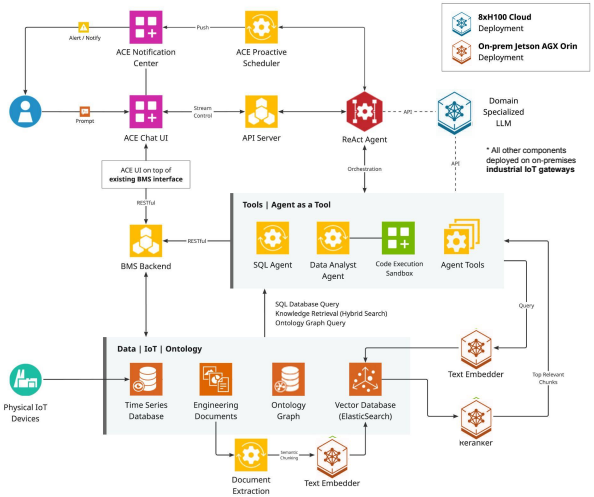
- Agentic AI system integrating building ontology, real-time IoT data, and multi-tool orchestration [9] for facility operations
- Domain-specialized LLM fine-tuned on HVAC datasets; custom HVAC benchmark datasets
- **Cloud-based LLM** on NVIDIA NGC (8×H100 GPUs) [10] with **on-premises retrieval** (NVIDIA Jetson Orin AGX + TensorRT-LLM) for secure handling of sensitive operational data
- Integration with existing Building Management System (BMS)



Conversational interface embedded existing BMS

Proactive alert monitoring interface

Architecture



Real-World Results

Multi Buildings Campus, Singapore: Centralized industrial cooling and air distribution systems integrated.

- Automated detection of sensor faults, equipment malfunctions, manual overrides, operational anomalies, and system-level coordination issues across air distribution and chiller plant

Results: 78% diagnostic time reduced, 60 hr/m saved

High Rise Hospitality Building, Bangkok: 441 rooms with IAQ and occupancy monitoring and industrial cooling system integrated.

- Correlates guest comfort complaints with sensor data and equipment status for automated root cause analysis
- Mobile interface on-site diagnostic capability for field staffs

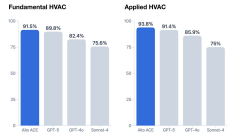
Results: 92% response time reduction, 90 hr/m saved

Technical Components

Domain Specialization

General-purpose LLMs lack HVAC expertise We fine-tuned on domain-specific data [3].

- Training dataset: industry handbooks [2], academic literature, expert-curated
- Custom HVAC benchmarks datasets [4]



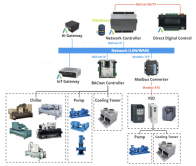
Building Ontology Integration

- Adopted **Project Haystack** [1], mapping entities to a standardized schema
- Agent queries equipment by function and spatial context

Project Haystack

IoT Data Access

IoT protocols (Modbus, BACnet, LoRaWAN) unified to time-series database and mapped to ontology. Allows Agent to execute **SQL query as interface** to historical and real-time telemetry.



Knowledge Management

Engineering documents processed through extraction [5], indexing, and hybrid search [6] with reranking for relevance filtering.

- Engineering documents stored on **local infrastructure**
- Embedding and reranking models [7] fit in edge hardware [8]
- **Eliminates cloud transmission** of proprietary data

Text Embedder

Reranker

Structured CSV/Excel like maintenance logs are unsuitable for retrieval methods. Code execution enables statistical analysis.

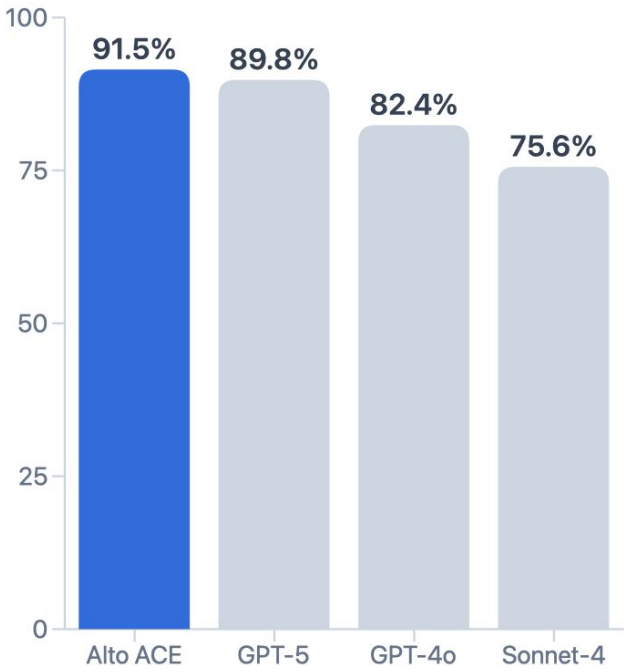
Memory system retains only diagnostic decisions, maintenance patterns, filtering noise.

References

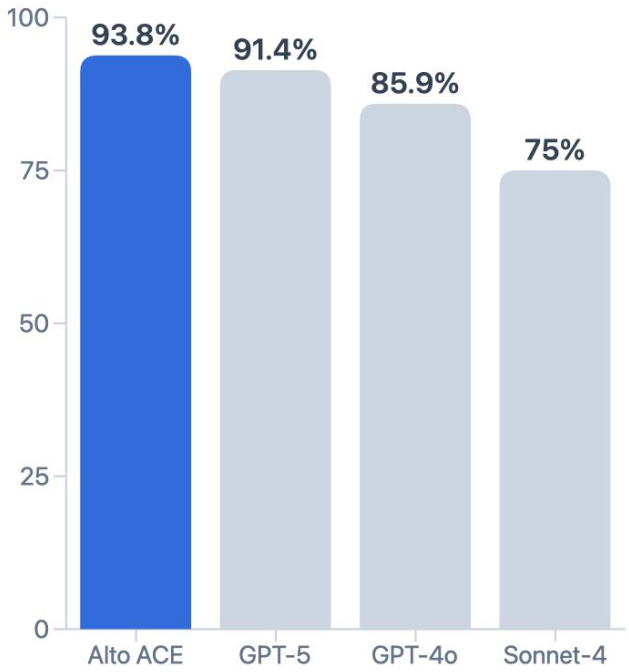
1) Project Haystack Corporation. "Project Haystack Specification Version 4.0." 2025, project-haystack.org.
2) ASHRAE. ASHRAE Handbook Collection (Fundamentals, HVAC Systems and Equipment, Applications, Refrigeration). 2021–2024, ashrae.org/technical-resources/ashrae-handbook.
3) "Qwen2.5-72B-Instruct." Alibaba Cloud, 2024, huggingface.co/Qwen/Qwen2.5-72B-Instruct.
4) NVIDIA. "NeMo Curator: GPU-Accelerated Data Curation for Large Language Models." 2024
5) Liu, J. "LlamaIndex: A Data Framework for LLM Applications." 2022, github.com/run-llama/llama_index.
6) Elasticsearch B.V. "Elasticsearch: Distributed Search and Analytics Engine." 2024, elastic.co/elasticsearch.
7) BAAI. "BGE-base-en-v1.5 Embeddings and BGE-Reranker-Base." 2024, github.com/FlagOpen/FlagEmbedding.
8) NVIDIA. "TensorRT: High-Performance Deep Learning Inference." 2024, developer.nvidia.com/tensorrt.
9) LangChain AI. "LangGraph: Multi-Agent Workflows." 2024, github.com/langchain-ai/langgraph.
10) FPT Software. "FPT AI Factory: NVIDIA GPU Cloud Platform." 2024, fptaicloud.io.

Alto ACE: HVAC Benchmark

Fundamental HVAC (เทียบเท่า ACAT ระดับต้น)



Applied HVAC (เทียบเท่า ACAT ระดับสูง)



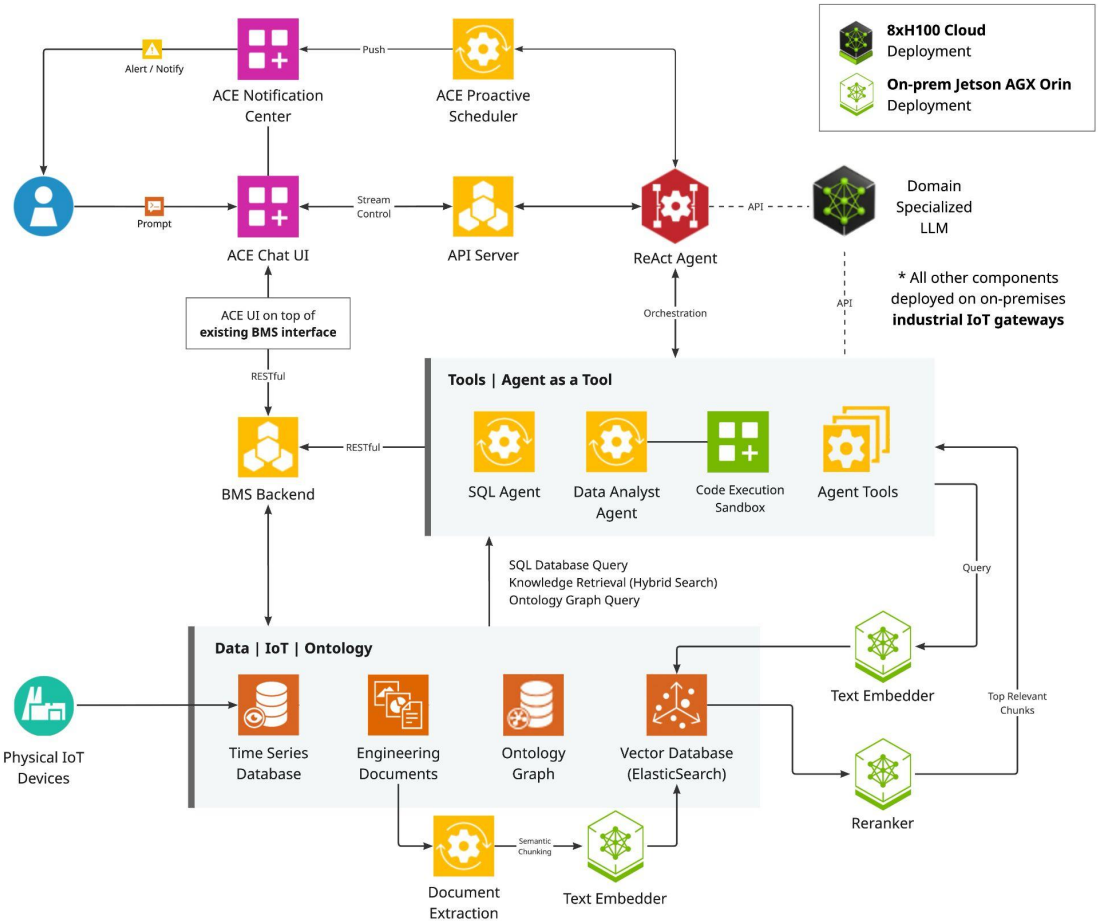
Result: Alto ACE leads both tracks — Fundamental (91.5%) and Applied (93.8%)

Powered by domain fine-tuning on GPT-5 base — HVAC-specialized AI performance benchmark

Alto ACE Hybrid-Cloud Architecture v1-2025.10.0

Enhancing HVAC Operations With Domain-Specialized AI Agents
That Leverage Building Ontology and Real-Time IoT Data

AltoTech Global

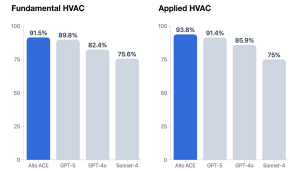


Technical Components

Domain Specialization

General-purpose LLMs lack HVAC expertise
We fine-tuned on domain-specific data [3].

- Training dataset: industry handbooks [2], academic literature, expert-curated
- Custom HVAC benchmarks datasets [4]



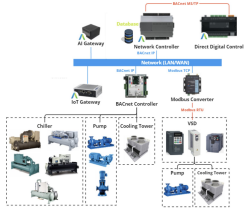
Building Ontology Integration

- Adopted **Project Haystack** [1], mapping entities to a standardized schema
- Agent queries equipment by function and spatial context

Project Haystack

IoT Data Access

IoT protocols (Modbus, BACnet, LoRaWAN) unified to time-series database and mapped to ontology.
Allows Agent to execute **SQL query as interface** to historical and real-time telemetry.



Knowledge Management

Engineering documents processed through extraction [5], indexing, and hybrid search [6] with reranking for relevance filtering.

- Engineering documents stored on **local infrastructure**
- Embedding and reranking models [7] fit in edge hardware [8]
- **Eliminates cloud transmission** of proprietary data

Structured CSV/Excel like maintenance logs are unsuitable for retrieval methods. Code execution enables statistical analysis.

Memory system retains only diagnostic decisions, maintenance patterns, filtering noise.