

DeepSpeed

Diego Andrade Canosa
Roberto López Castro



Índice



- Introducción a DeepSpeed y su integración con PyTorch.
- Mejoras de rendimiento y escalabilidad en el entrenamiento de modelos de gran escala.
- Optimizaciones clave.
- Paralelización de operaciones de entrenamiento y reducción de la comunicación entre GPUs.

Industry AI labs:

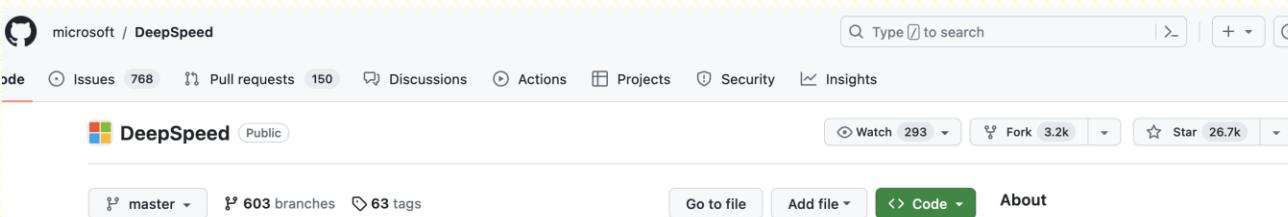
17 Billion Parameters! Microsoft
DeepSpeed Breeds World's Largest
NLP Model

Me finetuning at home:



¿Qué es DeepSpeed?

- DeepSpeed es una biblioteca de código abierto desarrollada por Microsoft.
- Diseñada para mejorar el rendimiento y la escalabilidad en entrenamientos de modelos de aprendizaje profundo.
- Es compatible con el marco de trabajo de PyTorch
- Proporciona optimizaciones y características adicionales para superar los desafíos en el entrenamiento de modelos a gran escala.



DL Training: retos

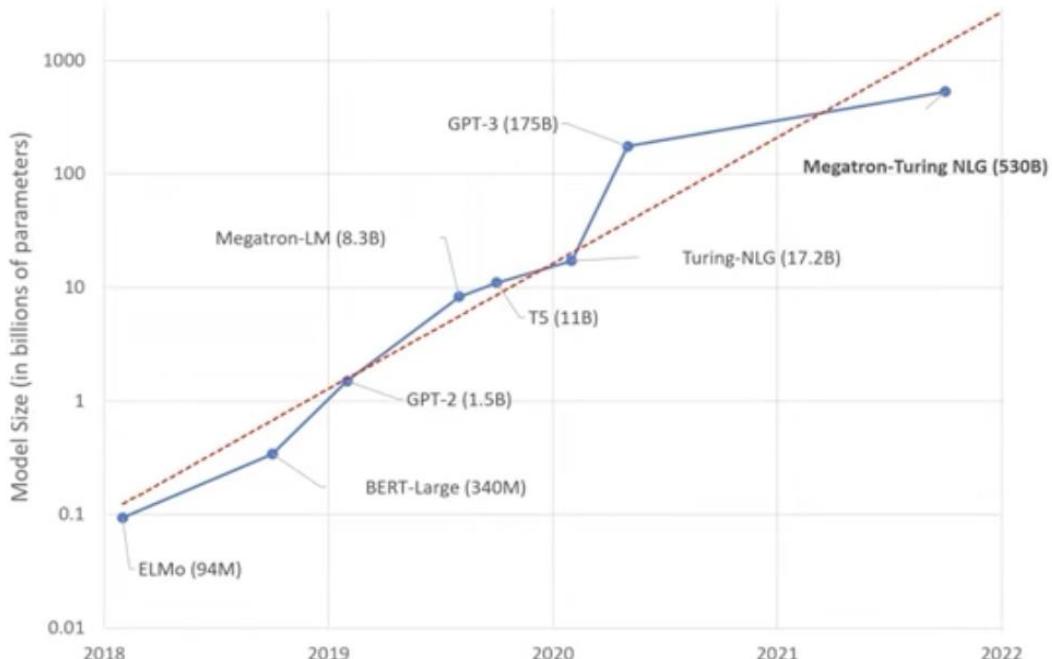
CHALLENGES

- Too slow to train high-quality models on massive data
- More hardware \neq higher throughput, bigger model
- Higher throughput \neq better accuracy, faster convergence
- Better techniques \neq handy to use

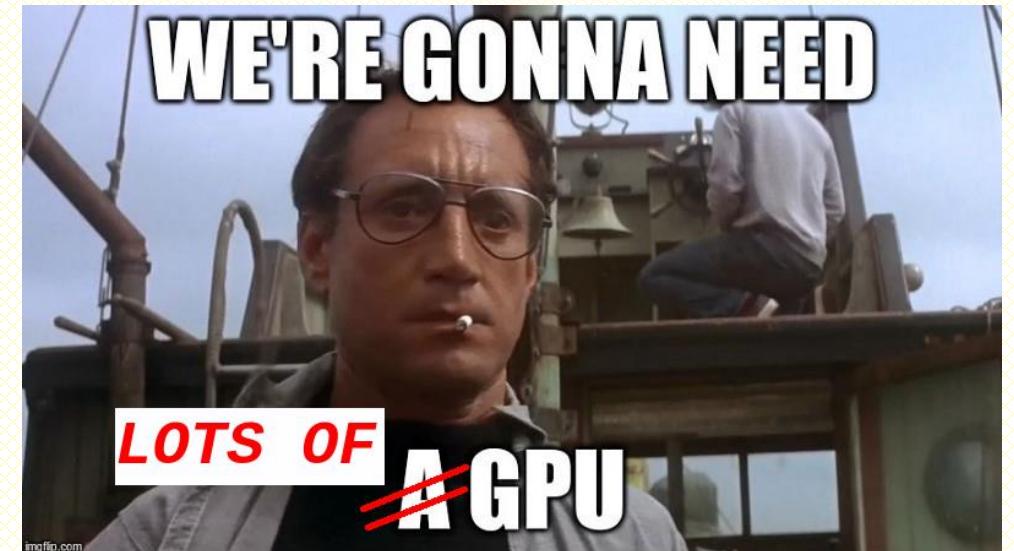
DESIRED CAPABILITY OF DEEPSPEED

- **Efficiency:** Efficient use of hardware for high throughput and scalability
- **Effectiveness:** High accuracy and fast convergence, lowering cost
- **Easy to use:** Improve development productivity of model scientists

Motivación



- Modelos más grandes -> major precision
- El tamaño de los modelos sigue creciendo
- No se ha alcanzado un techo en la precision todavía
- Más eficiencia computacional para entrenar modelos grandes con la misma precisión que modelos de menor tamaño



The image shows two side-by-side screenshots of the Microsoft Bing Chat interface.

Left Screenshot:

- Header:** Microsoft Bing, SEARCH, CHAT
- Text Input:** why large language models are important?
- Search Progress:** ✓ Searching for: why large language models are important
✓ Generating answers for you...
- Text Content:** Large language models play an important role in natural language processing (NLP) because they enable machines to better understand natural language and generate more accurate results when processing text¹. They can recognize, summarize, translate, predict and generate text and other content based on knowledge gained from massive datasets². They are among the most successful applications of transformer models³. Is there anything else you would like to know?
Learn more: 1. vitalflux.com 2. blogs.nvidia.com 3. kumar-vodex.medium.com +2 more 1 of 15
- Text Input:** Tell me a bit more with specific examples and bullet points.
- Text Content:** Sure! Here are some specific examples of how large language models can be used:
 - **Voicebots:** Large language models can more accurately capture the complexities of human speech and handle the large amount of data required for voice recognition. This is important for voicebots because they need to understand the user and respond accordingly.
 - **Translation:** Large language models can translate text from one language to another with high accuracy.
 - **Content generation:** Large language models can generate text such as essays or computer code with limited to no supervision.Is there anything else you would like to know?
- Page Number:** 2 of 15

Right Screenshot:

- Header:** Choose a conversation style (Preview)
- Conversation Style Options:** More Creative, More Balanced (selected), More Precise
- Text Input:** why large language models are important?
- Search Progress:** ✓ Searching for: why are large language models important?
✓ Generating answers for you...
- Text Content:** Large language models (LLMs) are deep learning algorithms that can recognize, summarize, translate, predict and generate text and other content based on knowledge gained from massive datasets. They are among the most successful applications of transformer models¹. Training large language models is an important part of Natural Language Processing (NLP). Large language models utilize large datasets to learn the structure and relationships between words in a language. This can be performed using various algorithms, such as recurrent neural networks or transformers². Large language models are unlocking new possibilities in areas such as search engines, natural language processing, healthcare, robotics and code generation³. They can more accurately capture the complexities of human speech and more easily handle the large amount of data that is required for voice recognition^{3, 4}. Is there anything else I can help you with?
Learn more: 1. blogs.nvidia.com 2. vitalflux.com 3. kumar-vodex.medium.com +2 more 1 of 15
- Text Input:** New topic
- Text Input:** Ask me anything...
- Helpful Buttons:** ? What are transformer models?, What is Natural Language Processing?, What is voice recognition?



Modelos grandes necesitan paralelismo

	Max Parameter (in billions)	Max Parallelism	Compute Efficiency	Usability (Model Rewrite)
Data Parallel (DP)	Approx. 1.2	>1000	Very Good	Great
Model Parallel (MP)	Approx. 20	Approx. 16	Good	Needs Model Rewrite
MP + DP	Approx. 20	> 1000	Good	Needs Model Rewrite
Pipeline Parallel (PP)	Approx. 100	Approx. 128	Very Good	Needs Model Rewrite
PP + DP	Approx. 100	> 1000	Very Good	Needs Model Rewrite
MP + PP + DP	> 1000	> 1000	Very Good	Needs Significant Model Rewrite
ZeRO	> 1000	> 1000	Very Good	Great

ZeRO



ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

Samyam Rajbhandari*, Jeff Rasley*, Olatunji Ruwase, Yuxiong He
`{samyamr, jerasley, olruwase, yuxhe}@microsoft.com`

Abstract

Large deep learning models offer significant accuracy gains, but training billions to trillions of parameters is challenging. Existing solutions such as data and model parallelisms exhibit fundamental limitations to fit these models into limited device memory, while obtaining computation,

Zero Redundancy Optimizer

- El Optimizador de Redundancia Cero (abreviado como ZeRO) es una novedosa tecnología de optimización de memoria para el aprendizaje profundo distribuido a gran escala.
- ZeRO puede entrenar modelos de aprendizaje profundo con 100 mil millones de parámetros en los clústeres actuales de GPU con una velocidad de procesamiento de tres a cinco veces mayor que el sistema base.
- También ofrece un camino claro para entrenar modelos con billones de parámetros. ZeRO representa una parte fundamental del core de DeepSpeed

Zero Redundancy Optimizer

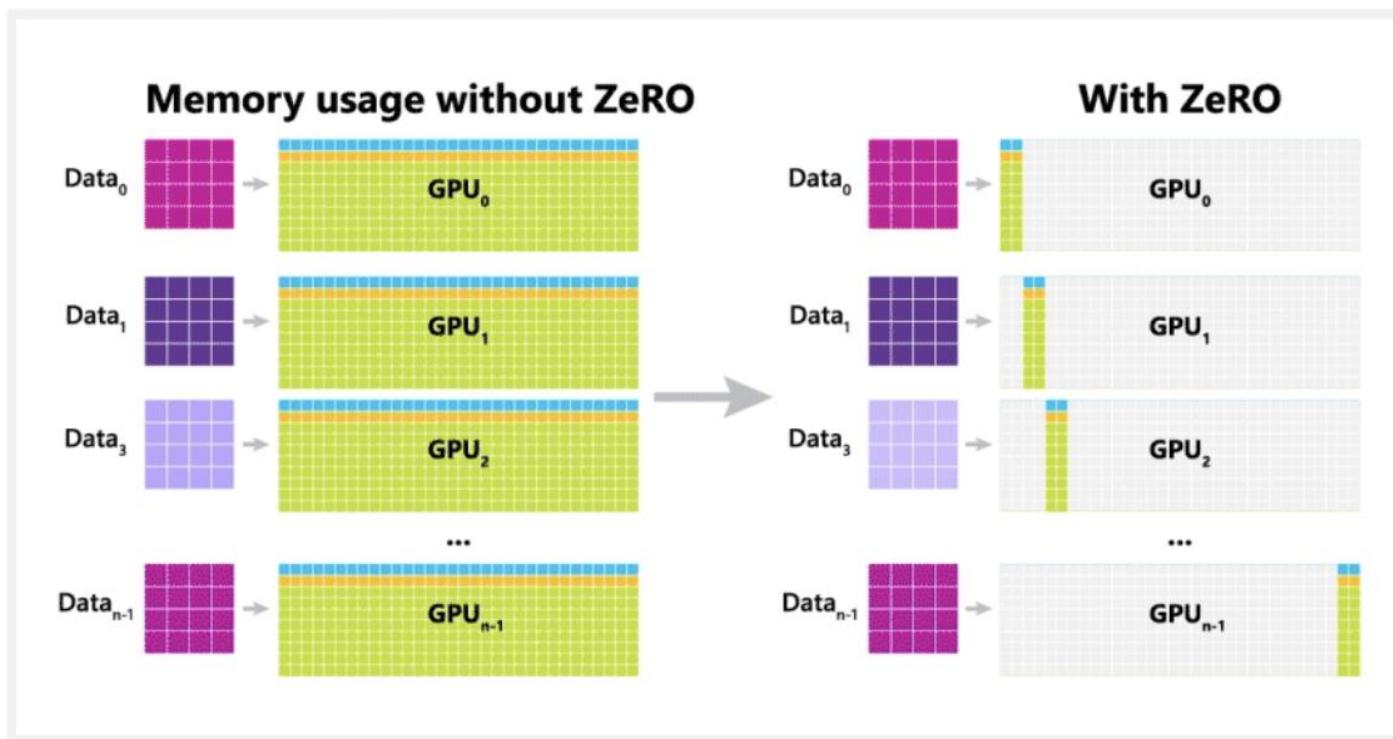
- ZeRO elimina las redundancias de memoria en los procesos de paralelismo de datos al **particionar** los estados del modelo (**parámetros, gradientes y estado del optimizador**) entre los procesos de paralelismo de datos en lugar de replicarlos.
- Utiliza un programa de comunicación dinámica durante el entrenamiento para compartir el estado necesario entre los dispositivos distribuidos y así mantener la granularidad computacional y el volumen de comunicación del paralelismo de datos.

Zero Redundancy Optimizer

- DeepSpeed llama a este paralelismo de datos impulsado por ZeRO, lo que permite que el uso de memoria por dispositivo aumente de manera lineal con el grado de paralelismo de datos y genere un volumen de comunicación similar al del paralelismo de datos.
- El paralelismo de datos impulsado por ZeRO puede adaptarse a modelos de tamaño arbitrario, siempre y cuando la memoria agregada de los dispositivos sea lo suficientemente grande como para compartir los estados del modelo.

Zero Redundancy Optimizer

DeepSpeed + ZeRO



Zero Redundancy Optimizer

- ZeRO tiene tres etapas principales de optimización, que corresponden a la partición de los estados del optimizador, los gradientes y los parámetros. Cuando se habilitan de forma acumulativa:
 1. Partición de los estados del optimizador (Pos) - reducción de memoria de 4 veces, mismo volumen de comunicación que el paralelismo de datos.
 2. Agregar partición de gradientes (Pos+g) - reducción de memoria de 8 veces, mismo volumen de comunicación que el paralelismo de datos.

Zero Redundancy Optimizer

3. Agregar partición de parámetros (Pos+g+p) - La reducción de memoria es lineal con el grado de paralelismo de datos Nd. Por ejemplo, dividir en 64 GPU (Nd = 64) resultará en una reducción de memoria de 64 veces. Hay un aumento modesto del 50% en el volumen de comunicación.

Zero Redundancy Optimizer

- ZeRO elimina las redundancias de memoria y pone a disposición toda la capacidad de memoria agregada de un clúster.
- Con las tres etapas habilitadas, ZeRO puede entrenar un modelo de billones de parámetros en "solo" 1024 GPU de NVIDIA.
- Un modelo de billones de parámetros con un optimizador como Adam en precisión de 16 bits requiere aproximadamente 16 terabytes (TB) de memoria para almacenar los estados del optimizador, los gradientes y los parámetros. 16TB divididos por 1024 son 16GB, lo cual está dentro de un límite razonable para una GPU.

Zero Redundancy Optimizer

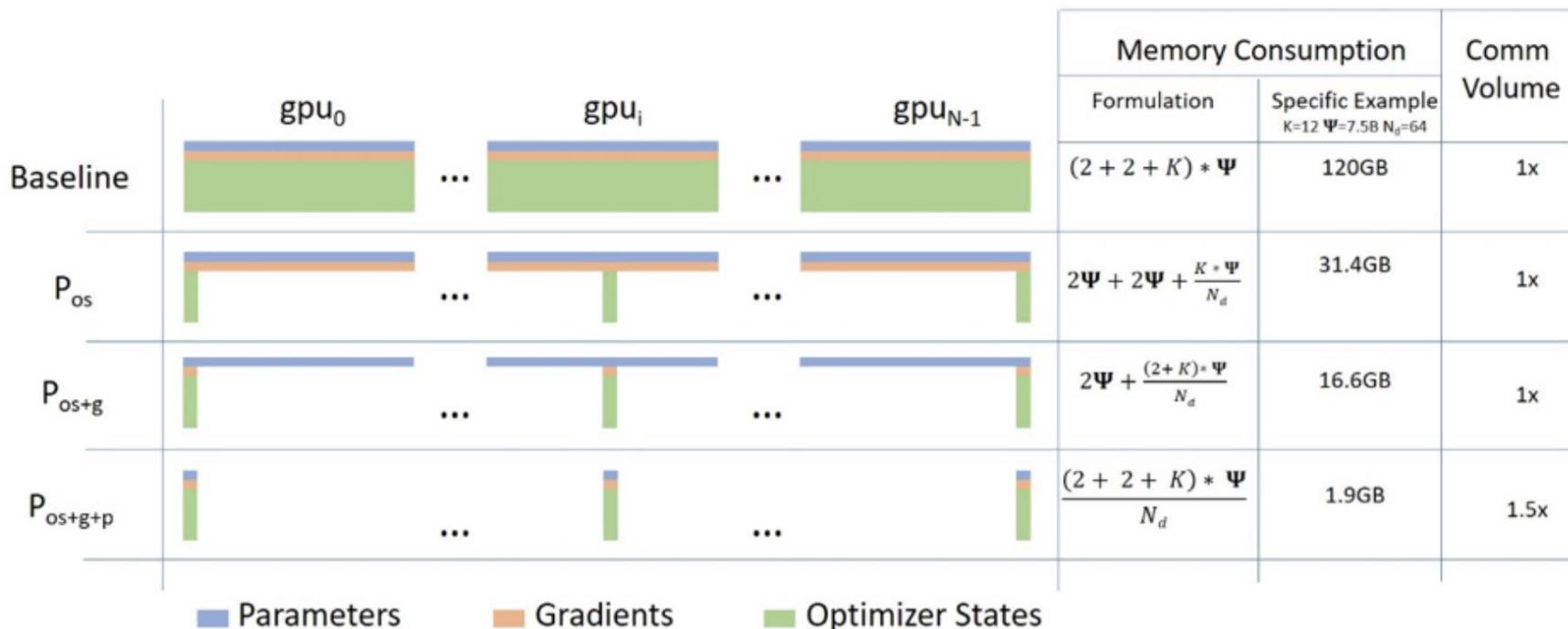


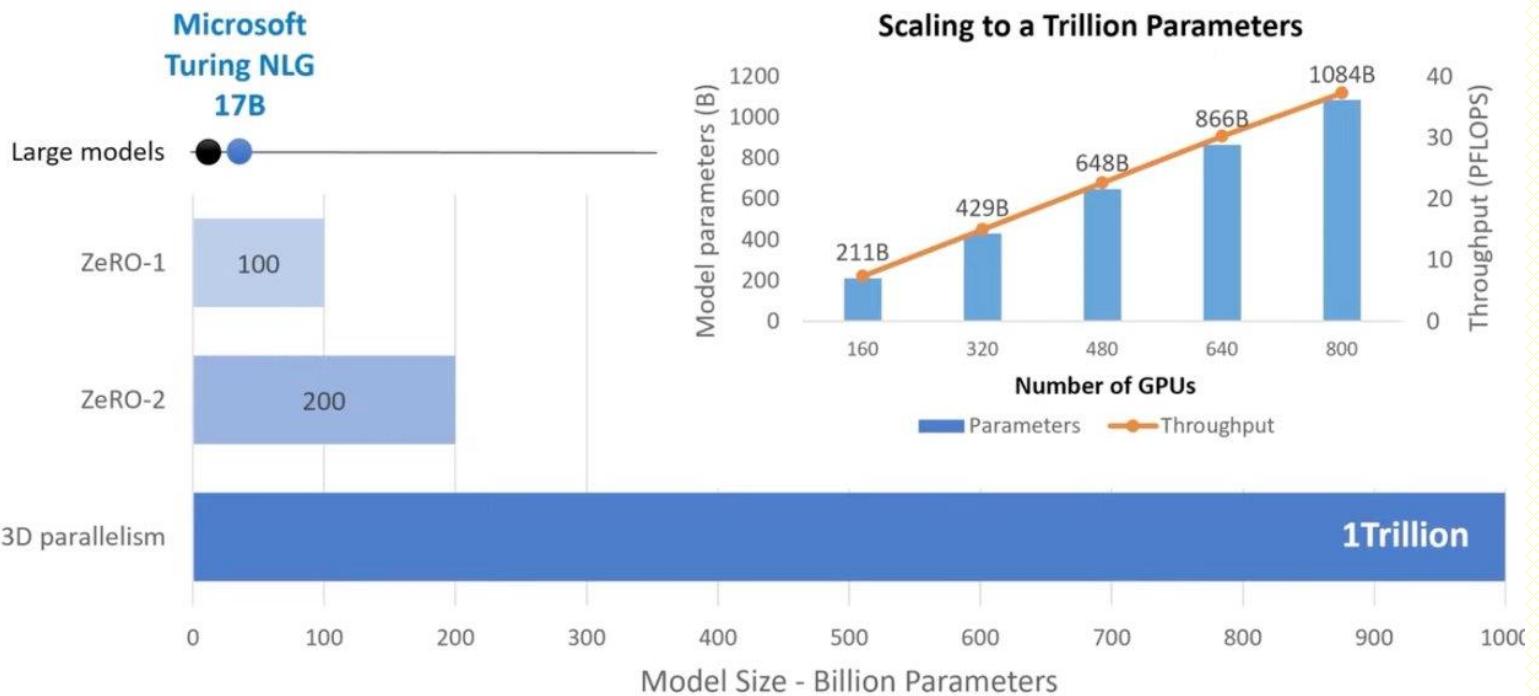
Figure 1: Memory savings and communication volume for the three stages of ZeRO compared with standard data parallel baseline. In the memory consumption formula, Ψ refers to the number of parameters in a model and K is the optimizer specific constant term. As a specific example, we show the memory consumption for a 7.5B parameter model using Adam optimizer where $K=12$ on 64 GPUs. We also show the communication volume of ZeRO relative to the baseline.

Características de DeepSpeed

Características de DeepSpeed

- Compatibilidad total con el marco de trabajo de PyTorch, facilitando su integración en proyectos existentes.
- Ajuste dinámico de la velocidad de aprendizaje para mejorar el rendimiento del modelo durante el entrenamiento.
- Capacidad de entrenar modelos más grandes que la memoria de la GPU disponible mediante técnicas de optimización.
- Adopción de compresión de gradientes para reducir la cantidad de datos necesarios en la comunicación entre dispositivos de GPU.
- Veremos: Escalabilidad, Velocidad, Entrenamientos dispersos, y Usabilidad

Escalabilidad de modelo: 10+ trillones de parámetros



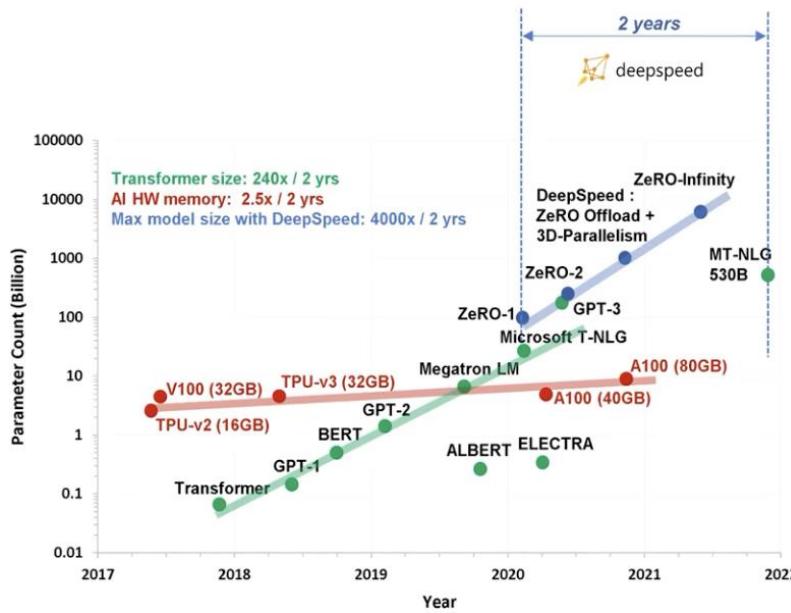
Escalabilidad de modelo: 10+ trillones de parámetros

DeepSpeed Powered Massive Models:

- METRO-LM (5.4B)
- Microsoft-Turing NLG (17B)
- GPT Neo-X (20B)
- AlexaTM (20B)
- YaLM (100B)
- GLM (130B)
- BLOOM: Big Science (176B)
- Jurasic-1 (178B)
- Megatron-Turing NLG (530B)
- ...

Key training technologies:

- Zero Redundancy Optimizer (ZeRO)
- ZeRO-Infinity
- 3D parallelism
- Memory and compute efficient MoE training
- Optimized CUDA/ROCm/CPU kernels
- Gradient compression 1-bit Adam/LAMB, 0/1 Adam
- Sparse Attention
- Mixture of quantization
- Progressive layer dropping
- Curriculum learning



*System capability to efficiently train models
with **trillions of parameters***

Money Is All You Need

Nick Debu
Tokyo Institute of Bamboo Steamer

Abstract

Transformer-based models routinely achieve state-of-the-art results on a number of tasks but training these models can be prohibitively costly, especially on long sequences. We introduce one technique to improve the performance of Transformers. We replace NVIDIA P100s by TPUs, changing its memory from hoge GB to piyo GB. The resulting model performs on par with Transformer-based models while being much more ""TSUYO TSUYO"".

SpeedUp

Fastest Transformer Kernels



World Fastest BERT Training

Scalable distributed training
through ZeRO-powered DP

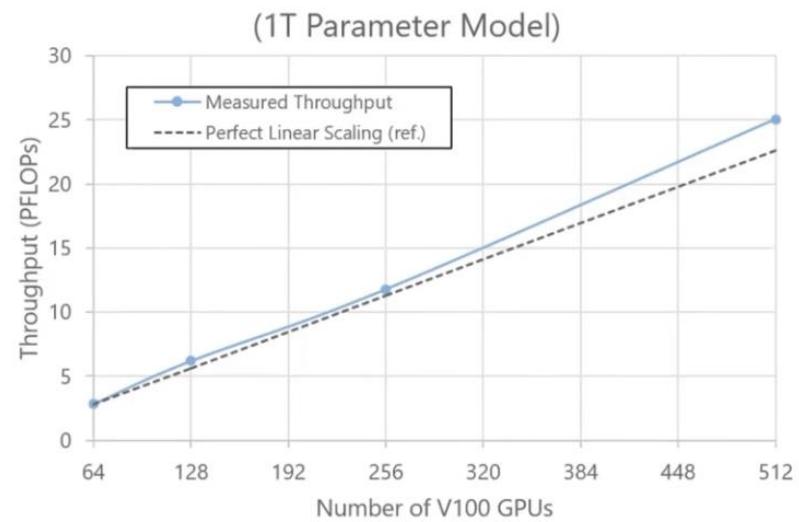


Superlinear speedup with
increasing #GPUs

DeepSpeed key technologies

- Efficiency: ZeRO, ultra-fast GPU kernels, IO/compute/communication overlapping
- Effectiveness: Advance HP tuning, large-batch scaling

#Devices	Source	Training Time
256 V100 GPUs	Nvidia	236 mins
256 V100 GPUs	DeepSpeed	144 mins
1024 TPU3 chips	Google	76 mins
1024 V100 GPUs	Nvidia	67 mins
1024 V100 GPUs	DeepSpeed	44 mins

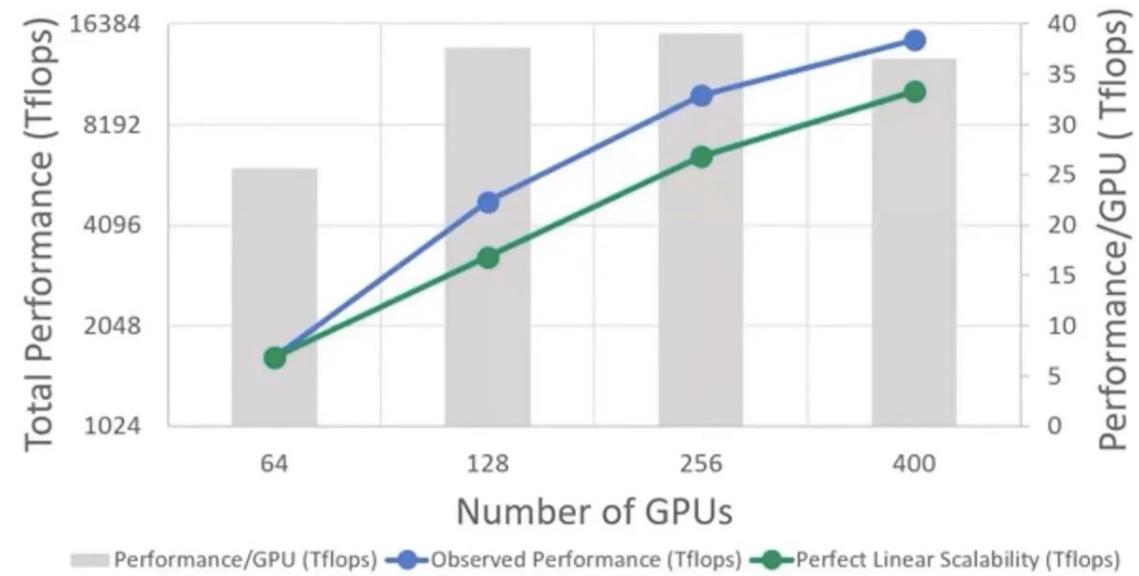


Superlinear SpeedUp

2x #resources => >2x #speedup
Aumentar #GPUS => reducir la cantidad de memoria necesaria por GPU para almacenar el estado del modelo

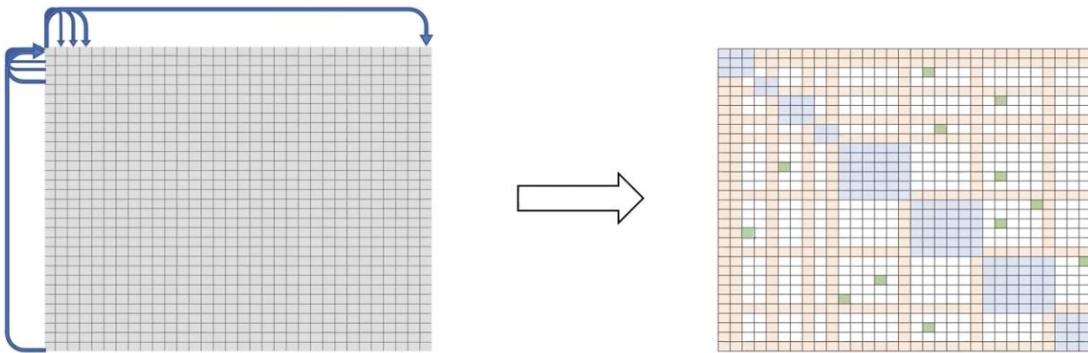
Más memoria de dispositivo disponible para las activaciones => tamaños de batch mayores y más eficiencia de GPU

Superlinear Speedup of 60B parameter model



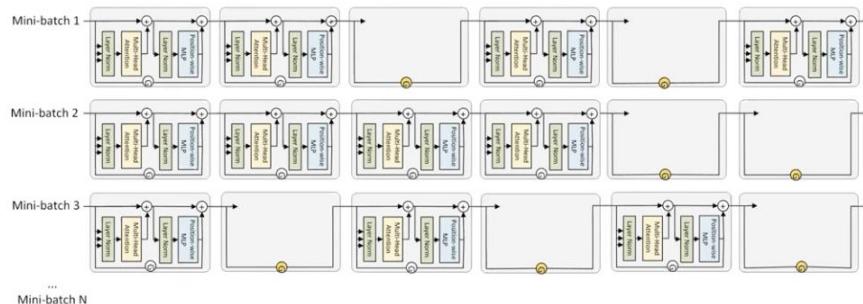
Compressed Training

- **Sparse attention:** 10x longer seq, up to 6x faster



- **Progressive Layer Drop:** Compressed robust training

- 24% faster when training the same number of samples
- 2.5X faster to get similar accuracy on downstream tasks



Compressed Training

- **SparsityConfig:** This module, is the parent class for all sparsity structures and contains the shared features of all sparsity structures. It takes the following parameters:
 - `num_heads` : an integer determining number of attention heads of the layer.
 - `block` : an integer determining the block size. Current implementation of sparse self-attention is based on blocked sparse matrices. In which this parameter defines size of such square blocks; Block X Block .
 - `different_layout_per_head` : a boolean determining if each head should be assigned a different sparsity layout; default is false and this will be satisfied based on availability.

Compressed Training

- **SparsityConfig:** this is an abstract class for sparsity structure. Any sparsity structure needs to extend this class and writes its own sparsity pattern construction; `make_layout` function. DeepSpeed currently provides the following structures that will be described in [How to config sparsity structures](#) section:
 - `FixedSparsityConfig`
 - `BSLongformerSparsityConfig`
 - `BigBirdSparsityConfig`
 - `VariableSparsityConfig`
 - `DenseSparsityConfig`

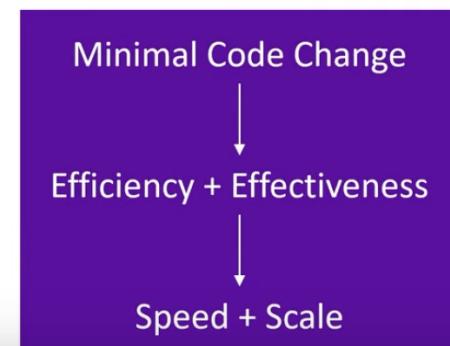
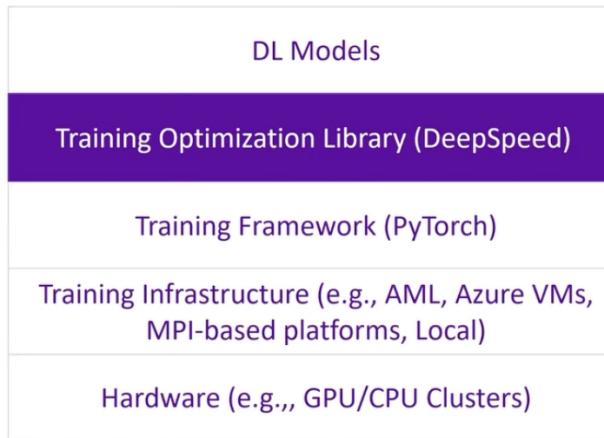
Note: Currently DeepSpeed Transformer Kernels do not support Sparse Attention. To use Sparse Attention, you need to disable Transformer Kernels!

Usabilidad



DL TRAINING OPTIMIZATION: DEEPSPEED

BERT - Original	BERT – w. DeepSpeed
# Construct distributed model model = BertMultiTask(...) Model = DistributedDataParallel(model) ...	# Construct Bert model model = BertMultiTask (...)
# Construct FP16 optimizer optimizer =FusedAdam(model_parameters,...) optimizer = FP16_Optimizer(optimizer, ...)	#Wrap to get distributed model and FP16 optimizer model, optimizer, _ = deepspeed.initialize(args=args, model=model, model_parameters=model_params, ...)
# Forward pass loss = model(batch)	# Forward pass loss = model(batch)
# Backward pass optimizer.backward(loss)	# Backward pass model.backward(loss)
# Parameter update optimizer.step()	# Parameter update model.step()



Pocos cambios necesarios para poder utilizar el modelo con DeepSpeed ->
Democratización de los entrenamientos a gran escala

Usabilidad

```
# Construct Bert model
Model = BertMultiTask(...)

# Wrap to get distributed model and FP16 optimizer
Model, optimizer, _, _ = deepspeed.initialize(
    args=args,
    model=model,
    model_parameters=model_parameters,
    ...
)

# Forward pass
loss = model(batch)

# Backward pass
model.backward(loss)

# Parameter update
model.step()
```

- Pocas líneas de código necesarias para habilitar DeepSpeed en modelos de Pytorch
- Escalable y paralelismo de datos a través de ZeRO

Usabilidad

"if you know pytorch,
you know deepspeed"

DeepSpeed/ZeRO Usability

```
# construct torch.nn.Module
model = MyModel()

# wrap w. DeepSpeed engine
engine, *_ = deepspeed.initialize(
    model=model,
    config=ds_config)

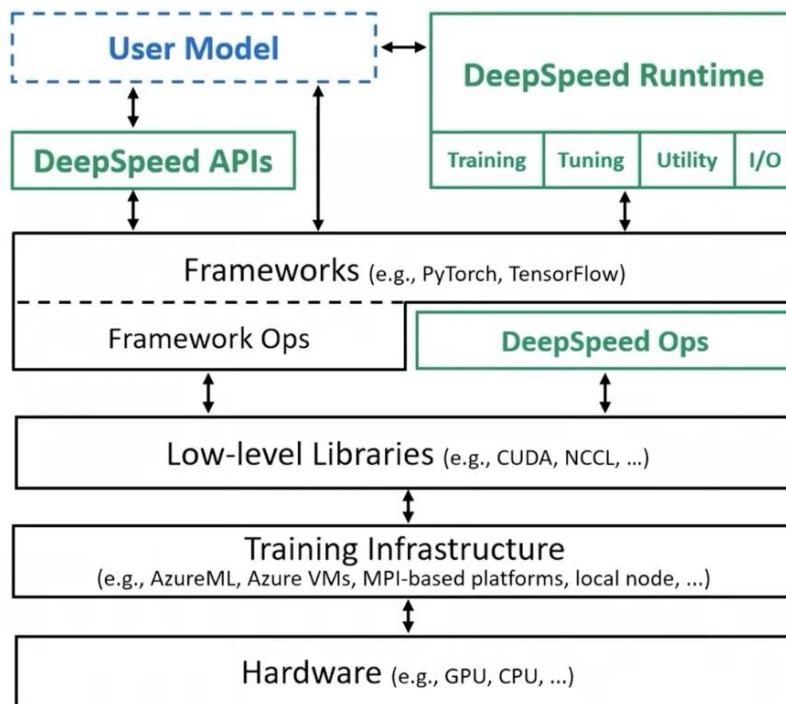
# training-loop w.r.t. engine
for batch in data_loader:
    loss = engine(batch)
    engine.backward(loss)
    engine.step()
```

```
ds_config = {
    "optimizer": {
        "type": "Adam",
        "params": {"lr": 0.001}
    },
    "zero": {
        "stage": 3,
        "offload_optimizer": {
            "device": "[cpu|nvme]"
        },
        "offload_param": {
            "device": "[cpu|nvme]"
        }
    }
}
```

Arquitectura

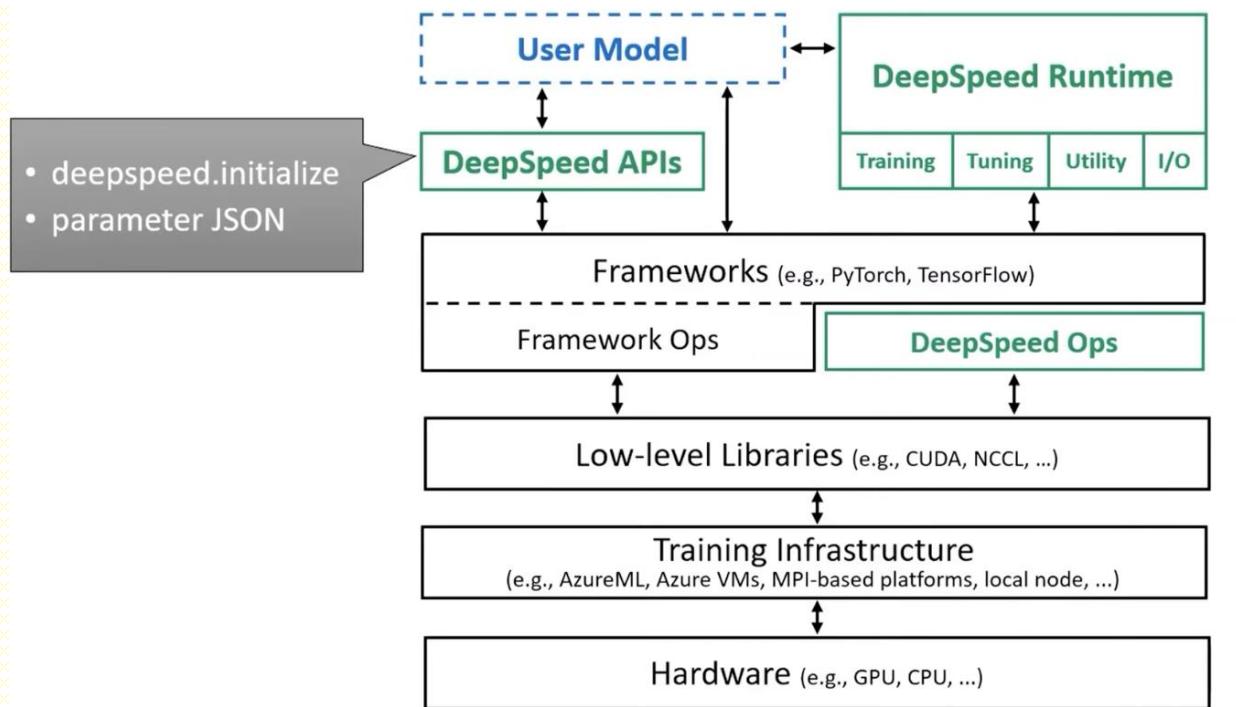
Arquitectura, 3 componentes principales

DeepSpeed Software Architecture



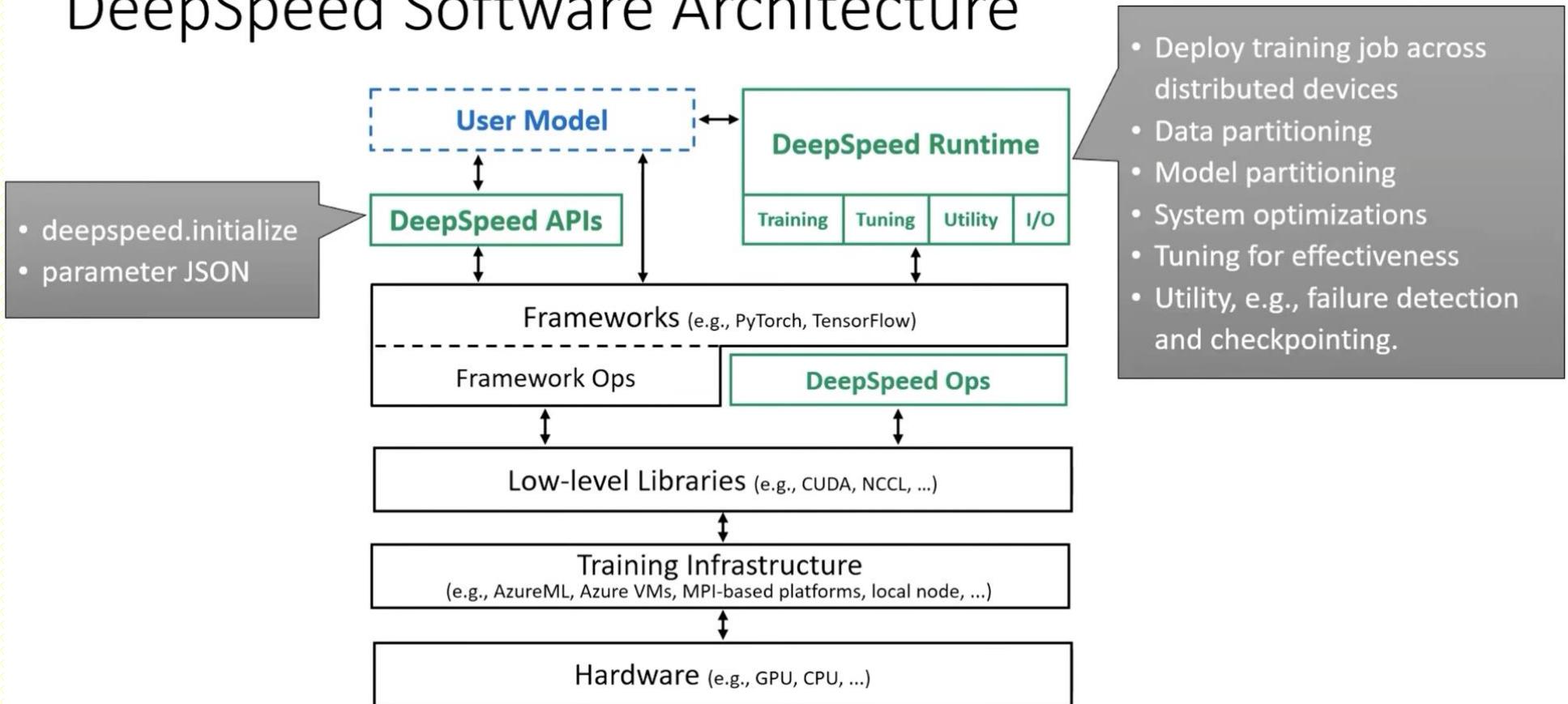
Arquitectura, 3 componentes principales

DeepSpeed Software Architecture



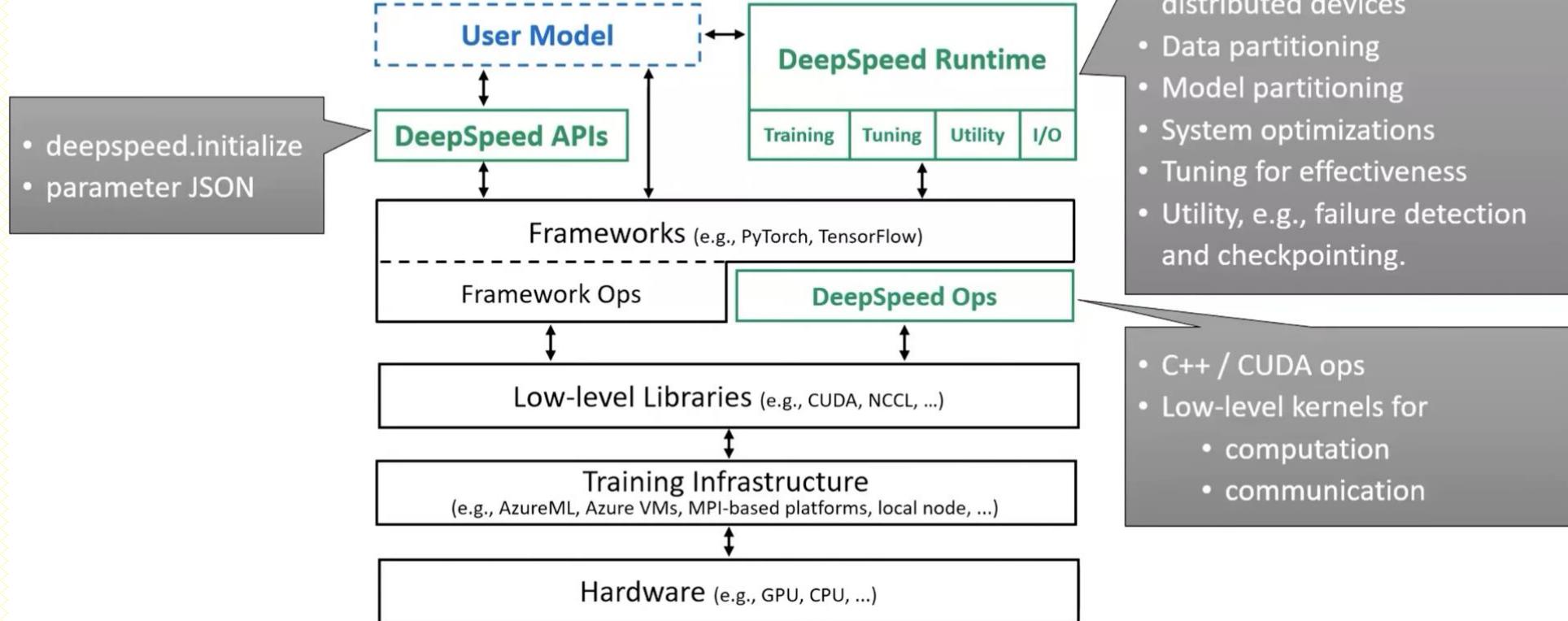
Arquitectura, 3 componentes principales

DeepSpeed Software Architecture



Arquitectura, 3 componentes principales

DeepSpeed Software Architecture



Otras técnicas implementadas

Fastest Bert Training with DeepSpeed

- DeepSpeed achieves the fastest BERT training record:
44 minutes on 1,024 NVIDIA V100 GPUs

Number of nodes	Number of V100 GPUs	DeepSpeed Time	Nvidia Time
1 DGX-2	16	33 hr 13 min	
4 DGX-2	64	8 hr 41 min	
16 DGX-2	256	144 min	236 min
64 DGX-2	1,024	44 min	67 min

Fastest Bert Training with DeepSpeed

1. Eficiencia: Optimizaciones single-GPU
 1. Kernels para transformers optimizados
 1. Fusión de Kernels
 2. Operaciones invertibles
 2. Optimizaciones del modelo
 1. Solape operaciones E/S con computación
 2. Procesamiento disperse de la salida
2. Eficacia: Convergencia del tuneado para escalado por lotes
 1. Entrenamiento con longitud de secuencia mixta (Batch size 64K)
 2. Normalización Pre-Layer
 3. LAMB optimizar
 4. Learning rate schedulers



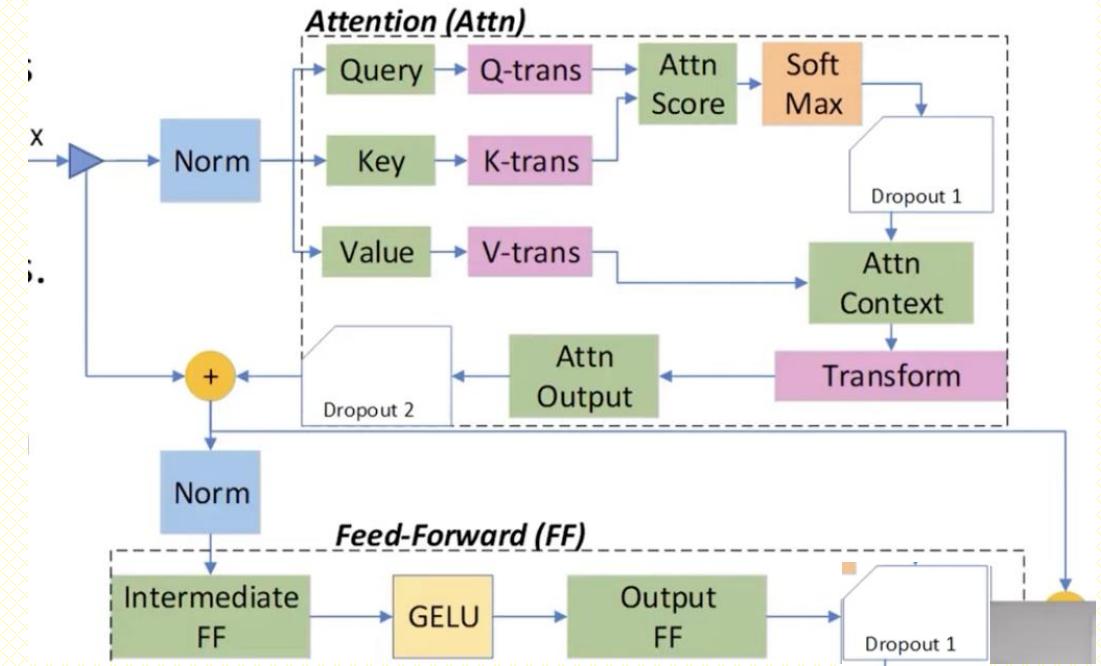
Consigue >50% del throughput pico de la NVIDIA V100 GPU



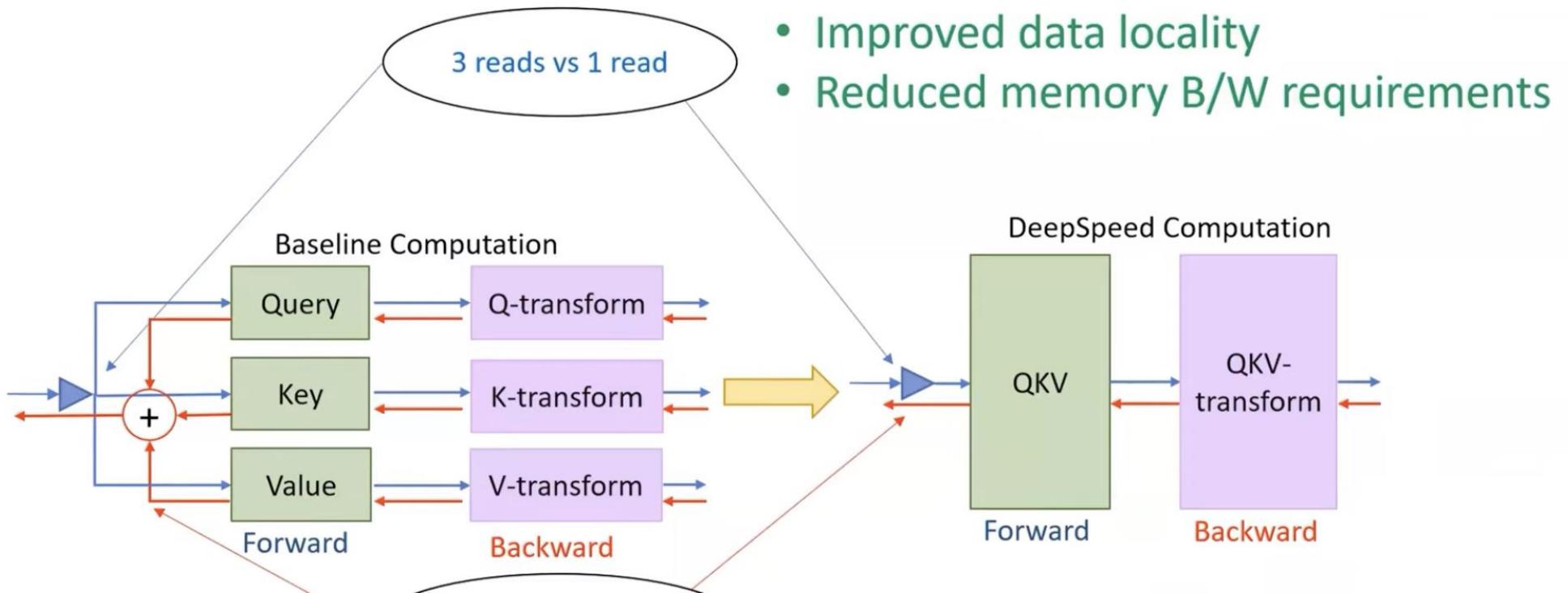
Converge con el 82% de las muestras

Optimizaciones single-GPU. Fusion de kernels

- Beneficios
 - Reducción de overheads del kernel
 - Mejora del uso de la GPU
 - Mejora de la localidad en la GPU
 - Reducción de los requerimientos B/W
- Kernels candidatos
 - Kernels paralelos
 - Kernels productor/consumidor



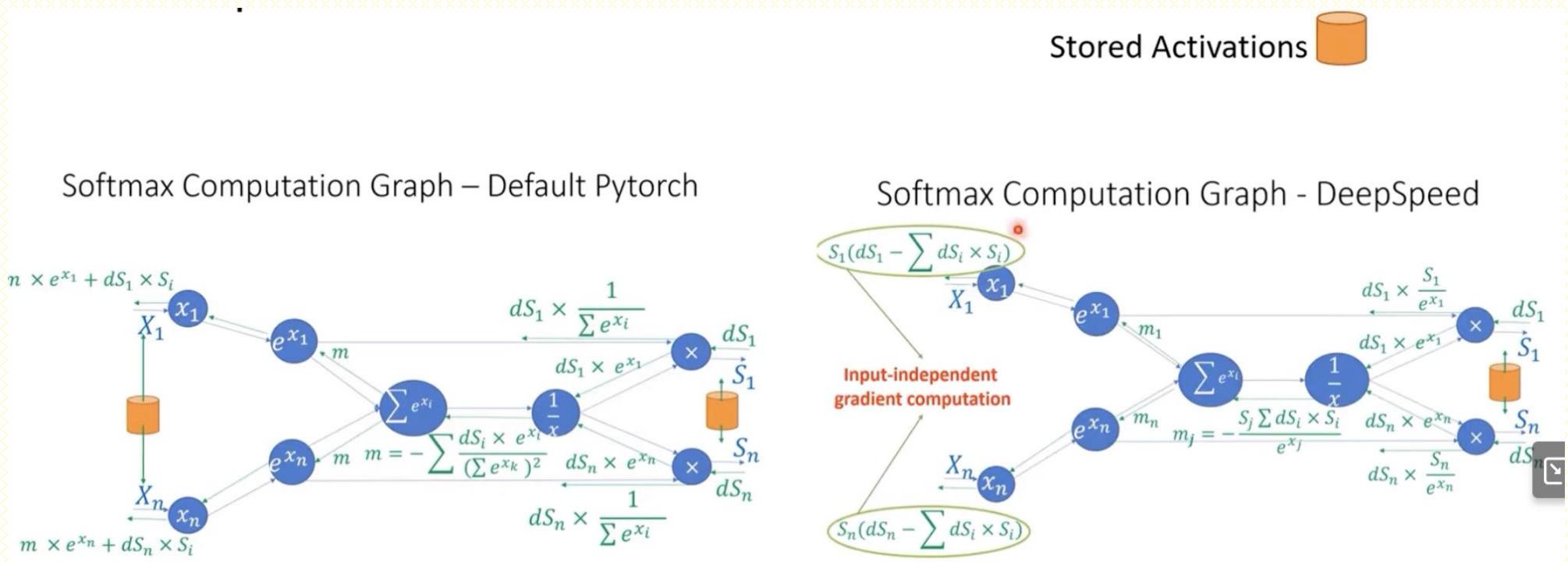
Ejemplo: Fusión de Kernels de QKV y transformación



Optimizaciones Single-GPU. Operaciones invertibles

- Cómputo de gradients sin activaciones (de entrada)
 - Reformulación algebraica
- Reducción del espacio de memoria de las activaciones
 - Evitar almacenar las activaciones de entrada para la retropropagacion
- Ejemplos en Transformers
 - Softmax
 - Layer Normalization

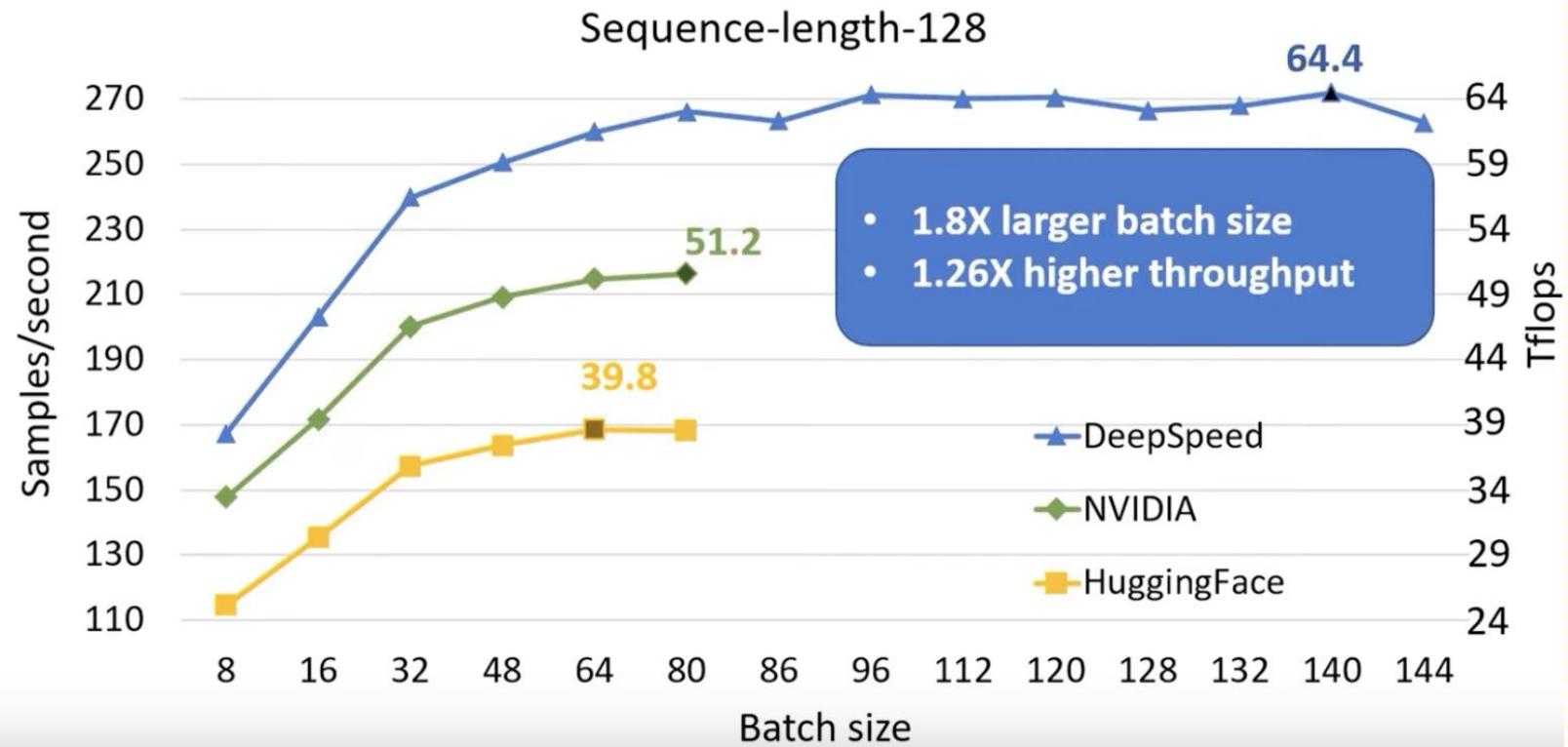
Ejemplo: Softmax invertible



Otras optimizaciones single-GPU

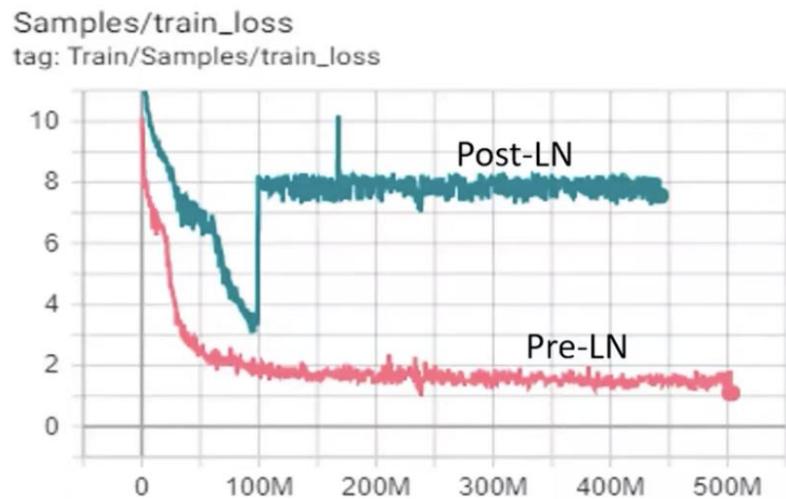
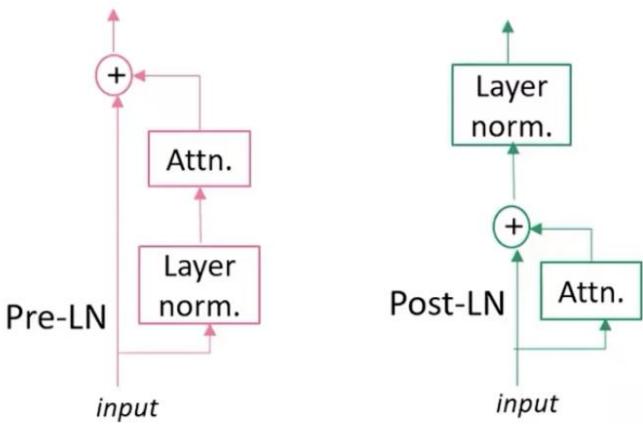
- Overlapping I/O and Computation
 - Prefetch next input batch into CPU while GPU processes current batch
 - Avoid data loading and tokenization on critical path
- Sparse output processing
 - Compute loss on only **15%** of output tokens
 - Reduce computations
 - Projection from hidden dimension to vocabulary size
 - Cross-entropy of output tokens

Single-GPU performance evaluation



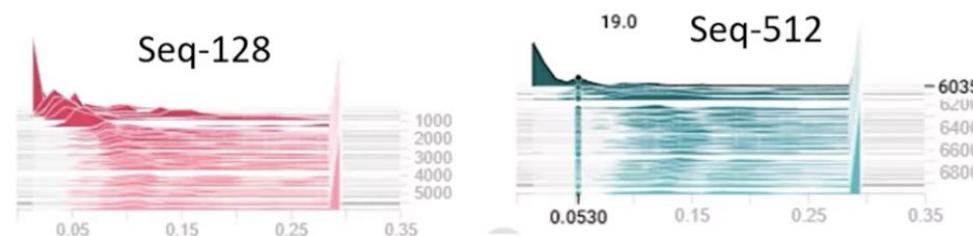
Convergencia

- Layer Normalization
 - Improve training stability for batch scaling
 - Avoid vanishing/exploding gradients
- Placement Options
 - Pre-LN: Before Attention
 - Post-LN: After Attention



Convergencia

- LAMB Optimizer
 - Layer-wise gradient adaptation
 - Configurable min/max coefficients
 - Recipe: [0.01, 0.3]
- Learning Rate schedule
 - Cyclic schedule with decay
 - Rewarm for 512-sequence length

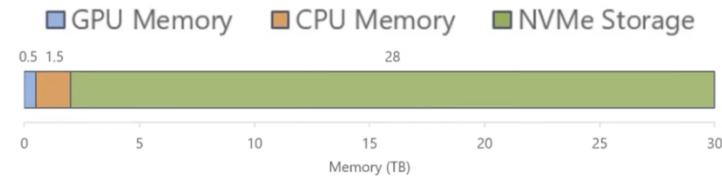


Zero-infinity

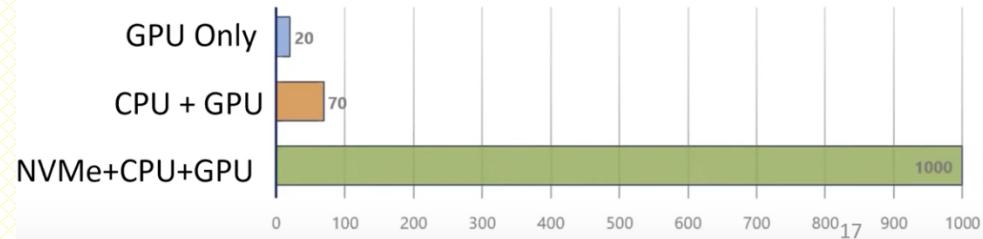
Zero-infinity: más allá de la memoria de la GPU

- Los clusters modernos cuentan con sistemas de memoria heterogéneos
- La memoria de la GPU representa una pequeña fracción
- Liberar la memoria GPU/CPU/NVMe
 - 32T de parámetros en 32 nodos
 - 1T de parámetros en un único nodo
- Se puede aplicar fine-tuning a GPT-3 en un único nodo

Memory available on a Single DGX-2 Node



Model Size on a Single DGX-2 Node



Como podemos liberar la memoria no GPU?

Podemos extender nuestra tecnología de entrenamiento paralelo para usar la memoria CPU/NVMe también?

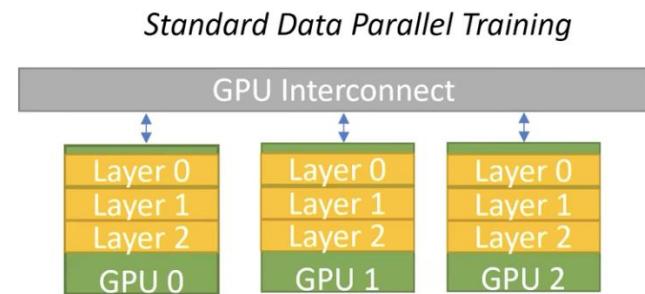
- Data Parallelism: La replicación causa una explosión de la memoria
- Tensor-Slicing: No escala más allá de un único nodo
- Pipeline-Parallelism: Requiere de mucha refactorización del Código

Qué hay de Zero Redundancy Optimizer (ZeRO)?

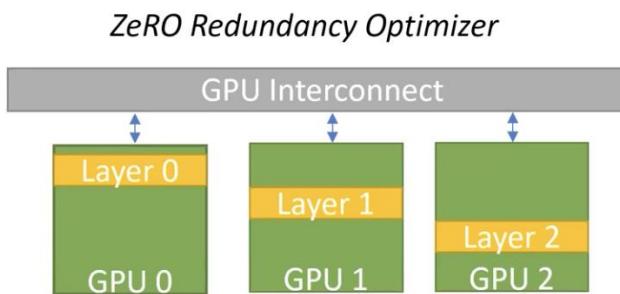
- Escala de forma eficiente a varios nodos - trillones de parámetros
- No es necesario refactorización del código

Zero Redundancy Optimizer

- Memory efficient form of data parallelism
- Each GPU stores a mutually exclusive subset of the parameters
- Broadcast parameters from owner to all the GPUs as needed



Model States mapping in **Data Parallel** Training

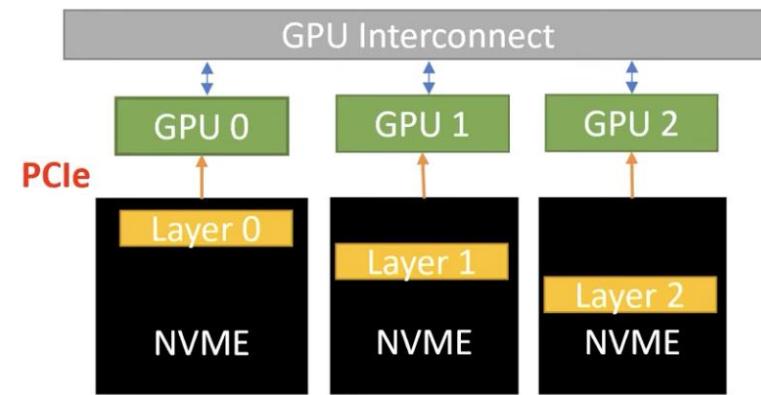


Model States mapping in **ZeRO** Training

19

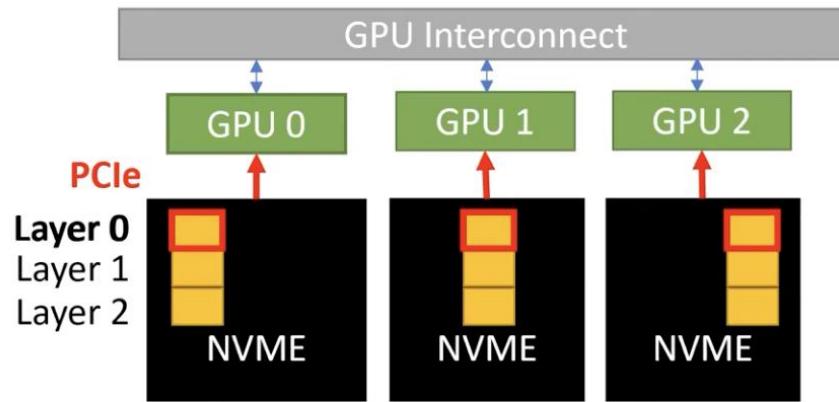
ZeRO with CPU/NVME Offload

- Store in NVME instead of GPU
 - Send from CPU/NVMe to GPU
 - Broadcast or reduce as ZeRO
-
- Is NVME \leftrightarrow GPU bandwidth sufficient?
 - Required bw for efficiency: 25-60 GB/s
 - NVMe peak bw on DGX-2: 28 GB/s



Zero-Infinity

- Partition each parameter across GPUs
 - Send from NVMe to GPU in parallel
 - Allgather and Reduce-Scatter
-
- Bandwidth Increases linearly with devices
 - Near peak 28 GB/s on a DGX-2
 - 1.8 TB/s on 64 DGX-2



ZeRO-Infinity in Action

22



Ejemplos

DeepSpeed – Pytorch example

Código Pytorch existente

- Modelo escrito usando `torch.nn.Module`

Modificaciones necesarias

- `deepspeed.initialize(..)` como envoltorio de
 - Modelo (obligatorio)
 - Optimizador (opcional)
 - LR scheduler (opcional)
 - Dataset usando `torch.data` (opcional)
- Actualizar Python parser
 - a) `parser=deepspeed.add_config_arguments(parser)`
- Devolver el wrapped model para la `forward/backward/actualizacion de parámetros`

CLASS `torch.nn.Module(*args, **kwargs)` [\[SOURCE\]](#)

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

DeepSpeed – Pytorch example

DeepSpeed launcher

Entrena un modelo por medio de:

- deepspeed
 <client_entry.py>
 <client_args>
 --deepspeed_config ds_config.json
- Ejemplo
 - deepspeed cifar10.py
--deepspeed_config ds_config.json

- Despliega trabajos de entrenamiento a través de los nodos distribuidos
- Maneja el entorno de propagación y la partición de datos
- Incorpora de forma transparente todo las optimizaciones eficientes del sistema

DeepSpeed Models: GPT2

Before

```
# Construct FP16, distributed, GPT2 model
model = GPT2Model(num_layers=args.num_layers, ...)
model = FP16_Module(model)
model = DistributedDataParallel(model, ...)

...
# Construct FP16 Adam optimizer
optimizer = Adam(param_groups, ...)
optimizer = FP16_Optimizer(optimizer, ...)

# Forward pass
output = model(tokens, ...)

# Backward pass
optimizer.backward(loss)

# Parameter update
optimizer.step()
```

After

```
# Construct GPT2 model
model = GPT2Model(num_layers=args.num_layers, ...)

# Construct Adam optimizer
optimizer = Adam(param_groups, ...)

# Wrap model, optimizer, and lr scheduler
model, optimizer, lr_scheduler, _ = deepspeed.initialize(
    args=args, model=model, optimizer=optimizer,
    lr_scheduler=lr_scheduler, mpu=mpu
)
```

```
# Forward pass
output = model(tokens, ...)

# Backward pass
model.backward(loss)

# Parameter update
model.step()
```



DeepSpeed Models: GPT2 JSON config

- Parámetros de DeepSpeed para GPT2
 - Tamaño de Batch
 - Optimizador Adam
 - Mixed precision
 - Habilitar optimización ZeRO

```
{  
    "train_batch_size": 512,  
    "train_micro_batch_size_per_gpu": 16,  
  
    "optimizer": {  
        "type": "Adam",  
        "params": {  
            "lr": 0.00015  
        }  
    },  
    "gradient_clipping": 1.0,  
  
    "fp16": {  
        "enabled": true,  
        "loss_scale": 0,  
        "loss_scale_window": 1000,  
        "hysteresis": 2,  
        "min_loss_scale": 1  
    },  
  
    "zero_optimization": true  
}
```

Ejercicios

- https://github.com/diegoandradecanosa/Cesga2023Courses/tree/main/pytorch_dist/deepspeed/translation
- <https://github.com/microsoft/DeepSpeedExamples>