Kazuki Osawa[1*], Shigang Li[2], Torsten Hoefler[1]

1: ETH Zurich (*: currently with Google DeepMind), 2: Beijing University of Posts and Telecommunications

PipeFisher: Efficient Training of Large Language Models Using Pipelining and Fisher Information Matrices

SPCL

Large Language Models

ChatGPT, https://chat.openai.com/

Image: Yang et al., "Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond", 2023.

k What are Large Language Models?

Large Language Models (LLMs) are powerful artificial intelligence systems that are designed to understand and generate human-like text. These models, such as OpenAI's GPT-3, are trained on vast amounts of text data to learn the statistical patterns and structures of language.

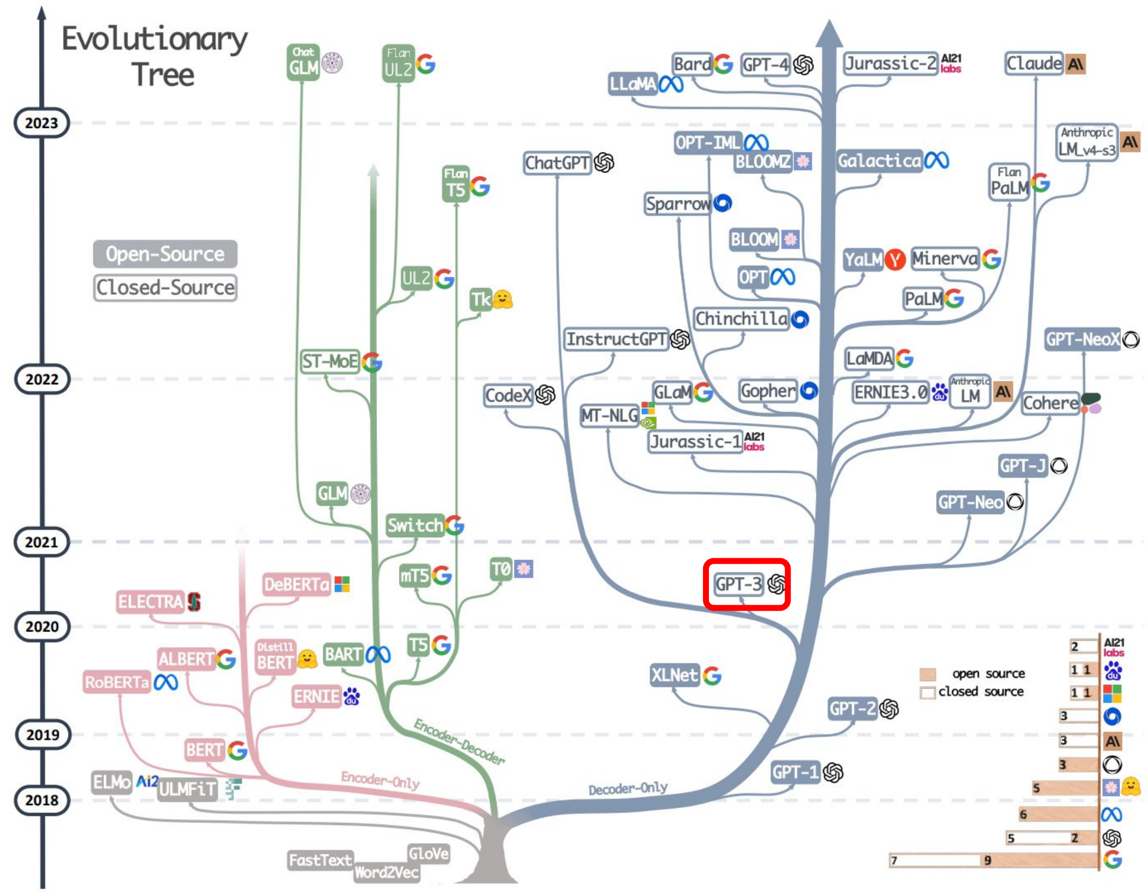
LLMs are based on deep learning techniques, particularly using a type of neural network called a transformer. Transformers allow LLMs to process and generate text in a hierarchical manner, taking into account the relationships between words, phrases, and sentences.

The training process for LLMs involves exposing the model to massive amounts of text from diverse sources such as books, websites, and articles. By predicting the next word in a sentence or completing a given prompt, the model learns to generate coherent and contextually appropriate responses.

(GPT-3: 300B tokens)

LLMs have a wide range of applications. They can be used for natural language processing tasks such as text classification, sentiment analysis, language translation, summarization, and question answering. They can also be employed in creative tasks like generating human-like text, poetry, or even composing music.

GPT-3, for example, with 175 billion parameters, is one of the largest LLMs developed to date. These massive models have demonstrated impressive language understanding and generation capabilities, although they also raise concerns related to ethical considerations, biases, and potential misuse.



Data scale ⤴
> 1T tokens
(e.g., Chinchilla, LLaMA)

+

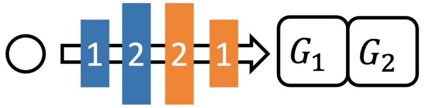
Model size ⤴
> 1T parameters
(e.g., GLaM)

⇨

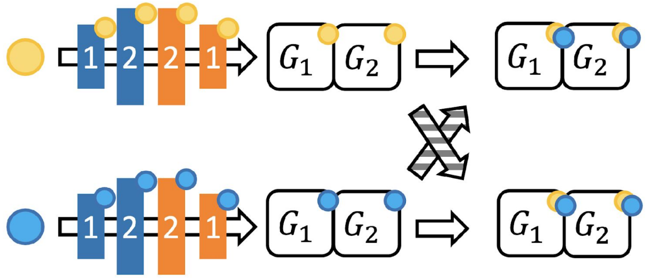
Cost ⤴
> 40K A100 GPU days
(e.g., BLOOM)

Parallelism for massive data and models

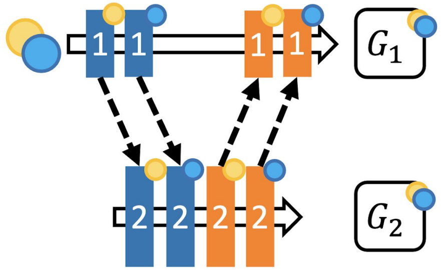
(i) No parallelism
(w/ 1 device)



(ii) Data parallelism
(w/ 2 devices)



(iii) Pipeline parallelism
(w/ 2 devices)



and more model parallelism (ZeRO, Megatron, MoE, etc)

Forward pass | Backward pass

pre activation

$$h_l = W_l a_{l-1}$$
$$(a_0 = x)$$

$$\frac{\partial \mathcal{L}}{\partial a_{l-1}} = W_l^{\,T} \frac{\partial \mathcal{L}}{\partial h_l}$$

activation

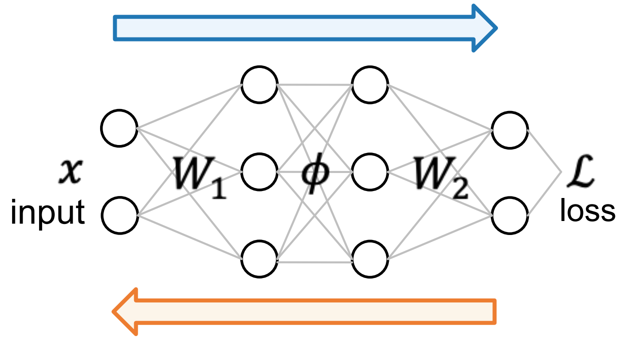
$$a_l = \phi(h_l)$$

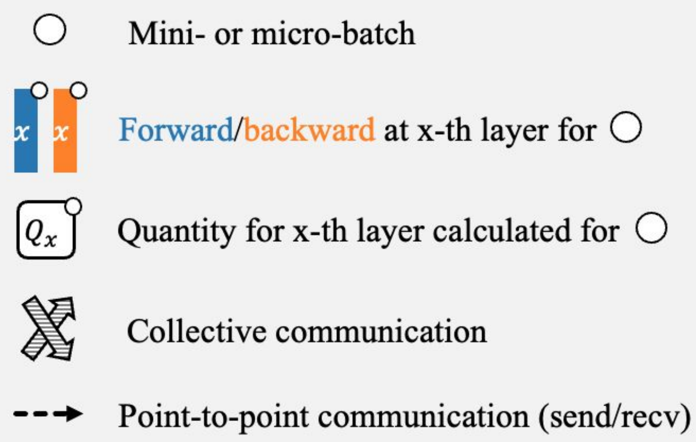
error

$$e_l = \frac{\partial \mathcal{L}}{\partial h_l} = \phi'\left(\frac{\partial \mathcal{L}}{\partial a_l}\right)$$

gradient

$$G_l = \frac{\partial \mathcal{L}}{\partial W_l} = e_l a_{l-1}^T$$



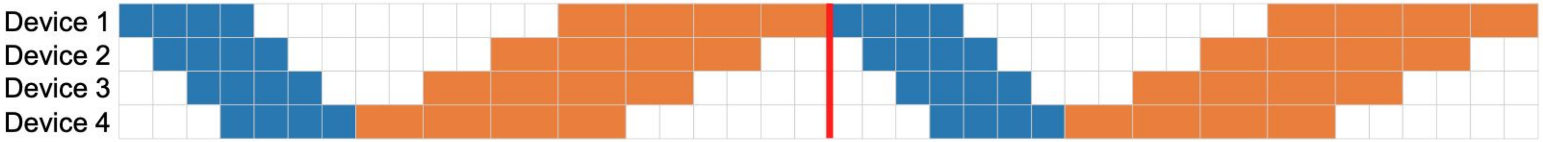


Bubbles in pipeline

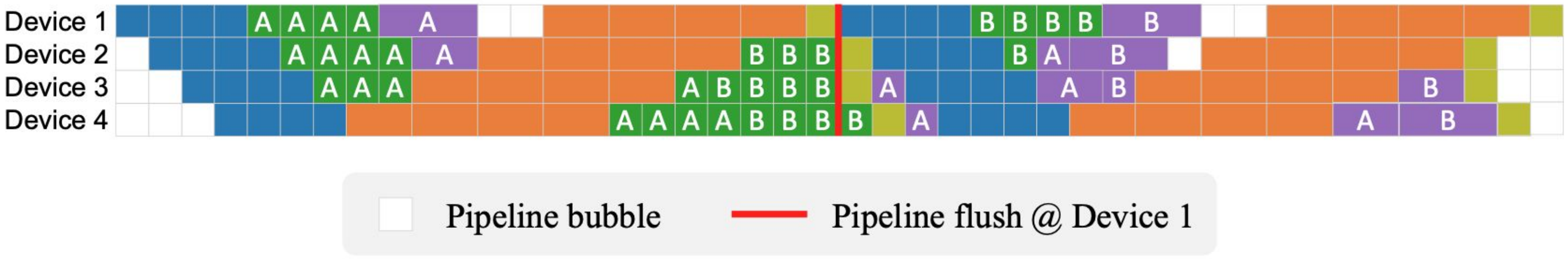
- Pipelining creates **bubbles** of time in which accelerators become idle.
- The overhead of pipelining mainly comes from the **low utilization of accelerators**.
- 💡 We can assign extra work to the bubbles to gain auxiliary benefits.
- [Our approach] **PipeFisher** *automatically* assigns the work of K-FAC (a second-order optimization method based on the Fisher information matrix) to the bubbles for accelerating training.

(4 pipeline stages, 4 micro-batches)

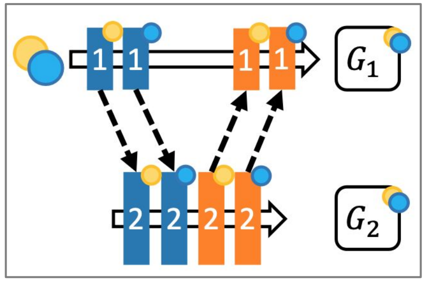
(a) GPipe (Huang et al.,2018)



(b) **PipeFisher** for GPipe



(2 pipeline stages, 2 micro-batches)



Why second-order optimization?

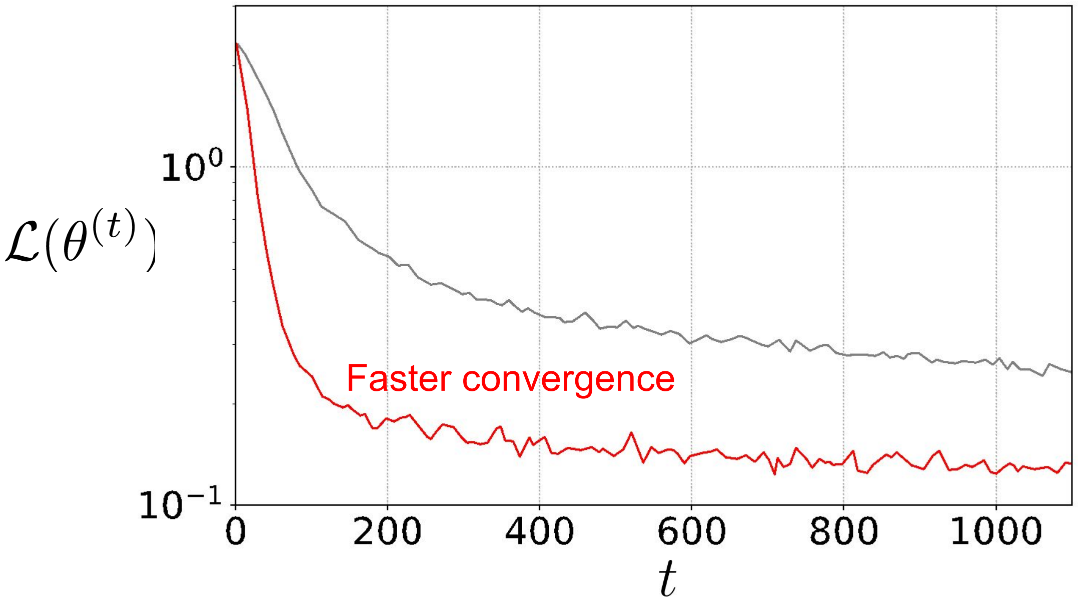
First-order Optimization (gradient descent)

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \nabla \mathcal{L}(\theta^{(t)})$$

Second-order Optimization

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta C^{-1} \nabla \mathcal{L}(\theta^{(t)})$$

Precondition the gradient by the curvature matrix



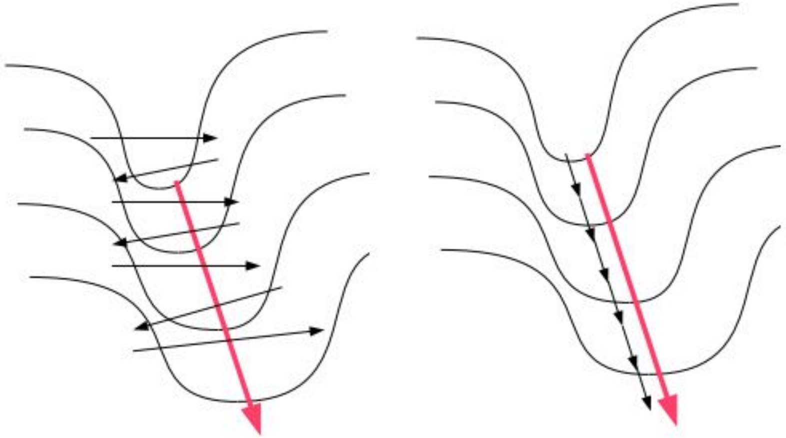
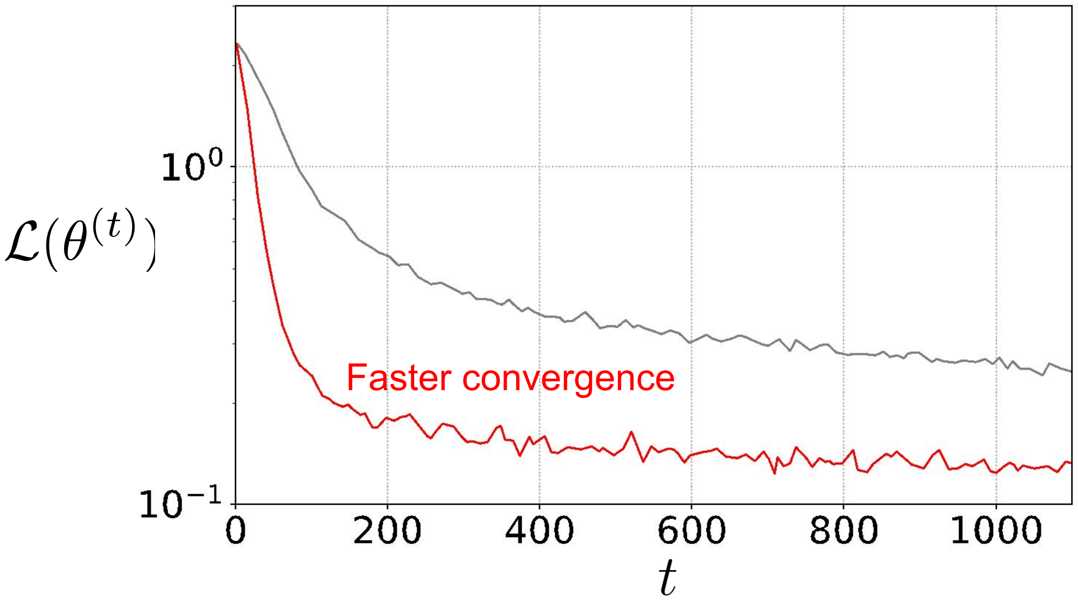


Figure from J. Martens, 2010

Why second-order optimization?

First-order Optimization (gradient descent)

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \nabla \mathcal{L}(\theta^{(t)})$$



Second-order Optimization

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta C^{-1} \nabla \mathcal{L}(\theta^{(t)})$$

Precondition the gradient by the curvature matrix

C ⇒
- Hessian -> Newton method
- Fisher information matrix -> Natural gradient method [Amari, 1998]
  (= positive-semi definite approx. of Hessian)

Why is second-order optimization *unpopular* in DL?

	First-order optimization		Second-order optimization	
Forward/backward	$\nabla\mathcal{L} \in \mathbb{R}^P$		$\nabla\mathcal{L} \in \mathbb{R}^P$	
		Curvature	$C \in \mathbb{R}^{P\times P}$	Overhead $\mathcal{O}(P^3)$
		Inverse	$C^{-1} \in \mathbb{R}^{P\times P}$	
		Precondition	$C^{-1}\nabla\mathcal{L} \in \mathbb{R}^P$	
Update	$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta\nabla\mathcal{L}(\theta^{(t)})$		$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta C^{-1}\nabla\mathcal{L}(\theta^{(t)})$	

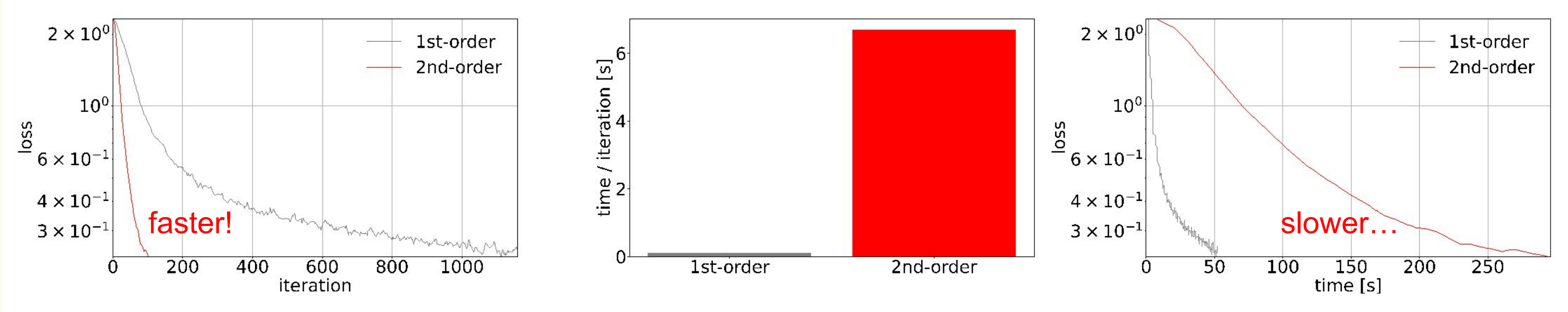
Why is second-order optimization *unpopular* in DL?

	First-order optimization	Second-order optimization
Forward/backward	$\nabla \mathcal{L} \in \mathbb{R}^P$	$\nabla \mathcal{L} \in \mathbb{R}^P$

Overhead $\mathcal{O}(P^3)$

$\mathcal{L}(\theta^{(t)})$

Training a 3-layered MLP on MNIST

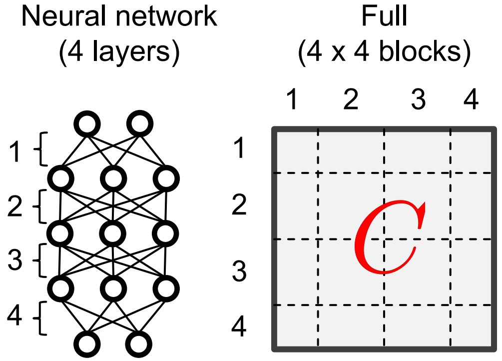


Iterations to converge × Time / iteration = Total time

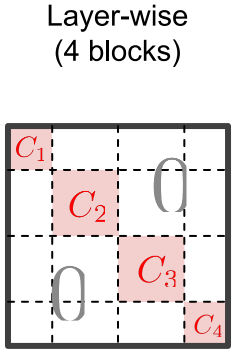
Kronecker-factored Approximate Curvature (K-FAC)

Martens and Grosse, 2015

Step1. Layer-wise block-diagonal

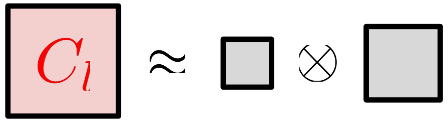


≈



$\mathcal{O}(P^3) \longrightarrow \mathcal{O}(P_l^3)$

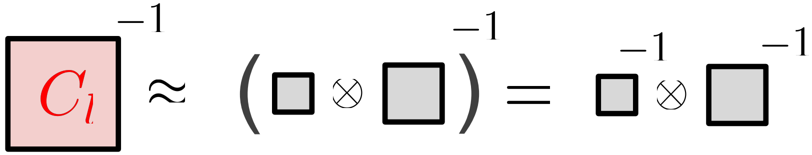
Step2. Kronecker-factorization (for each layer)



Kronecker product

$\boldsymbol{A} \in \mathbb{R}^{d_1 \times d_2}$ $\boldsymbol{B} \in \mathbb{R}^{d_3 \times d_4}$

$$\boldsymbol{A} \otimes \boldsymbol{B} := \begin{bmatrix} A_{1,1}\boldsymbol{B} & \cdots & A_{1,d_2}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ A_{d_1,1}\boldsymbol{B} & \cdots & A_{d_1,d_2}\boldsymbol{B} \end{bmatrix} \in \mathbb{R}^{d_1 d_3 \times d_2 d_4}$$

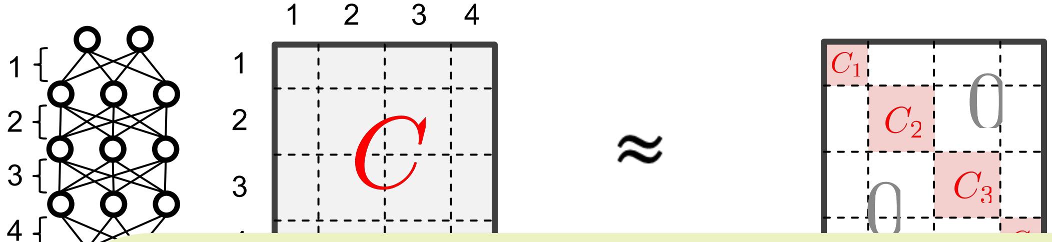


$\mathcal{O}(P_l^3) \longrightarrow \mathcal{O}(P_l^{3/2})$

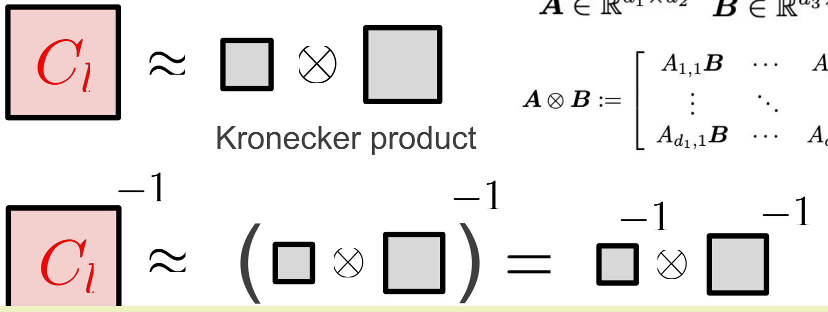
Kronecker-factored Approximate Curvature (K-FAC)

Martens and Grosse, 2015

Step1. Layer-wise block-diagonal



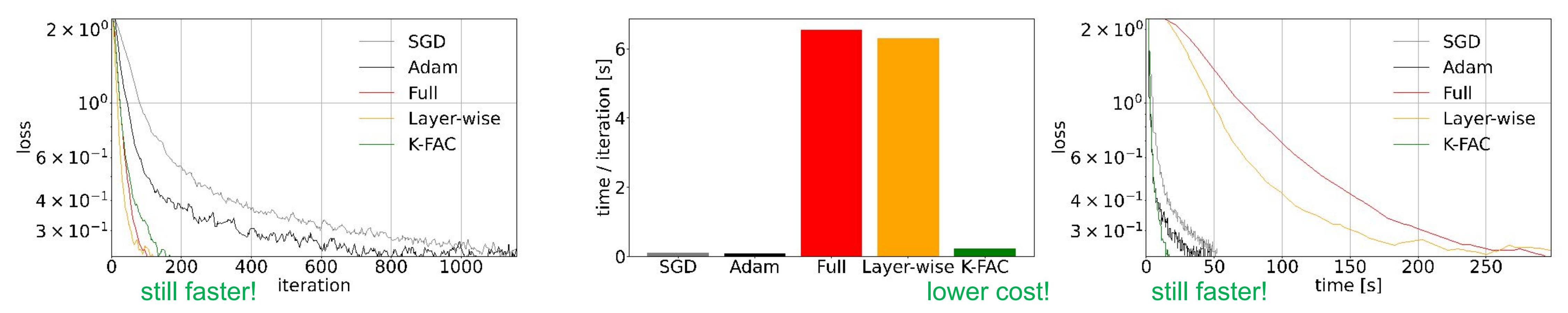
Step2. Kronecker-factorization (for each layer)



$\boldsymbol{A} \in \mathbb{R}^{d_1 \times d_2}$ $\boldsymbol{B} \in \mathbb{R}^{d_3 \times d_4}$

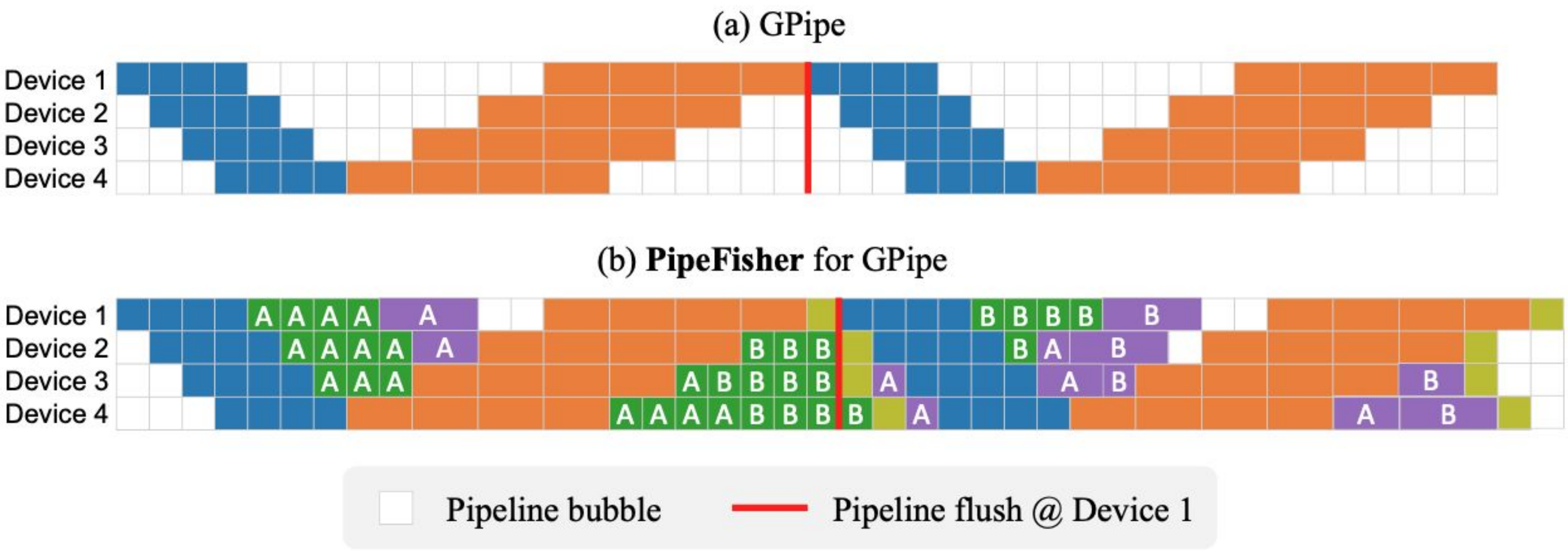
$$\boldsymbol{A} \otimes \boldsymbol{B} := \begin{bmatrix} A_{1,1}\boldsymbol{B} & \cdots & A_{1,d_2}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ A_{d_1,1}\boldsymbol{B} & \cdots & A_{d_1,d_2}\boldsymbol{B} \end{bmatrix} \in \mathbb{R}^{d_1 d_3 \times d_2 d_4}$$

Training a 3-layered MLP ($P_1 = 12560, P_2 = 136, P_3 = 90$) on MNIST



Iterations to converge × Time / iteration = Total time

PipeFisher



What exactly are the work of K-FAC?

Work of K-FAC

(mini-batch) SGD

batched error, batched activation

$$G_l = \frac{\partial \mathcal{L}}{\partial W_l} = U_{e,l} \cdot U_{a,l-1}^\top$$

$(d_l^{out} \times B)\,(B \times d_l^{in})$ B : mini-batch size

$$g_l = vec(G_l) \in \mathbb{R}^{P_l}$$

$$P_l = d_l^{out} \cdot d_l^{in}$$

Forward pass

pre activation

$$h_l = W_l a_{l-1} \quad (a_0 = x)$$

activation

$$a_l = \phi(h_l)$$

Backward pass

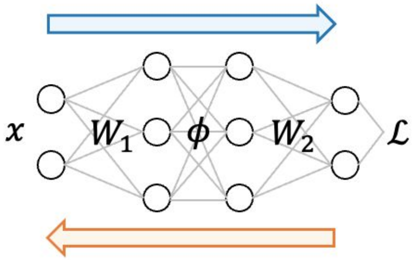
$$\frac{\partial \mathcal{L}}{\partial a_{l-1}} = W_l^T \frac{\partial \mathcal{L}}{\partial h_l}$$

error

$$e_l = \frac{\partial \mathcal{L}}{\partial h_l} = \phi'\left(\frac{\partial \mathcal{L}}{\partial a_l}\right)$$

gradient

$$G_l = \frac{\partial \mathcal{L}}{\partial W_l} = e_l a_{l-1}^T$$



K-FAC

$$g_l^{kfac} \approx F_l^{-1} g_l$$

Fisher block for l-th layer

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta C^{-1} \nabla \mathcal{L}(\theta^{(t)})$$

Precondition the gradient by the curvature matrix

Step1. Layer-wise block-diagonal

Full (4 x 4 blocks) $\approx$ Layer-wise (4 blocks)

$$\mathcal{O}(P^3) \longrightarrow \mathcal{O}(P_l^3)$$

Step2. Kronecker-factorization (for each layer)

$$\boxed{C_l} \approx \square \otimes \square$$

Kronecker product

$$\boldsymbol{A} \in \mathbb{R}^{d_1 \times d_2} \quad \boldsymbol{B} \in \mathbb{R}^{d_3 \times d_4}$$

$$\boldsymbol{A} \otimes \boldsymbol{B} := \begin{bmatrix} A_{1,1}\boldsymbol{B} & \cdots & A_{1,d_2}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ A_{d_1,1}\boldsymbol{B} & \cdots & A_{d_1,d_2}\boldsymbol{B} \end{bmatrix} \in \mathbb{R}^{d_1 d_3 \times d_2 d_4}$$

$$\boxed{C_l}^{-1} \approx \left(\square \otimes \square\right)^{-1} = \square^{-1} \otimes \square^{-1}$$

$$\mathcal{O}(P_l^3) \longrightarrow \mathcal{O}(P_l^{3/2})$$

Work of K-FAC

(mini-batch) SGD

batched error

batched activation

$$G_l = \frac{\partial \mathcal{L}}{\partial W_l} = U_{e,l} \cdot U_{a,l-1}^{\top}$$

$(d_l^{out} \times B)\,(B \times d_l^{in})$ $\quad B$: mini-batch size

$$g_l = vec(G_l) \in \mathbb{R}^{P_l}$$

$$P_l = d_l^{out} \cdot d_l^{in}$$

Forward pass

pre activation

$h_l = W_l a_{l-1}$
$(a_0 = x)$

activation

$a_l = \phi(h_l)$

Backward pass

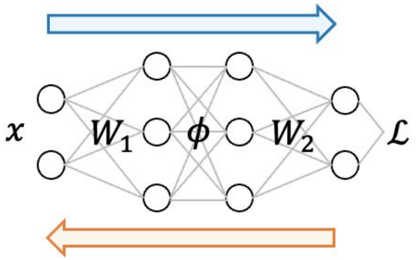
$\frac{\partial \mathcal{L}}{\partial a_{l-1}} = W_l^T \frac{\partial \mathcal{L}}{\partial h_l}$

error

$e_l = \frac{\partial \mathcal{L}}{\partial h_l} = \phi'\left(\frac{\partial \mathcal{L}}{\partial a_l}\right)$

gradient

$G_l = \frac{\partial \mathcal{L}}{\partial W_l} = e_l a_{l-1}^T$



K-FAC

$$g_l^{kfac} \approx F_l^{-1} g_l \approx (A_l \otimes B_l)^{-1} g_l = (A^{-1}{}_l \otimes B^{-1}{}_l) g_l = vec(B^{-1}{}_l G_l A^{-1}{}_l)$$

Curvature work

```
A = Ua @ Ua.T
B = Ue @ Ue.T
```

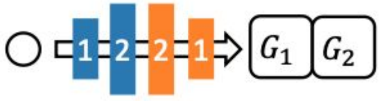
Inversion work

```
Ainv = inverse(A)
Binv = inverse(B)
```

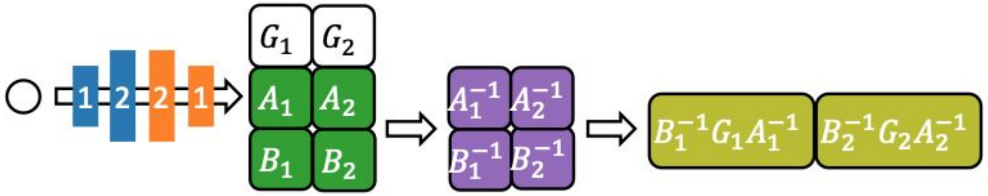
Precondition work

```
Gkfac = Binv @ G @ Ainv
```

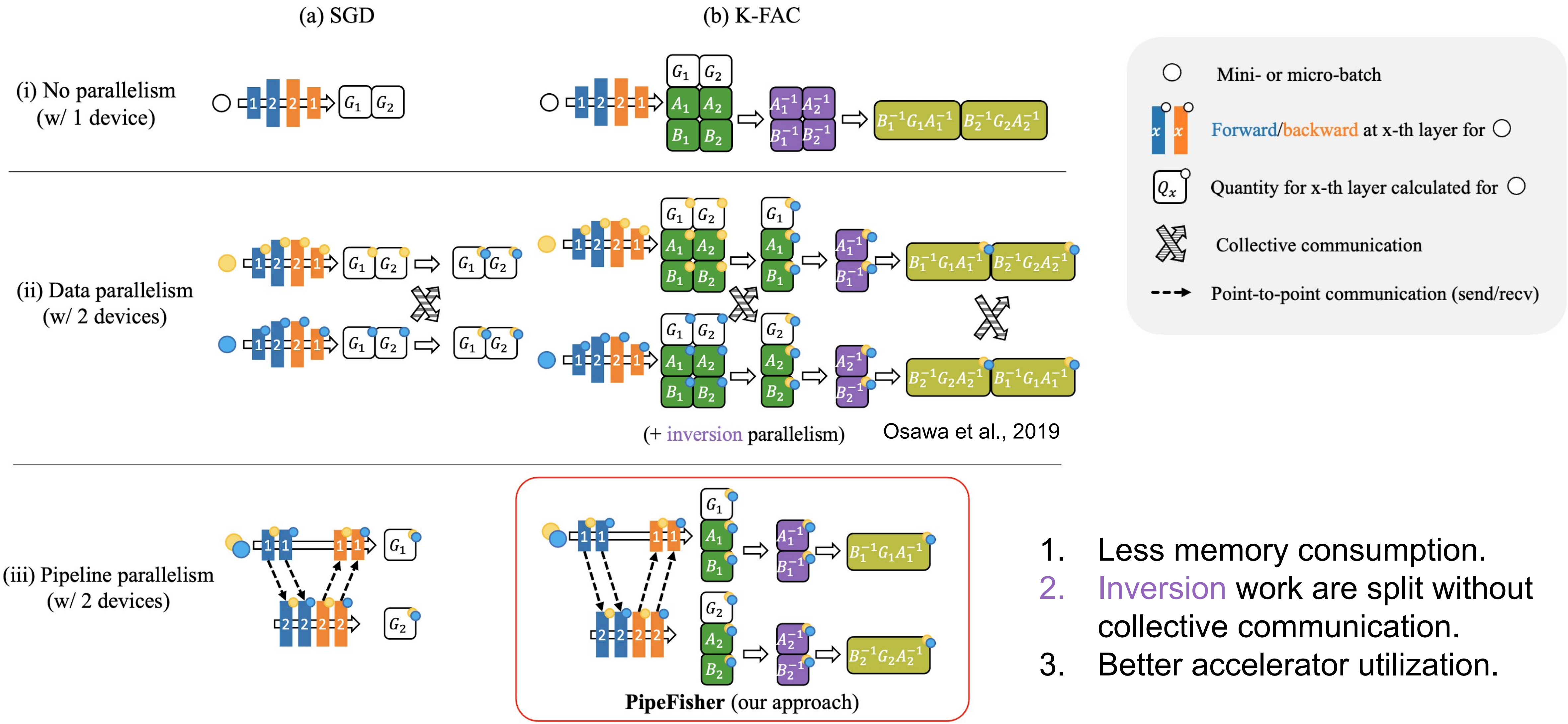
(a) SGD



(b) K-FAC



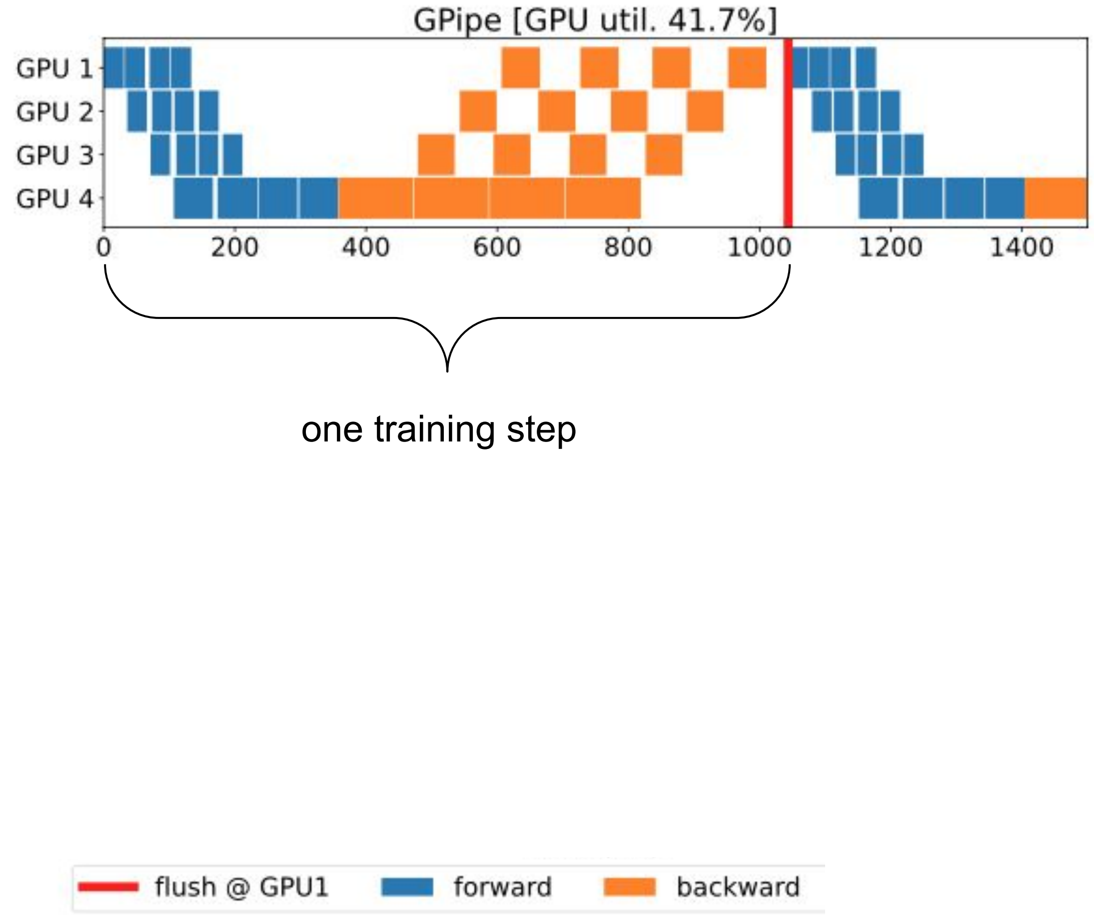
K-FAC with parallelism



1. Less memory consumption.
2. Inversion work are split without collective communication.
3. Better accelerator utilization.

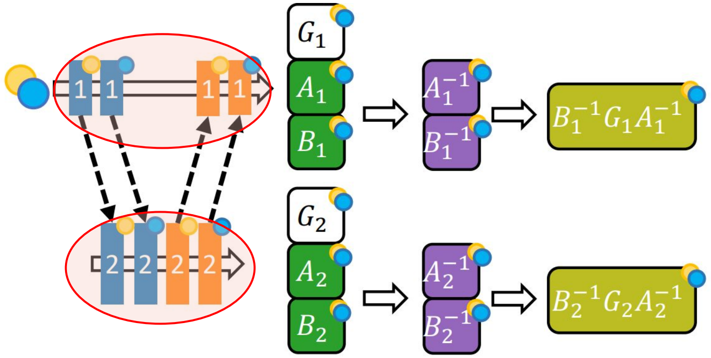
Profiled results (1/2) [GPipe (Huang et al, 2018)]

- BERT-Base (12 Transformer layers) w/ 4 pipeline stages (3 layers per stage) and 4 micro-batches
- CUDA kernel execution times on NVIDIA P100 GPUs



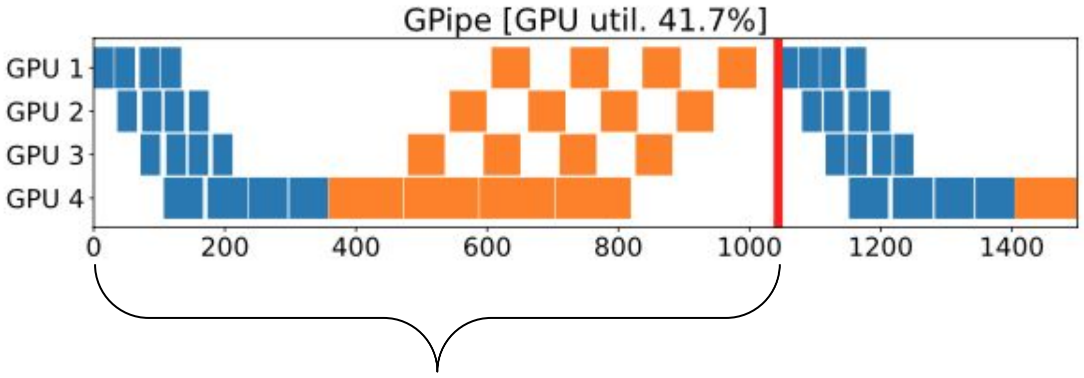
PipeFisher

- ❑ Step 1: **Measure** the times for the forward/backward works and <u>bubbles</u> in a training step.



Profiled results (1/2) [GPipe (Huang et al, 2018)]

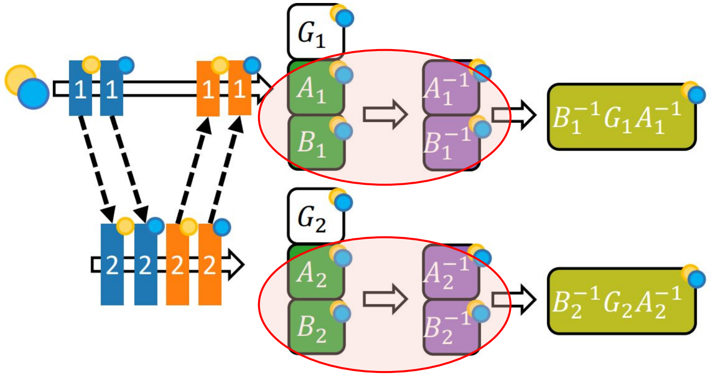
- BERT-Base (12 Transformer layers) w/ 4 pipeline stages (3 layers per stage) and 4 micro-batches
- CUDA kernel execution times on NVIDIA P100 GPUs



one training step

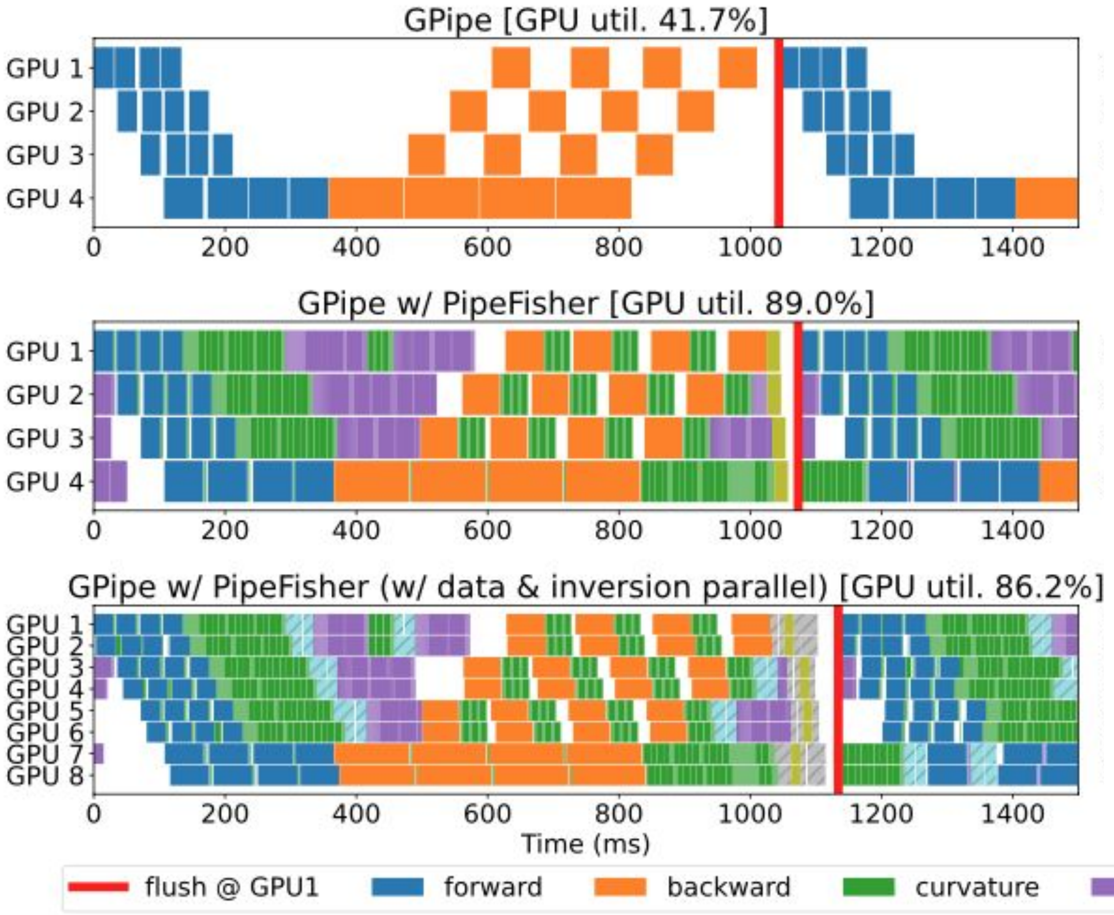
PipeFisher

- ❑ Step 1: **Measure** the times for the forward/backward works and <u>bubbles</u> in a training step.
- ❑ Step 2: **Measure** the times for the curvature/inverse works in a training step.



Profiled results (1/2) [GPipe (Huang et al, 2018)]

- BERT-Base (12 Transformer layers) w/ 4 pipeline stages (3 layers per stage) and 4 micro-batches
- CUDA kernel execution times on NVIDIA P100 GPUs



PipeFisher

- ❑ Step 1: **Measure** the times for the forward/backward works and <u>bubbles</u> in a training step.
- ❑ Step 2: **Measure** the times for the curvature/inverse works in a training step.
- ❑ Step 3: **Assign** the curvature/inverse works to the <u>bubbles</u> within a training step(s) and the precondition work at the end of every step.

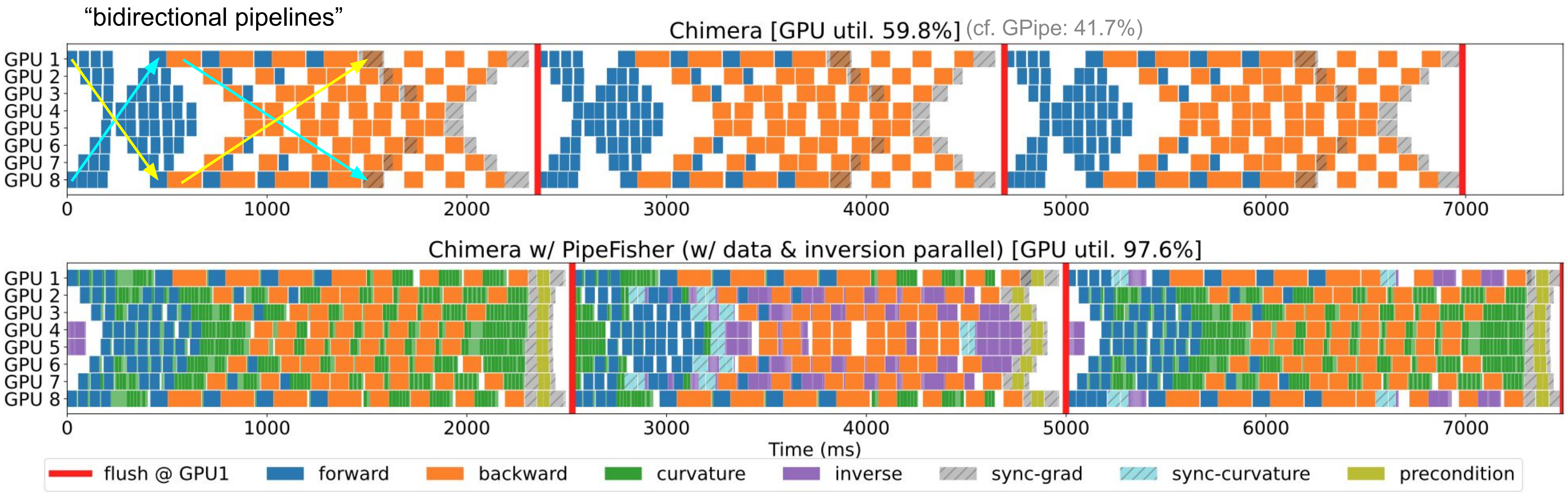
⚠️ the rato = (curvature+inverse) / <u>bubbles</u> determines the frequency of updating the preconditioner (e.g., every 1-2 steps).

$$g_l^{kfac} \approx F_l^{-1} g_l$$

10-100x higher freq. than common practice!

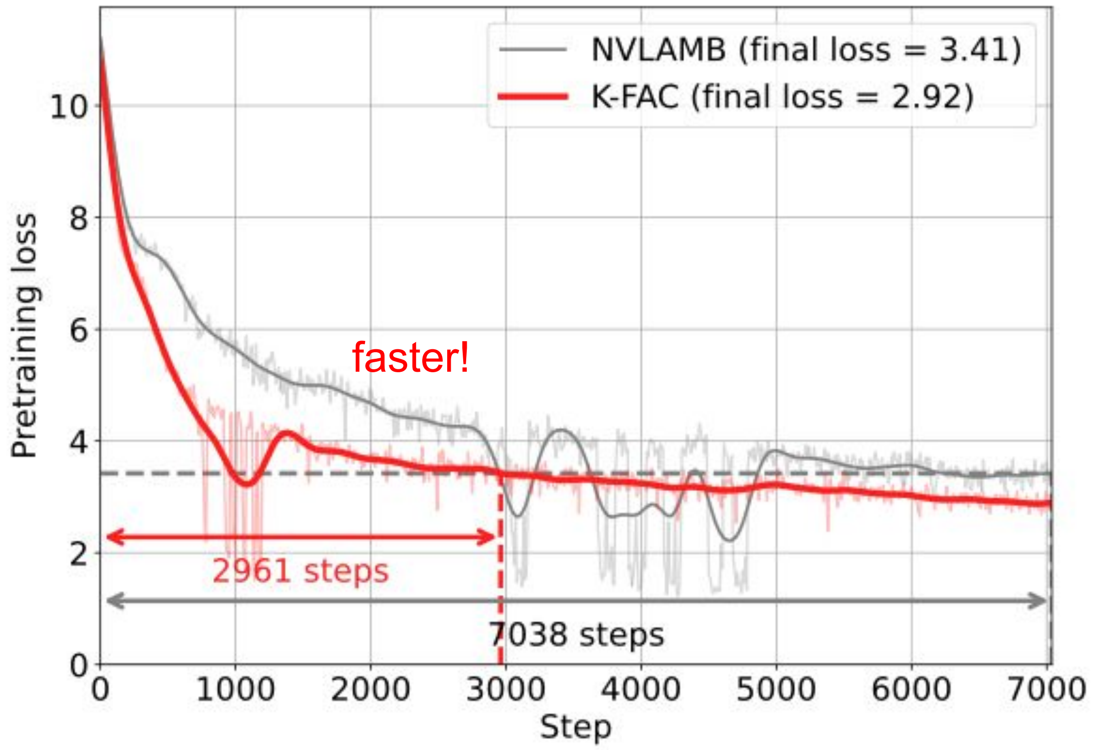
Profiled results (2/2) [Chimera (Li and Hoefler, 2021)]

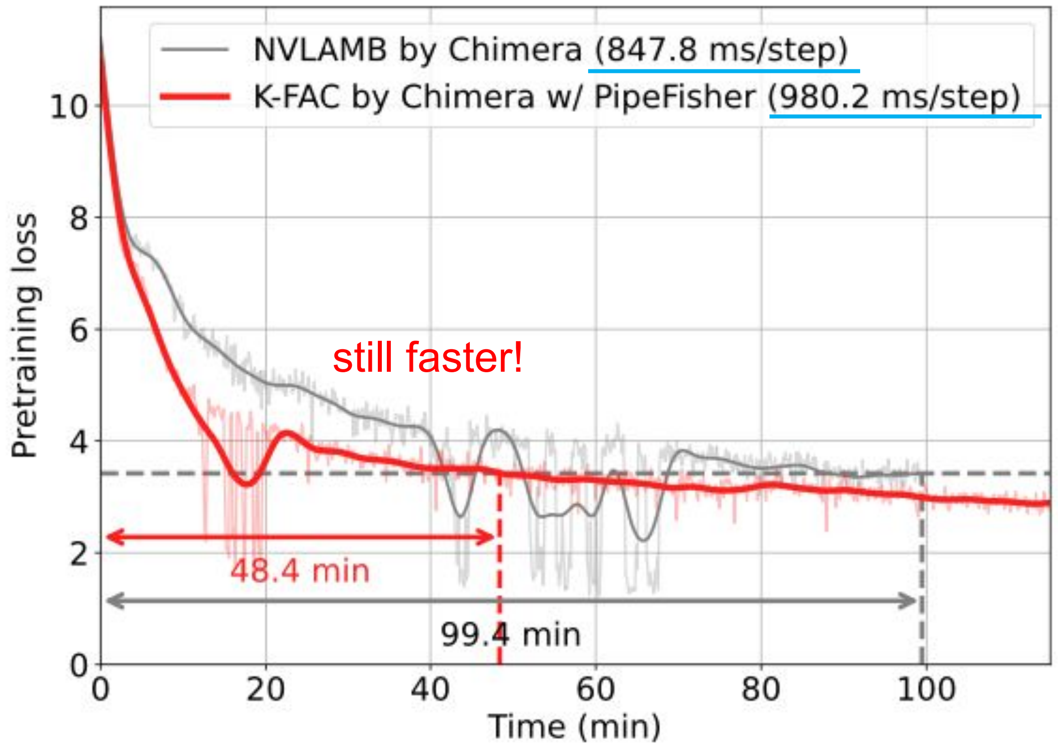
- BERT-Large (24 Transformer layers) w/ 8 pipeline stages (3 layers per stage) and 8 micro-batches
- CUDA kernel execution times on NVIDIA P100 GPUs



BERT-Base Pretraining

- BERT-Base (12 Transformer layers) w/ 4 pipeline stages (3 layers per stage) and 4 micro-batches
- Pretraining on the English Wikipedia
- Time measured on 256 NVIDIA P100 GPUs

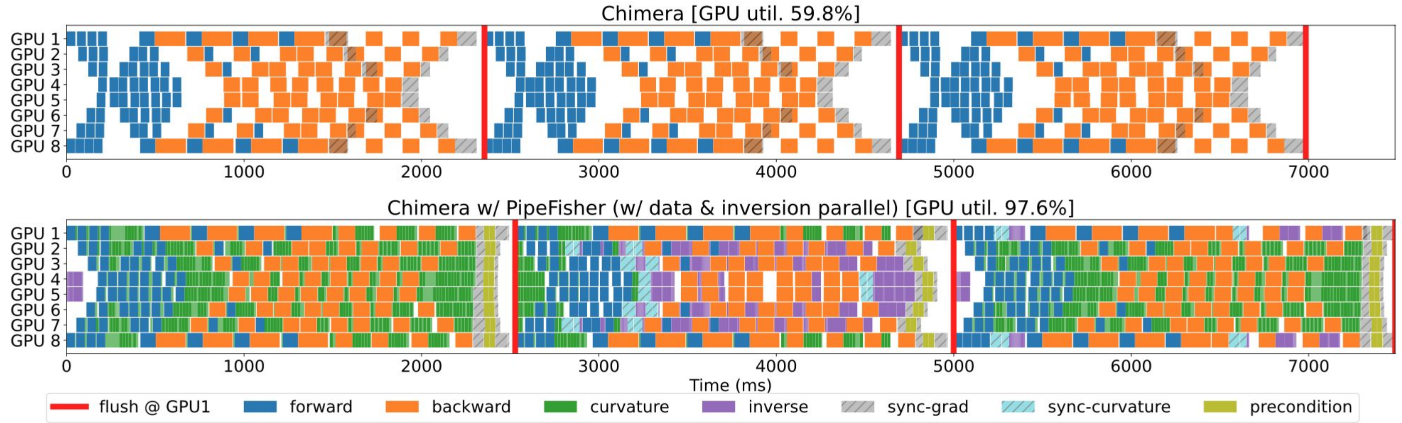




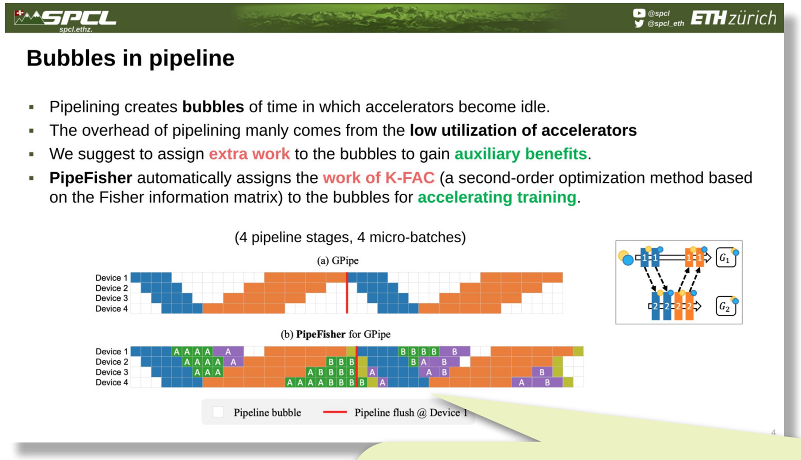
BERT-Large Pretraining

- BERT-Large (24 Transformer layers) w/ 8 pipeline stages (3 layers per stage) and 8 micro-batches
- Pretraining on the English Wikipedia
- Time measured on 8 NVIDIA P100 GPUs (total training time is simulated)

Optimizer	Pipeline scheme	Phase 1			Phase 2	F1
		Steps	Time/step*	Time*	Steps	
NVLAMB	Chimera	7038	2345.6 ms	275.1 min	1563	90.1%
K-FAC	Chimera **w/ PipeFisher**	5000	2499.5 ms	208.3 min	1563	90.15%

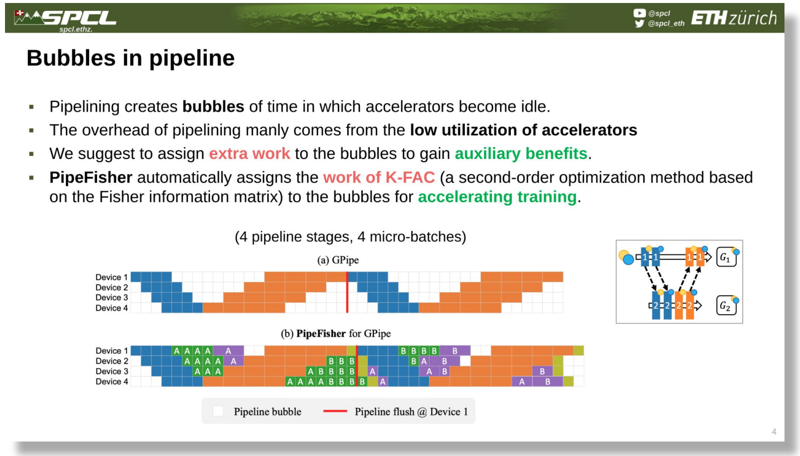


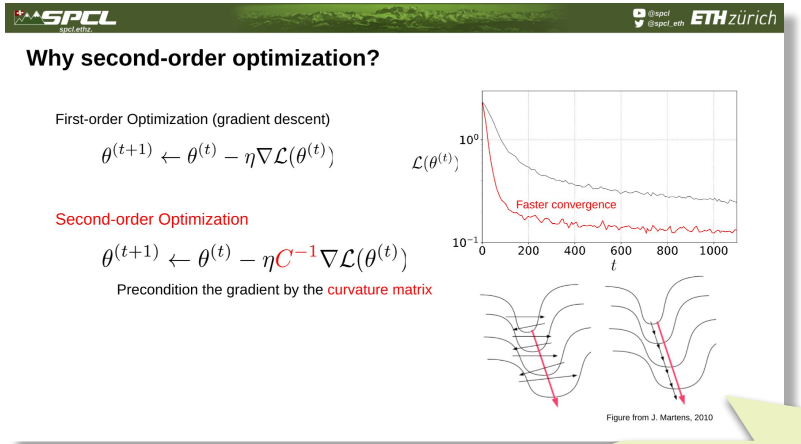
Conclusions



We can fill pipeline bubbles with **extra work**!

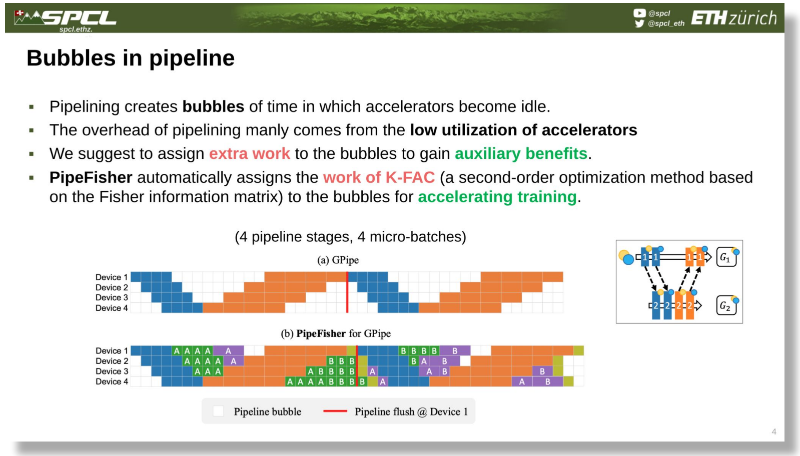
Conclusions

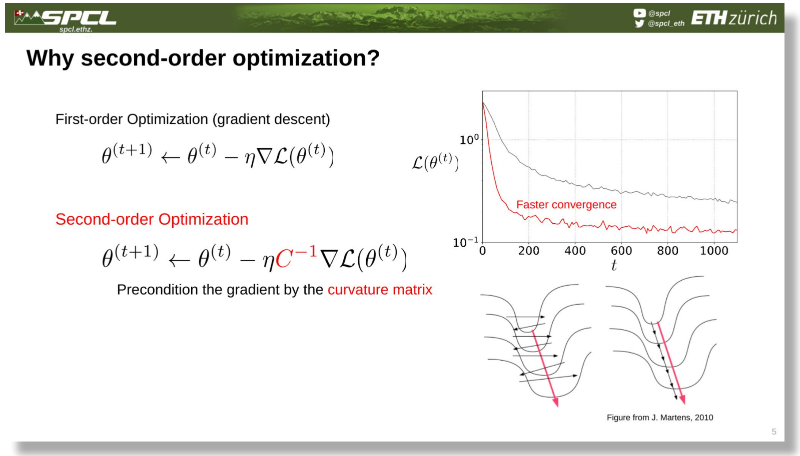


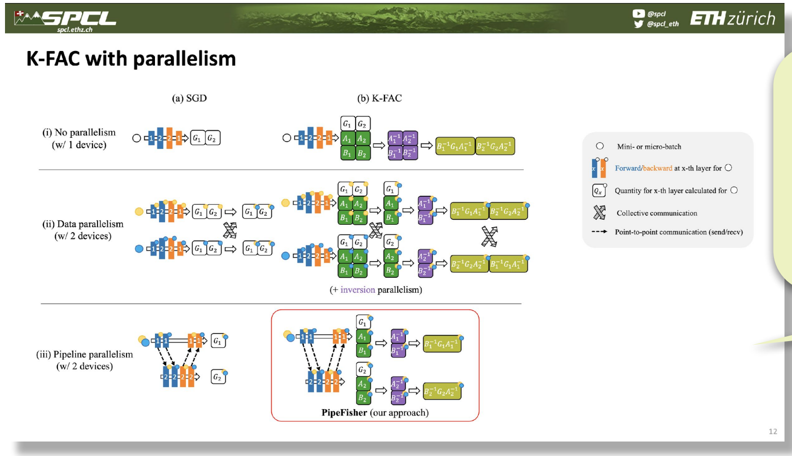


Faster convergence (loss vs # steps) !

Conclusions

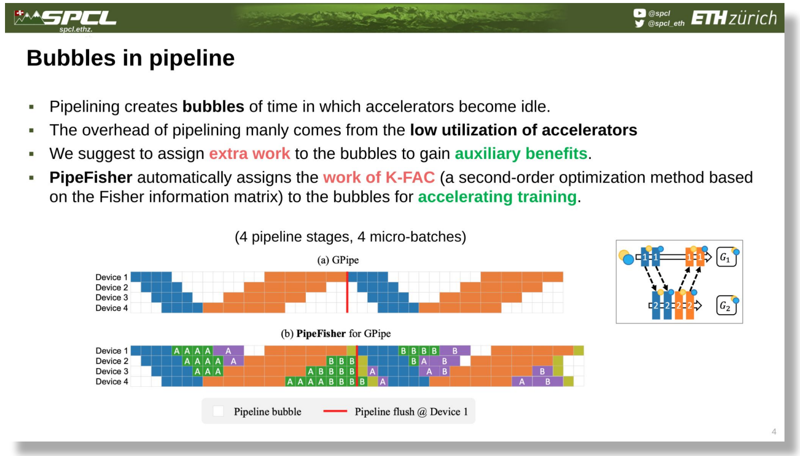


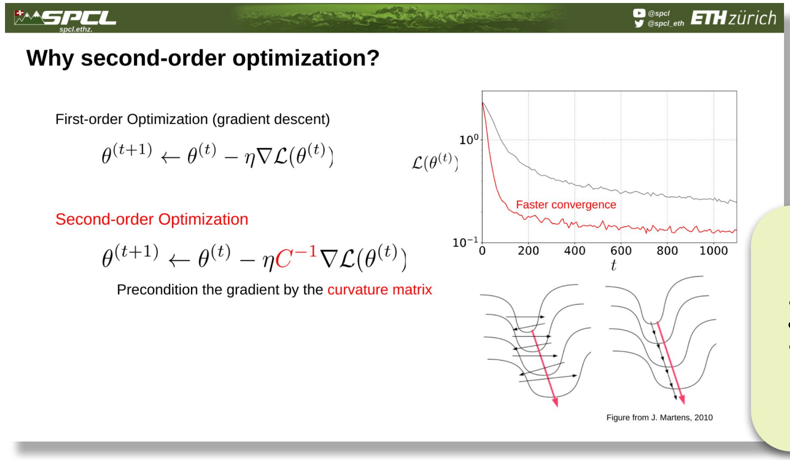


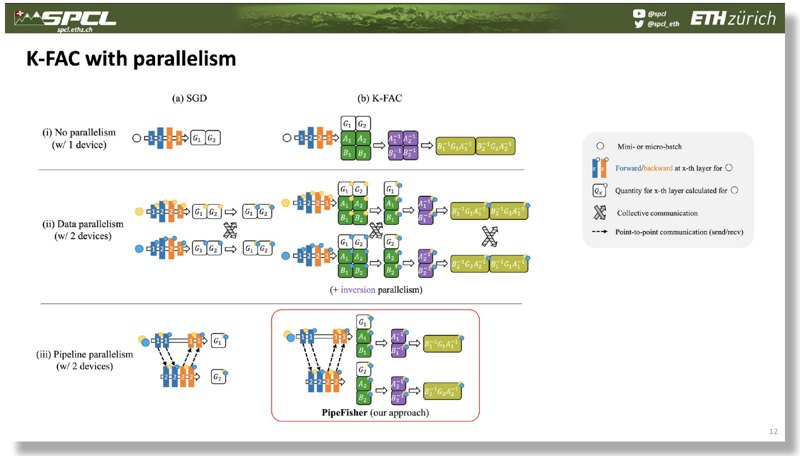


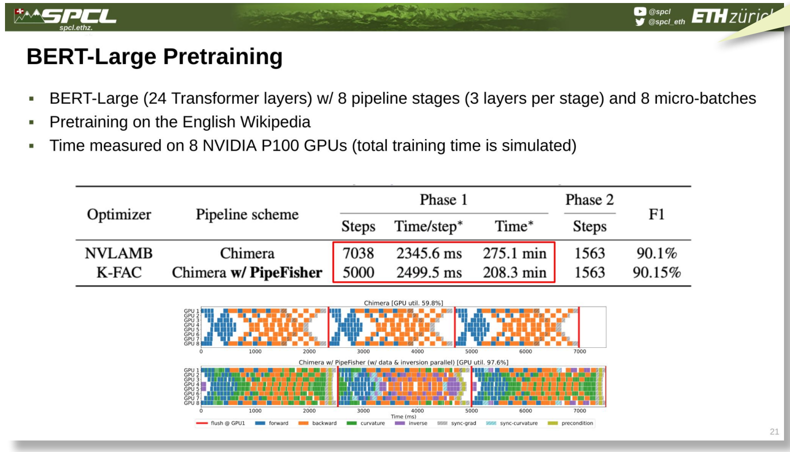
- **PipeFisher** -> K-FAC in pipeline bubbles
- K-FAC (layer-wise preconditioning) is compatible with pipelining!

Conclusions









- Higher GPU utilization!
- Faster training (F1 vs time) (reduction to 50-70%)!

Conclusions

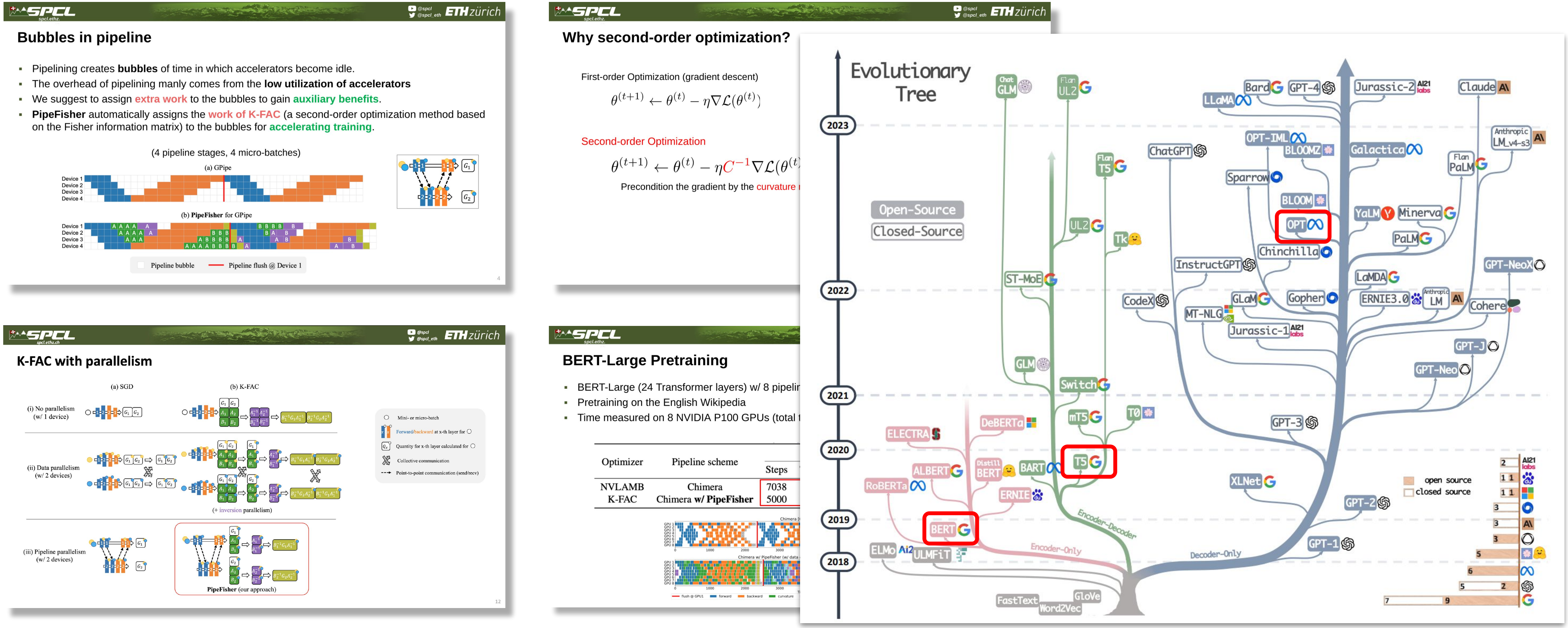
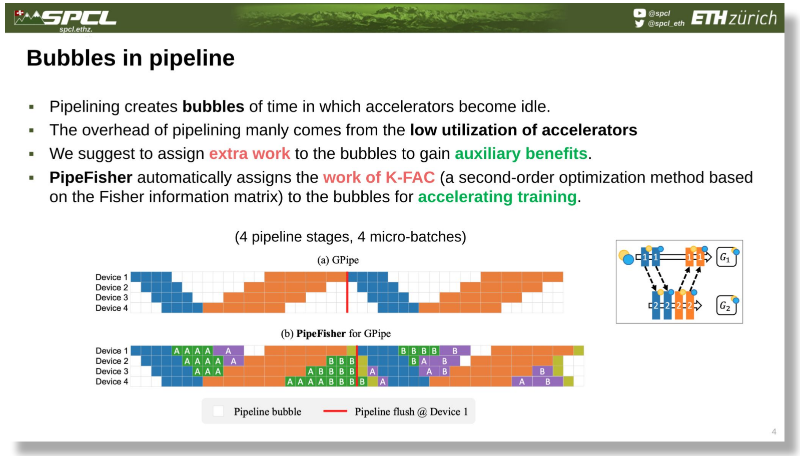
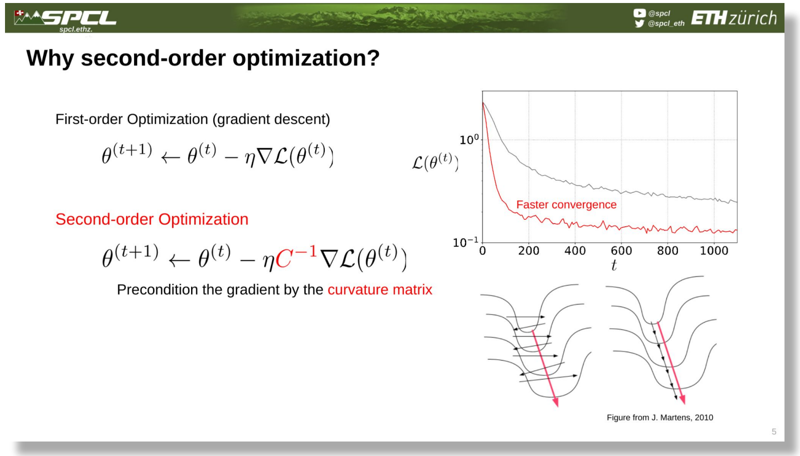
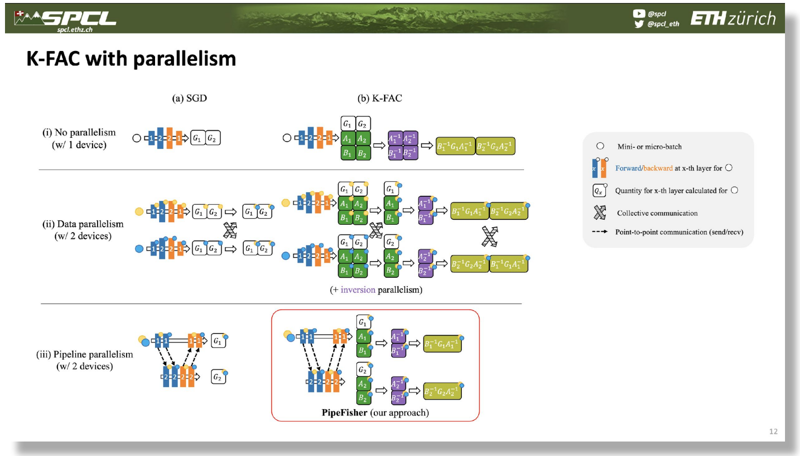


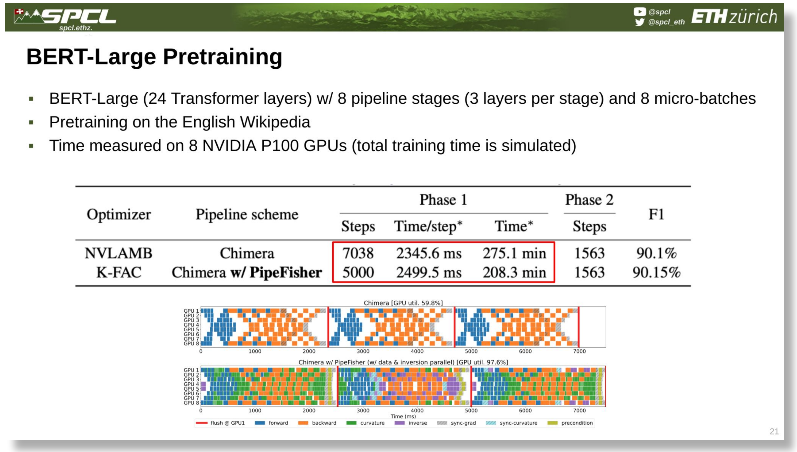
Image: Yang et al., “Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond”, 2023.

Conclusions









More of SPCL's research:

youtube.com/@spcl	180+ Talks
twitter.com/spcl_eth	1.3K+ Followers
github.com/spcl	2K+ Stars

… or spcl.ethz.ch

Paper link:

https://arxiv.org/abs/2211.14133