BETTER TOGETHER WITH intel

aws DEV DAY

LOCATION | September 28, 2021

D - 3

あなたのGPUちゃんと回っていますか？：クラウドでの機械学習のボトルネック特定と最適化

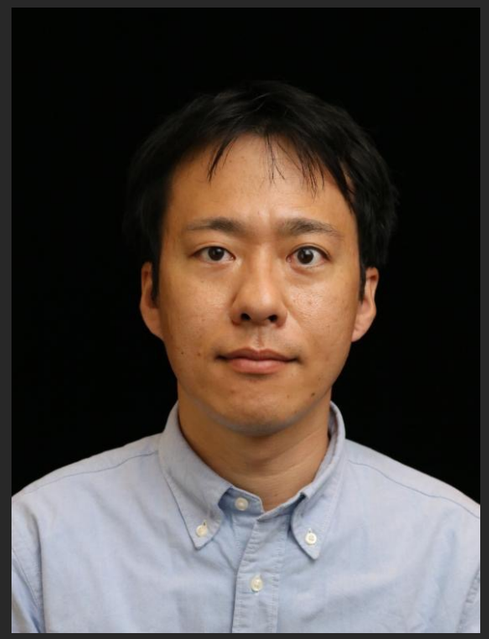
卜部 達也
アマゾン ウェブ サービス ジャパン株式会社

自己紹介

卜部 達也, Ph.D.

- 機械学習ソリューションアーキテクト
  - 機械学習/AI サービスを担当
  - 前職は電機メーカで自動運転開発
  - 前々職は化学の研究者

  - 好きなサービス: Amazon SageMaker



アジェンダ

1. はじめに
2. クラウドで機械学習を行うメリット
3. GPUを用いた学習のステップのおさらい
4. ボトルネック事例と改善方法
5. リソース最適化のベストプラクティス

はじめに：対象とするお客様

AWSでGPUを用いたMLの学習をしていて
「もっと学習を高速にしたい」と思ったことがある方

（特に大規模学習データセット）

今回は画像データを題材にお話します

学習が進まない…

GPUインスタンスを
変えるべき!?

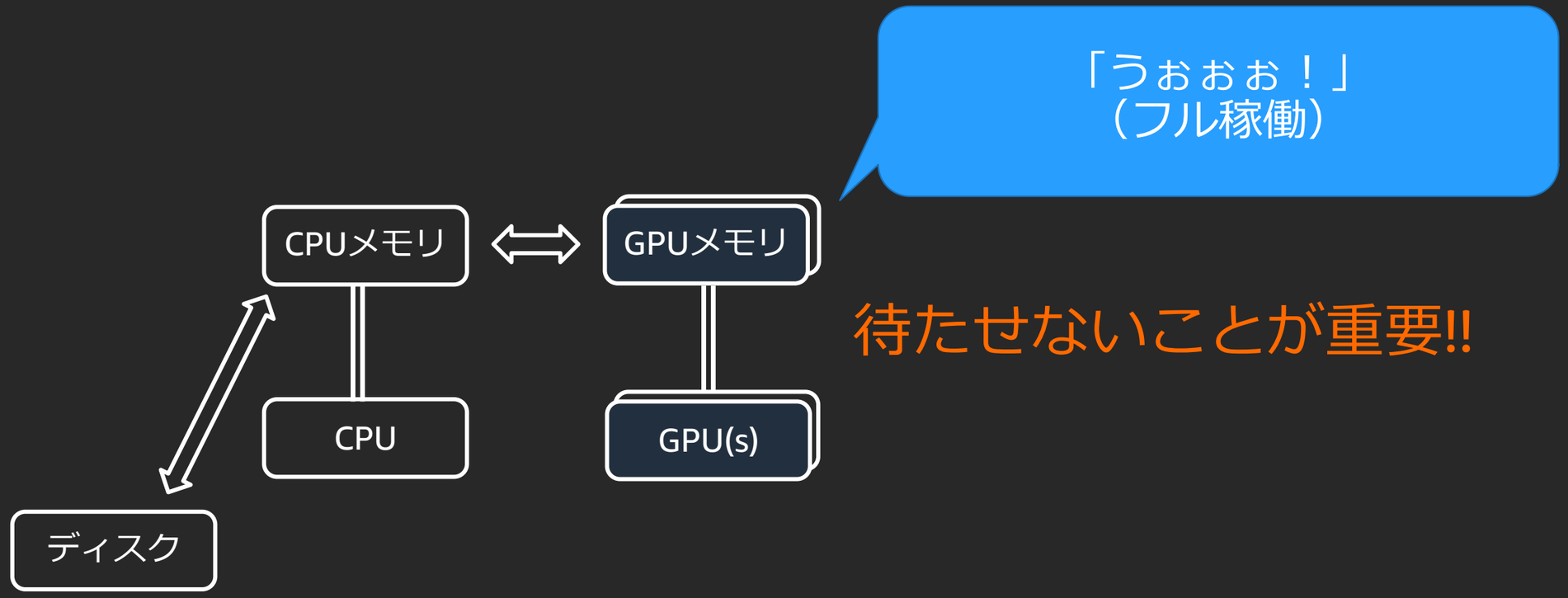
GPUを十分に活用しているか、
キチンと確認しましょう

もっとやれる！

基本事項

GPU：毎秒数兆回（TFLOPS）の浮動小数点演算をおこなう

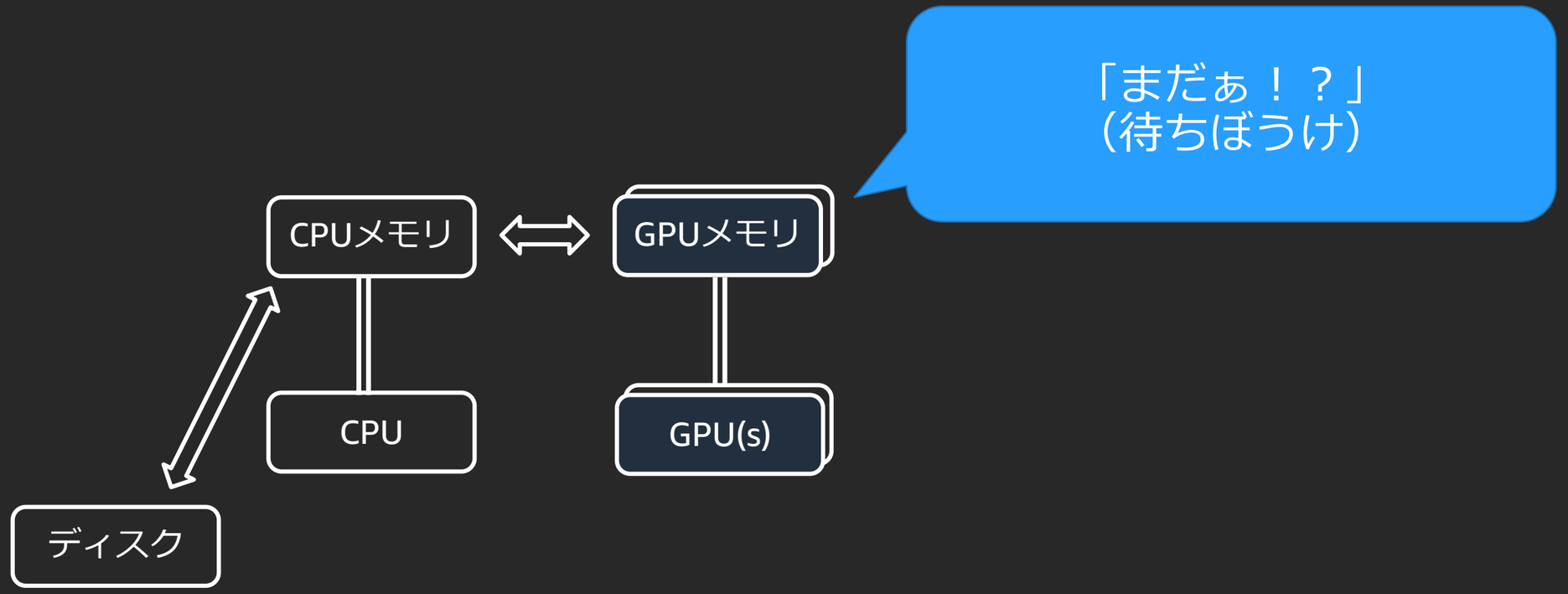
GPUで演算を効率的に実行するためにはデータがGPU上のメモリに乗っている必要がある



CPUメモリ: RAM　　GPUメモリ:VRAM

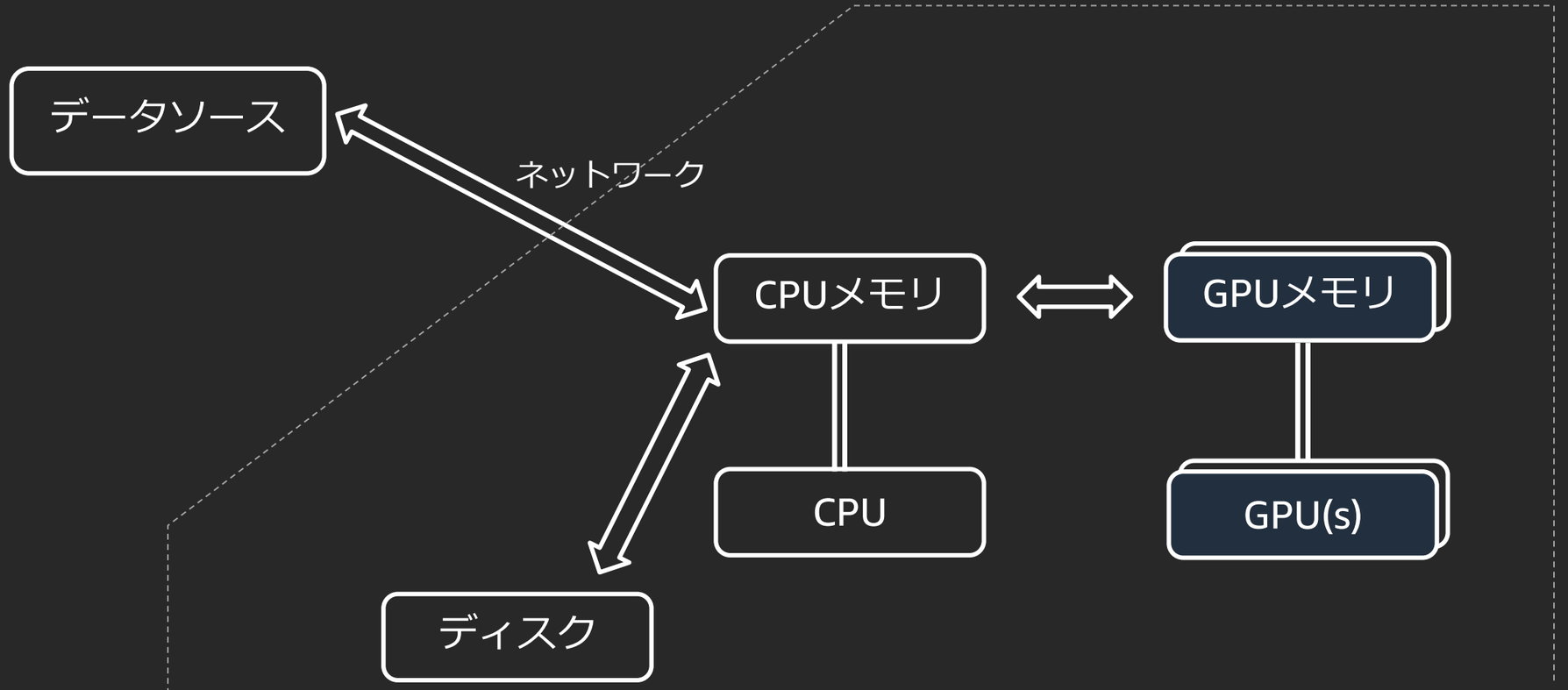
しかし...

現実にはGPUメモリにデータが渡る前に律速段階（ボトルネック）が
存在して性能を発生できないことがある



特に...

クラウド上の場合はデータのネットワーク転送がからみ、問題が発生しやすい



GPUをフル活用するために考慮すべき点

- 最新のライブラリとGPUドライバを使用する
- コードの最適化
- マルチGPUもしくは分散学習の際のGPU間通信最適化
- I/Oとネットワークの最適化

今回はクラウドで問題になることが多い
ファイルI/Oまわりを中心にお話しします

AWSクラウド上で機械学習をおこなうメリット

AWSクラウド上でMLをおこなうメリット

- インスタンス（CPU, GPU, RAM, etc...）の選択肢が豊富
- （実質）無制限のストレージ容量
- 従量課金制による導入・運用コストの削減
- 環境構築の手間を削減
- マネージドなMLプラットフォーム（Amazon SageMaker）
  - 実験管理のしやすさ、MLプロセスを加速する機能
- （部屋がPCの排熱で熱くならない）

AWS の機械学習スタック

AI サービス: 機械学習の深い知識なしに利用可能

HEALTH AI

NEW
Amazon HealthLake

Amazon Transcribe Medical

Amazon Comprehend Medical

INDUSTRIAL AI

NEW
AWS Panorama + Appliance

NEW
Amazon Monitron

NEW
Amazon Lookout for Equipment

NEW
Amazon Lookout for Vision

ANOMALY DETECTION

NEW
Amazon Lookout for Metrics

CODE AND DEVOPS

NEW
Amazon DevOps Guru

Amazon CodeGuru

VISION

Amazon Rekognition

SPEECH

Amazon Polly

Amazon Transcribe
+Medical

TEXT

Amazon Comprehend
+Medical

Amazon Translate

Amazon Textract

SEARCH

Amazon Kendra

CHATBOTS

Amazon Lex

PERSONALIZATION

Amazon Personalize

FORECASTING

Amazon Forecast

FRAUD

Amazon Fraud Detector

CONTACT CENTERS

Contact Lens

Voice ID
For Amazon Connect

ML サービス: 機械学習のプロセス全体を効率化するマネージドサービス

Amazon SageMaker

SAGEMAKER STUDIO IDE

Label data

NEW **Aggregate & prepare data**

NEW **Store & share features**

Auto ML

Spark/R

NEW **Detect bias**

Visualize in notebooks

Pick algorithm

Train models

Tune parameters

NEW **Debug & profile**

Deploy in production

Manage & monitor

NEW **CI/CD**

Human review

NEW: SageMaker JumpStart

NEW: Model management for edge devices

ML フレームワークとインフラストラクチャ: 機械学習の環境を自在に構築して利用

TensorFlow

mxnet

PyTorch

GLUON

Keras

scikit learn

HOROVOD

DeepGraphLibrary

Deep Learning AMIs & Containers

GPUs & CPUs

Elastic Inference

Trainium

Inferentia

FPGA

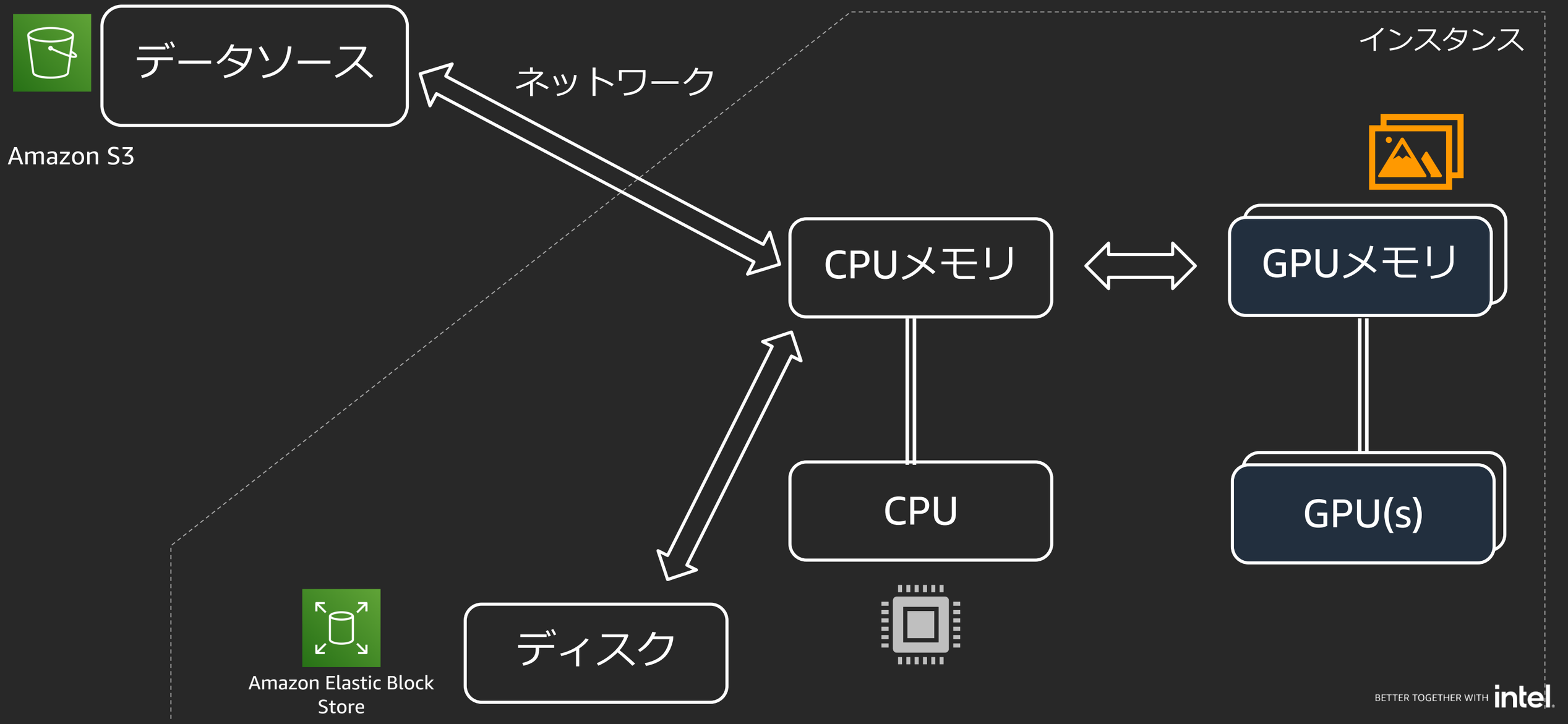
BETTER TOGETHER WITH intel

Amazon SageMaker による学習

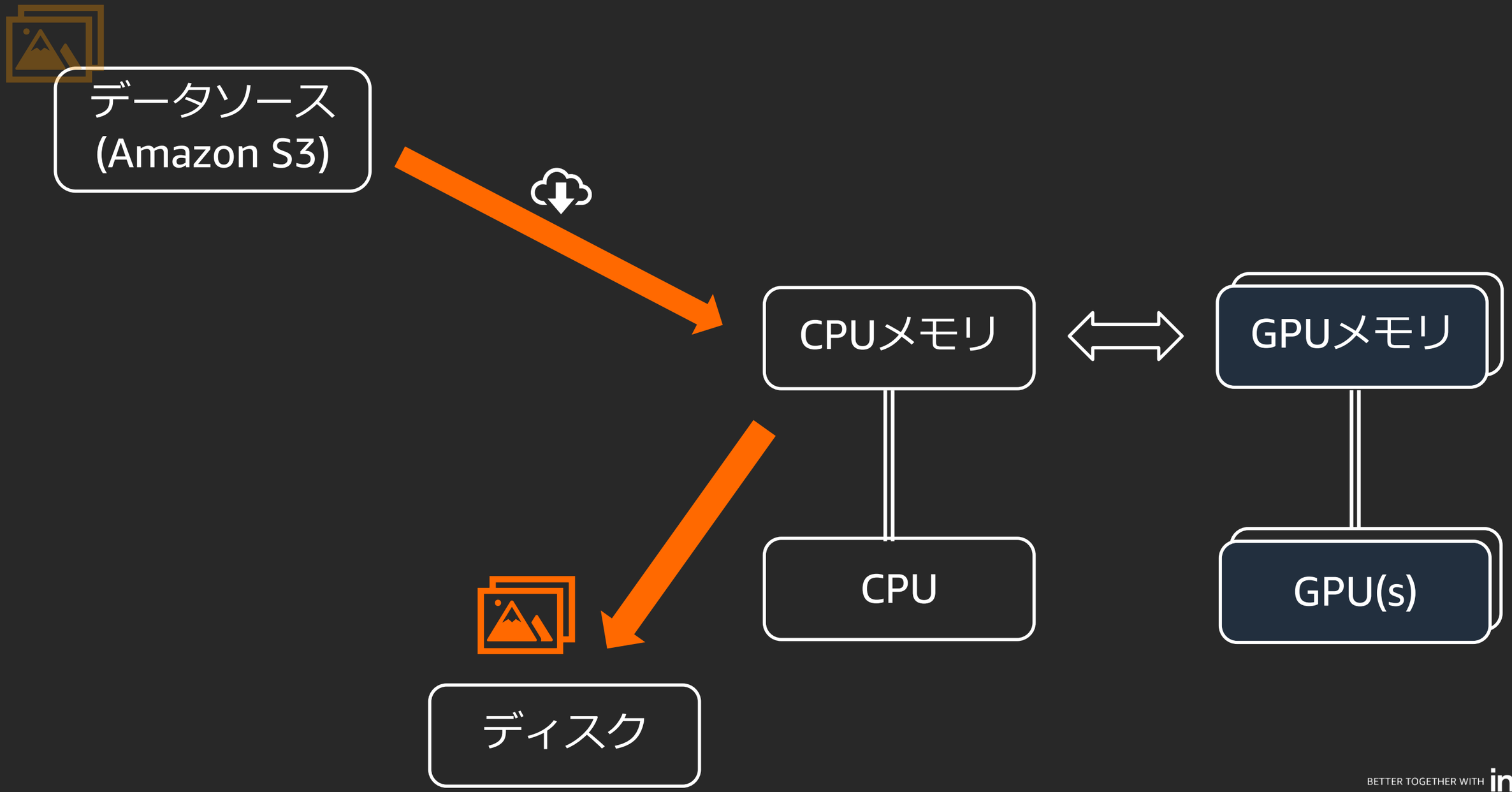
学習に必要なデータ、コード、コンテナイメージを用意すれば、
ユーザはインスタンスを自由に選んで学習を実行できる

GPUを用いた学習のステップ

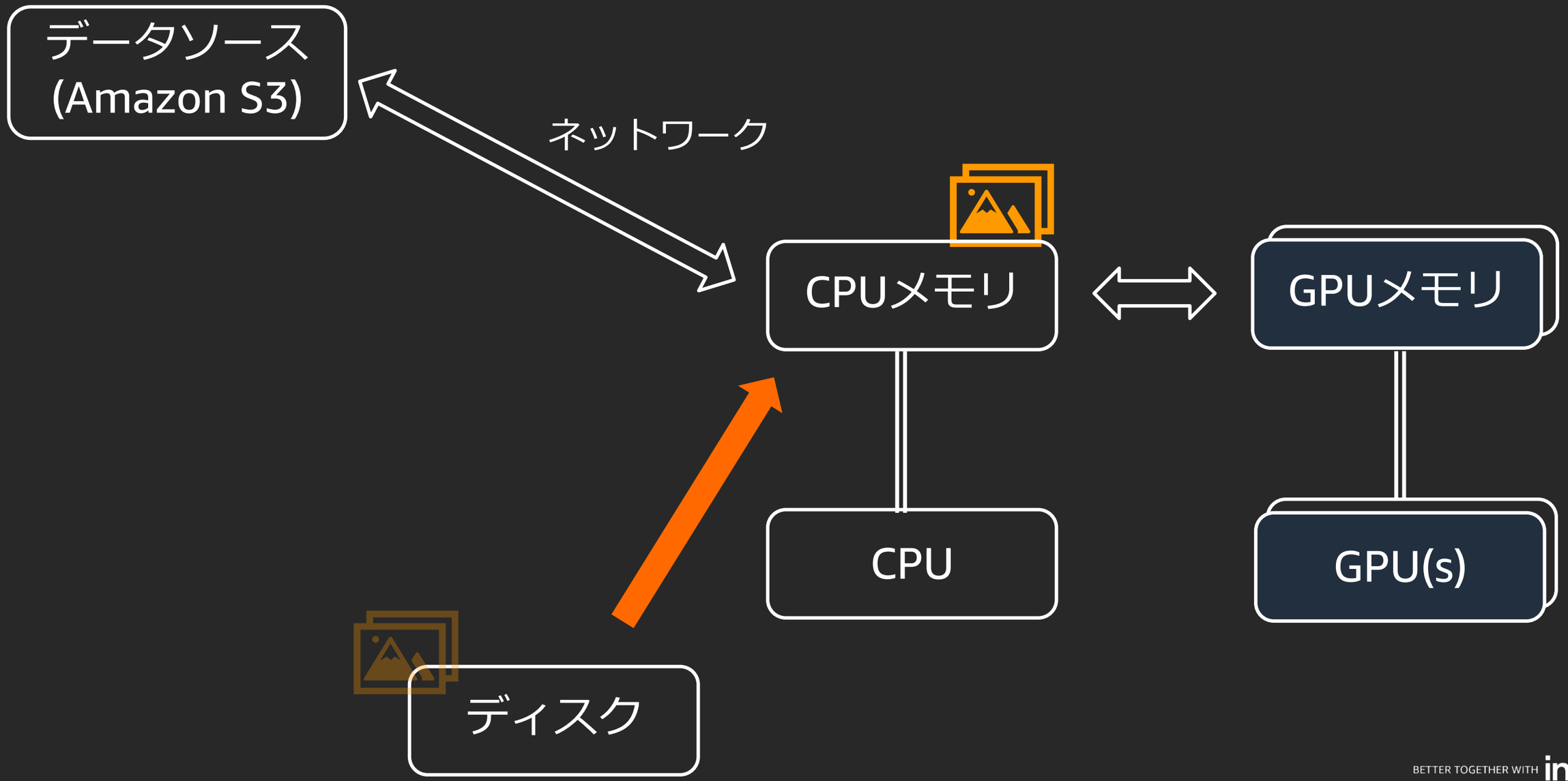
GPUで学習する際の一般的なステップ（全景）



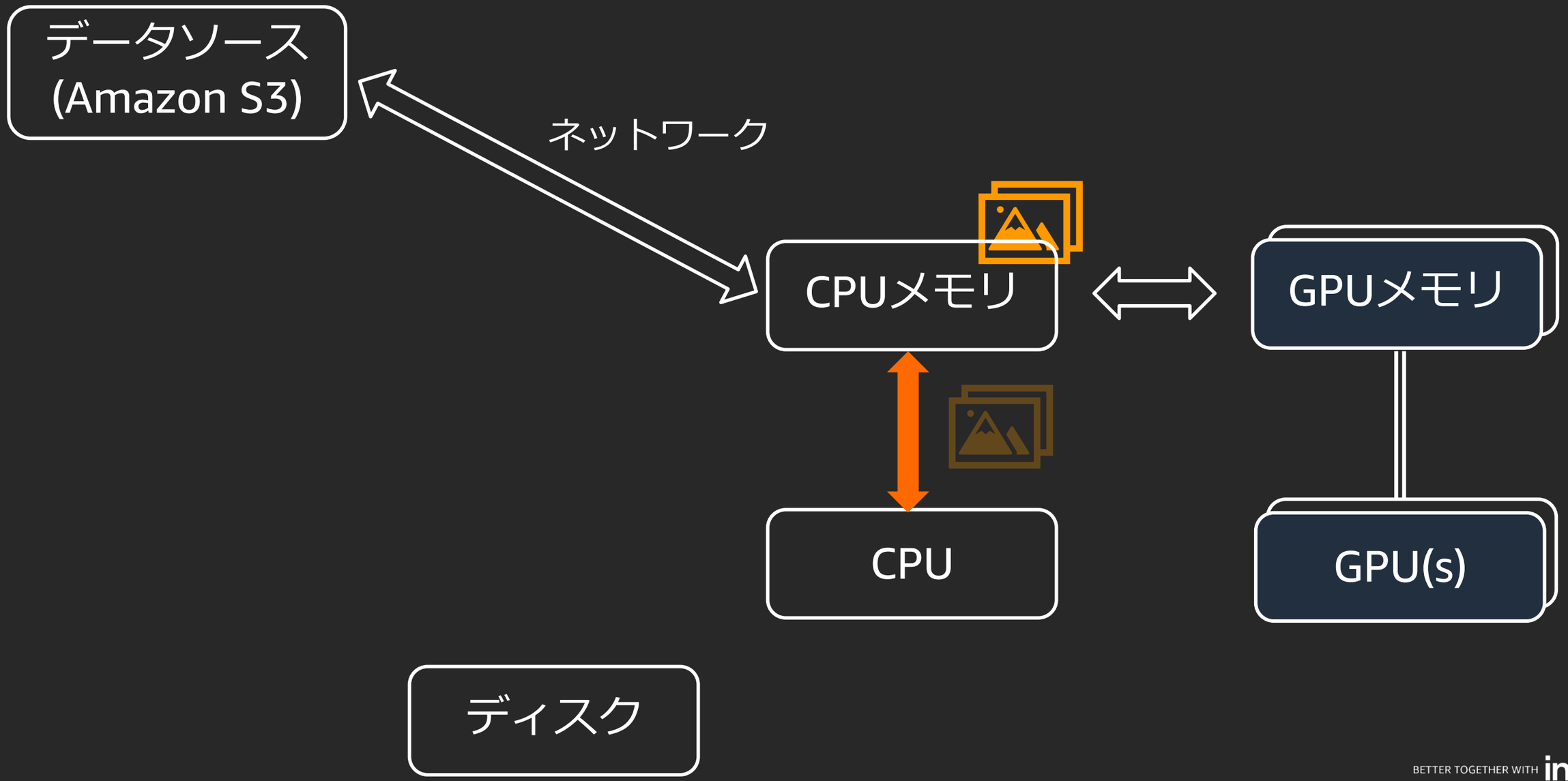
Step1. 前準備:データソースからデータセットをダウンロード



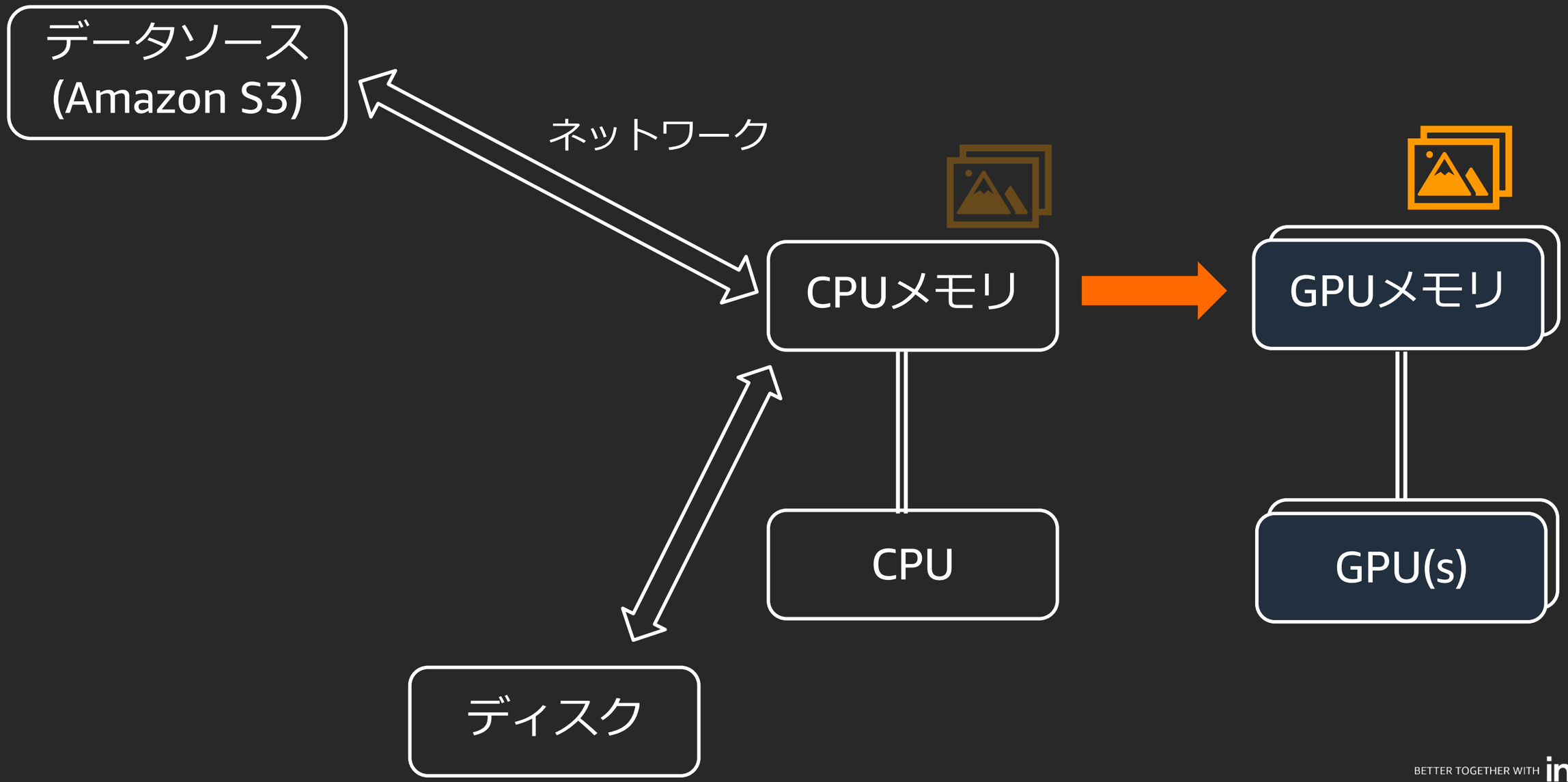
Step2. ローカルディスクからCPUメモリにデータを読み込む



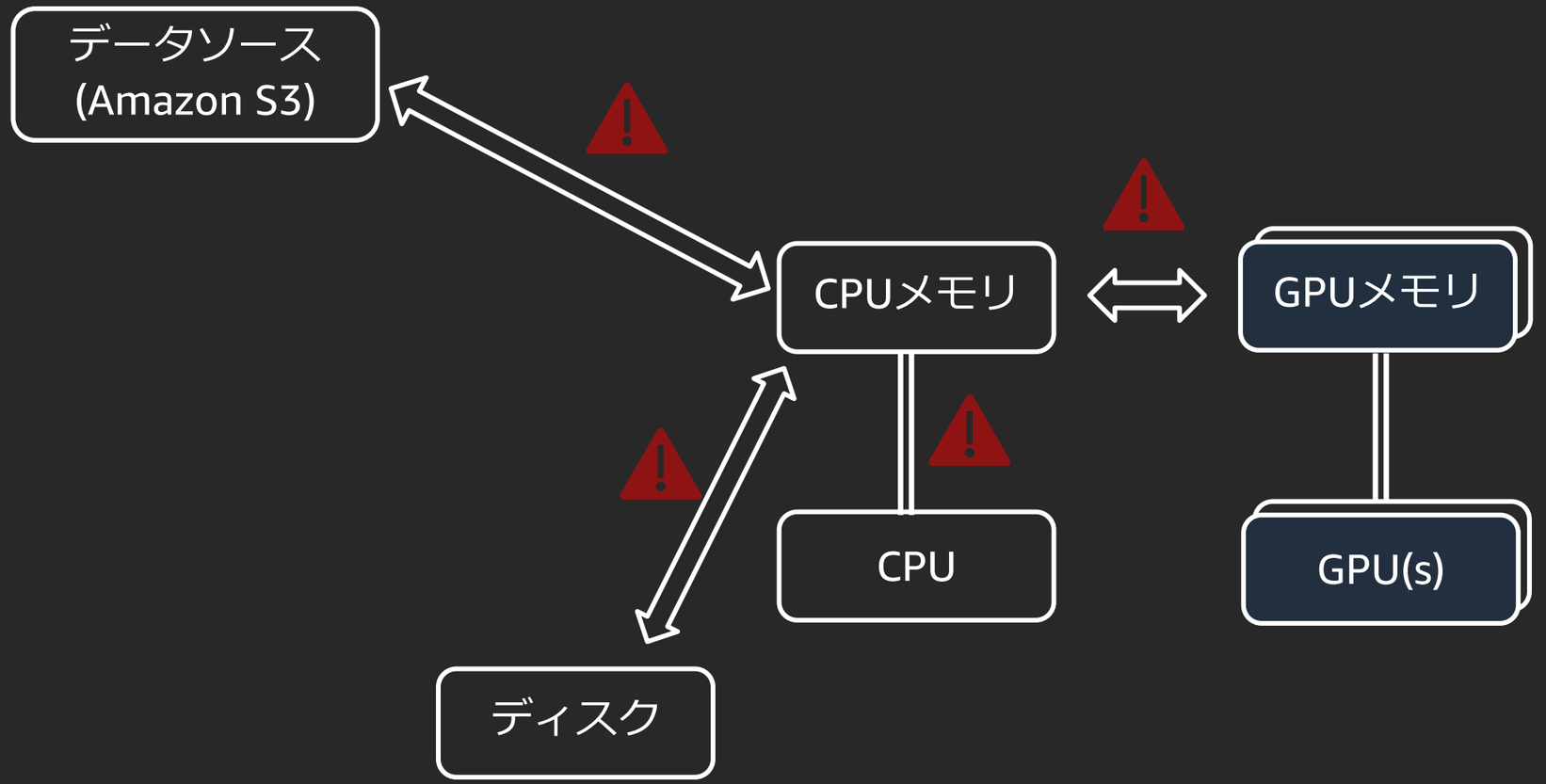
Step3. データ前処理（型変換やリサイズなど）



Step4. GPUメモリへデータを転送



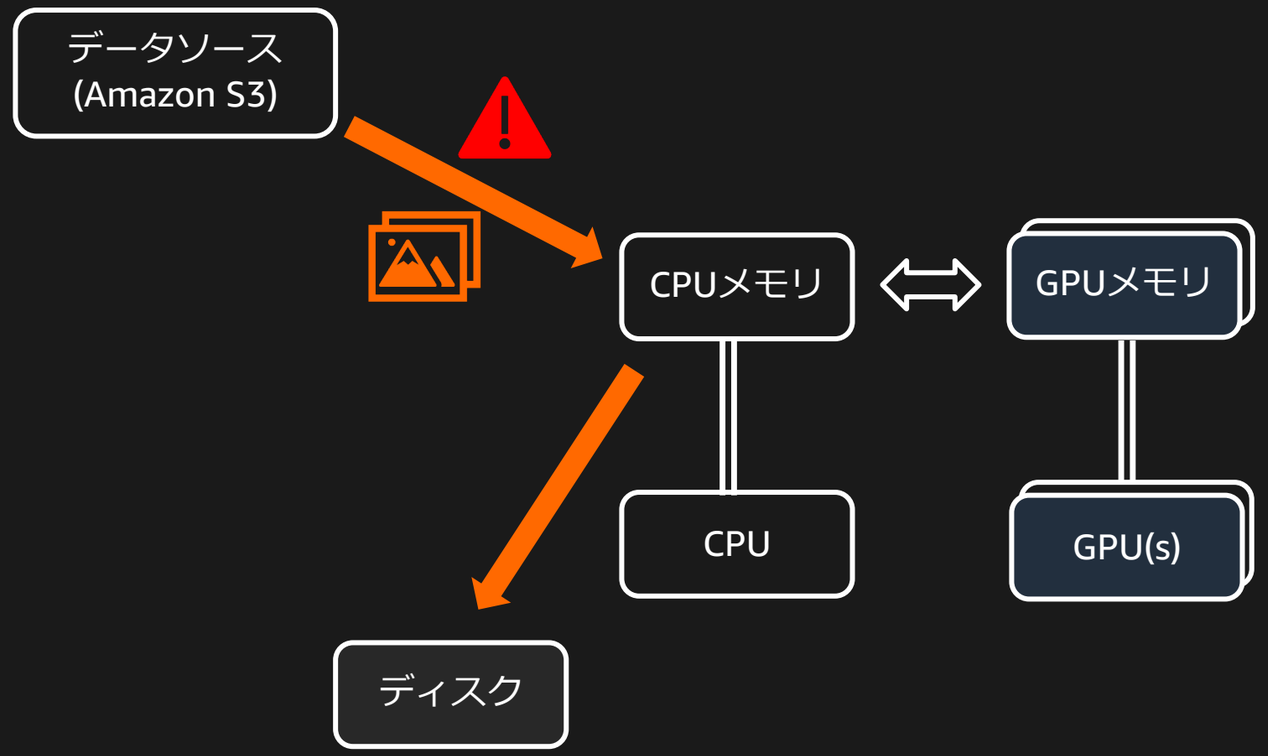
どこかに1つでもボトルネックがあるとGPUをフル活用できない



ボトルネック箇所を発見・特定し、対策することが重要

ボトルネック事例

事例1: S3からのダウンロード



事例1: S3からのダウンロード

発見方法：学習開始まで時間がかかる。CPU使用率低, GPU使用率低

解決方法：

1. 転送スピードを上げる
2. 逐次的にデータをダウンロードしつつ学習する

解決方法1: 転送スピードを上げる

ファイルサイズの最適化

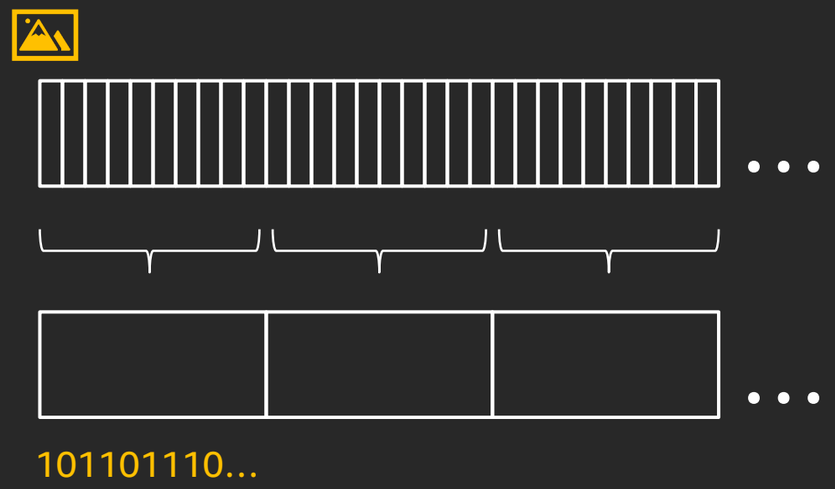
転送ブロックサイズ以下だと
オーバヘッドが発生する

並列処理の恩恵を受けにくい

一般的に、最適なファイルサイズは1-128 MB

フレームワークによっては大規模データに適したフォーマットを提供している

複数ファイルを圧縮・バイナリ化



- RecordIO (MXNet, etc.)
- TFRecord (TensorFlow)
- WebDataset (PyTorch)
- MLIO (TensorFlow, PyTorch)

解決方法1: 転送スピードを上げる (その2)

分散ファイルシステムを使って読み込みを高速化する

Amazon FSx for Lustre

低レイテンシー・高スループットの共有ストレージ

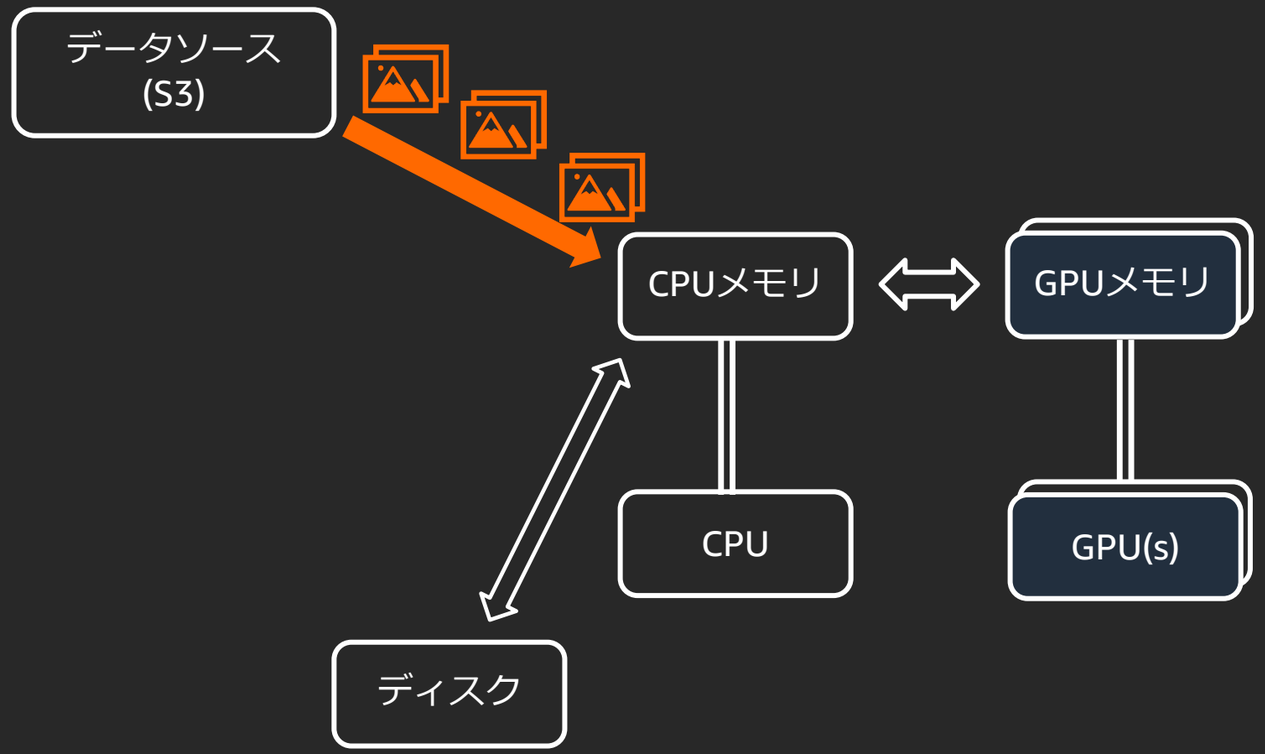
Amazon Elastic File System (EFS)

シンプルでスケーラブルなNFS ファイルシステム

注：ファイルサイズが小さいと効果が発揮できない（1.1の問題）

解決方法2：データを逐次的にダウンロードする

学習を素早く開始したい場合、ディスクに全データセットが乗り切らない場合に有効



解決方法3：データを逐次的にダウンロード・使用する

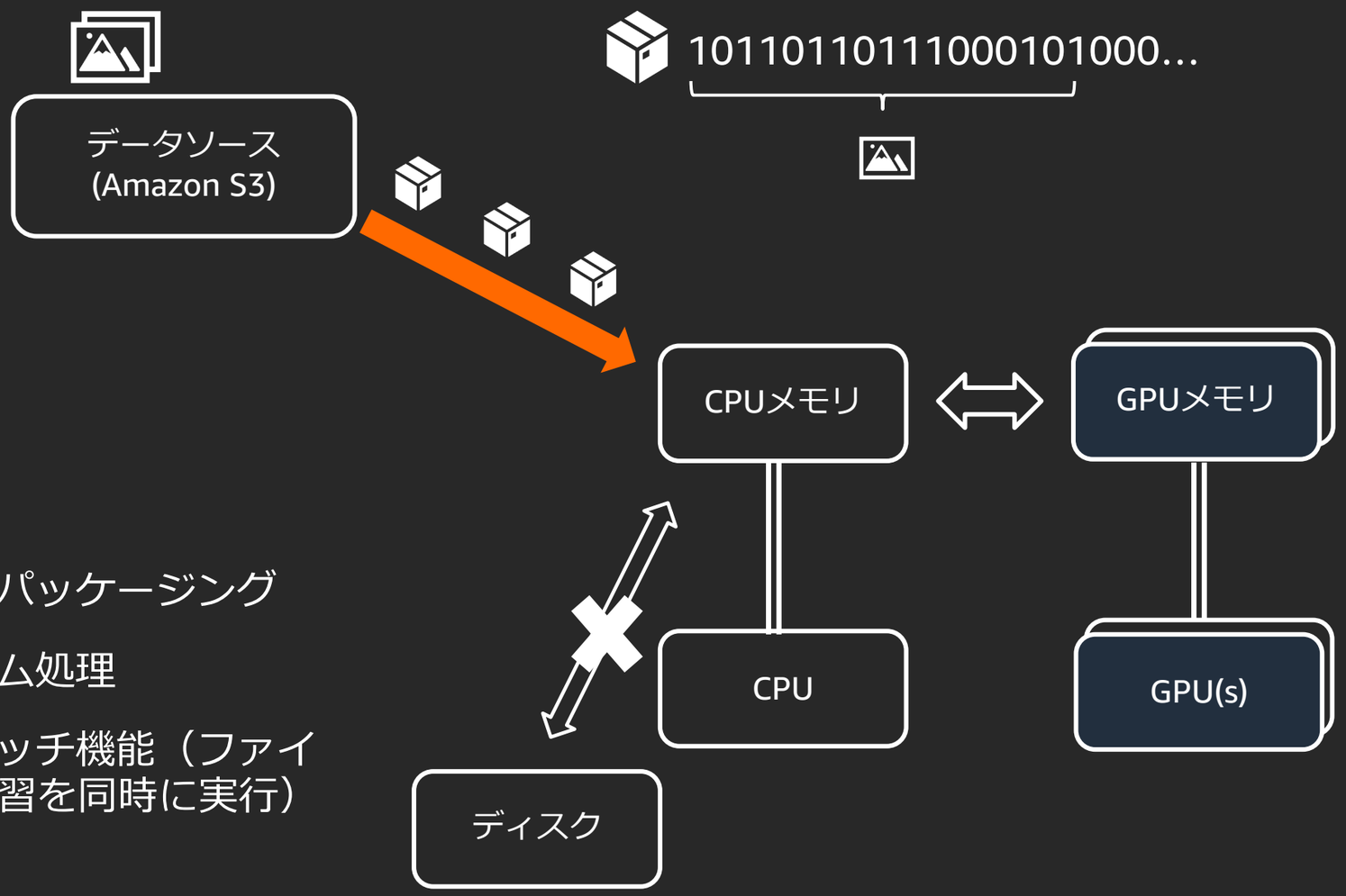
実装イメージ (PyTorch)

```
class S3Dataset(torch.utils.data.Dataset):
    def __init__(self, input_images_path_list, client, bucket):
        self.input_images_path_list = input_images_path_list
        self.client = client
        self.bucket = bucket

    def __len__(self):
        return len(self.input_images_path_list)

    def __getitem__(self, idx):
        # download image from S3
        obj = self.client.get_object(Bucket=self.bucket,
                    Key=self.input_images_path_list[idx])
        image = Image.open(BytesIO(obj['Body'].read()))

        return image
```

Dataset生成時ではなく、ミニバッチを供給時に都度S3からダウンロードする

パイプモード

データを継続的にストリーミングして利用する
（バイト列をデコードする必要あり）



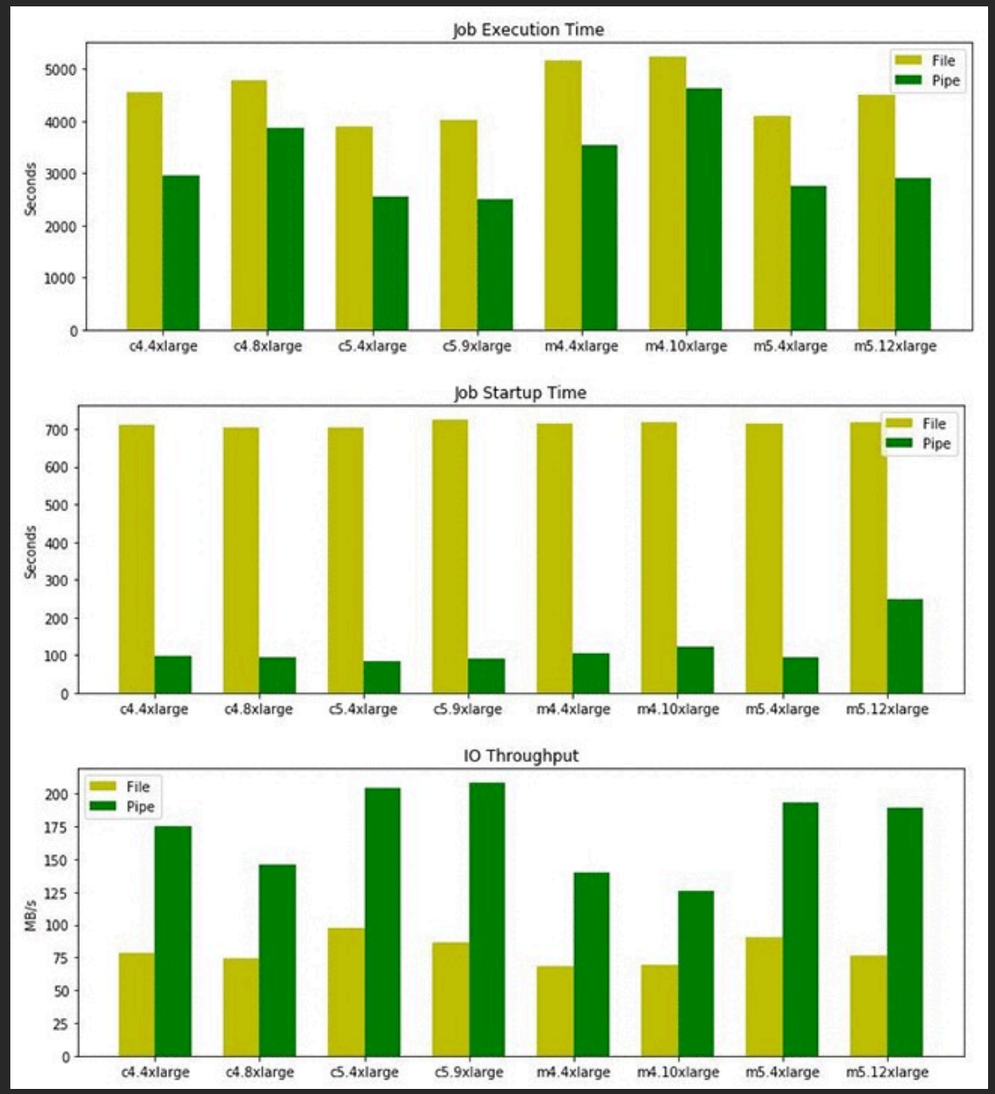
- メリット
  - データをパッケージング
  - ストリーム処理
  - プリフェッチ機能（ファイルIOと学習を同時に実行）

https://aws.amazon.com/jp/blogs/news/using-pipe-input-mode-for-amazon-sagemaker-algorithms/

パイプモードの特徴

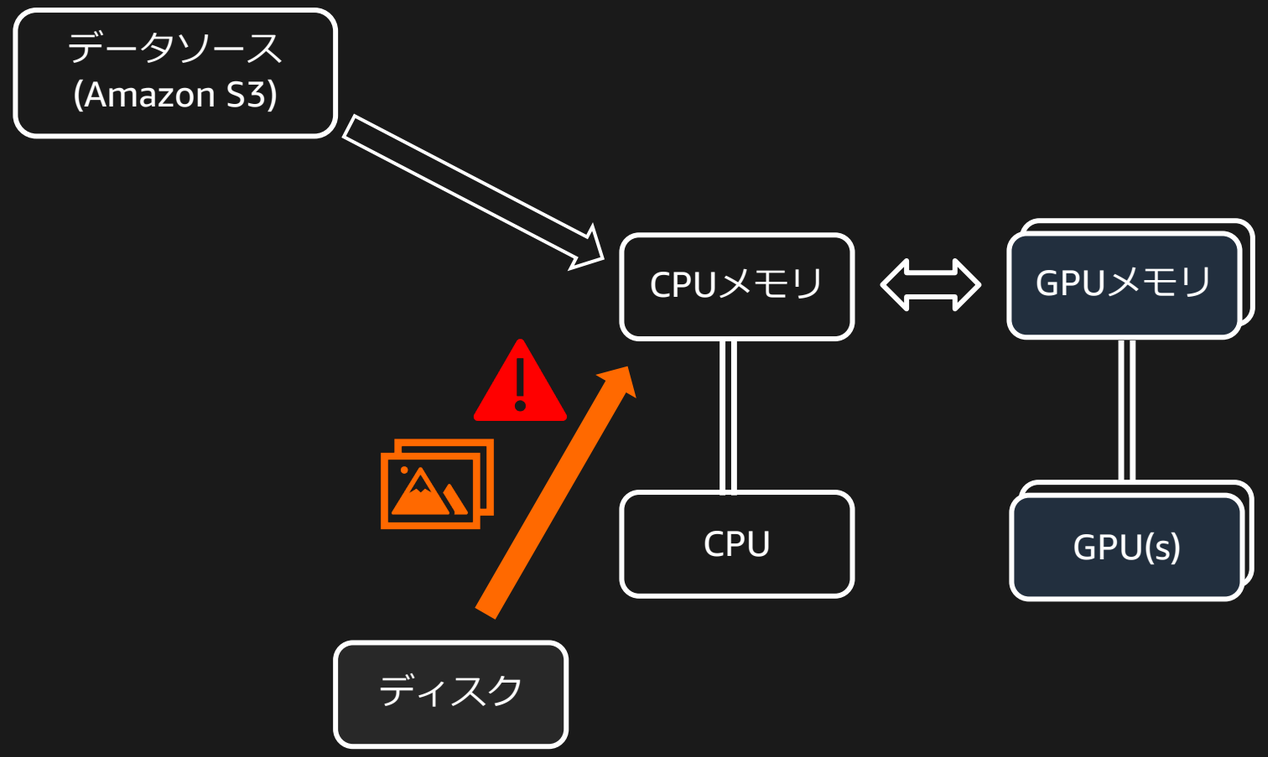
- 学習が始まるまでの待ち時間が短い
- ローカルディスクにデータを置かない（ストレージコスト減）
- エポックの回数分データを重複してダウンロードする（通信コスト増）
- CPUメモリ、ローカルディスクに載らない大規模データを捌ける
- プリフェッチ（IOと学習を並列で実行）

79.1GiBのデータをPCA処理



https://aws.amazon.com/jp/blogs/news/using-pipe-input-mode-for-amazon-sagemaker-algorithms/

事例2: データのロード



事例2: データのロード

発見方法：ディスク使用率高、CPU使用率低、GPU利用率低

解決方法：

1. S3からダウンロードしたオブジェクトをオンメモリで処理する
2. データロードを並列処理する（ダウンロードにも有効）
3. ローカルディスクのロード速度を上げる

解決方法1：オンメモリで処理する

S3からのダウンロードの際、オブジェクトをローカルディスクに一度保存するのではなく、
オンメモリで処理する。

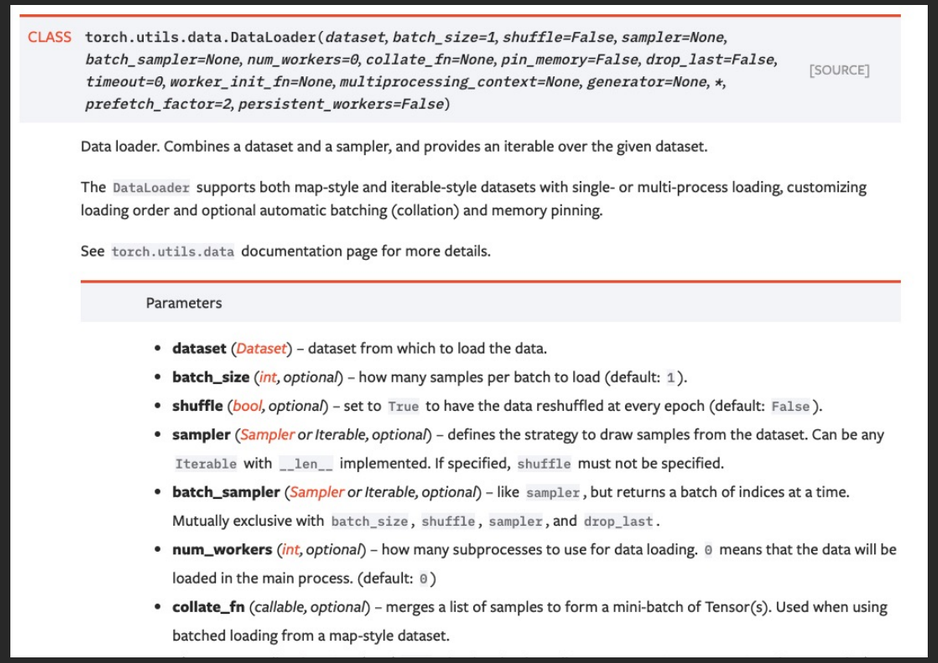
前掲のPyTorchのDataSetコード

```
obj = self.client.get_object(Bucket=self.bucket,
                             Key=self.input_images_path_list[idx])
image = Image.open(BytesIO(obj['Body'].read()))
```

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.get_object

解決方法2：並列処理する

データのロードを並列処理する



CLASS `torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, sampler=None, batch_sampler=None, num_workers=0, collate_fn=None, pin_memory=False, drop_last=False, timeout=0, worker_init_fn=None, multiprocessing_context=None, generator=None, *, prefetch_factor=2, persistent_workers=False)` [SOURCE]

Data loader. Combines a dataset and a sampler, and provides an iterable over the given dataset.

The `DataLoader` supports both map-style and iterable-style datasets with single- or multi-process loading, customizing loading order and optional automatic batching (collation) and memory pinning.

See `torch.utils.data` documentation page for more details.

Parameters

- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int*, *optional*) – how many samples per batch to load (default: `1`).
- **shuffle** (*bool*, *optional*) – set to `True` to have the data reshuffled at every epoch (default: `False`).
- **sampler** (*Sampler or Iterable*, *optional*) – defines the strategy to draw samples from the dataset. Can be any `Iterable` with `__len__` implemented. If specified, `shuffle` must not be specified.
- **batch_sampler** (*Sampler or Iterable*, *optional*) – like `sampler`, but returns a batch of indices at a time. Mutually exclusive with `batch_size`, `shuffle`, `sampler`, and `drop_last`.
- **num_workers** (*int*, *optional*) – how many subprocesses to use for data loading. `0` means that the data will be loaded in the main process. (default: `0`)
- **collate_fn** (*callable*, *optional*) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.

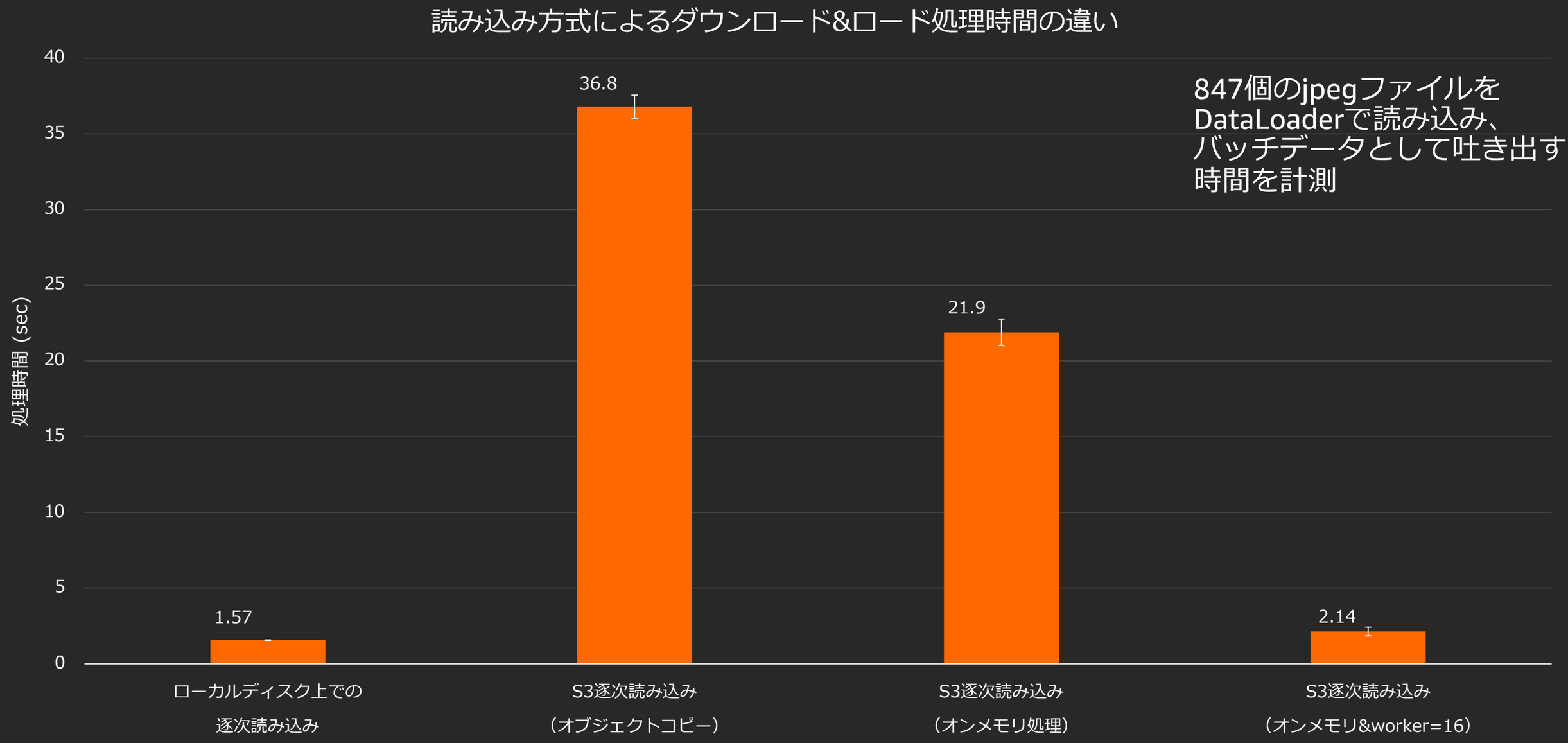
https://pytorch.org/docs/stable/data.html

例 (PyTorch)

```
train_loader = torch.utils.data.DataLoader(s3_dataset, batchsize=16, num_workers=4)
```

DataSetでS3からの逐次ダウンロードを行なっている場合、ダウンロード自体も並列化される

オンメモリ処理・並列化の効果



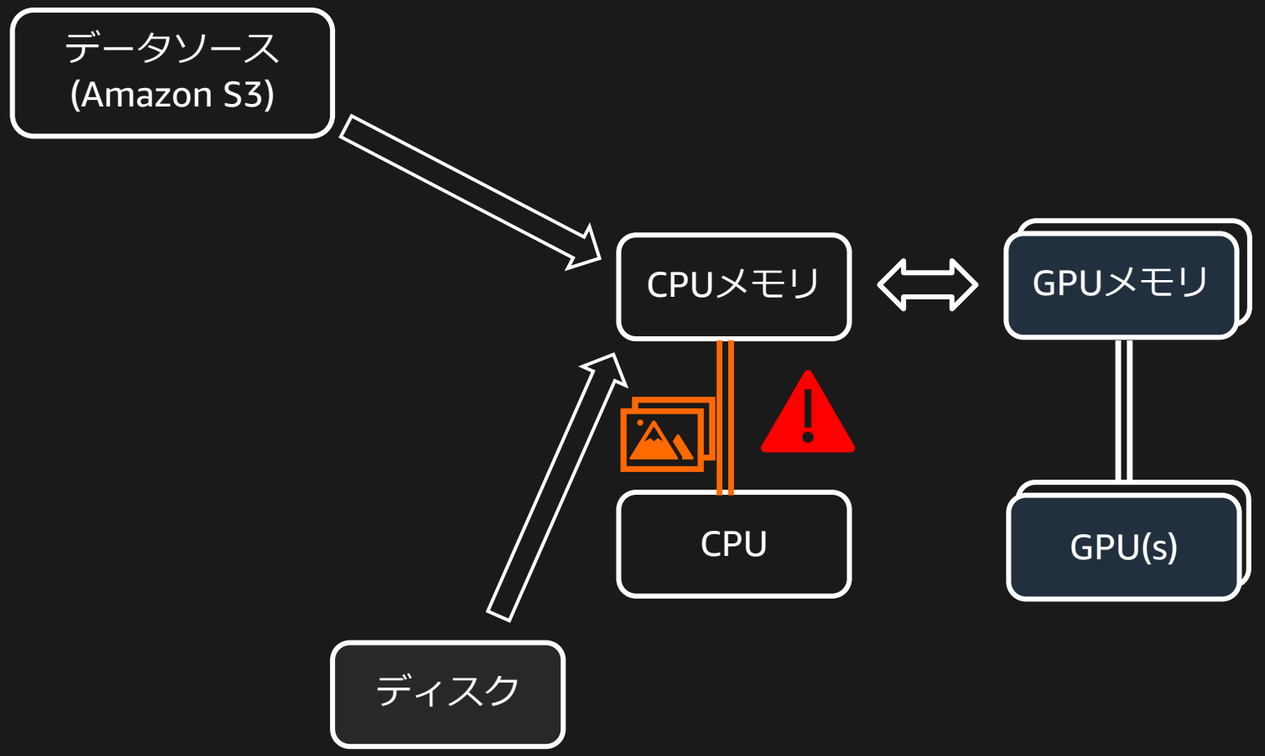
解決方法3：ローカルディスクからのロード速度の改善

NVMe SSDを搭載したインスタンスを選択することで、ファイルのロード速度を上げる

p3dn.24xlargeやg4dnはEBSの代わりにNVMe SSDを採用。

ローカルディスクにダウンロードした後のI/Oが飛躍的に向上

事例3: データの前処理



事例3: データの前処理

発見方法：ディスク使用率低、CPU使用率高、GPU利用率低

解決方法：

1. 前処理を並列処理する（事例2と同様なので割愛）
2. あらかじめ前処理しておく
3. GPUで前処理する

解決方法2：あらかじめ前処理しておく

前処理済みのデータを用意できればそれに越したことはない。
ただしそれが難しい場合もある。

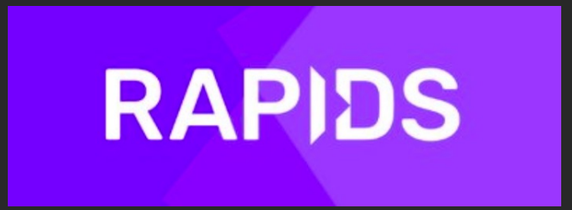
- データ拡張を行うため、データの保存・転送コストがかかる
- データの前処理自体を学習のパラメータとして検討したい

それでも、定型の処理に関しては事前に保存しておくのがベター

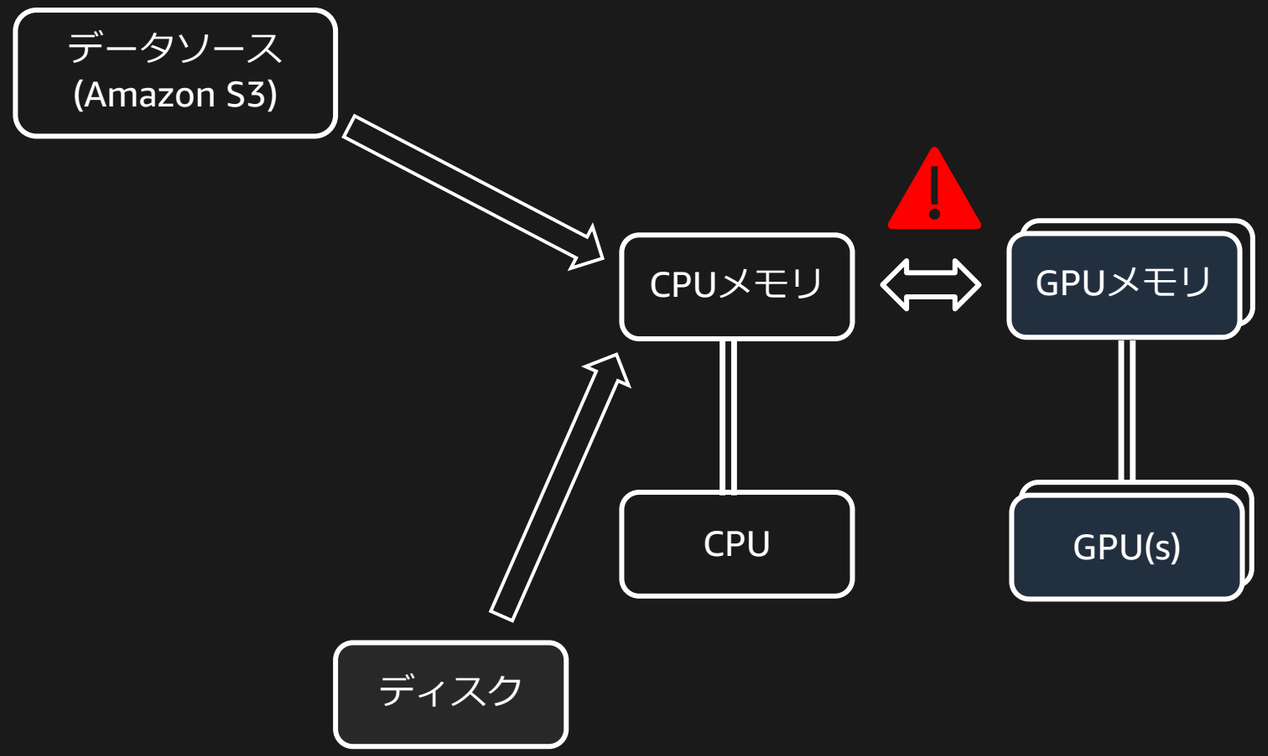
解決方法3：GPUで前処理する

- CuPy (数値データ)
- DALI (画像)
- cuDF in RAPIDS (DataFrame)

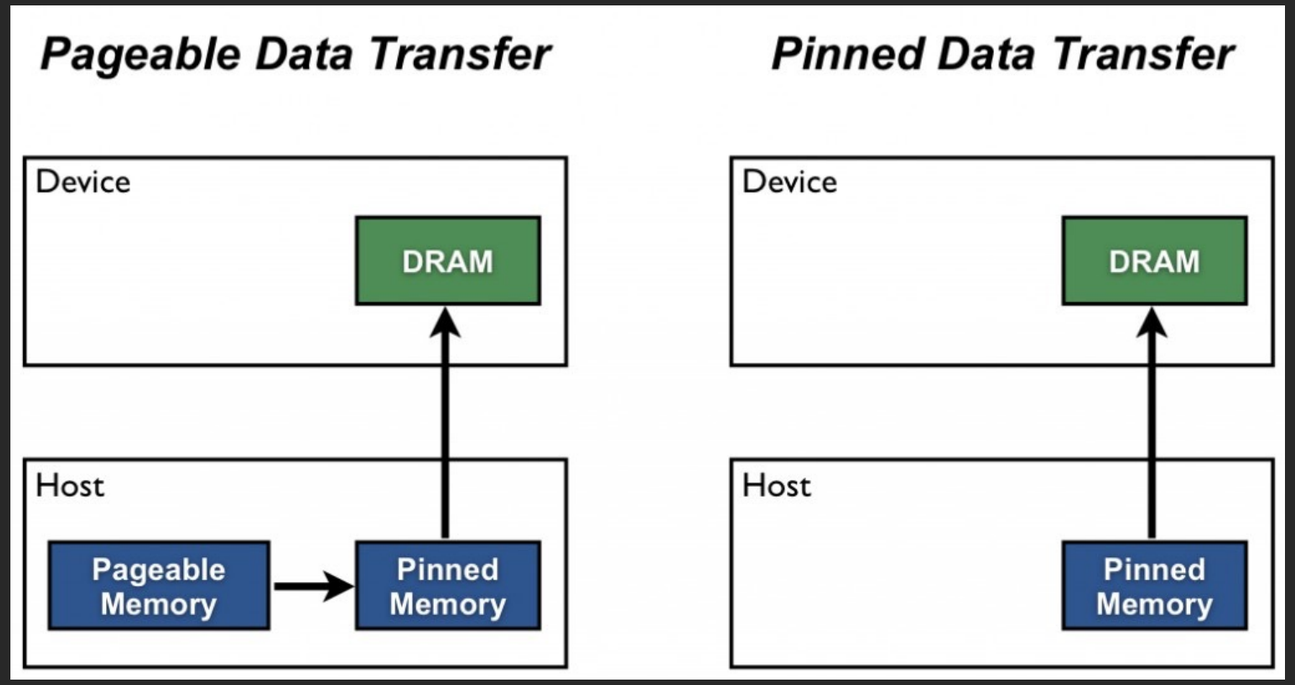
CuPy



事例4: GPUへのデータ転送



CPUからGPUへのデータ転送がボトルネックになることがわかっている場合、GPUのページロック領域メモリ (pinned memory)を直接確保することでパフォーマンスが向上する



https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

PyTorchの場合、DataLoaderのpin_memoryオプションをTrueにすることで設定できる

https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

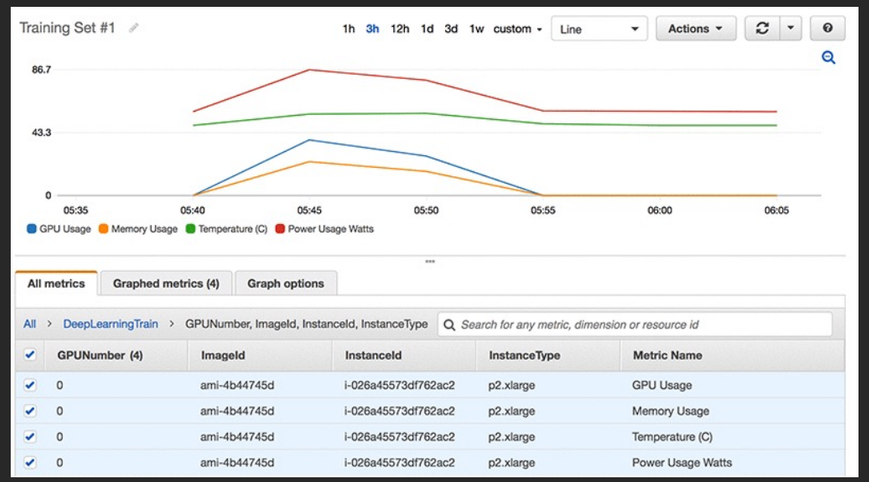
リソース最適化のベストプラクティス

まずはリソースの使用状況を確認する。

```
Every 1.0s: nvidia-smi                                  Wed Aug 11 09:29:44 202

Wed Aug 11 09:29:44 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.119.03   Driver Version: 450.119.03   CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| 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  Tesla V100-SXM2...  On   | 00000000:00:1E.0 Off |                    0 |
| N/A   67C    P0   258W / 300W |  11855MiB / 16160MiB |     94%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A     16341      C   python                          11853MiB |
+-----------------------------------------------------------------------------+
```

ターミナルからの確認 (nvidia-smi, top, etc...)

```
top - 09:33:07 up 21 days,  7:26,  0 users,  load average: 2.44, 1.80, 1.09
Tasks: 189 total,   4 running, 129 sleeping,   0 stopped,   1 zombie
Cpu(s): 19.7%us,  9.2%sy,  0.0%ni, 69.5%id,  0.0%wa,  0.0%hi,  1.6%si,  0.0%st
Mem:  62874704k total, 14213680k used, 48661024k free,  1147808k buffers
Swap:        0k total,        0k used,        0k free,  6903504k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
16341 ec2-user  20   0 32.1g 2.3g 585m R 101.4  3.9   4:12.22 python
19668 ec2-user  20   0 30.2g 1.9g  87m S 16.9  3.1   0:07.38 python
19665 ec2-user  20   0 30.2g 1.9g  87m S 14.9  3.1   0:07.38 python
 5338 root      20   0     0    0    0 D 12.9  0.0  40:52.91 nv_queue
```



CloudWatchでの確認方法

SageMakerのコンソール、学習ジョブからの確認方法

https://aws.amazon.com/jp/blogs/news/monitoring-gpu-utilization-with-amazon-cloudwatch/

ボトルネック特定のヒューリスティックス

- 学習開始まで時間がかかる
  - S3からのダウンロードがボトルネック（事例1）
- ディスク使用率高、CPU使用率低、GPU利用率低
  - ローカルディスクからのロードがボトルネック（事例2）
- ディスク使用率低、CPU使用率高、GPU利用率低
  - 前処理などのCPU処理がボトルネック（事例3：学習前の事前処理）
- ディスク使用率低、CPU使用率低～中、GPU利用率低
  - 前処理などのコードがCPUの性能を引き出せていない（CPU並列処理）

とはいえ、リソースの監視やボトルネック特定は大変

SageMaker Debugger

システムのリソース使用状況のモニタ・プロファイリング

システムリソース使用状況を自動的にモニタ

MLフレームワークのメトリクスを収集するために学習ジョブをプロファイリング

GPU, CPU, ネットワーク, メモリのリソース使用状況をSageMaker Studioを用いて可視化

リソース最適化のベストプラクティス

- 最初はデータのサブセットに対してローカルにテスト
  - GPUがフル稼働しない場合はデータロード、前処理、GPUデータ転送を改善
- フルセットのデータにスイッチ。必要に応じてデータをS3に配置
  - ネットワーク転送のボトルネックを検証。
- それでも学習時間がかかる場合はインスタンスの変更・分散学習を検討

Amazon SageMaker distributed training

巨大な深層学習モデルを学習するための最も高速かつ簡単な方法

トレーニング時間の短縮

GPU間の同期により、トレーニング時間を25%短縮します

AWS向けに最適化

AWS向けに設計されたデータ並列処理により、ほぼ線形のスケーリング効率を実現します

人気のあるMLフレームワークAPIのサポート

カスタムトレーニングコードなしでHorovodなどの既存のAPIを再利用する

自動で効率的なモデル分割

自動化されたモデルプロファイリングとパーティショニングの実験を回避

最小限のコード変更

10行未満のコード変更でモデルの並列処理を実装する

効率的なパイプライン

全てのGPUをアクティブに保つようなパイプラインを構築し、リソース使用率を最大化

Summary

- クラウド上でのML開発のメリット
  - インスタンスの選択肢、環境構築のラクさ
- GPUの学習ステップとボトルネックの種類
  - データ転送、ロード、前処理
- ボトルネックの対策方法とリソース最適化のベストプラクティス
  - リソースをモニタしつつ該当箇所を手当て

リソースを有効に使い、コストを抑えるためにも、GPUの稼働率をチェックしましょう

Thank you!!