第１２回計算科学技術特論B(2026)
GPUコンピューティング入門

NVIDIA
大友広幸

2026年7月2日（木）13:00－14:30

主催：高度情報科学技術研究機構(RIST)
　　次世代HPC・AI研究開発支援センター(HAIRDESC)
共催：東京大学物性研究所
後援：理化学研究所計算科学研究センター、
　　計算物質科学人材育成コンソーシアム(PCoMS)

GPU プログラミング入門

NVIDIA Developer Technology Engineer
大友広幸

講義について

いいから手を動かせ、いっぱい書け。

講義で使うサンプルコード: https://github.com/enp1s0/cst26b_gp

講義の流れ

- **1. GPU 入門**: 全員向け。CPU/GPU の違い、向く処理向かない処理、データ移動
- **2. Python から GPU を使う**: まず GPU 計算を試したい人。NumPy に近い形で始める
- **3. GPU ライブラリ**: よく使われる演算を速くしたい人。BLAS、FFT、SVD、疎行列、ベクトル検索
- **4. stdpar で GPU を使う**: C++ Standard algorithms を GPU で動かしたい人
- **5. 指示文ベース GPU 化**: 既存ループを大きく変えずに GPU 化したい人
- **6. CUDA カーネル**: GPU 上の計算を細かくカスタマイズしたい人
- **7. まとめ**: 自分のコードではどの方法から始めるべきかを整理
- **補遺. GPU コードの最適化**: GPU 化した後、遅い理由を測って改善したい人

Contents

GPU 入門：導入と全体像

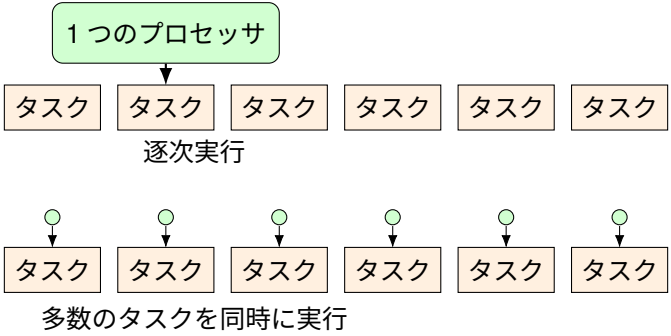
ここで見ること

- プログラムの並列性
- GPU がプログラム中で担う役割
- CPU と GPU の設計上の違い
- CPU/GPU メモリ空間とデータ移動（data movement）

ここで押さえること

- GPU は多数の独立した仕事に強い
- スループット重視
- 小規模・逐次依存・不規則アクセスは苦手
- 計算量だけでなく、データ移動もプログラム全体の性能を左右

逐次実行と並列実行



- GPU は並列計算に特化したプロセッサ

並列化しやすい処理: 処理同士の依存が少なく、処理内容がそろっている

並列化しにくい処理: 前のステップが終わるまで次へ進めない

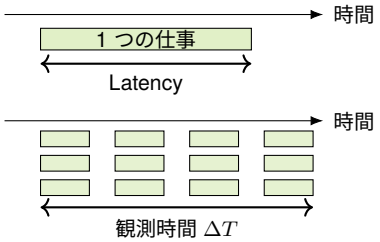
計算の速さ: スループットとレイテンシ

性能を考える 2 つの視点

- **Latency**: 1 つの処理を開始してから結果が返るまでの時間
- **Throughput**: 単位時間あたりに完了する処理の量

高 Throughput と低 Latency は別の目標。個々の仕事が短くならなくても、多数を並行して処理すれば Throughput は上げられる

時間軸で見る違い

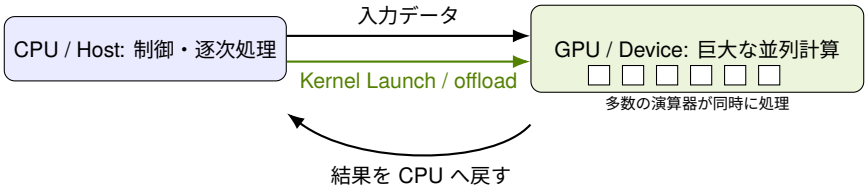


Throughput $=$ 完了した処理量 $/ \Delta T$

GPU の強み: 行列要素、ピクセル、格子点、粒子などの独立な処理を大量に用意し、まとめて実行して高い Throughput を得る。

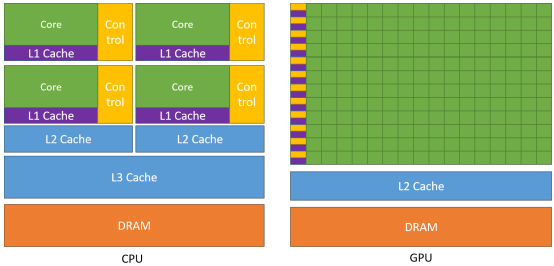
CPU-GPU 実行モデル

- GPU を使うプログラムは、CPU と GPU が協調するヘテロジニアスシステム上で動く
- CPU 側を **Host**、GPU 側を **Device** と呼ぶ
- CPU は大きな計算を GPU へオフロード（offload）し、結果を受け取る
- **Host code** が GPU 処理を起動し、必要なデータ移動を管理



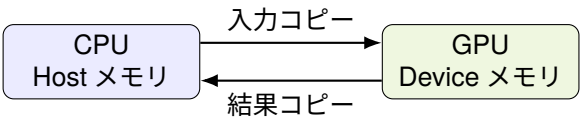
CPU と GPU の設計上の違い

- CPU は幅広い処理に対応するため、制御フローやキャッシュに多くの回路資源を使う
- GPU は、データ並列処理のユニットとメモリ帯域に多くの回路資源を振り向ける
- そのため、大規模で規則的なスループット重視の処理に強い



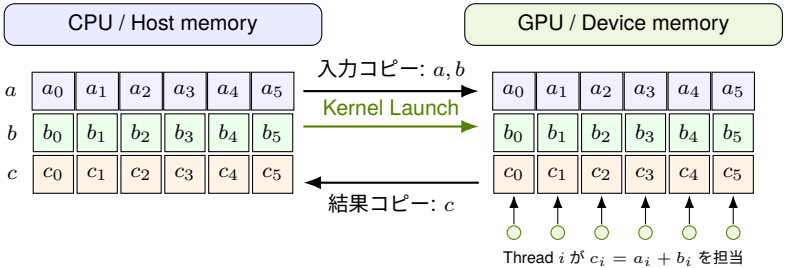
出典: NVIDIA CUDA Programming Guide, CPU/GPU のトランジスタ利用の図。

CPU と GPU のメモリ空間、データ移動



- データ転送には無視できないコスト
- 複数の GPU 演算で使うデータは、できるだけ GPU 上に置いたままにする
- GPU プログラミングでは、データをどこに置き、いつ動かすかも設計する

例：ベクトル加算



- 入力配列を GPU へ送り、GPU 上で多数の Thread が同じ式を別々の要素に適用

GPU に向いている処理、向いていない処理

向いている：演算器を使い切りやすい

- **大きな問題**: 多数の Thread へ仕事を配り、GPU 全体を動かせる
- **高い並列度**: 各要素をほぼ独立に処理可能
- **規則的な処理**: 演算量がそろい、連続したメモリアクセスが多い
- **GPU 上で計算を継続**: データを再利用し、転送コストを複数の演算で回収

向いていない：待ち時間や無駄が増えやすい

- **小さな問題**: Kernel Launch や data movement のコストが相対的に大きい
- **強い逐次依存**: 前の結果を待つため、同時に進められる仕事が少ない
- **不規則な分岐・アクセス**: Thread ごとの処理時間がばらついたり、メモリ帯域も使いにくい
- **頻繁な CPU–GPU 往復**: 計算よりデータ転送や同期が支配的

アルゴリズムだけでなく、問題サイズ、並列度、メモリアクセス、data movement を確認。同じアルゴリズムでも入力サイズや実装によって適性は変わる

まとめ：実行モデルと用語

ここまでの整理

- GPU は大量の独立した仕事を同時に処理
- GPU は基本的にスループット重視
- GPU に向くのは、大規模・反復的・規則的な処理
- CPU–GPU 間のデータ移動は性能に大きく影響
- 可能なら中間データは GPU 側で保持

並列計算 / GPU プログラミング Terms

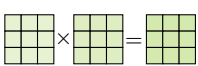
- **Host**: プログラムの CPU 側
- **Device**: プログラムの GPU 側
- **Kernel**: GPU 上で実行される関数
- **Launch**: CPU が Kernel を立ち上げ
- **Thread**: GPU 上の論理的な実行単位
- **Offload**: 大きな計算を GPU へ渡す考え方

まとめ：GPU を使う前のチェックリスト

GPU 化を検討するポイント

1. その部分は実行時間のホットスポットか
2. 問題に多数のデータ要素があるか
3. 多数の要素を独立に処理できるか
4. 演算はほぼ同じ種類か
5. 不要なデータ転送を避けられるか

GPU が得意な計算の例



行列・Tensor 演算
多数の積和演算

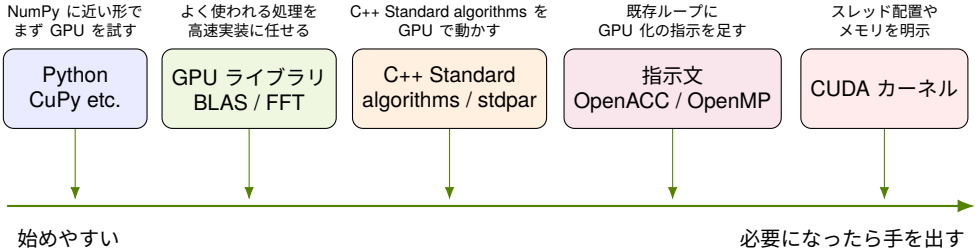
画像・格子計算
Pixel / 格子点ごとの処理

粒子計算
粒子ごとの状態更新

独立な試行・Batch 処理
Sample ごとの計算

多くが「はい」なら、GPU は有力な選択肢

以降の話：多様な GPU プログラミング手法



- まず左側から検討: 少ない変更で GPU 化できる可能性がある
- 右側ほど細かく制御できるが、データ移動、Thread 配置、同期などを自分で考える範囲が増える
- 以降の章では、この順にベクトル加算と内積を例に具体的な方法を見る

Contents

Python から GPU を使う：導入と全体像

ここで見ること

- NumPy に近い形で GPU を使う CuPy
- Python コード、配列、演算がどこで動くか
- CPU メモリと GPU メモリをまたぐ転送
- 中間データを GPU 上に残す考え方
- Python から使える GPU 対応パッケージ

ここで押さえること

- Python 自体は CPU 上で動き、GPU 処理を起動
- `cupy.ndarray` のデータは GPU メモリ上
- 大きな演算にまとめるほど GPU の性能を引き出しやすい
- GPU カーネルを書かずに GPU 計算を始められる

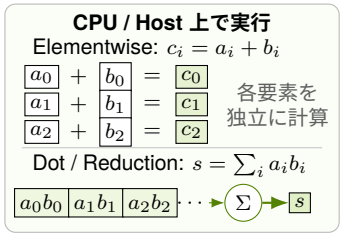
C 基準コード: ベクトルの加算・内積

```
// vector addition
for (int i = 0; i < n; ++i) {
  c[i] = a[i] + b[i];
}

// inner product
float s = 0.0f;
for (int i = 0; i < n; ++i) {
  s += a[i] * b[i];
}
```

- ベクトルの加算: 各入力要素から 1 つの出力要素を作る要素ごとの演算（Elementwise）
  - 全要素独立に計算可能 ⇒ 並列計算向き
- ベクトルの内積: 多数の積を 1 つのスカラーへ集約（**Reduction**）
  - 並列計算では要素の足し込みの部分を工夫の必要あり

NumPy: 簡潔な CPU 配列コード

```python
import numpy as np

n = 1_000_000
a = np.arange(n, dtype=np.float32)
b = np.ones(n, dtype=np.float32)

c = a + b              # vector addition
s = np.dot(a, b)       # inner product
```

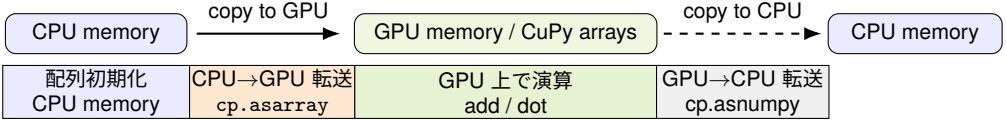


- NumPy では、明示的なループを書かずに配列演算を書ける
- ただし、データは CPU メモリ上。演算も CPU 上。

CuPy: GPU 上での処理へ

```python
import numpy as np
import cupy as cp

n = 1_000_000
a = np.arange(n, dtype=np.float32)
b = np.ones(n, dtype=np.float32)

a_gpu = cp.asarray(a)  # CPU to GPU
b_gpu = cp.asarray(b)  # CPU to GPU

c_gpu = a_gpu + b_gpu  # on GPU
s_gpu = cp.dot(a_gpu, b_gpu)  # on GPU

c = cp.asnumpy(c_gpu)  # GPU to CPU
```

- NumPy 配列: CPU メモリ上のデータ
- `cp.asarray`: 入力を CPU から GPU へコピー
- CuPy 配列への演算: GPU 上で実行
- `cp.asnumpy`: 必要な結果だけ CPU へ戻す



中間データを GPU 上にのみ保持する

- どちらも $d = 2(a + b)$ を求めるコード。違いはデータ移動

避けたい（極端な）例: 演算の間で転送

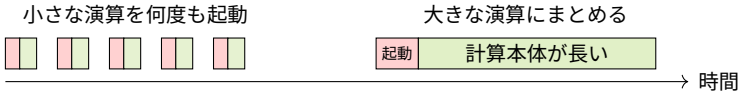
```
a_gpu = cp.asarray(a)
b_gpu = cp.asarray(b)

c_gpu = a_gpu + b_gpu
c_cpu = cp.asnumpy(c_gpu)  # GPU -> CPU

c_gpu = cp.asarray(c_cpu)  # CPU -> GPU
d_gpu = 2.0 * c_gpu
d_cpu = cp.asnumpy(d_gpu)
```

改善例: 中間結果を GPU 上にのみ保持

```
a_gpu = cp.asarray(a)
b_gpu = cp.asarray(b)

c_gpu = a_gpu + b_gpu
d_gpu = 2.0 * c_gpu

d_cpu = cp.asnumpy(d_gpu)  # copy once
```

- 不必要な CPU–GPU データ転送は避けたほうが良い

多数の小さな演算は非効率になりやすい



例: `A`, `B`, `C` は shape (N, D) の CuPy 配列。計算量は同じでも、GPU 演算の起動回数が違う。

避けたい例: N 回の小さな演算

```
for i in range(N):
    C[i, :] = A[i, :] + B[i, :]
```

- D 要素の加算を N 回起動
- D が小さいと起動コストが目立つ

改善例: 行列全体を 1 回で処理

```
C = A + B
```

- $N \times D$ 要素を 1 回の配列演算へ
- GPU に十分な量の計算をまとめて渡す

キーワード：Batch 計算

NumPy コードを CuPy へとりあえず移す

NumPy: CPU 上の配列

```
import numpy as np

n = 1_000_000
a = np.arange(n, dtype=np.float32)
b = np.ones(n, dtype=np.float32)

c = a + b
s = np.dot(a, b)
```

CuPy: GPU 上の配列

```
import cupy as np

n = 1_000_000
a = np.arange(n, dtype=np.float32)
b = np.ones(n, dtype=np.float32)

c = a + b
s = np.dot(a, b)
```

- `import numpy as np` を `import cupy as np` に変えるだけで GPU 配列コードになる
- 配列作成と配列演算の書き方はほぼ同じ。変わるのはデータの場所と実行場所
- 実コードで NumPy と CuPy を混ぜる場合は、`np` と `cp` を分けた方が可読性が高い

まとめ: Python から GPU を使う

変わるもの

- 配列の型: NumPy 配列から CuPy 配列へ
- データの場所: CPU メモリから GPU メモリへ
- 演算の実行場所: CPU 演算から GPU 演算へ
- 転送点: `asarray` と `asnumpy`

使うときのコツ

- 不必要に中間結果を CPU へ戻さない
- 必要な結果だけ CPU へ戻す
- 小さな演算を Python ループで何度も起動しない。できれば batch 処理。

GPU 対応 Python パッケージ例

Numerical and ML

- **CuPy**: NumPy-like GPU 配列
- **PyTorch**: 深層学習
- **TensorFlow**: 深層学習
- **JAX**: 自動微分・JIT compilation

データサイエンス

- **cuDF**: データフレーム
- **cuML**: 機械学習
- **cuGraph**: グラフ解析
- **cuVS**: ベクトル検索

Contents

3. GPU ライブラリ：導入と全体像

ここで見ること

- 標準的な演算をライブラリ関数として使う発想
- NVIDIA が提供しているいくつかのライブラリ

ここで押さえること

- 自分でカーネル関数を書く前にまずは既存ライブラリを探す
- ときには自分の処理を既存ライブラリに合うように変更
- OSS のライブラリは是非 Contribution を！

ライブラリから始める理由

ライブラリを使うと良いこと

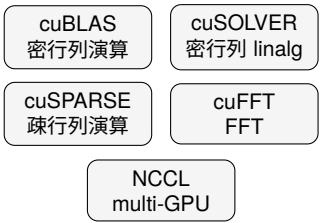
- 楽
  - 自分で GPU カーネルを最適化しなくて良い
  - 概ね信用できる。明らかなバグが少ない

こういった処理はライブラリがある可能性が高い

- 数学的に名前のある処理
- 多くのアプリケーションで繰り返し使われる処理

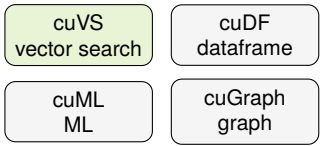
この講義で見る２種類のライブラリ

標準的な計算・通信の部品



- BLAS 相当、LAPACK 相当、FFT、sparse、collective communication
- 数値線形代数や複数 GPU 通信の標準部品を GPU で実行

アプリケーションに近い高水準ライブラリ

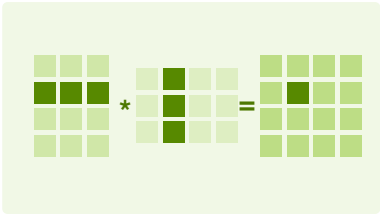


- 例：RAPIDS
- dataframe、ML、graph、nearest neighbor search
- この講義では cuVS を例として紹介

- 自分の計算を「どの標準部品に分けられるか」で見ると、使うライブラリを選びやすい

cuBLAS: 密なベクトル・行列演算

- 密行列数値線形代数の基本ライブラリ
- BLAS 演算を提供するライブラリ
- 行列積（GEMM）$C = AB$ は AI、数値シミュレーション、統計、などで頻出



BLAS level	演算の形	代表例
Level 1	vector-vector	AXPY、DOT、NRM2
Level 2	matrix-vector	GEMV、triangular solve
Level 3	matrix-matrix	GEMM、batched GEMM（cuBLAS 独自拡張）、SYRK

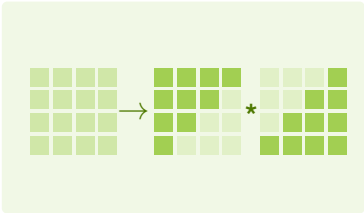
cuBLAS: AXPY と DOT の例

```
// AXPY: y = alpha * x + y
float alpha = 1.0f;
cublasSaxpy(handle, n, &alpha, d_x, 1, d_y, 1);

// DOT: s = x^T y
float s = 0.0f;
cublasSdot(handle, n, d_x, 1, d_y, 1, &s);
```

- `d_x`, `d_y`: GPU memory 上の配列
- AXPY は、多くの反復法や最適化アルゴリズムに現れるベクトル更新
- DOT は内積を計算する基本的な reduction。1 つの呼び出しで並列化済み
- 引数はドキュメントを確認: https://docs.nvidia.com/cuda/cublas/

cuSOLVER: 方程式求解と行列分解

- LAPACK-like な solve / factorization を GPU で実行
- 線形方程式、最小二乗、行列分解、固有値問題を扱う
- cuSolverDN: dense matrix 向け
- cuSolverSP: sparse matrix 向け（一部 cuDSS に移行中）

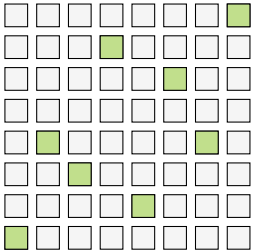


cuSolverDN でできる処理例

- 線形方程式
- 最小二乗法
- LU 分解
- QR 分解
- Cholesky 分解
- 特異値分解
- 固有値分解

- ドキュメント: https://docs.nvidia.com/cuda/cusolver/

cuSPARSE: 疎行列計算



保存形式

- CSR、COO、CSC、BSR 等

代表的な処理

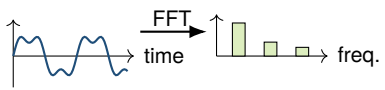
- SpMV: sparse matrix-dense vector multiply
- SpMM: sparse matrix-dense matrix multiply
- SpGEMM: sparse matrix-sparse matrix multiply
- triangular solve、format conversion

向く問題

- 非ゼロ要素だけを使い、memory bandwidth と計算量を節約
- グラフ処理、数値シミュレーション、AI・データサイエンスなどで利用

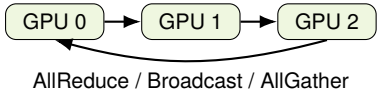
cuFFT と NCCL: 変換と通信

cuFFT: 周波数領域へ変換



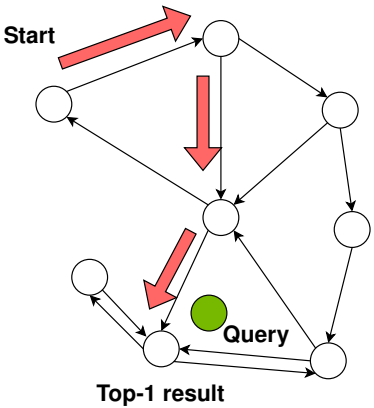
- DFT を高速に計算する $O(n \log n)$ algorithm
- `float`、`double`。1D / 2D / 3D。batched transform

NCCL: GPU 間通信



- multi-GPU / multi-node 向けの communication library
- 集団通信: AllReduce、Broadcast、Reduce、AllGather、ReduceScatter
- 一対一通信: send / receive
- PCIe、NVLink、InfiniBand などの topology を意識して通信を最適化

cuVS: ベクトル検索とクラスタリング



cuVS でできること

- 近似近傍探索（ANNS）、K-Means、etc
- データセットからクエリと似たものを大雑把に見つける

CAGRA の位置づけ

- GPU 向けの graph-based ANNS 手法・実装
- データ点をグラフとして持ち、近い点へたどって探索
- GitHub: https://github.com/NVIDIA/cuvs
- issue、PR お待ちしています

H. Ootomo et al., CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs

その他ライブラリ例

Math

- cuRAND: 乱数生成
- cuTENSOR: Tensor 縮約等

機械学習

- cuDNN: ニューラルネットの基本処理
- CUTLASS: 行列積

データ処理

- nvComp: データ圧縮
- cuGraph: Graph 解析

並列アルゴリズム

- Thrust: C++ STL ベースの並列データ処理
- CUB: 並列アルゴリズム primitives (e.g. in-kernel sort)

- まずはライブラリを組み合わせることで自分の計算を実現できないか考える。
- NVIDIA CUDA-X Libraries: https://developer.nvidia.com/cuda/cuda-x-libraries

Contents

C++の抽象化で GPU を使う：導入と全体像

ここで見ること

- CUDA カーネルを直接書かない C++による GPU プログラミング
- stdpar: C++ Standard algorithms を並列実行・GPU offload
- stdpar で書きやすい処理、書きにくい処理
- 参考: Kokkos

押さえてほしいこと

- C++ Standard algorithms で書きやすい処理・書きにくい処理
- 実行ポリシー（execution policy）とは
- コンパイラが GPU 実行へ変換する流れ

stdpar では、GPU 上での Thread 割り当てなどを考えずに、処理を GPU 上で並列実行することができる

stdpar は NVIDIA HPCSDK のコンパイラで利用可能

stdpar: C++ Standard algorithms を GPU で実行

C++ Standard algorithms とは

C++の標準ライブラリに含まれるデータ操作アルゴリズム群

- `std::transform`: 要素ごとの変換
- `std::reduce`: 縮約
- `std::transform_reduce`: 要素ごとの変換をしてから縮約
- sort、search など

コードに書くこと

- Standard algorithm
- 実行ポリシー
  - `std::execution::par` / `par_unseq`

裏で起きること

- 並列実行してよい範囲がコンパイラに伝わる
- NVIDIA HPC SDK が対応する実行ポリシー付き algorithm を GPU へ offload

例: stdpar を用いたベクトル加算と内積

```cpp
std::vector<float> a(n), b(n), c(n);

// vector add
std::transform(std::execution::par_unseq,
               a.begin(), a.end(), b.begin(), c.begin(),
               [](float x, float y) { return x + y; });

// inner product
float s = std::transform_reduce(std::execution::par_unseq,
               a.begin(), a.end(), b.begin(), 0.0f,
               std::plus<float>(), std::multiplies<float>());
```

- loop 本体ではなく、「何をしたいか」を C++ Standard algorithm で表す
- `transform`: 各要素を独立に計算。ベクトル加算と相性がよい
- `transform_reduce`: 要素ごとの積を作り、最後に 1 つへ集約。内積に対応
- `par_unseq`: 並列化と vector 化を許す実行ポリシー

stdpar: コンパイルと Memory mode

- NVIDIA HPC SDK のコンパイラでコンパイル可能

```
# GPU offload for C++ Standard algorithms
nvc++ -std=c++17 -O2 -stdpar=gpu main.cpp -o main
```

コンパイラオプション

- `-stdpar=gpu`: 実行ポリシー付き algorithm を GPU 実行へ

Memory mode

- `-stdpar=gpu` では、`std::vector` のような heap 領域のデータは GPU からアクセス可能
  - 備考: `mem:managed/unified` が暗黙的につく
- その他 stack 上の配列や global/static data は Memory mode の制約に注意

データ転送は必要時に自動的に行われる。

stdpar で書きやすい処理・書きにくい処理

書きやすい処理

- 各要素の処理がほぼ独立
- 各要素に対する同じ処理の繰り返し
- データの index に依存しない
- Elementwise な変換や Reduction として自然に書ける集約

書きにくい処理

- データの要素間の依存が強い処理
- Shared Memory や Warp 単位の制御が必要な処理
- 複雑な Memory layout を細かく最適化したい処理
- Thread 割り当てを自分で決めたい処理

C++ Standard algorithm で自然に書けるなら stdpar。Thread や Memory を直接制御したいなら指示文ベースや CUDA C++を検討

参考: Kokkos

詳細は計算科学技術特論 A の似鳥先生の講義にて。

位置づけ

- C++で GPU 向け並列処理を書く別の選択肢
- `parallel_for` / `parallel_reduce` などの API で並列 pattern を表す
- 配列や memory layout は `Kokkos::View` などで扱う

stdpar との違い

- C++ Standard algorithm ではなく専用 API
- データ・メモリ空間の抽象化
- より柔軟な並列パターン・性能チューニングが可能

Kokkos Core Programming Guide: https://kokkos.org/kokkos-core-wiki/

まとめ: stdpar で GPU を使う

この節で押さえること

- stdpar は、C++ Standard algorithm を実行ポリシーによって並列実行する方法
- `nvc++ -stdpar=gpu` で GPU offload を有効化
- CPU–GPU 間のデータ転送は自動

向き・不向き

- C++ Standard parallel で自然に書ける処理に向く
- 既存 C++コードを大きく壊さず試しやすい
- 低レベルでの性能最適化などは苦手

NVIDIA HPC SDK stdpar:
https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html#using-stdpar

NVIDIA HPC SDK memory modes:

https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html#gpu-memory-modes

Contents

5. 指示文ベース GPU 化：導入と全体像

ここで見ること

- 主に OpenACC
- 既存 CPU ループに指示文（directive）を足す方法
- OpenMP を使った multi thread 実行の C コードをまず GPU 化する例
- Managed Memory

ここで押さえること

- CPU コードの形を大きく変えずに GPU 化を試せる
- `malloc` / `free` のまま GPU 化を試せる
- Managed Memory でデータコピーを暗黙に扱う
- `-Minfo` で GPU 化された loop を確認

OpenMP での multi-thread 用コードとの差分は小さい

OpenMP/CPU 版

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
  c[i] = a[i] + b[i];
}
```

OpenACC/GPU 版

```
#pragma acc parallel loop
for (int i = 0; i < n; ++i) {
  c[i] = a[i] + b[i];
}
```

- ループ本体は同じ。変更するのはコンパイラへの指示
- まず GPU code が出ているかを確認し、次にデータ移動を詰める

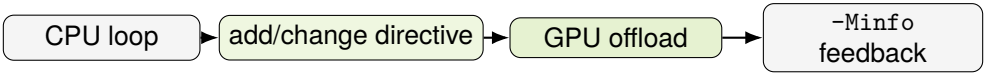
CPU コードから GPU コードへ: まず変える場所

最初は変えないもの

- 配列の型と添字アクセス
- `malloc` / `free`
- 初期化、結果確認、I/O
- ループ本体の数式

最初に足すもの・変更するもの

- 並列化したいループの直前に acc directive
- 内積などには reduction 句
- OpenACC/GPU 対応のコンパイラへ
- NVIDIA HPCSDK を使っているのであれば、最初はとりあえず`-gpu=mem:managed`



OpenACC: ベクトル加算

```cpp
auto *a = (float*)malloc(n * sizeof(float));
auto *b = (float*)malloc(n * sizeof(float));
auto *c = (float*)malloc(n * sizeof(float));

for (int i = 0; i < n; ++i) {
  a[i] = static_cast<float>(i); b[i] = 1.0f;
}

// Add one directive before the CPU loop
#pragma acc parallel loop
for (int i = 0; i < n; ++i) {
  c[i] = a[i] + b[i];
}

printf("c[0] = %f\n", c[0]);
free(a); free(b); free(c);
```

- `malloc` / `free`、初期化、結果確認は通常の CPU コードに近い形
- `pragma acc parallel loop` で、直後のループを GPU 実行候補としてコンパイラに伝える
- `-gpu=mem:managed` を使うと、明示的なデータ移動が不要

OpenACC: 内積は reduction を指定

ベクトル加算: 各要素が独立

```
#pragma acc parallel loop
for (int i = 0; i < n; ++i) {
  c[i] = a[i] + b[i];
}
```

内積: 要素積を集約

```
float sum = 0.0f;
#pragma acc parallel loop reduction(+:sum)
for (int i = 0; i < n; ++i) {
  sum += a[i] * b[i];
}
```

- `parallel loop`: 反復間に依存がないループを並列化
- `reduction(+:sum)`: 各並列実行単位の部分和を安全に結合
- CPU 版の数式を保ったまま、並列化に必要な情報だけを追加

コンパイル例:

```
# OpenACC with NVIDIA HPC SDK C++ compiler
nvc++ -acc -gpu=mem:managed -Minfo=accel \
    vector_add.cpp -o vector_add_acc
```

- `-acc`: OpenACC 指示文を有効に
- `-gpu=mem:managed`: `malloc` などの動的確保データで CUDA Managed Memory を使う
- 最初の例の `a`, `b`, `c` を CPU コードと GPU コードの両方から参照しやすくする
- Managed Memory は書き方を簡単にするが、データ移動のコストは消えない
- `-Minfo=accel`: コンパイラフィードバックを表示

Memory mode: なぜ `mem:managed` を使うのか

最初に使いやすい理由

- `malloc` / `free` ベースの既存コードを崩さず試しやすい
- 1 つのポインタを CPU コードと GPU コードの両方から参照できる
- 動いた後で、必要なら明示的な data 句へ進む（後のスライドで説明）

いらなくなるもの

- プログラマによる明示的なデータコピーの指示

残るもの

- データは依然として CPU メモリと GPU メモリの間を移動

コンパイラフィードバックを表示

ここで見ること

- GPU code が生成されたか
- どの loop が並列化されたか
- 多重ループが collapse されたか
- gang / vector などの並列形態
- 暗黙の copyin / copyout

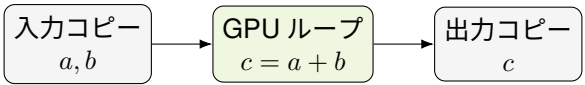
```
// ... -acc -Minfo=accel ...
main:
  12, Generating NVIDIA GPU code
  16, #pragma acc loop gang, vector(128)
      /* blockIdx.x threadIdx.x */
  12, Generating implicit copyin(a[:])
      Generating implicit copyin(b[:])
      Generating implicit copyout(c[:])
```

- `-Minfo=accel` は、コンパイラが「どこを GPU 化し、何をコピーすると判断したか」を見るための出力
- 性能最適化のための重要なヒント

OpenACC data 句: データ転送を陽的に指示

- 主に Managed memory 不使用時に必要

```
#pragma acc data copyin(a[0:n], b[0:n]) copyout(c[0:n])
{
#pragma acc parallel loop
  for (int i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
  }
}
```



代表的なもの

- `copy`: 入る時に Device へ、出る時に Host へ
- `copyin`: 入る時に Device へ
- `copyout`: 出る時に Host へ

補足: OpenMP offload によるベクトル加算

OpenMP offload のシンプルな例

```
// OpenMP 5.0
#pragma omp target teams loop
for (int i = 0; i < n; ++i) {
  c[i] = a[i] + b[i];
}
```

- `target`: GPU で実行する領域、`teams loop`: GPU 上でよしなに並列計算
- NVHPC では `-mp=gpu` で有効化。Managed Memory を使うなら `-gpu=mem:managed`

OpenACC と OpenMP offload の比較

観点	OpenACC	OpenMP offload
loop 指示	`acc parallel loop`	`omp target teams loop`
compile	`-acc`	`-mp=gpu`
memory mode	`mem:managed`	`mem:managed`
data 制御	`data/copy`	`target data/map`

指示文ベースが合うケース・合わないケース

- 既存のループベースの C/C++/Fortran コードがある
- 重要なループに、ほぼ独立した反復が多数ある
- stdpar などでは対応できない独自演算が必要だが、明示的な CUDA スレッドインデックスまでは不要

不向きな場合: アルゴリズムが GPU 固有の細かな同期、Shared Memory タイリング、独自のスレッド配置を必要とする場合、指示文オフロードは合いにくい

書き換えが必要なケース: 例: 並列化するループ内の処理が大きく複雑な場合は性能を出すために書き換えが必要になることがある

まとめ: 指示文ベース GPU 化

- OpenACC は、loop に `pragma acc parallel loop` を足して GPU 化を試す方法
- OpenACC を試す最小コマンド: `nvc++ -acc -gpu=mem:managed -Minfo=accel`
- 配列確保や loop 本体をあまり変えずに GPU 化が可能
- Managed Memory は最初の例を簡単にするが、性能はデータ移動に依存
- OpenMP offload も、既存 loop を GPU へ移す別の directive-based な方法
- GPU ライブラリ（cuBLAS など）との併用も可能

NVIDIA HPC Compilers OpenACC/OpenMP:

https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html

Contents

CUDA カーネル：導入と全体像

ここで見ること

- GPU の中がどうなっているのか
- GPU カーネルがどういうものか
- CUDA の thread 階層
- CUDA の memory 階層
- copy、sync、error check
- Reduction 処理の書き方

CUDA カーネルを自分で書くタイミング

- 既存ライブラリの組み合わせや C++ Standard algorithm、指示文ベースで記述が難しい
- 性能最適化のために細かく処理や thread 割り当て等を調整したい
- その他どうしようもなくなったとき

GPU 上での演算や thread 割り当て等を細かく指定することが可能

GPU の中: GH100 Full GPU with 144 SMs

출처: NVIDIA Technical Blog, NVIDIA Hopper Architecture In-Depth, Figure 3.

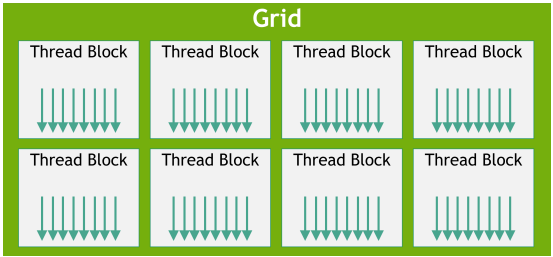
GPU の中: Streaming Multiprocessor (SM)

SM 内部の主な資源

- 演算器（FP64 / FP32 / INT32 Core、Tensor Core、etc）
- Load / Store Unit
- Register file
- Shared memory / L1 Cache
- Warp Schedulers / Dispatch Units

出典: NVIDIA Technical Blog, NVIDIA Hopper Architecture In-Depth, Figure 4.

CUDA の Thread 階層: Grid / Thread Block



出典: NVIDIA CUDA Programming Guide, “Grid of Thread Blocks.”

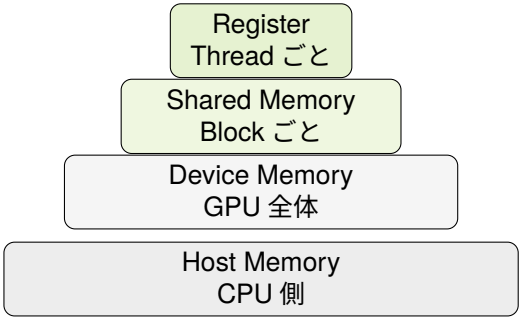
Thread 階層

- Thread は Thread Block というまとまりを持ち、Thread Block は Grid というまとまりを持つ
- Kernel Launch に Thread Block のサイズと Grid のサイズを指定

Hardware との関係

- 1 つの Thread Block は 1 つの SM に割り当てられる
- 同じ Thread Block 内の Thread は Shared memory を共有
- Warp と呼ばれる 32 Threads 単位で演算等は実行される

Memory 階層



- Register: 1 つの Thread 専用。最も演算器に近い。
- Shared Memory: 同じ Thread Block 内の Threads で共有。協調処理に使う
- Device Memory: `cudaMalloc` や `cudaMallocManaged` で扱う大きなメモリ

CUDA カーネル: ベクトル加算

- Device code:

```
__global__
void add_kernel(const float* a,
                const float* b,
                float* c,
                unsigned n) {
  const unsigned tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid < n) {
    c[tid] = a[tid] + b[tid];
  }
}
```

- `__global__`: CPU から起動し、GPU 上で実行される関数
- 全 Thread が同じコードを実行し、`blockIdx`、`blockDim`、`threadIdx` の値だけが異なる
- `tid`: この Thread が担当する配列要素番号
- まずは「1 Thread = 1 element」の形で読む
- `if (i < n)`: 余った Thread による配列外アクセスを防ぐ

CUDA Kernel Launch

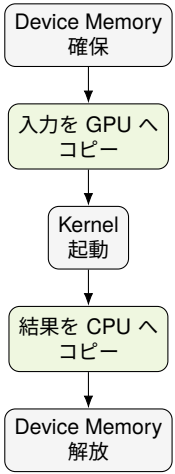
```
unsigned block_size = 256;
unsigned grid_size = (n+block_size-1)/block_size;

add_kernel<<<grid_size, block_size>>>(da, db, dc, n);
```

Launch に指定するもの

- 1 Thread block あたりの Thread 数
- Grid 全体の Thread block 数

- `(n+block_size-1)/block_size`: 整数除算で切り上げるよく使う書き方
  - `grid_size*block_size` が `n` 以上となる最小の `grid_size`
- `<<<grid_size, block_size>>>`: CUDA Kernel Launch configuration
- この時、十分な数の `grid_size` がないと、GPU の資源を使い切れない
- Kernel Launch は非同期。結果を CPU で読む前に完了待ち

ベクトル加算全体の流れ




```cpp
auto bytes = n * sizeof(float);
float *da, *db, *dc;

cudaMalloc(&da, bytes);
cudaMalloc(&db, bytes);
cudaMalloc(&dc, bytes);

cudaMemcpy(da, ha, bytes,
           cudaMemcpyDefault); // H2D da
cudaMemcpy(db, hb, bytes,
           cudaMemcpyDefault); // H2D db

// Kernel launch
add_kernel<<<grid_size, block_size>>>(da, db, dc, n);

cudaMemcpy(hc, dc, bytes,
           cudaMemcpyDefault); // D2H dc

cudaFree(da); cudaFree(db); cudaFree(dc);
```

- cudaMemcpy は同期処理を含む

CUDA プログラムのコンパイル

```
# Basic CUDA C++ compilation
nvcc -arch=sm_100 vector_add.cu -o vector_add
```

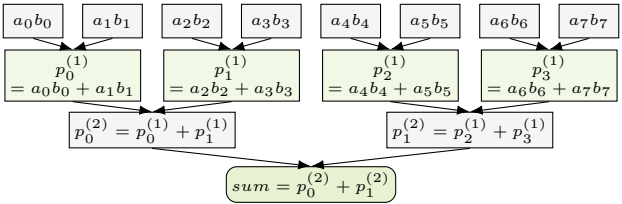
解説

- `nvcc`: CUDA 用のコンパイラ
- `.cu`: Host code と Device code を含む CUDA C++ソース
- `-arch=sm_100`: 実行する GPU 世代を指定
  - `sm_80`: Ampere
  - `sm_90`: Hopper
  - `sm_100`: Blackwell

コンパイラの動き

- .cu は Host code と Device code に分ける
- Device code から kernel image を作る
- Host code は host のコンパイラ（gcc 等）によってコンパイル
- 上記 2 つを 1 つのファイルに統合

内積: Reduction 処理



- 内積: pairwise にベクトル要素かけて、1 つの `sum` へ集約
- 縮約計算中の独立に計算できる部分を抽出し、並列に計算

- atomic 演算でも書けるが、他の thread の処理終了待つため性能が出ないことも

Warp shuffle や Shared memory を駆使しながら HW 構成に合わせて書くと良い

CUDA Managed Memory と同期

```
float *a, *b, *c;
cudaMallocManaged(&a, bytes);
cudaMallocManaged(&b, bytes);
cudaMallocManaged(&c, bytes);

add_kernel<<<grid_size, block_size>>>(a, b, c, n);

cudaDeviceSynchronize();
printf("%e\n", c[0]);
cudaFree(a); cudaFree(b); cudaFree(c);
```

- `cudaMallocManaged`: CPU コードと GPU コードの両方からアクセスできるメモリを確保
- ただしデータ移動が消えるわけではない。必要に応じて勝手にデータが転送される
- Kernel Launch は CPU に対して非同期。CPU で結果を読む前に同期が必要

CUDA エラー確認: よく使うマクロ

```
#define CUDA_CHECK(call) do {                          \
  cudaError_t err = (call);                            \
  if (err != cudaSuccess) {                            \
    printf("CUDA error %s:%d: %s\n", __FILE__, __LINE__, \
           cudaGetErrorString(err));                   \
    exit(EXIT_FAILURE);                                \
  }                                                    \
} while (0)

CUDA_CHECK(cudaMalloc(&da, bytes));
add_kernel<<<blocks, threads>>>(da, db, dc, n);
CUDA_CHECK(cudaGetLastError());
CUDA_CHECK(cudaDeviceSynchronize());
```

- CUDA Runtime API の多くは `cudaError_t` を返す
- API 呼び出しは、その場で戻り値を確認
- 初期のデバッグでは「失敗した場所」と「エラー文字列」がすぐ見える形にする

まとめ: CUDA のアーキテクチャ

メモリ階層

- Register、Shared Memory、Device Memory など、複数のメモリが用意されており、それぞれ Thread からアクセス可能な範囲が異なる
- Register は Thread Private
- Shared Memory は Thread Block で共有
- Device Memory はみんなアクセス可能

Thread 階層

- Grid 内に複数の Thread Block があり、Thread Block 内に複数の Thread がある
- 同じ Thread Block 内の Thread は Shared Memory を共有

HW との対応

- Thread Block は 1 つの SM に割り当てられる
- 1 Warp (32 Threads) 単位で命令が実行される

まとめ: CUDA カーネル

コードで明示するもの

- `__global__`: GPU で動く関数
- `<<<blocks, threads>>>`: 起動する Thread 数
- `blockIdx` / `threadIdx`: Kernel 内での Thread 割り当て

CUDA C++では、Thread 割り当て、Memory 管理・コピー、同期まで含めて「1 つの GPU 処理」として設計する

最適化キーワード: Coalescing access, Bank conflict, Warp divergence, Register spill, etc.

NVIDIA CUDA Programming Guide: https://docs.nvidia.com/cuda/cuda-programming-guide/index.html

Contents

まとめ：GPU プログラミングにはいろいろな方法がある

いろいろな方法

- NumPy 風の配列コード: CuPy
- 定番の処理: GPU ライブラリ
- C++ Standard algorithms で表現可能: stdpar
- 既存 C/C++ loop: OpenACC、OpenMP offload
- Thread 配置や Memory を制御したい処理: CUDA C++

どの方法でも共通

- GPU に十分な量の並列実行可能な仕事を渡す
- CPU–GPU 転送を減らす

Contents

補遺. GPU コードの最適化

補遺: GPU コードの最適化

ここで見ること

- 正しさ確認と時間測定
- GPU 処理の非同期性（asynchronous execution）
- Nsight Systems と Nsight Compute の使い分け

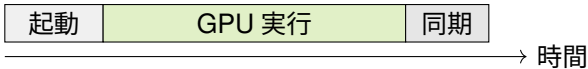
ここで押さえること

- 最適化は測定してから行う
- まずアプリケーション全体の時間内訳を見る
- 重要なカーネルだけを詳しく解析
- 1 つ変更し、再測定して比較

重要: 計算結果が正しいなら、つべこべ言わずに時間を測れ profile をとれ。推測するな、いいから測れ

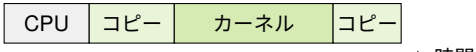
時間測定: GPU 処理は非同期

```
auto t0 = now();
kernel<<<blocks, threads>>>(...);
cudaDeviceSynchronize();  // wait for GPU
auto t1 = now();
```



- Kernel Launch は、GPU 処理の完了を待たずに CPU へ戻ることがある
- CPU タイマで測るなら明示的な待機が必要
- 手動計測だけでなく、プロファイラで確認

最初のプロファイリングの問い




```
$ nsys profile ./my_program
```

- CPU 処理が支配的: GPU カーネルを直しても効果は限定的
- コピーが支配的: 転送を減らす、または GPU 上にデータを残す
- 1 つのカーネルが支配的: そのカーネルを詳しく見る
- Nsight Systems は、どこに時間がかかっているか分からないときに最初に使うプロファイラ

タイムラインの読み方

観察結果	最初の解釈
GPU 処理前の長い CPU 領域	前処理やセットアップが支配的な可能性
大きなコピー区間	データ移動がボトルネックの可能性
多数の短いカーネル	起動オーバーヘッドが大きい可能性
GPU 行に空白が多い	CPU が GPU へ十分に仕事を渡せていない可能性
1 つの長いカーネル	Nsight Compute で詳細解析

ポイント: タイムラインは修正方法を直接教えるものではなく、次に見る場所を絞るためのもの

Nsight Compute: カーネル単位の詳細

- Nsight Systems で重要なカーネルを特定した後に使う
- メモリスループット、occupancy、命令の内訳、ボトルネックのヒントを確認
- メモリアクセス、計算スループット、実行設定のどこで制限されているかを確認

コマンド: `ncu ./my_program`

流れ: 先にアプリ全体、次にカーネル詳細

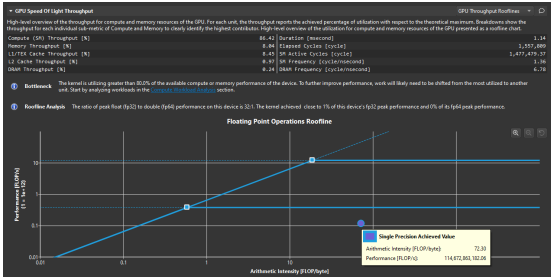
例：メモリアクセスのボトルネック

- GPU 性能は、隣接スレッドのメモリアクセスに敏感
- Coalesced access: 隣接スレッドが隣接アドレスを読み書き
- ランダムアクセス: 算術演算が単純でも帯域を浪費
- 式を変えるより、データレイアウトの変更が効くことがある

このたぐいの最適化は CUDA C++だけでなく、ライブラリ、CuPy、フレームワークの性能理解にも有効

Roofline の見方

- ルーフライン図: 達成性能とハードウェア上限を比較
- 低い arithmetic intensity: メモリ帯域制限の可能性
- 高い arithmetic intensity: 計算スループット制限の可能性



出典: NVIDIA Nsight Compute User Guide, ルーフライン図のセクション。

バンド幅ネックなら良いメモリアクセスやデータの再利用など、メモリアクセスに関する最適化をまず行う。演算ネックになったら演算側の最適化を行う

最適化の基本手順

1. 信頼できる結果と比較して正しさを確認
2. エンドツーエンド実行時間を測る
3. Nsight Systems でどこに時間がかかっているかを確認
4. 重要なカーネルまたは転送パターンを 1 つ選ぶ
5. 必要なら Nsight Compute で詳しく見る
6. 1 つだけ変更
7. 再測定して比較

チェックポイント: 最適化の考え方

- 速さより先に正しさ
- GPU 処理は非同期: 時間測定では完了を待つ
- Nsight Systems: アプリ全体のタイムライン
- Nsight Compute: カーネル単位の詳細解析
- 変更ごとに測定

参考

- Nsight Systems: https://docs.nvidia.com/nsight-systems/UserGuide/index.html
- Nsight Compute: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html