令和 6 年度 深層学習

二次元行列同士の積の計算と二次元行列の畳み込みにおける行列サイズと実行時間の関係と、python, numpy, pyTorch の比較

令和 7 年 4 月 24 日

大阪国際工科専門職大学
工科学部
情報工学科
AI 開発コース

OK240100 藤村勇仁

目次

1 概要

本課題では、2 つの正方行列の積と正方行列と 3×3 行列の畳み込みそれぞれを実行する python プログラムを作成して、それぞれの行列サイズと実行時間の関係を調べる。また、python, numpy, pytorch の 3 つのライブラリを用いて、行列の積と畳み込みを行い、実行時間を比較する。

2 行列の積

2 つの正方行列の積の計算のサイズと実行時間の関係を調べ、python, numpy, pyTorch の比較を行う。行列のサイズは 200 から 200 ずつ増加させ、python だけでの計算時間が 60 秒を超えたら 1000 ずつ増加させる。行列のサイズが大きくなるにつれて、実行時間がどのように変化するかを調べる。

2.1 プログラム

コード付録 A に、行列の積を計算する python プログラムを示す。

2.2 実行結果

コード 1 と、図 1 に、行列の積の計算結果を示す。

コード 1　付録付録 A の実行結果

```
Size: 200,  Python: 0.6059s, NumPy: 0.0020s,  Cupy: 0.0260s,  PyTorch: 0.0009s
Size: 400,  Python: 5.2042s, NumPy: 0.0025s,  Cupy: 0.0011s,  PyTorch: 0.0000s
Size: 600,  Python: 20.5605s, NumPy: 0.0039s,  Cupy: 0.0018s,  PyTorch: 0.0010s
Size: 800,  Python: 53.1748s, NumPy: 0.0060s,  Cupy: 0.0030s,  PyTorch: 0.0020s
Size: 1800,  Python: --------, NumPy: 0.0476s,  Cupy: 0.0179s,  PyTorch: 0.0180s
Size: 2800,  Python: --------, NumPy: 0.1761s,  Cupy: 0.0585s,  PyTorch: 0.0585s
Size: 3800,  Python: --------, NumPy: 0.3741s,  Cupy: 0.1394s,  PyTorch: 0.1396s
Size: 4800,  Python: --------, NumPy: 0.6643s,  Cupy: 0.2409s,  PyTorch: 0.2172s
Size: 5800,  Python: --------, NumPy: 1.2572s,  Cupy: 0.2860s,  PyTorch: 0.2317s
Size: 6800,  Python: --------, NumPy: 1.8989s,  Cupy: 0.3463s,  PyTorch: 0.3224s
Size: 7800,  Python: --------, NumPy: 3.0902s,  Cupy: 0.1120s,  PyTorch: 0.3983s
Size: 8800,  Python: --------, NumPy: 3.8734s,  Cupy: 0.1656s,  PyTorch: 0.1691s
Size: 9800,  Python: --------, NumPy: 5.4696s,  Cupy: 0.2353s,  PyTorch: 0.3565s
Size: 10800,  Python: --------, NumPy: 12.1399s,  Cupy: 0.3184s,  PyTorch: 0.5472s
Size: 11800,  Python: --------, NumPy: 9.9611s,  Cupy: 0.3828s,  PyTorch: OOM
```

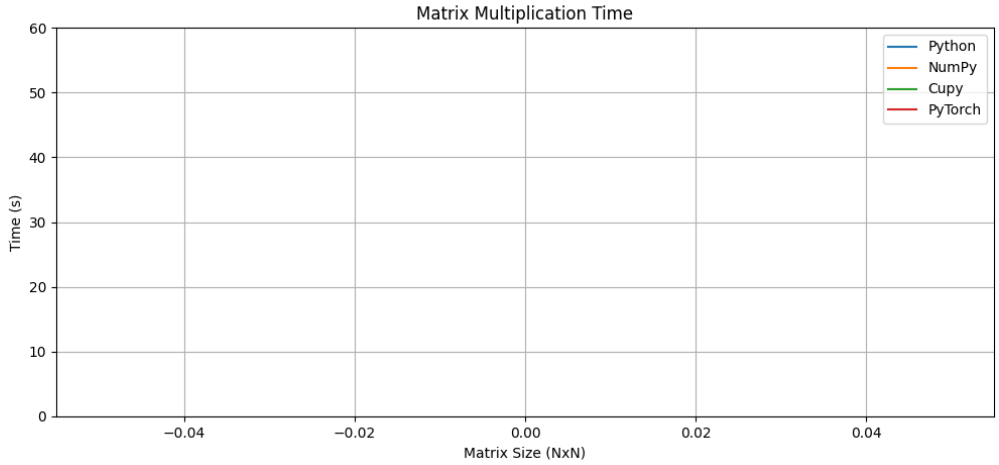

図 1　付録付録 A の実行結果

2.3　考察

実行結果より、Python、NumPy、CuPy、PyTorch のいずれの方法も、理論上は $O(n^3)$ に近い計算量であることがわかる。しかし、実際の計算時間としては Python > NumPy > CuPy ≃ PyTorch となっており、これは NumPy や CuPy、PyTorch が内部的に BLAS（Basic Linear Algebra Subprograms）ライブラリを使用しており、高速化されているためである。

BLAS とは、線形代数の基本的な演算（行列積、ベクトル演算など）を高速に実行するための標準ライブラリで、CPU のキャッシュやパイプラインの効率的な活用、マルチスレッドによる並列化といった最適化が行われている。

実際に、NumPy で使用されている BLAS 実装を確認するために `numpy.__config__.show()` を実行すると、以下のような出力が得られる：

```
"blas": {
"name": "scipy-openblas",
"found": true,
"version": "0.3.27",
"detection method": "pkgconfig",
"include directory": "C:/Users/runneradmin/AppData/Local/Temp/cibw-run-4yyn90e0/cp312-win_amd64/build/venv/Lib/site-packages/scipy_openblas64/include",
"lib directory": "C:/Users/runneradmin/AppData/Local/Temp/cibw-run-4yyn90e0/cp312-win_amd64/build/venv/Lib/site-packages/scipy_openblas64/lib",
"openblas configuration": "OpenBLAS 0.3.27 USE64BITINT DYNAMIC_ARCH NO_AFFINITY Haswell MAX_THREADS=24",
"pc file directory": "D:/a/numpy/numpy/.openblas"
},
```

このように、使用されている BLAS ライブラリが OpenBLAS であることが確認できる。

また、CuPy や PyTorch は、明示的に GPU を使用する設定をすることで、行列積などの処理を GPU 上で実行することができる。GPU 内部ではメモリ帯域幅が広く、大量の演算ユニットを活用した並列計算が可能であるため、CPU と比較して高速な処理が実現される。

その結果、同じアルゴリズム的には $O(n^3)$ の処理でも、ハードウェアとライブラリの違いにより実行時間には大きな差が生じる。

本実験では、PyTorch では OOM（Out Of Memory）エラーが発生した行列サイズでも、CuPy では正常に処理が行えた。これは、両者の GPU メモリ管理の違いに起因していると考えられる。具体的には、PyTorch

は自動微分機構や高速化のためのキャッシュを保持するため、メモリを多く使いやすいが、CuPy は中間データを持たず、必要な分だけメモリを確保・解放するため、効率的なメモリ利用ができる。このため、CuPy の方がより大きな行列サイズでも OOM になりにくいという結果になったと考える。

3 行列の畳み込み

正方行列と 3×3 行列の畳み込みの計算のサイズと実行時間の関係を調べ、python, numpy, pyTorch の比較を行う。行列のサイズは 1000 から 1000 ずつ増加させていく。行列のサイズが大きくなるにつれて、実行時間がどのように変化するかを調べる。

3.1 プログラム

コード付録 B に、行列の畳み込みを計算する python プログラムを示す。

3.2 実行結果

コード 2 と、図 2 に、行列の畳み込みの計算結果を示す。

コード 2　付録付録 B の実行結果

```
Size: 1000, Python: 0.9892s,  NumPy: 0.0250s, PyTorch: 0.2096s
Size: 2000, Python: 3.9049s,  NumPy: 0.0893s,  PyTorch: 0.0019s
Size: 3000, Python: 8.3927s,  NumPy: 0.1933s,  PyTorch: 0.0115s
Size: 4000, Python: 14.9831s,  NumPy: 0.3371s,  PyTorch: 0.0207s
Size: 5000, Python: 23.3619s,  NumPy: 0.5632s,  PyTorch: 0.0432s
Size: 6000, Python: 33.6339s,  NumPy: 0.7739s,  PyTorch: 0.0475s
Size: 7000, Python: --------,  NumPy: 1.0439s,  PyTorch: 0.0668s
Size: 8000, Python: --------,  NumPy: 1.3904s,  PyTorch: 0.0864s
Size: 9000, Python: --------,  NumPy: 1.7641s,  PyTorch: 0.0133s
Size: 10000, Python: --------,  NumPy: 2.6080s,  PyTorch: 0.0171s
Size: 11000, Python: --------,  NumPy: 7.7380s,  PyTorch: 0.0209s
Size: 12000, Python: --------,  NumPy: 9.2923s,  PyTorch: 0.0260s
Size: 13000, Python: --------,  NumPy: 11.3434s,  PyTorch: 0.0278s
Size: 14000, Python: --------,  NumPy: 6.0703s,  PyTorch: 0.0320s
Size: 15000, Python: --------,  NumPy: 14.5671s,  PyTorch: 0.0380s
Size: 16000, Python: --------,  NumPy: 15.5082s,  PyTorch: 0.0430s
Size: 17000, Python: --------,  NumPy: 6.8432s,  PyTorch: 0.0448s
Size: 18000, Python: --------,  NumPy: 7.9606s,  PyTorch: 0.0465s
Size: 19000, Python: --------,  NumPy: 8.6739s,  PyTorch: 0.0479s
Size: 20000, Python: --------,  NumPy: 9.1061s,  PyTorch: 0.0511s
```

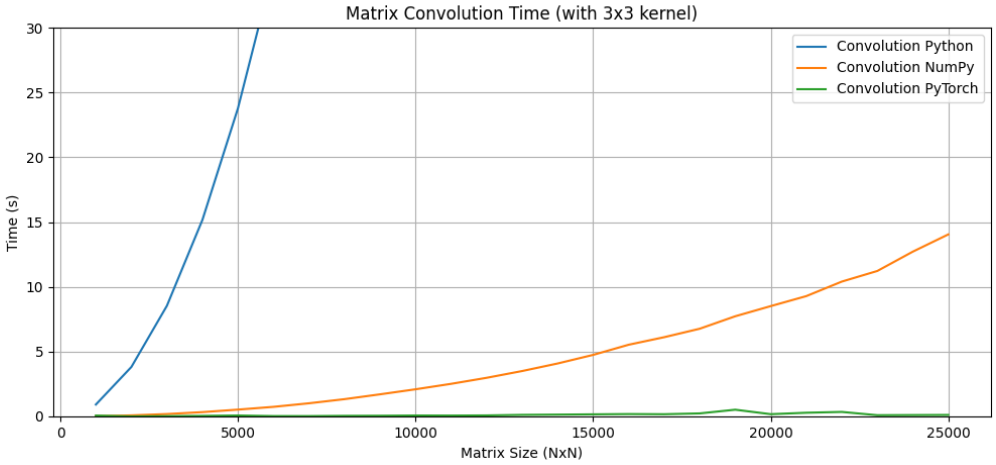

図 2　付録付録 B の実行結果

3.3　考察

実行結果より、純粋 Python（ネストした for 文）、NumPy+SciPy（‘scipy.signal.convolve2d‘）、CuPy（‘cupyx.scipy.ndimage.convolve‘）、PyTorch（‘torch.nn.functional.conv2d‘）のいずれも、理論上は $O(N^2k^2)$ の計算量を要するものの、実際の計測では大きく異なる挙動を示した。具体的には Python>NumPy/SciPy>CuPy ≃ PyTorch という順序で処理時間が短縮され、Python 実装ではループのオーバーヘッドや Python レイヤーでのメモリアクセスコストが大きいことが確認された。NumPy/SciPy は C 言語ベースのベクトル化処理により Python に比べて早いものの、あくまで CPU 上での実行であるため、並列化の限界やメモリ帯域幅の制約が性能を制限する要因となったと考える。一方、CuPy および PyTorch は GPU による並列計算を利用し、CPU ベースのライブラリに比べ桁違いに高速な畳み込みを実現した。

付録 A　2 次元行列同士の積を求めるプログラム

コード 3　二次元行列同士の積を求めるプログラム

```
import time
import numpy as np
import cupy as cp
import torch
import matplotlib.pyplot as plt
import gc


def generate_matrix_python(size):
    return [[(i + j) for j in range(size)] for i in range(size)]


def generate_matrix_np(size):
    return np.array(
        [[(i + j) for j in range(size)] for i in range(size)], dtype=np.float32
    )


def generate_matrix_cp(size):
    return cp.array(
        [[(i + j) for j in range(size)] for i in range(size)], dtype=cp.float32
    )


def generate_matrix_torch(size, device):
    return torch.tensor(
        [[(i + j) for j in range(size)] for i in range(size)],
        dtype=torch.float32,
        device=device,
    )


def matrix_multiplication_python(u, v):
    size = len(u)
    result = [[0.0] * size for _ in range(size)]
    for i in range(size):
        for j in range(size):
            for k in range(size):
                result[i][j] += u[i][k] * v[k][j]
    return result


def matrix_multiplication_numpy(u, v):
    return np.matmul(u, v)


def matrix_multiplication_cupy(u, v):
    cp.cuda.Stream.null.synchronize()
    result = cp.matmul(u, v)
    cp.cuda.Stream.null.synchronize()
    return result


def matrix_multiplication_torch(u, v):
    torch.cuda.synchronize()
    result = torch.matmul(u, v)
```

```
    torch.cuda.synchronize()
    return result


def benchmark():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    step = 200
    time_limit = 40

    python_times = []
    numpy_times = []
    cupy_times = []
    torch_times = []
    multiple_sizes = []

    i = 0
    size = 0
    flag_python = True
    flag_numpy = True
    flag_cupy = True
    flag_torch = True
    print("Starting benchmark...")
    with open("multiply-result.txt", "w") as f:
        while True:
            print(
                f"flag_python: {flag_python}, flag_numpy: {flag_numpy}, flag_cupy: {
                    flag_cupy}, flag_torch: {flag_torch}"
            )
            if not (flag_python or flag_numpy or flag_cupy or flag_torch):
                print("All methods exceeded the threshold.")
                break

            size += step if flag_python else step * 5
            print(f"Size: {size}")

            print(f"Size: {size}", end=", \t", file=f)

            # Python
            if flag_python:
                u_py = generate_matrix_python(size)
                v_py = generate_matrix_python(size)
                start = time.time()
                matrix_multiplication_python(u_py, v_py)
                python_times.append(time.time() - start)
                print(f"Python: {python_times[-1]:.4f}s", end=",\t", file=f)
                del u_py, v_py
                if python_times[-1] > time_limit:
                    flag_python = False
            else:
                python_times.append(python_times[-1])
                print("Python: --------", end=",\t", file=f)

            # NumPy
            if flag_numpy:
                u_np = generate_matrix_np(size)
                v_np = generate_matrix_np(size)
                start = time.time()
                matrix_multiplication_numpy(u_np, v_np)
                numpy_times.append(time.time() - start)
                print(f"NumPy: {numpy_times[-1]:.4f}s", end=", \t", file=f)
                del u_np, v_np
```

```
            if numpy_times[-1] > time_limit:
                flag_numpy = False
        else:
            numpy_times.append(numpy_times[-1])
            print("NumPy: --------", end=", \t", file=f)

        # CuPy
        if flag_cupy:
            u_cp = generate_matrix_cp(size)
            v_cp = generate_matrix_cp(size)
            start = time.time()
            matrix_multiplication_cupy(u_cp, v_cp)
            cupy_times.append(time.time() - start)
            print(f"Cupy: {cupy_times[-1]:.4f}s", end=", \t", file=f)
            del u_cp, v_cp
            if cupy_times[-1] > time_limit:
                flag_cupy = False
        else:
            cupy_times.append(cupy_times[-1])
            print("CuPy: --------", end=", \t", file=f)

        # PyTorch
        if flag_torch:
            try:
                u_th = generate_matrix_torch(size, device)
                v_th = generate_matrix_torch(size, device)
                start = time.time()
                matrix_multiplication_torch(u_th, v_th)
                torch_times.append(time.time() - start)
                print(f"PyTorch: {torch_times[-1]:.4f}s", file=f)
                if torch_times[-1] > time_limit:
                    flag_torch = False
            except Exception as e:
                if "out of memory" in str(e):
                    print("handling OOM error")
                    print("PyTorch: OOM", file=f)
                    u_th = None
                    v_th = None
                    torch_times.append(torch_times[-1])
                    flag_torch = False
                else:
                    raise e
            finally:
                if u_th is not None and v_th is not None:
                    del u_th, v_th
                torch.cuda.empty_cache()

        else:
            torch_times.append(torch_times[-1])
            print("PyTorch: --------", file=f)

        multiple_sizes.append(size)
        gc.collect()
        i += 1

    # === Plotting ===
    plt.figure(figsize=(12, 5))
    plt.plot(multiple_sizes, python_times[: len(multiple_sizes)], label="Python")
    plt.plot(multiple_sizes, numpy_times[: len(multiple_sizes)], label="NumPy")
    plt.plot(multiple_sizes, cupy_times[: len(multiple_sizes)], label="Cupy")
    plt.plot(multiple_sizes, torch_times[: len(multiple_sizes)], label="PyTorch")
```

```
    plt.title("Matrix Multiplication Time")
    plt.xlabel("Matrix Size (NxN)")
    plt.ylabel("Time (s)")
    plt.ylim(0, time_limit)
    plt.legend()
    plt.grid()
    plt.savefig("matrix_multiplication_time.png")
    plt.show()


if __name__ == "__main__":
    benchmark()
```

付録 B　2 次元行列の畳み込みを求めるプログラム

コード 4　二次元行列の畳み込みを求めるプログラム

```
import time
import numpy as np
import cupy as cp
import torch
import matplotlib.pyplot as plt
import gc
from scipy.signal import convolve2d
from cupyx.scipy.ndimage import convolve as cp_convolve


def generate_matrix(size):
    return [[(i + j) % 2 for j in range(size)] for i in range(size)]


def generate_matrix_np(size):
    return np.array(
        [[(i + j) % 2 for j in range(size)] for i in range(size)], dtype=np.float32
    )


def generate_matrix_cp(size):
    return cp.array(
        [[(i + j) % 2 for j in range(size)] for i in range(size)], dtype=cp.float32
    )


def generate_matrix_torch(size, device):
    return (
        torch.tensor(
            [[(i + j) % 2 for j in range(size)] for i in range(size)],
            dtype=torch.float32,
            device=device,
        )
        .unsqueeze(0)
        .unsqueeze(0)
    )


def convolution_python(img, kernel):
    h, w = len(img), len(img[0])
    kh, kw = len(kernel), len(kernel[0])
    output = [[0.0 for _ in range(w - kw + 1)] for _ in range(h - kh + 1)]

    for i in range(h - kh + 1):
        for j in range(w - kw + 1):
            for ki in range(kh):
                for kj in range(kw):
                    output[i][j] += img[i + ki][j + kj] * kernel[ki][kj]
    return output


def convolution_numpy(img, kernel):
    return convolve2d(img, kernel, mode="valid")


def convolution_cupy(img, kernel):
```

```
    cp.cuda.Stream.null.synchronize()
    result = cp_convolve(img, kernel, mode="constant", cval=0.0)
    cp.cuda.Stream.null.synchronize()
    return result


def convolution_torch(img, kernel):
    result = torch.nn.functional.conv2d(img, kernel, padding=0)
    torch.cuda.synchronize()
    return result


def benchmark():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    step = 1000
    time_limit = 30
    max_size = 25000

    kernel_py = [[1 / 9.0] * 3 for _ in range(3)]
    kernel_np = np.ones((3, 3)) / 9.0
    kernel_cp = cp.ones((3, 3), dtype=cp.float32) / 9.0
    kernel_torch = torch.tensor(kernel_np).unsqueeze(0).unsqueeze(0).to(device).float()

    python_times = []
    numpy_times = []
    cupy_times = []
    torch_times = []
    conv_sizes = []

    size = 0
    flag_python = True
    flag_numpy = True
    flag_cupy = True
    flag_torch = True

    print("Starting convolution benchmark...")

    with open("conv-result.txt", "w") as f:
        while True:
            if not (flag_python or flag_numpy or flag_cupy or flag_torch):
                print("All methods exceeded the threshold.", file=f)
                break
            if size >= max_size:
                print("Maximum size reached.", file=f)
                break

            size += step
            conv_sizes.append(size)
            print(f"Size: {size}", end=",\t", file=f)

            # Python
            if flag_python:
                u_py = generate_matrix(size)
                start = time.time()
                convolution_python(u_py, kernel_py)
                python_times.append(time.time() - start)
                print(f"Python: {python_times[-1]:.4f}s", end=", \t", file=f)
                del u_py
                if python_times[-1] > time_limit:
                    flag_python = False
            else:
```

```
            python_times.append(python_times[-1])
            print("Python: --------", end=", \t", file=f)

        # NumPy
        if flag_numpy:
            u_np = generate_matrix_np(size)
            start = time.time()
            convolution_numpy(u_np, kernel_np)
            numpy_times.append(time.time() - start)
            print(f"NumPy: {numpy_times[-1]:.4f}s", end=", \t", file=f)
            del u_np
            if numpy_times[-1] > time_limit:
                flag_numpy = False
        else:
            numpy_times.append(numpy_times[-1])
            print("NumPy: --------", end=", \t", file=f)

        # CuPy
        if flag_cupy:
            try:
                u_cp = generate_matrix_cp(size)
                start = time.time()
                convolution_cupy(u_cp, kernel_cp)
                cupy_times.append(time.time() - start)
                print(f"CuPy: {cupy_times[-1]:.4f}s", end=", \t", file=f)
                del u_cp
                if cupy_times[-1] > time_limit:
                    flag_cupy = False
            except Exception as e:
                print("CuPy: Error", file=f)
                cupy_times.append(cupy_times[-1] if cupy_times else 0)
                flag_cupy = False
        else:
            cupy_times.append(cupy_times[-1])
            print("CuPy: --------", end=", \t", file=f)

        # PyTorch
        if flag_torch:
            try:
                u_th = generate_matrix_torch(size, device)
                start = time.time()
                convolution_torch(u_th, kernel_torch)
                torch_times.append(time.time() - start)
                print(f"PyTorch: {torch_times[-1]:.4f}s", file=f)
                if torch_times[-1] > time_limit:
                    flag_torch = False
            except Exception as e:
                print("PyTorch: Error", file=f)
                torch_times.append(torch_times[-1] if torch_times else 0)
                flag_torch = False
            finally:
                if "u_th" in locals():
                    del u_th
                torch.cuda.empty_cache()
        else:
            torch_times.append(torch_times[-1])
            print("PyTorch: --------", file=f)

        gc.collect()

    # === Plotting ===
```

```
    plt.figure(figsize=(12, 5))
    plt.plot(conv_sizes, python_times[: len(conv_sizes)], label="Python")
    plt.plot(conv_sizes, numpy_times[: len(conv_sizes)], label="NumPy")
    plt.plot(conv_sizes, cupy_times[: len(conv_sizes)], label="CuPy")
    plt.plot(conv_sizes, torch_times[: len(conv_sizes)], label="PyTorch")
    plt.title("Matrix Convolution Time (3x3 kernel)")
    plt.xlabel("Matrix Size (NxN)")
    plt.ylabel("Time (s)")
    plt.ylim(0, time_limit)
    plt.legend()
    plt.grid()
    plt.savefig("matrix_convolution_time.png")
    plt.show()


if __name__ == "__main__":
    benchmark()
```