Problem 1: Determinants can be floppy

Problem Statement

Consider a non-singular matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, given by

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \tag{1}$$

Let us compute its determinant using the following formula:

$$\det(\mathbf{A}) \coloneqq \sum_{k=1}^{n} (-1)^k a_{1k} \det(\widehat{\mathbf{A}}^{1k})$$

where $\widehat{\mathbf{A}}^{1k} \in \mathbb{R}^{(n-1)\times(n-1)}$ is the matrix obtained by removing the first row and k-th column of $\mathbf{A}$.

a

Suppose for $\mathbf{A} \in \mathbb{R}^{n \times n}$, computing $\det(\mathbf{A})$ using the above formula takes y_n flops. Find the relationship between y_n and y_{n-1}. Assume that negating a number does not require a flop.

b

Use your recurrence relation in (a) to prove that $n! \leq y_n \leq \frac{1}{2}(n+1)!$ for each $n \geq 5$. Here, $z! = z(z-1)(z-2)\dots 1$ denotes the factorial of z.

c

Estimate how much time your computer takes for one flop. Please submit your code.

d

1. Using your answer in (c), estimate how long it will take to compute $\det(\mathbf{A})$ for the provided equation, using $n = 100$. Answer in number of years! The following Sterling's approximation may be useful:
$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$
2. Based on your asnwer in (i), will you use this method to compute the determinant 100×100 matrix? By the way, the age of the universe is 13.8 billion years.

Solution

a

Let $F(n) \coloneqq y_n$, i.e. the FLOP count of this calculation as a function of n. We can see that, for a given sub-term in the summation we have:

$$(-1)^k a_{1k} \det(\widehat{\mathbf{A}}^{1k})$$

The sub-matrix determinant will take $F(n-1)$ FLOPs. The negation will take none. The multiplication of a_{1k} and the sub-determinant will take one. Each sub-term will thus take $F(n-1)+1$ FLOPS, and there are n sub-terms. Then, the summations will require $n-1$ additional flops. So, we find:

$$F(n) = n(F(n-1)+1) + (n-1) = nF(n-1) + 2n - 1$$

Note that $F(0) = F(1) = 0$, so:

n	1	2	3	4	5	6	7	8	9
$F(n)$	0	3	14	63	324	1955	13698	109599	986408

b

Based on the prior identity, we can note that:

$$F(n) \geq nF(n-1)$$

Additionally, we know $F(2) = 3$. So:

$$F(3) \geq 3F(2) = 3 * 3 = 3!\frac{3}{2}$$

$$\implies F(n) \geq n!\frac{3}{2}$$

Furthermore,

$$n!\frac{3}{2} \geq n!$$

$$\implies \frac{3}{2} \geq 1 \checkmark$$

$$\therefore F(n) \geq n!$$

which establishes a lower bound. □

For an upper bound, consider $F(5) = 324 \geq \frac{5!}{2} = 360$. Use induction and assume:

$$F(n-1) \leq \frac{1}{2}((n-1)+1)! \forall n \geq 5$$

$$F(n) = 2n - 1 + nF(n-1)$$

$$F(n) \leq 2n - 1 + n(\frac{n!}{2})$$

Additionally:

$$2n - 1 + \frac{n}{2}n! \leq \frac{1}{2}(n+1)!$$

$$\implies 2n - 1 \leq \frac{1}{2}n! \forall n \geq 4$$

$$\therefore F(n) \leq \frac{1}{2}(n+1)!$$

□

Credit to Will for pointing me in the right direction for the upper bound proof.

c

My script output (see the "Code" section at the end) produces the following output:

```
Python list, naive for loop: 5.54E+07 FLOPS, or 1.81E-08 seconds for 1 FLOP
Time for 100x100 Matrix: 5.337560113947608e+142 years
NumPy array, naive for loop: 5.47E+07 FLOPS, or 1.83E-08 seconds for 1 FLOP
Time for 100x100 Matrix: 5.405708865911174e+142 years
NumPy array, vectorized sum: 2.82E+09 FLOPS, or 3.54E-10 seconds for 1 FLOP
Time for 100x100 Matrix: 1.046546214997435e+141 years
Torch Tensor (CPU), vectorized sum: 3.55E+09 FLOPS, or 2.81E-10 seconds for 1 FLOP
Time for 100x100 Matrix: 8.314569960344053e+140 years
Torch Tensor (GPU), vectorized sum: 4.16E+11 FLOPS, or 2.40E-12 seconds for 1 FLOP
Time for 100x100 Matrix: 7.102457520465304e+138 years
```

I'm running an AMD Ryzen 7 3700X 8 Core CPU (I think I have a mild overclock?) as well as an nVidia RTX 2070 Super. Nearly half a TeraFLOP in one card :D

d

1. See the above subsection code listing for script output (which does this calculation). Best case we're looking at about $7 * 10^{138}$ years, which is blazingly fast compared to the worst case of about $5 * 10^{142}$ years.

2. Sure, time is just like, a construct man. Although if I wait a year the hardware might improve enough to knock the exponent down by one or two...In all seriousness, no, absolutely not.

Problem 2: It Schur is Nice

Problem Statement

Now, let's compute the determinant of $\mathbf{A} \in \mathbb{R}^{n \times n}$ differently. Write $\mathbf{A}$ as

$$\mathbf{A} = \begin{pmatrix} a_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \tag{2}$$

where $\mathbf{A}_{22} \in \mathbb{R}^{(n-1)\times(n-1)}$. Other blocks have appropriate dimensions. If $a_{11} \neq 0$, then

$$\det(\mathbf{A}) = a_{11} \det(\mathbf{A}_{22} - \mathbf{A}_{21} a_{11}^{-1} \mathbf{A}_{12})$$

where $\mathbf{A}_{22} - \mathbf{A}_{21} a_{11}^{-1} \mathbf{A}_{12}$ is non-singular. Suppose you can swap two rows without any FLOPs, but such a swap changes the sign of the determinant.

a

If computing $\det A$ using (2) takes z_n flops, find a recurrence relation between z_n and z_{n-1}.

b

Find the dominant term in n in the expression of z_n as a function of n; i.e. without a recurrence relation.

c

Will you use (2) to compute the determinant of a 100×100 matrix? Answer logically.

Solution

a

Applying the scalar divison to one of the matrices will require $n-1$ operations.The matrix-matrix multiplication is really the outer product of a $(n-1) \times 1$ and $1 \times (n-1)$ vector, requiring $(n-1) \times (n-1) = (n-1)^2$ operations. Finally, the subtraction of Matrices will require $(n-1)^2$ operations. The result will be $n(n-1)^2 + (n-1)^2 = (n+1)(n-1)^2$ to produce an argument to the next determinant, and the result must be multiplied once more.

$$F(n) = F(n-1) + (n+1)(n-1)^2 + 1$$

b

Ignoring the $F(n-1)$ term, the resulting expression becomes:

$$F(n) = (n+1)(n-1)^2 + 1 = n^3 - n^2 - n + 2$$

so the algorithm is $\mathcal{O}(n^3)$.

c

Sure, for a 100×100 matrix, $n = 100 \implies n^3 = 1,000,000$. Given that the bulk of the expense are matrix-matrix operations (which are indeed highly parallelizable), I'd feel comfortable quoting the "theoretical" maximum FLOPS performance of my GPU, an nVidia RTX 2070 Super at 283.2 GFLOPS for FP64. This matrix would require $\approx \frac{100^3}{283.2*10^9} = 3.53$ microseconds to calculate. As an aside, this theoretical number is **LOWER** than my measured FLOPs estimate, and likely has something to do with sustained throughput vs. a one-off calculation.

Of course, this algorithm isn't perfectly parallelizable, but even an order of magnitude slower would be 3.53 milliseconds...I'm probably not worried about that unless I'm finding the determinants of LOTS of these matrices.

Problem 3: A Plethora of Inversion Methods

Problem Statement

Suppose $\mathbf{A} \in \mathbb{R}^{n \times n}$ is nonsingular. Then, the inverse of $\mathbf{A}$ ca be computed elementwise as

$$\left[\mathbf{A}^{-1}\right]_{ij} = (-1)^{i+j} \frac{\det(\mathbf{B}_i j)}{\det \mathbf{A}}$$

where $\mathbf{B}_{ij}$ is the matrix obtained by removing $j-$th row and the i-th column of $\mathbf{A}$. Count the number of FLOPs (only dominant terms of n) required to compute the inverse of $\mathbf{A}$ using the above equation. For computing determinants, assume that you are using the scheme in problem 2. Compare it with the number of FLOPs you will take to compute an LU decomposition followed by forward and backward substitutions to compute $\mathbf{A}^{-1}$.

***Hint**: To compute $\mathbf{A}^{-1}$ using LU decompositions and forward/backward substitutions, you can calculate the columns $x_1, \ldots, x_n$ of $\mathbf{A}^{-1}$ by solving $\mathbf{A}x_i = \mathbf{e}_i$ where $\mathbf{e}_i \in \mathbb{R}^n$ is a vector whose elements are all zeros except its i-th element that is unity.*

Solution

The computational requirements, per problem 2, are $\mathcal{O}(n^3)$. As such, each numerator will require $\mathcal{O}((n-1)^3)$, the denominator will only need to be calculated once and will require $\mathcal{O}(n^3)$ operations, their division will require one flop per n, thus is $\mathcal{O}(n)$. These will need to be conducted for each element of the matrix, or n^2 times.

So it will be:

$$F(n) = n^2((n-1)^3 + n) + n^3 = n^5 - 3n^4 + 5n^3 - n^2 \approx \mathcal{O}(n^5)$$

By contrast, an LU decomposition takes around $\frac{2}{3}n^3$ operations[1], which would only need to happen once. The per-column forward/backward substitution then takes an additional $2n^2 - n$ FLOPs, which will need to be repeated n times (one per basis vector/column solution). The resulting cost is then approximately:

$$F(n) \approx \frac{2}{3}n^3 + n(2n^2 - n) = \frac{8}{3}n^3 - n^2 \approx \mathcal{O}(n^3)$$

which is far better than the prior $\mathcal{O}(n^5)$ complexity.

Code

```python
#!/usr/bin/env python3
import time
from typing import Callable, Iterable

import numpy as np
import torch

N_FACT = np.sqrt(2 * np.pi * 100) * (100 / np.e) ** 100


def naive_loop(x: Iterable) -> None:
    sum_ = 0.0
    for i in x:
        sum_ += i


def numpy_vector_op(x: np.array) -> None:
    sum_ = np.sum(x)


def torch_cpu_or_gpu(x: torch.Tensor) -> None:
    sum_ = torch.sum(x)


def estimate_flops(
    f: Callable[[int], float], n: int, lib: str = "numpy", iterations: int = 10
) -> float:
    times_ns = []
    if lib == "native":
        x = [1.0] * n
    elif lib == "numpy":
        x = np.ones(n, dtype=np.float64)
    elif lib == "torch_cpu":
        x = torch.ones(n, dtype=torch.float64)
    elif lib == "torch_gpu":
        x = torch.randn(n, dtype=torch.float64).to(torch.device("cuda:0"))
    else:
        raise RuntimeError(f"Invalid library {lib} requested!")

    for _ in range(iterations):
        torch.cuda.synchronize()
        t0 = time.perf_counter_ns()
        f(x)
        t1 = time.perf_counter_ns()
        times_ns.append(t1 - t0)

    avg_time_ns = np.average(np.array(times_ns))

    return n / (avg_time_ns / int(1e9))


def print_results(desc: str, flops: float) -> None:
    naivest_loop_flops = estimate_flops(naive_loop, int(n), "native")
    flop_time = 1.0 / flops
    print(f"{desc}: {flops:.2E} FLOPS, or {flop_time:.2E} seconds for 1 FLOP")
    print(f"Time for 100x100 Matrix: {(N_FACT/3.15576e+7)*flop_time} years")


if __name__ == "__main__":
    n = 1e6
    naivest_loop_flops = estimate_flops(naive_loop, int(n), "native")
    print_results("Python list, naive for loop", naivest_loop_flops)

    naive_loop_flops = estimate_flops(naive_loop, int(n), "native")
    print_results("NumPy array, naive for loop", naive_loop_flops)

    n = 1e8
    numpy_flops = estimate_flops(numpy_vector_op, int(n), "numpy")
    print_results("NumPy array, vectorized sum", numpy_flops)

    torch_cpu_flops = estimate_flops(torch_cpu_or_gpu, int(n), "torch_cpu")
    print_results("Torch Tensor (CPU), vectorized sum", torch_cpu_flops)

    torch_gpu_flops = estimate_flops(torch_cpu_or_gpu, int(n), "torch_gpu")
    print_results("Torch Tensor (GPU), vectorized sum", torch_gpu_flops)
```

Bibliography

[1] Oct. 2023. URL: https://en.wikipedia.org/wiki/LU_decomposition#Using_Gaussian_elimination.