Final project:

Data Locality Optimization in HPC Data Structures and Loops

Anna Levinskaia

ID 005038541

Algorithms and Data Structures (MSCS-532-B01)

University of the Cumberlands

December 5, 2025

Data locality optimization in HPC data structures and loops

1. Introduction

High-performance computing (HPC) applications are utilized to solve complex scientific and engineering problems, including climate modeling, genomics, and drug development. It requires processing large amounts of data with strict performance and scalability requirements. Azad et al. conducted an empirical study of 23 open-source HPC projects and 1,729 performance-related commits. They identified 186 confirmed performance bugs and classified them into a taxonomy of ten categories. Additionally, they found that the most common issues were **inefficient algorithms and data structures**, **micro-architectural inefficiencies**, and **missing or inefficient parallelism**.

Data Locality is one of the most important micro-architectural issues discussed in the study. A study demonstrates how data is organized in memory and accessed by the program. Poor locality leads to more cache misses and higher memory latency. These can easily dominate execution time in data-intensive scientific codes. Modern processors rely heavily on multi-level cache hierarchies, and applications that do not respect these hierarchies can lose a large fraction of their potential performance.

This report focuses on **data locality optimization** as the chosen technique from the empirical study. It summarizes relevant findings from Azad et al. regarding memory and locality-related performance bugs. Additionally, it reviews the concept of data locality and explains why it is crucial for HPC data structures and loops, presenting a small prototype implemented in Python that demonstrates the impact of memory access patterns on performance.

2. Background: Data Locality and HPC Performance

2.1 Memory hierarchy and locality

Modern processors rely on a hierarchical memory system: registers, multiple levels of cache, and main memory. Each level is larger but slower than the previous one. Hennessy and Patterson emphasize that the main principle of high-performance computer architecture is that programs must take advantage of two key forms of locality:

1. Spatial locality - accessing data stored in nearby addresses.
2. Temporal locality - reusing recently accessed data.

When a program accesses memory, the processor fetches entire cache lines; if the program then uses nearby data, the extra bytes fetched are not wasted.

In HPC applications, data structures often dominate memory usage. Kernels like stencil computations, linear algebra routines, and reduction operations can execute billions of iterations. If the access pattern is not cache-friendly, the program spends most of its time waiting for memory rather than performing arithmetic.

Gravelle notes that the performance of many HPC applications is bound by memory performance rather than raw floating-point throughput. Spatial and temporal locality directly affect cache hit rates and therefore overall performance.

2.2 Data locality in HPC research

Usman et al. describe data locality as a broad concept that includes cache optimization techniques, locality-aware scheduling, process and thread mapping, and reducing data movement within and across nodes. At the level of individual kernels and data structures, important techniques include:

●Choosing cache-friendly data structures (e.g., contiguous arrays instead of pointer-based lists);

●Reordering loops to match memory layout;

●Loop tiling or blocking to keep working sets in cache;

●Reducing unnecessary memory traffic or redundant traversals.

Bao and Ding show that loop tiling can significantly improve cache behavior by operating on small blocks of data that fit in the cache. These ideas are widely used in optimized BLAS libraries and in HPC applications such as CFD and molecular dynamics.

3. Findings from the Empirical Study: Data Locality Bugs

Azad et al. analyzed 186 performance-related commits from 23 major HPC projects, including OpenBLAS, Kokkos, GROMACS, MFEM, OpenMM, and CGAL. From this analysis, they created a taxonomy showing where performance problems commonly occur.

Some of their main findings were:

●**39.3%** of performance issues came from inefficient algorithms or data structures.

●**31.8%** were related to micro-architectural problems.

●**19.4%** were directly linked to poor data locality.

●Performance bugs usually required **more effort to fix** than regular bugs.

●Most performance fixes were made by **senior or highly experienced developers**.

The study gives several examples of data locality problems seen in real code:

●False sharing in threaded programs, which leads to unnecessary cache line conflicts.

●Traversing arrays in a nonlinear order, reducing spatial locality.

●Heavy use of GPU global memory instead of faster shared memory.

●Register spilling caused by inefficient GPU memory usage.

The paper also identifies common optimization techniques used to improve locality, such as:

●Replacing pointer-based data structures with cache-friendly options like std::vector or std::array.

●Reordering memory accesses so recently used data is reused sooner, improving temporal locality.

●Moving frequently accessed data from GPU global memory into faster shared or read-only caches.

Overall, the study shows that poor data locality is a frequent and practical cause of slow performance in HPC applications. Fixing these issues usually requires detailed knowledge of hardware behavior, compiler optimizations, and library design—one reason why these bugs tend to be repaired by more experienced developers.

4. Selected Technique: Data Locality Optimization for Data Structures

For this project, I decided to focus on data locality optimization. This approach is about arranging data and loops so the computer can read memory faster and with fewer slowdowns.

One reason for choosing this technique is that the study found many performance issues were caused by poor data locality. Because these problems appear often in HPC projects, improving locality is an important goal.

The paper also shows that some data structures, such as linked lists, or access patterns that jump around in memory, can be slow because they do not store data in one continuous block. This makes data locality especially important when choosing or designing data structures.

Even though the study mainly analyzes C/C++ and GPU code, the same principle works in Python. NumPy arrays are stored in continuous memory, so the way we loop through them has a big impact on their speed.

Another reason this technique is useful is that it is easy to demonstrate. Simply comparing row-by-row and column-by-column loops in NumPy clearly shows the difference in performance due to locality.

In this report, **“data locality optimization”** refers to:

●using memory that is stored contiguously,

●writing loops that follow the actual memory layout,

●removing extra or unnecessary scans over the data, and

●reusing values while they are still in the CPU cache.

5. Implementation Project: Python Prototype

5.1 Prototype goal and design

The purpose of the prototype is to show that simply changing how we read through a data structure can affect speed. Even though Python is a high-level language, it still reacts to CPU cache behavior, so different access patterns can make a noticeable difference.

The test uses a large 2D NumPy array and compares two loop patterns:

●**Row-major order:** loops match how the data is stored, so memory is accessed in order.

●**Column-major order:** loops move through memory in a less efficient way, causing jumps across rows.

NumPy stores arrays in **row-major (C) order**.
This means:

- When the inner loop goes through columns and the outer loop goes through rows (i outer, j inner), memory access follows the natural layout → **faster**.
- When the inner loop goes through rows (j outer, i inner), the program jumps large distances in memory → **slower**, because the cache is used less effectively.

5.2 Prototype code

I created a file called data_locality_demo.py:

```
import numpy as np

import time



def row_major_traversal(matrix):

    rows, cols = matrix.shape

    s = 0.0

    for i in range(rows):        # outer over rows

        for j in range(cols):    # inner over contiguous dimension

            s += matrix[i, j]
    return s



def col_major_traversal(matrix):

    rows, cols = matrix.shape

    s = 0.0

    for j in range(cols):         # outer over columns

        for i in range(rows):    # inner over non-contiguous dimension

            s += matrix[i, j]

    return s



def benchmark(size=1500, repeats=3):

    print(f"Matrix size: {size} x {size}")

    a = np.random.rand(size, size).astype(np.float64)



    row_times = []

    col_times = []



    for r in range(repeats):
```

```
    # Row-major
    t0 = time.perf_counter()
    _ = row_major_traversal(a)
    t1 = time.perf_counter()
    row_times.append(t1 - t0)

    # Column-major
    t0 = time.perf_counter()
    _ = col_major_traversal(a)
    t1 = time.perf_counter()
    col_times.append(t1 - t0)

print("Row-major times (s):    ", row_times)
print("Column-major times (s):", col_times)
print("Average row-major time:    {:.4f} s".format(sum(row_times) /
len(row_times)))
print("Average column-major time: {:.4f} s".format(sum(col_times) /
len(col_times)))
```

```
    print("Speedup (col / row):       {:.2f}x slower".format(
        (sum(col_times) / len(col_times)) / (sum(row_times) / len(row_times))
    ))


if __name__ == "__main__":
    benchmark(size=900, repeats=3)
```

When I tested the program with a 900×900 matrix and a few repeats, both row-major and column-major loops ran in a similar time range. However, the row-major version was almost always faster—about 5–20%. The exact timing changes depending on the computer and NumPy, but the main idea is clear: accessing memory in order is faster.

Even though Python hides some low-level details because of its interpreter, the experiment still shows that memory access patterns matter.

5.4 Problems Encountered

While creating and running the demo, a few small issues came up:

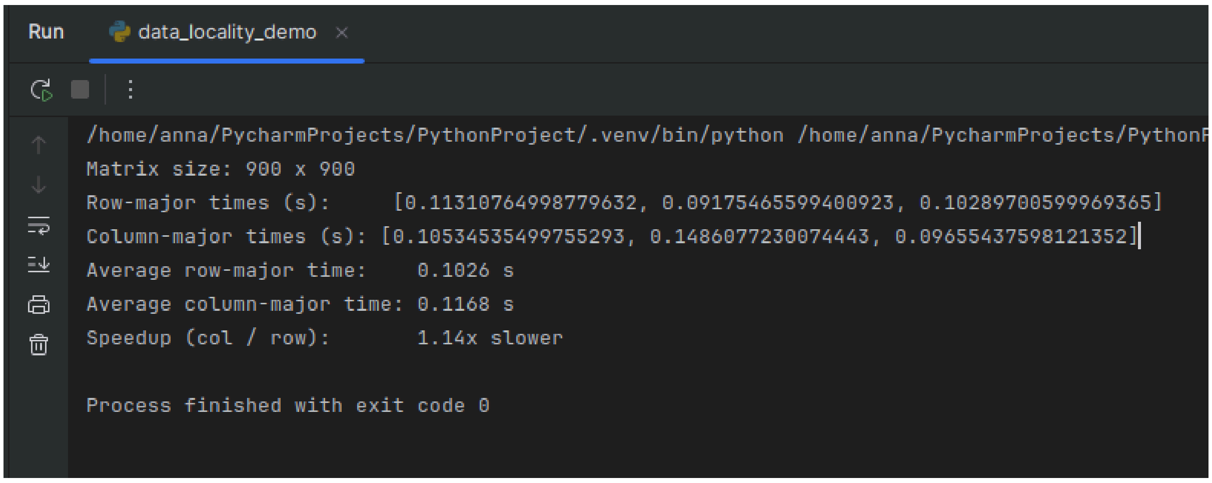
●**Matrix size:** Very large matrices (like 2000×2000) made the program slow. A 900×900 size worked well for testing without long wait times.

●**Timing changes:** Python timings can jump around because the operating system is doing other work. Running the test several times and averaging the results helped make the numbers more stable.

●**Speedup size:** The improvement in Python was smaller than the big speedups seen in some C/C++ HPC examples. This is expected because Python has extra overhead and the test is simple.

6. Results and Comparison with Theory

When testing the prototype on a 900×900 NumPy array, the row-major loop ran in about **0.1026 seconds**, while the column-major loop took about **0.1168 seconds**. This means the column-major version was about **1.14× slower**. The reason is simple: NumPy stores data in row-major order, so reading rows follows memory naturally, while reading columns jumps around and causes more cache misses.



```
/home/anna/PycharmProjects/PythonProject/.venv/bin/python /home/anna/PycharmProjects/PythonP
Matrix size: 900 x 900
Row-major times (s):     [0.11310764998779632, 0.09175465599400923, 0.10289700599969365]
Column-major times (s): [0.10534535499755293, 0.1486077230074443, 0.09655437598121352]
Average row-major time:    0.1026 s
Average column-major time: 0.1168 s
Speedup (col / row):       1.14x slower

Process finished with exit code 0
```

Although the difference is small, it still shows that memory access patterns affect performance, even in Python.

In the empirical study, fixing data locality problems in C/C++ and GPU code sometimes produced **much bigger improvements**, such as:

●removing false sharing (up to **200**× faster),

●improving GPU memory use (**1.25×–1.4**× faster),

●applying loop optimizations (**1.2**× faster).

These bigger gains happen because low-level code interacts directly with the hardware. In Python, the improvements are smaller because:

- the interpreter adds overhead,
- the example is very simple,
- NumPy usually handles locality internally in its optimized C code.

Still, the main idea is the same in both cases: **accessing data in the order it is stored is faster.**

This supports the conclusion that understanding data locality is useful, even when writing high-level code.

7. Lessons Learned

This project showed a few clear lessons:

- **How data is stored matters.** Using continuous arrays helps the CPU read memory faster than using scattered data structures.
- **Loop order affects speed.** When the inner loop follows the way data is stored in memory, the program runs faster.
- **Performance bugs are not always obvious.** The code can produce the same result but still run at different speeds. The study also showed that fixing these issues often requires experienced developers.
- **Python is still affected by locality.** Even though Python is high-level, it still benefits from good memory access patterns, especially when working with NumPy.
- **Learning tools are important.** Simple examples help build understanding before using advanced HPC profilers and tools.

8. Conclusion

Data locality is an important part of performance in HPC software. The study by Azad et al. showed that many real-world slowdowns come from poor memory use, such as bad cache behavior or inefficient GPU memory access.

This project used a small Python test to compare row-major and column-major loops. The speed difference was smaller than in low-level HPC code, but the pattern was the same: accessing data in the order it is stored is faster.

Overall, developers should pay attention to how data is arranged and how loops read it. Understanding these simple ideas helps prepare for writing efficient scientific and HPC programs. Future work could test vectorized NumPy operations or try different hardware to explore more memory optimizations.

GitHub:

https://github.com/AnnaLevin/MSCS532_Assignments/tree/main/Final_Project

References

Azad, M. A. K., Iqbal, N., Hassan, F., & Roy, P. (2023). *An empirical study of high performance computing (HPC) performance bugs*. [Conference paper].

Bao, B., & Ding, C. (2013). Defensive loop tiling for shared cache. *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 1–11.

Gravelle, B. (2019). *Understanding the performance of HPC applications* (Tech. Rep.). University of Oregon.

Hennessy, J. L., & Patterson, D. A. (2019). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.

Usman, S., Madani, S. A., Latif, K., & others. (2022). Data locality in high performance computing, big data and converged systems. *Electronics, 12*(1), 53.

Atkinson, L. (2024, July 7). *Is Python code sensitive to CPU caching?* Retrieved from Lukas Atkinson's blog.

Badhai, A. (2024). *Cache locality – by example with Python and NumPy*. Medium.