Loop Order Optimization in HPC

2D Histogram Creation Example

Dr A.Khalatyan
Researcher at eScience/Supercomputing/IT
Leibniz-Institut für Astrophysik Potsdam (AIP), Germany
AIP, 2025

Why Loop Order Matters

- Arrays in memory are not just random collections of values
- They have a specific layout that affects performance
- Understanding this layout is crucial for HPC applications

Memory Layout in C

C arrays are stored in **row-major** order:

```
memory: [00 01 02 ... 0N | 10 11 12 ... 1N | 20 21 ... ]
         ^-- Row 0 --^     ^-- Row 1 --^
```

Where `ij` represents the element at row `i` and column `j`.

Cache Behavior

When the CPU accesses memory:

- It doesn't fetch just one element
- It fetches entire **cache lines** (typically 64 bytes)
- This creates both opportunities and challenges

Performance Principles

1. **Spatial Locality**: Accessing elements close together in memory is faster
2. **Cache Efficiency**: Following the memory layout maximizes cache hits
3. **Prefetching**: CPUs can predict and load future data if access pattern is regular

Optimal Loop Order in C (Row-Major)

```c
// GOOD - follows memory layout
for (int i = 0; i < N; i++) {       // Row index
    for (int j = 0; j < N; j++) {   // Column index
        matrix[i][j] = value;
    }
}

// BAD - causes cache misses
for (int j = 0; j < N; j++) {       // Column index
    for (int i = 0; i < N; i++) {   // Row index
        matrix[i][j] = value;
    }
}
```

Performance Impact

The wrong loop order can be dramatically slower:

- Small matrices (N=256): 2-5× slower
- Large matrices (N=1024+): 10× or more slower
- Critical for HPC applications processing large datasets

Real-World Example: 2D Histogram

Creating a 2D histogram from (x,y) coordinates:

```c
// Suboptimal approach (column-major)
for (int j = 0; j < N; j++) {
    for (int i = 0; i < N; i++) {
        if (i == bin_x && j == bin_y) {
            histogram[i][j]++;
        }
    }
}
```

Real-World Example: 2D Histogram

Better approach (row-major):

```
// Optimal approach (row-major)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        if (i == bin_x && j == bin_y) {
            histogram[i][j]++;
        }
    }
}
```

Even Better: Direct Indexing

Most efficient approach:

```
// Direct indexing - most efficient
histogram[bin_x][bin_y]++;
```

Avoids unnecessary loops entirely!

Python & NumPy

NumPy also uses row-major order by default:

```python
# Good - follows memory layout
for i in range(N):
    for j in range(N):
        matrix[i, j] = value

# Bad - causes cache misses
for j in range(N):
    for i in range(N):
        matrix[i, j] = value
```

Using Built-in Functions (Python)

For histograms, NumPy provides highly optimized functions:

```
# Most efficient approach in Python
histogram, _, _ = np.histogram2d(
    x_coords, y_coords,
    bins=N,
    range=[[-1, 1], [-1, 1]]
)
```

Performance Optimization Best Practices

1. Always follow memory layout (row-major in C/Python)
2. Use direct indexing when possible
3. Leverage library functions (they're usually optimized)
4. Minimize calculations inside inner loops
5. Consider alternative data structures for sparse data

Common Mistakes

- Assuming loop order doesn't matter for small arrays
- Focusing on algorithm complexity while ignoring memory access patterns
- Optimizing prematurely without profiling
- Not considering the target architecture's cache behavior

Why This Matters in HPC

- Scientific simulations may run for days or weeks
- ML training can process billions of data points
- Physics simulations often work with large matrices
- Image processing deals with multi-dimensional arrays

Proper loop ordering can save hours or days of computation time!

Thank You!

Questions?