Cache-Friendly Programming in C

DS2040, Satyajit Das

March 6, 2025

Contents

1 Introduction

In modern CPUs, **cache memory** performance is a critical factor in overall program speed. This lab explores how different memory access patterns in C code affect **cache locality** (both **spatial** and **temporal**). Students will learn to:

- Measure and optimize memory access.
- Investigate row-major vs. column-major traversal.
- See temporal reuse in repeated passes.
- Compare naive vs. blocked matrix multiplication.
- Use **Valgrind** (`cachegrind`) to profile cache behavior.
- Finally, tackle a **matrix transpose** optimization problem.

2 Part 1: Spatial Locality Example (Row vs. Column Access)

Program: `spatial_locality.c`

```c
/* spatial_locality.c */
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define N 4000
static double A[N][N];

static inline double get_time_sec() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec * 1e-9;
}

int main() {
    // Initialize
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            A[i][j] = i + j;
        }
    }

    double start, end;
    double sum = 0.0;

    // Row-major traversal
    start = get_time_sec();
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            sum += A[i][j];
        }
    }
    end = get_time_sec();
    printf("Row-major time: %f s, sum=%f\n", end - start, sum);

    // Column-major traversal
    sum = 0.0;
    start = get_time_sec();
    for(int j = 0; j < N; j++) {
        for(int i = 0; i < N; i++) {
            sum += A[i][j];
        }
    }
    end = get_time_sec();
    printf("Column-major time: %f s, sum=%f\n", end - start, sum);

    return 0;
}
```

Compile and Run:

```
gcc -O2 -o spatial_locality spatial_locality.c -lrt
./spatial_locality
```

Analysis: - Expect `Row-major` to be faster because of contiguous memory accesses. - `Column-major` jumps in memory, leading to more cache misses.

3 Part 2: Temporal Locality Example (Repeated Access)

Program: `temporal_locality.c`

```
/* temporal_locality.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 1000000
int *arr;

static inline double get_time_sec() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec * 1e-9;
}

int main() {
    arr = (int *)malloc(SIZE * sizeof(int));
    if(!arr) {
        perror("malloc");
        return 1;
    }
    // Initialize
    for(int i = 0; i < SIZE; i++) {
        arr[i] = i;
    }

    double start, end;
    long long sum = 0;

    // Single pass
    start = get_time_sec();
    for(int i = 0; i < SIZE; i++) {
        sum += arr[i];
    }
    end = get_time_sec();
    printf("Single pass: %f s, sum=%lld\n", end - start, sum);

    // Repeated pass (10 times)
    sum = 0;
    start = get_time_sec();
    for(int repeat = 0; repeat < 10; repeat++) {
        for(int i = 0; i < SIZE; i++) {
            sum += arr[i];
        }
    }
    end = get_time_sec();
    printf("Repeated pass: %f s, sum=%lld\n", end - start, sum);
```

```
    free(arr);
    return 0;
}
```

Analysis: - The first pass loads the array into cache lines gradually. - Subsequent passes (if data remains in cache) might be faster, showing **temporal reuse**. - If `SIZE` exceeds your CPU's cache, repeated passes may or may not help as much.

4 Part 3: Mixed Example: Matrix Multiplication (Naive vs. Blocked)

Program: `matrix_multiply.c`

```
/* matrix_multiply.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 512
#define BLOCK 32

static double A[N][N], B[N][N], C[N][N];

static inline double get_time_sec() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec * 1e-9;
}

void matmul_naive() {
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            double sum = 0.0;
            for(int k=0; k<N; k++){
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}

void matmul_blocked() {
    for(int i=0; i<N; i+=BLOCK){
        for(int j=0; j<N; j+=BLOCK){
            for(int k=0; k<N; k+=BLOCK){
                for(int i2 = i; i2 < i + BLOCK; i2++){
                    for(int j2 = j; j2 < j + BLOCK; j2++){
                        double sum = C[i2][j2];
                        for(int k2 = k; k2 < k + BLOCK; k2++){
                            sum += A[i2][k2] * B[k2][j2];
                        }
                        C[i2][j2] = sum;
                    }
```

```
                }
            }
        }
    }
}

int main(){
    // Init A, B
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            A[i][j] = i + j;
            B[i][j] = i - j;
            C[i][j] = 0.0;
        }
    }

    double start, end;

    // Naive
    start = get_time_sec();
    matmul_naive();
    end = get_time_sec();
    printf("Naive multiplication time: %f s\n", end - start);

    // Re-init C
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            C[i][j] = 0.0;
        }
    }

    // Blocked
    start = get_time_sec();
    matmul_blocked();
    end = get_time_sec();
    printf("Blocked multiplication time: %f s\n", end - start);

    return 0;
}
```

Analysis: - The **naive** approach repeatedly scans entire rows & columns. - **Blocked** approach keeps sub-blocks in cache longer, exploiting both spatial & temporal locality. - Compare timings and see how changing `BLOCK` affects performance.

5 Part 4: Profiling with Valgrind

Valgrind's `cachegrind` tool can measure cache misses and CPU instruction usage.

Installation

```
sudo apt-get install valgrind  # Debian/Ubuntu
```

Basic Usage

```
valgrind --tool=cachegrind ./spatial_locality
```

This produces an output like:

```
==12345== Cachegrind, a cache and branch-prediction profiler
...
==12345== D   refs: ...
==12345== D1  misses: ...
==12345== LL refs: ...
==12345== LL misses: ...
==12345== Instruction fetches: ...
...
```

Key lines:

- `D1 misses` : data cache misses (L1)
- `LL misses`: last-level cache (LL) misses

Comparisons:

- Run `valgrind -tool=cachegrind` on `row-major` vs. `column-major` or `naive` vs. `blocked` matmul.
- Observe how the **miss counts** differ.

Example Command

```
valgrind --tool=cachegrind --cachegrind-out-file=naive.out ./
   matrix_multiply
valgrind --tool=cachegrind --cachegrind-out-file=blocked.out ./
   matrix_multiply --mode=blocked
# Then compare naive.out vs. blocked.out
```

You can also use `cg_annotate naive.out` to see function-by-function details.

6 Part 5: Student Assignment — Matrix Transpose Optimization

Problem Statement

Matrix transpose is a common operation in linear algebra: the rows of a matrix become columns, and vice versa. A naive implementation:

$$C[j, i] = A[i, j]$$

can suffer from **poor cache utilization**, especially for large matrices. Your task:

1. Write a **naive transpose** for an `N x N` matrix.
2. **Profile it** with Valgrind to measure cache misses.
3. Implement an **optimized transpose** using **blocking**, similar to the blocked matmul approach.

4. **Try different block sizes** (e.g., 8, 16, 32, 64) and measure performance with:
   - Real-time usage (e.g., `clock_gettime`).
   - Cache miss counts from Valgrind (`cachegrind`).
5. **Plot the results** (block size vs. time or block size vs. cache misses) to see which block size is optimal for your system.
6. **Write a short report** justifying your choice of optimal block size based on data from **Valgrind**.

Naive Transpose Skeleton Code

```
/* transpose.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 1024
static double A[N][N];
static double B[N][N];  // B will hold transpose of A

static inline double get_time_sec() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec * 1e-9;
}

// Naive transpose
void transpose_naive() {
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            B[j][i] = A[i][j];
        }
    }
}

// Blocked transpose - to be implemented by you
void transpose_blocked(int blockSize) {
    // e.g., for i in steps of blockSize
    //        for j in steps of blockSize
    //          transpose sub-block
    // hints: B[j2][i2] = A[i2][j2]
}

int main() {
    // init A
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            A[i][j] = (double)(i + j);
            B[i][j] = 0.0;
        }
    }
```

```
    double start, end;

    // naive
    start = get_time_sec();
    transpose_naive();
    end = get_time_sec();
    printf("Naive transpose time: %f s\n", end - start);

    // clear B
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            B[i][j] = 0.0;
        }
    }

    // blocked (TODO: vary blockSize = 8,16,32,...,64)
    int blockSize = 32;
    start = get_time_sec();
    transpose_blocked(blockSize);
    end = get_time_sec();
    printf("Blocked transpose (blockSize=%d) time: %f s\n", blockSize, end
        - start);

    return 0;
}
```

Assignment Steps

1. Implement `transpose_blocked` with nested loops over sub-blocks.
2. Use `valgrind -tool=cachegrind` on naive vs. blocked to see `D1 misses`, `LL misses`, etc.
3. Try multiple block sizes (8,16,32,64, maybe 128) and record:
    - Execution time.
    - Cache misses (Valgrind).
4. Create a **graph** of blockSize vs. time and blockSize vs. misses.
5. Determine **optimal block size** for your environment.
6. Justify your findings in a brief write-up.

7 Wrap-Up and Deliverables

What to Submit

- **Source codes** for each part (spatial locality, temporal locality, matrix multiply, and transpose).
- **Timing results** (tables or graphs).

- **Valgrind cachegrind outputs** (or summarized results) for naive vs. blocked approaches.
- **Short report** explaining the performance differences, referencing your measurements, and discussing how cache-friendly code can reduce misses.

End of Lab — Happy Cache-Friendly Coding!