Clean Excel Code

```
import pandas as pd
import numpy as np
import struct

# Load the dataset
df = pd.read_csv('earthquakes.csv')

print(f"Original dataset: {df.shape[0]} rows")

# Keep only the essential columns we need for 3D heatmap
essential_cols = ['time', 'latitude', 'longitude', 'mag']
df_clean = df[essential_cols].copy()

print(f"After keeping essential columns: {df_clean.shape[0]} rows")

# Remove any rows with missing values
df_clean = df_clean.dropna()
print(f"After removing NaN values: {df_clean.shape[0]} rows")

# Filter valid coordinates
df_clean = df_clean[
    (df_clean['latitude'] >= -90) & (df_clean['latitude'] <= 90) &
    (df_clean['longitude'] >= -180) & (df_clean['longitude'] <= 180) &
    (df_clean['mag'] > 0)
]
print(f"After coordinate and magnitude filtering: {df_clean.shape[0]} rows")

# Convert time to datetime and then to Unix timestamp
df_clean['time'] = pd.to_datetime(df_clean['time'], errors='coerce', utc=True)
df_clean = df_clean.dropna(subset=['time'])  # Remove any invalid timestamps

# Convert to Unix timestamp (seconds since 1970-01-01)
df_clean['timestamp'] = df_clean['time'].astype(np.int64) // 10**9

print(f"Time conversion completed")

# Display cleaned data info
print(f"\nFinal cleaned dataset: {df_clean.shape[0]} rows")
print(f"Date range: {df_clean['time'].min()} to {df_clean['time'].max()}")
print(f"Timestamp range: {df_clean['timestamp'].min()} to {df_clean['timestamp'].max()}")
print(f"Latitude range: {df_clean['latitude'].min():.2f} to {df_clean['latitude'].max():.2f}
print(f"Longitude range: {df_clean['longitude'].min():.2f} to {df_clean['longitude'].max():.
print(f"Magnitude range: {df_clean['mag'].min():.2f} to {df_clean['mag'].max():.2f}")

# Save as CSV
df_clean[['timestamp', 'latitude', 'longitude', 'mag']].to_csv('cleaned_earthquake_data.csv'
print("\nCleaned data saved to 'cleaned_earthquake_data.csv'")

# Save as binary file with proper padding to match C struct alignment (24 bytes per record)
with open('earthquake_data.bin', 'wb') as f:
    # Write number of records first
    f.write(struct.pack('i', len(df_clean)))
```

```
    # Write each record with padding: timestamp(8), lat(4), lon(4), mag(4), padding(4) = 24
    for _, row in df_clean.iterrows():
        # Pack with padding to align to 24 bytes (struct alignment in C)
        f.write(struct.pack('qfffI',
                            int(row['timestamp']),      # 8 bytes (long)
                            float(row['latitude']),     # 4 bytes (float)
                            float(row['longitude']),    # 4 bytes (float)
                            float(row['mag']),          # 4 bytes (float)
                            0))                         # 4 bytes padding

print(f"\nBinary data saved to 'earthquake_data.bin'")
print(f"Binary file size: {len(df_clean) * 24 + 4} bytes")

# Verify the binary file by reading back a few records
print("\nVerifying binary file (first 3 records):")
with open('earthquake_data.bin', 'rb') as f:
    n = struct.unpack('i', f.read(4))[0]
    print(f"Number of records in file: {n}")
    for i in range(min(3, n)):
        data = struct.unpack('qfffI', f.read(24))
        timestamp_date = pd.Timestamp(data[0], unit='s')
        print(f"Record {i}: timestamp={data[0]} ({timestamp_date}), lat={data[1]:.2f}, lon={
```

```
Original dataset: 78404 rows
After keeping essential columns: 78404 rows
After removing NaN values: 78404 rows
After coordinate and magnitude filtering: 78404 rows
Time conversion completed

Final cleaned dataset: 78404 rows
Date range: 2021-02-14 00:15:02.066000+00:00 to 2025-01-01 23:21:45.743000+00:00
Timestamp range: 1613261702 to 1735773705
Latitude range: -69.77 to 87.38
Longitude range: -180.00 to 180.00
Magnitude range: 3.00 to 8.20

Cleaned data saved to 'cleaned_earthquake_data.csv'

Binary data saved to 'earthquake_data.bin'
Binary file size: 1881700 bytes

Verifying binary file (first 3 records):
Number of records in file: 78404
Record 0: timestamp=1735773705 (2025-01-01 23:21:45), lat=-20.02, lon=-177.56, mag=4.20
Record 1: timestamp=1735772841 (2025-01-01 23:07:21), lat=-2.16, lon=100.10, mag=4.40
Record 2: timestamp=1735772144 (2025-01-01 22:55:44), lat=-16.93, lon=167.30, mag=4.50
```

The Header File

```
header_code = """#ifndef EARTHQUAKE_ANALYSIS_H
#define EARTHQUAKE_ANALYSIS_H

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
#include <time.h>

// Grid resolution parameters
#define LAT_BINS 180    // 1 degree per bin
#define LON_BINS 360    // 1 degree per bin
#define TIME_BINS 120   // Adjustable based on date range

typedef struct {
    long timestamp;
    float latitude;
    float longitude;
    float magnitude;
} EarthquakeEvent;

typedef struct {
    int count;
    float total_magnitude;
    float max_magnitude;
} GridCell;

// Function declarations
int load_binary_data(const char *filename, EarthquakeEvent **events);
void compute_bin_indices(EarthquakeEvent event, long min_time, long max_time,
                         int *lat_idx, int *lon_idx, int *time_idx);
GridCell*** allocate_histogram();
void free_histogram(GridCell ***histogram);
double get_time_ms();

#endif
"""

with open('earthquake_analysis.h', 'w') as f:
    f.write(header_code)

print("Header updated!")
```

```
Header updated!
```

Sequential Code

```
sequential_code = """#include "earthquake_analysis.h"
#include <sys/time.h>
#include <string.h>

double get_time_ms() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec * 1000.0) + (tv.tv_usec / 1000.0);
}

int load_binary_data(const char *filename, EarthquakeEvent **events) {
    FILE *fp = fopen(filename, "rb");
    if (!fp) {
        perror("Error opening file");
        return -1;
    }
```

```
    int n_events;
    fread(&n_events, sizeof(int), 1, fp);

    *events = (EarthquakeEvent *)malloc(n_events * sizeof(EarthquakeEvent));
    if (!*events) {
        fprintf(stderr, "Memory allocation failed\\n");
        fclose(fp);
        return -1;
    }

    for (int i = 0; i < n_events; i++) {
        fread(&(*events)[i], sizeof(EarthquakeEvent), 1, fp);
    }

    fclose(fp);
    return n_events;
}

void compute_bin_indices(EarthquakeEvent event, long min_time, long max_time,
                         int *lat_idx, int *lon_idx, int *time_idx) {
    // Latitude: -90 to 90 -> 0 to LAT_BINS-1
    *lat_idx = (int)((event.latitude + 90.0) * LAT_BINS / 180.0);
    if (*lat_idx >= LAT_BINS) *lat_idx = LAT_BINS - 1;
    if (*lat_idx < 0) *lat_idx = 0;

    // Longitude: -180 to 180 -> 0 to LON_BINS-1
    *lon_idx = (int)((event.longitude + 180.0) * LON_BINS / 360.0);
    if (*lon_idx >= LON_BINS) *lon_idx = LON_BINS - 1;
    if (*lon_idx < 0) *lon_idx = 0;

    // Time: min_time to max_time -> 0 to TIME_BINS-1
    *time_idx = (int)((event.timestamp - min_time) * TIME_BINS /
                      (double)(max_time - min_time + 1));
    if (*time_idx >= TIME_BINS) *time_idx = TIME_BINS - 1;
    if (*time_idx < 0) *time_idx = 0;
}

GridCell*** allocate_histogram() {
    GridCell ***histogram = (GridCell ***)malloc(LAT_BINS * sizeof(GridCell **));
    for (int i = 0; i < LAT_BINS; i++) {
        histogram[i] = (GridCell **)malloc(LON_BINS * sizeof(GridCell *));
        for (int j = 0; j < LON_BINS; j++) {
            histogram[i][j] = (GridCell *)calloc(TIME_BINS, sizeof(GridCell));
        }
    }
    return histogram;
}

void sequential_histogram(EarthquakeEvent *events, int n_events,
                          GridCell ***histogram, long min_time, long max_time) {
    for (int i = 0; i < n_events; i++) {
        int lat_idx, lon_idx, time_idx;
        compute_bin_indices(events[i], min_time, max_time,
                            &lat_idx, &lon_idx, &time_idx);

        GridCell *cell = &histogram[lat_idx][lon_idx][time_idx];
        cell->count++;
```

```
            cell->total_magnitude += events[i].magnitude;
            if (events[i].magnitude > cell->max_magnitude) {
                cell->max_magnitude = events[i].magnitude;
            }
        }
    }

    void save_histogram_csv(GridCell ***histogram, const char *filename) {
        FILE *fp = fopen(filename, "w");
        if (!fp) {
            perror("Error opening output file");
            return;
        }

        //write csv header
        fprintf(fp, "lat_bin,lon_bin,time_bin,count,total_magnitude,max_magnitude\\n");

        //non-empty cells only
        for (int i = 0; i < LAT_BINS; i++) {
            for (int j = 0; j < LON_BINS; j++) {
                for (int k = 0; k < TIME_BINS; k++) {
                    if (histogram[i][j][k].count > 0) {
                        fprintf(fp, "%d,%d,%d,%d,%.2f,%.2f\\n",
                               i, j, k,
                               histogram[i][j][k].count,
                               histogram[i][j][k].total_magnitude,
                               histogram[i][j][k].max_magnitude);
                    }
                }
            }
        }

        fclose(fp);
        printf("Histogram saved to %s\\n", filename);
    }

    void free_histogram(GridCell ***histogram) {
        for (int i = 0; i < LAT_BINS; i++) {
            for (int j = 0; j < LON_BINS; j++) {
                free(histogram[i][j]);
            }
            free(histogram[i]);
        }
        free(histogram);
    }

    int main(int argc, char *argv[]) {
        if (argc != 2) {
            fprintf(stderr, "Usage: %s <binary_data_file>\\n", argv[0]);
            return 1;
        }

        EarthquakeEvent *events;
        int n_events = load_binary_data(argv[1], &events);

        if (n_events <= 0) {
            fprintf(stderr, "Failed to load data\\n");
            return 1;
```

```
}

printf("=== Sequential Implementation ===\\n");
printf("Loaded %d earthquake events\\n", n_events);

//time range
long min_time = events[0].timestamp;
long max_time = events[0].timestamp;
for (int i = 1; i < n_events; i++) {
    if (events[i].timestamp < min_time) min_time = events[i].timestamp;
    if (events[i].timestamp > max_time) max_time = events[i].timestamp;
}

printf("Time range: %ld to %ld seconds\\n", min_time, max_time);
printf("Grid dimensions: %d x %d x %d = %d cells\\n",
       LAT_BINS, LON_BINS, TIME_BINS, LAT_BINS * LON_BINS * TIME_BINS);

//allocate histogram
GridCell ***histogram = allocate_histogram();

//run computation
double start = get_time_ms();
sequential_histogram(events, n_events, histogram, min_time, max_time);
double end = get_time_ms();

printf("Sequential execution time: %.2f ms\\n", end - start);


int non_empty = 0;
int max_count = 0;
float total_events = 0;

for (int i = 0; i < LAT_BINS; i++) {
    for (int j = 0; j < LON_BINS; j++) {
        for (int k = 0; k < TIME_BINS; k++) {
            if (histogram[i][j][k].count > 0) {
                non_empty++;
                total_events += histogram[i][j][k].count;
                if (histogram[i][j][k].count > max_count) {
                    max_count = histogram[i][j][k].count;
                }
            }
        }
    }
}

printf("Non-empty cells: %d out of %d (%.2f%%)\\n",
       non_empty, LAT_BINS * LON_BINS * TIME_BINS,
       100.0 * non_empty / (LAT_BINS * LON_BINS * TIME_BINS));
printf("Max events in single cell: %d\\n", max_count);
printf("Average events per non-empty cell: %.2f\\n", total_events / non_empty);

save_histogram_csv(histogram, "histogram_sequential.csv");

free_histogram(histogram);
free(events);

return 0;
```

```
}
"""

with open('sequential.c', 'w') as f:
    f.write(sequential_code)
```

```
!gcc -o sequential sequential.c -lm
!./sequential earthquake_data.bin
```

```
=== Sequential Implementation ===
Loaded 78404 earthquake events
Time range: 1613261702 to 1735773705 seconds
Grid dimensions: 180 x 360 x 120 = 7776000 cells
Sequential execution time: 4.08 ms
Non-empty cells: 45499 out of 7776000 (0.59%)
Max events in single cell: 467
Average events per non-empty cell: 1.72
Histogram saved to histogram_sequential.csv
```

PThreads Computation

```
pthreads= """#include "earthquake_analysis.h"
#include <sys/time.h>
#include <pthread.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {
    const int *idx;
    int idx_count;
    int thread_id;
    int lat_lo;
    int lat_hi;

    const EarthquakeEvent *events;
    const int *lat_bin;
    const int *lon_bin;
    const int *time_bin;

    GridCell ***global_histogram;
} ThreadData;

double get_time_ms(void) {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec * 1000.0) + (tv.tv_usec / 1000.0);
}

int load_binary_data(const char *filename, EarthquakeEvent **events) {
    FILE *fp = fopen(filename, "rb");
    if (!fp) { perror("fopen"); return -1; }

    int n_events = 0;
    if (fread(&n_events, sizeof(int), 1, fp) != 1) {
```

```
            perror("fread n_events");
            fclose(fp);
            return -1;
        }

        *events = (EarthquakeEvent *)malloc((size_t)n_events * sizeof(EarthquakeEvent));
        if (!*events) { perror("malloc events"); fclose(fp); return -1; }

        for (int i = 0; i < n_events; i++) {
            if (fread(&(*events)[i], sizeof(EarthquakeEvent), 1, fp) != 1) {
                perror("fread event");
                free(*events);
                fclose(fp);
                return -1;
            }
        }
        fclose(fp);
        return n_events;
    }

    static inline int lat_to_bin_clamped(double lat) {
        int v = (int)((lat + 90.0) * LAT_BINS / 180.0);
        if (v < 0) v = 0;
        else if (v >= LAT_BINS) v = LAT_BINS - 1;
        return v;
    }

    static inline int lon_to_bin_clamped(double lon) {
        int v = (int)((lon + 180.0) * LON_BINS / 360.0);
        if (v < 0) v = 0;
        else if (v >= LON_BINS) v = LON_BINS - 1;
        return v;
    }

    static inline int time_to_bin_clamped(long ts, long min_time, long max_time) {
        double denom = (double)(max_time - min_time + 1);
        int v = (int)((ts - min_time) * TIME_BINS / denom);
        if (v < 0) v = 0;
        else if (v >= TIME_BINS) v = TIME_BINS - 1;
        return v;
    }

    GridCell ***allocate_histogram(void) {
        GridCell ***hist = (GridCell ***)malloc((size_t)LAT_BINS * sizeof(GridCell **));
        for (int i = 0; i < LAT_BINS; i++) {
            hist[i] = (GridCell **)malloc((size_t)LON_BINS * sizeof(GridCell *));
            for (int j = 0; j < LON_BINS; j++) {
                hist[i][j] = (GridCell *)calloc((size_t)TIME_BINS, sizeof(GridCell));
            }
        }
        return hist;
    }

    void free_histogram(GridCell ***hist) {
        for (int i = 0; i < LAT_BINS; i++) {
            for (int j = 0; j < LON_BINS; j++) {
                free(hist[i][j]);
            }
```

```
            free(hist[i]);
        }
        free(hist);
    }

    void save_histogram_csv(GridCell ***histogram, const char *filename) {
        FILE *fp = fopen(filename, "w");
        if (!fp) {
            perror("Error opening output file");
            return;
        }

        fprintf(fp, "lat_bin,lon_bin,time_bin,count,total_magnitude,max_magnitude\\n");

        for (int i = 0; i < LAT_BINS; i++) {
            for (int j = 0; j < LON_BINS; j++) {
                for (int k = 0; k < TIME_BINS; k++) {
                    if (histogram[i][j][k].count > 0) {
                        fprintf(fp, "%d,%d,%d,%d,%.2f,%.2f\\n",
                                i, j, k,
                                histogram[i][j][k].count,
                                histogram[i][j][k].total_magnitude,
                                histogram[i][j][k].max_magnitude);
                    }
                }
            }
        }

        fclose(fp);
        printf("Histogram saved to %s\\n", filename);
    }

    static void *thread_histogram(void *arg) {
        ThreadData *d = (ThreadData *)arg;

        for (int k = 0; k < d->idx_count; k++) {
            int i = d->idx[k];
            int li = d->lat_bin[i];
            int lj = d->lon_bin[i];
            int tk = d->time_bin[i];

            GridCell *cell = &d->global_histogram[li][lj][tk];
            cell->count += 1;
            cell->total_magnitude += d->events[i].magnitude;
            if (d->events[i].magnitude > cell->max_magnitude) {
                cell->max_magnitude = d->events[i].magnitude;
            }
        }
        return NULL;
    }

    void pthread_histogram(const EarthquakeEvent *events, int n_events,
                           GridCell ***hist, long min_time, long max_time,
                           int num_threads, double *bucket_ms, double *compute_ms) {

        double t0 = get_time_ms();

        int *lat_bin = (int *)malloc((size_t)n_events * sizeof(int));
```

```
int *lon_bin = (int *)malloc((size_t)n_events * sizeof(int));
int *time_bin = (int *)malloc((size_t)n_events * sizeof(int));

for (int i = 0; i < n_events; i++) {
    lat_bin[i] = lat_to_bin_clamped(events[i].latitude);
    lon_bin[i] = lon_to_bin_clamped(events[i].longitude);
    time_bin[i] = time_to_bin_clamped(events[i].timestamp, min_time, max_time);
}

int *lat_lo = (int *)malloc(num_threads * sizeof(int));
int *lat_hi = (int *)malloc(num_threads * sizeof(int));
for (int t = 0; t < num_threads; t++) {
    lat_lo[t] = (LAT_BINS * t) / num_threads;
    lat_hi[t] = (LAT_BINS * (t + 1)) / num_threads - 1;
}

int *counts = (int *)calloc(num_threads, sizeof(int));
for (int i = 0; i < n_events; i++) {
    int t = (lat_bin[i] * num_threads) / LAT_BINS;
    if (t >= num_threads) t = num_threads - 1;
    counts[t]++;
}

int **buckets = (int **)malloc(num_threads * sizeof(int *));
for (int t = 0; t < num_threads; t++) {
    buckets[t] = (int *)malloc((size_t)counts[t] * sizeof(int));
    counts[t] = 0;
}

for (int i = 0; i < n_events; i++) {
    int t = (lat_bin[i] * num_threads) / LAT_BINS;
    if (t >= num_threads) t = num_threads - 1;
    buckets[t][counts[t]++] = i;
}

double t1 = get_time_ms();
*bucket_ms = t1 - t0;

pthread_t *ths = (pthread_t *)malloc(num_threads * sizeof(pthread_t));
ThreadData *td = (ThreadData *)malloc(num_threads * sizeof(ThreadData));

double c0 = get_time_ms();
for (int t = 0; t < num_threads; t++) {
    td[t].idx = buckets[t];
    td[t].idx_count = counts[t];
    td[t].thread_id = t;
    td[t].lat_lo = lat_lo[t];
    td[t].lat_hi = lat_hi[t];
    td[t].events = events;
    td[t].lat_bin = lat_bin;
    td[t].lon_bin = lon_bin;
    td[t].time_bin = time_bin;
    td[t].global_histogram = hist;

    pthread_create(&ths[t], NULL, thread_histogram, &td[t]);
}
for (int t = 0; t < num_threads; t++) {
    pthread_join(ths[t], NULL);
```

```
    }
    double c1 = get_time_ms();
    *compute_ms = c1 - c0;

    for (int t = 0; t < num_threads; t++) free(buckets[t]);
    free(buckets);
    free(counts);
    free(lat_lo);
    free(lat_hi);
    free(lat_bin);
    free(lon_bin);
    free(time_bin);
    free(ths);
    free(td);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <binary_data_file> <num_threads>\\n", argv[0]);
        return 1;
    }

    int num_threads = atoi(argv[2]);
    if (num_threads < 1) {
        fprintf(stderr, "num_threads must be >= 1\\n");
        return 1;
    }

    EarthquakeEvent *events = NULL;
    int n = load_binary_data(argv[1], &events);
    if (n <= 0) {
        fprintf(stderr, "Failed to load data\\n");
        return 1;
    }

    printf("=== Pthreads Implementation (Optimized) ===\\n");
    printf("Loaded %d earthquake events\\n", n);
    printf("Number of threads: %d\\n", num_threads);
    printf("Grid: %d x %d x %d = %d cells\\n",
           LAT_BINS, LON_BINS, TIME_BINS, LAT_BINS * LON_BINS * TIME_BINS);

    long min_time = events[0].timestamp;
    long max_time = events[0].timestamp;
    for (int i = 1; i < n; i++) {
        if (events[i].timestamp < min_time) min_time = events[i].timestamp;
        if (events[i].timestamp > max_time) max_time = events[i].timestamp;
    }

    GridCell ***hist = allocate_histogram();

    double bucket_ms = 0.0, compute_ms = 0.0;
    double t0 = get_time_ms();
    pthread_histogram(events, n, hist, min_time, max_time,
                      num_threads, &bucket_ms, &compute_ms);
    double t1 = get_time_ms();

    printf("Pthreads total time: %.2f ms\\n", t1 - t0);
    printf("  - Bucketing: %.2f ms\\n", bucket_ms);
```

```
    printf("  - Computation: %.2f ms\\n", compute_ms);
    printf("  - Assembly: 0.00 ms (direct write)\\n");

    int non_empty = 0, maxc = 0;
    for (int i = 0; i < LAT_BINS; i++)
        for (int j = 0; j < LON_BINS; j++)
            for (int k = 0; k < TIME_BINS; k++)
                if (hist[i][j][k].count > 0) {
                    non_empty++;
                    if (hist[i][j][k].count > maxc)
                        maxc = hist[i][j][k].count;
                }

    printf("Non-empty cells: %d (%.2f%%)\\n",
           non_empty, 100.0 * non_empty / (LAT_BINS * LON_BINS * TIME_BINS));
    printf("Max events in single cell: %d\\n", maxc);

    //filename has size of thread count
    char filename[256];
    snprintf(filename, sizeof(filename), "histogram_pthreads_%d.csv", num_threads);
    save_histogram_csv(hist, filename);

    free_histogram(hist);
    free(events);
    return 0;
}
"""

with open("pthreads.c","w") as f:
    f.write(pthreads)
```

```
!gcc -O3 -march=native -o pthreads pthreads.c -lpthread -lm
!echo "Pthreads with 2 threads:"
!./pthreads earthquake_data.bin 2
!echo ""
!echo "Pthreads with 4 threads:"
!./pthreads earthquake_data.bin 4
!echo ""
!echo "Pthreads with 8 threads:"
!./pthreads earthquake_data.bin 8
```

```
Pthreads with 2 threads:
=== Pthreads Implementation (Optimized) ===
Loaded 78404 earthquake events
Number of threads: 2
Grid: 180 x 360 x 120 = 7776000 cells
Pthreads total time: 2.99 ms
  - Bucketing: 1.69 ms
  - Computation: 1.21 ms
  - Assembly: 0.00 ms (direct write)
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Histogram saved to histogram_pthreads_2.csv

Pthreads with 4 threads:
=== Pthreads Implementation (Optimized) ===
Loaded 78404 earthquake events
Number of threads: 4
Grid: 180 x 360 x 120 = 7776000 cells
```

```
Pthreads total time: 2.83 ms
  - Bucketing: 1.33 ms
  - Computation: 1.43 ms
  - Assembly: 0.00 ms (direct write)
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Histogram saved to histogram_pthreads_4.csv

Pthreads with 8 threads:
=== Pthreads Implementation (Optimized) ===
Loaded 78404 earthquake events
Number of threads: 8
Grid: 180 x 360 x 120 = 7776000 cells
Pthreads total time: 2.91 ms
  - Bucketing: 1.30 ms
  - Computation: 1.56 ms
  - Assembly: 0.00 ms (direct write)
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Histogram saved to histogram_pthreads_8.csv
```

OpenMP Version

```
openmp_code = r"""
#include "earthquake_analysis.h"
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

/* ---- local helpers so we can compile standalone ---- */
double get_time_ms(void){
    struct timeval tv; gettimeofday(&tv, NULL);
    return (tv.tv_sec * 1000.0) + (tv.tv_usec / 1000.0);
}
int load_binary_data(const char *filename, EarthquakeEvent **events){
    FILE *fp = fopen(filename, "rb");
    if(!fp){ perror("fopen"); return -1; }
    int n = 0;
    if(fread(&n, sizeof(int), 1, fp) != 1){ perror("fread n"); fclose(fp); return -1; }
    *events = (EarthquakeEvent*) malloc((size_t)n * sizeof(EarthquakeEvent));
    if(!*events){ perror("malloc events"); fclose(fp); return -1; }
    for(int i=0;i<n;i++){
        if(fread(&(*events)[i], sizeof(EarthquakeEvent), 1, fp) != 1){
            perror("fread event"); free(*events); fclose(fp); return -1;
        }
    }
    fclose(fp);
    return n;
}
/* ------------------------------------------------- */

static inline int lat_to_bin_clamped(double lat){
    int v = (int)((lat + 90.0) * LAT_BINS / 180.0);
    if (v < 0) v = 0; else if (v >= LAT_BINS) v = LAT_BINS - 1; return v;
}
static inline int lon_to_bin_clamped(double lon){
```

```
    int v = (int)((lon + 180.0) * LON_BINS / 360.0);
    if (v < 0) v = 0; else if (v >= LON_BINS) v = LON_BINS - 1; return v;
}
static inline int time_to_bin_clamped(long ts,long mn,long mx){
    double d = (double)(mx - mn + 1);
    int v = (int)((ts - mn) * TIME_BINS / d);
    if (v < 0) v = 0; else if (v >= TIME_BINS) v = TIME_BINS - 1; return v;
}

static GridCell*** alloc_hist(void){
    GridCell ***h = (GridCell ***)malloc((size_t)LAT_BINS * sizeof(GridCell **));
    for(int i=0;i<LAT_BINS;i++){
        h[i] = (GridCell **)malloc((size_t)LON_BINS * sizeof(GridCell *));
        for(int j=0;j<LON_BINS;j++){
            h[i][j] = (GridCell *)calloc((size_t)TIME_BINS, sizeof(GridCell));
        }
    }
    return h;
}
static void free_hist(GridCell ***h){
    for(int i=0;i<LAT_BINS;i++){ for(int j=0;j<LON_BINS;j++) free(h[i][j]); free(h[i]); }
    free(h);
}

void save_histogram_csv(GridCell ***histogram, const char *filename) {
    FILE *fp = fopen(filename, "w");
    if (!fp) {
        perror("Error opening output file");
        return;
    }

    fprintf(fp, "lat_bin,lon_bin,time_bin,count,total_magnitude,max_magnitude\n");

    for (int i = 0; i < LAT_BINS; i++) {
        for (int j = 0; j < LON_BINS; j++) {
            for (int k = 0; k < TIME_BINS; k++) {
                if (histogram[i][j][k].count > 0) {
                    fprintf(fp, "%d,%d,%d,%d,%.2f,%.2f\n",
                            i, j, k,
                            histogram[i][j][k].count,
                            histogram[i][j][k].total_magnitude,
                            histogram[i][j][k].max_magnitude);
                }
            }
        }
    }

    fclose(fp);
    printf("Histogram saved to %s\n", filename);
}

int main(int argc,char**argv){
    if(argc!=3){ fprintf(stderr,"Usage: %s <binary_data_file> <num_threads>\n",argv[0]); retu
    int T = atoi(argv[2]); if(T<1){ fprintf(stderr,"num_threads must be >=1\n"); return 1; }
    omp_set_num_threads(T);

    EarthquakeEvent *ev=NULL; int N=load_binary_data(argv[1], &ev);
    if(N<=0){ fprintf(stderr,"Failed to load data\n"); return 1; }
```

```
long tmin=ev[0].timestamp, tmax=ev[0].timestamp;
for(int i=1;i<N;i++){ if(ev[i].timestamp<tmin)tmin=ev[i].timestamp; if(ev[i].timestamp>tma

printf("=== OpenMP Implementation (Stripe + Bucket) ===\n");
printf("Loaded %d earthquake events\n",N);
printf("Number of threads: %d\n",T);
printf("Grid dimensions: %d x %d x %d = %d cells\n", LAT_BINS,LON_BINS,TIME_BINS, LAT_BINS

GridCell ***H = alloc_hist();

//1 precompute bins
double t0=get_time_ms();
int *latb=(int*)malloc((size_t)N*sizeof(int));
int *lonb=(int*)malloc((size_t)N*sizeof(int));
int *timeb=(int*)malloc((size_t)N*sizeof(int));
#pragma omp parallel for schedule(static)
for(int i=0;i<N;i++){
    latb[i]=lat_to_bin_clamped(ev[i].latitude);
    lonb[i]=lon_to_bin_clamped(ev[i].longitude);
    timeb[i]=time_to_bin_clamped(ev[i].timestamp,tmin,tmax);
}

//2 bucket counts per stripe
int *counts=(int*)calloc((size_t)T,sizeof(int));
for(int i=0;i<N;i++){ int t=(latb[i]*T)/LAT_BINS; if(t>=T)t=T-1; counts[t]++; }

//3 buckets of indices
int **bucket=(int**)malloc((size_t)T*sizeof(int*));
for(int t=0;t<T;t++){ bucket[t]=(int*)malloc((size_t)counts[t]*sizeof(int)); counts[t]=0;
for(int i=0;i<N;i++){ int t=(latb[i]*T)/LAT_BINS; if(t>=T)t=T-1; bucket[t][counts[t]++]=i;
double t1=get_time_ms(); double bucketing_ms=t1-t0;

//4 parallel compute / direct writes
double c0=get_time_ms();
#pragma omp parallel
{
    int tid=omp_get_thread_num();
    int *lst=bucket[tid]; int cnt=counts[tid];
    for(int k=0;k<cnt;k++){
        int i=lst[k]; int li=latb[i], lj=lonb[i], tk=timeb[i];
        GridCell *cell=&H[li][lj][tk];
        cell->count += 1;
        cell->total_magnitude += ev[i].magnitude;
        if(ev[i].magnitude > cell->max_magnitude) cell->max_magnitude = ev[i].magnitude;
    }
}
double c1=get_time_ms(); double compute_ms=c1-c0;

//validate results
int non_empty=0, maxc=0;
for(int i=0;i<LAT_BINS;i++)
    for(int j=0;j<LON_BINS;j++)
        for(int k=0;k<TIME_BINS;k++)
            if(H[i][j][k].count>0){ non_empty++; if(H[i][j][k].count>maxc) maxc=H[i][j][k]

printf("OpenMP total time: %.2f ms\n", bucketing_ms + compute_ms);
printf("  - Bucketing/Indexing: %.2f ms\n", bucketing_ms);
printf("  - Computation: %.2f ms\n", compute_ms);
```

```
    printf("  - Assembly: 0.00 ms (direct write)\n");
    printf("Non-empty cells: %d (%.2f%%)\n", non_empty, 100.0*non_empty/(LAT_BINS*LON_BINS*TI
    printf("Max events in single cell: %d\n", maxc);

    char filename[256];
    snprintf(filename, sizeof(filename), "histogram_openmp_%d.csv", T);
    save_histogram_csv(H, filename);

    for(int t=0;t<T;t++) free(bucket[t]);
    free(bucket); free(counts);
    free(latb); free(lonb); free(timeb);
    free_hist(H); free(ev);
    return 0;
}
"""
with open("openmp.c","w") as f:
    f.write(openmp_code)
```

```
!gcc -O3 -march=native -fopenmp -funroll-loops -fno-math-errno openmp.c -o openmp
!echo "OpenMP with 2 threads:"
!./openmp earthquake_data.bin 2
!echo ""
!echo "OpenMP with 4 threads:"
!./openmp earthquake_data.bin 4
!echo ""
!echo "OpenMP with 8 threads:"
!./openmp earthquake_data.bin 8
```

```
OpenMP with 2 threads:
=== OpenMP Implementation (Stripe + Bucket) ===
Loaded 78404 earthquake events
Number of threads: 2
Grid dimensions: 180 x 360 x 120 = 7776000 cells
OpenMP total time: 2.38 ms
  - Bucketing/Indexing: 1.34 ms
  - Computation: 1.04 ms
  - Assembly: 0.00 ms (direct write)
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Histogram saved to histogram_openmp_2.csv

OpenMP with 4 threads:
=== OpenMP Implementation (Stripe + Bucket) ===
Loaded 78404 earthquake events
Number of threads: 4
Grid dimensions: 180 x 360 x 120 = 7776000 cells
OpenMP total time: 2.71 ms
  - Bucketing/Indexing: 1.39 ms
  - Computation: 1.32 ms
  - Assembly: 0.00 ms (direct write)
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Histogram saved to histogram_openmp_4.csv

OpenMP with 8 threads:
=== OpenMP Implementation (Stripe + Bucket) ===
Loaded 78404 earthquake events
Number of threads: 8
```

```
Grid dimensions: 180 x 360 x 120 = 7776000 cells
OpenMP total time: 2.66 ms
  - Bucketing/Indexing: 1.44 ms
  - Computation: 1.22 ms
  - Assembly: 0.00 ms (direct write)
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Histogram saved to histogram_openmp_8.csv
```

MPI

```
mpi_code = r"""
#include "earthquake_analysis.h"
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

double get_time_ms(void){
    struct timeval tv; gettimeofday(&tv, NULL);
    return (tv.tv_sec*1000.0) + (tv.tv_usec/1000.0);
}

int load_binary_data(const char *filename, EarthquakeEvent **events){
    FILE *fp = fopen(filename, "rb");
    if (!fp) { perror("fopen"); return -1; }
    int n = 0;
    if (fread(&n, sizeof(int), 1, fp) != 1) { perror("fread n"); fclose(fp); return -1; }
    *events = (EarthquakeEvent*)malloc((size_t)n * sizeof(EarthquakeEvent));
    if (!*events) { perror("malloc events"); fclose(fp); return -1; }
    for (int i = 0; i < n; ++i){
        if (fread(&(*events)[i], sizeof(EarthquakeEvent), 1, fp) != 1){
            perror("fread event"); free(*events); fclose(fp); return -1;
        }
    }
    fclose(fp);
    return n;
}

static inline int lat_to_bin_clamped(double lat){
    int v = (int)((lat + 90.0) * LAT_BINS / 180.0);
    if (v < 0) v = 0; else if (v >= LAT_BINS) v = LAT_BINS - 1;
    return v;
}

static inline int lon_to_bin_clamped(double lon){
    int v = (int)((lon + 180.0) * LON_BINS / 360.0);
    if (v < 0) v = 0; else if (v >= LON_BINS) v = LON_BINS - 1;
    return v;
}

static inline int time_to_bin_clamped(long ts, long min_time, long max_time){
    double denom = (double)(max_time - min_time + 1);
    int v = (int)((ts - min_time) * TIME_BINS / denom);
    if (v < 0) v = 0; else if (v >= TIME_BINS) v = TIME_BINS - 1;
```

```
    return v;
}

GridCell*** allocate_histogram_stripe(int lat_count) {
    GridCell ***hist = (GridCell ***)malloc((size_t)lat_count * sizeof(GridCell **));
    for (int i = 0; i < lat_count; i++) {
        hist[i] = (GridCell **)malloc((size_t)LON_BINS * sizeof(GridCell *));
        for (int j = 0; j < LON_BINS; j++) {
            hist[i][j] = (GridCell *)calloc((size_t)TIME_BINS, sizeof(GridCell));
        }
    }
    return hist;
}

void free_histogram_stripe(GridCell ***hist, int lat_count) {
    for (int i = 0; i < lat_count; i++) {
        for (int j = 0; j < LON_BINS; j++) {
            free(hist[i][j]);
        }
        free(hist[i]);
    }
    free(hist);
}

void save_rank_stripe(GridCell ***histogram, int lat_start, int lat_count, int rank) {
    char filename[256];
    snprintf(filename, sizeof(filename), "histogram_mpi_rank_%d.tmp", rank);
    FILE *fp = fopen(filename, "w");
    if (!fp) {
        perror("Error opening temp file");
        return;
    }

    for (int i = 0; i < lat_count; i++) {
        for (int j = 0; j < LON_BINS; j++) {
            for (int k = 0; k < TIME_BINS; k++) {
                if (histogram[i][j][k].count > 0) {
                    fprintf(fp, "%d,%d,%d,%d,%.2f,%.2f\n",
                           i + lat_start, j, k,
                           histogram[i][j][k].count,
                           histogram[i][j][k].total_magnitude,
                           histogram[i][j][k].max_magnitude);
                }
            }
        }
    }

    fclose(fp);
}

void consolidate_histogram_files(int num_procs, const char *final_filename) {
    FILE *out = fopen(final_filename, "w");
    if (!out) {
        perror("Error opening final output file");
        return;
    }

    fprintf(out, "lat_bin,lon_bin,time_bin,count,total_magnitude,max_magnitude\n");
```

```
        for (int rank = 0; rank < num_procs; rank++) {
            char temp_filename[256];
            snprintf(temp_filename, sizeof(temp_filename), "histogram_mpi_rank_%d.tmp", rank);

            FILE *in = fopen(temp_filename, "r");
            if (!in) {
                fprintf(stderr, "Warning: Could not open %s\n", temp_filename);
                continue;
            }

            char line[512];
            while (fgets(line, sizeof(line), in)) {
                fputs(line, out);
            }

            fclose(in);
            remove(temp_filename);
        }

        fclose(out);
        printf("Histogram saved to %s\n", final_filename);
    }

    int main(int argc, char **argv){
        MPI_Init(&argc, &argv);
        int rank, num_procs;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

        if (argc != 2){
            if (rank == 0) fprintf(stderr, "Usage: %s <binary_data_file>\n", argv[0]);
            MPI_Finalize();
            return 1;
        }

        EarthquakeEvent *events = NULL;
        int n_events = load_binary_data(argv[1], &events);
        if (n_events <= 0){
            if (rank == 0) fprintf(stderr, "Failed to load data\n");
            MPI_Finalize();
            return 1;
        }

        long min_time = events[0].timestamp;
        long max_time = events[0].timestamp;
        for (int i = 1; i < n_events; i++){
            if (events[i].timestamp < min_time) min_time = events[i].timestamp;
            if (events[i].timestamp > max_time) max_time = events[i].timestamp;
        }

        int lat_per_proc = LAT_BINS / num_procs;
        int lat_start = rank * lat_per_proc;
        int lat_end = (rank == num_procs - 1) ? LAT_BINS : (rank + 1) * lat_per_proc;
        int lat_count = lat_end - lat_start;

        if (rank == 0){
            printf("=== MPI Implementation (Stripe + Bucket) ===\n");
```

```
    printf("Loaded %d earthquake events\n", n_events);
    printf("Number of processes: %d\n", num_procs);
    printf("Grid: %d x %d x %d = %d cells\n",
           LAT_BINS, LON_BINS, TIME_BINS, LAT_BINS * LON_BINS * TIME_BINS);
}

MPI_Barrier(MPI_COMM_WORLD);
double t0 = MPI_Wtime();

int *lat_bin = (int *)malloc((size_t)n_events * sizeof(int));
int *lon_bin = (int *)malloc((size_t)n_events * sizeof(int));
int *time_bin = (int *)malloc((size_t)n_events * sizeof(int));

for (int i = 0; i < n_events; i++){
    lat_bin[i] = lat_to_bin_clamped(events[i].latitude);
    lon_bin[i] = lon_to_bin_clamped(events[i].longitude);
    time_bin[i] = time_to_bin_clamped(events[i].timestamp, min_time, max_time);
}

GridCell ***histogram = allocate_histogram_stripe(lat_count);

for (int i = 0; i < n_events; i++){
    int li = lat_bin[i];

    if (li < lat_start || li >= lat_end) continue;

    int lj = lon_bin[i];
    int tk = time_bin[i];

    int local_lat = li - lat_start;
    GridCell *cell = &histogram[local_lat][lj][tk];

    cell->count += 1;
    cell->total_magnitude += events[i].magnitude;
    if (events[i].magnitude > cell->max_magnitude){
        cell->max_magnitude = events[i].magnitude;
    }
}

MPI_Barrier(MPI_COMM_WORLD);
double t1 = MPI_Wtime();

int local_non_empty = 0;
int local_maxc = 0;
for (int i = 0; i < lat_count; i++){
    for (int j = 0; j < LON_BINS; j++){
        for (int k = 0; k < TIME_BINS; k++){
            if (histogram[i][j][k].count > 0){
                local_non_empty++;
                if (histogram[i][j][k].count > local_maxc){
                    local_maxc = histogram[i][j][k].count;
                }
            }
        }
    }
}

int *all_non_empty = NULL;
```

```
    int *all_maxc = NULL;
    if (rank == 0){
        all_non_empty = (int *)malloc(num_procs * sizeof(int));
        all_maxc = (int *)malloc(num_procs * sizeof(int));
    }

    MPI_Gather(&local_non_empty, 1, MPI_INT, all_non_empty, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Gather(&local_maxc, 1, MPI_INT, all_maxc, 1, MPI_INT, 0, MPI_COMM_WORLD);

    save_rank_stripe(histogram, lat_start, lat_count, rank);

    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0){
        int total_non_empty = 0;
        int global_maxc = 0;
        for (int p = 0; p < num_procs; p++){
            total_non_empty += all_non_empty[p];
            if (all_maxc[p] > global_maxc) global_maxc = all_maxc[p];
        }

        printf("MPI total time: %.2f ms\n", (t1 - t0) * 1000.0);
        printf("Non-empty cells: %d (%.2f%%)\n",
               total_non_empty, 100.0 * total_non_empty / (LAT_BINS * LON_BINS * TIME_BINS));
        printf("Max events in single cell: %d\n", global_maxc);

        char filename[256];
        snprintf(filename, sizeof(filename), "histogram_mpi_%d.csv", num_procs);
        consolidate_histogram_files(num_procs, filename);

        free(all_non_empty);
        free(all_maxc);
    }

    free(lat_bin);
    free(lon_bin);
    free(time_bin);
    free_histogram_stripe(histogram, lat_count);
    free(events);

    MPI_Finalize();
    return 0;
}
"""

with open("mpi.c","w") as f:
    f.write(mpi_code)
```

```
!mpicc -O3 -march=native mpi.c -o mpi_prog
!echo "MPI with 2 processes:"
!mpirun --allow-run-as-root --oversubscribe -np 2 ./mpi_prog earthquake_data.bin
!echo ""
!echo "MPI with 4 processes:"
!mpirun --allow-run-as-root --oversubscribe -np 4 ./mpi_prog earthquake_data.bin
!echo ""
!echo "MPI with 8 processes:"
!mpirun --allow-run-as-root --oversubscribe -np 8 ./mpi_prog earthquake_data.bin
```

```
MPI with 2 processes:
=== MPI Implementation (Stripe + Bucket) ===
Loaded 78404 earthquake events
Number of processes: 2
Grid: 180 x 360 x 120 = 7776000 cells
MPI total time: 33.29 ms
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Histogram saved to histogram_mpi_2.csv

MPI with 4 processes:
=== MPI Implementation (Stripe + Bucket) ===
Loaded 78404 earthquake events
Number of processes: 4
Grid: 180 x 360 x 120 = 7776000 cells
MPI total time: 47.87 ms
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Histogram saved to histogram_mpi_4.csv

MPI with 8 processes:
=== MPI Implementation (Stripe + Bucket) ===
Loaded 78404 earthquake events
Number of processes: 8
Grid: 180 x 360 x 120 = 7776000 cells
MPI total time: 86.06 ms
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Histogram saved to histogram_mpi_8.csv
```

CUDA

```
cuda_basic_code = """#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

#define LAT_BINS 180
#define LON_BINS 360
#define TIME_BINS 120

typedef struct {
    long timestamp;
    float latitude;
    float longitude;
    float magnitude;
} EarthquakeEvent;

__device__ void compute_bin_indices(EarthquakeEvent event, long min_time, long max_time,
                                    int *lat_idx, int *lon_idx, int *time_idx) {
    *lat_idx = (int)((event.latitude + 90.0f) * LAT_BINS / 180.0f);
    if (*lat_idx >= LAT_BINS) *lat_idx = LAT_BINS - 1;
    if (*lat_idx < 0) *lat_idx = 0;

    *lon_idx = (int)((event.longitude + 180.0f) * LON_BINS / 360.0f);
    if (*lon_idx >= LON_BINS) *lon_idx = LON_BINS - 1;
    if (*lon_idx < 0) *lon_idx = 0;
```

```
    *time_idx = (int)((event.timestamp - min_time) * TIME_BINS /
                      (double)(max_time - min_time + 1));
    if (*time_idx >= TIME_BINS) *time_idx = TIME_BINS - 1;
    if (*time_idx < 0) *time_idx = 0;
}

__device__ float atomicMaxFloat(float* address, float val) {
    int* address_as_int = (int*)address;
    int old = *address_as_int, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_int, assumed,
                        __float_as_int(fmaxf(val, __int_as_float(assumed))));
    } while (assumed != old);
    return __int_as_float(old);
}

__global__ void histogram_kernel(EarthquakeEvent *events, int n_events,
                                 int *counts, float *mags, float *maxs,
                                 long min_time, long max_time) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n_events) {
        int lat_idx, lon_idx, time_idx;
        compute_bin_indices(events[idx], min_time, max_time,
                            &lat_idx, &lon_idx, &time_idx);

        int cell_idx = lat_idx * LON_BINS * TIME_BINS + lon_idx * TIME_BINS + time_idx;

        atomicAdd(&counts[cell_idx], 1);
        atomicAdd(&mags[cell_idx], events[idx].magnitude);
        atomicMaxFloat(&maxs[cell_idx], events[idx].magnitude);
    }
}

int load_binary_data(const char *filename, EarthquakeEvent **events) {
    FILE *fp = fopen(filename, "rb");
    if (!fp) {
        perror("Error opening file");
        return -1;
    }

    int n_events;
    fread(&n_events, sizeof(int), 1, fp);

    *events = (EarthquakeEvent *)malloc(n_events * sizeof(EarthquakeEvent));
    if (!*events) {
        fprintf(stderr, "Memory allocation failed\\n");
        fclose(fp);
        return -1;
    }

    for (int i = 0; i < n_events; i++) {
        fread(&(*events)[i], sizeof(EarthquakeEvent), 1, fp);
    }

    fclose(fp);
    return n_events;
```

```
}

void write_histogram_csv(const char *filename, int *counts, float *mags, float *maxs) {
    FILE *fp = fopen(filename, "w");
    if (!fp) {
        perror("Error opening output file");
        return;
    }

    fprintf(fp, "lat_bin,lon_bin,time_bin,count,total_magnitude,max_magnitude\\n");

    for (int lat = 0; lat < LAT_BINS; lat++) {
        for (int lon = 0; lon < LON_BINS; lon++) {
            for (int time = 0; time < TIME_BINS; time++) {
                int idx = lat * LON_BINS * TIME_BINS + lon * TIME_BINS + time;
                if (counts[idx] > 0) {
                    fprintf(fp, "%d,%d,%d,%d,%.2f,%.2f\\n",
                           lat, lon, time, counts[idx], mags[idx], maxs[idx]);
                }
            }
        }
    }

    fclose(fp);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <binary_data_file> <block_size>\\n", argv[0]);
        return 1;
    }

    int block_size = atoi(argv[2]);
    if (block_size < 1) block_size = 256;

    EarthquakeEvent *h_events;
    int n_events = load_binary_data(argv[1], &h_events);

    if (n_events <= 0) {
        fprintf(stderr, "Failed to load data\\n");
        return 1;
    }

    printf("=== CUDA Basic Implementation (Block Size %d) ===\\n", block_size);
    printf("Loaded %d earthquake events\\n", n_events);

    long min_time = h_events[0].timestamp;
    long max_time = h_events[0].timestamp;
    for (int i = 1; i < n_events; i++) {
        if (h_events[i].timestamp < min_time) min_time = h_events[i].timestamp;
        if (h_events[i].timestamp > max_time) max_time = h_events[i].timestamp;
    }

    int total_cells = LAT_BINS * LON_BINS * TIME_BINS;
    printf("Grid dimensions: %d x %d x %d = %d cells\\n",
           LAT_BINS, LON_BINS, TIME_BINS, total_cells);

    EarthquakeEvent *d_events;
```

```
int *d_counts;
float *d_mags, *d_maxs;

cudaMalloc(&d_events, n_events * sizeof(EarthquakeEvent));
cudaMalloc(&d_counts, total_cells * sizeof(int));
cudaMalloc(&d_mags, total_cells * sizeof(float));
cudaMalloc(&d_maxs, total_cells * sizeof(float));

cudaMemset(d_counts, 0, total_cells * sizeof(int));
cudaMemset(d_mags, 0, total_cells * sizeof(float));
cudaMemset(d_maxs, 0, total_cells * sizeof(float));

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
cudaMemcpy(d_events, h_events, n_events * sizeof(EarthquakeEvent), cudaMemcpyHostToDevic

int grid_size = (n_events + block_size - 1) / block_size;
histogram_kernel<<<grid_size, block_size>>>(d_events, n_events, d_counts, d_mags, d_maxs
                                            min_time, max_time);

cudaDeviceSynchronize();
cudaEventRecord(stop);
cudaEventSynchronize(stop);

float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);

printf("\\nExecution time: %.5f ms\\n", milliseconds);

int *h_counts = (int *)malloc(total_cells * sizeof(int));
float *h_mags = (float *)malloc(total_cells * sizeof(float));
float *h_maxs = (float *)malloc(total_cells * sizeof(float));

cudaMemcpy(h_counts, d_counts, total_cells * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(h_mags, d_mags, total_cells * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(h_maxs, d_maxs, total_cells * sizeof(float), cudaMemcpyDeviceToHost);

int non_empty = 0;
int max_count = 0;
long long total_events = 0;
for (int i = 0; i < total_cells; i++) {
    if (h_counts[i] > 0) {
        non_empty++;
        total_events += h_counts[i];
        if (h_counts[i] > max_count) max_count = h_counts[i];
    }
}

printf("Non-empty cells: %d (%.2f%%)\\n",
       non_empty, 100.0 * non_empty / total_cells);
printf("Max events in single cell: %d\\n", max_count);
printf("Total events processed: %lld\\n\\n", total_events);

char output_filename[256];
sprintf(output_filename, "histogram_cuda_%d.csv", block_size);
```

```
    write_histogram_csv(output_filename, h_counts, h_mags, h_maxs);
    printf("Histogram saved to %s\\n", output_filename);

    free(h_events);
    free(h_counts);
    free(h_mags);
    free(h_maxs);
    cudaFree(d_events);
    cudaFree(d_counts);
    cudaFree(d_mags);
    cudaFree(d_maxs);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}
"""

with open('cuda_basic.cu', 'w') as f:
    f.write(cuda_basic_code)
```

```
!nvcc -arch=sm_75 -o cuda_basic cuda_basic.cu
!echo "CUDA with block size 128:"
!./cuda_basic earthquake_data.bin 128
!echo ""
!echo "CUDA with block size 256"
!./cuda_basic earthquake_data.bin 256
!echo ""
!echo "CUDA with block size 512"
!./cuda_basic earthquake_data.bin 512
!echo ""
!echo "CUDA with block size 1024"
!./cuda_basic earthquake_data.bin 1024
```

```
CUDA with block size 128:
=== CUDA Basic Implementation (Block Size 128) ===
Loaded 78404 earthquake events
Grid dimensions: 180 x 360 x 120 = 7776000 cells

Execution time: 1.70928 ms
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Total events processed: 78404

Histogram saved to histogram_cuda_128.csv

CUDA with block size 256
=== CUDA Basic Implementation (Block Size 256) ===
Loaded 78404 earthquake events
Grid dimensions: 180 x 360 x 120 = 7776000 cells

Execution time: 0.65891 ms
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Total events processed: 78404

Histogram saved to histogram_cuda_256.csv

CUDA with block size 512
=== CUDA Basic Implementation (Block Size 512) ===
```

```
Loaded 78404 earthquake events
Grid dimensions: 180 x 360 x 120 = 7776000 cells

Execution time: 0.69078 ms
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Total events processed: 78404

Histogram saved to histogram_cuda_512.csv

CUDA with block size 1024
=== CUDA Basic Implementation (Block Size 1024) ===
Loaded 78404 earthquake events
Grid dimensions: 180 x 360 x 120 = 7776000 cells

Execution time: 0.66518 ms
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Total events processed: 78404

Histogram saved to histogram_cuda_1024.csv
```

Optimized CUDA Version with Shared Memory

```
cuda_shared_code = """#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

#define LAT_BINS 180
#define LON_BINS 360
#define TIME_BINS 120
#define SHARED_HIST_SIZE 4096  // Reduced to fit in 64KB limit

typedef struct {
    long timestamp;
    float latitude;
    float longitude;
    float magnitude;
} EarthquakeEvent;

__device__ void compute_bin_indices(EarthquakeEvent event, long min_time, long max_time,
                                    int *lat_idx, int *lon_idx, int *time_idx) {
    *lat_idx = (int)((event.latitude + 90.0f) * LAT_BINS / 180.0f);
    if (*lat_idx >= LAT_BINS) *lat_idx = LAT_BINS - 1;
    if (*lat_idx < 0) *lat_idx = 0;

    *lon_idx = (int)((event.longitude + 180.0f) * LON_BINS / 360.0f);
    if (*lon_idx >= LON_BINS) *lon_idx = LON_BINS - 1;
    if (*lon_idx < 0) *lon_idx = 0;

    *time_idx = (int)((event.timestamp - min_time) * TIME_BINS /
                      (double)(max_time - min_time + 1));
    if (*time_idx >= TIME_BINS) *time_idx = TIME_BINS - 1;
    if (*time_idx < 0) *time_idx = 0;
}

__device__ float atomicMaxFloat(float* address, float val) {
    int* address_as_int = (int*)address;
```

```
    int old = *address_as_int, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_int, assumed,
                        __float_as_int(fmaxf(val, __int_as_float(assumed))));
    } while (assumed != old);
    return __int_as_float(old);
}

//shared memory kernel, each block maintains a local histogram
__global__ void histogram_kernel_shared(EarthquakeEvent *events, int n_events,
                                        int *counts, float *mags, float *maxs,
                                        long min_time, long max_time) {
    //shared memory allocation for local histogram size 4096 bins which is the memory limit
     __shared__ int local_counts[SHARED_HIST_SIZE];
    __shared__ float local_mags[SHARED_HIST_SIZE];
    __shared__ float local_maxs[SHARED_HIST_SIZE];

    int tid = threadIdx.x;

    //initialize shared memory, and each thread initalizes specific elements
    for (int i = tid; i < SHARED_HIST_SIZE; i += blockDim.x) {
        local_counts[i] = 0;
        local_mags[i] = 0.0f;
        local_maxs[i] = 0.0f;
    }
    __syncthreads();

    //each thread processes
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n_events) {
        int lat_idx, lon_idx, time_idx;
        compute_bin_indices(events[idx], min_time, max_time,
                            &lat_idx, &lon_idx, &time_idx);

        int cell_idx = lat_idx * LON_BINS * TIME_BINS + lon_idx * TIME_BINS + time_idx;

        //if cell fits use shared memory
        if (cell_idx < SHARED_HIST_SIZE) {
            atomicAdd(&local_counts[cell_idx], 1);
            atomicAdd(&local_mags[cell_idx], events[idx].magnitude);
            atomicMaxFloat(&local_maxs[cell_idx], events[idx].magnitude);
        } else {
            atomicAdd(&counts[cell_idx], 1);
            atomicAdd(&mags[cell_idx], events[idx].magnitude);
            atomicMaxFloat(&maxs[cell_idx], events[idx].magnitude);
        }
    }

    __syncthreads();

    // write back from shared memory to global one
    for (int i = tid; i < SHARED_HIST_SIZE; i += blockDim.x) {
        if (local_counts[i] > 0) {
            atomicAdd(&counts[i], local_counts[i]);
            atomicAdd(&mags[i], local_mags[i]);
            atomicMaxFloat(&maxs[i], local_maxs[i]);
```

```
            }
        }
    }

    int load_binary_data(const char *filename, EarthquakeEvent **events) {
        FILE *fp = fopen(filename, "rb");
        if (!fp) {
            perror("Error opening file");
            return -1;
        }

        int n_events;
        fread(&n_events, sizeof(int), 1, fp);

        *events = (EarthquakeEvent *)malloc(n_events * sizeof(EarthquakeEvent));
        if (!*events) {
            fprintf(stderr, "Memory allocation failed\\n");
            fclose(fp);
            return -1;
        }

        for (int i = 0; i < n_events; i++) {
            fread(&(*events)[i], sizeof(EarthquakeEvent), 1, fp);
        }

        fclose(fp);
        return n_events;
    }

    void write_histogram_csv(const char *filename, int *counts, float *mags, float *maxs) {
        FILE *fp = fopen(filename, "w");
        if (!fp) {
            perror("Error opening output file");
            return;
        }

        fprintf(fp, "lat_bin,lon_bin,time_bin,count,total_magnitude,max_magnitude\\n");

        for (int lat = 0; lat < LAT_BINS; lat++) {
            for (int lon = 0; lon < LON_BINS; lon++) {
                for (int time = 0; time < TIME_BINS; time++) {
                    int idx = lat * LON_BINS * TIME_BINS + lon * TIME_BINS + time;
                    if (counts[idx] > 0) {
                        fprintf(fp, "%d,%d,%d,%d,%.2f,%.2f\\n",
                               lat, lon, time, counts[idx], mags[idx], maxs[idx]);
                    }
                }
            }
        }

        fclose(fp);
    }

    int main(int argc, char *argv[]) {
        if (argc != 2) {
            fprintf(stderr, "Usage: %s <binary_data_file>\\n", argv[0]);
            return 1;
        }
```

```
    int block_size = 256;

    EarthquakeEvent *h_events;
    int n_events = load_binary_data(argv[1], &h_events);

    if (n_events <= 0) {
        fprintf(stderr, "Failed to load data\\n");
        return 1;
    }

    printf("=== CUDA Shared Memory Implementation ===\\n");
    printf("Loaded %d earthquake events\\n", n_events);

    long min_time = h_events[0].timestamp;
    long max_time = h_events[0].timestamp;
    for (int i = 1; i < n_events; i++) {
        if (h_events[i].timestamp < min_time) min_time = h_events[i].timestamp;
        if (h_events[i].timestamp > max_time) max_time = h_events[i].timestamp;
    }

    int total_cells = LAT_BINS * LON_BINS * TIME_BINS;
    printf("Grid dimensions: %d x %d x %d = %d cells\\n",
           LAT_BINS, LON_BINS, TIME_BINS, total_cells);
    printf("Shared memory histogram size: %d cells (%.1f KB per block)\\n",
           SHARED_HIST_SIZE, (SHARED_HIST_SIZE * (sizeof(int) + 2*sizeof(float))) / 1024.0);

    EarthquakeEvent *d_events;
    int *d_counts;
    float *d_mags, *d_maxs;

    cudaMalloc(&d_events, n_events * sizeof(EarthquakeEvent));
    cudaMalloc(&d_counts, total_cells * sizeof(int));
    cudaMalloc(&d_mags, total_cells * sizeof(float));
    cudaMalloc(&d_maxs, total_cells * sizeof(float));

    cudaMemset(d_counts, 0, total_cells * sizeof(int));
    cudaMemset(d_mags, 0, total_cells * sizeof(float));
    cudaMemset(d_maxs, 0, total_cells * sizeof(float));

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);
    cudaMemcpy(d_events, h_events, n_events * sizeof(EarthquakeEvent), cudaMemcpyHostToDevic

    int grid_size = (n_events + block_size - 1) / block_size;
    histogram_kernel_shared<<<grid_size, block_size>>>(d_events, n_events, d_counts, d_mags,
                                                       min_time, max_time);

    cudaDeviceSynchronize();
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
```

```
    printf("\\nExecution time: %.5f ms\\n", milliseconds);

    int *h_counts = (int *)malloc(total_cells * sizeof(int));
    float *h_mags = (float *)malloc(total_cells * sizeof(float));
    float *h_maxs = (float *)malloc(total_cells * sizeof(float));

    cudaMemcpy(h_counts, d_counts, total_cells * sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(h_mags, d_mags, total_cells * sizeof(float), cudaMemcpyDeviceToHost);
    cudaMemcpy(h_maxs, d_maxs, total_cells * sizeof(float), cudaMemcpyDeviceToHost);

    int non_empty = 0;
    int max_count = 0;
    long long total_events = 0;
    for (int i = 0; i < total_cells; i++) {
        if (h_counts[i] > 0) {
            non_empty++;
            total_events += h_counts[i];
            if (h_counts[i] > max_count) max_count = h_counts[i];
        }
    }

    printf("Non-empty cells: %d (%.2f%%)\\n",
           non_empty, 100.0 * non_empty / total_cells);
    printf("Max events in single cell: %d\\n", max_count);
    printf("Total events processed: %lld\\n\\n", total_events);

    write_histogram_csv("histogram_cuda_shared.csv", h_counts, h_mags, h_maxs);
    printf("Histogram saved to histogram_cuda_shared.csv\\n");

    free(h_events);
    free(h_counts);
    free(h_mags);
    free(h_maxs);
    cudaFree(d_events);
    cudaFree(d_counts);
    cudaFree(d_mags);
    cudaFree(d_maxs);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}
"""

with open('cuda_shared.cu', 'w') as f:
    f.write(cuda_shared_code)
```

```
!nvcc -arch=sm_75 -o cuda_shared cuda_shared.cu
!echo "CUDA with shared memory:"
!./cuda_shared earthquake_data.bin
```

```
CUDA with shared memory:
=== CUDA Shared Memory Implementation ===
Loaded 78404 earthquake events
Grid dimensions: 180 x 360 x 120 = 7776000 cells
Shared memory histogram size: 4096 cells (48.0 KB per block)

Execution time: 0.77005 ms
```

```
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Total events processed: 78404

Histogram saved to histogram_cuda_shared.csv
```

```
cuda_tiled_code = """#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

#define LAT_BINS 180
#define LON_BINS 360
#define TIME_BINS 120
#define TILE_SIZE 32  //process 32x32x32 tiles

typedef struct {
    long timestamp;
    float latitude;
    float longitude;
    float magnitude;
} EarthquakeEvent;

__device__ void compute_bin_indices(EarthquakeEvent event, long min_time, long max_time,
                                    int *lat_idx, int *lon_idx, int *time_idx) {
    *lat_idx = (int)((event.latitude + 90.0f) * LAT_BINS / 180.0f);
    if (*lat_idx >= LAT_BINS) *lat_idx = LAT_BINS - 1;
    if (*lat_idx < 0) *lat_idx = 0;

    *lon_idx = (int)((event.longitude + 180.0f) * LON_BINS / 360.0f);
    if (*lon_idx >= LON_BINS) *lon_idx = LON_BINS - 1;
    if (*lon_idx < 0) *lon_idx = 0;

    *time_idx = (int)((event.timestamp - min_time) * TIME_BINS /
                      (double)(max_time - min_time + 1));
    if (*time_idx >= TIME_BINS) *time_idx = TIME_BINS - 1;
    if (*time_idx < 0) *time_idx = 0;
}

__device__ float atomicMaxFloat(float* address, float val) {
    int* address_as_int = (int*)address;
    int old = *address_as_int, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_int, assumed,
                        __float_as_int(fmaxf(val, __int_as_float(assumed))));
    } while (assumed != old);
    return __int_as_float(old);
}

// tiled kernel,process events that belong to current tile
__global__ void histogram_kernel_tiled(EarthquakeEvent *events, int n_events,
                                       int *counts, float *mags, float *maxs,
                                       long min_time, long max_time,
                                       int tile_lat_start, int tile_lon_start, int tile_tim
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n_events) {
        int lat_idx, lon_idx, time_idx;
```

```
            compute_bin_indices(events[idx], min_time, max_time,
                                &lat_idx, &lon_idx, &time_idx);

            //check if event belongs to current tile
            if (lat_idx >= tile_lat_start && lat_idx < tile_lat_start + TILE_SIZE &&
                lon_idx >= tile_lon_start && lon_idx < tile_lon_start + TILE_SIZE &&
                time_idx >= tile_time_start && time_idx < tile_time_start + TILE_SIZE) {

                int cell_idx = lat_idx * LON_BINS * TIME_BINS + lon_idx * TIME_BINS + time_idx;

                atomicAdd(&counts[cell_idx], 1);
                atomicAdd(&mags[cell_idx], events[idx].magnitude);
                atomicMaxFloat(&maxs[cell_idx], events[idx].magnitude);
            }
        }
    }

    int load_binary_data(const char *filename, EarthquakeEvent **events) {
        FILE *fp = fopen(filename, "rb");
        if (!fp) {
            perror("Error opening file");
            return -1;
        }

        int n_events;
        fread(&n_events, sizeof(int), 1, fp);

        *events = (EarthquakeEvent *)malloc(n_events * sizeof(EarthquakeEvent));
        if (!*events) {
            fprintf(stderr, "Memory allocation failed\\n");
            fclose(fp);
            return -1;
        }

        for (int i = 0; i < n_events; i++) {
            fread(&(*events)[i], sizeof(EarthquakeEvent), 1, fp);
        }

        fclose(fp);
        return n_events;
    }

    void write_histogram_csv(const char *filename, int *counts, float *mags, float *maxs) {
        FILE *fp = fopen(filename, "w");
        if (!fp) {
            perror("Error opening output file");
            return;
        }

        fprintf(fp, "lat_bin,lon_bin,time_bin,count,total_magnitude,max_magnitude\\n");

        for (int lat = 0; lat < LAT_BINS; lat++) {
            for (int lon = 0; lon < LON_BINS; lon++) {
                for (int time = 0; time < TIME_BINS; time++) {
                    int idx = lat * LON_BINS * TIME_BINS + lon * TIME_BINS + time;
                    if (counts[idx] > 0) {
                        fprintf(fp, "%d,%d,%d,%d,%.2f,%.2f\\n",
                                lat, lon, time, counts[idx], mags[idx], maxs[idx]);
```

```
                    }
                }
            }
        }

        fclose(fp);
    }

    int main(int argc, char *argv[]) {
        if (argc != 2) {
            fprintf(stderr, "Usage: %s <binary_data_file>\\n", argv[0]);
            return 1;
        }

        int block_size = 256;

        EarthquakeEvent *h_events;
        int n_events = load_binary_data(argv[1], &h_events);

        if (n_events <= 0) {
            fprintf(stderr, "Failed to load data\\n");
            return 1;
        }

        printf("=== CUDA Tiling Implementation ===\\n");
        printf("Loaded %d earthquake events\\n", n_events);

        long min_time = h_events[0].timestamp;
        long max_time = h_events[0].timestamp;
        for (int i = 1; i < n_events; i++) {
            if (h_events[i].timestamp < min_time) min_time = h_events[i].timestamp;
            if (h_events[i].timestamp > max_time) max_time = h_events[i].timestamp;
        }

        int total_cells = LAT_BINS * LON_BINS * TIME_BINS;

        //calculate nb of tiles
        int num_lat_tiles = (LAT_BINS + TILE_SIZE - 1) / TILE_SIZE;
        int num_lon_tiles = (LON_BINS + TILE_SIZE - 1) / TILE_SIZE;
        int num_time_tiles = (TIME_BINS + TILE_SIZE - 1) / TILE_SIZE;
        int total_tiles = num_lat_tiles * num_lon_tiles * num_time_tiles;

        printf("Grid dimensions: %d x %d x %d = %d cells\\n",
               LAT_BINS, LON_BINS, TIME_BINS, total_cells);
        printf("Tile size: %d x %d x %d = %d cells per tile\\n",
               TILE_SIZE, TILE_SIZE, TILE_SIZE, TILE_SIZE * TILE_SIZE * TILE_SIZE);
        printf("Number of tiles: %d x %d x %d = %d tiles\\n",
               num_lat_tiles, num_lon_tiles, num_time_tiles, total_tiles);

        EarthquakeEvent *d_events;
        int *d_counts;
        float *d_mags, *d_maxs;

        cudaMalloc(&d_events, n_events * sizeof(EarthquakeEvent));
        cudaMalloc(&d_counts, total_cells * sizeof(int));
        cudaMalloc(&d_mags, total_cells * sizeof(float));
        cudaMalloc(&d_maxs, total_cells * sizeof(float));
```

```
cudaMemset(d_counts, 0, total_cells * sizeof(int));
cudaMemset(d_mags, 0, total_cells * sizeof(float));
cudaMemset(d_maxs, 0, total_cells * sizeof(float));

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
cudaMemcpy(d_events, h_events, n_events * sizeof(EarthquakeEvent), cudaMemcpyHostToDevi

int grid_size = (n_events + block_size - 1) / block_size;

//process each tile seperately
for (int lat_tile = 0; lat_tile < num_lat_tiles; lat_tile++) {
    for (int lon_tile = 0; lon_tile < num_lon_tiles; lon_tile++) {
        for (int time_tile = 0; time_tile < num_time_tiles; time_tile++) {
            int tile_lat_start = lat_tile * TILE_SIZE;
            int tile_lon_start = lon_tile * TILE_SIZE;
            int tile_time_start = time_tile * TILE_SIZE;

            histogram_kernel_tiled<<<grid_size, block_size>>>(
                d_events, n_events, d_counts, d_mags, d_maxs,
                min_time, max_time,
                tile_lat_start, tile_lon_start, tile_time_start);
        }
    }
}

cudaDeviceSynchronize();
cudaEventRecord(stop);
cudaEventSynchronize(stop);

float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);

printf("\\nExecution time: %.5f ms\\n", milliseconds);

int *h_counts = (int *)malloc(total_cells * sizeof(int));
float *h_mags = (float *)malloc(total_cells * sizeof(float));
float *h_maxs = (float *)malloc(total_cells * sizeof(float));

cudaMemcpy(h_counts, d_counts, total_cells * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(h_mags, d_mags, total_cells * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(h_maxs, d_maxs, total_cells * sizeof(float), cudaMemcpyDeviceToHost);

int non_empty = 0;
int max_count = 0;
long long total_events = 0;
for (int i = 0; i < total_cells; i++) {
    if (h_counts[i] > 0) {
        non_empty++;
        total_events += h_counts[i];
        if (h_counts[i] > max_count) max_count = h_counts[i];
    }
}

printf("Non-empty cells: %d (%.2f%%)\\n",
```

```
               non_empty, 100.0 * non_empty / total_cells);
        printf("Max events in single cell: %d\\n", max_count);
        printf("Total events processed: %lld\\n\\n", total_events);

        write_histogram_csv("histogram_cuda_tiled.csv", h_counts, h_mags, h_maxs);
        printf("Histogram saved to histogram_cuda_tiled.csv\\n");

        free(h_events);
        free(h_counts);
        free(h_mags);
        free(h_maxs);
        cudaFree(d_events);
        cudaFree(d_counts);
        cudaFree(d_mags);
        cudaFree(d_maxs);
        cudaEventDestroy(start);
        cudaEventDestroy(stop);

        return 0;
    }
    """
```

```
!nvcc -arch=sm_75 -o cuda_tiled cuda_tiled.cu
!echo "CUDA with tiling:"
!./cuda_tiled earthquake_data.bin
```

```
CUDA with tiling:
=== CUDA Tiling Implementation ===
Loaded 78404 earthquake events
Grid dimensions: 180 x 360 x 120 = 7776000 cells
Tile size: 32 x 32 x 32 = 32768 cells per tile
Number of tiles: 6 x 12 x 4 = 288 tiles

Execution time: 8.14282 ms
Non-empty cells: 45499 (0.59%)
Max events in single cell: 467
Total events processed: 78404

Histogram saved to histogram_cuda_tiled.csv
```