Presented to the College of Computer Studies

De La Salle University - Manila

Term 3, A.Y. 2023-2024

In partial fulfillment of the course

In CEPARCO S11

Data-level Parallelism Integrating Project Update (Milestone 3):

MOVEMENT RECOGNITION IN SIMT

Group No. 5

Submitted by:

Cai, Edison B.

Dequico, Beverly Joyce P.

La'O, Erin Denise C.

Relucio, Jan Jhezaree L.

Submitted to:

Prof. Roger Luis Uy

July 30, 2024

PROJECT OVERVIEW

The objective of this project is to implement movement recognition using data-level parallelism to improve the efficiency and speed of processing accelerometer data. By leveraging CUDA and parallel programming techniques, the aim is to achieve significant performance enhancements over traditional sequential methods.

For this particular project, the main focus will be on the R-squared regression statistic to evaluate the relationship between one axis and the other two.

$$R^2 = 1 - \frac{\Sigma(\widehat{y_i} - \bar{y})^2}{\Sigma(y_i - \bar{y})^2}$$

The R-squared value is calculated as the sum of the squares of the predicted values minus the mean, divided by the sum of the squares of the actual values minus the mean. R-squared values range from 0 to 1, where values closer to 1 indicate a better fit of the model to the data.

PROGRAM FLOW

I. **Data Generation and Preprocessing**

- **Data Retrieval:**
  - The data can be obtained from a csv file, specifically extracting columns related to the **x**, **y**, and **z** axes for the acceleration data. Users can upload their own files to the Google Colab session and specify the file name so the system can read data from it.
  - The data can also be manually entered by the user. It must be an array of arrays. This format was done to accommodate data obtained from accelerometer recordings done via Edge Impulse for example.
- **Input Validation and Preparation:**
  - In the event that the user chooses to input a .csv file, then the system will only proceed with attempting to open the file when the file name includes the extension (.csv). Should the system be unable to open the file (a reason is that the file does not exist), the system will exit.
  - In the event that the user chooses to manually enter an array of arrays, the input format is validated by looping through the whole input, ensuring it starts and ends with square brackets (“[“ and “]”)

and has a balanced number of square brackets. Additionally, the data is checked for non-numeric values to maintain data integrity.

- **Outcome:**
  - The data is successfully validated and prepared for further processing, ensuring its integrity for analysis.

II. Multiple Linear Regression Implementation

- Current Implementation of basic input handling and logic flow.
  - Utilizes standard input functions to manage user input and basic control flow structures.
  - Developed foundational logic for handling user input and setting up initial processing parameters.
- Program Steps:
  - Input Validation Loop
    - A loop function is used to validate the input format, ensuring it is an array of arrays by checking that it begins and ends with square brackets.
    - It also checks for an equal number of opening square brackets and closing square brackets.
    - It is then checked for non-numeric characters within the input to ensure the validity of the input data and to maintain data integrity.
  - Number of Sets Input
    - Prompts the user to input the index of the set to be processed for analysis.
    - Stores the input value in inputIndex for further processing.
  - Logic
    - Process the input data (inputArr) based on the specified index of the set (inputIndex).
    - Function for R-squared is called. The function calculations are based on the formula provided under Project Overview.

III. R-Squared Calculation

The R-Squared is used to determine if the specified set of coordinates may exhibit linear movement. The acceptable threshold for linear movement has been coded to be above 0.9 or around 90%.

IV. Code Testing and Verification

- Testing Environment: Initial tests were conducted in Google Colab to verify code functionality and performance for the input array.
- Details: Code input retrieval from the user to obtain the input array and the number of sets to the group.
- Outcome: The code performs well in the Colab environment.

FINAL IMPLEMENTATIONS OF THE PROJECT

V. SEQUENTIAL IMPLEMENTATION:

A. C Implementation: The sequential implementation is written in C. It reads accelerometer data and computes the R-squared value to evaluate the model's performance. The code processes input data, checks for errors, extracts coordinates, and calculates the R-squared value sequentially.

VI. PARALLEL IMPLEMENTATION:

A. CUDA-no Prefetch: This CUDA implementation parallelizes the computation of the R-squared value by distributing the workload across multiple GPU threads. It does not use data prefetching, which means the data is directly accessed from the global memory.

B. CUDA-prefetch: Implemented CUDA with data prefetching for optimized performance. This CUDA implementation parallelizes the computation of the R-squared value by distributing the workload across multiple GPU threads. It does not use data prefetching, which means the data is directly accessed from the global memory.

VII. *Highlight your implementation. Compare and contrast with existing implementation.*

Sequential Implementation

Description:

- Implemented in C.
- Computes the R-squared regression statistic for accelerometer data.
- Sequentially processes the data by calculating sums of products and squares needed for the regression formula.

Key Code Snippet:

```
float sum_x = 0, sum_y = 0, sum_xy = 0, sum_x2 = 0, sum_y2 = 0;

for (size_t i = 0; i < ARRAY_SIZE; i++) {
    sum_x += x[i];
    sum_y += y[i];
    sum_xy += x[i] * y[i];
    sum_x2 += x[i] * x[i];
    sum_y2 += y[i] * y[i];
}

float numerator = (ARRAY_SIZE * sum_xy) - (sum_x * sum_y);
float denominator = sqrtf((ARRAY_SIZE * sum_x2 - sum_x * sum_x) *
                          (ARRAY_SIZE * sum_y2 - sum_y * sum_y));
if (denominator != 0) {
    r_square = powf(numerator / denominator, 2);
} else {
    r_square = 0;
}
```

Parallel Implementation: CUDA without Prefetching

Description:

- Utilizes CUDA for parallel processing.
- Computes the R-squared regression statistic using GPU.
- Uses shared memory to reduce global memory accesses and improve performance.

Key Code Snippet:

```
__global__
void rSquare(size_t n, float *r_square, float *predict_y, float *actual_y) {
    extern __shared__ float shared_mem[];
    float *sum_x = &shared_mem[0];
    float *sum_y = &shared_mem[1];
    float *sum_xy = &shared_mem[2];
    float *sum_x2 = &shared_mem[3];
    float *sum_y2 = &shared_mem[4];

    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
```

```
    if (threadIdx.x == 0) {
        *sum_x = 0;
        *sum_y = 0;
        *sum_xy = 0;
        *sum_x2 = 0;
        *sum_y2 = 0;
    }

    __syncthreads();

    for (int i = index; i < n; i += stride) {
        atomicAdd(sum_x, predict_y[i]);
        atomicAdd(sum_y, actual_y[i]);
        atomicAdd(sum_xy, predict_y[i] * actual_y[i]);
        atomicAdd(sum_x2, predict_y[i] * predict_y[i]);
        atomicAdd(sum_y2, actual_y[i] * actual_y[i]);
    }

    __syncthreads();

    if (threadIdx.x == 0) {
        float numerator = (n * (*sum_xy)) - (*sum_x * *sum_y);
        float denominator = sqrtf((n * (*sum_x2) - (*sum_x) * (*sum_x)) *
                                  (n * (*sum_y2) - (*sum_y) * (*sum_y)));
        if (denominator != 0) {
            *r_square = powf(numerator / denominator, 2);
        } else {
            *r_square = 0;
        }
    }
}
```

Parallel Implementation: CUDA with Prefetching

Description:

- Enhances the CUDA implementation by adding data prefetching.
- Aims to further optimize memory access efficiency and reduce latency.

VIII. How much percentage of the sequential implementation is original? How much percentage of the parallel implementation is original?

Comparison with Existing Implementations

Sequential Implementation

- **Our Implementation:**
  - Straightforward calculation of R-squared using loops and basic arithmetic.
  - Focuses on clarity and correctness of the algorithm.
- **Existing Implementations:**
  - Similar sequential approaches are common, often used as a baseline for performance comparisons.

Parallel Implementation

- **Our Implementation (CUDA without Prefetching):**
  - Utilizes shared memory and atomic operations to manage sums efficiently.
  - Designed to leverage the parallel nature of GPUs for performance gains.
- **Existing Implementations:**
  - Many parallel implementations use similar strategies with CUDA or other parallel computing frameworks.
  - Advanced implementations might include more sophisticated memory management, kernel optimizations, and dynamic adjustments based on data characteristics.
- **Our Implementation (CUDA with Prefetching):**
  - Adds prefetching to further optimize memory access patterns.
  - Aims to reduce latency by preloading data into cache.
- **Existing Implementations:**
  - Prefetching is a common optimization technique in high-performance computing.
  - Advanced implementations may use a combination of prefetching, caching strategies, and kernel optimizations.

Originality Percentage

Sequential Implementation

- **Originality:**
  - Basic R-squared calculation is a well-known method in statistics.
  - Our implementation follows standard practices, focusing on clarity and correctness.
- **Estimated Originality Percentage:**

- Approximately 90%. The core algorithm is standard, the specific implementation details and style are original to our project, and the reference was commented in the code for the timer segment.

Parallel Implementation

- **Originality:**
  - Uses standard CUDA practices like shared memory and atomic operations.
  - Prefetching technique is a known optimization but applied uniquely to our problem.
- **Estimated Originality Percentage:**
  - Without Prefetching: Approximately 90%. While using common CUDA techniques, the specific application and implementation details are tailored to our project.
  - With Prefetching: Approximately 90%. Adding prefetching introduces more originality, as the optimization strategy is tailored to improve our specific use case.

VII. (RESULTS) Execution time comparison between sequential and parallel

(CUDA without Prefetching): Avg. Execution Time is at **386.06 us**

```
%%shell
nvcc CUDA_movement.cu -o CUDA_movement
nvprof ./CUDA_movement
```

```
Enter file name (extension included): xyz.csv
Enter the index number: 2
==907== NVPROF is profiling process 907, command: ./CUDA_movement
*** rSquare function ***
numElements = 50
numBlocks = 1, numThreads = 1024
R^2: 0.992948
==907== Profiling application: ./CUDA_movement
==907== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  388.06us         1  388.06us  388.06us  388.06us  rSquare(unsigned long, float*, float*, float*)
      API calls:   99.52%  237.98ms         3  79.326ms  8.8680us  237.94ms  cudaMallocManaged
                    0.16%  393.08us         1  393.08us  393.08us  393.08us  cudaDeviceSynchronize
                    0.14%  330.30us         1  330.30us  330.30us  330.30us  cudaLaunchKernel
                    0.09%  218.98us       114  1.9200us     196ns  78.903us  cuDeviceGetAttribute
                    0.07%  166.12us         3  55.373us  11.767us  120.81us  cudaFree
                    0.01%  21.730us         1  21.730us  21.730us  21.730us  cuDeviceGetName
                    0.00%  7.8520us         1  7.8520us  7.8520us  7.8520us  cuDeviceGetPCIBusId
                    0.00%  7.0050us         1  7.0050us  7.0050us  7.0050us  cuDeviceTotalMem
                    0.00%  2.6520us         3     884ns     286ns  2.0600us  cuDeviceGetCount
                    0.00%  1.0240us         2     512ns     229ns     795ns  cuDeviceGet
                    0.00%     388ns         1     388ns     388ns     388ns  cuModuleGetLoadingMode
                    0.00%     358ns         1     358ns     358ns     358ns  cuDeviceGetUuid

==907== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  10.78400us  Host To Device
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  8.480000us  Device To Host
       1         -         -         -           -  379.6770us  Gpu page fault groups
Total CPU Page faults: 2
```

(CUDA with Prefetching): Avg. Execution Time is at **400.18us**

```
%%shell
nvcc CUDA_movement.cu -o CUDA_movement
nvprof ./CUDA_movement
```

```
Enter file name (extension included): xyz.csv
Enter the index number: 2
==535== NVPROF is profiling process 535, command: ./CUDA_movement
*** rSquare function ***
numElements = 60
numBlocks = 1, numThreads = 1024
R^2: 0.992948
==535== Profiling application: ./CUDA_movement
==535== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  400.18us         1  400.18us  400.18us  400.18us  rSquare(unsigned long, float*, float*, float*)
      API calls:   63.89%  241.02ms         3  80.339ms  5.7660us  240.98ms  cudaMallocManaged
                   35.66%  134.52ms         1  134.52ms  134.52ms  134.52ms  cudaLaunchKernel
                    0.24%  896.28us         3  298.76us     903ns  892.26us  cudaMemPrefetchAsync
                    0.11%  404.11us         1  404.11us  404.11us  404.11us  cudaDeviceSynchronize
                    0.06%  212.32us         3  70.772us  12.544us  160.09us  cudaFree
                    0.03%  130.24us       114  1.1420us     138ns  51.026us  cuDeviceGetAttribute
                    0.00%  15.861us         1  15.861us  15.861us  15.861us  cuDeviceGetName
                    0.00%  12.373us         2  6.1860us     561ns  11.812us  cudaMemAdvise
                    0.00%  6.1220us         1  6.1220us  6.1220us  6.1220us  cuDeviceGetPCIBusId
                    0.00%  3.4850us         1  3.4850us  3.4850us  3.4850us  cuDeviceTotalMem
                    0.00%  2.3990us         1  2.3990us  2.3990us  2.3990us  cudaGetDevice
                    0.00%  1.4660us         3     488ns     197ns  1.0480us  cuDeviceGetCount
                    0.00%  1.2430us         2     621ns     216ns  1.0270us  cuDeviceGet
                    0.00%     477ns         1     477ns     477ns     477ns  cuModuleGetLoadingMode
                    0.00%     242ns         1     242ns     242ns     242ns  cuDeviceGetUuid

==535== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  10.91200us  Host To Device
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  7.872000us  Device To Host
       1         -         -         -           -  391.5120us  Gpu page fault groups
Total CPU Page faults: 2
```

VIII. Conclusion

CUDA does best, especially when dealing with larger amounts of data as parallelizing the system means that the program can perform efficiently by distributing workload to threads and ensuring that the whole process can be done in less time compared to sequential execution.