Optimizing GEMM and SpMV Operations on Parallel Hardware

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Thanmayee Gunja
(112001013)

COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the project entitled "**Optimizing GEMM and SpMV Operations on Parallel Hardware**" is a bonafide work of **Thanmayee Gunja** (**Roll No. 112001013**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under the guidance of **Dr. Unnikrishnan Cheramangalath** and that it has not been submitted elsewhere for a degree.*

Dr. Unnikrishnan Cheramangalath

Assistant/Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

I would like to express my sincere gratitude to my mentor Dr.Unnikrishnan Cheramangalath for providing valuable guidance, comments, and suggestions throughout the project.

Contents

List of Figures

Chapter 1

Introduction

The efficient handling of large sets of data is extremely important for a wide range of applications, from scientific simulations to artificial intelligence. Matrix computation is a key element across various fields and provides a structured way of organizing and manipulating data, helping solve problems ranging from simple algebraic equations to complex machine learning algorithms.

Matrix operations such as general matrix multiplication(GEMM) and sparse matrix vector multiplication (SpMV) are the basic building blocks of many computational tasks. The performance of these operations directly affects the efficiency and scalability of different computational processes.

1.1 Motivation

The increasing demand for faster and more efficient computational solutions in diverse domains is the motivation for accelerating matrix multiplication on parallel hardware. Matrix multiplication, being a computation-intensive algorithm, represents a huge bottleneck in system overall performance. With the increasing use of Graphics Processing Units(GPUs), there has been a lot of interest in using their powerful parallel processing to speed up

matrix operations. Efficient matrix operations have a direct impact on how well systems handle complex calculations, so making them work as fast as possible is really important for unlocking new possibilities in real-time.

1.2 Problem Statement

Optimizing parallel algorithms to accelerate matrix multiplication (GEMM) and sparse matrix vector multiplication(SpMV) on GPUs and CPUs. The project seeks to address the performance limitations of traditional sequential approaches by taking advantage of the extensive parallel processing power of GPUs to significantly increase the speed and make better use of resources.

The project goal is to evaluate and compare the performance of GPU implementations with multi - threaded CPU implementations.

1.3 Objectives

1. Develop Parallel Algorithms: Design and implement parallel algorithms for GEMM and SpMV suitable for execution on GPUs.

2. Optimize Algorithm Performance: Explore optimization techniques such as memory coalescing, shared memory utilization, and thread cooperation to maximize the efficiency of the parallel algorithms on GPU architectures.

3. Implement Sequential/Naive and Multi-threaded CPU Versions: Develop sequential and multi-threaded CPU implementations of the matrix multiplication and sparse matrix vector multiplication algorithms for comparison purposes.

4. Performance Evaluation: Measure and compare the performance of GPU-accelerated implementations with sequential and multi-threaded CPU implementations.

These objectives provide a structured approach to achieving the project goals and ensure all the key aspects of algorithm development, optimization, and performance evaluation.

1.4 Mid-term Summary

During the initial phase of this project, significant progress has been made in understanding the fundamental concepts and challenges associated with accelerating matrix multiplication on parallel hardware architectures. The implementation phase delved into the architecture overview of NVIDIA GPUs, the CUDA programming model, the memory hierarchy, and optimization strategies tailored for CUDA-enabled GPUs. The NVIDIA T4 GPU was selected as the hardware platform for accelerating matrix multiplication tasks due to its significant computational power and suitability for parallel processing tasks. Detailed code snippets and explanations were provided for GPU and CPU implementations of matrix multiplication(GEMM) algorithms, laying the groundwork for performance evaluation and comparison.

In the next phase, the focus shifted towards optimizing sparse matrix-vector multiplication(SpMV) operations, building upon the foundational knowledge gained from the GEMM phase.

Chapter 2

Literature Review

2.1 Introduction to Performance Engineering and Evolution of Optimization Techniques

Performance engineering is a vital aspect of software development that influences the usability and effectiveness of computing systems. Analogous to smart money management, the optimization of program execution liberates computational resources for strategic pursuits such as usability, enhancement, and computational precision. Matrix multiplication, a fundamental operation in various computational tasks, serves as an illustrative example of the importance of performance optimization.

Over time, optimization techniques have evolved alongside advancements in hardware. The end of Dennard scaling in 2004 marked a renewed focus on the importance of performance engineering, necessitating efficient software solutions tailored to modern hardware configurations.

It is clear that unoptimized software strains resources and leads to suboptimal system performance, highlighting the need for effective performance engineering techniques. To fully utilize the computational capacity of modern hardware, parallel programming approaches had to be adopted to transition to multi-core processors. Traditional programming

languages like Python exhibit performance limitations compared to compiler languages like C, highlighting Python's slower execution speed. This underscores the importance of architecture-aware optimizations.

2.2 GPU

A Graphic Processing Unit(GPU) is a type of specialized electronic chip designed to process data and perform calculations quickly in parallel. GPUs are faster and more efficient than CPUs, especially when it comes to processing large amounts of data. In contrast to CPUs, which usually have a small number of cores that can only handle a few threads at a time, GPUs have hundreds of cores that can execute thousands of threads in parallel, making data computations faster and more effective.

2.2.1 Architectural Differences Between CPUs and GPUs

On CPUs, two fundamental optimizations are important: increasing the cache hit rate of memory access and using SIMD units (Single Instruction, Multiple Data) for vector processing. Using the CPU cache structure and parallel execution capabilities, these improvements aim to increase computational throughput. The architecture of GPUs differ significantly from that of CPUs, particularly in terms of memory organization. GPUs incorporate multiple memory types, each serving specific purposes within the computation process. The hierarchy includes:

- Global Memory: The slowest type of memory with read/write capability. It is accessible to every thread of the grid.
- Shared Memory: All threads inside a block have access to shared memory, which is allocated to each block. It offers faster access than constant memory, yet being slower than registers.

- Constant Memory: Compared to global memory, constant memory offers faster read-only access. It is available to all threads within the grid, just as global memory.
- Local Memory: Automatic variables that are too large to fit into registers are stored in the local memory space, which is part of global memory. Local memory accesses are less common since they can only be accessed by the active thread and have latency and bandwidth characteristics similar to those of global memory.
- Registers: The quickest memory type to access and is used to store automated variables. Only the active thread can access them.

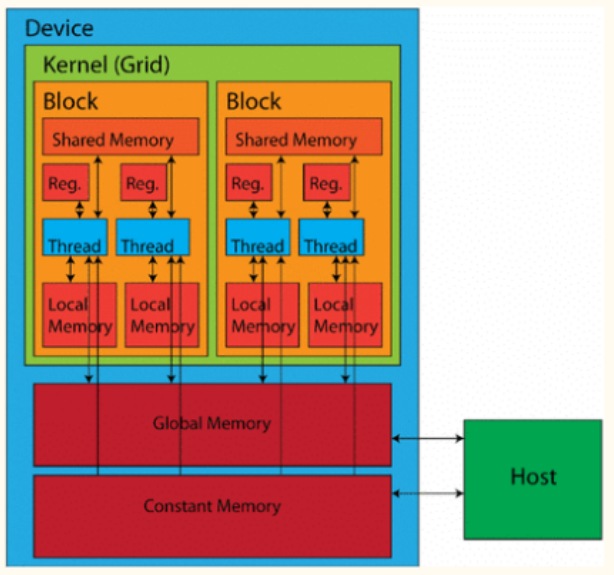

Fig. 2.1 The memory hierarchy of GPU [1]

2.2.2 Components of NVIDIA GPU

An NVIDIA GPU comprises three primary components:

- Processing Clusters(PC): Consists of clusters of Streaming Multiprocessors.
- Streaming Multiprocessors(SM): Contains multiple processor cores and a layer-1 cache.
- Layer-2 cache: Shared cache connecting SMs together.

- DRAM: Global memory holding instructions processed by all SMs.

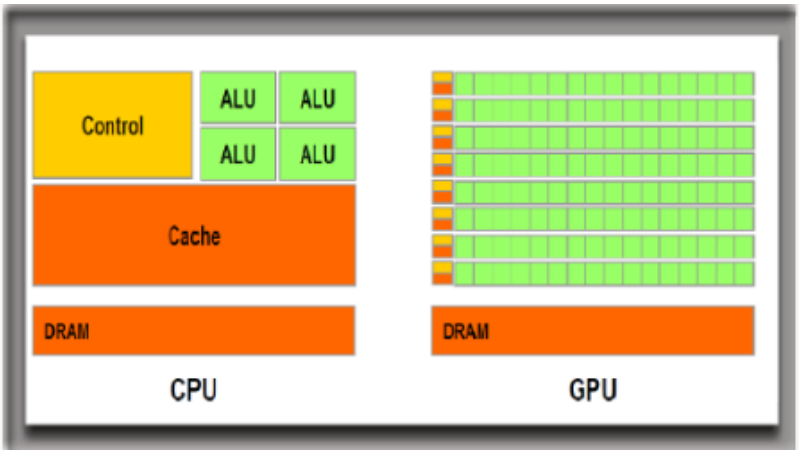

Fig. 2.2 Architectural difference [2]

Considering these architectural differences, it becomes clear why CPUs may not be the best choice for parallel programming tasks. Their limited number of cores and slower memory bandwidth make them less efficient for handling massively parallel tasks compared to GPUs, which are specifically designed for such workloads.

2.3 CUDA

A programming model and a parallel computing platform that enables developers to access the computational power of NVIDIA GPUs to accelerate a wide range of parallel computing tasks. Abstracts the GPU architecture into three main components: a hierarchy of threads, thread blocks, and grids. Individual tasks that can be completed in parallel are known as threads, and they are arranged into blocks that can communicate and coordinate with each other. Multiple thread blocks form a grid, providing a scalable way to execute large numbers of threads across the GPU at once.

The process of utilizing CUDA kernels typically involves the following steps:

1. Allocation of GPU Storage: Memory is allocated from the CPU to the GPU utilizing `cudaMalloc()`.

2. Data Transfer: Input data are transferred from the CPU to the GPU using `cudaMemcpy()`.

3. Kernel Launch: The CPU processes the GPU data by launching numerous copies of the kernel on simultaneous threads. Parallel computing is done by a kernel, which is simply a GPU function.

4. Result retrieval: Once the computation is complete, the CPU retrieves the results from the GPU using `cudaMemcpy()`.

5. Output Utilization: Finally, the output obtained from the GPU can be used or displayed as required.

2.3.1 CUDA API functions

CUDA provides a rich set of API functions and libraries that developers can leverage to optimize their GPU-accelerated applications. Some CUDA API functions are as follows:

1. Device Management: Functions like `cudaGetDeviceCount` and `cudaSetDevice` are used to manage CUDA enabled devices and select the current device for computation.

2. Memory Management: Functions such as `cudaMalloc`, `cudaMemcpy`, `cudaMemset`, and `cudaFree` are used to allocate, transfer, and store memory on the GPU device.

3. Kernel Launch: Kernels, which are functions designed to run in parallel on the GPU, are launched using syntax like `<<<...>>>` to specify execution configuration, and the `cudaLaunchKernel` function.

4. Synchronization and Error Handling: Synchronization functions like `syncthreads()` and `cudaDeviceSynchronize()` are used to coordinate threads and synchronize the host thread with the CUDA device.

2.4 General Matrix Multiply(GEMM)

GEMM, a fundamental operation in linear algebra and computational mathematics, involves multiplying two matrices to produce a third matrix. The general form of GEMM can be represented as:

$$C = \alpha AB + \beta C$$

where A, B, and C are matrices, and α and β are scalar constants. Many scientific simulations, numerical computations, and machine learning algorithms respond heavily to efficient matrix multiplication operations. Optimizing GEMM directly translates into improved performance and responsiveness of these applications, enabling faster insight and decision-making.

2.5 Sparse Matrix-Vector Multiplication (SpMV)

SpMV is commonly expressed as $y = Ax$,where

- A is a sparse matrix of size $m \times n$, where m and n represent the number of rows and columns respectively.
- x is a dense vector of length n.
- y is the dense vector of length m that results, after the multiplication.

Compared to dense matrices, a sparse matrix has more zero-entries, making computation and representation more difficult. However, SpMV algorithms can improve the multiplication process to eliminate needless operations with zero members, hence lowering computational cost because of the sparsity of the matrix.

Efficient SpMV operation is crucial for enhancing the performance of scientific applications, given its widespread use in solving linear systems and eigenvalue problems.

2.5.1 Choice of CSR format for SpMV

Sparse matrices can be represented in various formats, including CSR, CSC (Compressed Sparse Column), COO (Coordinate List), etc. The selection of the CSR format for SpMV is often driven by several factors:

1. Memory Efficiency:

    - Compact Representation: CSR stores only the non-zero elements and their corresponding indices, resulting in a more memory-efficient representation compared to dense matrices or other sparse formats.
    - Reduced Overhead: The absence of explicit zero entries minimizes memory overhead, making CSR particularly suitable for large, sparsely populated matrices common in scientific simulations and engineering applications.

2. Computational Efficiency:

    - Optimized Matrix-Vector Multiplication: CSR's structure enables efficient traversal of non-zero elements during SpMV, reducing computational complexity and improving performance.

3. Application Suitability:

    - Iterative Solvers: CSR is commonly used in iterative solvers like Conjugate Gradient (CG) and Generalized Minimal Residual (GMRES), which are widely employed in solving large linear systems arising from finite element simulations, computational fluid dynamics, and other scientific computations.
    - Graph Algorithms: CSR is also well-suited for graph algorithms such as PageRank, graph traversal, and network analysis, where sparse matrix operations are prevalent.

The CSR format compresses a sparse matrix into three vectors:

- The **Value** vector: Contains all the non-zero entries.
- The **Column** vector: Contains the column index of each non-zero entry.
- The **RowPtr** vector: Contains the index of the first non-zero entry of each row in the "Value" vector. We add the number of non-zero entries in the sparse matrix as the last element of the RowPtr vector.

2.6 OpenMP

C, C++, and Fortran shared memory multiprocessing programming is supported by the OpenMP (Open Multiprocessing) application programming interface (API). It makes it possible for programmers to create concurrent programs that run on several CPU cores on a single system. OpenMP provides a set of compiler directives, run-time library routines, and environment variables that facilitate parallel programming. OpenMP is widely adopted in scientific computing, as it simplifies parallel programming by allowing developers to add parallelism to their existing codebase with minimal changes.

2.6.1 Features

- Thread Management: Handles the creation and management of threads automatically, abstracting away low-level thread management details and making parallel programming more accessible to developers.
- Work-sharing constructs: OpenMP provides constructs for distributing loop iterations and other computational tasks among multiple threads, such as `parallel for` directive for parallel loop execution.
- Synchronization: Offers mechanisms for synchronizing threads and managing data sharing between threads, including partitioning barriers, atomic operations, and shared variables.

OpenMP is designed for shared memory multiprocessing and is not suitable for distributed memory parallelism or GPU acceleration.

2.7 Tiling

The foundation of machine learning and graphic rendering is the speed with which matrix multiplication can be computed. Using shared memory on the GPU, tiling helps minimize global memory accesses. One technique to improve kernel execution performance is by tiling. In order to fit the data into the cache memory more effectively, it is divided into smaller, continuous blocks or tiles. Tiling reduces the need to access data from slower memory tiers, such as RAM or disk, by processing these smaller chunks consecutively. This reduces memory latency and enhances overall speed. In addition to improving cache usage, tiling makes parallelization and vectorization easier and allows for optimization on a variety of hardware architectures, such as CPUs and GPUs.

Chapter 3

Implementation

3.1 Hardware Configuration

In this project, we will utilize the NVIDIA T4 GPU to accelerate matrix multiplication tasks. The T4 GPU offers significant computational power and is well suited for parallel processing tasks

3.1.1 Device Specifications

Upon querying the NVIDIA T4 GPU, the following specifications were obtained:

- Number of CUDA Cores: 2560
- Number of Streaming Multiprocessors (SMs): 40
- Number of Streaming Processors (SPs) per SM: 64

```
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:         7
Minor revision number:         5
Name:                          Tesla T4
Total global memory:           2950758400
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    1590000
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     40
Kernel execution timeout:      No
```

Fig. 3.1 CUDA Device Information

3.2 Pseudocode

3.2.1 GEMM

Algorithm 1: Matrix Multiplication Pseudo Code

Input : Matrices A, B of size $size \times size$, scalars α and β

Output: Matrix C of size $size \times size$

for $i \leftarrow 0$ **to** $size - 1$ **do**

  for $j \leftarrow 0$ **to** $size - 1$ **do**

    $c \leftarrow 0$;

    for $k \leftarrow 0$ **to** $size - 1$ **do**

      $c \leftarrow c + A(k, i) \times B(j, k)$;

    end

    $C(j, i) \leftarrow \alpha \times c + \beta \times C(j, i)$;

  end

end

3.2.2 SpMV

Algorithm 2: Sequential CSR SpMV Procedure

Data: CSR matrix properties: rowptr, values, colindices; Input vector x; Output vector y; Scalars α and β

Result: Vector y after SpMV operation

for $i \leftarrow 0$ **to** $R - 1$ **do**

  $sum \leftarrow 0$;

  for $j \leftarrow$ *rowptr*$[i]$ **to** *rowptr*$[i + 1] - 1$ **do**

    $sum \leftarrow sum +$ values$[j] \times x[$colindices$[j]]$;

  end

  $y[i] \leftarrow \alpha \times sum + \beta \times y[i]$;

end

3.3 Matrix Generation Strategy

The `generate_random_csr_matrix` function is used to create a random CSR matrix. This approach is chosen to simulate real-world scenarios where sparse matrices are common in scientific computing and other domains. Given the density of non-zero elements, the resulting matrix closely resembles the characteristics of real sparse matrices encountered in practical applications. Additionally, by using a random matrix, various matrix densities and sizes can be explored, allowing for a thorough examination of the algorithm's efficiency and scalability under various input conditions.

Algorithm 3: Basic Implementation of Generate Random CSR Matrix

Input: Integer $rows$, Integer $cols$, Float $density$

Output: CSR matrix

1. Create an empty CSR matrix structure
2. Set the number of rows and columns in the matrix
3. Allocate memory for row_ptr, values, and col_indices arrays
4. $max\_non\_zeros \leftarrow density \times rows \times cols$
5. Set the number of non-zero elements in the matrix to max_non_zeros
6. Initialize the row_ptr array:
   (a) $row\_ptr[0] \leftarrow 0$
   (b) For i from 0 to $rows - 1$:
      i. Determine the number of non-zero elements in the current row ($non\_zeros\_in\_row$)
      ii. $row\_ptr[i + 1] \leftarrow row\_ptr[i] + non\_zeros\_in\_row$
7. Populate the values and col_indices arrays:
   (a) $current\_index \leftarrow 0$
   (b) For i from 0 to $rows - 1$:
      i. For j from 0 to $non\_zeros\_in\_row - 1$:
         A. Generate a random value for the matrix element ($value$)
         B. Generate a random column index (col_index)
         C. Assign $value$ to $values[current\_index]$
         D. Assign col_index to $col\_indices[current\_index]$
         E. Increment $current\_index$
8. Return the generated CSR matrix structure

Chapter 4

Algorithm 1

4.1 CPU Implementation:

4.1.1 Single-threaded implementation:

Naive GEMM

The code utilizes a straightforward algorithm to perform matrix multiplication between two square matrices of size N. Memory allocation for matrices a, b and c is achieved by dynamic allocation using the `malloc` function. The core of the algorithm involves three nested loops to sequentially iterate over the resulting matrix c and computes its value by performing a dot product of the corresponding row in matrix a and column in matrix b.

```
// Function to perform matrix multiplication on CPU
void matrixMul(float *a, float *b, float *c) {
    for (int row = 0; row < N; ++row) {
        for (int col = 0; col < N; ++col) {
            float sum = 0.0f;
            // Iterate over each element of the row in 'a' and column in 'b'
            for (int i = 0; i < N; ++i) {
                // Accumulate the product of corresponding elements
                sum += a[row * N + i] * b[i * N + col];
            }
            // Store the sum in the corresponding position of matrix 'c'
            c[row * N + col] = sum;
        }
    }
}
```

Fig. 4.1 Naive GEMM Code Snippet

Naive SpMV

The single-threaded version of the SpMV CPU operation aims to demonstrate a basic SpMV operation, using the CSR format, crucial in numerical computing for various scientific and engineering applications. The code generates a random CSR matrix with a specified density of non-zero elements and initializes an input vector `x`. It sequentially computes the SpMV operation, iterating on each row of the matrix, and performing the dot product with the input vector `x`. The code specifies a simple, sequential approach to SpMV computation, suitable for small to moderate-sized matrices.

```
// Function to perform SpMV on CPU (single-threaded)
void cpu_csr_spmv(const csr_matrix* matrix, const float* x, float* y) {
    for (int i = 0; i < matrix->rows_count; ++i) {
        y[i] = 0.0f; // Initialize the result for this row
        for (int j = matrix->row_ptr[i]; j < matrix->row_ptr[i + 1]; ++j) {
            y[i] += matrix->values[j] * x[matrix->col_indices[j]]; // Perform dot product
        }
    }
}
```

Fig. 4.2 Naive SpMV Code Snippet

Random values for the non-zero elements of the matrix are generated uniformly between 0 and 1000 using the `rand` function seeded with current time `srand(time(NULL))`, which will ensure that each run of the program produces a different sequence of random numbers. This is important for generating unique random matrices in each execution, enhancing the variability and realism of the generated matrices across different experiments or simulations.

```
// Function to generate a random CSR matrix
csr_matrix generate_random_csr_matrix(int rows, int cols, float density) {
    csr_matrix matrix;
    matrix.rows_count = rows;
    matrix.cols_count = cols;
    matrix.row_ptr = (int*)malloc((rows + 1) * sizeof(int));

    // Generate random values for the matrix
    srand(time(NULL));

    int max_non_zeros = (int)(density * rows * cols);
    matrix.non_zero_count = max_non_zeros;
    matrix.values = (float*)malloc(max_non_zeros * sizeof(float));
    matrix.col_indices = (int*)malloc(max_non_zeros * sizeof(int));

    int current_index = 0;
    matrix.row_ptr[0] = 0;
    for (int i = 0; i < rows; ++i) {
        int non_zeros_in_row = (int)(density * cols); // Determine number of non-zeros in this row
        matrix.row_ptr[i + 1] = matrix.row_ptr[i] + non_zeros_in_row; // Update row_ptr
        for (int j = 0; j < non_zeros_in_row; ++j) {
            matrix.values[current_index] = (float)rand() / RAND_MAX * 1000.0f; // Random value between 0 and 1000
            matrix.col_indices[current_index] = rand() % cols; // Random column index
            ++current_index;
        }
    }

    return matrix;
}
```

Fig. 4.3 Random Matrix Generation

Performance Measurement of Naive GEMM and Naive SpMV on CPU

We measured the execution time of the matrix multiplication and SpMV operation on the CPU using the `clock()` function. This time represents the total time taken by the CPU to perform the operations.

```
clock_t start = clock(); // Record the starting time

// Perform matrix multiplication on CPU
matrixMul(a, b, c);

clock_t stop = clock(); // Record the stopping time
double cpu_time = ((double)(stop - start)) / CLOCKS_PER_SEC * 1000.0; // Calculate the elapsed time in milliseconds
```

Fig. 4.4 Performance measure of Naive GEMM

```
// Perform SpMV on CPU for random matrix
printf("Performance metrics for CPU (single-threaded) random CSR matrix:\n");
clock_t start_time_random = clock();
cpu_csr_spmv(&random_matrix, x, y_cpu_random);
clock_t end_time_random = clock();
double elapsed_time_random = ((double)(end_time_random - start_time_random)) / CLOCKS_PER_SEC * 1000.0; // Convert to milliseconds
```

Fig. 4.5 Performance measure of Naive SpMV

4.1.2 Multi-threaded implementation

Parallel implementation of matrix multiplication and SpMV on the CPU using OpenMP, a popular library for parallel programming. This implementation utilizes thread-level parallelism to improve performance by distributing computation across multiple CPU cores.

GEMM

Key Features:

- Matrix Multiplication with Tiling: The `matrixMulTiled` function performs matrix multiplication using a tiled approach, where the matrices are divided into smaller submatrices (tiles) to exploit locality and enhance the use of cache.
- Parallelization with OpenMP: The matrix multiplication process is parallelized in the code by means of OpenMP directives. The outer loops are parallelized with the `#pragma omp parallel for collapse(2)` command, which allows many threads to

carry out the computations in parallel. Efficient parallelization across many CPU cores is accomplished by condensing the nested loops using the `collapse(2)` clause into a single parallel loop.

```
// Function to perform matrix multiplication using OpenMP for multi-threading
void matrixMulTiled(float *a, float *b, float *c) {
    #pragma omp parallel for collapse(2)
    for(int row = 0; row < N; ++row) {
        for(int col = 0; col < N; ++col) {
            // Accumulator for the dot product
            float sum = 0.0f;

            // Iterate over tiles
            for (int t = 0; t < N / BLOCK_SIZE; ++t) {
                // Compute dot product of tiles
                for (int k = 0; k < BLOCK_SIZE; ++k) {
                    sum += a[row * N + t * BLOCK_SIZE + k] * b[(t * BLOCK_SIZE + k) * N + col];
                }
            }

            // Write the result to global memory
            c[row * N + col] = sum;
        }
    }
}
```

Fig. 4.6 MatrixMulTiled Code Snippet

SpMV

Key Features:

- Random Matrix Generation: It generates a random CSR matrix with a specified number of rows, columns, and density of non-zero elements. Refer to Fig. 4.3
- Parallelization with OpenMP: The `#pragma omp parallel for` directive enables concurrent execution of SpMV computations on multiple CPU cores.

```
// Function to perform SpMV on CPU (parallelized using OpenMP)
void cpu_csr_spmv_parallel(const csr_matrix* matrix, const float* x, float* y) {
    #pragma omp parallel for
    for (int i = 0; i < matrix->rows_count; ++i) {
        y[i] = 0.0f; // Initialize the result for this row
        for (int j = matrix->row_ptr[i]; j < matrix->row_ptr[i + 1]; ++j) {
            y[i] += matrix->values[j] * x[matrix->col_indices[j]]; // Perform dot product
        }
    }
}
```

Fig. 4.7 Multi-threaded SpMV on CPU

Performance measurement of GEMM and SpMV:

It accurately measures the execution time of the parallel SpMV operation using OpenMP's `omp_get_wtime()` function.

```
double start_time = omp_get_wtime();

// Perform matrix multiplication
matrixMulTiled(a, b, c);

double end_time = omp_get_wtime();

// Calculate execution time
double execution_time = end_time - start_time;
```

Fig. 4.8 Performance measurement of GEMM on CPU

```
// Perform SpMV on CPU for random matrix (parallelized)
printf("Performance metrics for CPU (parallelized with OpenMP) random CSR matrix:\n");
double start_time_parallel_random = omp_get_wtime();
cpu_csr_spmv_parallel(&random_matrix, x, y_cpu_parallel_random);
double end_time_parallel_random = omp_get_wtime();
double elapsed_time_parallel_random = (end_time_parallel_random - start_time_parallel_random) * 1000.0; // Convert to milliseconds

// Output performance metrics
printf("Elapsed Time: %f milliseconds\n", elapsed_time_parallel_random);
```

Fig. 4.9 Performance measurement of SpMV on CPU

Chapter 5

Algorithm 2

5.1 GPU Implementation:

The goal of the CUDA implementation of GEMM and SpMV is to speed up the respective processes utilizing the processing capabilities of NVIDIA GPUs. The implementation strategy, optimization methods used, and performance evaluation criteria are mentioned in the following sections below.

5.1.1 GEMM

The GEMM kernel(`matrixMulTiled`) utilizes a tiled matrix multiplication approach to exploit data locality and improve memory access patterns. Key aspects of the implementation include:

- Tiled Matrix Multiplication: The matrices a and b are divided into smaller submatrices(tiles) to facilitate efficient computation.
- Shared Memory Usage: Utilized to store tiles of matrices a and b, allowing fast data reuse within thread blocks.
- Grid and Block Configuration: The grid and block dimensions are configured to effectively utilize GPU resources and maximize parallelism.

- **Threads per block**: Each block consists of **BLOCK_SIZE x BLOCK_SIZE** threads, enabling data parallelism with a block.
- **Number of Blocks**: The number of thread blocks in each dimension is calculated to cover the entire matrix, ensuring that all elements are processed in parallel.

```
// Set grid and block dimensions
dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 numBlocks((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (N + BLOCK_SIZE - 1) / BLOCK_SIZE);

// Launch kernel
matrixMulTiled<<<numBlocks, threadsPerBlock>>>(d_a, d_b, d_c);
```

Fig. 5.1 Threads and Blocks in GEMM

- **Thread Index Calculation**: Each thread computes its row and column indices within the matrices using the formula:

$$\text{int row} = \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y};$$

$$\text{int col} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x};$$

- **Synchronization**: The `__syncthreads()` function is used to synchronize threads within a block. It ensures that all threads have finished loading data into shared memory before proceeding with computations, preventing data hazards and ensuring correctness of results.

Optimization Strategies

Impact of Tiling and Shared Memory usage:

1. **Before Tiling**: In the initial implementation of matrix multiplication in a straightforward, non-optimized, sequential approach, each element in the output matrix requires N thread accesses for matrices `a` and `b`, resulting in $2 \times N^3$ thread accesses for the entire operation.

2. **After Tiling**: To optimize memory access patterns and improve efficiency, tiling is employed. With tiling, each thread accesses a portion of the matrix a and b corresponding to the tile size BLOCK_SIZE x BLOCK_SIZE. This reduces the number of thread accesses per element in the output matrix to $2 \times T$, where T is the tile size. The total number of thread accesses after tiling is $2 \times T^2 \times (N/T)^2$

   Comparison and Impact: Comparing before and after tiling, it is evident that tiling significantly reduces the number of thread accesses by grouping multiple memory accesses into shared memory accesses within each tile. This optimization greatly improves memory access efficiency, especially for large matrices, as it reduces the number of global memory accesses and promotes coalesced memory accesses.

3. **Loop Unrolling**: Unroll inner loops within the kernel to reduce loop overhead and improve instruction-level parallelism. In the kernel function `matrixMulTiled` we have a nested loop structure to iterate over tiles and compute the dot product. Although explicit loop unrolling is not applied in the code, the compiler may perform loop-unrolling optimizations automatically when it generates machine code for the kernel. CUDA compilers, such as NVCC, often perform loop-unrolling optimizations when it benefits performance.

4. **Memory Coalescing**: Ensuring coalesced memory accesses to global memory by aligning memory accesses of threads within a warp, enhancing memory throughput. In the kernel function `matrixMulTiled`, memory coalescing is implicitly achieved by accessing contiguous memory locations when loading tiles of matrices a and b from global memory into shared memory. Each thread accesses adjacent memory locations within a warp, which promotes coalesced memory accesses.

```cpp
// Tiled matrix multiplication kernel using shared memory
__global__ void matrixMulTiled(float *a, float *b, float *c) {
    // Define shared memory for tiles of matrix 'a' and 'b'
    __shared__ float tile_a[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float tile_b[BLOCK_SIZE][BLOCK_SIZE];

    // Calculate row and column indices
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Accumulator for the dot product
    float sum = 0.0f;

    // Iterate over tiles
    for (int t = 0; t < N / BLOCK_SIZE; ++t) {
        // Load tiles of matrix 'a' and 'b' into shared memory
        tile_a[threadIdx.y][threadIdx.x] = a[row * N + t * BLOCK_SIZE + threadIdx.x];
        tile_b[threadIdx.y][threadIdx.x] = b[(t * BLOCK_SIZE + threadIdx.y) * N + col];

        // Synchronize threads to ensure all tiles are loaded
        __syncthreads();

        // Compute dot product of tiles
        for (int k = 0; k < BLOCK_SIZE; ++k) {
            sum += tile_a[threadIdx.y][k] * tile_b[k][threadIdx.x];
        }

        // Synchronize threads before loading next tiles
        __syncthreads();
    }

    // Write the result to global memory
    if (row < N && col < N) {
        c[row * N + col] = sum;
    }
}
```

Fig. 5.2 Optimized GEMM on GPU

5.1.2 SpMV

The SpMV operation is parallelized on the GPU using CUDA, leveraging the massively parallel architecture of modern GPUs. This allows for concurrent execution of multiple threads, each responsible for computing a portion of the result vector.

Key Features:

- `generate_random_csr_matrix` Function: This function generates a random CSR matrix with the specified dimensions and density. Initializes the CSR matrix with random values and column indices using the C++ `<random>` library.

```
// Function to generate a random CSR matrix
csr_matrix generate_random_csr_matrix(int rows, int cols, float density) {
    csr_matrix matrix;
    matrix.rows_count = rows;
    matrix.cols_count = cols;
    matrix.row_ptr = (int*)malloc((rows + 1) * sizeof(int));

    // Generate random values for the matrix
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(1.0, 1000.0); // Random value between 1 and 1000
    std::uniform_int_distribution<> col_dis(0, cols - 1); // Random column index between 0 and cols - 1

    int max_non_zeros = (int)(density * rows * cols);
    matrix.non_zero_count = max_non_zeros;
    matrix.values = (float*)malloc(max_non_zeros * sizeof(float));
    matrix.col_indices = (int*)malloc(max_non_zeros * sizeof(int));

    int current_index = 0;
    matrix.row_ptr[0] = 0;
    for (int i = 0; i < rows; ++i) {
        int non_zeros_in_row = (int)(density * cols); // Determine number of non-zeros in this row
        matrix.row_ptr[i + 1] = matrix.row_ptr[i] + non_zeros_in_row; // Update row_ptr
        for (int j = 0; j < non_zeros_in_row; ++j) {
            matrix.values[current_index] = dis(gen); // Random value
            matrix.col_indices[current_index] = col_dis(gen); // Random column index
            ++current_index;
        }
    }

    return matrix;
}
```

Fig. 5.3 Random Matrix in SpMV CUDA

- Grid and Block Configuration: In the figure below, `BLOCK_SIZE` defines the number of threads per block. To determine the number of blocks needed on the grid, the total number of rows in the matrix `matrix -> rows_count` is divided by the number of threads per block. The `+ threads_per_block - 1` term ensures that there are enough blocks to cover all rows, even if the total number of rows is not evenly divisible

by the number of threads per block.

```
// Launch kernel
int threads_per_block = BLOCK_SIZE;
int blocks_per_grid = (matrix->rows_count + threads_per_block - 1) / threads_per_block;

csr_spmv_kernel_shared<<<blocks_per_grid, threads_per_block>>>(matrix->rows_count, d_row_ptr, d_col_indices, d_values, d_x, d_y);
```

Fig. 5.4 Grid and Block Configuration in SpMV

Thread Index Calculation: Each thread calculates its index within the block(`tid`) and uses it to determine the corresponding row index in the matrix(`row`)

```
// Calculate the thread index within the block
int tid = threadIdx.x;
int row = blockIdx.x * blockDim.x + tid;
```

Fig. 5.5 Thread Index Calculation in SpMV CUDA

Benefits:

- **Efficient Resource Utilization**: A well-configured grid and block arrangement ensure that the GPU's whole computing capacity is used efficiently.
- **Load Balancing**: To ensure that each block completes a comparable amount of computation, load balancing can be achieved by distributing the workload equally across blocks.
- **Scalability**: The code is scalable and adaptive, since the grid and block configurations can be dynamically changed to support matrices with varying sizes and hardware configurations.

Optimization Strategies:

1. Memory Optimization: Shared memory is used to cache data that are repeatedly accessed by threads within a block. In the code, the shared memory arrays `shared_values` and `shared_col_indices` are declared to cache values and column indices for a tile of the matrix. Threads within the block efficiently access this shared memory, reducing the need to access data from slower global memory.

```
// Define shared memory for values and column indices
__shared__ float shared_values[BLOCK_SIZE];
__shared__ int shared_col_indices[BLOCK_SIZE];
```

Fig. 5.6 Shared Memory Code Snippet

2. Tiling and Memory Coalescing: In the `csr_spmv_kernel_shared` kernel, the memory accesses for loading values and column indices into shared memory are structured to maximize memory coalescing. This helps reduce memory access latency and improve memory bandwidth utilization. Loop blocking or tiling enhances data locality by reducing the working set size and improving cache utilization. Here, `offset + tid`

```
// Loop over the tiles of the matrix
for (int tile_index = 0; tile_index < (rows_count + BLOCK_SIZE - 1) / BLOCK_SIZE; ++tile_index) {
    // Load values and column indices for this tile into shared memory
    int col_index = tile_index * BLOCK_SIZE + tid;
    if (col_index < rows_count) {
        int offset = row_ptr[col_index];
        shared_values[tid] = values[offset + tid];
        shared_col_indices[tid] = col_indices[offset + tid];
    } else {
        shared_values[tid] = 0.0f;
        shared_col_indices[tid] = -1; // Invalid index
    }
```

Fig. 5.7 Memory Coalescing Code Snippet

ensures that each thread accesses a consecutive memory location in the global memory arrays *values* and *col_indices*, promoting memory coalescing.

```cuda
// CUDA kernel for SpMV computation with memory coalescing and shared memory optimization
__global__ void csr_spmv_kernel_shared(int rows_count, int* row_ptr, int* col_indices, float* values, float* x, float* y) {
    // Define shared memory for values and column indices
    __shared__ float shared_values[BLOCK_SIZE];
    __shared__ int shared_col_indices[BLOCK_SIZE];
    // Calculate the thread index within the block
    int tid = threadIdx.x;
    int row = blockIdx.x * blockDim.x + tid;
    // Initialize the result for this thread's row
    float dot = 0.0f;

    // Loop over the tiles of the matrix
    for (int tile_index = 0; tile_index < (rows_count + BLOCK_SIZE - 1) / BLOCK_SIZE; ++tile_index) {
        // Load values and column indices for this tile into shared memory
        int col_index = tile_index * BLOCK_SIZE + tid;
        if (col_index < rows_count) {
            int offset = row_ptr[col_index];
            shared_values[tid] = values[offset + tid];
            shared_col_indices[tid] = col_indices[offset + tid];
        } else {
            shared_values[tid] = 0.0f;
            shared_col_indices[tid] = -1; // Invalid index
        }

        // Synchronize to make sure all threads have loaded the tile
        __syncthreads();

        // Perform the dot product within the tile
        for (int i = 0; i < BLOCK_SIZE; ++i) {
            if (shared_col_indices[i] != -1) { // Check for valid column index
                dot += shared_values[i] * x[shared_col_indices[i]];
            }
        }

        // Synchronize before loading the next tile
        __syncthreads();
    }

    // Write the result back to global memory
    if (row < rows_count) {
        y[row] = dot;
    }
}
```

Fig. 5.8 Optimized SpMV on GPU

5.1.3 Performance Measurement of GEMM and SpMV in CUDA

- Execution Time: Provides insights into the time it takes for the GPU kernel to execute, allowing assessment of computational efficiency.
- GFLOPS: Indicates the rate of floating-pount operations executed per second, helping to understand the computational performance.
- Throughput: Offers a measure of the total number of operations executed per second, providing a holistic view of computational efficiency.

We measure the time taken for the kernel to execute using CUDA events.

```cuda
// Set grid and block dimensions
dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 numBlocks((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (N + BLOCK_SIZE - 1) / BLOCK_SIZE);

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

// Launch kernel
matrixMulTiled<<<numBlocks, threadsPerBlock>>>(d_a, d_b, d_c);

cudaEventRecord(stop);
cudaEventSynchronize(stop);

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

// Calculate the total number of floating-point operations
long long total_flops = 2 * (long long)N * (long long)N * (long long)N;

// Convert time to seconds
double seconds = milliseconds / 1000.0;

// Calculate GFLOPS
double gflops = total_flops / (seconds * 1e9); // Convert time to seconds and GFLOPS to 1e9 scale
double throughput = total_flops / seconds; // Throughput in operations per second
```

Fig. 5.9 Performance Measurement Metrics of GEMM in CUDA

```cuda
// Launch kernel
int threads_per_block = BLOCK_SIZE;
int blocks_per_grid = (matrix->rows_count + threads_per_block - 1) / threads_per_block;

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

csr_spmv_kernel_shared<<<blocks_per_grid, threads_per_block>>>(matrix->rows_count, d_row_ptr, d_col_indices, d_values, d_x, d_y);

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

// Copy result back to CPU
cudaMemcpy(y, d_y, matrix->rows_count * sizeof(float), cudaMemcpyDeviceToHost);

// Check for kernel launch errors
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
    fprintf(stderr, "CUDA kernel launch error: %s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Free GPU memory
cudaFree(d_values);
cudaFree(d_col_indices);
cudaFree(d_row_ptr);
cudaFree(d_x);
cudaFree(d_y);

// Calculate total number of floating-point operations
double total_operations = (double)matrix->non_zero_count * 2; // Assuming one multiplication and one addition per non-zero element

// Calculate GFLOPS
double seconds = milliseconds / 1000.0; // Convert milliseconds to seconds
double gflops = (total_operations / seconds) / 1e9; // Divide by elapsed time in seconds and 1 billion
double throughput = total_operations / seconds; // Throughput in operations per second
```

Fig. 5.10 Performance Measurement Metrics of SpMV in CUDA

How it works:

1. Event Creation: CUDA events are created using `cudaEventCreate`.

2. Record Start and Stop Event: Just before launching the GPU kernel, we record the start event using `cudaEventRecord` and after the GPU kernel execution is complete, we record the stop event.

3. Synchronization: We synchronize the CPU thread with the GPU using `cudaEventSynchronize` to ensure that the stop event is recorded after the GPU kernel execution finishes.

4. Elapsed Time Calculation: We calculate the elapsed time between the start and stop events using the `cudaEventElapsedTime` which provides the time difference between the two events in milliseconds.

Chapter 6

Results

6.1 Comparing the performances of different implementations

GEMM

This comparison focuses on execution times for matrix sizes ranging from 128×128 to 2048×2048, keeping the density constant (0.1).

6.1.1 Sequential CPU vs Multi-threaded CPU

The sequential CPU code encountered limitations beyond the matrix size of 2048×2048, which prevented it from producing results, showing that it could not handle large matrices effectively.

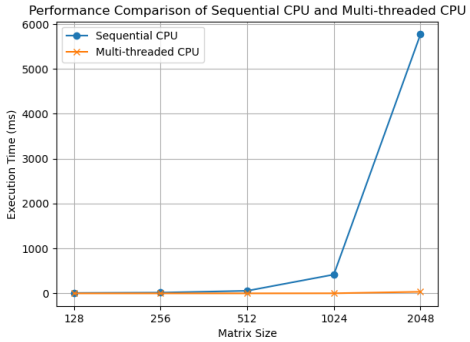

Fig. 6.1 GEMM Sequential vs Multi-threaded

6.1.2 Sequential CPU vs GPU

Unlike the sequential CPU implementation, the GPU implementation successfully processed larger matrices, showcasing its ability to handle higher computational loads efficiently.

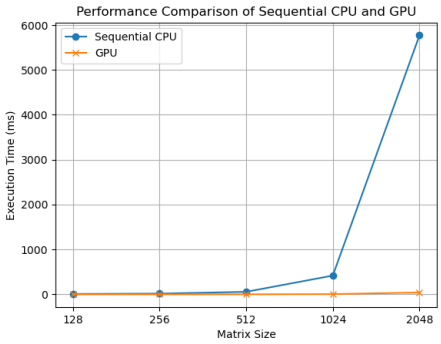

Fig. 6.2 GEMM Sequential vs GPU

6.1.3 Multi-threaded CPU vs GPU

The matrix sizes ranged from 128 to 4096, with a fixed density of 0.1.

As depicted in the graph, the execution times for both implementations varied with the size of the matrix. Notably, as the matrix size increased, the performance gap between the GPU and multi-threaded CPU implementations widened significantly across all tested matrix sizes. It is worth noting a significant increase in execution time with larger matrix sizes, even in the multi-threaded CPU implementation. This observation suggests that as the size of the matrices grows, the computational complexity increases, leading to longer processing times.

6.1.4 GPU Performance Metrics

The execution times increased with matrix size, indicating increased computational load, and the GFLOPS metric demonstrated the GPU's ability to efficiently handle larger datasets by exploiting parallelism, thereby achieving higher computational throughput.

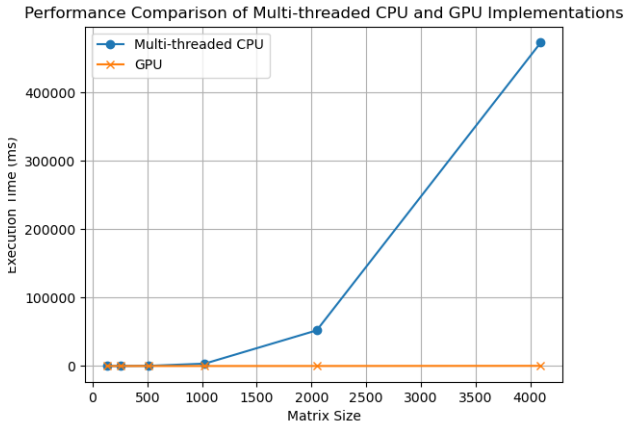

Fig. 6.3 GEMM Multi-threaded vs GPU

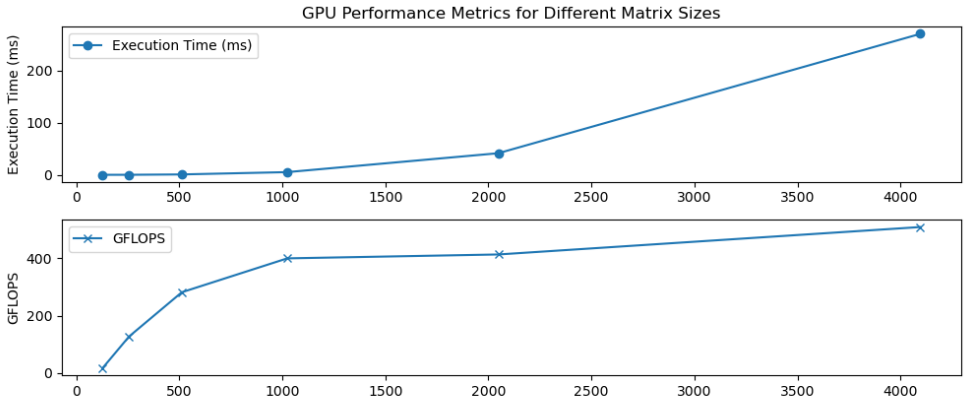

Fig. 6.4 GEMM Performance Metrics on GPU

SpMV

Evaluating the performance of matrices ranging from 128×128 to 32768×32768, maintaining a constant density of 0.1 for all matrices.

6.1.5 Sequential CPU, Multi-threaded CPU and GPU

For smaller matrix sizes, the sequential CPU execution times are comparable to those of the multi-threaded CPU. However, as the matrix size increases, the execution times for the sequential CPU grow significantly faster compared to the multi-threaded CPU and GPU implementations. The GPU consistently outperforms both CPU implementations across all matrix sizes, demonstrating its superior computational efficiency for parallelizable tasks like sparse matrix-vector multiplication.

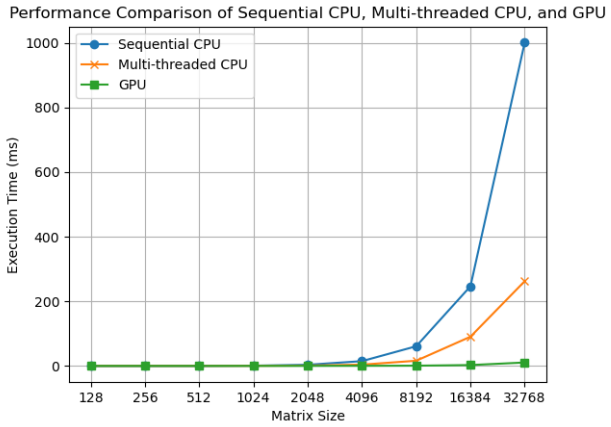

Fig. 6.5 SpMV Sequential, Multi-threaded and GPU

6.1.6 Multi-threaded CPU vs GPU

Comparing the execution times of the multi-threaded CPU and GPU implementations for a large matrix size $2^{15} \times 2^{15}$, a clear performance advantage emerges for the GPU solution. This observation highlights the scalability and efficiency offered by GPU acceleration, particularly for large-scale computational operations.

Performance metrics for random CSR matrix for CPU (parallelized with OpenMP):

- Elapsed time: 267.147445 milliseconds

Performance metrics for random CSR matrix for GPU (parallelized with CUDA):

- Elapsed time: 10.773568 milliseconds
- GFLOPS: 19.932892
- Throughput: 19932891772.035614 OPS

Throughput, measured in operations per second, is very high, indicating that the GPU is effectively processing a large number of operations within the given time frame.

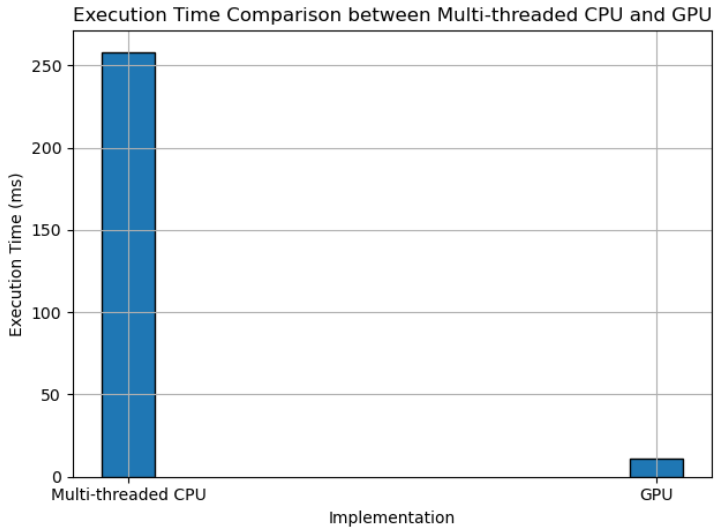

Fig. 6.6 SpMV Multi-threaded vs GPU

6.1.7 Comparing Multi-threaded CPU and GPU implementation for different densities

Our analysis reveals that GPU execution times remain relatively consistent across different densities, while CPU execution times increase significantly with higher densities. This suggests that GPU implementations offer more consistent performance across different sparse matrix densities compared to CPU implementations.

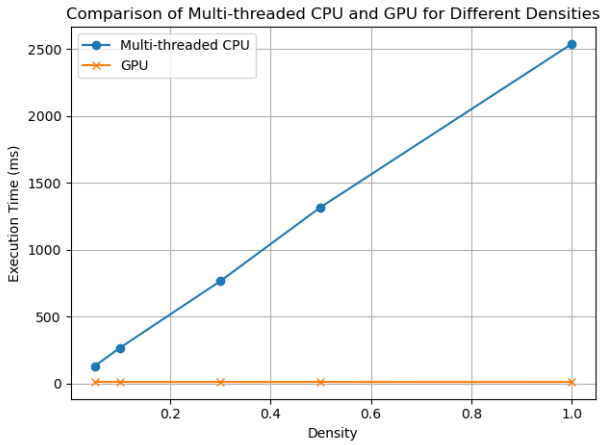

Fig. 6.7 SpMV Multi-threaded CPU and GPU for different densities

6.1.8 Plotting GPU Execution times, GFLOPs and Throughput

Varying Densities while keeping the Matrix Size Constant $2^{15} \times 2^{15}$

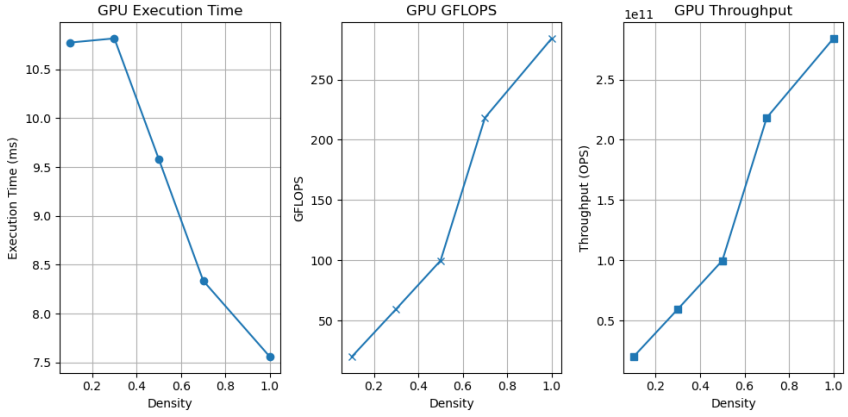

Fig. 6.8 SpMV on GPU

Chapter 7

Conclusion

This report has provided valuable insights into the performance optimization of matrix multiplication and sparse-matrix vector multiplication through various implementations on both the CPU and GPU platforms. Our findings underscore the significant impact of GPU acceleration on improving computational efficiency, particularly for large-scale matrix operations. The comparison between sequential CPU, multi-threaded CPU, and GPU implementations has revealed clear performance advantages for GPU solutions, with GPU consistently outperforming CPU counterparts across diverse matrix sizes and densities. Although the sequential approach many suffices for smaller matrices, its performance diminishes significantly as the problem size increases. In contrast, the GPU implementation showcases superior scalability, making it better suited for handling larger and more computationally demanding matrices efficiently.

7.1 Future Works

Future research could potentially optimize the implemented methods even more using NVIDIA profiling tools. NVIDIA Nsight Systems and NVIDIA Visual Profiler are two tools that provide deep insights into memory usage, kernel performance, and GPU utilization. Researchers can locate performance bottlenecks and optimize algorithms for increased effi-

ciency using these profiling tools. Further advancements in computing performance might result from the investigation of more sophisticated optimization strategies such as warp divergence minimization, memory access patterns optimization, and kernel fusion. Additional speedups could be unlocked by looking at the integration of asynchronous data transfer methods and mixed-precision arithmetic, particularly for deep learning and scientific computing applications.

References

[1] NVIDIA. The memory hierarchy of gpu. https://www.researchgate.net/figure/A-block-diagram-of-the-GPU-architecture_fig1_294139209.

[2] Architectural difference. https://slideplayer.com/slide/16140477/.

[3] "Introduction and Matrix Multiplication," https://www.youtube.com/watch?v=o7h_sYMk_oc.

[4] S. Fang, M. Huang, and S. Lin, "An efficient parallel architecture for matrix computations," *International Journal of High Performance Computing Applications*, vol. 15, no. 2, pp. 165–178, 2001.

[5] NVIDIA, "Nvidia cuda compute unified device architecture," https://www.nvidia.com/docs/io/66889/nvr-2008-004.pdf, NVIDIA Corporation, Tech. Rep., 2008.

[6] "TVM Documentation: How to Optimize Operators," https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html.

[7] Understanding NVIDIA CUDA. [Online]. Available: https://www.turing.com/kb/understanding-nvidia-cuda

[8] Everything You Need to Know About GPU Architecture. [Online]. Available: https://www.cherryservers.com/blog/everything-you-need-to-know-about-gpu-architecture

[9] CUDA Memory Hierarchy. [Online]. Available: http://thebeardsage.com/cuda-memory-hierarchy/#:~:text=The%20CPU%20and%20GPU%20have,CPU%20once%20processing%20has%20completed

[10] NVIDIA Tesla T4 Product Page. [Online]. Available: https://www.nvidia.com/en-in/data-center/tesla-t4/

[11] How to Implement Performance Metrics in CUDA C/C++. [Online]. Available: https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/

[12] "How to Calculate GMAC and GFLOPS with Python," *Medium*. [Online]. Available: https://medium.com/@ajithkumarv/how-to-calculate-gmac-and-gflops-with-python-62004a753e3b

[13] Tiled Matrix Multiplication. [Online]. Available: https://penny-xu.github.io/blog/tiled-matrix-multiplication

[14] "SpMV CUDA Repository," https://github.com/bsampson1/SpMV-CUDA.

[15] W. Liu, D. Wei, J. Wei, and K. Li, "Gpu parallel computing for feature selection in big data," *Journal of Parallel and Distributed Computing*, vol. 115, pp. 55–65, 2018.

[16] CPU vs GPU: Everything You Need to Know. [Online]. Available: https://www.run.ai/guides/multi-gpu/cpu-vs-gpu

[17] D. Rajan and R. Govindarajan, "A parallel gpu architecture for accelerating long short-term memory," https://arxiv.org/html/2404.06047v1#bib.bib2.

[18] K.-C. Wang and M.-S. Chen, "Gpu-based simulation for evaluating the effect of virtualization on memory systems," *Applied Sciences*, vol. 6, no. 3, p. 45, 2016.

[19] M. A. Raj and M. Balaji, "Challenges in parallel gpu computing," *Applied Sciences*, vol. 12, no. 14, p. 7073, 2022.