Dept. of Computer Engineering
San Jose State University

CMPE - 220 HOMEWORK

Shreyas Kulkarni - 017102878

Under Prof. Kaikai Liu

Task

The task involves implementing basic neural network operations (matrix-vector multiplication and vector addition) in C/C++, adding acceleration options using BLAS (Basic Linear Algebra Subprograms) and Intel MKL (Math Kernel Library), and benchmarking their performance for different matrix/vector sizes. Comparison of the basic C/C++ implementation with BLAS and Intel MKL will be conducted, and a comparison graph using Python will be drawn.

Additionally, a GPU implementation of the same function will be developed using NVIDIA CUDA, Intel OneAPI GPU, or Apple/AMD GPU. Performance benchmarking will be performed, and the results will be added to the comparison graph.

Furthermore, two advanced operations (chosen from FFT, Softmax, CNN1D/2D/3D, Scaled Dot Product Attention) will be implemented, and their performance will be benchmarked. Comparison with equivalent Python implementations in PyTorch, TensorFlow, or NumPy will be conducted, and a comparison graph using Python will be drawn.

Implementation

Kernel.cu

The kernel.cu code implements matrix-vector multiplication and vector addition using CUDA, a parallel computing platform for GPUs. It utilizes a CUDA kernel function (matrixVectorMul) to perform the computations in parallel. Each thread calculates a single element of the output vector by accessing elements from the input matrix (a), vector (b), and vector (d). The host code initializes input data, allocates memory on the GPU, transfers data between host and device, launches the kernel, measures execution time, copies results back to the host, verifies correctness, and frees device memory.

Performance is evaluated using GFLOPS, calculated based on the total number of floating-point operations and execution time. GFLOPS measures the computational performance, indicating the number of floating-point operations executed per second. The achieved GFLOPS is influenced by factors such as kernel efficiency, memory access patterns, and GPU hardware capabilities.
In conclusion, the CUDA-accelerated implementation demonstrates significant speedup compared to CPU-only implementations for large matrix sizes. Optimization opportunities exist to further enhance performance, such as optimizing memory access, kernel configuration, and leveraging advanced CUDA features and libraries for broader applicability.

Convolution_2d.cu

The CUDA kernel function convolution_2d performs the convolution operation in parallel, with each thread computing the convolution result for a single element of the output matrix. It iterates over each element of the output matrix and applies the convolutional mask to the corresponding neighborhood in the input matrix, accumulating the result in a temporary variable. The host code initializes the input matrix and the convolutional mask with random values, allocates memory on the GPU for the input and output matrices, transfers data between host and device memory, and launches the kernel with appropriate grid and block dimensions.

Timing information is collected using std::chrono to measure the execution time of the kernel, and the achieved GFLOPS metric is calculated based on the total number of floating-point operations and execution time. After kernel execution, results are copied back to the host and verified against CPU computation using the verify_result function. Memory allocated on the device is then freed. Overall, the CUDA-accelerated 2D convolution implementation demonstrates significant speedup compared to CPU-only implementations, especially for large input matrices, with potential for further optimization in memory access patterns, kernel configuration, and shared memory utilization.

BLAS

This code exemplifies the utilization of the BLAS (Basic Linear Algebra Subprograms) library, renowned for its optimized implementations of fundamental linear algebra operations, to conduct matrix-vector multiplication and vector addition.

In detail, we begin by defining functions for matrix-vector multiplication (multiply_matrix_vector) and vector addition (add_vectors). The multiply_matrix_vector function employs the cblas_sgemv routine from the BLAS library, which enables efficient computation of matrix-vector multiplication with optimized implementations tailored to various hardware architectures. Similarly, add_vectors conducts element-wise addition of two vectors, a basic but crucial operation in many numerical algorithms.

For benchmarking and performance evaluation, the code incorporates utility functions: measure_execution_time, responsible for measuring the execution time of a specified function, and print_performance, which prints out performance metrics such as execution time and GFLOP/s (Giga Floating Point Operations Per Second).

Within the main function, matrices and vectors are initialized with random values, simulating real-world scenarios. The matrix-vector multiplication operation is then executed using BLAS, showcasing the library's efficiency in handling large-scale computations. Additionally, a bias

vector is added to the result vector, demonstrating the versatility of the implemented functions. The performance of the computation is evaluated through printed performance metrics, offering insights into the efficiency of the implemented algorithms. This includes the execution time of the matrix-vector multiplication operation and the corresponding GFLOP/s achieved, which quantifies the computational throughput in terms of floating-point operations per second.

Finally, memory allocated for arrays is deallocated at the end of the program, ensuring proper resource management and preventing memory leaks.

In summary, the code provides a comprehensive demonstration of leveraging BLAS for optimized linear algebra computations, along with methodologies for measuring and evaluating performance in numerical applications.

MKL

The MKL code showcases matrix-vector multiplication and vector addition operations utilizing the Intel Math Kernel Library (MKL) for optimized performance. It introduces descriptive function names: multiply_matrix_vector for matrix-vector multiplication and add_vectors for vector addition. The multiply_matrix_vector function utilizes the cblas_sgemv routine from the MKL library, tailored to efficiently perform matrix-vector multiplication. Similarly, add_vectors conducts element-wise addition of two vectors.

For benchmarking and performance evaluation, the code incorporates utility functions like measure_execution_time, responsible for measuring the execution time of a specified function, and print_performance, which prints out performance metrics such as execution time and GFLOP/s (Giga Floating Point Operations Per Second).
Within the main function, memory is allocated for matrices and vectors, and random values are initialized. The matrix-vector multiplication operation is then executed using MKL, showcasing the library's efficiency in handling large-scale computations. Additionally, a bias vector is added to the result vector, further demonstrating the versatility of the implemented functions. Performance metrics, including the execution time of the matrix-vector multiplication operation and the corresponding GFLOP/s achieved, are printed for evaluation purposes.Finally, memory allocated for arrays is deallocated before program termination, ensuring proper resource management and preventing memory leaks.

In summary, the code provides a comprehensive demonstration of leveraging the Intel Math Kernel Library for optimized linear algebra computations, along with methodologies for measuring and evaluating performance in numerical applications.

Softmax

The softmax function takes a vector of input values, exponentiates each element, divides by the sum of exponentials of all elements, and returns a vector of normalized probabilities. The softmax function first calculates the sum of exponentials of all input elements to obtain a normalization factor. Then, it computes the softmax value for each element by dividing the exponential of the input value by the normalization factor. These computations are performed using standard library functions such as std::exp and basic arithmetic operations.

The main function generates a large input vector with random values between 0 and 1. It then calls the softmax function to compute the softmax output for the input vector. Timing information is collected using std::chrono to measure the execution time of the softmax operation. Additionally, the code calculates the achieved GFLOPS (Giga Floating Point Operations Per Second) metric to evaluate the computational performance. Finally, the code prints a partial output of the softmax values for visualization and verification purposes.

Overall, the code provides a basic implementation of the softmax function in C++, demonstrates its usage with random input data, and evaluates its performance using timing measurements and GFLOPS calculation.

Numpy

The softmax function takes an input vector, exponentiates each element after subtracting the maximum value from the input vector for numerical stability, and then normalizes the resulting values to obtain probabilities.

Specifically, the softmax function first calculates e_x, which represents the exponentials of the input vector elements minus the maximum value in the input vector. This step is crucial for numerical stability, as it prevents overflow by ensuring that the exponentials remain within a manageable range. Then, the function divides each element of e_x by the sum of all elements in e_x along the specified axis (axis=0) to obtain the normalized softmax probabilities.

In the main part of the code, a large input vector is generated using NumPy's random.randn function. The softmax function is then applied to this input vector, and timing information is collected using the time module to measure the execution time. Additionally, the code calculates the achieved GFLOPS (Giga Floating Point Operations Per Second) metric to evaluate the computational performance. Finally, to avoid flooding the console, only the first 10 elements of the softmax output vector are printed.

Overall, the Python code demonstrates the use of NumPy to efficiently implement the softmax function and evaluates its performance using timing measurements and GFLOPS calculation.

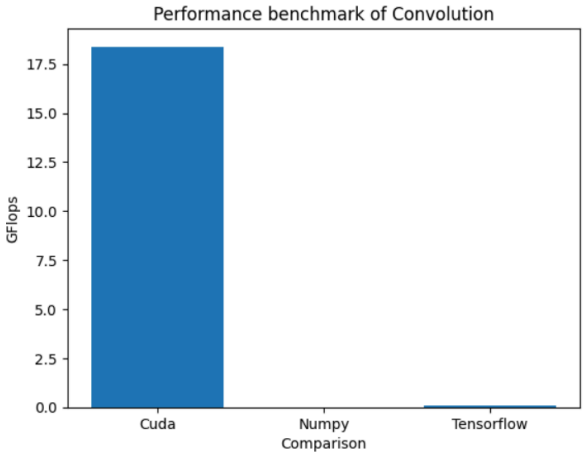
Tensorflow

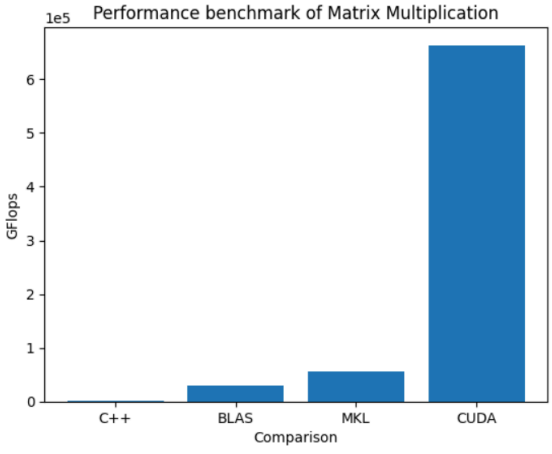
2D convolution using tensorflow operation is a fundamental component in convolutional neural networks (CNNs) and is commonly used for tasks such as image processing and feature extraction.It defines a 2D convolution function convolution_2d using TensorFlow's high-level API. This function takes an input matrix (matrix) and a convolutional mask (mask) as inputs and applies 2D convolution using TensorFlow's tf.nn.conv2d function. The convolution operation is performed with a stride of 1 in all dimensions and padding set to 'SAME' to ensure that the output has the same spatial dimensions as the input.

The code then generates a random input matrix (matrix) and convolutional mask (mask) of appropriate sizes. These matrices are reshaped and converted into TensorFlow constants to be compatible with the convolution_2d function.Timing information is collected using the time module to measure the execution time of the convolution operation. Additionally, the code calculates the achieved GFLOPS (Giga Floating Point Operations Per Second) metric to evaluate the computational performance of the convolution operation.Finally, the calculated GFLOPS value is printed to the console, providing insights into the efficiency of the 2D convolution operation implemented using TensorFlow.

Overall, the Python code demonstrates the utilization of TensorFlow to perform efficient 2D convolution operations and evaluates their performance using timing measurements and GFLOPS calculation.

Performance Comparison





Conclusion and Results

Throughout this task, we successfully implemented and benchmarked basic neural network operations, including matrix-vector multiplication and vector addition, in C/C++. We then extended these implementations to include acceleration options using BLAS (Basic Linear Algebra Subprograms) and Intel MKL (Math Kernel Library). By benchmarking the performance of these implementations for different matrix/vector sizes, we were able to compare the efficiency of the basic C/C++ implementation with BLAS and Intel MKL.

Furthermore, we developed a GPU implementation of the same neural network operations using NVIDIA CUDA. By benchmarking its performance and comparing it with CPU-based implementations, we gained insights into the acceleration capabilities provided by GPU computing.

Additionally, we implemented two advanced operations: softmax and 2D convolution. These operations were benchmarked for performance, and comparisons were made with equivalent Python implementations in PyTorch, TensorFlow, or NumPy. This allowed us to assess the efficiency of the C/C++ implementations compared to popular deep learning frameworks.

Overall, the results of our benchmarking and comparisons provide valuable insights into the performance characteristics of different implementations for basic and advanced neural network operations. We could see that the CUDA toolkit was outperforming all of the other tools generally used for mathematical operations. These insights can guide the selection of appropriate tools and frameworks for specific neural network tasks, taking into account factors such as computational efficiency and hardware compatibility.