Implementing GEMM with Block Floating Point on CUDA and Performance Comparison with cuBLAS

Abstract

This work presents a custom compute unified device architecture (CUDA) kernel for general matrix multiplication (GEMM) optimized for large matrices using block floating-point (BFP) arithmetic with brain float 16 byte (BF16) datatype. The kernel's performance and precision are evaluated against the highly optimized cuBLAS (CUDA based basic linear algebra subprograms) library operating on float32 datatype. Our implementation features global memory coalescing, shared memory utilization, and dynamic BFP quantization. The quantization process operates at CUDA warp granularity, where each sub-block of threads within a warp collectively manages its own shared exponent for BFP representation.

Introduction

GEMM is one of a crucial matrix operation in scientific computing and Artificial Intelligence (AI), particularly within neural networks where large matrix computations are essential. This work implements GEMM using BFP arithmetic on CUDA, leveraging BFP's ability to share a single exponent across a block of numbers [1]. This approach optimally balances computation on graphics processing units (GPU) by utilizing both fixed and floating-point units [1,2]. Our implementation is particularly advantageous for memory and computationally resource-constrained GPU systems performing large matrix operations. Performance evaluations against the cuBLAS library demonstrate the benefits of this implementation in terms of computational speed and resource efficiency.

Background and Motivation

BFP approach optimizes memory usage and computational efficiency by sharing exponents across blocks of values, rather than storing individual exponents for each value. This enables the use of fixed-point processors for floating-point operations, reducing both memory bandwidth and computational demands. However, this approach can introduce precision loss when handling values with diverse magnitudes, primarily due to mantissa truncation during BF16 alignment [3, 5]. To address this numerical stability challenge, we implemented Kahan summation algorithm [4], which systematically tracks and compensates for rounding errors. This compensation technique ensures numerical stability during extensive computations like large matrix multiplications in neural network inference, while maintaining BFP's inherent performance advantages.

Implementation of GEMM with BFP on CUDA

This matrix multiplication implementation balances performance and precision by combining optimization techniques with dynamic BFP quantization with CUDA’s intrinsic BF16 computations. This approach is particularly effective for deep learning and scientific computing applications that demand both speed and accuracy with memory constrained devices.

1. Global Memory Coalescing:

Memory coalescing is optimized by pre-transposing matrixB to row-major order and linearizing both input matrices. This ensures efficient row-major access patterns within tile blocks, enabling contiguous memory access by CUDA threads and maximizing bandwidth utilization.

2. Shared Memory Usage:

The implementation optimizes memory access through shared memory techniques including tiling, double buffering, and L2 caching. Tiles partition computation, allowing thread blocks to process matrix segments as a BF16 datatype in shared memory to reduce the memory burden in a constrained devices, while double buffering enables concurrent data loading and processing. L2 prefetching further reduces latency, minimizing global memory access and enhancing overall performance.

3. Block Floating Point Implementation:
BFP quantization and dequantization is performed on the fly in kernel without dynamic parallelism in cuda. It processes the input data from BF16 matrix elements into 16-bit BFP integer for storing mantissa and scaling factors. During the quantization and matrix operation is performed at warp level within efficient shared memory space at BF16 precision intrinsic arithmatic functions in CUDA, prevents the overhead could have arouse through dynamic kernel launches and global memory access.

4. Thread Block configuration:
The kernel uses a <<<128, 512, 1>>> grid and <<<32, 32, 1>>> block configuration, enabling 1,024 threads per block for 100% theoretical occupancy. While this maximizes resource usage, reducing per-thread register count could allow more concurrent threads for better performance.

Performance comparison with cuBLAS
1.Testing procedure:
The input matrices have dimensions 4096x4096 for Matrix A and 4096x16384 for Matrix B, resulting in a 4096x16384 output matrix. During matrix operations, 4-byte single-precision floating-point data is converted to 2-byte BFP within the kernel and then output as 4-byte floating-point values. Both the naive implementation and cuBLAS utilize linearized input for matrix operations, with cuBLAS relying on standard floating-point arithmetic. Performance is compared to the standard cublasGemmEx function in cuBLAS. Both Naive and cuBLAS implementation is tested on NVIDIA A100 GPU and launched by AMD EPYC 7302 CPU.
2.Performance metrics:

Table 1: Performance Comparison between Naive CUDA BFP and cuBLAS

Operation	Naive CUDA BFP Time (ms)	cuBLAS Time (ms)
GEMM (A: 4096x4096, B: 4096x16384)	1595	254

3.Analysis:
This BFP matrix multiplication achieves relative better accuracy, with a mean relative error of approximately 1.46×10^{-02} and 1.08×10^{-02} compared to naive CPU and cuBLAS GEMM implementations respectively. This precision is maintained using Kahan summation for error compensation. In terms of performance, the naive implementation runs in 1595 ms, while cuBLAS completes the same task in 254 ms, offering a speed advantage of about 5x with floating point arithmatics.

Disscusion:

This BF16 matrix multiplication implementation balances performance and precision, making it suitable for memory-constrained AI applications, hardware without cuBLAS, and scenarios requiring customization. Although cuBLAS offers faster processing, this approach supports resource-limited environments while maintaining reliable accuracy.

Future Improvements:

Future optimizations could leverage CUDA's Warp Matrix Multiply Accumulate (WMMA) to enable efficient warp-level matrix operations and fused arithmetic, such as fused addition and multiplication functions in cuda [5]. As the quantization is performed at warp level, WMMA can further enhance performance while aligning computations with warp size and utilizing collaborative thread processing. This approach is designed to operate under shared memory space and achieve global memory coalescence, reducing memory overhead and network latency. Consequently, leveraging WMMA for tensor processors could represent a significant improvement over this implementation, especially for large matrices.

References:

1. David Elam et al, A Block Floating Point Implementation for an N-Point FFT on the TMS320C55x DSP, 2003.
2. Marcelo Gennari do Nascimento et al, HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation, 2023.
3. Shibo Wang, BFloat16: The secret to high performance on Cloud TPUs, 2019.
4. Nicholas J. Higham, The Accuracy of Floating Point Summation, SIAM Journal on Scientific Computing, 1993.
5. CUDA Math API Reference Manual, v12.3, 2024