DFT & FFT Implementation with CUDA

Group 17
A59023194 Ya-Chi Liao
A59026653 Tsung-Hsiang Ma

UC San Diego

ECE 277 GPU Programming | December 8, 2024

Introduction

- **Importance of DFT and FFT in Modern Applications**
  - Widely used in signal processing, image processing, communications, and scientific simulations.
  - Essential for efficient frequency domain analysis.
- **Performance Bottleneck in Traditional Approaches**
  - Computationally intensive for large datasets.
  - Serial implementations struggle with real-time processing demands.
- **Growing Need for High-Performance Solutions**
  - Explosion of big data in AI, IoT, and multimedia applications.
  - Real-time requirements in fields like radar, medical imaging, and financial modeling.
- **Role of GPUs and CUDA**
  - GPUs offer massive parallelism ideal for computationally heavy tasks.
  - CUDA programming allows fine-grained control of GPU resources.

Objective

- **Optimize Computational Performance:**

  Exploit parallel processing capabilities of GPUs to reduce computation time.

- **Resource Efficiency:**

  Minimize memory usage and optimize GPU resource utilization.

Methodologies

UC San Diego

- **Discrete Fourier Transform (DFT)**
  - Benchmark
  - Acceleration with CUDA global memory
  - Acceleration with CUDA shared memory
- **Fast Fourier Transform (FFT)**
  - Benchmark
  - Acceleration with CUDA global memory
  - Implementation with CuFFT for reference

DFT Introduction

- **Discrete Fourier Transform (DFT):**
  - The DFT converts a finite sequence of equally spaced samples of a function into a sequence of coefficients of a finite combination of complex sinusoids.
  - Direct computation of DFT has $O(n^2)$ complexity, which can be significantly reduced using the Fast Fourier Transform algorithm with $O(N \log N)$ complexity.
  - $X[k] = \sum x[n] \cdot e^{(-2\pi knj/N)}$, $k, n = 0, 1, \ldots, N-1$

DFT Implementation

- Thread Organization
  - Assign one CUDA thread for computing each output frequency component X[k].
  - Use shared memory to minimize global memory access during computation.
- Kernel Design
  - Each thread computes X[k] by iterating over all x[n] values, performing the sum of the products x[n]·e^(-2πknj/N).
- Optimization
  - Use Coalesced Memory Access to fetch x[n] from global memory.
  - Use Fast Math Operations to compute sine and cosine efficiently.

DFT Benchmark

UC San Diego

- This benchmark serves as a reference for comparing speed and accuracy with CUDA-accelerated implementation.
- $X[k] = \sum x[n] \cdot e^{(-2\pi knj/N)}$, k, n=0, 1, …, N-1

```
void computeDFT_CPU(const float* input_real, const float* input_imag,
    float* output_real, float* output_imag, int N) {
    for (int k = 0; k < N; ++k) { // Iterate over each frequency bin
        double sum_real = 0.0;
        double sum_imag = 0.0;

        for (int n = 0; n < N; ++n) { // Sum over the input signal
            double angle = -2.0 * M_PI * k * n / N;
            double cos_val = cos(angle);
            double sin_val = sin(angle);

            sum_real += input_real[n] * cos_val - input_imag[n] * sin_val;
            sum_imag += input_real[n] * sin_val + input_imag[n] * cos_val;
        }

        output_real[k] = sum_real;
        output_imag[k] = sum_imag;
    }
}
```

Global Memory DFT

- The input signal consists of N complex numbers, where each number has a real and imaginary component.
- The output consist of
  - $\text{real}(X[k]) = \sum x[n] \cdot \cos(2\pi kn/N)$
  - $\text{img}(X[k]) = \sum x[n] \cdot \sin(2\pi kn/N)$
- Compute X[k] for k = 1, 2, …, N-1

Shared Memory DFT

- Input Tiling:
  - Divide the input data into tiles that fit into the shared memory of each block.
- Shared Memory Allocation:
  - Allocate shared memory dynamically within the kernel to store a portion of the input data for each thread block.
- Loading Data into Shared Memory
- Synchronization
- Local Computation
- Iterate Over Tiles

Implementing DFT with shared memory is good but inefficient for large N. FFT algorithms are more practical in real-world applications.

Result

N	16	256	1024
Benchmark	37.79 us	936 us	3.70 ms
Global Mem	37.82 us	934 us	3.71 ms
Shared Mem	38.50 us	917 us	3.52 ms

FFT Introduction

- **Fast Fourier Transform (FFT):**
  - An efficient algorithm to compute the Discrete Fourier Transform (DFT).
  - Converts a signal from the time domain to the frequency domain.
  - Reduces computational complexity from O(n^2) (naive DFT) to O(nlogn)
- **FFT Implementation:**
  - Bit-Reversal Reordering
    - Reorganizes input data for efficient memory access.
  - Iterative Cooley-Tukey Algorithm
    - Processes data in iterative stages.
    - Combines results using "butterfly" operations for each pair of input points.
    - Utilizes twiddle factors to compute complex multiplications.

FFT Benchmark

- This benchmark serves as a reference for comparing speed and accuracy with CUDA-accelerated implementation.
- **Code Structure:**
  - Bit-reversal reordering
  - Iterative Cooley-Tukey FFT algorithm:
    - Radix-2 Iterative Implementation
    - Twiddle Factor Precomputation

Global Memory FFT

UC San Diego

- **Bit-Reversal Reordering**
    - Parallel Execution: Each thread processes one index, calculating its bit-reversed counterpart independently.
    - GPU Threads: Kernel bit_reversal_kernel launches N threads for N-point FFT.
    - Reduced Time Complexity: Instead of serial reordering, all elements are processed concurrently.
- **Twiddle Factor Computation**
    - Dynamic Calculation in Each Thread: Each thread computes its unique twiddle factor during the "butterfly" operation.
    - Thread Independence: Each thread works on its portion of the data, eliminating dependencies and enabling parallelism.
- **Iterative Cooley-Tukey**
    - Parallel Butterfly Operations: Each thread handles one butterfly computation (even-odd pair).
    - Efficient Memory Access:Reordered input (bit-reversal) ensures coalesced memory access patterns.

Result

N	16	256	1024
Benchmark	41.7 us	7.74 ms	37.1 ms
Global Mem	36.28 us	76.45 us	112.37 us
CuFFT(for reference)	10.21 us	10.21 us	10.27 us

Conclusion

- **Performance Trends**:
  - Shared memory optimizations slightly reduce computation time for DFT, especially for larger N. However, gains are marginal.
- **FFT Advantage**:
  - FFT with GPU is significantly faster than DFT and FFT with CPU due to its optimized O(NlogN) complexity and the capability of being parallel computing.
- **Applicability**:
  - DFT with CUDA is educational, but FFT is the practical choice for efficient frequency-domain transformations in GPU applications.

Reference

- Govindaraju, N. K., Lloyd, B., Dotsenko, Y., Smith, B., & Manferdelli, J. (2008, November). High performance discrete Fourier transforms on graphics processors. In SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing (pp. 1-12). IEEE.
- Nvidia, CuFFT Library, https://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf