Performance Improvement of Least Squares Algorithm using CUDA and OpenMP

Introduction

The least squares method is a fundamental technique in data fitting, used to find the best-fitting curve to a given set of points by minimizing the sum of the squares of the residuals. For a linear model $y = mx + c$, the goal is to find the slope m and intercept c that best fit the data.

Derivation of Least Squares for $y = mx + c$

Given a set of n points (x_i, y_i), the least squares method minimizes the error function:

$$E = \sum_{i=1}^{n} (y_i - (mx_i + c))^2$$

To minimize E, we take partial derivatives with respect to m and c and set them to zero:

$$\frac{\partial E}{\partial m} = -2 \sum_{i=1}^{n} x_i (y_i - (mx_i + c)) = 0$$

$$\frac{\partial E}{\partial c} = -2 \sum_{i=1}^{n} (y_i - (mx_i + c)) = 0$$

Solving these equations, we get:

$$m = \frac{n \sum xy - (\sum x)(\sum y)}{n \sum x^2 - (\sum x)^2}$$

$$c = \frac{\sum y - m \sum x}{n}$$

where:

$$\sum x = \sum_{i=1}^{n} x_i, \quad \sum y = \sum_{i=1}^{n} y_i, \quad \sum xy = \sum_{i=1}^{n} x_i y_i, \quad \sum x^2 = \sum_{i=1}^{n} x_i^2$$

Compute Power of the System

The system is powered by the **NVIDIA GeForce GTX 1650** graphics card, based on the *Turing* architecture. Below are the key specifications of the GPU relevant to CUDA-based computations:

Core Configuration

- **Shading Units (CUDA Cores):** 896
- **Streaming Multiprocessors (SM):** 14
- **L2 Cache:** 1024 KB

Memory Specifications

- **Memory Size:** 4 GB GDDR5
- **Memory Bus:** 128 bit
- **Memory Bandwidth:** 128.1 GB/s

Compute Performance

- **FP32 Performance (float):** 2.984 TFLOPS
- **FP16 Performance (half):** 5.967 TFLOPS
- **FP64 Performance (double):** 93.24 GFLOPS

Power and Thermal Design

- **Thermal Design Power (TDP):** 75 W

Relevance to CUDA Performance

The GTX 1650's 896 CUDA cores and 128.1 GB/s memory bandwidth make it suitable for parallel computations, such as those required by the least squares algorithm. Its FP32 performance of 2.984 TFLOPS ensures efficient handling of floating-point operations, which are critical for numerical algorithms.

Serial Implementation

The serial implementation of the least squares algorithm computes the sums $\sum x$, $\sum y$, $\sum xy$, and $\sum x^2$ in a loop. Here is the code:

```cpp
#include <iostream>
#include <cmath>
#include <ctime>

using namespace std;

const double LsEpsilon = 1.0e-12;

bool calcLeastSquaresCPP(const double* x, const double* y, int n,
    double* m, double* b) {
    if (n <= 0) return false;

    double sumX = 0, sumY = 0, sumXY = 0, sumXX = 0;

    for(int i = 0; i < n; i++) {
        sumX += x[i];
        sumY += y[i];
        sumXX += x[i] * x[i];
        sumXY += x[i] * y[i];
    }

    double denom = n * sumXX - sumX * sumX;
    if (LsEpsilon >= fabs(denom)) return false;

    *m = (n * sumXY - sumX * sumY) / denom;
    *b = (sumXX * sumY - sumX * sumXY) / denom;

    return true;
}

const int n = 10000 * 10000;
double x[n];
double y[n];

int main() {
    const double slope = 1.0;
    const double y_int = 0.5;
    double start_time, end_time;

    // Initialize the input vectors
    for (int i = 0; i < n; i++) {
        x[i] = i;
        y[i] = slope * x[i] + y_int;
    }

    // Start time
    start_time = static_cast<double>(clock()) / CLOCKS_PER_SEC;

    double m1 = 0, b1 = 0;
    bool rv1 = calcLeastSquaresCPP(x, y, n, &m1, &b1);

    // End time
    end_time = static_cast<double>(clock()) / CLOCKS_PER_SEC;

    // Calculate elapsed time
    double time_taken = end_time - start_time;

```

```
    // Output results
    printf("slope = %6.4lf, m1 = %6.4lf\n", slope, m1);
    cout << "y_int = " << y_int << ", b1 = " << b1 << endl;
    cout << "rv1 = " << rv1 << endl;
    printf("Time: %f seconds\n", time_taken);

    return 0;
}
```

Listing 1: Serial Implementation

Parallel Implementation with OpenMP and CUDA

To improve performance, we parallelize the initialization using OpenMP and the least squares computation using CUDA.

OpenMP Initialization

OpenMP is used to parallelize the initialization of the arrays x and y. The following code snippet shows the parallelized initialization:

```
#pragma omp parallel for
for (int i = 0; i < n; i += 4) {
    x[i] = i;
    y[i] = slope * x[i] + y_int;
    x[i+1] = (i+1);
    y[i+1] = slope * x[i+1] + y_int;
    x[i+2] = (i+2);
    y[i+2] = slope * x[i+2] + y_int;
    x[i+3] = (i+3);
    y[i+3] = slope * x[i+3] + y_int;
}
```

Listing 2: OpenMP Initialization

CUDA Kernel

The CUDA kernel computes the sums $\sum x$, $\sum y$, $\sum xy$, and $\sum x^2$ in parallel. Here is the CUDA implementation:

```
__global__ void calcLeastSquaresKernel(const double* x, const
    double* y, int n,
    double* sumX, double* sumY, double* sumXY, double* sumXX) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        atomicAdd(sumX, x[idx]);
        atomicAdd(sumY, y[idx]);
        atomicAdd(sumXX, x[idx] * x[idx]);
        atomicAdd(sumXY, x[idx] * y[idx]);
    }
}
```

Listing 3: CUDA Kernel

Performance Analysis

We analyze the performance in two scenarios: (1) including host-to-device copy time and (2) excluding host-to-device copy time.

Including Host-to-Device Copy Time

When the host-to-device copy time is included, the results are as follows:

$$\text{Serial Time} = 1.078125\,\text{seconds}$$

$$\text{Parallel Time} = 0.437841\,\text{seconds}$$

$$\text{Speedup} = \frac{1.078125}{0.437841} \approx 2.46$$

Excluding Host-to-Device Copy Time

When the host-to-device copy time is excluded, the results are as follows:

$$\text{Serial Time} = 0.340106\,\text{seconds}$$

$$\text{Parallel Time} = 0.064518\,\text{seconds}$$

$$\text{Speedup} = \frac{0.340106}{0.064518} \approx 5.27$$

Explanation of NVCC Flags

The ‘nvcc‘ command used to compile the CUDA and OpenMP code is:

```
nvcc -Xcompiler -fopenmp -arch=sm_60 openMP_cuda2.cu -lgomp
```

- **-Xcompiler -fopenmp**: Passes the ‘-fopenmp‘ flag to the host compiler to enable OpenMP.
- **-arch=sm_60**: Specifies the compute capability of the GPU (e.g., ‘sm_60‘ for Pascal architecture). Other options include ‘sm_50‘ (Maxwell), ‘sm_70‘ (Volta), etc.
- **-lgomp**: Links the OpenMP library.

Amdahl's Law and OpenMP Threads

Amdahl's Law states that the speedup of a program using multiple processors is limited by the fraction of the program that cannot be parallelized. The formula is:

$$\text{Speedup} = \frac{1}{(1 - P) + \frac{P}{N}}$$

where P is the parallel fraction of the program, and N is the number of processors. In this case, OpenMP was tested with 1, 2, 4, 8, 16, 32, and 64 threads. Among these, 8 threads provided the best parallelization fraction of $p = 97.4\%$, leading to a significant speedup. This optimal thread count balances the overhead of thread management with the benefits of parallel execution.

Conclusion

The least squares algorithm benefits significantly from parallelization using CUDA and OpenMP. When excluding the host-to-device copy time, the speedup increases to approximately 5.27x, demonstrating the true computational power of the GPU. The small difference in results between the serial and parallel implementations (e.g., $b1 = 0.500124$ vs. $b1 = 0.500037$) is due to the non-associative nature of floating-point addition, which can lead to slight variations in results when computations are performed in parallel. The use of 8 threads in OpenMP, guided by Amdahl's Law, provided an optimal balance between parallelization and overhead.