Project 5: GPU Acceleration with CUDA

12012936 王雅淇

This project contains two parts, exploring the comparative efficiencies of CPU and GPU in handling computations that are parallel in nature. It consists of two parts that evaluate these components through different computational tasks.

I. Scale Addition using CPU and GPU

In this part, we examine a basic arithmetic operation—scaling and adding matrices—and compare the performance between using CPU and GPU. The experiment is conducted using custom functions written for both hardware types.

GPU implementation

The GPU version of the function is implemented using CUDA, which allows for parallel processing directly on the GPU. Here, the `scaleAddKernel` kernel function is designed to perform the matrix operation. The function utilizes a simple data parallelism approach where each thread computes one element of the result matrix:

```
__global__ void scaleAddKernel(const float * input, float * output, size_t
len, float a, float b)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < len)
        output[i] = a * input[i] + b;
}
```

The main function, `scaleAddGPU`, prepares data for the GPU, calls the kernel, and then copies the results back to the host:

```
bool scaleAddGPU(const Matrix * pMat, Matrix * pResult, float a, float b)
{
    if(pMat == NULL || pResult == NULL)
    {
        fprintf(stderr, "Null pointer.\n");
        return false;
    }
    if (pMat->rows != pResult->rows || pMat->cols != pResult->cols)
    {
        fprintf(stderr, "The matrices are not of the same size.\n");
        return false;
    }

    cudaError_t ecode = cudaSuccess;
    size_t len = pMat->rows * pMat->cols;
```

```
    cudaMemcpy(pMat->data_device, pMat->data, sizeof(float)*len,
cudaMemcpyHostToDevice);
    scaleAddKernel<<<(len+255)/256, 256>>>(pMat->data_device, pResult-
>data_device, len, a, b);
    if ((ecode = cudaGetLastError()) != cudaSuccess)
    {
        fprintf(stderr, "CUDA Error: %s\n", cudaGetErrorString(ecode));
        return false;
    }
    cudaMemcpy(pResult->data, pResult->data_device, sizeof(float)*len,
cudaMemcpyDeviceToHost);

    return true;
}
```

CPU implementation

The CPU implementation of the matrix operation is straightforward, iterating over each matrix element and applying the scale and addition operation:

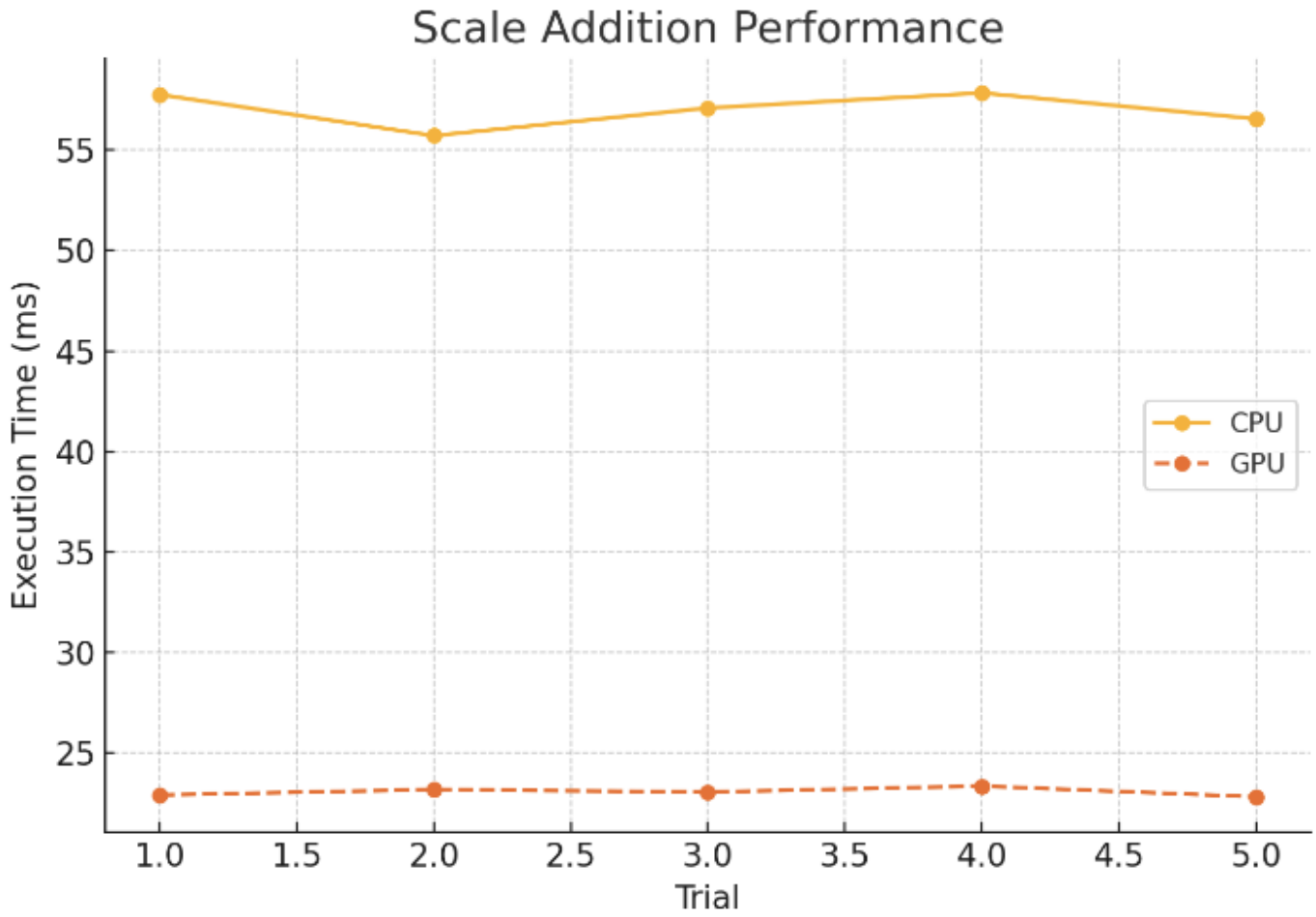
```
bool scaleAddCPU(const Matrix * pMat, Matrix * pResult, float a, float b)
{
    if(pMat == NULL || pResult == NULL)
    {
        fprintf(stderr, "Null pointer.\n");
        return false;
    }
    if(pMat->rows != pResult->rows || pMat->cols != pResult->cols)
    {
        fprintf(stderr, "The matrices are not of the same size.\n");
        return false;
    }

    size_t len = pMat->rows * pMat->cols;
    for(size_t i = 0; i < len; i++)
    {
        pResult->data[i] = a * pMat->data[i] + b;
    }
    return true;
}
```

We conducted five trials for each implementation to assess performance variations between the CPU and GPU. Here are the average execution times:

Trial	CPU Time (ms)	GPU Time (ms)
1	57.74800	22.90700
2	55.70400	23.18200

Trial	CPU Time (ms)	GPU Time (ms)
3	57.08200	23.05600
4	57.83100	23.35500
5	56.54300	22.82000
Average	**56.98160**	**23.06400**



The average execution times for the scale addition task are as follows:

- CPU Average Time: 56.98160 ms
- GPU Average Time: 23.06400 ms

Here, the GPU performs the task approximately 2.47 times faster than the CPU. This demonstrates the effectiveness of GPUs in handling parallel computations, significantly reducing execution time for operations that can be distributed across multiple threads.

II. OpenBLAS on CPU v.s. cuBLAS on GPU

In the second part of the project, we explore the performance of matrix multiplication using optimized libraries for both the CPU (OpenBLAS) and GPU (cuBLAS).

OpenBLAS CPU implementation

For the CPU, we use OpenBLAS, a library that provides high-performance matrix multiplication. The function tracks execution time from start to finish:

```
    struct timeval start, end;
    gettimeofday(&start, NULL);

    // Perform matrix multiplication using OpenBLAS
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1.0,
A, N, B, N, 0.0, C, N);

    gettimeofday(&end, NULL);
```

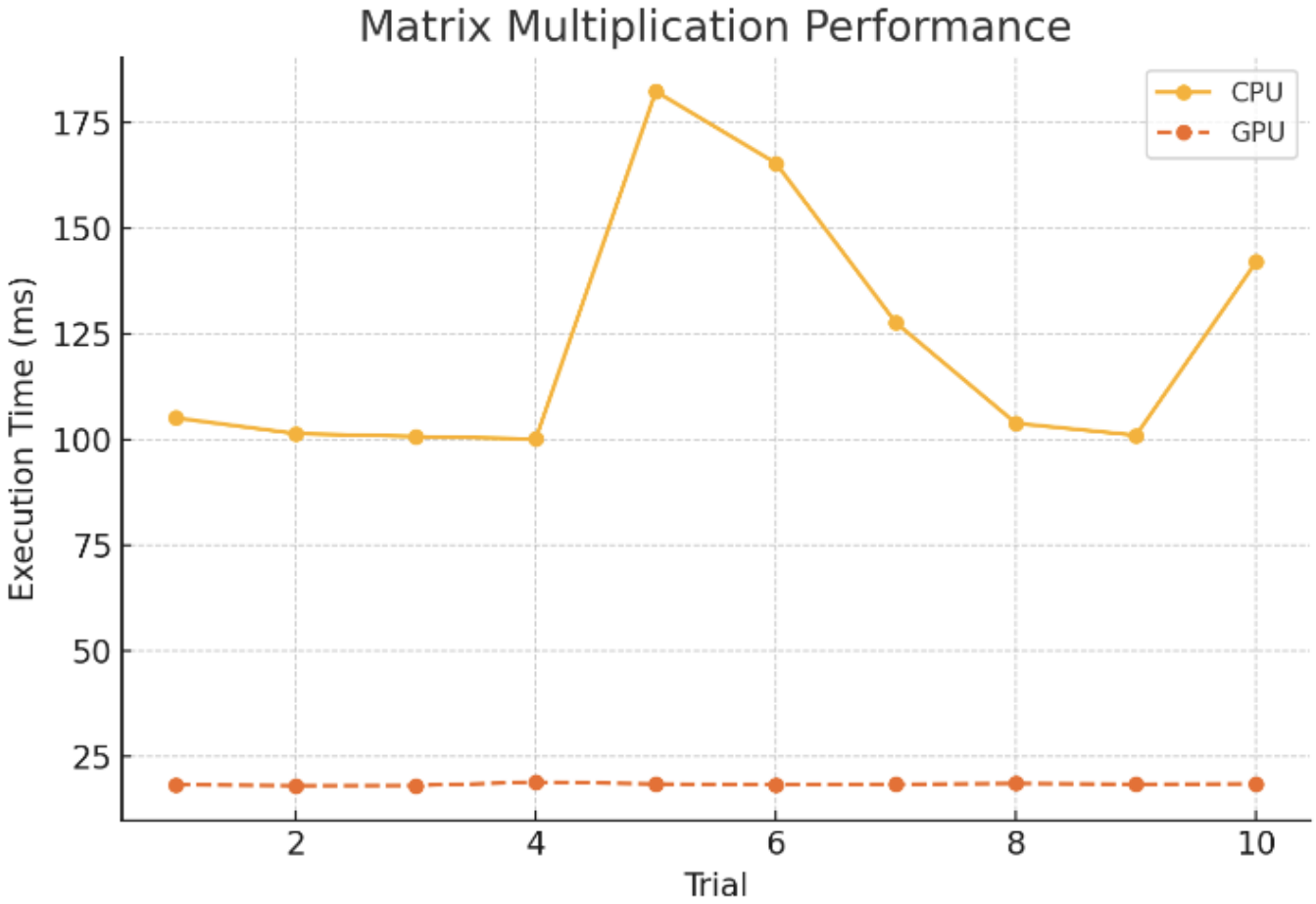
cuBLAS GPU implementation

The GPU counterpart uses cuBLAS, which is highly optimized for NVIDIA GPUs. The execution is timed using CUDA events:

```
    float alpha = 1.0f, beta = 0.0f;
    CUDA_CHECK(cudaEventRecord(start));
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, N, N, &alpha, d_A, N,
d_B, N, &beta, d_C, N);
    CUDA_CHECK(cudaEventRecord(stop));
```

The results from ten trials demonstrate significant differences in execution times between CPU and GPU processing:

Trial	CPU Time (ms)	GPU Time (ms)
1	105.089	18.252
2	101.382	18.0653
3	100.698	18.0926
4	100.106	18.8913
5	182.38	18.4751
6	165.406	18.2843
7	127.621	18.4046
8	103.817	18.5796
9	100.981	18.4181
10	142.004	18.5093
Average	**122.9484**	**18.40732**



The average execution times for the matrix multiplication task are:

- CPU Average Time: 122.9484 ms
- GPU Average Time: 18.40732 ms

In this task, the GPU is about 6.68 times faster than the CPU. This greater disparity in performance highlights the GPU's superior ability to handle more complex and intensive parallel computations like matrix multiplication.