Project 5

Jiawei Xiong SID:12011022

1. Introduction

The objective of this project is to perform matrix addition and multiplication on both CPU and GPU, using matrices of size 4096 x 4096, and compare the execution times of these operations to understand the performance differences between CPU and GPU computations.

2. Code Implementation

2.1 Initialize Matrix

```
typedef struct
{
    size_t rows;
    size_t cols;
    float * data; // CPU memory
    float * data_device; //GPU mememory
} Matrix;

Matrix * initializeMatrix(size_t r, size_t c)
{   size_t len = r * c;
    if(len == 0)
    {return NULL;}
    Matrix * p  = (Matrix *) malloc(sizeof(Matrix));
    if (p == NULL)
    {fprintf(stderr, "Allocate host memory failed.\n");
        goto ERR_TAG;
    }
    p->rows = r;
    p->cols = c;
    p->data = (float*)malloc(sizeof(float)*len);
...
}
```

The `initializeMatrix` function is designed to initialize a matrix by allocating memory for both the CPU and GPU. This process begins by calculating the total number of elements in the matrix. If the calculated size is zero, an error message is displayed, and the function returns `NULL`. The function then allocates memory for the `Matrix` structure. If this allocation fails, an error message is displayed, and the function proceeds to clean up any allocated resources before returning `NULL`.

Next, the function allocates memory for the matrix data on the CPU. If this allocation fails, it follows a similar error-handling procedure. The final step involves allocating memory on the GPU using `cudaMalloc`. If any of these steps fail, the function ensures that all previously allocated memory is freed to prevent memory leaks. If all allocations are successful, the function returns a pointer to the initialized `Matrix` structure. This robust approach ensures proper resource management and error handling during matrix initialization.

2.2 Scale addition

CPU Implementation

```
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, "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;}
```

In the provided C++ function `scaleAddCPU`, the operation involves scaling each element of a given matrix (`pMat`) by a scalar `a` and then adding another scalar `b` to each scaled element, with the result stored in a separate matrix (`pResult`). The function proceeds to iterate through each element of the matrix, applying the formula `a * pMat->data[i] + b` to compute the corresponding element in the result matrix. The loop iterates `len` times, where `len` is the total number of elements in the matrix, calculated as the product of the number of rows and columns (`rows * cols`).

GPU Implementation

```
__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;
}
bool scaleAddGPU(const Matrix * pMat, Matrix * pResult, float a, float b)
{
//some check...
    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);
//some check..
    cudaMemcpy(pResult->data, pResult->data_device, sizeof(float)*len,
cudaMemcpyDeviceToHost);
    return true;
}
```

The `scaleAddKernel` is a GPU kernel that performs the operation in parallel. It is launched with an appropriate configuration to map each matrix element to a GPU thread, which computes the new value by applying the formula `a * input[i] + b`.

In the `scaleAddGPU` function, GPU acceleration is utilized to enhance the performance of the scaling and addition operation on matrix data. The process begins with ensuring the input and result matrices are not null and are of the same size. The matrix data is then transferred from the host (CPU) to the GPU device memory using `cudaMemcpy`.

After the GPU computation, the updated matrix data is transferred back to the host memory. The function includes error handling to check for issues during data transfer or kernel execution. If all steps are completed successfully, the function returns `true`, indicating that the GPU-accelerated operation has been performed.

2.3 Scale Multiplication

```
void initializeMatrix(float* matrix, int N) {
    for (int i = 0; i < N * N; i++) {
        matrix[i] = static_cast<float>(rand()) / RAND_MAX;
    }
}
```

For multiplication, the `initializeMatrix` function populates a matrix of size N×N with random floating-point values between 0 and 1.

CPU Implementation

```
blas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1.0, A, N, B, N, 0.0, C,
N);
```

The `cblas_sgemm` function is part of the BLAS (Basic Linear Algebra Subprograms) library, specifically within the OpenBLAS implementation. The `cblas_sgemm` function performs a single-precision general matrix multiplication (SGEMM).

GPU Implementation

```
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, N, N, &alpha, d_A, N, d_B, N, &beta, d_C,
N);
```

Uses the cuBLAS library and the `cublasSgemm` function to perform matrix multiplication on the GPU. The `cublasSgemm` function is a method provided by the cuBLAS library to perform single-precision general matrix multiplication (SGEMM). This function computes the matrix product of two matrices with single-precision floating-point elements.

3. Performance Comparison

3.1 Scale Addition

The provided macros `TIME_START` and `TIME_END` are used to measure the execution time of a code block in milliseconds. `TIME_START` records the start time, while `TIME_END` calculates the elapsed time and prints the result in milliseconds along with the provided name.

```
#define TIME_START gettimeofday(&t_start, NULL);
#define TIME_END(name)    gettimeofday(&t_end, NULL); \
                    elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;   \
                    elapsedTime += (t_end.tv_usec - t_start.tv_usec) / 1000.0;  \
                    printf(#name " Time = %f ms.\n", elapsedTime);
```

For the addition operation, we performed matrix addition on a 4096x4096 matrix using both the CPU and GPU. The operation is defined as B=aA+b, where a and b are scalars. Each computation was repeated 5 times for both the CPU and GPU. The results of these operations are as follows:

trial	CPU time (ms)	GPU time (ms)
1	59.08	22.13
2	57.09	21.53
3	60.13	22.17
4	55.79	20.76
5	59.75	21.43
avg	**58.37**	**21.60**

The average execution times for scale addition task for CPUs is 58.37ms, and for GPUs is 21.60ms. We can notice that GPUs performs about 3 times faster than CPUs. This implies that GPUs leverage parallel computing to achieve high throughput and performance by executing thousands of threads concurrently.

3.2 Scale Multiplication

The `cudaEventRecord()` function in CUDA is used to record a CUDA event, which essentially marks a point in time within the GPU's execution timeline. This function is often used to measure the elapsed time of GPU operations.

```
cudaEventRecord(start);
cublasSgemm(...);
cudaEventRecord(stop);
```

For the multiplication operation, we conducted matrix multiplication on a 4096x4096 matrix using both the CPU and GPU. The operation is defined as C=A×B, where A and B are matrices and C is the resulting matrix. Each computation was repeated 5 times for both the CPU and GPU. The results of these operations are as follows:

trial	CPU time (ms)	GPU time (ms)
1	108.08	18.11
2	102.89	17.91
3	102.17	17.80
4	102.95	17.86
5	106.89	17.90
avg	**104.59**	**17.91**

The average execution times for scale multiplication task for CPUs is 104.59ms, and for GPUs is 17.91ms. CPUs' perform time is 5.84 times of GPUs'. It is interesting that for GPUs, the multiplication is even faster than addition. This may due to powerful parallel computing capability, specialized hardware acceleration, high memory bandwidth, optimized libraries and algorithms, as well as the combined effects of stream processing and asynchronous execution.