Practice 2

Name: 魏程涌潇

Student ID: 515030910598

Practical 5: CUBLAS and CUFFT libraries

1. Make and run the two programs and the results are:

```
GPU Device 0: "Tesla K20m" with compute capability 3.5

simpleCUBLAS test running..
simpleCUBLAS test passed.
[simpleCUFFT] is starting...
GPU Device 0: "Tesla K20m" with compute capability 3.5

Transforming signal cufftExecC2C
Launching ComplexPointwiseMulAndScale<<< >>>
Transforming signal back cufftExecC2C
```

   From the above results, we shall see that the error between the library result and the corresponding CPU “Gold” code is very trivial.

2. Summarize the steps of using the interfaces：

   CUBLAS

   The CUBLAS library is an implementation of **BLAS (Basic Linear Algebra Subprograms)** on top of the NVIDIA®CUDA™ runtime. It allows the user to access the computational resources of NVIDIA Graphics Processing Unit (GPU).

   According to the source code, by including the header file `cublas_v2.h`, it uses the updated API. The basic steps of its usage are as follows:

   1. create a handle using `cublasCreateHandle`
   2. allocate resources using `cudaMalloc`
   3. fill data into device using `cublasSetVector` and `cublasSetMartix`
   4. perform matrix and vector multiplication using `cublasSgemv` or `cublasSgemm`
   5. obtain the results using `cublasGetVector`
   6. release the resources

   CUFFT

   CUFFT, the NVIDIA® CUDA™ **Fast Fourier Transform (FFT)** product, consists of two separate libraries: CUFFT and CUFFTW. The CUFFT library is designed to provide high performance on NVIDIA GPUs. The CUFFTW library is provided as a porting tool to enable users of FFTW to start using NVIDIA GPUs with a minimum amount of effort.

According to the source code, by including the header file `cufft.h`, it uses the CUFFT library. The basic steps of its usage are as follows:

1. create a handle using `cufftHandle`
2. configure the handle using `cufftPlan1d()`, `cufftPlan3d()`, `cufftPlan3d()`, `cufftPlanMany()`, mainly to configure the signal length corresponding to the handle, signal type, storage form in memory and some other information.
3. execute fft using `cufftExec()`
4. release GPU resources using `cufftDestroy()`

Practical 6:pattern-matching

Run the CPU version program and the results are:

```
CPU matches = 3 3 22 42
```

For the GPU version, I transfer the match codes into:

```
__global__ void match_g(unsigned int *text, unsigned int *words, int
*matches, int nwords, int length){
        /*get the absolute tid*/
        int tid = threadIdx.x + blockIdx.x*blockDim.x;
        unsigned int word;

        if (tid < length){
            for (int offset=0; offset<4; offset++) {
                if (offset==0)
                    word = text[tid];
                else
                    word = (text[tid]>>(8*offset)) + (text[tid+1]<<(32-
8*offset));


                for (int w=0; w<nwords; w++) {
                    if(word == words[w])
                      atomicAdd(matches+w, 1);
                }
            }
        }
}
```

Note that we should use `atomicAdd` here to update the value in `matches`, otherwise, we'll get wrong results instead. The source code is compressed in the zip file and will be not put here. The results are as follows:

```
 CPU matches = 3 3 22 42
GPU Device 0: "Tesla K20m" with compute capability 3.5

 GPU matches = 3 3 22 42
```

Practical 7: scan operation and recurrence equations

After refering to the paper *A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations*, I learned the algorithm to solve the recurrence equations.

The key point is:

in every pass, $u_n = u_{n-1} \times s_n + u_n$ and $s_n = s_n \times s_{n-1}$ where $s_1 = 1$.

Then after $\lceil \log_2 N \rceil$ passes, $u_{\{1..n\}}$ is just $V_{1..n}$, the values we are seeking.

Single thread block:

The kernel codes:

```
__global__ void scan(float *u_idata, float *s_idata, float *result)
{
    // dynamically allocated shared memory
    // temp for cofficience, tempq for scalar

    extern  __shared__  float temp[];
    int tid = threadIdx.x;
    float old_un, old_sn, cur_un, cur_sn, new_un, new_sn;

    // first, each thread loads data into shared memory+

    temp[tid] = s_idata[tid];
    temp[tid+blockDim.x]= u_idata[tid];

    // printf("input s%.2f u%.2f\n",temp[tid],temp[tid+blockDim.x]);

    for (int d = 1; d < blockDim.x; d <<= 1) {
        __syncthreads();
        if (tid >= d) {
            // copy shared variable into thread local variable

            old_un = temp[tid + blockDim.x - d];
            old_sn = temp[tid - d];
            cur_un = temp[tid + blockDim.x];
            cur_sn = temp[tid];
```

```
            // calculation

            new_sn = old_sn * cur_sn;
            new_un = old_un * cur_sn + cur_un;

            // write back

            __syncthreads();
            temp[tid + blockDim.x] = new_un;
            temp[tid] = new_sn;
            // printf("tid%d d%d s%.2f
u%.2f\n",tid,d,temp[tid+d+blockDim.x],temp[tid+d]);
        }
    }
    __syncthreads();
    new_un = temp[tid + blockDim.x];
    result[tid]=temp[tid+blockDim.x];
    // printf("id%d  %.2f\n",tid,result[tid]);
}
```

The results are as follows:

```
GPU Device 0: "Tesla K20m" with compute capability 3.5

gpu result, cpu result = 0.356286, 0.356286
gpu result, cpu result = 0.576100, 0.576100
gpu result, cpu result = 0.251505, 0.251505
gpu result, cpu result = 0.734573, 0.734573
gpu result, cpu result = 0.530169, 0.530169
gpu result, cpu result = 0.511153, 0.511153
gpu result, cpu result = 0.286515, 0.286515
gpu result, cpu result = 0.247815, 0.247815
gpu result, cpu result = 0.653083, 0.653083
gpu result, cpu result = 0.752044, 0.752044
gpu result, cpu result = 0.116572, 0.116572
gpu result, cpu result = 1.044332, 1.044332
gpu result, cpu result = 0.577464, 0.577464
gpu result, cpu result = 1.331560, 1.331560
gpu result, cpu result = 1.070298, 1.070298
gpu result, cpu result = 0.428092, 0.428092
gpu result, cpu result = 1.029723, 1.029723
gpu result, cpu result = 0.402940, 0.402940
gpu result, cpu result = 0.995624, 0.995624
gpu result, cpu result = 0.710418, 0.710419
gpu result, cpu result = 0.820290, 0.820290
gpu result, cpu result = 0.621932, 0.621932
gpu result, cpu result = 0.303677, 0.303677
gpu result, cpu result = 0.418447, 0.418447
gpu result, cpu result = 0.684415, 0.684415
```

```
gpu result, cpu result = 0.162205, 0.162205
gpu result, cpu result = 0.440205, 0.440205
gpu result, cpu result = 0.798509, 0.798509
gpu result, cpu result = 1.391823, 1.391823
gpu result, cpu result = 1.004442, 1.004442
gpu result, cpu result = 0.951991, 0.951991
gpu result, cpu result = 1.576078, 1.576078
scan error  = 0.000000
```

I have calculated the standard error between the gpu results and the cpu results, which named *scan error* here, valuing **0.000000** with single thread block.

Multiple thread blocks:

Since shared memory is an inner concept in blocks, we can't simply apply the shared memory like the method in single thread block. However, we can use **global memory** to deal with the problem, though may **hinder the effectiveness**. The kernel codes are:

```
__global__ void scan(float *u_idata, float *s_idata, float *tempu, float
*temps, float *result)
{
    // temp for cofficience, tempq for scalar

    // extern  __shared__  float temp[];
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float old_un, old_sn, cur_un, cur_sn, new_un, new_sn;

    cur_un = u_idata[tid];
    cur_sn = s_idata[tid];
    tempu[tid] = cur_un;
    temps[tid] = cur_sn;
    // printf("input s%.2f u%.2f\n",temp[tid],temp[tid+blockDim.x]);

    for (int d = 1; d < blockDim.x; d <<= 1) {
        __threadfence();
        __syncthreads();
        if (tid >= d) {
            // copy global variables into thread local variable

            old_un = tempu[tid - d];
            old_sn = temps[tid - d];
            cur_un = tempu[tid];
            cur_sn = temps[tid];

            // calculation

            new_sn = old_sn * cur_sn;
            new_un = old_un * cur_sn + cur_un;
```

```
            // write back

            __threadfence();
            __syncthreads();
            atomicExch(tempu + tid, new_un);
            atomicExch(temps + tid, new_sn);
            // printf("tid%d d%d s%.2f
u%.2f\n",tid,d,temp[tid+d+blockDim.x],temp[tid+d]);
        }
    }
    __syncthreads();
    __threadfence();
    new_un = tempu[tid];
    result[tid] = new_un;
    // printf("id%d  %.2f\n",tid,result[tid]);
}
```

And the results are as follows:

```
GPU Device 0: "Tesla K20m" with compute capability 3.5

gpu result, cpu result = 0.932396, 0.932396
gpu result, cpu result = 1.562571, 1.562571
gpu result, cpu result = 0.712177, 0.712177
gpu result, cpu result = 0.879139, 0.879139
gpu result, cpu result = 0.561586, 0.561586
gpu result, cpu result = 0.636606, 0.636606
gpu result, cpu result = 0.265313, 0.265313
gpu result, cpu result = 1.052314, 1.052314
gpu result, cpu result = 1.475810, 1.476784
gpu result, cpu result = 0.164060, 0.164077
gpu result, cpu result = 0.814484, 0.814485
gpu result, cpu result = 0.803813, 0.803814
gpu result, cpu result = 1.326338, 1.326339
gpu result, cpu result = 1.562672, 1.562674
gpu result, cpu result = 2.146781, 2.146785
gpu result, cpu result = 2.218265, 2.218285
gpu result, cpu result = 1.002787, 1.002798
gpu result, cpu result = 1.288368, 1.288468
gpu result, cpu result = 1.769237, 1.799178
gpu result, cpu result = 1.630115, 1.653527
gpu result, cpu result = 0.527291, 0.535894
gpu result, cpu result = 1.094948, 1.104749
gpu result, cpu result = 1.685829, 1.699080
gpu result, cpu result = 1.643164, 1.651133
gpu result, cpu result = 1.158044, 1.162228
gpu result, cpu result = 1.356713, 1.362984
gpu result, cpu result = 0.987969, 0.995748
gpu result, cpu result = 1.358311, 1.363020
```

```
gpu result, cpu result = 1.107874, 1.112989
gpu result, cpu result = 0.937163, 0.939039
gpu result, cpu result = 1.315385, 1.317623
gpu result, cpu result = 1.685325, 1.688964
scan error  = 0.007989
```

From the results, we shall see the *scan error*, the standard error between gpu results and cpu results, values **0.007989**. After continuously thinking about it, I speculate that the order differs in multiplication of float point number will cause some accuracy errors. But this can't explain why there are no obvious accuracy errors in single thread block's implementation.