ECE5720 HW4

Chen-Tung Chu cc2396

1. General Remarks

In this assignment, we are asked to solve a least square problem via truncated SVD by using CUDA. For a given matrix, we want to find an approximate solution x by solving the least square equation $argmin_{x \in R^n} ||Ax - b||_2$. We let the vector b be a single numeric vector with value of ones.

After having matrices A, b, and x, we can start calculating the SVD of matrix A. For a $n \times n$ Matrix A, it can be decomposed into $A = U\Sigma V^T$, where U and V are $n \times n$ orthogonal matrices, and Σ is an $n \times n$ diagonal matrix with the ordered elements $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$.

To find the solution x, we have two kinds of solving methods. The first one is implementing the matrix-matrix multiply method. We multiply matrix V, inverse matrix Σ^{-1}, transpose matrix U^T, and matrix b. The equation is as follows: $x = V\Sigma^{-1}U^T b$; another method is the iterative method. We can solve x by implementing the equation $\sum_{i=1}^{n} \frac{u_i^T b}{\sigma_i} u_i$, I used the second method to find the approximate solution. We can find a certain integer k that $\sigma_k \gg \sigma_{k+1}, 1 < k < n$, in our assignment case, we can find the first k for which $\frac{\sigma_{k+1}}{\sigma_k} \leq 10^{-3}$. Once we find the value k, we can collect the first k columns of U and V, and the leading k singular values from Σ, then we can calculate the truncated SVD $A_k = U_k \Sigma_k V_k^T$. For the truncated SVD decomposition we will find the error $e = ||x - x_k||_2$ and the residual error $r = ||A_k x_k - b||_2$.

2. CUDA Implementation

Once we know the background mechanism, we can start to implement the aforementioned instructions with CUDA.

The first thing is the initialization of the CUBLAS and CUSOLVER. Also, I initialized the timer to count the whole process time. After the initialization, we can start to import the matrix from `MyMatrix.txt` file through `read_matrix` function and mark it as the variable name `h_A`, which means matrix A on host. By following the steps of the instruction the Professor gave us, now we have initialized matrices $A, x,$ and b for both and device. For the memory allocation for these metrices on the device, we use `cudaMalloc` function and `cudaMemcpy` to copy the data from the host to the device. Figure 1 is the code.

```
113     // read matrix and initialize matrices for host and device
114     int row, col;
115     float *h_A = read_matrix("MyMatrix.txt", &row, &col);
116     printf("Matrix size: %d * %d\n\n", row, col);
117     float *h_x = (float*)malloc(sizeof(float)*col);
118     for (int i = 0; i < col; i++)
119     {
120         h_x[i] = 1.0;
121     }
122     float *h_b = (float*)malloc(sizeof(float)*row);
123     float *d_A, *d_b, *d_x;
124     cudaMalloc((void**)&d_A, sizeof(float)*row*col);
125     cudaMalloc((void**)&d_b, sizeof(float)*row);
126     cudaMalloc((void**)&d_x, sizeof(float)*col);
127     cudaMemcpy(d_A, h_A, sizeof(float)*row*col, cudaMemcpyHostToDevice);
128     cudaMemcpy(d_x, h_x, sizeof(float)*col, cudaMemcpyHostToDevice);
```

Figure 1. Code for matrix initialization

Now we have successfully initialized the matrices A, x, and b, we can now start the matrix-vector multiplication for calculating $b = Ax$ by using the library `cublasSgemv`. The code is showed in Figure 2.

```
130     // b = Ax
131     cublasSetMatrix(row, col, sizeof(float), h_A, row, d_A, row);
132     cublasSetVector(col, sizeof(float), h_x, 1, d_x, 1);
133     cublas_status = cublasSgemv(cublasH, CUBLAS_OP_N, row, col, &alpha, d_A, row, d_x, 1, &beta, d_b, 1);
```

Figure 2. b=Ax matrix-vextor multiplication

For the next step, we now can find the SVD of the matrix A. By using `cuSolver` library, we followed the equation $A = U\Sigma V^T$ to find the singular values. However, we need to calculate the size of work buffer first. I first initiated the work buffer and buffer size, as well as the matrices Σ, U, and V^T. After the memory allocation on the device, we now can start to find the SVD of A using the `cusolverDnSgevd` library. The description is described in Figure 3.

```
135     // A = U*S*V^T
136     float *buffer;
137     int bufferSize = 0;
138     cusolver_status = cusolverDnSgesvd_bufferSize(cusolverH, row, col, &bufferSize);
139     cudaMalloc((void**)&buffer, sizeof(float)*bufferSize);
140     float *d_S, *d_U, *d_VT;
141     int *devInfo;
142     cudaMalloc((void**)&devInfo, sizeof(int));
143     cudaMalloc((void**)&d_S, sizeof(float)*col);
144     cudaMalloc((void**)&d_U, sizeof(float)*row*row);
145     cudaMalloc((void**)&d_VT, sizeof(float)*col*col);
146     cusolver_status = cusolverDnSgesvd(cusolverH, 'A', 'A', row, col, d_A, row, d_S, d_U, row, d_VT, col, buffer, bufferSize, NULL, devInfo);
```

Figure 3. Find the SVD of matrix A

With the singular value decomposition, we now find the value k. In this assignment case, Professor provided us the formula for finding the first k, that is $\frac{\sigma_{k+1}}{\sigma_k} \leq 10^{-3}$. We use the matrix Σ from the device and implement the function above in a for-loop. Once we find the satisfied k, we break the loop. To make it uniform with the value k in MATLAB, we add increment k with one. The code is showed in Figure 4.

```
148     // find k
149     int k = 1;
150     float *h_s = (float*)malloc(sizeof(float)*col);
151     cublasGetVector(col, sizeof(float), d_S, 1, h_s, 1);
152     for (int i = 0; i < col - 1; i++){
153         if ((h_s[i] / h_s[i+1]) < 0.001){     // find the first k
154             k = i;
155             break;
156         }
157     }
158     k += 1;     // to let k is the same as int MATLAB
```

Figure 4. Find k

Now we can form the U_k, V_k, and Σ_k on the device for computing the matrix A_k, using the equation $A_k = U_k \Sigma_k V_k^T$. I implemented three kernel function on the device to get the matrices: `get_S_k`, `get_U_k`, and `get_V_k`. For `get_S_k`, the diagonal elements we will save the value from the matrix Σ from the device. For `get_U_k` and `get_V_k`, they are the similar methodology. Through these kernel functions, we now get U_k, V_k, and Σ_k. Figure 5 is the main code to form U_k, V_k, and Σ_k and Figure 6 is the function code.

```
161     // Form U_k, V_k, S_k
162     float *d_Ak, *d_Uk, *d_Vk, *d_Sk;
163     cudaMalloc((void**)&d_Ak, sizeof(float)*row*col);
164     cudaMalloc((void**)&d_Uk, sizeof(float)*row*k);
165     cudaMalloc((void**)&d_Vk, sizeof(float)*col*k);
166     cudaMalloc((void**)&d_Sk, sizeof(float)*k*k);
167     get_S_k<<<k, 1>>>(d_S, d_Sk, k, k);
168     get_U_k<<<k, 1>>>(d_U, d_Uk, k, row);
169     get_V_k<<<k, 1>>>(d_VT, d_Vk, k, col);
```

Figure 5. Main code to form U_k, V_k, and S_k

```
__global__ void get_S_k(float *S, float *S_k, int k, int row)
{
    int i = blockIdx.x;
    if (i < k){
        for (int j = i * row; j < i * row + row; j++){
            // diagonal element
            if(j == i * row + i){
                S_k[i * row + i] = S[i];
            }
            else{
                S_k[j] = 0;
            }
        }
    }
}
__global__ void get_U_k(float *U, float *U_k, int k, int row)
{
    int i = blockIdx.x;
    if (i < k){
        for (int j = i * row; j < i * row + row; j++){
            U_k[j] = U[j];
        }
    }
}
__global__ void get_V_k(float *V, float *V_k, int k, int col)
{
    int i = blockIdx.x;
    if (i < k){
        for (int j = i * col; j < i * col + col; j++){
            V_k[j] = V[j];
        }
    }
}
```

Figure 6. Function codes

The next step is to find x_k using the matrix-vector multiplication routine `cublasSgemv` to form $b_k = U_k^T b$. Following the instruction steps that provided by the Professor, I wrote a CUDA kernel function called `get_x` to scale the elements of b_k by the corresponding values. Following the solving method from the equation $x_k = \sum_{i=1}^{n} \frac{u_i^T b}{\sigma_i} u_i$, as for the thread configuration, the matrix read from `MyMatrix.txt` has the dimension of 256×128, so the block size I set was `dim3 block(16,16)`, and for the grid dimension, we need 256 threads in x dimension and 128 threads in y dimension. Since we set the block size is 16×16, we need 16 blocks in x dimension and 8 blocks in y dimension, so the grid size I set was `dim3 grid(16,8)`. Once we have the modified b_k, we can compute $d_k = \Sigma_k^{-1} U_k^T b_k$ by using two `cublasSgemm` library functions to compute d_k. Figure 7 is the `get_x` function and Figure 8 is the main code of `get_x` and the computation of $\Sigma_k^{-1} U_k^T b_k$.

```
__global__ void get_x(float *x, float *UtB, float *sigma, float *V, int N, int col)
{
    int i = blockIdx.x;
    if (i < N){
        for (int j = 0; j < col; j++){
            x[i] += UtB[j] * V[j*N + i] / sigma[j];
        }
    }
}
```

Figure 7. get_x function

```
185     // get xk
186     float *d_xk;
187     cudaMalloc((void**)&d_xk, sizeof(float)*col);
188     get_x<<<128, 1>>>(d_xk, d_UT_b, d_S, d_V, col, k);
189
190
191     // U_k * S_k * V_k^T
192     float *d_Uk_Sk;
193     cudaMalloc((void**)&d_Uk_Sk, sizeof(float)*row*k);
194     cublasSgemm(cublasH, CUBLAS_OP_N, CUBLAS_OP_N, row, k, k, &alpha, d_Uk, row, d_Sk, k, &beta, d_Uk_Sk, row);
195     cublasSgemm(cublasH, CUBLAS_OP_N, CUBLAS_OP_T, row, col, k, &alpha, d_Uk_Sk, row, d_Vk, col, &beta, d_Ak, row);
196     float *d_x_copy;
197     cudaMalloc((void**)&d_x_copy, sizeof(float)*col);
198     cudaMemcpy(d_x_copy, d_real_x, sizeof(float)*col, cudaMemcpyDeviceToDevice);
199     float neg_alpha = -1.0f;
200     cublasSaxpy(cublasH, col, &neg_alpha, d_xk, 1, d_x_copy, 1);
```

*Figure 8. Main function of get_x and U_k*S_k*V_k^T*

Now we can proceed to the final step, calculate the error, residual error and print out the first eight entries of x_k. The first thing is to move the x_k, error, and the residual error from the device to host. Once we move the data to the host, we can compute the error and the residual error by simply call the `cublasSaxpy` and `cublasSnrm2` functions. Figure 9 is the code to compute the error and the residual error.

```
203     // compute error
204     float error;
205     float *d_Ak_xk;
206     cublasSnrm2(cublasH, col, d_x_copy, 1, &error);
207     cudaMalloc((void**)&d_Ak_xk, sizeof(float)*row);
208     cudaMemcpy(d_A, h_A, sizeof(float)*row*col, cudaMemcpyHostToDevice);
209
210     // compute residual error
211     float residual_error;
212     cublas_status = cublasSgemv(cublasH, CUBLAS_OP_N, row, col, &alpha, d_Ak, row, d_xk, 1, &beta, d_Ak_xk, 1);
213     cublasSaxpy(cublasH, row, &neg_alpha, d_b, 1, d_Ak_xk, 1);
214     cublasSnrm2(cublasH, row, d_Ak_xk, 1, &residual_error);
215     cudaDeviceSynchronize();
```

Figure 9. Compute error and residual error

3. Results

For the results, I have printed out the error, the residual error, and the first eight entries. Also, I calculated the total processing time. Figure 10 shows the whole results.

```
Matrix size: 256 * 128

Error: 1.123e+01

Residual error: 1.325e+00

First 8 entries of x_k:
-3.800e-11
-6.293e-08
-5.624e-08
1.430e-10
-1.105e-09
1.423e-09
-1.064e-09
4.123e-10

Time elapsed: 48.279 ms
```

Figure 10. Results

For more inspection and evaluation of the impact of different grid size and block size, I have experimented the computation with different grid size and block size to see the process time difference and the errors. Table 1 shows the experimental results.

	grid(8,8) block(8,8)	grid(16,16) block(8,8)	grid(16,16) block(16,8)	grid(16,16) block(16,16)	grid(16,16) block(32,32)
Error	1.123e+01	1.123e+01	1.123e+01	1.123e+01	1.123e+01
Residual error	1.325e+00	1.325e+00	1.325e+00	1.325e+00	1.325e+00
Elapsed time (ms)	52.600	51.682	50.649	48.905	47.994

Table 1. Experiment with different grid and block size

Since the GPU we use for the virtual machine is NVIDIA® Tesla® K40 processor, the maximum thread per lock is 1024, so my experimental setup for the number of threads per block is 1024. From the table above we can see that for more threads per block, the processing time reduces, as the block size increased to the maximum level, we have a relatively shorter elapsed time. On the other time, as we cut down the number of threads per block, it is obvious that the process time grows.

4. Conclusion

In conclusion, combined with the previous homework with openMPI implementation, from the conventional serial processing method to compute SVD, we can see that as the matrix size becomes larger and larger, the machine cannot handle such memory requirements, eventually the process been killed. With openMPI parallelism, we can see that parallelism truly solved the problem happened in the serial method. However, the performance reflects that we can have a better performance with the help of GPU. In this assignment, with the help of GPU, we can compare the performance difference with the same dimension of a specific matrix. Take a 256×128 matrix for example, to compute the singular value decomposition matrix, the overall process time with openMPI implementation is 17256.303ms; and with CUDA implementation, the overall process time is merely 50.649ms, approximately x340 the speed. From serial to parallel, and get further to GPU support, we can observe the evolution of the speed of computing, especially in matrix calculation.

5. Reference

[1] https://docs.nvidia.com/cuda/cusolver/index.html
[2] https://docs.nvidia.com/cuda/cublas/index.html
[3] https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html
[4] https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html
[5] https://linux.die.net/man/2/gettimeofday
[6] Course lectures