HPC HW4

maoyiluo

April 2020

1 Matrix-vector operations on a GPU

To implement a inner product operations on GPU, we have to separate this operation into two step:

1. computes $c_i = a_i * b_i$.
2. sums up c_i

So the first step is on each thread compute $a_i * b_i$ for the corresponding thread id, then we do the reduction on each block. At the end of our function, we use atomicAdd to sums up the result from each block.

```
__global__
void vec_inner_product_kernel(double* res, const double* a, const double* b, long N){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ double temp[blockSize];
    if (idx < N) temp[threadIdx.x] = a[idx] * b[idx];
    int i = blockSize/2;
    __syncthreads();
    while(i != 0){
        if(threadIdx.x < i){
            temp[threadIdx.x] += temp[threadIdx.x + i];
        }
        __syncthreads();
        i /= 2;
    }

    if(threadIdx.x == 0){
        atomicAdd(res, temp[0]);
    }
}
```

To implement a matrix times vector version, we need to maintain a 2D cache array, to store $C_{i,j} = a_{i,j} * b_j$. Then we do the reduction on each row of this cache array.

```
__global__
void matrix_vec_product_kernel(double* res, const double* a, const double* b, long N){
    long col = blockIdx.x * blockDim.x + threadIdx.x;
    long row = blockIdx.y * blockDim.y + threadIdx.y;
    __shared__ double temp[blockSize][blockSize];
    if (row < N && col < N) temp[threadIdx.y][threadIdx.x] = a[row*N +col] * b[col];
    __syncthreads();

        for(int y = 0; y < blockSize; y++){
        int i = blockSize/2;
        while(i != 0){
            if(threadIdx.x < i){
                temp[y][threadIdx.x] += temp[y][threadIdx.x + i];
            }
            __syncthreads();
```

```
            i /= 2;
        }
    }
    __syncthreads();
    if(threadIdx.x == 0 && threadIdx.y == 0){
        for(int y = 0; y < blockSize; y++)
                    atomicAdd(&(res[row+y]), temp[y][0]);
    }
}
```

Table 1 shows the bandwidth of my algorithm on different GPU.
For unknown reason, the test on the 2080ti machine failed even for a small matrix(I'm guessing it's

	GPU	CPU
GTX TITAN Black	0.151307 GB/s	0.001409 GB/s
GeForce RTX 2080 Ti	ERROR: malloc x failed: out of memory	0.005486 GB/s
GTX TITAN V	0.006497 GB/s	0.000065 GB/s

Table 1: Bandwidth of the matrix vector multiplication on different GPU.

because some one else is running some intensive task). But we can see the improvement from CPU to GPU.

2 2D Jacobi method on a GPU

To implement a 2D Jacobi method, I defined two kernel function, one for the iteration step (gain u_{k+1} from u_k) and one for the update step (assign u_{k+1} to the new u_k).

```
__global__
void iterate(double* u_kp1, double* u_k, double* f, double *h_c){
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    double h = *h_c;
    if(row >=1 && row <= N && col >=1 && col <= N){
        u_kp1[row*(N+2) + col] = 1.0/4*(h*h*f[(row-1)*N + col]
        + u_k[(row-1)*(N+2) + col] + u_k[row*(N+2)
        + col - 1] + u_k[(row+1)*(N+2) + col] + u_k[row*(N+2) + col+1]);
        //printf("%f\n", u_kp1[row*(N+2) + col]);
    }
}

__global__
void update(double* u_kp1, double* u_k){
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if(row >=1 && row <= N && col >=1 && col <= N){
        u_k[row*(N+2) + col] = u_kp1[row*(N+2) + col];
    }
}
```

To verify our implementation we can see that whether the iteration decrease the residual or not. Here is one of my test.

```
[ml6363@cuda3 hw4]$ ./p2
inital_residual = 6801622670.369197 last residual = 351633723.209747
```

We can see that it does decrease.

3 project

I'm still at the first part of implementing the sequential version. I was busy with preparing interview and looking for a job for the pass few weeks so I didn't have much time for . My current goal is finish the sequential and openmp part before the end of this week.