CSED490C Heterogeneous Parallel Computing

Lab7 Report

20232241 Dookyung Kang

#1. This is my version of template.

```
// macros
#define BLOCK_SIZE 256
#define SMEM_SIZE 1024
#define FUNC SPMV_JDS


// kernel (global)
__global__ void SPMV_JDS (int rows, int numMaxCols, float* data, int *columnIndices, int *jdsTColPtrs, int* jdsRowIndices, float *b, float *c)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < rows) {
        c[tid] = 0.0f;
    }
    __syncthreads();
    for (int i = blockDim.x * blockIdx.x; i < min(blockDim.x * (blockIdx.x + 1), numMaxCols); i++) {
        int extent = jdsTColPtrs[i + 1] - jdsTColPtrs[i];
        for (int j = threadIdx.x; j < extent; j += blockDim.x) {
            int k = j + jdsTColPtrs[i];
            atomicAdd(&c[jdsRowIndices[j]], data[k] * b[columnIndices[k]]);
        }
    }
}

// kernel (shared)
__global__ void SPMV_JDS_SHARED (int rows, int numMaxCols, float* data, int *columnIndices, int *jdsTColPtrs, int* jdsRowIndices, float *b,
float *c) {
    __shared__ int columnIndicesShared[SMEM_SIZE];
    __shared__ float dataShared[SMEM_SIZE];
    __shared__ int jdsTColPtrsShared[BLOCK_SIZE];
    __shared__ int jdsTColPtrSubShared[BLOCK_SIZE];

    unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;

    if (tid < numMaxCols) {
        jdsTColPtrsShared[threadIdx.x] = jdsTColPtrs[tid];
        jdsTColPtrSubShared[threadIdx.x] = jdsTColPtrs[tid + 1] - jdsTColPtrs[tid];
    }

    if (tid < rows) {
        c[tid] = 0.0f;
    }
    __syncthreads();

    for (unsigned int i = 0; i < blockDim.x; i++) {
        if (blockDim.x * blockIdx.x + i >= numMaxCols) break;
        int extent = jdsTColPtrSubShared[i];
        for (int chunk = 0; chunk < (extent + SMEM_SIZE - 1) / SMEM_SIZE; chunk++) {
            int chunk_size = min(SMEM_SIZE, extent - chunk * SMEM_SIZE);
            for (int j = threadIdx.x; j < chunk_size; j += blockDim.x) {
                int k = chunk * SMEM_SIZE + j + jdsTColPtrsShared[i];
                dataShared[j] = data[k];
                columnIndicesShared[j] = columnIndices[k];
```

```
            }
            __syncthreads();
            for (int j = threadIdx.x; j < chunk_size; j += blockDim.x) {
                atomicAdd(&c[jdsRowIndices[chunk * SMEM_SIZE + j]], dataShared[j] * b[columnIndicesShared[j]]);
            }
            __syncthreads();
        }
    }
}

int main(int argc, char **argv) {
    gpuTKArg_t args;
    float *hostA; // The A matrix
    float *hostB; // The B matrix
    float *hostC; // The output C matrix
    // float *deviceA;
    float* deviceNonzeroValues;
    int* deviceColumnIndices;
    int* deviceJdsRowIndices;
    int* deviceJdsTColPtrs;
    /** skip **/
    numCRows        = numARows;
    numCColumns = numBColumns;
    //@@ Allocate the hostC matrix
    hostC = (float *)malloc(sizeof(float) * numCRows * numCColumns);
    gpuTKTime_stop(Generic, "Importing data and creating memory on host");
    gpuTKCheck(cudaMalloc((void **)&deviceC, numCRows * numCColumns * sizeof(float)));

    gpuTKLog(TRACE, "The dimensions of A are ", numARows, " x ", numAColumns);
    gpuTKLog(TRACE, "The dimensions of B are ", numBRows, " x ", numBColumns);

    gpuTKTime_start(GPU, "Converting matrix A to JDS format (transposed).");

    //@@ Create JDS format data
    std::vector<jds_row_type> jdsRows;
    jdsRows.reserve(numARows);
    int nonzeroCnt = 0;
    for (int i = 0; i < numARows; i++) {
        std::vector<std::pair<int, float>> nonzeroPairs;
        int localNonzeroCnt = 0;
        for (int j = 0; j < numAColumns; j++) {
            if (hostA[i * numAColumns + j] != 0.0f) {
                localNonzeroCnt++;
            }
        }
        nonzeroPairs.reserve(localNonzeroCnt);
        for (int j = 0; j < numAColumns; j++) {
            if (hostA[i * numAColumns + j] != 0.0f) {
                nonzeroPairs.push_back(std::pair<int, float>(j, hostA[i * numAColumns + j]));
            }
        }
        nonzeroCnt += localNonzeroCnt;
        jdsRows.push_back({ i, nonzeroPairs });
    }
    std::sort(jdsRows.begin(), jdsRows.end(),
        [](const jds_row_type& a, const jds_row_type& b) { return a.second.size() > b.second.size(); });

    int numMaxCols = jdsRows[0].second.size();
    float* hostNonzeroValues = (float*)malloc(sizeof(float) * nonzeroCnt);
    int* hostColumnIndices = (int*)malloc(sizeof(int) * nonzeroCnt);
```

```
int* hostJdsRowIndices = (int*)malloc(sizeof(int) * numARows);
int* hostJdsTColPtrs = (int*)malloc(sizeof(int) * (numMaxCols + 1));

hostJdsTColPtrs[0] = 0;
int j = 1, i = numARows - 1, prev = 0;
while (j <= numMaxCols && i >= 0) {
   if (jdsRows[i].second.size() >= j) {
      prev += i + 1;
      hostJdsTColPtrs[j++] = prev;
   } else {
      i--;
   }
}

for (int row = 0; row < numARows; row++) {
   hostJdsRowIndices[row] = jdsRows[row].first;
   std::vector<std::pair<int, float>> pairs = jdsRows[row].second;
   for (int col = 0; col < pairs.size(); col++) {
      int idx = hostJdsTColPtrs[col] + row;
      hostNonzeroValues[idx] = pairs[col].second;
      hostColumnIndices[idx] = pairs[col].first;
   }
}

gpuTKTime_stop(GPU, "Converting matrix A to JDS format (transposed).");

gpuTKTime_start(GPU, "Allocating GPU memory.");
//@@ Allocate GPU memory here
cudaMalloc((void **)&deviceNonzeroValues, sizeof(float) * nonzeroCnt);
cudaMalloc((void **)&deviceColumnIndices, sizeof(int) * nonzeroCnt);
cudaMalloc((void **)&deviceJdsRowIndices, sizeof(int) * numARows);
cudaMalloc((void **)&deviceJdsTColPtrs, sizeof(int) * (numMaxCols + 1));
cudaMalloc((void **)&deviceB, sizeof(float) * numBRows * numBColumns);
gpuTKTime_stop(GPU, "Allocating GPU memory.");

gpuTKTime_start(GPU, "Copying input memory to the GPU.");
//@@ Copy memory to the GPU here
cudaMemcpy(deviceNonzeroValues, hostNonzeroValues, sizeof(float) * nonzeroCnt, cudaMemcpyHostToDevice);
cudaMemcpy(deviceColumnIndices, hostColumnIndices, sizeof(int) * nonzeroCnt, cudaMemcpyHostToDevice);
cudaMemcpy(deviceJdsRowIndices, hostJdsRowIndices, sizeof(int) * numARows, cudaMemcpyHostToDevice);
cudaMemcpy(deviceJdsTColPtrs, hostJdsTColPtrs, sizeof(int) * (numMaxCols + 1), cudaMemcpyHostToDevice);
cudaMemcpy(deviceB, hostB, sizeof(float) * numBRows * numBColumns, cudaMemcpyHostToDevice);
gpuTKTime_stop(GPU, "Copying input memory to the GPU.");

//@@ Initialize the grid and block dimensions here
int dimGrid = (numMaxCols + (BLOCK_SIZE - 1)) / BLOCK_SIZE;
int blockSize = BLOCK_SIZE;
gpuTKTime_start(Compute, "Performing CUDA computation");
//@@ Launch the GPU Kernel here
FUNC<<<dimGrid, blockSize>>>(numARows, numMaxCols, deviceNonzeroValues, deviceColumnIndices, deviceJdsTColPtrs,
deviceJdsRowIndices, deviceB, deviceC);

cudaDeviceSynchronize();
gpuTKTime_stop(Compute, "Performing CUDA computation");

gpuTKTime_start(Copy, "Copying output memory to the CPU");
//@@ Copy the GPU memory back to the CPU here
cudaMemcpy(hostC, deviceC, sizeof(float) * numCRows * numCColumns, cudaMemcpyDeviceToHost);

gpuTKTime_stop(Copy, "Copying output memory to the CPU");
```

```
    gpuTKTime_start(GPU, "Freeing GPU Memory");
    //@@ Free the GPU memory here
    cudaFree(deviceNonzeroValues);
    cudaFree(deviceColumnIndices);
    cudaFree(deviceJdsRowIndices);
    cudaFree(deviceJdsTColPtrs);
    cudaFree(deviceB);
    cudaFree(deviceC);

    gpuTKTime_stop(GPU, "Freeing GPU Memory");

    gpuTKSolution(args, hostC, numCRows, numCColumns);

    free(hostA);
    free(hostB);
    free(hostC);

    free(hostNonzeroValues);
    free(hostColumnIndices);
    free(hostJdsRowIndices);
    free(hostJdsTColPtrs);

    return 0;
}
```

#2. Evaluation is performed in NVIDIA RTX A5000, with CUDA 12.0 environment. Block_size = **1024**. To be honest, I've failed failed to achieve performance improvement through shared memory use

R	16	64	64	112	168	512	1024	1024	4096
C	16	64	128	48	168	510	1024	2048	8000
Conversion	0.041385	0.333612	0.577128	0.454435	1.97225	17.5761	72.1025	151.756	2748.66
Alloc	0.019611	0.020652	0.020587	0.021224	0.021558	0.022759	0.259646	0.348774	0.664947
H->D	0.052082	0.056112	0.066641	0.05987	0.073534	0.29539	0.63069	1.25324	21.8483
Kernel	0.028997	0.028997	0.028997	0.028997	0.028997	0.028997	0.028997	0.028997	0.028997
Kernel(SMEM)	0.024929	0.024929	0.024929	0.024929	0.024929	0.024929	0.024929	0.024929	0.024929
D->H	0.014276	0.01579	0.01445	0.014302	0.013585	0.014051	0.01636	0.016108	0.024627
Free	0.135172	0.134489	0.137803	0.1347	0.144839	0.152502	0.463466	0.693337	3.54433