❮ (/mp/124?show=code) Sparse Matrix Multiplication (JDS) Attempt

Attempt Summary

Submit Attempt for Grading

Remember to answer the questions before clicking.

Dataset Id:	6
Created:	about a minute ago (2022-04-24 19:03:15 +0000 UTC)
Status:	Correct solution for this dataset.

Timer Output

Kind	Location	Time (ms)	Message
Generic	main.cu::65	489.387291	Importing data and creating memory on host
GPU	main.cu::78	1.178791	Allocating GPU memory.
GPU	main.cu::89	0.25224	Copying input memory to the GPU.
Compute	main.cu::98	0.170311	Performing CUDA computation
Copy	main.cu::103	0.031733	Copying output memory to the CPU
GPU	main.cu::107	0.23004	Freeing GPU Memory

Program Code

```
#include <wb.h>

#define wbCheck(stmt)
  do {
    cudaError_t err = stmt;
    if (err != cudaSuccess) {
      wbLog(ERROR, "Failed to run stmt ", #stmt);
      wbLog(ERROR, "Got CUDA error ...  ", cudaGetErrorString(err));
      return -1;
    }
  } while (0)

__global__ void spmvJDSKernel(float *out, int *matColStart, int *matCols,
  //@@ insert spmv kernel for jds format

  //1D array method
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  if (row < dim){
    float prod = 0;
    for (int i = 0; i < matRows[row]; i++){
      prod += matData[matColStart[i] + row] * vec[matCols[matColStart[i]
    }

    //output
    out[matRowPerm[row]] = prod;
  }
}

static void spmvJDS(float *out, int *matColStart, int *matCols, int *matR

  //@@ invoke spmv kernel for jds format

  //dimensions
  dim3 grid(ceil((float)dim/512.0), 1, 1);
  dim3 block(512, 1, 1);

  //call kernel
  spmvJDSKernel<<<grid, block>>>(out, matColStart, matCols, matRowPerm, m
}

int main(int argc, char **argv) {
  wbArg_t args;
  int *hostCSRCols;
  int *hostCSRRows;
  float *hostCSRData;
  int *hostJDSColStart;
  int *hostJDSCols;
  int *hostJDSRowPerm;
  int *hostJDSRows;
  float *hostJDSData;
  float *hostVector;
  float *hostOutput;
  int *deviceJDSColStart;
  int *deviceJDSCols;
  int *deviceJDSRowPerm;
  int *deviceJDSRows;
  float *deviceJDSData;
  float *deviceVector;
  float *deviceOutput;
  int dim, ncols, nrows, ndata;
  int maxRowNNZ;

  args = wbArg_read(argc, argv);

  wbTime_start(Generic, "Importing data and creating memory on host");
  hostCSRCols = (int *)wbImport(wbArg_getInputFile(args, 0), &ncols, "Int
  hostCSRRows = (int *)wbImport(wbArg_getInputFile(args, 1), &nrows, "Int
  hostCSRData = (float *)wbImport(wbArg_getInputFile(args, 2), &ndata, "R
  hostVector = (float *)wbImport(wbArg_getInputFile(args, 3), &dim, "Real

  hostOutput = (float *)malloc(sizeof(float) * dim);

  wbTime_stop(Generic, "Importing data and creating memory on host");

  CSRToJDS(dim, hostCSRRows, hostCSRCols, hostCSRData, &hostJDSRowPerm, &
  maxRowNNZ = hostJDSRows[0];

  wbTime_start(GPU, "Allocating GPU memory.");
  cudaMalloc((void **)&deviceJDSColStart, sizeof(int) * maxRowNNZ);
  cudaMalloc((void **)&deviceJDSCols, sizeof(int) * ndata);
  cudaMalloc((void **)&deviceJDSRowPerm, sizeof(int) * dim);
  cudaMalloc((void **)&deviceJDSRows, sizeof(int) * dim);
  cudaMalloc((void **)&deviceJDSData, sizeof(float) * ndata);

  cudaMalloc((void **)&deviceVector, sizeof(float) * dim);
  cudaMalloc((void **)&deviceOutput, sizeof(float) * dim);
  wbTime_stop(GPU, "Allocating GPU memory.");

  wbTime_start(GPU, "Copying input memory to the GPU.");
  cudaMemcpy(deviceJDSColStart, hostJDSColStart, sizeof(int) * maxRowNNZ,
  cudaMemcpy(deviceJDSCols, hostJDSCols, sizeof(int) * ndata, cudaMemcpyH
  cudaMemcpy(deviceJDSRowPerm, hostJDSRowPerm, sizeof(int) * dim, cudaMem
  cudaMemcpy(deviceJDSRows, hostJDSRows, sizeof(int) * dim, cudaMemcpyHos
  cudaMemcpy(deviceJDSData, hostJDSData, sizeof(float) * ndata, cudaMemcp
  cudaMemcpy(deviceVector, hostVector, sizeof(float) * dim, cudaMemcpyHos
  wbTime_stop(GPU, "Copying input memory to the GPU.");

  wbTime_start(Compute, "Performing CUDA computation");
  spmvJDS(deviceOutput, deviceJDSColStart, deviceJDSCols, deviceJDSRowPer
  cudaDeviceSynchronize();
  wbTime_stop(Compute, "Performing CUDA computation");

  wbTime_start(Copy, "Copying output memory to the CPU");
  cudaMemcpy(hostOutput, deviceOutput, sizeof(float) * dim, cudaMemcpyDev
  wbTime_stop(Copy, "Copying output memory to the CPU");

  wbTime_start(GPU, "Freeing GPU Memory");
  cudaFree(deviceVector);
  cudaFree(deviceOutput);
  cudaFree(deviceJDSColStart);
  cudaFree(deviceJDSCols);
  cudaFree(deviceJDSRowPerm);
  cudaFree(deviceJDSRows);
  cudaFree(deviceJDSData);

  wbTime_stop(GPU, "Freeing GPU Memory");

  wbSolution(args, hostOutput, dim);

  free(hostCSRCols);
  free(hostCSRRows);
  free(hostCSRData);
  free(hostVector);
  free(hostOutput);
  free(hostJDSColStart);
  free(hostJDSCols);
  free(hostJDSRowPerm);
  free(hostJDSRows);
  free(hostJDSData);

  return 0;
```

```
}

```

Designed and architected by Abdul Dakkak (https://www.dakkak.dev/).