[CSED490C] Assignment Report: Lab6_cuda

- Student Id : 20220848
- Name : 선민수

1. Template.cu

1.1. template_wo_staging.cu

```
#include <gputk.h>
#include <algorithm>

#define THREADS_PER_BLOCK 512

using namespace std;

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


//@@ Insert code to implement SPMV using JDS with transposed input here
__global__ void SpMV_JDS_T(int numRows, float *data, int *col_idx, int *col_ptr, int *r
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  if(row < numRows) {
    float value = 0;
    unsigned int idx = 0;
    while(col_ptr[idx + 1] - col_ptr[idx] > row) {
      value += data[col_ptr[idx] + row] * B[col_idx[col_ptr[idx] + row]];
      idx++;
    }
    C[row_idx[row]] = value;
  }
}


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 *deviceB;
float *deviceC;
int numARows;    // number of rows in the matrix A
int numAColumns; // number of columns in the matrix A
int numBRows;    // number of rows in the matrix B
int numBColumns; // number of columns in the matrix B
int numCRows;    // number of rows in the matrix C (you have to set this)
int numCColumns; // number of columns in the matrix C (you have to set
                 // this)

int numTotalElems = 0;
int *cnt_o;
int *cnt_s;

float *jds_data;
int *jds_col_idx;
int *jds_row_ptr;
int *jds_row_idx;

float *jds_t_data;
int *jds_t_col_idx;
int *jds_t_col_ptr;
int *jds_t_row_idx;

float *device_data;
int *device_col_idx;
int *device_col_ptr;
int *device_row_idx;

args = gpuTKArg_read(argc, argv);

gpuTKTime_start(Generic, "Importing data and creating memory on host");
hostA = (float *)gpuTKImport(gpuTKArg_getInputFile(args, 0), &numARows,
                             &numAColumns);
hostB = (float *)gpuTKImport(gpuTKArg_getInputFile(args, 1), &numBRows,
                             &numBColumns);
//@@ Set numCRows and numCColumns
numCRows    = numARows;
numCColumns = numBColumns;
//@@ Allocate the hostC matrix
hostC = (float *)malloc(numCRows * numCColumns * sizeof(float));
gpuTKTime_stop(Generic, "Importing data and creating memory on host");
```

```
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
cnt_o = (int *)malloc(numARows * sizeof(int));
cnt_s = (int *)malloc(numARows * sizeof(int));
for(int i = 0; i < numARows; i++)
  cnt_o[i] = 0;

for(int i = 0; i < numARows; i++) {
  for(int j = 0; j < numAColumns; j++)
    cnt_o[i] += (hostA[i * numAColumns + j] != 0.0);
  numTotalElems += cnt_o[i];
  cnt_s[i] = cnt_o[i];
}

sort(cnt_s, cnt_s + numARows, greater<float>());

jds_data = (float *)malloc(numTotalElems * sizeof(float));
jds_col_idx = (int *)malloc(numTotalElems * sizeof(int));
jds_row_idx = (int *)malloc(numARows * sizeof(int));
jds_row_ptr = (int *)malloc((numARows + 1) * sizeof(int));

jds_row_ptr[0] = 0;
for(int i = 0, idx = 0; i < numARows; i++) {
  for(int j = 0; j < numARows; j++) {
    if(cnt_s[i] == cnt_o[j]) {
      jds_row_idx[i] = j;
      for(int k = 0; k < numAColumns; k++) {
        if(hostA[j * numAColumns + k] != 0.0) {
          jds_data[idx] = hostA[j * numAColumns + k];
          jds_col_idx[idx] = k;
          idx++;
        }
      }
      cnt_o[j] = -1;
      jds_row_ptr[i + 1] = idx;
      break;
    }
  }
}
```

```
jds_t_data = (float *)malloc(numTotalElems * sizeof(float));
jds_t_col_idx = (int *)malloc(numTotalElems * sizeof(int));
jds_t_row_idx = (int *)malloc(numARows * sizeof(int));
jds_t_col_ptr = (int *)malloc((numAColumns + 1) * sizeof(int));

jds_t_col_ptr[0] = 0;
for(int i = 0; i < numARows; i++) jds_t_row_idx[i] = jds_row_idx[i];
for(int i = 0, idx = 0; i < numAColumns; i++) {
  for(int j = 0; j < numARows && jds_row_ptr[j] + i < jds_row_ptr[j + 1]; j++) {
    jds_t_data[idx] = jds_data[jds_row_ptr[j] + i];
    jds_t_col_idx[idx] = jds_col_idx[jds_row_ptr[j] + i];
    idx++;
  }
  jds_t_col_ptr[i + 1] = idx;
}

free(cnt_o);
free(cnt_s);

free(jds_data);
free(jds_col_idx);
free(jds_row_idx);
free(jds_row_ptr);

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


gpuTKTime_start(GPU, "Allocating GPU memory.");
//@@ Allocate GPU memory here
gpuTKCheck(cudaMalloc((void **)&device_data, numTotalElems * sizeof(float)));
gpuTKCheck(cudaMalloc((void **)&device_col_idx, numTotalElems * sizeof(int)));
gpuTKCheck(cudaMalloc((void **)&device_col_ptr, (numAColumns + 1) * sizeof(int)));
gpuTKCheck(cudaMalloc((void **)&device_row_idx, numARows * sizeof(int)));

gpuTKCheck(cudaMalloc((void **)&deviceB, numBRows * numBColumns * sizeof(float)));
gpuTKCheck(cudaMalloc((void **)&deviceC, numCRows * numCColumns * sizeof(float)));

gpuTKTime_stop(GPU, "Allocating GPU memory.");

gpuTKTime_start(GPU, "Copying input memory to the GPU.");
//@@ Copy memory to the GPU here
gpuTKCheck(cudaMemcpy(device_data, jds_t_data, numTotalElems * sizeof(float), cudaMem
```

```
gpuTKCheck(cudaMemcpy(device_col_idx, jds_t_col_idx, numTotalElems * sizeof(int), cud
gpuTKCheck(cudaMemcpy(device_col_ptr, jds_t_col_ptr, (numAColumns + 1) * sizeof(int),
gpuTKCheck(cudaMemcpy(device_row_idx, jds_t_row_idx, numARows * sizeof(int), cudaMemc

gpuTKCheck(cudaMemcpy(deviceB, hostB, numBRows * numBColumns * sizeof(float), cudaMem

gpuTKTime_stop(GPU, "Copying input memory to the GPU.");

//@@ Initialize the grid and block dimensions here
dim3 dimGrid((numARows - 1) / THREADS_PER_BLOCK + 1, 1, 1);
dim3 dimBlock(THREADS_PER_BLOCK, 1, 1);

gpuTKTime_start(Compute, "Performing CUDA computation");
//@@ Launch the GPU Kernel here
SpMV_JDS_T<<<dimGrid, dimBlock>>>(numARows, device_data, device_col_idx, device_col_p

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
gpuTKCheck(cudaMemcpy(hostC, deviceC, numCRows * numCColumns * sizeof(float), cudaMem

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


gpuTKTime_start(GPU, "Freeing GPU Memory");
//@@ Free the GPU memory here
cudaFree(device_data);
cudaFree(device_col_idx);
cudaFree(device_col_ptr);
cudaFree(device_row_idx);

cudaFree(deviceB);
cudaFree(deviceC);

gpuTKTime_stop(GPU, "Freeing GPU Memory");

gpuTKSolution(args, hostC, numCRows, numCColumns);

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

```
  free(jds_t_data);
  free(jds_t_col_idx);
  free(jds_t_col_ptr);
  free(jds_t_row_idx);

  return 0;
}
```

1.2. template_w_staging.cu

```
#include <gputk.h>
#include <algorithm>

#define THREADS_PER_BLOCK 512

using namespace std;

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


//@@ Insert code to implement SPMV using JDS with transposed input here
__global__ void SpMV_JDS_T(int numRows, int numColumns, float *data, int *col_idx, int
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  extern __shared__ float shm[];
  for(int i = 0; i < len; i++)
    if(threadIdx.x * len + i < numColumns)
      shm[threadIdx.x * len + i] = B[threadIdx.x * len + i];
  __syncthreads();
  if(row < numRows) {
    float value = 0;
    unsigned int idx = 0;
    while(col_ptr[idx + 1] - col_ptr[idx] > row) {
      value += data[col_ptr[idx] + row] * shm[col_idx[col_ptr[idx] + row]];
      idx++;
    }
    C[row_idx[row]] = value;
  }
}


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 *deviceB;
float *deviceC;
int numARows;    // number of rows in the matrix A
int numAColumns; // number of columns in the matrix A
int numBRows;    // number of rows in the matrix B
int numBColumns; // number of columns in the matrix B
int numCRows;    // number of rows in the matrix C (you have to set this)
int numCColumns; // number of columns in the matrix C (you have to set
                 // this)

int numTotalElems = 0;
int *cnt_o;
int *cnt_s;

float *jds_data;
int *jds_col_idx;
int *jds_row_ptr;
int *jds_row_idx;

float *jds_t_data;
int *jds_t_col_idx;
int *jds_t_col_ptr;
int *jds_t_row_idx;

float *device_data;
int *device_col_idx;
int *device_col_ptr;
int *device_row_idx;

args = gpuTKArg_read(argc, argv);

gpuTKTime_start(Generic, "Importing data and creating memory on host");
hostA = (float *)gpuTKImport(gpuTKArg_getInputFile(args, 0), &numARows,
                             &numAColumns);
hostB = (float *)gpuTKImport(gpuTKArg_getInputFile(args, 1), &numBRows,
                             &numBColumns);
//@@ Set numCRows and numCColumns
numCRows    = numARows;
numCColumns = numBColumns;
//@@ Allocate the hostC matrix
```

```
hostC = (float *)malloc(numCRows * numCColumns * sizeof(float));
gpuTKTime_stop(Generic, "Importing data and creating memory on host");

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
cnt_o = (int *)malloc(numARows * sizeof(int));
cnt_s = (int *)malloc(numARows * sizeof(int));
for(int i = 0; i < numARows; i++)
  cnt_o[i] = 0;

for(int i = 0; i < numARows; i++) {
  for(int j = 0; j < numAColumns; j++)
    cnt_o[i] += (hostA[i * numAColumns + j] != 0.0);
  numTotalElems += cnt_o[i];
  cnt_s[i] = cnt_o[i];
}

sort(cnt_s, cnt_s + numARows, greater<float>());

jds_data = (float *)malloc(numTotalElems * sizeof(float));
jds_col_idx = (int *)malloc(numTotalElems * sizeof(int));
jds_row_idx = (int *)malloc(numARows * sizeof(int));
jds_row_ptr = (int *)malloc((numARows + 1) * sizeof(int));

jds_row_ptr[0] = 0;
for(int i = 0, idx = 0; i < numARows; i++) {
  for(int j = 0; j < numARows; j++) {
    if(cnt_s[i] == cnt_o[j]) {
      jds_row_idx[i] = j;
      for(int k = 0; k < numAColumns; k++) {
        if(hostA[j * numAColumns + k] != 0.0) {
          jds_data[idx] = hostA[j * numAColumns + k];
          jds_col_idx[idx] = k;
          idx++;
        }
      }
      cnt_o[j] = -1;
      jds_row_ptr[i + 1] = idx;
      break;
```

```
    }
  }
}

jds_t_data = (float *)malloc(numTotalElems * sizeof(float));
jds_t_col_idx = (int *)malloc(numTotalElems * sizeof(int));
jds_t_row_idx = (int *)malloc(numARows * sizeof(int));
jds_t_col_ptr = (int *)malloc((numAColumns + 1) * sizeof(int));

jds_t_col_ptr[0] = 0;
for(int i = 0; i < numARows; i++) jds_t_row_idx[i] = jds_row_idx[i];
for(int i = 0, idx = 0; i < numAColumns; i++) {
  for(int j = 0; j < numARows && jds_row_ptr[j] + i < jds_row_ptr[j + 1]; j++) {
    jds_t_data[idx] = jds_data[jds_row_ptr[j] + i];
    jds_t_col_idx[idx] = jds_col_idx[jds_row_ptr[j] + i];
    idx++;
  }
  jds_t_col_ptr[i + 1] = idx;
}

free(cnt_o);
free(cnt_s);

free(jds_data);
free(jds_col_idx);
free(jds_row_idx);
free(jds_row_ptr);

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


gpuTKTime_start(GPU, "Allocating GPU memory.");
//@@ Allocate GPU memory here
gpuTKCheck(cudaMalloc((void **)&device_data, numTotalElems * sizeof(float)));
gpuTKCheck(cudaMalloc((void **)&device_col_idx, numTotalElems * sizeof(int)));
gpuTKCheck(cudaMalloc((void **)&device_col_ptr, (numAColumns + 1) * sizeof(int)));
gpuTKCheck(cudaMalloc((void **)&device_row_idx, numARows * sizeof(int)));

gpuTKCheck(cudaMalloc((void **)&deviceB, numBRows * numBColumns * sizeof(float)));
gpuTKCheck(cudaMalloc((void **)&deviceC, numCRows * numCColumns * sizeof(float)));

gpuTKTime_stop(GPU, "Allocating GPU memory.");
```

```
gpuTKTime_start(GPU, "Copying input memory to the GPU.");
//@@ Copy memory to the GPU here
gpuTKCheck(cudaMemcpy(device_data, jds_t_data, numTotalElems * sizeof(float), cudaMem
gpuTKCheck(cudaMemcpy(device_col_idx, jds_t_col_idx, numTotalElems * sizeof(int), cud
gpuTKCheck(cudaMemcpy(device_col_ptr, jds_t_col_ptr, (numAColumns + 1) * sizeof(int),
gpuTKCheck(cudaMemcpy(device_row_idx, jds_t_row_idx, numARows * sizeof(int), cudaMemc

gpuTKCheck(cudaMemcpy(deviceB, hostB, numBRows * numBColumns * sizeof(float), cudaMem

gpuTKTime_stop(GPU, "Copying input memory to the GPU.");

//@@ Initialize the grid and block dimensions here
dim3 dimGrid((numARows - 1) / THREADS_PER_BLOCK + 1, 1, 1);
dim3 dimBlock(THREADS_PER_BLOCK, 1, 1);

gpuTKTime_start(Compute, "Performing CUDA computation");
//@@ Launch the GPU Kernel here
SpMV_JDS_T<<<dimGrid, dimBlock, numBRows * numBColumns * sizeof(float)>>>(numARows, n

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
gpuTKCheck(cudaMemcpy(hostC, deviceC, numCRows * numCColumns * sizeof(float), cudaMem

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


gpuTKTime_start(GPU, "Freeing GPU Memory");
//@@ Free the GPU memory here
cudaFree(device_data);
cudaFree(device_col_idx);
cudaFree(device_col_ptr);
cudaFree(device_row_idx);

cudaFree(deviceB);
cudaFree(deviceC);

gpuTKTime_stop(GPU, "Freeing GPU Memory");

gpuTKSolution(args, hostC, numCRows, numCColumns);
```

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

  free(jds_t_data);
  free(jds_t_col_idx);
  free(jds_t_col_ptr);
  free(jds_t_row_idx);

  return 0;
}
```

3. Execution times

Execution Systems

All compilation and the executions are made on docker container.

NVIDIA GeForce GTX 1650

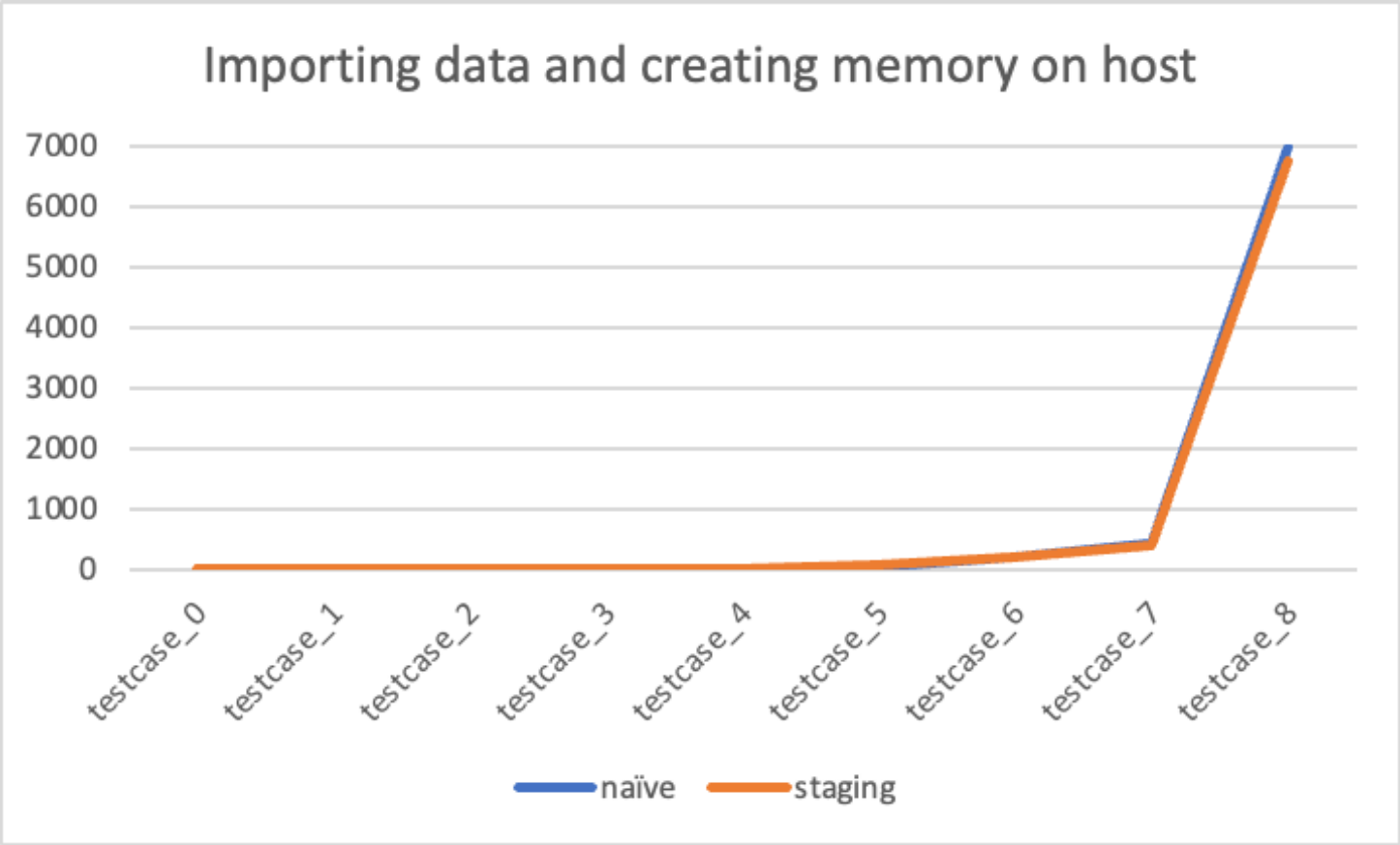
- Windows 10 + WSL2(Ubuntu 22.04)
- Docker Image : `nvidia:cuda/12.0.1-devel-ubuntu22.04`
- Driver Version : `546.17`
- Cuda Version : `12.3`
- (cuda version shown in the container does not match with the docker image, but could not figure out why versions do not match)

Execution Script

```
base="/workspace/Lab7_cuda"
cd $base/sources
make template
echo > $base/result
for idx in {0..8}
do
    echo "Testcase $idx"
    cd $base/sources/SparseMV/Dataset/$idx
    ./../../../JDS_T_template -e output.raw -i input0.raw,input1.raw -o o.raw -t vector
    echo >> $base/result
done
```

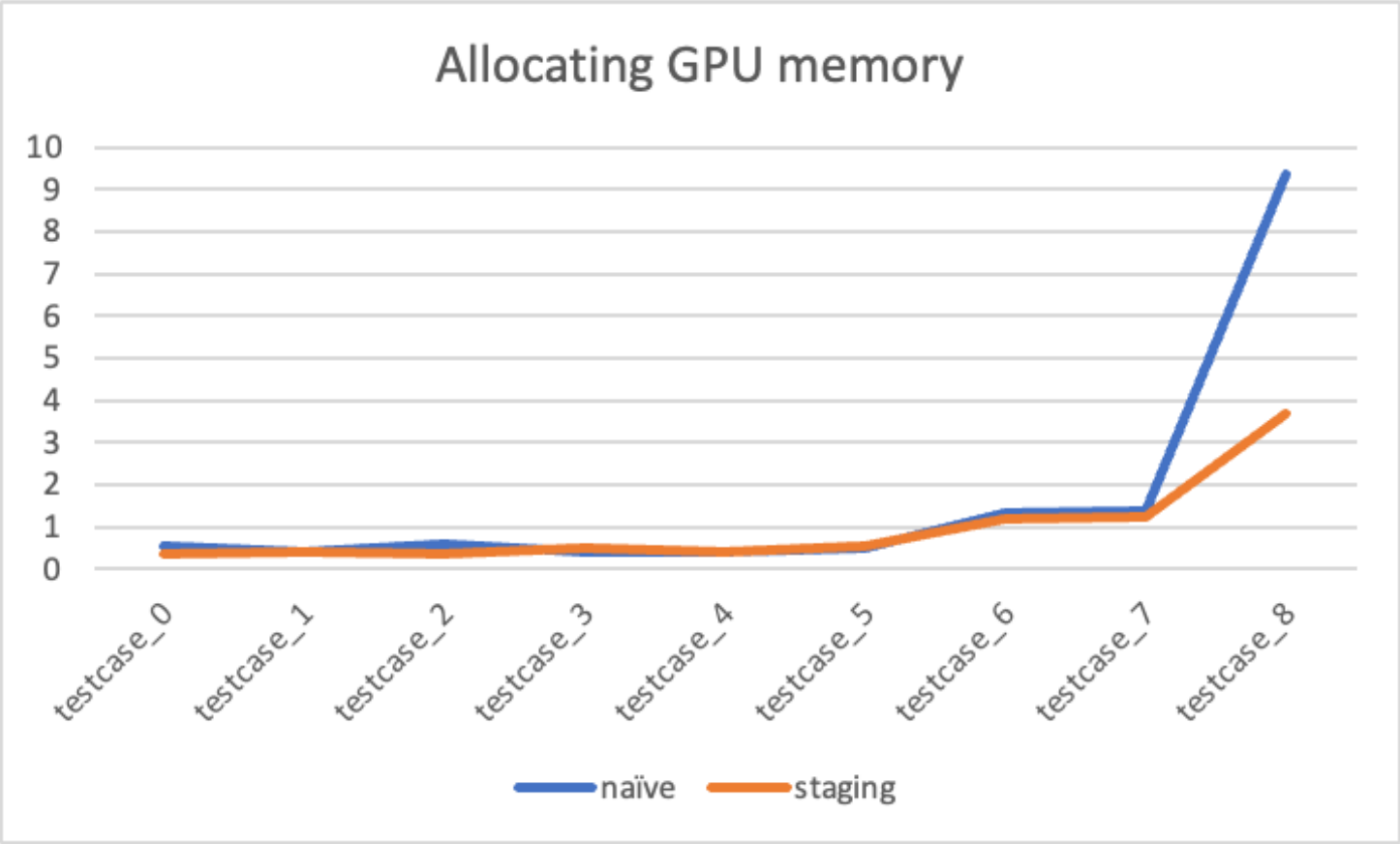
1 [Importing data and creating memory on host]

	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
naïve	0.2937	1.1372	2.2369	1.4478	9.4045	50.3491	211.146	429.667	6970.07
staging	3.3335	2.4996	3.6502	3.4242	7.9779	55.3007	203.396	408.102	6748.07



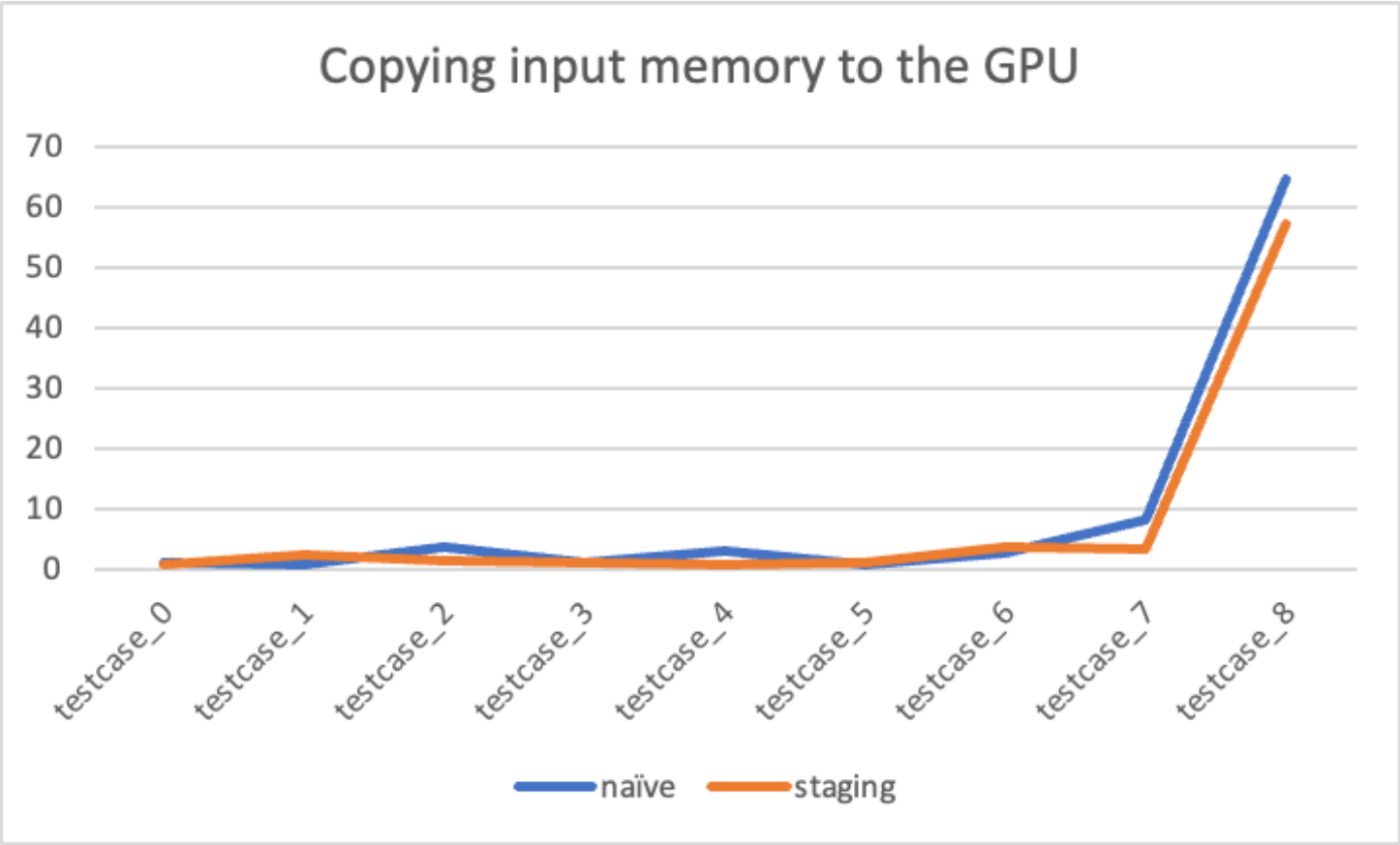
2 [Allocating GPU memory]

	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
naïve	0.564001	0.4269	0.6005	0.4271	0.3835	0.4915	1.3217	1.3948	9.3536
staging	0.3805	0.4123	0.3712	0.4857	0.4084	0.5275	1.1808	1.2232	3.6611



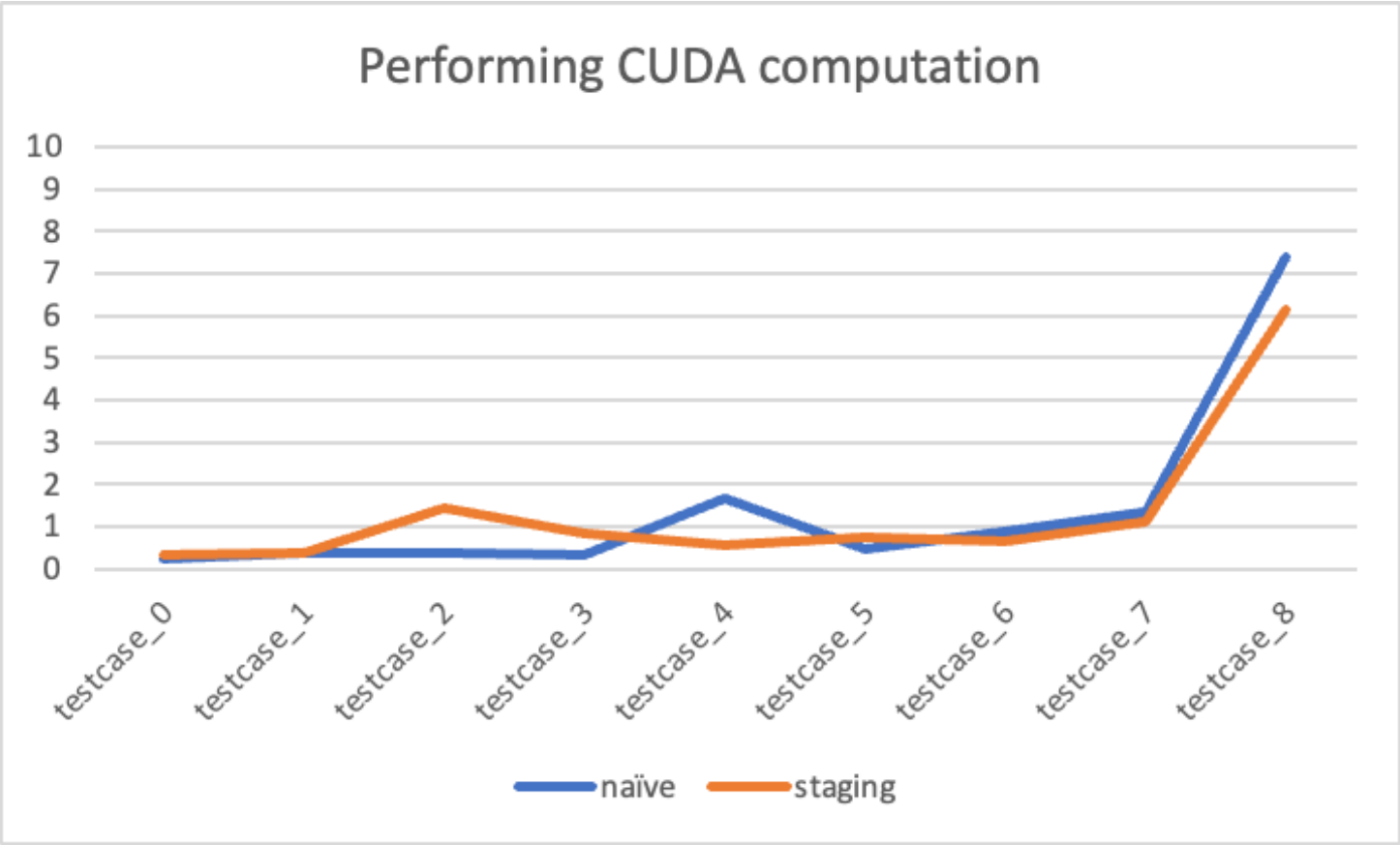
3 [Copying input memory to the GPU]

	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
naïve	0.9089	0.5442	3.46	0.9726	2.8365	0.762601	2.4838	7.9818	64.7071
staging	0.588	2.458	1.442	1.0853	0.6557	0.8656	3.4666	3.4105	57.3245



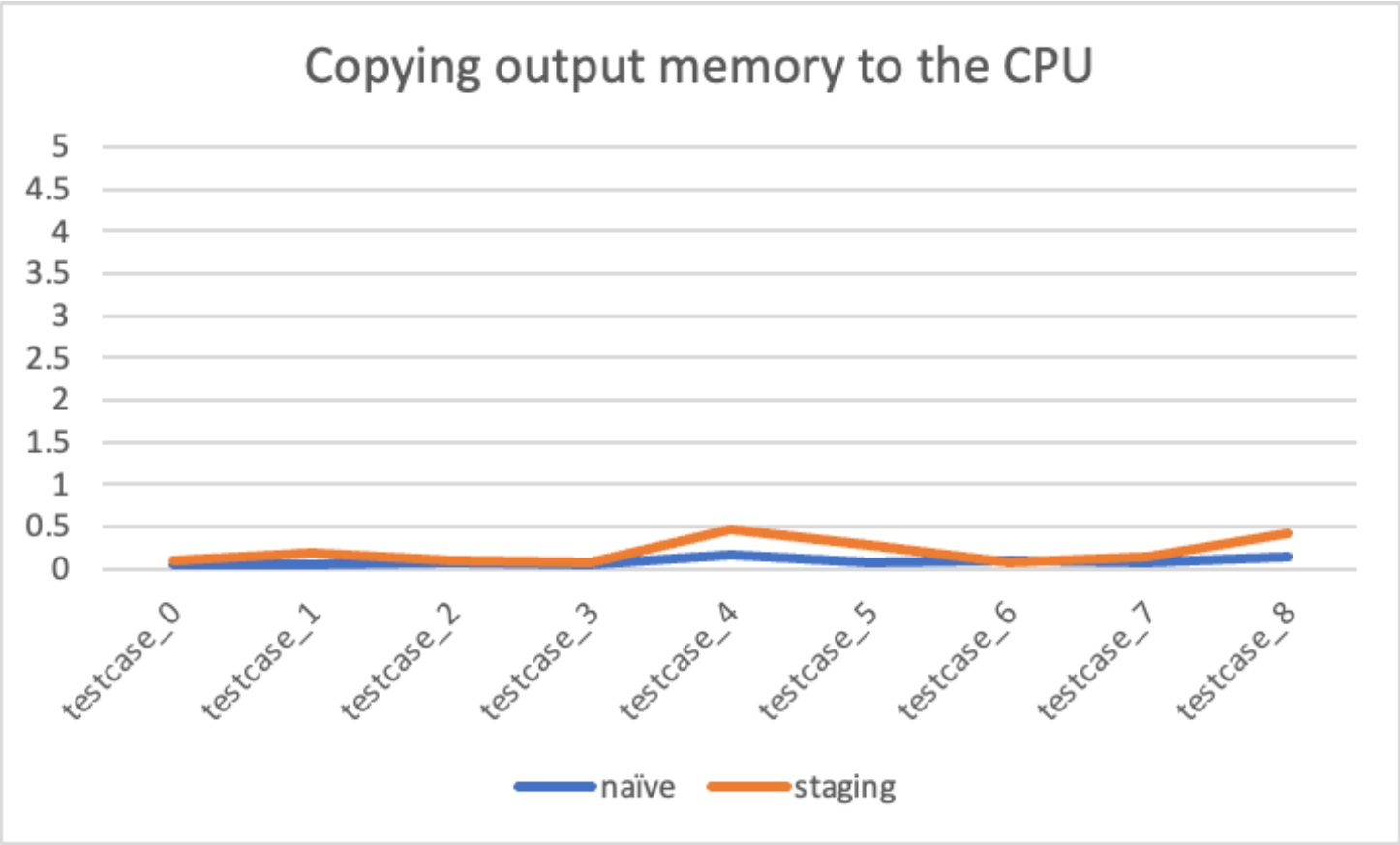
4 [Performing CUDA computation]

	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
naïve	0.2227	0.3794	0.3541	0.3304	1.663	0.4574	0.8977	1.3436	7.3752
staging	0.3321	0.393	1.4337	0.8368	0.5573	0.7626	0.6584	1.1234	6.1316



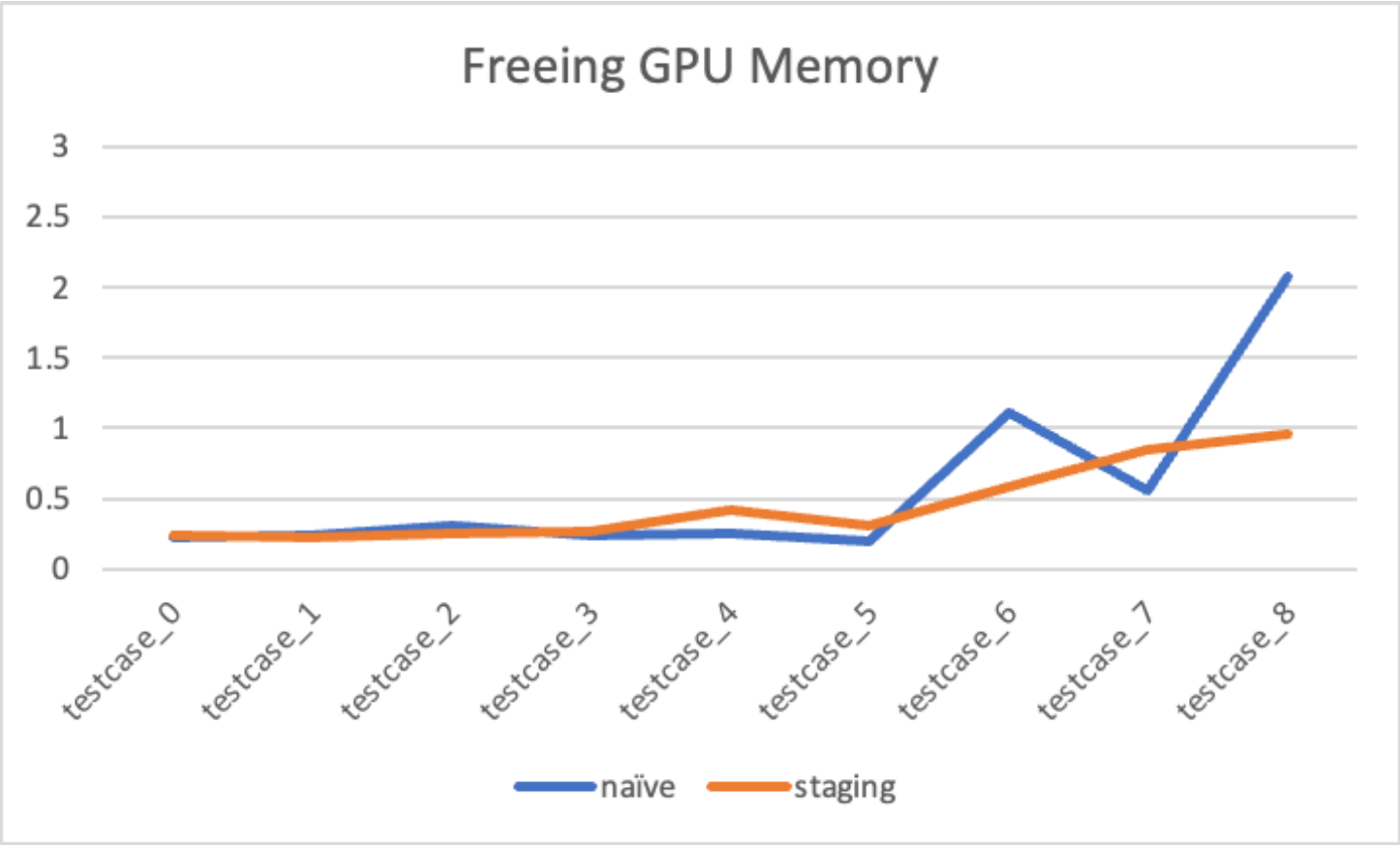
5 [Copying output memory to the CPU]

	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
naïve	0.0481	0.0556	0.0661	0.0482	0.1732	0.0829	0.0854	0.0754	0.1468
staging	0.0851	0.1969	0.0881	0.0694	0.461	0.2904	0.0659	0.1402	0.4125



6 [Freeing GPU Memory]

	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
naïve	0.2239	0.2384	0.3014	0.2386	0.245	0.1902	1.1067	0.5626	2.0798
staging	0.2317	0.2188	0.2546	0.2706	0.416	0.3105	0.5817	0.842	0.9528


Freeing GPU Memory
3
2.5
2
1.5
1
0.5
0
testcase_0
testcase_1
testcase_2
testcase_3
testcase_4
testcase_5
testcase_6
testcase_7
testcase_8
naïve
staging