CSED490C-01 Heterogeneous Parallel Computing

Lab Assignment 3

20232241 Dookyung Kang

1. In my version of code, the size of moving data is exactly same as the size of vector * 2. Only remainder of vectors are transferred for the last stream, not aligned to stream size: **2 * inputLength**

2. Pinned memory is used when cudaMemcpyAsync called, in order to transfer memory between device and host concurrently, especially using stream.

3. This is entire code.

```
#include <gputk.h>


__global__ void vecAdd(float *in1, float *in2, float *out, int len) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if (index < len) {
    out[index] = in1[index] + in2[index];
  }
}

#ifndef STREAM
#define stream 4
#endif

int main(int argc, char **argv) {
  gpuTKArg_t args;
  int inputLength;
  float *hostInput1;
  float *hostInput2;
  float *hostOutput;
  float *deviceInput1;
  float *deviceInput2;
  float *deviceOutput;
  unsigned int numStreams;

  args = gpuTKArg_read(argc, argv);

  gpuTKTime_start(Generic, "Importing data and creating memory on host");
  hostInput1 =
      (float *)gpuTKImport(gpuTKArg_getInputFile(args, 0), &inputLength);
  hostInput2 =
      (float *)gpuTKImport(gpuTKArg_getInputFile(args, 1), &inputLength);
  hostOutput = (float *)malloc(inputLength * sizeof(float));
  gpuTKTime_stop(Generic, "Importing data and creating memory on host");

  gpuTKLog(TRACE, "The input length is ", inputLength);

  gpuTKTime_start(GPU, "Allocating Pinned memory.");

  //@@ Allocate GPU memory here using pinned memory here
  cudaMallocHost((void **)&deviceInput1, inputLength * sizeof(float));
  cudaMallocHost((void **)&deviceInput2, inputLength * sizeof(float));
  cudaMallocHost((void **)&deviceOutput, inputLength * sizeof(float));

  //@@ Create and setup streams
  numStreams = STREAM;
  cudaStream_t streams[numStreams];
  for (int i = 0; i < numStreams; i++) {
    cudaStreamCreate(&streams[i]);
  }

  //@@ Calculate data segment size of input data processed by each stream
  int streamSizes[numStreams];
  int offsets[numStreams];
  int streamSizeBase = inputLength / numStreams;
  for (int i = 0; i < numStreams; i++) {
    if (i < numStreams - 1)
      streamSizes[i] = streamSizeBase;
    else
      streamSizes[i] = inputLength - streamSizeBase * (numStreams - 1);
    offsets[i] = i * streamSizeBase;
  }

  int blockSize = 256;
  int numBlocks = (inputLength + blockSize - 1) / blockSize;


  gpuTKTime_start(Compute, "Performing CUDA computation");
  //@@ Perform parallel vector addition with different streams.
  for (unsigned int s = 0; s<numStreams; s++){
    //@@ Asynchronous copy data to the device memory in segments
    //@@ Calculate starting and ending indices for per-stream data
    int offset = offsets[s];
    int streamSize = streamSizes[s];
    cudaStream_t stream = streams[s];

    cudaMemcpyAsync(&deviceInput1[offset], &hostInput1[offset],
      streamSize * sizeof(float), cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync(&deviceInput2[offset], &hostInput2[offset],
      streamSize * sizeof(float), cudaMemcpyHostToDevice, stream);

    //@@ Invoke CUDA Kernel
    //@@ Determine grid and thread block sizes (consider ococupancy)
    vecAdd<<<numBlocks, blockSize, 0, stream>>>
      (&deviceInput1[offset], &deviceInput2[offset], &deviceOutput[offset], streamSize);

    //@@ Asynchronous copy data from the device memory in segments
    cudaMemcpyAsync(&hostOutput[offset], &deviceOutput[offset],
      streamSize * sizeof(float), cudaMemcpyDeviceToHost, stream);

  }

  //@@ Synchronize
  for (int i = 0; i < numStreams; i++) {
    cudaStreamSynchronize(streams[i]);
  }

  gpuTKTime_stop(Compute, "Performing CUDA computation");


  gpuTKTime_start(GPU, "Freeing Pinned Memory");
  //@@ Destory cudaStream
  for (int i = 0; i < numStreams; i++) {
    cudaStreamDestroy(streams[i]);
  }

  //@@ Free the GPU memory here
  cudaFreeHost(deviceInput1);
  cudaFreeHost(deviceInput2);
  cudaFreeHost(deviceOutput);

  gpuTKTime_stop(GPU, "Freeing Pinned Memory");

  gpuTKSolution(args, hostOutput, inputLength);

  free(hostInput1);
  free(hostInput2);
  free(hostOutput);

  return 0;
}
```

4. Evaluation is performed in NVIDIA RTX A5000, with CUDA 12.0 environment.

N	16	64	93	112	1120	9921	14000	25365	48000	96000
Allocating (ms)	0.20638	0.438018	0.594588	0.687433	5.68884	49.6007	69.8475	126.265	239.515	477.572
Computation (ms)	0.106404	0.108997	0.108298	0.104469	0.108586	0.125712	0.138378	0.16151	0.187456	0.264895
Freeing (ms)	0.348197	0.34155	0.341255	0..343108	0.340594	0.350368	0.34517	0.342886	0.357332	0.36363

5. Evaluation is performed in NVIDIA RTX A5000, with CUDA 12.0 environment.

N	2	4	8	12	16	24	32
Allocating (ms)	238.669	239.327	238.956	239.956	238.905	239.025	238.824
Computation (ms)	0.144958	0.181115	0.244066	0.320699	0.36287	0.467215	0.562694
Freeing (ms)	0.364924	0.353059	0.355825	0.367033	0.366532	0.38028	0.391648