

```

#ifndef MATRIXTRANSPOSE_H_
#define MATRIXTRANSPOSE_H_

#include <CL/cl.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <SDKUtil/SDKCommon.hpp>
#include <SDKUtil/SDKApplication.hpp>
#include <SDKUtil/SDKCommandArgs.hpp>
#include <SDKUtil/SDKFile.hpp>

/**
 * MatrixTranspose
 * Class implements OpenCL Matrix Transpose sample
 * Derived from SDKSample base class
 */

class MatrixTranspose : public SDKSample
{
    cl_uint      seed;      /**< Seed value for random number generation */
    cl_double    setupTime; /**< Time for setting up OpenCL */
    cl_double    totalKernelTime; /**< Time for kernel execution */
    cl_double    totalProgramTime; /**< Time for program execution */
    cl_double    referenceKernelTime; /**< Time for reference implementation */
    cl_int       width;    /**< width of the input matrix */
    cl_int       height;   /**< height of the input matrix */
    cl_float     *input;   /**< Input array */
    cl_float     *output;  /**< Output Array */
    cl_float     *verificationOutput; /**< Output array for reference implementation */
    cl_uint      blockSize; /**< use local memory of size blockSize x blockSize */
    cl_context    context; /**< CL context */
    cl_device_id *devices; /**< CL device list */
    cl_mem        inputBuffer; /**< CL memory buffer */
    cl_mem        outputBuffer; /**< CL memory output Buffer */
    cl_command_queue commandQueue; /**< CL command queue */
    cl_program    program; /**< CL program */
    cl_kernel     kernel;  /**< CL kernel */

    size_t      maxWorkGroupSize; /**< Device Specific Information */
    cl_uint     maxDimensions;
    size_t *    maxWorkItemSizes;
    cl_ulong    totalLocalMemory;
    cl_ulong    usedLocalMemory;
    cl_ulong    availableLocalMemory;
    cl_ulong    neededLocalMemory;

public:
    /**
     * Constructor
     * Initialize member variables
     * @param name name of sample (string)
     */
    MatrixTranspose(std::string name)
        : SDKSample(name) {
        seed = 123;
        input = NULL;
        output = NULL;
        verificationOutput = NULL;
        blockSize = 8;
        width = 64;
    }
};

```

```

        height = 64;
        setupTime = 0;
        totalKernelTime = 0;
    }

/**
 * Constructor
 * Initialize member variables
 * @param name name of sample (const char*)
 */
MatrixTranspose(const char* name)
: SDKSample(name) {
    seed = 123;
    input = NULL;
    output = NULL;
    verificationOutput = NULL;
    blockSize = 8;
    width = 64;
    height = 64;
    setupTime = 0;
    totalKernelTime = 0;
}

/**
 * Allocate and initialize host memory array with random values
 * @return 1 on success and 0 on failure
 */
int setupMatrixTranspose();

/**
 * OpenCL related initialisations.
 * Set up Context, Device list, Command Queue, Memory buffers
 * Build CL kernel program executable
 * @return 1 on success and 0 on failure
 */
int setupCL();

/**
 * Set values for kernels' arguments, enqueue calls to the kernels
 * on to the command queue, wait till end of kernel execution.
 * Get kernel start and end time if timing is enabled
 * @return 1 on success and 0 on failure
 */
int runCLKernels();

/**
 * Reference CPU implementation of matrix transpose
 * @param output stores the transpose of the input
 * @param input input matrix
 * @param width width of the input matrix
 * @param height height of the array
 */
void matrixTransposeCPUReference(
    cl_float * output,
    cl_float * input,
    const cl_uint width,
    const cl_uint height);

/**
 * Override from SDKSample. Print sample stats.
 */
void printStats();

```

```

/**
 * Override from SDKSample. Initialize
 * command line parser, add custom options
 */
int initialize();

/**
 * Override from SDKSample, adjust width and height
 * of execution domain, perform all sample setup
 */
int setup();

/**
 * Override from SDKSample
 * Run OpenCL matrix transpose
 */
int run();

/**
 * Override from SDKSample
 * Cleanup memory allocations
 */
int cleanup();

/**
 * Override from SDKSample
 * Verify against reference implementation
 */
int verifyResults();
};

#endif
#include "MatrixTranspose.hpp"

int MatrixTranspose::setupMatrixTranspose()
{
    cl_uint inputSizeBytes;

    /* allocate and init memory used by host */
    inputSizeBytes = width * height * sizeof(cl_float);
    input = (cl_float *) malloc(inputSizeBytes);
    if(input==NULL)
    {
        sampleCommon->error("Failed to allocate host memory. (input)");
        return SDK_FAILURE;
    }

    /* random initialisation of input */
    sampleCommon->fillRandom<cl_float>(input, width, height, 0, 255);

    output = (cl_float *) malloc(inputSizeBytes);
    if(output==NULL)
    {
        sampleCommon->error("Failed to allocate host memory. (output)");
        return SDK_FAILURE;
    }

    if(verify)
    {
        verificationOutput = (cl_float *) malloc(inputSizeBytes);
        if(verificationOutput==NULL) {

```

```

        sampleCommon->error("Failed to allocate host memory. (verificationOutput)");
        return SDK_FAILURE;
    }
}

    if(!quiet)
    {
        sampleCommon->printArray<cl_float>(
            "Input",
            input,
            width,
            1);
    }

return SDK_SUCCESS;
}

int MatrixTranspose::setupCL(void)
{
    cl_int status = 0;
    size_t deviceListSize;

    cl_device_type dType;

    if(deviceType.compare("cpu") == 0)
    {
        dType = CL_DEVICE_TYPE_CPU;
    }
    else //deviceType = "gpu"
    {
        dType = CL_DEVICE_TYPE_GPU;
    }

    context = clCreateContextFromType(
        0,
        dType,
        NULL,
        NULL,
        &status);

    if(status != CL_SUCCESS && dType == CL_DEVICE_TYPE_GPU && !sampleArgs->isArgSet("device"))
    {
        std::cout << "Unsupported GPU device; falling back to CPU ..." << std::endl;
        context = clCreateContextFromType(
            0,
            CL_DEVICE_TYPE_CPU,
            NULL,
            NULL,
            &status);
    }

    if(!sampleCommon->checkVal(status,
        CL_SUCCESS,
        "clCreateContextFromType failed.))
        return SDK_FAILURE;

    /* First, get the size of device list data */
    status = clGetContextInfo(
        context,
        CL_CONTEXT_DEVICES,
        0,
        NULL,
        &deviceListSize);
}

```

```

if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clGetContextInfo failed.))
return SDK_FAILURE;

/* Now allocate memory for device list based on the size we got earlier */
devices = (cl_device_id *)malloc(deviceListSize);
if(devices==NULL) {
    sampleCommon->error("Failed to allocate memory (devices).");
    return SDK_FAILURE;
}

/* Now, get the device list data */
status = clGetContextInfo(
    context,
    CL_CONTEXT_DEVICES,
    deviceListSize,
    devices,
    NULL);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clGetContextInfo failed.))
return SDK_FAILURE;

/* Get Device specific Information */
status = clGetDeviceInfo(
    devices[0],
    CL_DEVICE_MAX_WORK_GROUP_SIZE,
    sizeof(size_t),
    (void *)&maxWorkGroupSize,
    NULL);

if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clGetDeviceInfo CL_DEVICE_MAX_WORK_GROUP_SIZE failed.))
return SDK_FAILURE;

status = clGetDeviceInfo(
    devices[0],
    CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
    sizeof(cl_uint),
    (void *)&maxDimensions,
    NULL);

if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clGetDeviceInfo CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS failed.))
return SDK_FAILURE;

maxWorkItemSizes = (size_t *)malloc(maxDimensions*sizeof(size_t));

status = clGetDeviceInfo(
    devices[0],
    CL_DEVICE_MAX_WORK_ITEM_SIZES,
    sizeof(size_t)*maxDimensions,
    (void *)maxWorkItemSizes,
    NULL);

```

```
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clGetDeviceInfo CL_DEVICE_MAX_WORK_ITEM_SIZES failed.))
return SDK_FAILURE;
```

```
status = clGetDeviceInfo(
    devices[0],
    CL_DEVICE_LOCAL_MEM_SIZE,
    sizeof(cl_ulong),
    (void *)&totalLocalMemory,
    NULL);
```

```
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clGetDeviceInfo CL_DEVICE_LOCAL_MEM_SIZES failed.))
return SDK_FAILURE;
```

```
{
    /* The block is to move the declaration of prop closer to its use */
    cl_command_queue_properties prop = 0;
    if(timing)
        prop |= CL_QUEUE_PROFILING_ENABLE;

    commandQueue = clCreateCommandQueue(
        context,
        devices[0],
        prop,
        &status);
    if(!sampleCommon->checkVal(
        status,
        0,
        "clCreateCommandQueue failed.))
        return SDK_FAILURE;
}
```

```
inputBuffer = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
    sizeof(cl_float) * width * height,
    input,
    &status);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clCreateBuffer failed. (inputBuffer)"))
return SDK_FAILURE;
```

```
outputBuffer = clCreateBuffer(
    context,
    CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
    sizeof(cl_float) * width * height,
    output,
    &status);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clCreateBuffer failed. (outputBuffer)"))
return SDK_FAILURE;
```

```

/* create a CL program using the kernel source */
streamsdk::SDKFile kernelFile;
kernelFile.open("MatrixTranspose_Kernels.cl");
const char * source = kernelFile.source().c_str();
size_t sourceSize[] = { strlen(source) };
program = clCreateProgramWithSource(
    context,
    1,
    &source,
    sourceSize,
    &status);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clCreateProgramWithSource failed."))
    return SDK_FAILURE;

/* create a cl program executable for all the devices specified */
status = clBuildProgram(program, 1, devices, NULL, NULL, NULL);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clBuildProgram failed."))
    return SDK_FAILURE;

/* get a kernel object handle for a kernel with the given name */
kernel = clCreateKernel(program, "matrixTranspose", &status);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clCreateKernel failed."))
    return SDK_FAILURE;

return SDK_SUCCESS;
}

int MatrixTranspose::runCLKernels(void)
{
    cl_int status;
    cl_event events[2];

    size_t globalThreads[2]= {width, height};
    size_t localThreads[2] = {blockSize, blockSize};

    long long kernelsStartTime;
    long long kernelsEndTime;

    status = clGetKernelWorkGroupInfo(
        kernel,
        devices[0],
        CL_KERNEL_LOCAL_MEM_SIZE,
        sizeof(cl_ulong),
        &usedLocalMemory,
        NULL);
    if(!sampleCommon->checkVal(
        status,
        CL_SUCCESS,
        "clGetKernelWorkGroupInfo failed.(usedLocalMemory)"))
        return SDK_FAILURE;

    availableLocalMemory = totalLocalMemory - usedLocalMemory;

```

```

neededLocalMemory = blockSize*blockSize*sizeof(cl_float);

if(neededLocalMemory > availableLocalMemory)
{
    std::cout << "Unsupported: Insufficient local memory on device." << std::endl;
    return SDK_SUCCESS;
}

if(localThreads[0] > maxWorkItemSizes[0] ||
   localThreads[1] > maxWorkItemSizes[1] ||
   localThreads[0]*localThreads[1] > maxWorkGroupSize )
{
    std::cout << "Unsupported: Device does not support requested number of work items." << std::endl;
    return SDK_SUCCESS;
}

status = clSetKernelArg(
    kernel,
    0,
    sizeof(cl_mem),
    (void *)&outputBuffer);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clSetKernelArg failed. (outputBuffer)"))
    return SDK_FAILURE;

/* 2nd kernel argument - input */
status = clSetKernelArg(
    kernel,
    1,
    sizeof(cl_mem),
    (void *)&inputBuffer);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clSetKernelArg failed. (inputBuffer)"))
    return SDK_FAILURE;

/* 3rd kernel argument - block of blockSize x blockSize floats */
status = clSetKernelArg(
    kernel,
    2,
    sizeof(cl_float)*blockSize*blockSize,
    NULL);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clSetKernelArg failed. (block)"))
    return SDK_FAILURE;

/* 4th kernel argument - width */
status = clSetKernelArg(
    kernel,
    3,
    sizeof(cl_int),
    (void *)&width);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clSetKernelArg failed. (width)"))
    return SDK_FAILURE;

```

```

/* 5th kernel argument - height */
status = clSetKernelArg(
    kernel,
    4,
    sizeof(cl_int),
    (void*)&height);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clSetKernelArg failed. (height)"))
    return SDK_FAILURE;

/* 6th kernel argument - blockSize */
status = clSetKernelArg(
    kernel,
    5,
    sizeof(cl_int),
    (void*)&blockSize);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clSetKernelArg failed. (blockSize)"))
    return SDK_FAILURE;

/*
 * Enqueue a kernel run call.
 */
status = clEnqueueNDRangeKernel(
    commandQueue,
    kernel,
    2,
    NULL,
    globalThreads,
    localThreads,
    0,
    NULL,
    &events[0]);

if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clEnqueueNDRangeKernel failed."))
    return SDK_FAILURE;

/* wait for the kernel call to finish execution */
status = clWaitForEvents(1, &events[0]);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clWaitForEvents failed0."))
    return SDK_FAILURE;

if(timing)
{
    status = clGetEventProfilingInfo(
        events[0],
        CL_PROFILING_COMMAND_START,
        sizeof(long long),
        &kernelsStartTime,
        NULL);
    if(!sampleCommon->checkVal(

```

```

        status,
        CL_SUCCESS,
        "clGetEventProfilingInfo failed.")
    return SDK_FAILURE;

    status = clGetEventProfilingInfo(
        events[0],
        CL_PROFILING_COMMAND_END,
        sizeof(long long),
        &kernelsEndTime,
        NULL);
    if(!sampleCommon->checkVal(
        status,
        CL_SUCCESS,
        "clGetEventProfilingInfo failed."))
        return SDK_FAILURE;

    /* Compute total time (also convert from nanoseconds to seconds) */
    totalKernelTime = (double)(kernelsEndTime - kernelsStartTime)/1e9;
}

clReleaseEvent(events[0]);

/* Enqueue readBuffer*/
status = clEnqueueReadBuffer(
    commandQueue,
    outputBuffer,
    CL_TRUE,
    0,
    width * height * sizeof(cl_float),
    output,
    0,
    NULL,
    &events[1]);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clEnqueueReadBuffer failed."))
    return SDK_FAILURE;

/* Wait for the read buffer to finish execution */
status = clWaitForEvents(1, &events[1]);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clWaitForEvents failed1."))
    return SDK_FAILURE;

clReleaseEvent(events[1]);

return SDK_SUCCESS;
}

```

```

/*
 * Naive matrix transpose implementation
 */
void
MatrixTranspose::matrixTransposeCPUReference(
    cl_float * output,
    cl_float * input,
    const cl_uint width,
    const cl_uint height)
{
    for(cl_uint j=0; j < height; j++)
    {
        for(cl_uint i=0; i < width; i++)
        {
            output[i*height + j] = input[j*width + i];
        }
    }
}

int
MatrixTranspose::initialize()
{
    /* Call base class Initialize to get default configuration */
    if(!this->SDKSample::initialize())
        return SDK_FAILURE;

    /* add command line option for blockSize */
    streamsdk::Option* xParam = new streamsdk::Option;
    if(!xParam)
    {
        sampleCommon->error("Memory Allocation error.\n");
        return SDK_FAILURE;
    }

    xParam->_sVersion = "x";
    xParam->_IVersion = "width";
    xParam->_description = "width of input matrix";
    xParam->_type = streamsdk::CA_ARG_INT;
    xParam->_value = &width;

    sampleArgs->AddOption(xParam);

    streamsdk::Option* yParam = new streamsdk::Option;
    if(!yParam)
    {
        sampleCommon->error("Memory Allocation error.\n");
        return SDK_FAILURE;
    }

    yParam->_sVersion = "y";
    yParam->_IVersion = "height";
    yParam->_description = "height of input matrix";
    yParam->_type = streamsdk::CA_ARG_INT;
    yParam->_value = &height;

    sampleArgs->AddOption(yParam);

    streamsdk::Option* blockSizeParam = new streamsdk::Option;
    if(!blockSizeParam)
    {
        sampleCommon->error("Memory Allocation error.\n");
        return SDK_FAILURE;
    }
}

```

```

blockSizeParam->_sVersion = "b";
blockSizeParam->_lVersion = "blockSize";
blockSizeParam->_description = "Use local memory of dimensions blockSize x blockSize";
blockSizeParam->_type = streamsdk::CA_ARG_INT;
blockSizeParam->_value = &blockSize;
sampleArgs->AddOption(blockSizeParam);

return SDK_SUCCESS;
}

```

```

int
MatrixTranspose::setup()
{
    /* width should be multiples of blockSize */
    if(width%blockSize !=0)
    {
        width = (width/blockSize + 1)*blockSize;
    }

    height = width;

    if(setupMatrixTranspose()!=SDK_SUCCESS)
        return SDK_FAILURE;

    int timer = sampleCommon->createTimer();
    sampleCommon->resetTimer(timer);
    sampleCommon->startTimer(timer);

    if(setupCL()!=SDK_SUCCESS)
        return SDK_FAILURE;

    sampleCommon->stopTimer(timer);

    setupTime = (cl_double)sampleCommon->readTimer(timer);

    return SDK_SUCCESS;
}

```

```

int
MatrixTranspose::run()
{
    /* Arguments are set and execution call is enqueued on command buffer */
    if(runCLKernels()!=SDK_SUCCESS)
        return SDK_FAILURE;
    if(!quiet) {
        sampleCommon->printArray<cl_float>("Output", output, width, 1);
    }

    return SDK_SUCCESS;
}

```

```

int
MatrixTranspose::verifyResults()
{
    if(verify)
    {
        /*
        * reference implementation
        */
        int refTimer = sampleCommon->createTimer();
        sampleCommon->resetTimer(refTimer);
    }
}

```

```

sampleCommon->startTimer(refTimer);
matrixTransposeCPUReference(verificationOutput, input, width, height);
sampleCommon->stopTimer(refTimer);
referenceKernelTime = sampleCommon->readTimer(refTimer);

/* compare the results and see if they match */
if(sampleCommon->compare(output, verificationOutput, width*height))
{
    std::cout<<"Passed!\n";
    return SDK_SUCCESS;
}
else
{
    std::cout<<"Failed!\n";
    return SDK_FAILURE;
}
}

return SDK_SUCCESS;
}

void
MatrixTranspose::printStats()
{
    std::string strArray[2] = {"WxH" , "Time(sec)"};
    std::string stats[2];

    totalTime = setupTime + totalKernelTime;

    stats[0] = sampleCommon->toString(width, std::dec)
        +"x"+sampleCommon->toString(height, std::dec);
    stats[1] = sampleCommon->toString(totalTime, std::dec);

    this->SDKSample::printStats(strArray, stats, 2);
}

int
MatrixTranspose::cleanup()
{
    /* Releases OpenCL resources (Context, Memory etc.) */
    cl_int status;

    status = clReleaseKernel(kernel);
    if(!sampleCommon->checkVal(
        status,
        CL_SUCCESS,
        "clReleaseKernel failed.))
        return SDK_FAILURE;

    status = clReleaseProgram(program);
    if(!sampleCommon->checkVal(
        status,
        CL_SUCCESS,
        "clReleaseProgram failed.))
        return SDK_FAILURE;

    status = clReleaseMemObject(inputBuffer);
    if(!sampleCommon->checkVal(
        status,
        CL_SUCCESS,
        "clReleaseMemObject failed.))
        return SDK_FAILURE;
}

```

```

status = clReleaseMemObject(outputBuffer);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clReleaseMemObject failed.))
return SDK_FAILURE;

status = clReleaseCommandQueue(commandQueue);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clReleaseCommandQueue failed.))
return SDK_FAILURE;

status = clReleaseContext(context);
if(!sampleCommon->checkVal(
    status,
    CL_SUCCESS,
    "clReleaseContext failed.))
return SDK_FAILURE;

/* release program resources (input memory etc.) */
if(input)
    free(input);

if(output)
    free(output);

if(verificationOutput)
    free(verificationOutput);

if(devices)
    free(devices);

if(maxWorkItemSizes)
    free(maxWorkItemSizes);

return SDK_SUCCESS;
}

int main(int argc, char * argv[])
{
    MatrixTranspose clMatrixTranspose("OpenCL Matrix Transpose");

    clMatrixTranspose.initialize();
    if(!clMatrixTranspose.parseCommandLine(argc, argv))
        return SDK_FAILURE;
    if(clMatrixTranspose.setup()!=SDK_SUCCESS)
        return SDK_FAILURE;
    if(clMatrixTranspose.run()!=SDK_SUCCESS)
        return SDK_FAILURE;
    if(clMatrixTranspose.cleanup()!=SDK_SUCCESS)
        return SDK_FAILURE;
    clMatrixTranspose.printStats();

    return SDK_SUCCESS;
}

```

```

/*
 * Copies a block to the local memory
 * and copies back the transpose from local memory to output
 * @param output output matrix
 * @param input input matrix
 * @param block local memory of size blockSize x blockSize
 * @param width width of the input matrix
 * @param height height of the input matrix
 * @param blockSize size of the block
 */

__kernel
void matrixTranspose(__global float * output,
                    __global float * input,
                    __local float * block,
                    const uint width,
                    const uint height,
                    const uint blockSize
                    )
{
    uint globalIdx = get_global_id(0);
    uint globalIdy = get_global_id(1);

    uint localIdx = get_local_id(0);
    uint localIdy = get_local_id(1);

    /* copy from input to local memory */
    block[localIdy*blockSize + localIdx] = input[globalIdy*width + globalIdx];

    /* wait until the whole block is filled */
    barrier(CLK_LOCAL_MEM_FENCE);

    uint groupIdx = get_group_id(0);
    uint groupIdy = get_group_id(1);

    /* calculate the corresponding target location for transpose by inverting x and y values*/
    uint targetGlobalIdx = groupIdy*blockSize + localIdx;
    uint targetGlobalIdy = groupIdx*blockSize + localIdx;

    /* calculate the corresponding raster indices of source and target */
    uint targetIndex = targetGlobalIdy*height + targetGlobalIdx;
    uint sourceIndex = localIdy * blockSize + localIdx;

    output[targetIndex] = block[sourceIndex];
}

```