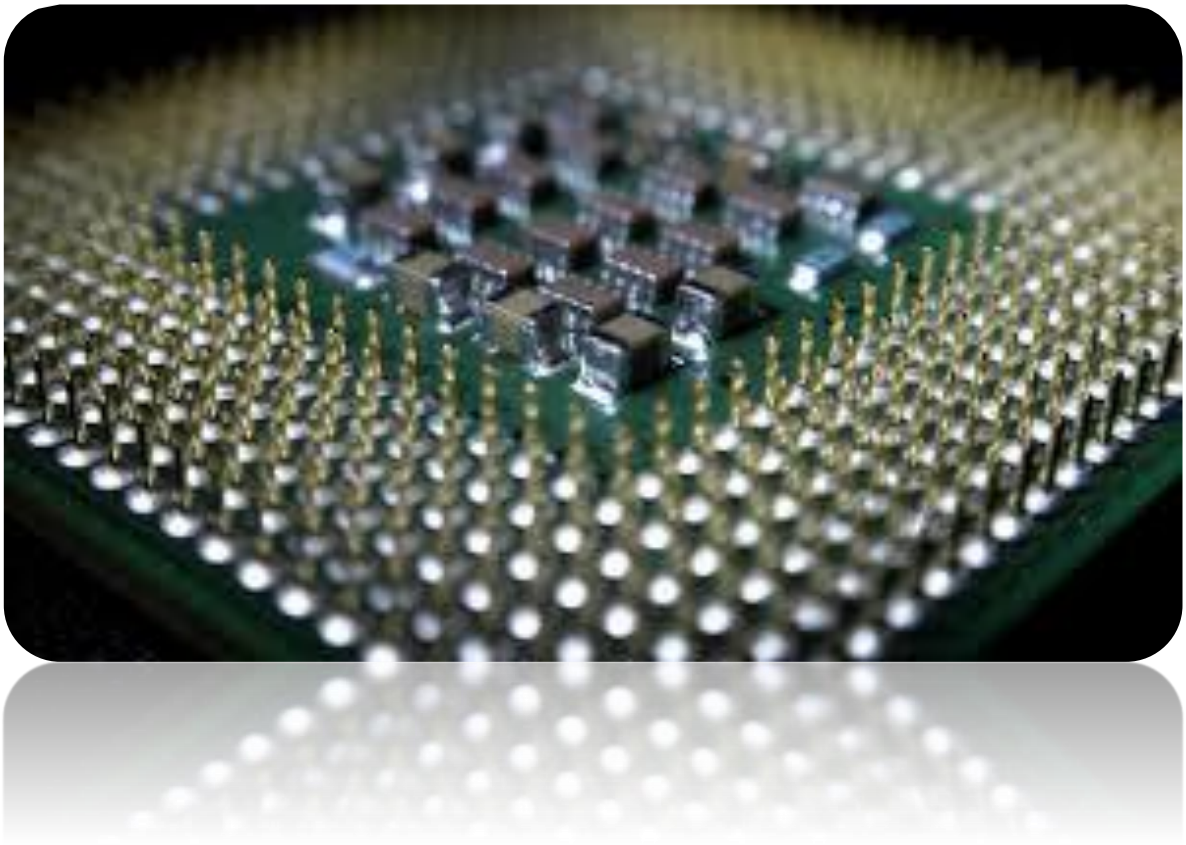
PARALLEL & DISTRIBUTED COMPUTING (CSE 4001)

PROJECT

KUMAR R



FALL SEMESTER 17-18

CUDA PROGRAMMING(NVIDIA)

Kashish Miglani - 15BCE1003
Osho Agyeya - 15BCE1326
Utsav Rai - 15BCE1352

ADDING 2 ARRAYS(VECTORS)

Key Function :

1. cudaMemcpy : This function helps in copying the data from host to device and vice-versa
2. cudaMalloc : This function allocates memory in the GPU
3. cudaFree : This function will free the memory allocated.

Functioning :

1. Creating 2 arrays and a result array will be initialised with zero.
2. All these arrays are then passed as parameter to a function , in that function first the input arrays are copies into the GPU memory.
3. We launched a kernel on the GPU with one thread for each element.
4. In the GPU the thread id is fetched and then using that id the corresponding location in the result array is modified.
5. Then we print the final array

CODE :

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int
size);

__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

int main()
{
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };

    // Add vectors in parallel.
    cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
    if (cudaStatus != cudaSuccess) {
```

```
        fprintf(stderr, "addWithCuda failed!");
        return 1;
    }

    printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
        c[0], c[1], c[2], c[3], c[4]);

    // cudaDeviceReset must be called before exiting in order for
profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete
traces.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }

    return 0;
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int
size)
{
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;
    cudaError_t cudaStatus;

    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable
GPU installed?");
        goto Error;
    }

    // Allocate GPU buffers for three vectors (two input, one output)    .
    cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
```

```
    }

    // Copy input vectors from host memory to GPU buffers.
    cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    // Launch a kernel on the GPU with one thread for each element.
    addKernel<<<size, size>>>(dev_c, dev_a, dev_b);

    // Check for any errors launching the kernel
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addKernel launch failed: %s\n",
cudaGetErrorString(cudaStatus));
        goto Error;
    }

    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceSynchronize returned error code %d
after launching addKernel!\n", cudaStatus);
        goto Error;
    }

    // Copy output vector from GPU buffer to host memory.
    cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int),
cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

Error:
    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);

    return cudaStatus;
}
```

Output:

ADDING 2 MATRICES

Key Function :

1. cudaMemcpy : This function helps in copying the data from host to device and vice-versa
2. cudaMalloc : This function allocates memory in the GPU
3. cudaFree : This function will free the memory allocated.
4. dim3 : this is the function which is used to generate threads in multi-dimension
   eg dim3 X(N,N)

Functioning :

1. Creating 2 matrix and a result matrix will be initialised with zero.
2. All these matrices will be copied into the GPU memory
3. We launched a kernel on the GPU with threads in two dimensions x and y.
4. In the GPU the thread id is fetched for x component and y component and then using that id the corresponding location in the result matrix is modified.
5. Then we print the final matrix

CODE:

```
#include "cuda_runtime.h"
//matrix Addition
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
```

```
#include <cuda_runtime.h>
//#include "kernel.h"

#define N 3
//int fin[N][N] = { 0 };
__global__ void MatAdd(int A[][N], int B[][N],int C[][N]) {
int i = threadIdx.x;
int j = threadIdx.y;

C[i][j] = A[i][j] + B[i][j];
/*
#if __CUDA_ARCH__ >= 200
atomicAdd(&C[i][j], A[i][j] + B[i][j]);
#endif
*/

}


int main() {

int A[N][N] = { { 1,2,3 },{ 3,4,2 },{1,2,3 } };
int B[N][N] = { { 5,6,1 },{ 7,8,2 },{1,2,3} };
int C[N][N] = { { 0,0,0 },{ 0,0,0 },{0,0,0} };

int(*pA)[N], (*pB)[N], (*pC)[N],(*pf)[N];

cudaMalloc((void**)&pA, (N*N) * sizeof(int));
cudaMalloc((void**)&pB, (N*N) * sizeof(int));
cudaMalloc((void**)&pC, (N*N) * sizeof(int));

cudaMemcpy(pA, A, (N*N) * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(pB, B, (N*N) * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(pC, C, (N*N) * sizeof(int), cudaMemcpyHostToDevice);
//cudaMemcpy(pf, fin, (N*N) * sizeof(int), cudaMemcpyHostToDevice);
int numBlocks = 1;
dim3 threadsPerBlock(N, N);
MatAdd << <numBlocks, threadsPerBlock >>>(pA, pB, pC);

cudaMemcpy(C, pC, (N*N) * sizeof(int), cudaMemcpyDeviceToHost);

int i, j; printf("C = \n");
for (i = 0; i<N; i++) {
for (j = 0; j<N; j++) {
printf("%d ", C[i][j]);
}
printf("\n");
}

cudaFree(pA);
cudaFree(pB);
cudaFree(pC);

printf("\n");
```

```
return 0;
}
```

Output :

MULTIPLYING 2 MATRICES

Key Function :

1. cudaMemcpy : This function helps in copying the data from host to device and vice-versa
2. cudaMalloc : This function allocates memory in the GPU
3. cudaFree : This function will free the memory allocated.
4. dim3 : this is the function which is used to generate threads in multi-dimension
   i. eg dim3 X(N,N)

Functioning :

1. Creating 2 matrix and a result matrix will be initialised with zero.
2. All these matrices will be copied into the GPU memory
3. We launched a kernel on the GPU with threads in two dimensions x and y.
4. Here is a catch , we have followed a different mechanism of multiplying as at a time we will be able to get the id of a particular element in a matrix.
5. So to allow multiplication to work fine , what we did is :
   a. We summed the value of that block of the matrix to all the indexes in the final result matrix.
   b. We used atomic add for this so that add all the threads will update the matrix value one at a time.

6. Then we print the final matrix.

CODE:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <device_functions.h>
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>


#define N 3
__global__ void MatMul(int A[][N], int B[][N], int C[][N]) {
   int i = threadIdx.x;
   int j = threadIdx.y;
   int temp;
   for (int j1 = 0; j1 < N; j1++)
   {
         temp = A[i][j] * B[j][j1];
         #if __CUDA_ARCH__ >= 200
         atomicAdd(&C[i][j1], temp);
         #endif
   }
}


int main() {

   int A[N][N] = { { 1,2,3 },{ 4,5,6 },{ 7,8,9 } };
   int B[N][N] = { { 1,2,3 },{ 4,5,6 },{ 7,8,9 } };
   int C[N][N] = { { 0,0,0 },{ 0,0,0 },{ 0,0,0 } };

   int(*pA)[N], (*pB)[N], (*pC)[N], (*pf)[N];

   cudaMalloc((void**)&pA, (N*N) * sizeof(int));
   cudaMalloc((void**)&pB, (N*N) * sizeof(int));
   cudaMalloc((void**)&pC, (N*N) * sizeof(int));

   cudaMemcpy(pA, A, (N*N) * sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(pB, B, (N*N) * sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(pC, C, (N*N) * sizeof(int), cudaMemcpyHostToDevice);
   int numBlocks = 1;
   dim3 threadsPerBlock(N,N);
   MatMul << <numBlocks, threadsPerBlock >> >(pA, pB, pC);

   cudaMemcpy(C, pC, (N*N) * sizeof(int), cudaMemcpyDeviceToHost);

   int i, j; printf("\nC = \n");
   for (i = 0; i<N; i++) {
         for (j = 0; j<N; j++) {
```

```
                printf("%d ", C[i][j]);
            }
            printf("\n");
        }

        cudaFree(pA);
        cudaFree(pB);
        cudaFree(pC);

        printf("\n");

        return 0;
    }
```

Output :

HISTOGRAM PROCESSING

Key Function :

1. cudaMemcpy : This function helps in copying the data from host to device and vice-versa
2. cudaMalloc : This function allocates memory in the GPU
3. cudaFree : This function will free the memory allocated.

Functioning :

1. Creating one arrays and a result array will be initialised with zero.
2. All these arrays are then passed as parameter to a function , in that function first the input arrays are copies into the GPU memory.
3. We launched a kernel on the GPU with one thread for each element.

4. In the GPU the thread id is fetched and then using that id the corresponding location in the result array is modified in the following way :
    i. We will update the index equal to the value of the array element by ‘1’.
    ii. We will use atomic add for this purpose as we want one thread to update the value of the result array at a time so that the final array obtained is consistent.
5. Then we print the final array.

CODE :

```
//histogram processing

#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

cudaError_t findFreqWithCuda(int *c, unsigned int fsize, const int *b,
unsigned int size);



__global__ void addKernel(int *c, const int *b)
{
int i = threadIdx.x;
#if __CUDA_ARCH__ >= 200
atomicAdd(&c[b[i]],1);
#endif
}

int main()
{
const int arraySize = 11;
const int fsize = 10005;//maximum value of the number in the array can be
at max 10005
const int a[arraySize] = { 1, 2, 3, 4, 5,1,1,1,2,3,6 };
int maxx = INT_MIN;
for (int i = 0; i < arraySize; i++)
{
//maxx = max(maxx, arr[i]);
if (maxx < a[i])
maxx=a[i];
}
int c[fsize] = { 0 };

// Add vectors in parallel.
cudaError_t cudaStatus = findFreqWithCuda(c,fsize,a,arraySize);
if (cudaStatus != cudaSuccess) {
```

```
fprintf(stderr, "findFreqWithCuda failed!");
return 1;
}

for (int i = 0; i <= maxx; i++)
{
if (c[i] != 0)
{
printf("%d occurs %d times\n", i, c[i]);
}
}

// cudaDeviceReset must be called before exiting in order for profiling
and
// tracing tools such as Nsight and Visual Profiler to show complete
traces.
cudaStatus = cudaDeviceReset();
if (cudaStatus != cudaSuccess) {
fprintf(stderr, "cudaDeviceReset failed!");
return 1;
}

return 0;
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t findFreqWithCuda(int *c, unsigned int fsize, const int *b,
unsigned int size)
{
// int *dev_a = 0;
int *dev_b = 0;
int *dev_c = 0;
cudaError_t cudaStatus;

// Choose which GPU to run on, change this on a multi-GPU system.
cudaStatus = cudaSetDevice(0);
if (cudaStatus != cudaSuccess) {
fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU
installed?");
goto Error;
}

// Allocate GPU buffers for three vectors (two input, one output)    .
cudaStatus = cudaMalloc((void**)&dev_c, fsize * sizeof(int));
if (cudaStatus != cudaSuccess) {
fprintf(stderr, "cudaMalloc failed!");
goto Error;
}



cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
if (cudaStatus != cudaSuccess) {
fprintf(stderr, "cudaMalloc failed!");
```

```
goto Error;
}

// Copy input vectors from host memory to GPU buffers.


cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int),
cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
fprintf(stderr, "cudaMemcpy failed!");
goto Error;
}

// Launch a kernel on the GPU with one thread for each element.
//dim3 threadsPerBlock(11,11,11);
addKernel<<<1,size>>>(dev_c,dev_b);

// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
fprintf(stderr, "addKernel launch failed: %s\n",
cudaGetErrorString(cudaStatus));
goto Error;
}

// cudaDeviceSynchronize waits for the kernel to finish, and returns
// any errors encountered during the launch.
cudaStatus = cudaDeviceSynchronize();
if (cudaStatus != cudaSuccess) {
fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
launching addKernel!\n", cudaStatus);
goto Error;
}

// Copy output vector from GPU buffer to host memory.
cudaStatus = cudaMemcpy(c, dev_c, fsize * sizeof(int),
cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess) {
fprintf(stderr, "cudaMemcpy failed!");
goto Error;
}

Error:
cudaFree(dev_c);
cudaFree(dev_b);

return cudaStatus;
}
```

OUTPUT:

CONVERTING COLOURED TO GREYSCALE IMAGE

Key Function :

1. program.cpp - main function.
2. image.h and image.cpp - responsible for loading and saving images, initializing CUDA kernel and managing memory;
3. utilities - checks runtime errors and compares pixels from GPU conversion to reference conversion on CPU
4. gputimer - events to measure execution time on the GPU in millisecond

Functioning :

1. We'll create a console application where our main function will get RGB and gray image paths from the input arguments and call into a helper class to do all the processing.

2. The image class will contain representation of images on the host (CPU) and the device (GPU) and one function to do the conversion from RGBA to gray.

3. The convert to gray function has to do the following 3 things:

- Initialize kernel and copy RGBA image to the GPU memory;
- Execute CUDA kernel;
- Copy gray image from the GPU back to main memory.

4. In the kernel.cu CUDA file, we have to allocate number of parallel executions on which we will process each pixel in the image and call actual kernel

5. Here, before calling the kernel, we have to decide on the number of threads that we want to process the image on. Naturally, we want threads to be the same or close to the image size: x corresponding to image rows, and y to image columns. That's what this code does where M is number of threads per block, I selected 32. Variable block below means block width, or number of threads per block. Variable grid is number of blocks. To remind, GPU runs blocks of threads on Streaming Multiprocessors where a block can have up to 1024 threads. Block of threads executes on the same SM.

6. In the kernel, we have to compute pixel location from block id along x and y coordinates, block width (32 threads per block), and thread id within the block.

7. Finally, we check conversion results by running the same calculation on CPU and comparing pixels.

CODE :

ImageKernel.cu

```
//-------------------------------------------------------------------------
-----
// File: ImageKernel.cu
//
// Desc: CUDA kernel to convert RGBA image to gray.
//
//           This kernel assumes 4-bytes per pixel RGBA image with
channels Red, Green, Blue, and Alpha
//           represented each by one byte (8-bits) and a range of values
between 0 and 255 (2^8 - 1).
//
//           Grey scale images are represented by a single intensity value
per pixel
//           where each pixel is only 1 byte.
//
//           Human eye perceives red, green, and blue colors unequally
(humans are more sensitive to green and least to blue)
//           and for that reason we will use weighted formula
(http://en.wikipedia.org/wiki/Grayscale):
//
//          I = 0.2126 * R + 0.7152 * G + 0.0722 * B
//
//
//
```

```
//-----------------------------------------------------------------------
-----

#include "stdafx.h"
#include "Image.h"
//////////////////////
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
//////////////////////

using namespace Bisque;

using std::ceilf;

#define N (1024 * 1024)                  // blocks
#define BLOCK_WIDTH 32                   // threads per block

// Converts RGBA image to gray scale intensity using the following
formula:
// I = 0.2126 * R + 0.7152 * G + 0.0722 * B
__global__
void rgba_to_grayscale(unsigned char* const gray, const uchar4* const
rgba, int rows, int cols)
{
      int r = blockIdx.y * blockDim.y + threadIdx.y;        // current row
      int c = blockIdx.x * blockDim.x + threadIdx.x;        // current
column

      if ((r < rows) && (c < cols))
      {
            int idx = c + cols * r;       // current pixel index

            uchar4 pixel = rgba[idx];
            float intensity = 0.2126f * pixel.x + 0.7152f * pixel.y +
0.0722f * pixel.z;

            gray[idx] = (unsigned char)intensity;
      }
}

// Runs r8g8b8a8 to gray kernel
void RunRGBAtoGrayKernel(
      unsigned char*    gray,                   // gray image: 1 byte per
image --> this will be returned from this function
      uchar4*                 rgba,                   // rgba image: 4
bytes per image
      int                     rows,                   // image size: number
of rows
      int                     cols                    // image size: number
of columns
)
{
      const char* func = "RunGrayKernel";
```

```
	cudaError hr = cudaSuccess;

	int x = static_cast<int>(ceilf(static_cast<float>(cols) /
BLOCK_WIDTH));
	int y = static_cast<int>(ceilf(static_cast<float>(rows) /
BLOCK_WIDTH));

	const dim3 grid(x, y, 1);
	// number of blocks
	const dim3 block(BLOCK_WIDTH, BLOCK_WIDTH, 1);		// block
width: number of threads per block

	rgba_to_grayscale<<<grid, block>>>(gray, rgba, rows, cols);

	hr = cudaDeviceSynchronize();
	CHECK_CUDA_ERROR(hr,
func, "rgba_to_grayscale failed.");
}
```

Program.cpp

```
////////////////////////////////////////////////////

// program.cpp
//
// Main entry point into the application.
//

#include "stdafx.h"
#include "Image.h"

using std::cout;
using std::cerr;
using std::endl;
using std::exception;
using std::string;

using namespace Bisque;

// Main entry into the application
int main(int argc, char** argv)
{
	string imagePath="C:\\Users\\oshoa\\Documents\\Visual Studio
2015\\Projects\\greyImage\\greyImage\\Media\\tulip.jpg";
	string outputPath="C:\\Users\\oshoa\\Documents\\Visual Studio
2015\\Projects\\greyImage\\greyImage\\Media\\tulip-g.jpg";

	/*if (argc > 2)
	{
		imagePath = string(argv[1]);
```

```
            outputPath = string(argv[2]);
      }
      else
      {
            cerr << "Please provide input and output image files as
arguments to this application." << endl;
            exit(1);
      }
      */

      Image image;

      try
      {
            image.ConvertRGBAtoGray(imagePath, outputPath);
      }
      catch (exception& e)
      {
            cerr << endl << "ERROR: " << e.what() << endl;
            exit(1);
      }

      cout << "Done!" << endl << endl;
      return 0;
}
```

Image.cpp

```
#include "stdafx.h"
#include "Image.h"

using namespace Bisque;

using cv::Mat;
using cv::cvtColor;
using cv::imread;
using cv::imwrite;

using std::cout;
using std::endl;
using std::exception;
using std::runtime_error;

using std::chrono::duration_cast;
using std::chrono::milliseconds;
using std::chrono::microseconds;
using std::chrono::system_clock;
using std::chrono::time_point;

// Forward declarations
void RunRGBAtoGrayKernel(
```

```
	unsigned char*	gray,			// gray image: 1 byte per
image --> this will be returned from this function
	uchar4*			rgba,			// rgba image: 4
bytes per image
	int				rows,			// image size: number
of rows
	int				cols			// image size: number
of columns
);

// Ctor
Image::Image(void)
{
	m_device.gray = nullptr;
	m_device.rgba = nullptr;
}

Image::~Image(void)
{
}

//Converts r8g8b8a8 image to one channel gray
void Image::ConvertRGBAtoGray(const string& imagePath, const string&
outputPath)
{
	const char* func = "Image::ConvertRGBAtoGray";

	cudaError hr = cudaSuccess;

	time_point<system_clock> start;
	time_point<system_clock> stop;

	// Load image and initialize kernel
	KernelMap host;
	KernelMap device;

	start = system_clock::now();

	InitializeKernel(host, device, imagePath);

	stop = system_clock::now();
	long long ms = duration_cast<milliseconds>(stop - start).count();
	long long us = duration_cast<microseconds>(stop - start).count();

	cout << "InitializeKernel executed in " << us << "us (" << ms <<
"ms)" << endl;

	// Run kernel: convert rgba image to gray
	start = system_clock::now();

	GpuTimer gpuTimer;
	gpuTimer.Start();

	RunRGBAtoGrayKernel(
```

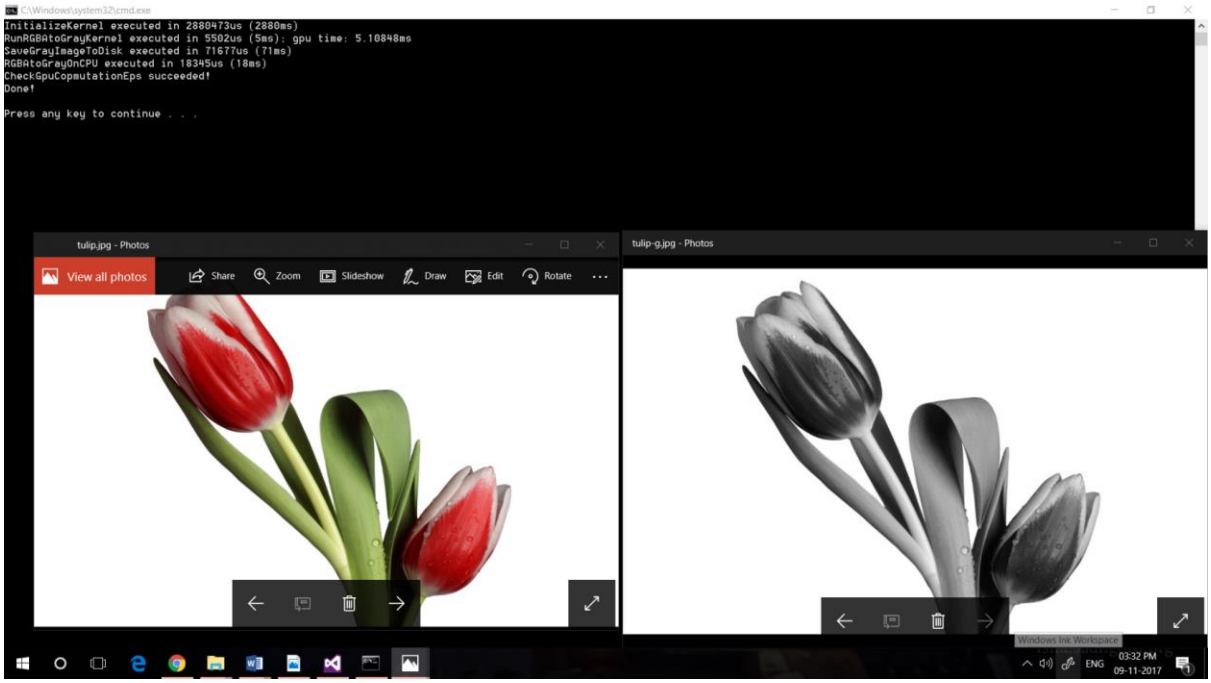
```
            device.gray,
            device.rgba,
            m_host.rgba.rows,
            m_host.rgba.cols
      );

      gpuTimer.Stop();

      stop = system_clock::now();
      ms = duration_cast<milliseconds>(stop - start).count();
      us = duration_cast<microseconds>(stop - start).count();

      cout << "RunRGBAtoGrayKernel executed in " << us << "us (" << ms <<
"ms); gpu time: " << gpuTimer.Elapsed() << "ms" << endl;

      // Save gray image to disk
      start = system_clock::now();

      SaveGrayImageToDisk(outputPath);

      stop = system_clock::now();
      ms = duration_cast<milliseconds>(stop - start).count();
      us = duration_cast<microseconds>(stop - start).count();

      cout << "SaveGrayImageToDisk executed in " << us << "us (" << ms <<
"ms)" << endl;

#if 1             // Change to 1 to enable
      // Validate GPU convertion against CPU result.
      // Only turn it when you want to run validation because CPU
calculation will be slow.
      VerifyGpuComputation(host.rgba);
#endif

      // Release cuda
      hr = cudaFree(m_device.gray);
      hr = cudaFree(m_device.rgba);
}

// Initializes CUDA kernel, as well as loads teh image from disk and
prepares
// image rgba and gray-single-channel handles both on the host and the
device.
void Image::InitializeKernel(KernelMap& host, KernelMap& device, const
string& imagePath)
{
      const char* func = "Image::InitializeKernel";
      CV_FUNCNAME("Image::InitializeKernel");
      __CV_BEGIN__

            cudaError hr = cudaFree(nullptr);

            CHECK_CUDA_ERROR(hr, func, "Could not free CUDA memory
space.");
```

```
	// Load image
	Mat image = imread(imagePath.c_str(), CV_LOAD_IMAGE_COLOR);
	if (image.empty())
	{
		string msg = "Could not open image file: " + imagePath;
		throw runtime_error(msg);
	}

	// Convert image from BGR to RGB
	cvtColor(image, m_host.rgba, CV_BGR2RGBA);

	// Allocate memory for the gray image (on the host)
	m_host.gray.create(image.rows, image.cols, CV_8UC1);

	CV_ASSERT(m_host.rgba.isContinuous());
	CV_ASSERT(m_host.gray.isContinuous());

	host.rgba = reinterpret_cast<uchar4*>(m_host.rgba.ptr<unsigned
char>(0));
	host.gray = m_host.gray.ptr<unsigned char>(0);

	const int numPixels = m_host.rgba.rows * m_host.rgba.cols;

	// Allocate memory on the device for both rgba and gray images,
	// then fill gray image with zeros to make sure there is no memory
left laying around.
	// Finally, copy rgba image to the GPU.
	hr = cudaMalloc(&device.rgba, numPixels * sizeof(uchar4));
							CHECK_CUDA_ERROR(hr, func,
"Could not allocate device memory for RGBA image.");
	hr = cudaMalloc(&device.gray, numPixels * sizeof(unsigned char));
						CHECK_CUDA_ERROR(hr, func,
"Could not allocate device memory for gray image.");
	hr = cudaMemset(device.gray, 0, numPixels * sizeof(unsigned char));
							CHECK_CUDA_ERROR(hr, func,
"Could not fill gray image with constant values.");
	hr = cudaMemcpy(device.rgba, host.rgba, numPixels * sizeof(uchar4),
cudaMemcpyHostToDevice);			CHECK_CUDA_ERROR(hr, func,
"Could not copy rgba image from host to device.");

	// Save device pointers
	m_device.gray = device.gray;
	m_device.rgba = device.rgba;

	__CV_END__
}

// Copies gray image from device to the host and saves it to disk
void Image::SaveGrayImageToDisk(const string& outputPath)
{
	const char* func = "Image::SaveGrayImageToDisk";
	cudaError hr = cudaSuccess;
```

```
	const int numPixels = m_host.rgba.rows * m_host.rgba.cols;

	// Copy gray image from device to the host
	hr = cudaMemcpy(
		m_host.gray.ptr<unsigned char>(0),
		m_device.gray,
		numPixels * sizeof(unsigned char),
		cudaMemcpyDeviceToHost
	);

			CHECK_CUDA_ERROR(hr, func, "Could not copy gray image
from device to the host.");

	// Save image to disk
	imwrite(outputPath.c_str(), m_host.gray);
}

// Verifies correctness of the GPU computation running the same algorithm
on the CPU
// and comparing pixel values.
//
// NOTE: Because we recompute pixels on the CPU, call this function only
when you need
//		to validate GPU computation.
void Image::VerifyGpuComputation(const uchar4* const rgba)
{
	const char* func = "Image::VerifyGpuComputation";
	cudaError hr = cudaSuccess;

	const int numPixels = m_host.rgba.rows * m_host.rgba.cols;
	unsigned char* gpu = new unsigned char[numPixels];
	// image computed on GPU
	unsigned char* ref = new unsigned char[numPixels];
	// reference image that will be computed on CPU


				// Copy device gray image to the host
	hr = cudaMemcpy(
		gpu,
		m_device.gray,
		numPixels * sizeof(unsigned char),
		cudaMemcpyDeviceToHost
	);

		CHECK_CUDA_ERROR(hr, func, "Could not copy gray image from
device to the host.");

	// Copmute gray image on the CPU
	time_point<system_clock> start = system_clock::now();
	time_point<system_clock> stop = system_clock::now();

	RGBAtoGrayOnCPU(ref, rgba, m_host.rgba.rows, m_host.rgba.cols);

	stop = system_clock::now();
```

```
	long long ms = duration_cast<milliseconds>(stop - start).count();
	long long us = duration_cast<microseconds>(stop - start).count();

	cout << "RGBAtoGrayOnCPU executed in " << us << "us (" << ms <<
"ms)" << endl;

	// Compare results
	// NOTE: Since this sample works with 8-bit images and ranges of
colors between 1 and 255 per channel,
	//		we only allow color to differ by value of 1 between GPU
and CPU computations.
	//		As this is a gray image with only one channel of color
intensity, we allow intensity error
	//		to be no more than 1 between the same pixel in GPU and
CPU images, and total percent of errors
	//		is expected not to be higher than 0.001
	Utilities::CheckGpuComputationEps(ref, gpu, numPixels, 1, .001);
	//Utilities::CheckGpuComputationExact(ref, gpu, numPixels);
	//Utilities::CheckGpuComputationAutodesk(ref, gpu, numPixels, 1.0,
100);

	// Clean up
	delete[] gpu;
	delete[] ref;
}

// Converts RGBA image to grayscale intensity on CPU.
// NOTE: This is a reference computation for validating results on GPU.
//		 Otherwise, do not call this function.
void Image::RGBAtoGrayOnCPU(unsigned char* gray, const uchar4* const rgba,
int rows, int cols)
{
	for (int r = 0; r < rows; ++r)
	{
		for (int c = 0; c < cols; ++c)
		{
			int	idx = c + cols * r;		// current pixel
index
			uchar4 pixel = rgba[idx];
			float  intensity = 0.2126f * pixel.x + 0.7152f * pixel.y
+ 0.0722f * pixel.z;

			gray[idx] = static_cast<unsigned char>(intensity);
		}
	}
}
```

OUTPUT


InitializeKernel executed in 2880473us (2880ms)
RunRGBAtoGrayKernel executed in 5502us (5ms): gpu time: 5.10848ms
SaveGrayImageToDisk executed in 71677us (71ms)
RGBAtoGrayOnCPU executed in 18345us (18ms)
CheckGpuCopmutationEps succeeded!
Done!
Press any key to continue . . .
tulip.jpg - Photos
View all photos
Share
Zoom
Slideshow
Draw
Edit
Rotate
tulip-g.jpg - Photos
ENG
03:32 PM
09-11-2017