Inverting Large Images Using CUDA

Finnbarr P. Murphy
(fpm@fpmurphy.com)

This is a simple example of how to invert a very large image, stored as a *vector* using nVidia's CUDA programming environment and an EVGA GeForce GTX 260 graphics card.

A GeForce GTX 260 card should have only 24 cores on it, but the nVidia SDK *deviceQuery* utility reports that there are 27 multiprocessors on my GeForce GTX 260 graphics card and I am able to use all 27 of them.

```
$ deviceQuery
CUDA Device Query (Runtime API) version (CUDART static linking)
There is 1 device supporting CUDA

Device 0: "GeForce GTX 260"
  CUDA Driver Version:                           2.30
  CUDA Runtime Version:                          2.30
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         3
  Total amount of global memory:                 938803200 bytes
  Number of multiprocessors:                     27
  Number of cores:                               216
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.35 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads can use th
is device simultaneously)
```

The application is fairly trivial. It reads in an image file in PPM P6 format, stores the image in a vector of integers, one per pixel, with 24-bits used to store the RGB values. Depending on which command line option is selected (*-c* invert using CPU, *-g* invert using GPU), it then inverts the image vector either using the CPU or the GPU (Graphical Processing Unit). Finally it writes the image out to an output file in PPM P6 format.

It uses a block size of 16 and a single kernel called *myCudaInvertPixel* which works on 16 rows of the image at the time via tiling.

```
#include
#include
#include
#include
#include
```



```
#include
#include
#include
#include

#include
#include

#define BLOCK_SIZE 16
// #define DEBUG 0

// ------------ prototypes --------------
__global__ void myCudaInvertPixel(int *, int);


class CCL {

    int width;
    int height;

public:

    CCL() { width = height = 0; }
    ~CCL() { width = height = 0; }

    void cuda_ccl(vector& img, int w)
    {
        width = w;
        height = img.size()/w;

#ifdef DEBUG
        cout = 2020
    cudaDriverGetVersion(&driverVersion);
    cudaRuntimeGetVersion(&runtimeVersion);
    printf("  CUDA Driver Version:                    %d.%d\n", driverVersion/1000, driverV
ersion%100);
    printf("  CUDA Runtime Version:                   %d.%d\n", runtimeVersion/1000, runtim
eVersion%100);
#endif
    printf("  CUDA capability revision number:        %d.%d\n", deviceProp.major, devicePro
p.minor);
#if CUDART_VERSION >= 2000
    printf("  Number of multiprocessors:             %d\n", deviceProp.multiProcessorCount)
;
    printf("  Number of cores:                        %d\n", 8 * deviceProp.multiProcessorC
ount);
    printf("  Maximum number of threads per block:    %d\n", deviceProp.maxThreadsPerBlock)
;
#endif
    printf("  Clock rate:                             %.2f GHz\n", deviceProp.clockRate * 1
e-6f);
    printf("  Global memory:                          %d\n", globalMem);
    printf("  Shared memory per block:                %d\n", deviceProp.sharedMemPerBlock);
    printf("  Registers per block:                    %d\n\n", deviceProp.regsPerBlock);
#endif

    cudaSetDevice(0);      // not needed - default is 0.

    int rowBytes = width * 4;                                                       // siz
e of a row
    int maxrow  = BLOCK_SIZE;                                                       // max
 rows that can be handled in global memory
    strake_size = maxrow * rowBytes;                                                // mem
ory to allocate
    strakes = image_size/strake_size + ((image_size % strake_size == 0) ? 0 : 1);   // the
re X strakes necessary
    last_strake_size = image_size - (strake_size * (strakes - 1));                  // fig
ure out size of last strake (may be smaller)
```



```
#if DEBUG
    printf("  Image size %d\n", image_size);
    printf("  Image vector size %d\n", image.size());
    printf("  Row size %d\n", rowBytes);
    printf("  Maxximum rows %d\n", maxrow);
    printf("  Strake size %d\n", strake_size);
    printf("  Strakes %d\n", strakes);
    printf("  Last strake size %d\n\n", last_strake_size);
#endif

    // uncomment if you want to using CUDA's elapsed timer
    // float elapsed_time_ms = 0.0f;
    // cudaEvent_t  start, stop;
    // cudaEventCreate( &start);
    // cudaEventCreate( &stop);
    // cudaEventRecord(start, 0);

    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(((width*4)/dimBlock.x) + (((width*4)%dimBlock.x)?1:0), BLOCK_SIZE);

    cudaMalloc((void**)&cuda_image_ptr, strake_size);
    if ((err = cudaGetLastError()) != cudaSuccess) {
        fprintf(stderr, "ERROR: Cuda: %s\n", cudaGetErrorString(err));
        exit(2);
    }

    for (int i = 0; i  0 && i == strakes - 1) {
            this_strake_size = last_strake_size;
        } else {
            this_strake_size = strake_size;
        }
        offset = 0 + i *(strake_size/4);

        cudaMemcpy(cuda_image_ptr, &image[offset], this_strake_size, cudaMemcpyHostToDevic
e);
        if ((err = cudaGetLastError()) != cudaSuccess) {
            fprintf(stderr, "ERROR: Cuda: %s\n", cudaGetErrorString(err));
            exit(2);
        }

        // invoke CUDA kernel
        myCudaInvertPixel>>(cuda_image_ptr, width);
        if ((err = cudaGetLastError()) != cudaSuccess) {
           fprintf(stderr, "ERROR: Cuda: %s\n", cudaGetErrorString(err));
           exit(2);
        }

        // not needed here - cudaThreadSynchronize();

        cudaMemcpy( &image[offset], cuda_image_ptr, this_strake_size, cudaMemcpyDeviceToHo
st);
        if ((err = cudaGetLastError()) != cudaSuccess) {
            fprintf(stderr, "ERROR: Cuda: %s\n", cudaGetErrorString(err));
            exit(2);
        }
    }

    cudaFree(cuda_image_ptr);

    // uncomment if you want to using CUDA's elapsed timer
    // cudaEventRecord(stop, 0);
    // cudaEventSynchronize(stop);
    // cudaEventElapsedTime( &elapsed_time_ms, start, stop);
    // printf("Elapsed time per CUDA: %f\n", elapsed_time_ms);

    return 0;
}
```



```
// CUDA kernel definition - invert pixel
__global__ void myCudaInvertPixel(int *pixel, int width)
{
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = ix + iy*width;

    if (ix = BLOCK_SIZE) return;

    pixel[idx] = 0x00FFFFFF - pixel[idx];
}


class Timer {

    private:

        timespec starttime, endtime;
        float elapsed;

        float difftime(timespec start, timespec end) {
            float temp;

            temp = (end.tv_sec - start.tv_sec)  +
                   (float) (end.tv_nsec - start.tv_nsec) / 1000000000.0;

            return temp;
        }

    public:

        Timer() { elapsed = 0.00; }

        void reset() { elapsed = 0.00; }

        void start() {
            elapsed = 0.00;
            clock_gettime(CLOCK_MONOTONIC, &starttime);
        }

        void stop() {
            clock_gettime(CLOCK_MONOTONIC, &endtime);
            elapsed = elapsed + difftime(starttime, endtime);
        }

        void restart() { clock_gettime(CLOCK_MONOTONIC, &starttime); }

        void print() { cout  image;
    friend class CCL;

  public:

    PPM() {
        width = 0;
        height = 0;
        maxColor = 0;
    }

    int read(char *filename);

    int write(char* filename);

    void printsize() {
       cout ::iterator it = image.begin(); it != image.end(); ++it) {
*it = 0x00FFFFFF - *it;
}
}
```



```
void invertpixelCuda() {
CCL ccl;
ccl.cuda_ccl(image, width);
}

int getwidth() { return width; }

};


int
PPM::read(char *filename)
{
FILE *fp = fopen(filename, "r");
int rgb, x, y;
unsigned char red, green, blue;
char buf[256];
int *pwidth = &width;
int *pheight = &height;

if (fp== NULL) return 1;

do {
fgets(buf, 256, fp);
} while(buf[0] == '#');

if (buf[0] != 'P' || buf[1] != '6') {
fclose(fp);
return 1;
}

do {
fgets(buf, 256, fp);
} while(buf[0] == '#');

sscanf(buf, "%u %u", pwidth, pheight);
fscanf(fp, "%u\n", &maxColor);

image.resize(height*width);
for (y = 0; y > 16;
green = (rgb & 0x0000FF00) >> 8;
red = (rgb & 0x000000FF);
fprintf(fp, "%c%c%c", red, green, blue);
}
}

fclose(fp);

return(0);
}


void
usage(char *name,
int mode)
{
cerr << "Usage: " << name << " [-c] [-d] [-g] infile outfile" << endl;
exit(mode);
}


int
main(int argc,
char *argv[])
{
int c;
int cflag = 0, dflag = 0, gflag = 0;
PPM ppm;
```



```
Timer timer;

if (argc == 1)
usage(argv[0], 0);

opterr = 0;
while ((c = getopt( argc, argv, "cdgh")) != -1) {
switch (c) {
case 'c':
cflag = 1;
break;
case 'd':
dflag = 1;
break;
case 'g':
gflag = 1;
break;
case '?':
case 'h':
usage(argv[0], 0);
break;
}
}

if (optind != argc-2) {
cerr << "ERROR: An input file and an output file must be specified" << endl;
usage(argv[0], 1);
}

if (ppm.read(argv[optind++])) {
cerr << "ERROR: Failed to read input file" << endl;
exit(1);
}

timer.start();

if (dflag) ppm.printsize();

if (cflag) ppm.invertpixelNoCuda();

if (gflag) ppm.invertpixelCuda();

timer.stop();

if (ppm.write(argv[optind])) {
cerr << "ERROR: Failed to write output file" << endl;
exit(1);
}

timer.print();

exit(0);
}
```


Well that is all there is to this application. To compile it you need to have at least the nVidia CUDA 2.3 SDK installed and the latest nVidia graphics card driver.

If you are new to CUDA programming, I suggest you download the source code, compile it and play around with it on your own computer. The man pages for Linux do not work (they are all mixed up) but good up-to-date documentation can be found online at nVidia CUDA Library Documentation.

You might also want to play around with the block (*dimBlock*) and grid (*dimGrid*) dimensions. nVidia has a useful Microsoft Excel Occupancy Calculator Spreadsheet.

P.S. I tested the application on images up to a 32000 x 32000 pixels on RHEL5.4 using the CUDA 2.3 SDK and an EVGA GeForce GTX 260 graphics card with 896Mb of memory.

