HPCE Coursework 5 Summary

Henry Poulton (hgp10) and Richard Worrall (rew09)

NOTE: It was found at a very late stage that the specific use of <tr1/tuple> and <sys/time.h> in the current implementation are not windows compatible. The implementation should be fully Unix-compatible as is however.

Overview

Description of Files:

main.cpp – The main function checks that input arguments are sensible, then calls the most appropriate transform function, depending on the arguments passed to the function and the availability of OpenCL devices.

utilities.cpp – Contains auxiliary functions to handle stream I/O etc.

transforms.cpp – Contains the code to set up and run sequences of OpenCL kernels (defined in *pipeline_kernels.cl*) in order to perform open/close operations.

pipeline_kernels.cl – Contains the actual OpenCL kernels to perform individual erode or dilate operations for each possible pixel size (in terms of bits).

recursive_sse.cpp – Contains CPU-based code to perform erode/dilate operations, optimized using the SSE SIMD instructions.

Implementation Details

The 'algorithm' used in our optimized implementation of this application is to process the image input from stdin on a on a line-by-line basis, i.e. performing erode or dilate operations on whole image lines at a time. If an image line is taken as a whole, the only data dependencies for eroding or dilating this line are the image lines immediately preceding and immediately following this line. Therefore, this seemed the most natural way to split the incoming data to enable pipelining of image processing, and to enable the application to begin writing processed pixels to stdout before the whole image had been read in.

The application uses the x86 SSE2 extensions for CPU-based SIMD processing of input data by default. This is pipelined using recursive function calls for the sake of simplicity. Although the CPU may typically have lower throughput than a GPU, having to copy data between GPU memory and main memory is avoided, in theory reducing latency. The input data is repacked directly into 128-bit data structures for SIMD computation (i.e., multiple pixels within a single machine word are processed in parallel), both increasing throughput and reducing

latency. For example, the 8-bit SSE function processes 16 8-bit unsigned integers in parallel.

For very wide images, where the time taken to process each image line on the CPU is sufficiently high, the application switches to using OpenCL kernels to process the pixels in each image line in parallel, if an OpenCL-compatible device is available. The application times how long each available OpenCL-compatible device takes to process 100 dummy image lines, and uses the fastest for performing computations on actual image data.
Similar to the CPU-based computation, the OpenCL computations are fully pipelined, with multiple erode and dilate operations proceeding in parallel. Moreover, Intel Threaded Building Blocks are used to enable reading from stdin, image transformation, and writing to stdout to all occur in parallel in pipelined fashion. Circular buffers exist between each stage, each with separate read and write pointers, to enable simultaneous reading to and writing from each buffer, whether on the CPU or the GPU. Unlike the purely CPU-based transform functions, the OpenCL implementation does not use recursive function calls, but only for-loops and while-loops.

Like the CPU-based transform functions, the OpenCL implementation processes the input image data without unpacking it (except converting each 64-bit input into two 32-bit values), reducing and increasing the level of parallelism. A separate kernel exists for each line-wise erode and dilate for each valid bit depth.

To optimize for GPU (SIMD) computation, non of the code in the kernels is branching. Non-branching code is used to perform index bounds checking, for performing modular arithmetic for calculating offsets in the circular buffers, and for dealing with the special cases of pixels along the top, bottom, far left and far right which do not have 4 neighbours to compare pixel values against.

Performance Testing Methodology

Execution time was measured using the built in profiling tools in XCode 5 (“Instruments”). These tools enable the user to drill down into the execution tree and examine which function calls are taking the most time etc.

An attempt at measuring pixel latency was made using a separate application *LatencyTimer*. This application writes one byte at a time to stdout and adds a timestamp to a queue at the same time. This continues until bytes are available to be read from stdin, at which time the application tries to read as many bytes as possible, and for each byte compares the time difference between the time of reading and the timestamp taken from the front of the queue, effectively calculating the ‘round-trip time’ for this byte. Once no more bytes are available to read, the application begins to write to stdout once again, and so on.

Various different image sizes were tested when considering performance (sampled at various bits/pixel), from 512x512 to 15744x14700, as well as ‘ultra-wide’ reshaped variants of this large image, up to 4723200x49.

Correctness Verification Methodology

To verify correctness, we redirected stdin and stdout from/to files for both our optimized implementation and the reference implementation provided in the original coursework skeleton. We then performed binary difference checks on the output file generated by our application, compared to the output created by the reference application, for identical input files and parameters. (This is done using the *diff* command on Mac OS X, and the *fc* command on Microsoft Windows).

However, for diagnostic purposed, we wrote a short MATLAB function, *raw2mat.m*, that would import the two raw image files from the optimized and reference versions of the application as pixel matrices. These can then be compared within MATLAB to see *where* in the image discrepancies occur.

Summary of Division of Labour

High-level planning and design was performed together; it was decided that Henry would focus on exploring the use of Intel Threaded Building Blocks/SSE Instructions for application acceleration, while Richard would focus on OpenCL. Implementation then proceeded separately in parallel. TBB was found to have a very high overhead in terms of latency, so it is used sparingly in the final application.

Each person was separately responsible for the correctness of their implementation. An emphasis was placed on modularity to enable swift final assembly, which was also done together.