The Inverse Discrete Cosine Transform (IDCT):
A performance comparison on OpenCL and OpenMP

Artem Afanasyev (u5231713)

12 February 2016
Abstract

In this report the Inverse Discrete Cosine Transform algorithm was explored. It has a number of applications, specifically in digital signal processing and compression algorithms. As part of this project, the IDCT and one IFCT algorithms have been implemented using a simple CPU approach, OpenMP and OpenCL in order to compare performance. It has been found that the IFCT (Inverse Fast Cosine Transform) implemented in OpenCL for the GPU has the best performance in terms of execution speed. This was to be expected, given the known performance of non-graphical GPU routines. However, we came across a surprising result: OpenCL IDCT (Inverse Discrete Cosine Transform) has executed slower than the OpenMP (multi-threaded) algorithm for IDCT. In terms of accuracy, it has been shown that IDCT produces more accurate results in comparison to IFCT. It has also been confirmed that the accuracy of the particular algorithm (IDCT or IFCT) is independent of the implementation framework.
Acknowledgements

I would like to express my gratitude to Dr. Eric McCreath for guiding me through this project, and taking me on in the first place. I would also like to thank the Australian National University and the College of Engineering and Computer Science in particular, for giving me an opportunity to undertake this project.
Chapter 1

Introduction

This project aims to explore the Inverse Discrete Cosine Transform (IDCT). In this case, the IDCT's formula is applied to a two-dimensional 8x8 block. The IDCT algorithm is implemented on GPU and multicore systems, with performances on each system compared in terms of time taken to compute and accuracy.

DCT, or the Discrete Cosine Transform, has multiple applications when it comes to image and audio compression. When DCT is applied to a finite data sequence, it is represented in terms of a sum of cosine functions that oscillate at different frequencies. Discrete Cosine Transform has eight standard variants, four of which are most common. Particularly in image compression, the DCT-II is most commonly used. Its inverse, the DCT-III, or IDCT is used for decoding, rather than encoding. Exploring various IDCT algorithms is the aim of this project.

There are also faster ways to implement IDCT similar to Fast Fourier Transforms, one of which is also implemented on GPU and multicore systems.

In short, the Direct Cosine Transform transform the data sequence (input signal) from the time domain to frequency domain, like any other Fourier transforms, whereby the energy of the signal is compacted into low frequency bins [1]. DCT outputs a set of coefficients, each of which corresponds to a DCT basis function. These are cosine functions with increasing frequencies [1]. Each of these basis functions is then multiplied by the corresponding coefficients, and the sum of those values is the reconstruction of the original signal. This process is carried out with the use of the Inverse
Discrete Cosine Transform - IDCT[1].

For analysing the complexity of the algorithms we will use Big O notation. When describing complexity of the DCT, N represents the total number of elements in the matrix, in our case \( N = 64 \) elements.

Directly computing the DCT requires \( O(N^2) \) operations, however, it is also possible to achieve the same thing with \( O(N \log N) \) complexity. This is achieved by factorising the computation, similarly to the Fast Fourier Transform (FFT). Methods that utilise \( O(N \log N) \) complexity are known as Fast Cosine Transform (FCT) algorithms [2].

To summarise, the difference between DCT and FCT is the complexity \( O(N^2) \) for DCT and \( O(N \log N) \) for FCT.

The project implements the simple IDCT approach along with the faster FCT approach in OpenCL for the GPU and OpenMP for the multicore systems. The performance of different approaches is also discussed.

In summary, the following was carried out over the course of this project:

Implementing IDCT and IFCT algorithms using:

- **Simple CPU approach**
  Sequential IDCT and IFCT calculation

- **Multi-core CPU approach**
  Implementation of IDCT and IFCT using OpenMP framework (multi-threaded approach)

- **GPU approach**
  Implementation of IDCT and IFCT using OpenCL framework for GPU

The six of the above approaches are then compared based on the following:

- **Time taken to compute**
  All of the six algorithms have been computed for 1, 1000 and 10,000 blocks, and time taken to compute each one has been recorded.

- **Accuracy**
  Accuracy was measured by encoding with the most accurate approach of simple DCT, and then decoded using all of the IDCT implementations discussed above. The standard deviation between the input and reconstructed values is then computed.
Chapter 2

Background

2.1 Discrete Cosine Transform (DCT)

The discrete cosine transform (DCT) is a technique for converting a signal into elementary frequency components. It is widely used in image and sound compression [3]. Some of the formats DCT is used for include but are not limited to JPEG, MPEG audio and digital VCR [4]. From a mathematical standpoint, the Discrete Cosine Transforms allows to analyse complex signal in terms of separate frequency components in a way that is appropriate for compression.

DCT is used for lossy compression, which is based on the principle of removing undetectable components without altering the perceptible details. In practical cases, most of the signal information (or energy) is unevenly distributed and stored in a few of the low-frequency coefficients of the DCT, which implies that many DCT coefficients can be eliminated without much loss of information [6].

DCT is used for encoding, and the IDCT, being the inverse of the encoding transform, is used for decoding of information:

The two-dimensional 8x8 Inverse Discrete Cosine Transform is given by:

\[
G_{u,v} = \frac{1}{4} \alpha(u)\alpha(v) \sum_{x=0}^{7} \sum_{y=0}^{7} g_{x,y} \cos\left[\frac{(2x+1)u\pi}{16}\right] \cos\left[\frac{(2y+1)v\pi}{16}\right],
\]

where:
• $x$ is the pixel row ($0 \leq x < 8$)

• $y$ is the pixel column ($0 \leq y < 8$)

• $\alpha(u)$ is a normalising scale factor that makes the transform orthonormal, given by

$$\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } u = 0 \\ 1, & \text{otherwise} \end{cases} \quad (2.2)$$

• $g_{x,y}$ is the reconstructed pixel value at coordinates $(x, y)$

• $G_{u,v}$ is the reconstructed approximate coefficient at coordinates $(u, v)$

**History and Relationship to Karhunen-Loeve Transform**

DCT is used in digital signal processing due to its efficiency. It is real, separable and orthogonal, and approaches the statistically optimal KLT (Karhunen-Loeve transform). The KLT itself suffers from computational problems [7].

Karhunen-Loeve transform is a series representation of a given random function (e.g. signal sequence). Mathematical details of the transform are beyond the scope of this paper, its importance, however, lies within the fact that it completely decorrelates signal in the transform domain. It also minimises the mean square error between the signal representation and the actual signal [7].

The KLT (Karhunen-Loeve Transform) works by identifying certain statistical properties of the signal and then utilising those properties to construct an optimal decomposition. The KLT analysis is however extremely complicated, as it involves analysing the signal and constructing a transform based on the statistical parameters that cannot be otherwise predetermined. This is what makes the KLT an impractical tool when it comes to signal processing [4],[7].

KLT thus provides a benchmark against which other transforms may be judged. It has been shown that the DCT-II and DCT-III (or IDCT, which is the inverse of DCT-II) have the minimal variance distribution compared to
other non-KLT transforms, hence their extensive use in signal processing [7], [8].

As mentioned before, DCT is much simpler to compute, and it matches the KLT for common types of data. When the DCT algorithm is implemented, a single block of data is converted into a collection of DCT coefficients. Those coefficients represent the frequency components in frequency domain. The first coefficient (the DC coefficient) is simply the average of the entire block. Later coefficients (the AC coefficients) represent successively higher frequencies. For lossy graphics compression, “higher frequency” roughly corresponds to “finer detail”, and can be left out [4].

Use of Discrete Cosine Transform

Compression algorithms operate by breaking data into small blocks. DCT is then applied to each of the blocks, which is how the DCT coefficients are produced. These coefficients are multiplied by a predetermined fixed weight, where higher frequency components use smaller weights. This results in higher frequency components becoming negligible. After this, standard compression techniques, which are beyond the scope of this paper, are used in order to condense the coefficients into a smaller number of bits. This process is often iterative [4], [5].

IDCT comes into play when the data needs to be decompressed. Decompression works in reverse to compression. First, a series of weighted coefficients are obtained through decoding the bits. Then, each of those coefficients is divided by the corresponding weight. The IDCT is then applied to recover the final values [4], [5].

It is important to mention that the DCT and IDCT are not the main reason the compression algorithms that use these transforms are lossy. It is the weighting and inverse weighting that round off higher frequency components [4].

2.2 Fast Cosine Transform (FCT)

The use of DCT has not been as extensive as one would imagine, despite its properties (real, separable and orthogonal) due to lack of an efficient algorithm. However, the DCT can be optimised and its complexity reduced for the ease of computation. Complexity can be reduced through row-column decomposition, where two-dimensional DCT is constructed by executing a one-dimensional DCT over each row and then each column. The 1D DCT
is:

\[ F_u = \frac{1}{2} C_u \sum_{x=0}^{N-1} f_x \cos(\frac{u\pi}{2N} + \frac{1}{2N}), \]  

(2.3)

where:

- \( x \) is the coordinate in this now one-dimensional vector
- \( C_u \) is some coefficient
- \( F_u \) is the reconstructed approximate coefficient at coordinate \((u)\)
- \( f_x \) is the reconstructed pixel value at coordinate \((x)\)

The complexity of the above equation is \( O(N^2) \), and running it \( 2n \) times results in 2D DCT with complexity of \( O(N^3) \).

The \( O(N^2) \) DCT algorithm can be replaced yet again with an algorithm that is factored similarly to a Fast Fourier Transform (FFT), whereby the complexity is reduced to \( O(n \log n) \). The AAN (Arai, Agui, Nakajima - named after the authors) algorithm is one of the fastest known one-dimensional DCTs [9]. This algorithm has been selected and implemented over the course of this project.

The straightforward DCT approach takes 64 * 64 * 2 multiplication for the full DCT (which applies for the IDCT as well). In contrast, the AAN DCT (of FCT) uses only 5 multiplications and 8 postmultiplications. These post-multiplications all involve multiplying by powers of two, which further speeds up the algorithm by using bit-shifting [4].

### 2.3 Pseudo code

This section provides pseudo code for the DCT algorithm and the FCT (faster) algorithm.

#### 2.3.1 DCT

```c
float output[64];

for (int y = 0; y < 8; y++) {
  for (int x = 0; x < 8; x++) {
    float result = .0;
    for (int v = 0; v < 8; v++) {
```
for (int u = 0; u < 8; u++) {
    float alpha_u = u == 0 ? s2 : 1.0;
    float alpha_v = v == 0 ? s2 : 1.0;
    result += alpha_u * alpha_v * input[v * 8 + u] * cos(((2 * x + 1) * u * PI) / 16) * cos(((2 * y + 1) * v * PI) / 16);
}
output[y * 8 + x] = result / 4;
}

2.3.2 FCT

The following flowgraph shows the order of computational operations in 1D AAN DCT algorithm. To obtain fast IDCT the graph needs to be reversed (i.e. read from right to left).

![Flowgraph of Arai, Agui, and Nakajima DCT algorithm](image)

Figure 2.1: Arai, Agui, and Nakajima DCT algorithm. Boxes represent multiplication. a1=0.707, a2=0.541, a3=0.707, a4=1.307, a5=0.383 [10].

The idea behind the faster FCT approach is to separate the two-dimensional transformation into two one-dimensional transformations. The pseudo code for this approach is shown below, and the full FCT algorithm code can be found in Chapter 7 (Appendix).

```c
for (int i = 0; i < 8; i++) {
    // transform rows
}
for (int i = 0; i < 8; i++) {
    // transform columns
}
```
2.4 Findings

In terms of time taken to compute the algorithms and their accuracy, the following has been found:

Time

OpenCL IFCT (GPU) was found to be the fastest, and the single-threaded sequential approach took the longest. There was also one unexpected result: OpenMP DCT has performed better time-wise than the OpenCL DCT. In terms of efficiency, OpenMP IDCT has proven to operate at the highest percentage of the theoretical hardware limit. See results in Table 5.3.

Accuracy

It has been found that IDCT is more accurate than IFCT, as expected. It has also been shown that this result is independent of the framework used, as CPU, OpenMP and OpenCL have produced the same results for IDCT and then IFCT. The results are discussed in more detail in Chapter 5.
Chapter 3

OpenCL & OpenMP

This chapter provides a background to OpenCL and OpenMP, giving an overview of the programming model used in these languages. This section will also provide a history of their development.

3.1 OpenCL

GPU, or Graphics Processing Units had not been used for non-graphical routines before 2010, and the idea was considered a novelty in the field of high performance scientific computing. The first supercomputer to ever use general purpose GPU computing (GPGPU computing) was the Chinese Tianhe-1A, which quickly made it the fastest supercomputer in the world in 2010. Today, engineers and scientists agree that CPU/GPU systems are the future of supercomputing, as it has been proven that using GPUs ensures fast performance of the system as well as power efficiency [11].

With the increasing popularity of these heterogeneous processing platforms, the demand has appeared for a hybrid programming language that would target both CPU and GPU. OpenCL (Open Computing Language) is a language for programming hybrid systems that consist of multiple types of processors. It is an open royalty-free standard for parallel programming across CPUs, GPUs and other processors, which enables software developers to utilise the full power of the hybrid processing platforms [12].

In 2008, OpenCL Working Group was formed as part of the Khronos Group, which is a group of companies that seek to advance graphics and graphical media [11].
3.1.1 Advantages of OpenCL

OpenCL is not a programming language in itself, rather it is a standard that characterises a collection of data types, data structures and functions that enhance C and C++ [11]. The three main advantages that set OpenCL apart are portability, standardised vector processing and parallel programming. These advantages are discussed in more detail below:

**Portability**
With OpenCL there is no need for developers to learn vendor-specific language to program certain types of hardware. Rather, all OpenCL-compliant devices are able to compile OpenCL code. Moreover, OpenCL is not dependent on the architecture of the device, so OpenCL routines can target multiple devices [11].

**Standardised Vector Processing**
Standardised vector processing is one of the main advantages of OpenCL. This refers to the definition of computational vector, which is a data structure consisting of multiple elements of the same data type. In computational vectors, each component is operated upon in the same clock cycle [11]. High-performance processors (superscalar and vector processors) operate on multiple values at once. Again, with OpenCL, vector programs can be run on any OpenCL-compliant processor.

**Parallel Programming**
OpenCL facilitates parallel programming, whereby computational operations are assigned to multiple processing elements that are performed at the same time. These tasks are called kernels. OpenCL enables full task-parallelism, which is a form of parallelism that where tasks are assigned to processes or threads; and each processor executes a different thread. This is an advantage OpenCL has over some other parallel programming tools, which only facilitate data-parallelism, where each processor executes the same instructions on different types of data [11].

In summary, OpenCL has clear advantages over regular C and C++. However, it is also more complex. In real-world OpenCL applications, multiples data structures need to be created and their operation coordinated.

Over the course of this project, OpenCL was used in order to implement DCT and FCT algorithms on GPU.
3.1.2 Example of Code for OpenCL

```c
kernel void square(global float* input, global float* output) {
    int id = get_global_id(0);
    output[id] = input[id] * input[id];
}
```

To be able to use this kernel the host program should do the following:
1. Build the kernel from the source;
2. Create memory buffers and map memory;
3. Set kernel arguments;
4. Set up the number of work units per kernel 5. Execute the kernel;
6. Read the memory.

3.2 OpenMP

OpenMP is an API that enables shared memory multiprocessing programming in C, C++ and Fortran. In other words, OpenMP API is a portable, scalable model that provides developers with a simple user-friendly interface for working with parallel applications on a wide range of platforms, from embedded systems to multicore and shared-memory systems [13], [15]. OpenMP was defined by computer hard- and software heavyweights. The OpenMP ARB (Architecture Review Board) is the entity that owns the OpenMP brand and oversees, produces and approves the specification [15].

OpenMP is an implementation of a method of parallelising, whereby a series of instructions are executed sequentially (the master thread), which then spawn a predefined number of slave threads, and the system then divides the task among them. The threads are run concurrently, and are allocated to different processors [16].

3.2.1 Advantages of OpenMP

The OpenMP Application Programming Interface was intended as a user-friendly, flexible, easy to learn and apply tool for programmers to develop memory-shared parallel applications. It was designed to allow a step-by-step approach to parallelising existing programs, whereby only parts of the code are parallelised. This idea contrasts some of the other parallel programming tools, where a single step
conversion of the entire program from sequential to parallel is typically required [13].

Another argument in favour of OpenMP is that it enables developers to only work with a single source code. In other words, a single program can be sequential and parallel at the same time. A set of OpenMP directives is used to tell the compiler which instructions are to be executed in parallel, and how they are to be distributed among the threads. The OpenMP directives are special-format instructions that are only recognised by OpenMP compilers. To a C/C++ compiler those instruction appear like pragmas, and as regular comments to a Fortran compiler. This enables the program to run sequentially on a non-OpenMP compiler, and change to a parallel program on an OpenMP-compatible ones [13].

For this project, OpenMP was used to implement DCT and FCT algorithms, in order to utilise multiple CPU cores in parallel.

### 3.2.2 Example of Code for OpenMP

This program creates a number of threads and run a simple “hello world” program from each of these threads.

```cpp
#include <iostream>
#include <omp.h>

using namespace std;

int main() {
    int id;

    #pragma omp parallel private(id)
    {
        id = omp_get_thread_num();
        #pragma omp critical
        cout << "Hello from thread." << id << "!" << endl;
    }
}
```

[14]
Chapter 4

Implementation of IDCT

A description of the implementation of IDCT in OpenCL and OpenMP. This will include an overview of the development and testing framework.

4.1 IDCT and IFCT Development

4.1.1 Simple

The simple IDCT approach implementation simply follows the straightforward approach described in pseudo code in Chapter 2. To avoid repeating calculations and to speed up the running time the cosine table was precalculated and stored in memory. The same principle was applied to the FCT calculation.

4.1.2 OpenMP

For the OpenMP two different groups of approaches were tested. The idea behind the first approach was to spread the blocks between threads, so 1 thread fully decodes a single block. The second approach aims allocated different parts of the algorithm itself to different threads. For the IFCT approach, matrix rows and columns were attempted to be divided into different threads. This approach proved to be inefficient since the single block consists of only 8 rows and 8 columns and the overhead of thread creating negates the speed effects of parallel execution.

For the both algorithms (IDCT and IFCT) the first approach proved to be faster while the second approach only extended the execution time.
Hence, the first approach was used in the final implementation and in the tests described in the next chapters.

4.1.3 OpenCL

To be able to implement the IDCT on a GPU two the code was written in two parts. The first part is the kernel code which is to be executed by the GPU. The second part is the host code, which compiles and builds the kernel, allocates the memory buffer and starts the kernel execution.

The blocks to be decoded were merged into a single block of memory and mapped onto the GPU memory. The number of kernel work groups were set to be equal to the number of blocks to be decoded.

For the IDCT approach the kernel accepted 3 arguments: the input matrix, the pre-calculated cosine table and the output matrix. For the IFCT approach, the arguments were only the input and output matrices.

After the execution of the kernel, the memory containing the output matrices was mapped back onto the CPU memory and split back into separate blocks.

The code for both IDCT and IFCT kernels can be found in Chapter 7 (Appendix).

4.2 Testing Framework

To test the performance of the algorithms described above two main metrics were measured: speed of the execution and accuracy of the calculations.

The 8x8 blocks to be encoded were represented by a block class which contained a 1-dimensional array of 64 floating point numbers and a range of methods to access and manipulate this data.

The tests were performed on randomly generated sets of blocks containing numbers ranging from 0 to 255 which simulated RGB values.

A number of functions were established in order to measure the execution speed. The blocks generation time and any other non-framework related overhead was excluded from the calculations.
Chapter 5

Performance

This chapter describes the hardware used for evaluation, what experiments were done to evaluate the overall performance of different algorithms, and the results obtained.

The following hardware was used:

<table>
<thead>
<tr>
<th>Table 5.1: Hardware used</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td>Processor Name: Intel Core i5</td>
</tr>
<tr>
<td>Core Speed: 2.6 GHz, up to 83.2 GFLOPS</td>
</tr>
<tr>
<td>Total Number of Cores: 4</td>
</tr>
<tr>
<td>L2 Cache (per Core): 256 KB</td>
</tr>
<tr>
<td>L3 Cache: 3 MB</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
</tr>
<tr>
<td>Model: Intel Iris 5100</td>
</tr>
<tr>
<td>Core Speed: 1.3GHz, up to 832 GFLOPS</td>
</tr>
<tr>
<td>Peak theoretical bandwidth: 25.6 GB/s</td>
</tr>
<tr>
<td>Pipelines/Execution Units: 40</td>
</tr>
</tbody>
</table>

5.1 Speed

To test the speed, the algorithm was run for three different numbers of blocks (1, 1000 and 10,000 blocks). The purpose of running this algorithm on one block is to represent the set up time overhead for different approaches. It appears that OpenCL has the highest set up time. To obtain this data, the algorithms were run 1000 times on a single block and then the average of obtained values was taken. The difference between 1000 and 10,000 block demonstrates the proportion of frameworks (OpenCL and OpenMP) overhead time to the actual run time.
Figure 5.1: Testing of IDCT speed for different approaches; blocks decoded 1; note that the time is represented in $\mu$s to better graphically represent the difference between bars.

The graphs for 1000 and 10,000 blocks are presented for the purpose of comparing the speed of different implementation and determining whether there are any anomalies.
Figure 5.2: Testing of IDCT speed for different approaches; blocks decoded 1000; time in ms

Figure 5.3: Testing of IDCT speed for different approaches; blocks decoded 10,000; time in ms

It is expected that OpenCL would take the lowest time to compute, for both DCT and FCT. However, in case of the DCT, OpenCL takes longer than OpenMP. Further analysis of the data suggests that the abnormal OpenCL
DCT performance (see Figure 5.1 and Figure 5.1) is caused by an implementation flaw. In case of the OpenCL FCT implementation only the input matrix is passed to the kernel. However, with the DCT approach precalculated cosine table is also passed to the kernel, which could possibly double the amount of the memory to be mapped to the GPU.

In order to estimate the computation speed as a fraction of theoretical hardware limits, the GFLOPS were calculated using the generated data, and then compared to the processor capabilities (refer to Table 5.3).

<table>
<thead>
<tr>
<th>Approach</th>
<th>Calculated Speed (GFLOPS)</th>
<th>Theoretical Limit (GFLOPS)</th>
<th>% of theoretical hardware limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple IDCT</td>
<td>1.95</td>
<td>83.2</td>
<td>2.3%</td>
</tr>
<tr>
<td>OpenMP</td>
<td>4.55</td>
<td>83.2</td>
<td>5.5%</td>
</tr>
<tr>
<td>OpenCL DCT</td>
<td>2.82</td>
<td>832</td>
<td>0.3%</td>
</tr>
<tr>
<td>Simple IFCT</td>
<td>6.83</td>
<td>83.2</td>
<td>8.2%</td>
</tr>
<tr>
<td>OpenMP IFCT</td>
<td>11.7</td>
<td>83.2</td>
<td>14.1%</td>
</tr>
<tr>
<td>OpenCL IFCT</td>
<td>20.48</td>
<td>832</td>
<td>2.5%</td>
</tr>
</tbody>
</table>

Despite the GPU being very powerful (832 GFLOPS), we only get limited performance. This is mainly due to the bottlenecks in performance (memory bandwidth, memory access patterns), but also due to implementation imperfections. All of the above numbers are below the theoretical limits, and further investigation is required.

### 5.2 Accuracy

For the purpose of accuracy testing the following was carried out: An 8x8 block M is encoded using the most accurate approach of simple DCT(M) and then decoded using also the simple approach for the IDCT(M). The IDCT, being the inverse of the DCT, should produce the same result. In practice, however, the components of the resulting matrix are not exactly the same as the components of the starting matrix. The corresponding values of the two matrices are then compared via the standard deviation method as follows:

- Step 1: Calculating the difference between corresponding compo-
nents of the two matrices

- Step 2: Calculating the square of the difference from Step 1
- Step 3: Calculating the sum of all values from Step 2*  
  *note that for this step the matrix $M$ is represented as a one-dimensional vector
- Step 4: Computing the square root of the result from Step 3, divided by 64 (number of components in an 8x8 matrix)

The formula for computing the standard deviation is:

$$STDEV = \sqrt{\sum (M_i - M'_i) / 64},$$  \hspace{1cm} (5.1)

where $M_i$ is the initial matrix to be encoded and $M'_i$ is the final decoded matrix [17].

Note that, for the purposes of accuracy testing, the matrices are treated as one-dimensional vectors, hence they only have one index $i$.

Decoding was done on matrices with randomised RGB (8-bit) values ranging from 0 to 255.

Table 5.3: Accuracy

<table>
<thead>
<tr>
<th></th>
<th>IDCT</th>
<th>IFCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>1.37789e-5</td>
<td>0.515388</td>
</tr>
<tr>
<td>OpenMP</td>
<td>1.37789e-5</td>
<td>0.515388</td>
</tr>
<tr>
<td>OpenCL</td>
<td>1.37789e-5</td>
<td>0.515388</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusion

To conclude, the algorithms of IDCT and IFCT were explored over the course of this project. Both of those algorithms were implemented using the simple CPU approach, OpenMP for the multicore approach and OpenCL for the GPU approach. The results were then tested based on their accuracy and time taken to compute, which was also compared to the maximum theoretical speed the hardware could operate at.

To compare the computation speed, the six implementations of IDCT were run on 1, 1000 and 10,000 blocks. The result obtained for 1-block operation represents the set up overhead time. It was concluded that the sequential algorithm for IDCT took the longest, and the OpenCL IFCT was the fastest. These results were as expected, based on know performance superiority of the GPU. The OpenCL algorithm for the IDCT, however, took longer to compute than the OpenMP IDCT. We speculate that this is perhaps due to an implementation flaw, which is to be investigated in future projects.

Accuracy of IDCT and IFCT was also evaluated. Not surprisingly, it has been confirmed that IDCT is a lot more accurate compared to the IFCT. It was also shown that the accuracy of the algorithms is independent of the development framework.

The following work could be analysed to potentially obtain better results:

- Memory access patterns
- Thread scheduling for the OpenMP approach
- Cache lines performance
Chapter 7

Appendix

The following is the code for FCT implementation

```c
int row[9], col[9];
int rows[8][8];

const int c1 = 251;
const int s1 = 50;
const int c3 = 213;
const int s3 = 142;
const int r2c6 = 277;
const int r2s6 = 669;
const int r2 = 181;

// rows
for (int i = 0; i < 8; i++) {
    // stage iv
    row[0] = (int)(*b)(0, i) << 9;
    row[1] = (int)(*b)(1, i) << 7;
    row[2] = (*b)(2, i);
    row[3] = (*b)(3, i) * r2;
    row[4] = (int)(*b)(4, i) << 9;
    row[5] = (*b)(5, i) * r2;
    row[6] = (*b)(6, i);
    row[7] = (int)(*b)(7, i) << 7;
    row[1] = row[7];

    // stage iii
    row[7] = row[0] + row[4];
    row[0] = row[4];
}
```
row[1] -= row[5];
row[8] -= row[3];

// stage ii
row[2] = row[0] + row[6];
row[0] -= row[6];

// stage i
row[7] += 512;
row[2] += 512;
row[0] += 512;
row[3] += 512;
rows[i][0] = (row[3] + row[4]) >> 10;
rows[i][1] = (row[2] + row[8]) >> 10;
rows[i][2] = (row[0] + row[1]) >> 10;
rows[i][3] = (row[7] + row[5]) >> 10;
rows[i][5] = (row[0] - row[1]) >> 10;
}

// cols
for (int i = 0; i < 8; i++) {

  // stage iv
  col[0] = rows[0][i] << 9;
  col[1] = rows[1][i] << 7;
  col[2] = rows[2][i];
  col[3] = rows[3][i] * r2;
  col[5] = rows[5][i] * r2;
  col[6] = rows[6][i];
  col[7] = rows[7][i] << 7;
  col[1] -= col[7];
/ stage iii
  col[7] = col[0] + col[4];
  col[0] -= col[4];
  col[1] -= col[5];
  col[8] -= col[3];

// stage ii
  col[2] = col[0] + col[6];
  col[0] -= col[6];
  col[4] >>= 6;
  col[5] >>= 6;
  col[1] >>= 6;
  col[8] >>= 6;

// stage i
  col[7] += 1024;
  col[2] += 1024;
  col[0] += 1024;
  col[3] += 1024;
  res(i, 0) = (col[3] + col[4]) >> 11;
  res(i, 1) = (col[2] + col[8]) >> 11;
  res(i, 2) = (col[0] + col[1]) >> 11;
  res(i, 3) = (col[7] + col[5]) >> 11;
  res(i, 4) = (col[7] - col[5]) >> 11;
  res(i, 5) = (col[0] - col[1]) >> 11;
  res(i, 6) = (col[2] - col[8]) >> 11;

OpenCL kernel

kernel void idct(global float* input, global const float*cos_table, 
global float* output) {
  int id = get_global_id(0);
for (int y = 0; y < 8; y++) {
    for (int x = 0; x < 8; x++) {
        float result = 0.0;
        for (int v = 0; v < 8; v++) {
            for (int u = 0; u < 8; u++) {
                float alpha_u = u == 0 ? M_SQRT1_2_F : 1.0;
                float alpha_v = v == 0 ? M_SQRT1_2_F : 1.0;
                result += alpha_u * alpha_v
                    * input[id * 64 + v * 8 + u]
                    * cos_table[x * 8 + u]
                    * cos_table[y * 8 + v];
            }
        }
        output[id * 64 + y * 8 + x] = result / 4;
    }
}
Bibliography


INDEPENDENT STUDY CONTRACT

Note: Enrolment is subject to approval by the projects co-ordinator

SECTION A (Students and Supervisors)

UniiID: ______u5231713____________

SURNAME: ______Afanasyev________ FIRST NAMES: ___________Artem________________

PROJECT SUPERVISOR (may be external): ___Dr__Eric McCreath________________

COURSE SUPERVISOR (a RSCS academic):

COURSE CODE, TITLE AND UNIT: ___COMP6470___Special Topics in Computing__________

SEMESTER  Spring  Session YEAR: ___2015/2016________

PROJECT TITLE:

The Inverse Discrete Cosine Transform (IDCT) – a performance comparison on OpenCL and OpenMP

LEARNING OBJECTIVES:

This project aims to cover the learning outcomes of Parallel Systems (COMP4300) which are:

• Be proficient at programming multiple parallel machines in more than one special programming language or programming system [1]
• Be able to descriptively compare the performance of different programs and methods on one machine [2]
• Demonstrate advanced knowledge of the elements of parallel programming language and system implementation [3]
• Recall the history of parallel systems and describe the developments in the field [4]

These have been numbered and the numbers are referred to in the description.

The project will also have learning objectives that relate to doing an individual project such as implementation of a artifact, doing a literature survey, writing a report, and doing a seminar.

Research School of Computer Science

Form updated Jun-12
PROJECT DESCRIPTION:

The project will explore the Inverse Discrete Cosine Transform (IDCT) (2D on 8x8 blocks – which are used in JPEG and video formats) and a comparison of performance on GPU and multi-core systems. This would use OpenCL for the GPU and OpenMP for the multi-core system. This will involve gaining a good understanding of programming these parallel machines in these specialized programming languages. (Covering [1] of the learning outcomes).

There is a simple implementation of the IDCT based on the definition, however, there are also “faster ones” based on FFT approaches. The project would implement both the simple approach along with one of these faster approaches both in OpenCL and OpenMP. These implementations would be compared and the performance of the different approaches would be discussed. This would evaluate the bottlenecks in the performance and analysis the relative performance gains (or losses) between GPU and multi-core (this will cover [2] of the learning outcomes). The performance bottlenecks will be evaluated for these approach, in particular the analyses of thread scheduling and memory access patterns will be explored to see how these limit and shape the final performance (this will cover [3] of the learning outcomes).

The project report will contain:
+ An introduction to the topic.
+ A background section which describes the IDCT which includes the pseudo code for the basic approach and a faster approach.
+ A section with provides a background to OpenCL and OpenMP, giving an overview of the programming model used in these languages. This section will also provide a history of their development. (this will cover [4] of the learning outcomes)
+ A description of the implementation of IDCT in OpenCL and OpenMP. This will include an overview of the development and testing framework.
+ Experimental chapter which: describes the hardware used for evaluation, the experiments done, and the results tabulated/graphed.
+ Conclusion/discussion/limitations/future work chapter.
ASSESSMENT (as per course’s project rules web page, with the differences noted below):

<table>
<thead>
<tr>
<th>Assessed project components:</th>
<th>% of mark</th>
<th>Due date</th>
<th>Evaluated by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report: name style:</td>
<td>50%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e.g. research report, software description...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artefact: name kind:</td>
<td>40%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e.g. software, user interface, robot...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation:</td>
<td>10%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MEETING DATES (IF KNOWN):

Meeting weekly.

STUDENT DECLARATION: I agree to fulfil the above defined contract:

.................................................................  21/12/2015
Signature                                      Date

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student’s academic record and believe this student can complete the project.

.................................................................  21/12/2015
Signature                                      Date

REQUIRED DEPARTMENT RESOURCES:

Research School of Computer Science

Form updated Jun-12
SECTION C (Course coordinator approval)

................................................................. .................................................................
Signature Date

SECTION D (Projects coordinator approval)

................................................................. .................................................................
Signature Date