Utilizing GPUs on Cluster Computers

Project work in TDT4715 Algorithm construction and visualization, depth study

Fall 2006

Leif Christian Larsen
Main supervisor: Dr. Anne Cathrine Elster
Co-supervisor: Tore Fevang, Schlumberger Ltd.

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Abstract

Preface

This is the report for the project work done in the depth study course TDT4715 Algorithm construction, visualization and computational science at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU), Norway. The work was done over a period of four months, and was assigned by Schlumberger Limited. The main supervisor for the project was Associate Professor Dr. Anne Cathrine Elster of IDI-NTNU. The project was co-supervized by Tore Fevang of Schlumberger Limited, Trondheim.

I am very grateful for the excellent support I have received from my supervisors during the project. I would like to thank Dr. Elster for her excellent support, for hooking me up with Schlumberger and this project assignment, and for always taking the time to talk with me *even* when she was busy preparing a 50 MNOK supercomputer.

A very special thanks goes to my co-supervisor Tore Fevang of Schlumberger Trondheim, who has been supportive and helpful at a level which far exceeded everything I could have hoped for. Mr. Fevang has been *very* helpful throughout the entire project, *always* had time for questions, and *always* tried very hard answering all my questions.

I would also like to thank Schlumberger Limited and the manager at Schlumberger Trondheim, Wolfgang Hochweller, for providing me with a great place to work and giving me access to very high-end hardware during the project. I would like to emphasize that *without Schlumberger's* ***very significant*** *support, the quality of this project work would be significantly degraded.*

I also want to thank everyone else at Schlumberger Trondheim for providing an excellent atmosphere to work in (even though I unfortunately did not manage to beat anyone at the table football game).

Trondheim, December 21st, 2006

Leif Christian Larsen

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

A *Graphics Processing Unit* or *GPU* is a processor dedicated to manipulating and rendering computer graphics. Since about 1995, GPUs capable of advanced 3D graphics processing have been developed, mainly driven by the ever-increasing requirements of computer games. Games are getting increasingly complex, and require large amounts of graphics processing to display advanced 3D scenes at high resolutions. Without the use of a GPU, such games would be practically impossible to make, since the GPU is up to several hundred times quicker than the *central processing unit* (*CPU*) for these compuations due to its specialized and highly parallel design. Furthermore, since such advanced GPUs are fitted with practically every new personal computer sold today, economies of scale make sure that they are also very cheap. Accelerated 3D graphics processing systems were available prior to the introduction of the GPU, but such systems were extremely expensive and not made for the mass market.

In recent years, GPUs have evolved from being static processors, only capable of doing limited 3D graphics rendering, to increasingly programmable processors. This evolution started becoming a revolution in 2002, when the market-leading GPU vendor NVIDIA Corporation launched a new, specialized C-like programming language, *Cg* (an acronym for *C* for *g*raphics), for programming the GPU. Prior to Cg, developers had to program GPUs through low-level assembly language. Since then, other, similar languages have been released by other vendors. While Cg and other GPU programming languages (called *shader languages*) were (and still are) primarily designed for offloading advanced graphics calculations previously done by the CPU to the GPU, the low cost, high performance and increasing programmability of GPUs made it attractive to attempt to use the GPU for other computationally intensive tasks unrelated to graphics processing. Since computational capability is particularly important in computer clusters, which typically work on computationally intensive tasks, it is interesting to look at how GPUs can be utilized in applications running in clustered environments.

Recently, several tasks which traditionally have been executed on the CPU, such as protein structure prediction [1], linear algebra [2, 3], financial algorithms [4], flow simulation [5], fast Fourier transforms (FFTs) [6] and even sorting [7], have been implemented to run on the GPU and speedups of above 10 times over using the CPU have in some cases been obtained. Thus, the GPU is and will likely continue to be increasingly used as a general parallel vector processor.

1.1 Project Goals

This project evaluates how Graphics Processing Units (GPUs) may be utilized to offload computations in clustered environments. The emphasis will be on data compression of large datasets for use in client-server applications.

Data compression of large datasets is important in several applications running on a cluster. For example, in seismological visualization applications, where a client displays data to the user while a clustered server executes computations on seismological data and transmits results to the client, compressing the image data before transmission (both from the server to the client and between the cluster nodes) can increase application performance and responsiveness. Reducing the transmission time is particularly important due to the trend of building clusters using very cheap (and relatively high latency) Gigabit Ethernet links.

However, data compression requires significant computational resources. Data compression algorithms must detect and remove redundancy, which is a computationally intensive task. But since clusters today are often built using regular computers, which today almost always have GPUs (which love to do jobs requiring large amounts of computations), it is a good idea to outsource at least parts of the data compression process to the GPU, particularly since the GPU is usually not utilized at all in such clusters. Using the GPU for data compression can be a very good way to do just that.

1.2 Report Outline

This report is structured in the following way:

- **Chapter 2** gives detailed background information on GPUs. In this chapter, we will describe GPUs in terms of how they operate internally, how they are programmed, what tasks they are suited for and their price/performance characteristics, and compare them with CPUs. This chapter motivates why one should attempt to run certain computationally intensive tasks on GPUs instead of CPUs.

- **Chapter 3** gives an introduction to data compression methods, and describes in detail two mathematical transforms used in compression — the discrete cosine transform and the lapped orthogonal transform — that have been implemented both on the GPU and single- and multi-core CPUs in this project work. We also consider algorithms for implementing the transforms.

- **Chapter 4** describes how the transforms of Chapter 3 have been implemented in this project.

- **Chapter 5** compares the GPU implementations of the transform algorithms to the single and multi-core CPU versions, and discusses the results.

- **Chapter 6** summarizes the findings in the report, and attempts to answer our main question: *Under which, if any, circumstances should GPUs be used for data compression?*

Chapter 2

Background

The goal of this chapter is to give an understanding of how GPUs work, how they are programmed, what they are good and bad at, and how they compare to CPUs.

In Section 2.1, we give a brief history of how GPUs have evolved. Then we describe the structure and internals of current GPUs in Section 2.2, which is important in order to understand how they can be programmed. Next, we look at programming the GPU in Section 2.3. Then, we look at the price and performance characteristics of some of the currently available GPUs compared to current CPUs in Section 2.4. Finally, which tasks GPUs are suitable or unsuitable for are discussed in Section 2.5.

2.1 GPUs: A Brief History

We now give a brief history of how GPUs have developed, to gain an understanding of how they have evolved to where they are at today. For further information, see for example [8, 9].

In the late 1970s, and the beginning of the 1980s, most calculations concerning graphics drawing were done in software by the CPU. The first accelerated graphics operation came with the introduction of a special *bit block transfer* or *bitblt* instruction on the Xerox Alto computer. This operation copies rectangular arrays of bitmap data from a source to a potentially overlapping destination. This can, for instance, be used for animation of one image on top of a background image.

Through the 1980s, special graphics chips with an increasing number of features were developed. A revolution came with the introduction of the Commodore Amiga machine, which incorporated the first mass-market video accelerator able to draw and fill shapes and draw animations (*sprites*) in hardware. The Amiga's graphics subsystem comprised several chips. For instance, it had one chip dedicated exclusively to bit block transfers.

2D acceleration was further developed during the first half of the 1990s, when improvements in component manufacturing led to vendors integrating the features of several chips into single chips. Also, features to accelerate video playback were added. But in about 1995, 3D games were becoming increasingly popular. This led to the release of dedicated chips — 3D GPUs — which were able to do 3D calculations. Later, these chips were integrated with the 2D GPUs. 3D GPUs gradually went from only doing 3D *rasterization* (converting simple three-dimensional geometric primitives, such as lines, triangles and rectangles, to two-dimensional screen pixels) and *texture mapping* (mapping a two-dimensional texture image on to a planar three-dimensional surface) from 1995–1998 to supporting more advanced calculations like 3D translation, rotation and scaling, lighting and smoothing (1999–2000).

This is, however, not enough to produce all the effects a developer of a modern game wants to implement. Although GPUs were becoming increasingly configurable towards year 2000, they were not programmable. For example, there was no way to modify how the GPU calculated the lighting effects *individually* for each pixel drawn. This is a requirement in order to produce highly realistic 3D scenes. Developers wanting to create games employing such effects were therefore forced to compute the lighting effects for each pixel on the CPU, which is not a very good idea, such the CPU typically is only able to compute the lighting of one or a few simultaneously, and has traditionally been much better at integer rather than floating-point operations. Since the GPUs were designed from the ground to process hundreds of pixels simultaneously and were optimized for floating-point computation, it was soon evident that using the GPU for such computations was the *only* way realistic 3D scenes could be produced with acceptable frame rates. This lead to GPUs becoming increasingly *programmable*, and such GPUs were introduced from 2001 and onwards.

In recent years, GPU programmability has gradually increased. GPUs can now execute a small program which is run for each *vertex* passed to it (a *vertex program*), and a separate program for each pixel the GPU decides to draw (a *fragment program*). These programs enable developers to create all types of advanced effects. For instance, the complicated calculations of determining how smoke and fire moves in a 3D scene can be offloaded to the GPU using such programs [10]. And, of course, the programmability of the GPU is increasingly being used for applications unrelated to graphics, examples of which were mentioned in the previous chapter.

Initially, GPUs could only be programmed in low-level assembly language and the vertex and fragment programs were very limited in what operations they could execute. Today, high-level languages for GPU programming have been introduced, and with every new GPU generation, more programming limitations are removed. In the rest of this chapter, we take a look at how modern, programmable GPUs are structured and how they can be programmed.

2.2 The Structure of a modern GPU

Modern GPUs are structured around a *graphics pipeline*, which is a sequence of several steps operating in parallel and in a particular order to transform a set of geometric primitives and operations to 2D projection of a 3D image appearing on the screen. We begin this section by describing the graphics pipeline and how it is implemented on the GPU. Then, we take a deeper look at the GPU's vertex and fragment processors, which are the two currently programmable units within the GPU.

The following description is based on [11, 12]. Since we focus on the NVIDIA GeForce 6 and 7 series, other GPUs might be structured in a slightly different manner.

2.2.1 The Graphics Pipeline

The graphics pipeline of the GPU is structured to transform a *stream* of vertices and commands from a software application to output an image which is displayed on screen. A (simplified!) block diagram of a typical modern GPU, adapted from [11] and [12], is shown in Figure 2.1 on the facing page.

Initially, the host CPU sends commands (such as setting up where the camera in the scene is located, setting up colors and lighting, setting up whether squares, triangles or other figures

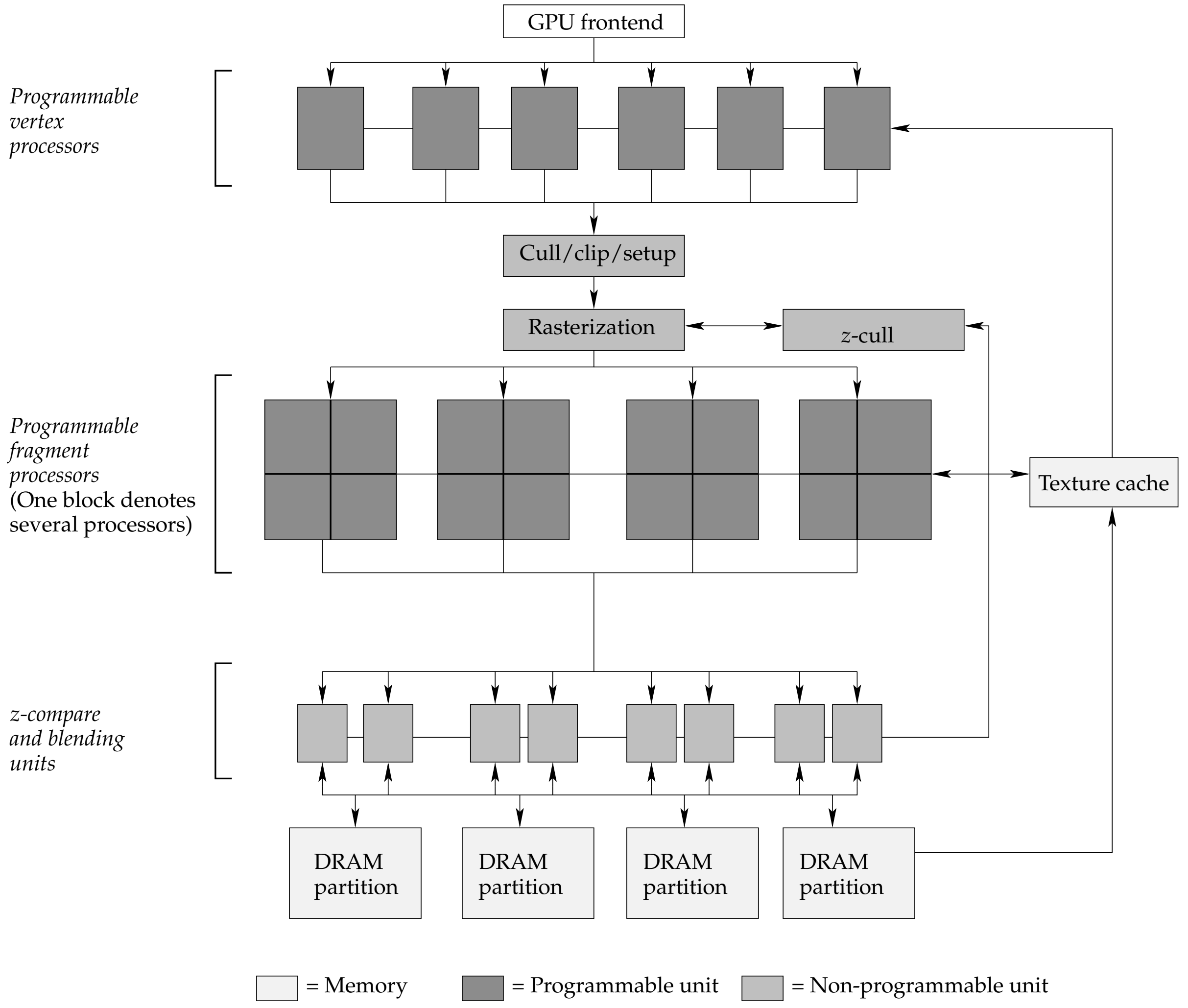

Figure 2.1: Block diagram of the internal structure of a GPU.

should be drawn), texture data (such as an image of a building being mapped to a large cube, to simulate a building), and vertex data (the coordinates of all the objects in the scene) to the GPU frontend via a bus such as PCI Express.

Next, commands, texture data and vertices are processed by the *vertex processors* or *vertex shaders* in the GPU. The vertex shaders run a *vertex program* for each vertex in the scene. The vertex programs can modify the vertex's position, the color and the texture coordinate associated with the vertex. The software application can upload the vertex program to the GPU in advance to transmitting rendering commands. On newer GPUs (from 2004 and onwards), vertex programs can access the texture memory of the GPU. An example of a situation where a vertex shader will be useful could be when rendering a mountain terrain. The software application running on the CPU would send a 2D grid of points along with a mountain terrain texture, where light colors in the texture indicate high elevations in the terrain and dark colors indicate lower elevations, to the GPU. The vertex program could read texture memory and determine, based on the color of the texture at the vertex's location, determine what elevation level the vertex should have, and thus generate a 3D terrain by outputting a vertex at the correct elevation level. The alternative would be to do the same work on the CPU, which would likely be much slower, since the GPU is able to process several vertices simultaneously and has heavily optimized floating-point units which have built-in instructions for dealing with small vectors.

After the vertex programs have been run, the vertex processors group vertices into *geometric primitives*. For example, when drawing triangles, three vertices are grouped into a triangle primitive. The vertex processors thus produce a stream of *primitives*, which are passed on to the cull/clip/setup block. This block uses efficient visibility clipping methods to remove primitives which are not visible to the camera and calculates equations for the geometric primitives to aid the rasterization block. (see [13] for information on visibility clipping methods.)

Next, the rasterization block converts the geometric data produced by the cull/clip/setup block into *pixels*, or points on the rendering target (e.g. the screen), by employing efficient algorithms for converting mathematical descriptions of lines, circles and other geometric figures to screen pixels (see [13] for an in-depth description of these algorithms). The rasterization block also uses information from the z-cull block to determine if the current pixels being processed are occluded by some previously processed pixels. If this is the case, the current pixels can be discarded and are not processed by the later stages in the pipeline. However, it is not certain that all the points produced by the rasterization block will be drawn. Some of them might end up being occluded by other objects (but this was not detected by the z-cull block at that time or the pixel is occluded by pixels processed after the current pixel), and some pixels might have to be blended with other pixels — i.e. two pixels are fused together into one pixel. Thus the rasterization block only produces *candidate* pixels, or *fragments*. Some fragments will end up as pixels, and some will not.

The rasterization block transfers fragments to the *fragment processors* or *pixel processors*, which are programmable processors which do per-fragment processing. Just as the vertex processors can run vertex programs, the fragment processors can run fragment programs, which are invoked for each fragment. The fragment programs can write up to four 16 or 32-bit values to up to four fixed memory locations, and can read texture memory at any location. The fragment programs are called with the texture coordinates of the current pixel as an argument (which is passed to it from the vertex processor). For example, a fragment program could be used to calculate how each pixel should be lighted, using sophisticated lighting models based on real-world physics. Or, each pixel could represent a value in a numerical approximation to the solution of a two-dimensional differential equation, and the fragment processor could execute a program to calculate the Gauss-Seidel numerical method for computing solutions to systems of linear equations. The fragment processors operate on several hundred pixels simultaneously, and, as the vertex processors, have optimized floating-point hardware capable with built-in vector instructions.

The final stage in the rendering process is to do z-compares and blending (collectively referred to as *raster operations*). The z-compare units determine whether each fragment is occluded by some other fragment. This information is also sent to the z-cull block, so that new fragments can be discarded at an early stage (without going through the fragment processors) if it is already known that some pixels will be drawn in front of them. Finally, pixels are blended (if necessary) and their final color is written to one of the GPU's DRAM partitions. Having several DRAM partitions lets several pixels be written to memory simultaneously. The DRAM of the GPU stores textures and other input data to the rendering pipeline, and the color values of the image being drawn to the screen. The GPU DRAM memory typically has far more bandwidth than the normal DRAM of the CPU, but also has greater latency.

The GPU also employs several caches (which are not shown in Figure 2.1) and uses aggressive caching strategies . For instance, there is a *vertex cache* which checks whether a vertex is passed to a vertex processor twice. If that is the case, the vertex program is run only once. Also, textures are heavily cached in a separate texture cache.

It should be evident from Figure 2.1 that the graphics pipeline is implemented as several functional units which operate serially. But it is also evident that even though the input to one functional unit is the output of some other functional unit, the units do not process the *entire* stream of vertices before outputing data to the next functional unit. For instance, the vertex processors typically only read three vertices before they assemble the vertices into a triangle and pass them on to the cull/clip/setup block, which also pass simple primitives to the rasterizer. And the rasterizer passes large amounts of fragments to the fragment processors, which can process each fragment independently and in parallel. Hence if the vertex stream is very large — which is usually the case, since there typically will be hundreds of thousands of vertices and even more resulting pixels in a large scene — the functional units can operate in parallel. This enables so-called *task-level parallelism* since the individual blocks of the graphics pipeline can run in parallel. And since functional units which are connected lie very close to each other on the chip, communication between them is very fast. But *instruction-level parallelism* is also employed, since individual functional units process several elements at once. For example, there are many vertex and fragment processors which operate in parallel on several pixels simultaneously, and each individual vertex and fragment processor is capable of operating on a 4-value vector in parallel.

The fact that task-level and instruction-level parallelism is used, along with the fact that the individual blocks do not have complex dependencies between them, makes it possible to design very efficient GPUs whose designs are tailored to push large amounts of vertices through the graphics pipeline. We will look closer at how the GPUs perform in comparison to CPUs, which are structured in an entirely different and much more complicated manner, in Section 2.4. We now take a closer look at the vertex and fragment processors.

2.2.2 The Vertex Processors

The vertex processors run a vertex program for each vertex. A very simplified block diagram of the NVIDIA GeForce 6 and 7 vertex processors is shown in Figure 2.2 on the next page.

The vertex processor runs a program for each vertex passed to the GPU. Each processor consists of one scalar unit able to do floating-point operations, with at most 32 bits of precision. The vector unit operates on four values *simultaneously*, and is capable of doing a four-way SIMD (single instruction, multiple data) multiply-and-add (i.e. calculating $W = X * Y + Z$ where W, X, Y and Z are four-component vectors) instruction per clock cycle. In addition, the scalar unit can also in parallel to the vector unit be instructed to do a special scalar function per clock cycle, such as exponential functions, calculating reciporals and trigonometric functions. It can therefore be regarded as a MIMD (multiple instructions, multiple data) processor since it can execute different instructions on different data at the same time.

The vertex processor also includes a simple branching unit. Programs on the vertex processor may also access texture memory through the texture cache (since GeForce 6). Also, there is a primitive assembly unit which groups vertices into primitives for subsequent graphics pipeline stages, and a viewport processing unit which can transform vertices into global coordinates if they have been passed to the GPU in a different reference coordinate system.

The vertex processor on the GeForce 6 series processors allows for programs of up to 512 instructions and will run a maximum of 65,536 instructions per vertex. It also has 32 local registers available to programs. The vertex processor has not changed much in GeForce 7, except that vertex programs can have more complicated branching and can execute an infinite amount of instructions per vertex.

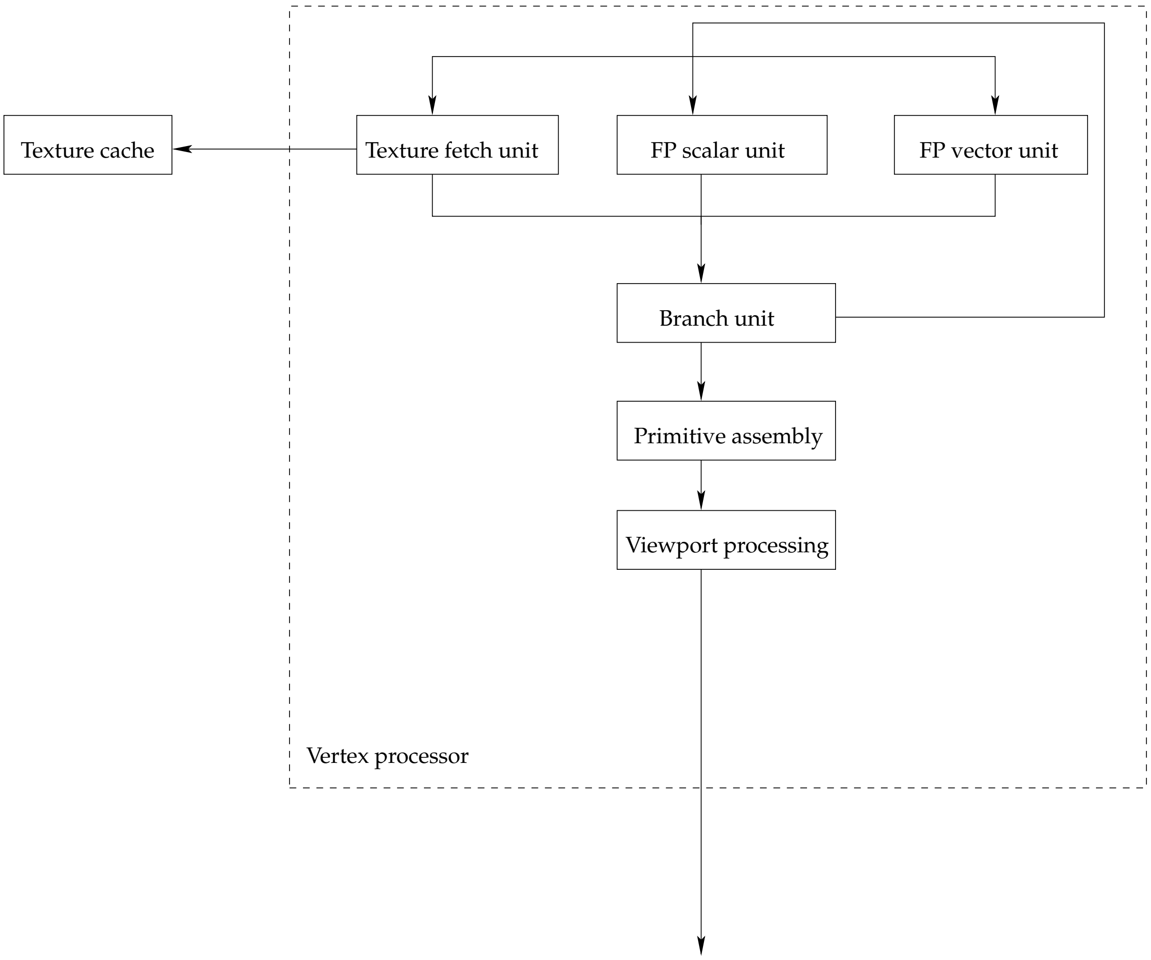

Figure 2.2: Block diagram of the internal structure of a vertex processor, from *GPU Gems 2* [11].

2.2.3 The Fragment Processors

The fragment processors run a fragment program for each vertex. They are slightly different than the vertex processors. A simplified block diagram is shown in Figure 2.3 on the facing page.

Fragment programs are run for each fragment passed from the rasterizer. On the NVIDIA GeForce 6 and 7 series, each fragment processor operates on four pixels at a time. A pixel is again a vector of four scalar values: (R, G, B, A) where R is the red intensity, G is the green intensity, B is the blue intensity and A is the alpha/transperancy value. Hence each processor operates on 16 values at a time, and there are usually many processors. The number of fragment processors multiplied by the number of pipelines per fragment processors is referred to as the number of *pixel pipelines* on the GPU; i.e. how many pixels the GPU can do operations on simultaneously.

The two FP shader units, which do the actual work, are shown in Figure 2.4 on the next page.

On the GeForce 6 and 7 GPUs, the FP shader units, which each have 4 pipelines, are able to:

- Do one multiply-add (MADD) instruction on each component of a pixel or a four-term dot product and another multiplication in series per clock (GeForce 6), or *two* MADD

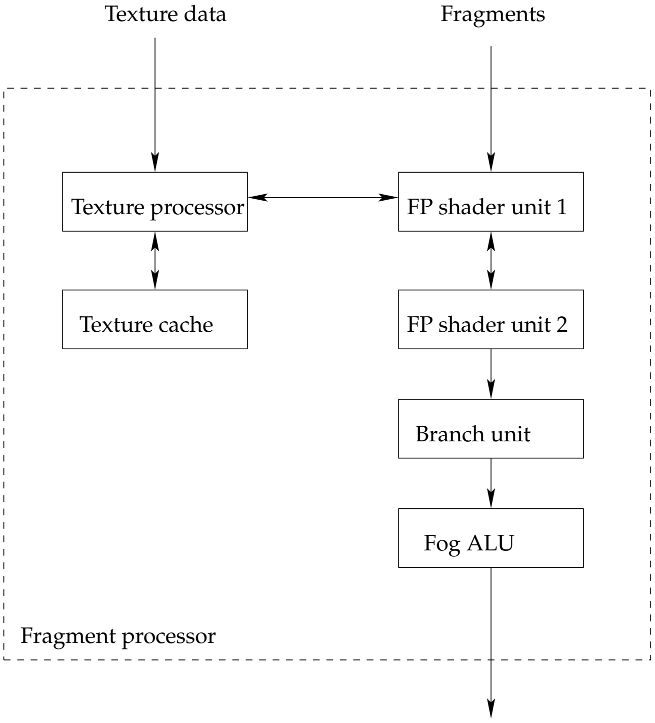

Figure 2.3: Block diagram of the internal structure of a fragment processor, from *GPU Gems 2* [11].

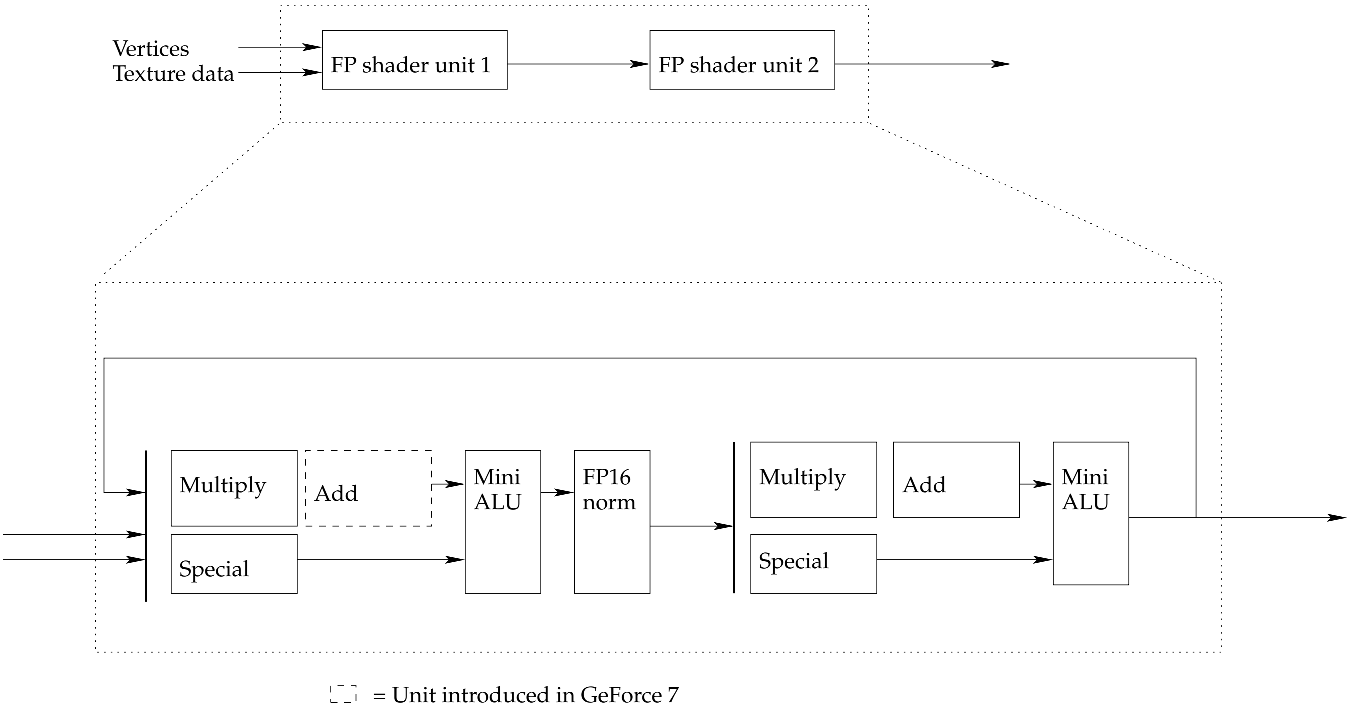

Figure 2.4: Block diagram of the FP shader units inside the fragment processors, adapted from *BeHardware.com*'s NVIDIA GeForce 7 article [12].

instructions on each pixel component or two four-term dot products in series per clock

(GeForce 7). In addition, the MADD operation on the pixel's alpha component may be replaced by some other scalar operation (the "special" unit).

- Do an independent reciprocal operation in parallel with the multiply-add instructions. This is done by the multiply/add unit.
- Normalize the result, but only with 16 bits of precision.
- Do very simple operations, such as multiplying by 2^k for some k, after the MADD with the mini ALU.

Since both FP shader units perform 4 operations per cycle (one per pixel component, excluding the fp16 normalization), each pipeline can do a maximum of 8 operations per fragment per clock cycle (or four if it has to fetch texture memory). Since there are 4 fragment pipelines in each fragment processor, a maximum of 32 operations can be performed per clock cycle per fragment processor.

The fragment processors also have several local registers available for program use (not shown in Figure 2.4). It can also filter textures and interpolate texture coordinates on to pixels coordinates, which has to be done if the fragment program reads the texture data value for the current pixel being processed. As for the vertex processors, the fragment processors support branching, but have no branch prediction units. Therefore, the fragment processors always execute both branches when a conditional branch instruction occurs, but only allow register writes from the correct branch. This has to be done so that (*a*) no branch prediction units are needed, and (*b*) execution is not stalled. This implies that if branches occur frequently in a program, the fragment processors' throughput will be negatively impacted.

Finally, the fragment processor also has a separate ALU for fog, as seen in Figure 2.3. The fog ALU can only perform the operation $P = F_c F_f + P(1 - F_f)$ where P is the pixel, F_c is the fog color, and F_f is the fog fraction. This operation can only be done using fixed-precision numbers, and is therefore not much used for general programming applications.

2.2.4 View of the GPU when doing general programming

Consider Figure 2.5, which shows how to look at the GPU from the outside.

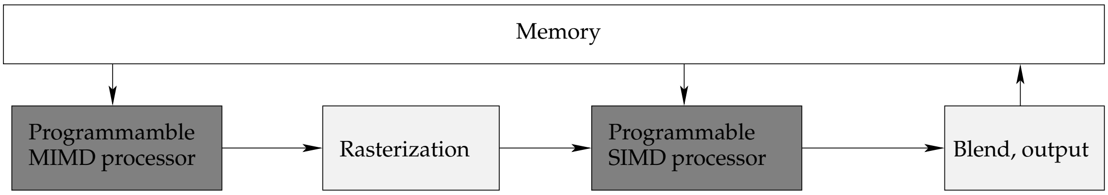

Figure 2.5: The GPU viewed from the outside.

The GPU is essentially a programmable MIMD processor (the vertex processors, with one vector and one scalar operation per clock cycle), taking in vertices or data, followed by a rasterization block which can either pass data directly to the fragment processors or *interpolate* values (for example, merely passing four coordinates to draw a rectangle will lead to the rasterizer interpolating the pixel coordinates of the vertices and passing them to the fragment processors), followed by a programmable SIMD processor (the fragment processors, with up to 32 operations per clock cycle per cycle), and finally a unit which can do simple blending, z-compares and write results to memory.

The vertex processors can pass three types of values to the next stages of the pipeline: A color value associated with the vertex, a transformed vertex position, and several *texture coordinates*, where each texture coordinate consists of two 16 or 32-bit floating-point values. The fragment processors can read the texture coordinates generated by the vertex processors, which are linearly interpolated over pixels inside the region spanned by the vertices. The ability to pass values as texture coordinates has been exploited in the implementations in this project, since it allows certain computations to be offloaded from the fragment to the vertex processors.

The fragment processors can output one four-component value, where each component is of 32 or 16 bits of precision, to a maximum of four rendering targets. Thus, a single invocation of a fragment program can generate at most 16 floating-point values. These values must be written to the fragment's position in the GPU's memory (recall that a fragment is a candidate pixel, and thus, each fragment is associated with a specific position in the GPU's texture memory). However, fragment programs can *read* from any position in texture memory.

It is apparent that the GPU architecture is quite different from CPU architecture, and therefore uses very different programming models. We will now look closer at ways of programming the GPU and its performance characteristics in the next sections.

2.3 GPU Programming

We will now look at how programming can be done on the GPU. First, we begin with an introduction to how traditional graphics applications use the GPU, before we look at how the same methods and programming interfaces can be used for general GPU programming (GPGPU).

2.3.1 OpenGL and DirectX

The GPU drivers typically support the *OpenGL* [14] and Microsoft's *DirectX* [15] application programming interfaces (APIs). We will mostly use OpenGL, as it is not bound to one particular platform, which is the case with DirectX. With these APIs, a developer may easily draw objects and apply color, lighting, shading and many advanced effects to a scene.

Typically, GPGPU applications use OpenGL or DirectX for passing commands, data and uploading the vertex and fragment programs to the GPU. The typical data flow from a software application to the GPU is illustrated in Figure 2.6 on the next page. Note that the GPU part of the figure is a coarser view of Figure 2.1.

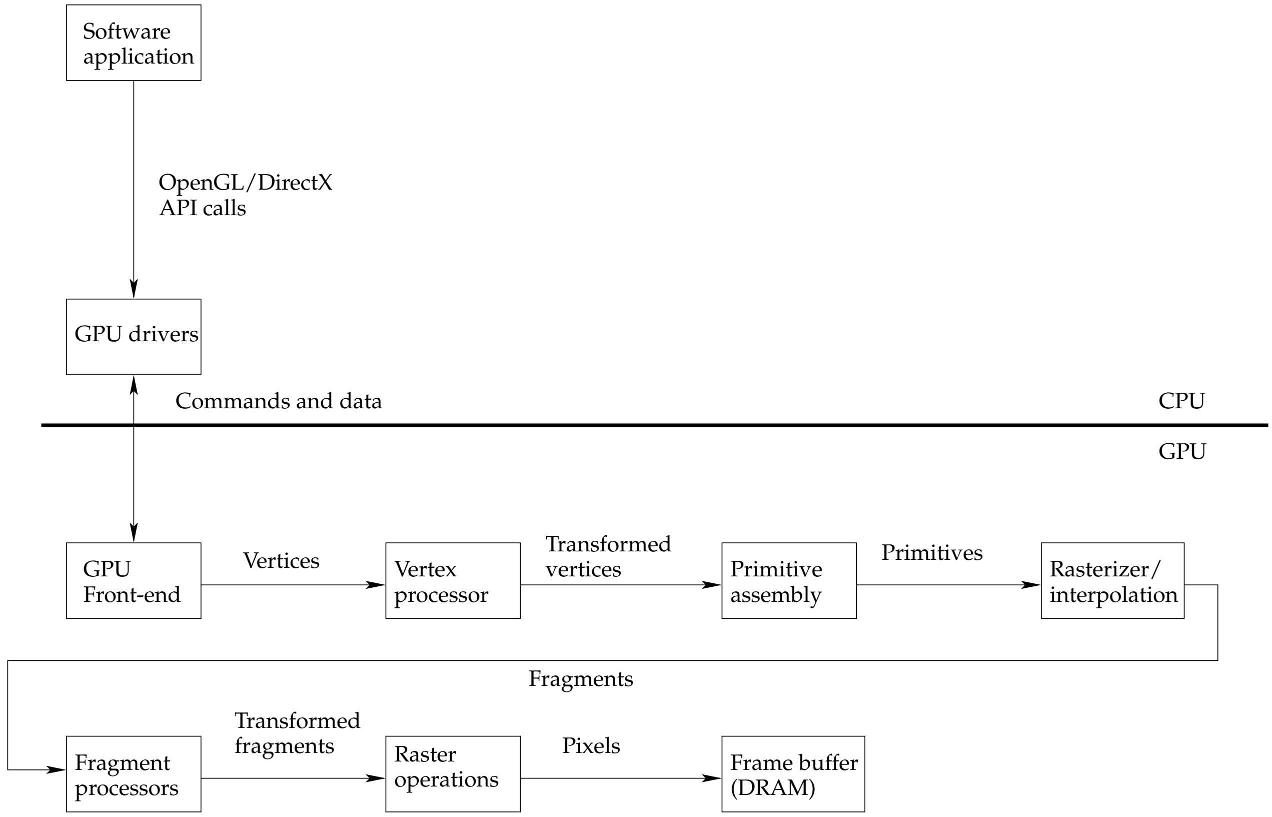

Figure 2.6: Data flow from a software application to the GPU, adapted from NVIDIA's Cg book [9]

2.3.2 Shader and GPGPU Languages

There are several specialized high-level languages for programming the vertex and fragment shaders.[1] Prior to the introduction of the high-level shader programming languages, the only way to program the shaders was through a low-level assembly language. With high-level languages, GPU vendors can do agressive optimization on the high-level code to optimize for a particular GPU. Since far less is known about GPU internals than CPU internals (mainly due to the highly proprietary nature of GPUs), using a high-level language usually gives better results than attempting to hand-optimize a low-level assembly language program.

But languages have also been created on top of high-level shader languages. These are called *GPGPU languages* and are created explicitly for doing more general programming on the GPU. Typically, these languages use a compiler to translate from the GPGPU language to a high-level shader language.

We now look at some of the high-level shader and GPGPU languages. Low-level shader languages are practically never used anymore, since writing code in a high-level shader language usually yields better performing shader programs.

[1] Note that special-purpose shader languages for per-vertex and per-pixel processing is not something new. Rendering packages such as Pixar's RenderMan, used to create several animated movies, have included such special-purpose languages since the 1980s. However, RenderMan is not made for *real-time rendering*, and the shader programs execute on the CPU. It is therefore not used for speeding up rendering, but for creating more advanced effects.

Shader Languages

The three main shader languages used today are: NVIDIA's Cg, Microsoft's HLSL, and The OpenGL Shading Language (OpenGL SL). We have already briefly mentioned HLSL, which is DirectX's high-level shading language, and OpenGL SL, which is the OpenGL equivalent. However, NVIDIA's Cg is perhaps the most interesting from a GPGPU standpoint.

NVIDIA Cg NVIDIA's Cg (*C* for *g*raphics) [9] shader language is probably the most popular language. It provides a simple, C-like syntax with arrays, structures, vectors and matrices as native types. It also has a standard library with many commonly used functions. NVIDIA ships a separate Cg compiler, available for most operating systems, which compiles Cg code into heavily optimized fragment and vertex processor instructions.

Although Cg is made and maintained by NVIDIA, it is perfectly possible to write Cg programs which run on GPUs from other vendors. The NVIDIA Cg compiler currently supports over 20 *profiles* and when compiling programs, a *target profile* must be chosen. The target profile defines a subset of Cg which is supported on a specified GPU or API version. For instance, the `arbfp1` profile creates fragment code which is executable on all GPUs supporting the standard OpenGL 1.3/1.4 fragment program extension. Since nearly all GPUs support this standard, the program can be executed on all types of GPUs.

However, compiling for a general profile limits what is possible to do in a vertex or fragment program. Since `arbfp1` is implemented by nearly *all* GPUs, such programs fail to take advantage of the more advanced features available in the most recent GPUs. For example, using the more specialized `fp40` and `vp40` profiles, which are only supported by recent NVIDIA GPUs, adds support for executing loops which can not be unrolled at compile-time, increased program length and more available registers. Thus, a Cg program may be valid only for some specialized profiles and invalid for the more general ones.

The major advantage of Cg is that it can be used with nearly all GPUs and both OpenGL and DirectX, and is to a certain degree independent of OpenGL and DirectX versions. To achieve indepdendence of 3D API, GPU and operating system, Cg includes the *Cg runtime*, which acts as a layer between the application and OpenGL or DirectX, as shown in Figure 2.7 on the following page. Figure 2.7(*a*) shows how an OpenGL program using Cg operates, while Figure 2.7(*b*) shows an OpenGL program utilizing OpenGL SL works. The Cg compiler translates the Cg program into a suitable format which can be read and translated to vertex and fragment processor instructions by the OpenGL driver. For example, when compiling for the `arbvp1`/`arbfp1` profiles, the Cg runtime converts the Cg program to a low-level assembly-like language (specified by the OpenGL standard) and passes the program to the OpenGL driver, which transforms the assembly program to vertex and fragment processor instructions.

Cg shader programs can be compiled either dynamically at run-time or by using the standalone Cg compiler. Usually, Cg programs are compiled dynamically in order to optimize the program for the GPU the program is currently running on. The Cg runtime provides functions for determining the optimal vertex and fragment profiles for the GPU the program is running on. Also note that even though using Cg requires the Cg runtime, it introduces practically no overhead compared to using the OpenGL SL (Figure 2.7 on the next page), since high-level OpenGL SL applications must also be compiled, but the compilation always happens at run-time within the OpenGL driver. The compilation overhead is merely moved into the OpenGL driver.[2]

[2] When compiling Cg programs on some NVIDIA GPUs at run-time with certain target profiles, compilation

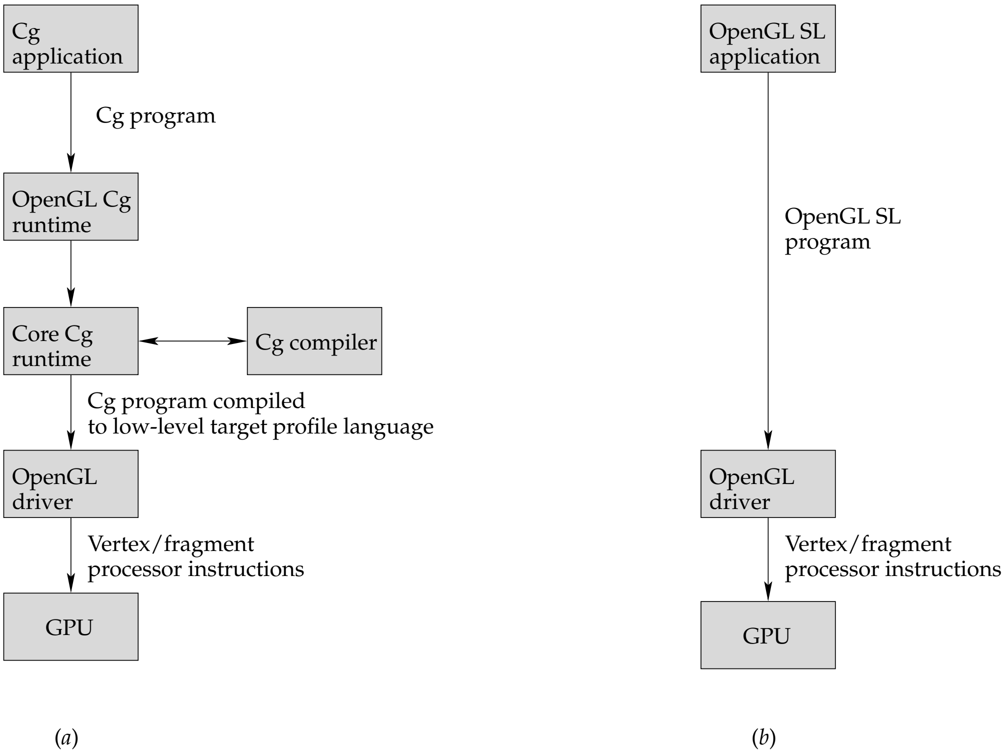

Figure 2.7: (*a*) Passing Cg programs to the GPU — (*b*) Passing OpenGL SL programs to the GPU

Since some of the features in the profiles for newer GPUs can be very useful in GPGPU applications, the GPGPU code in this project will be created for the most recent `glslv`/`glslf` profiles. This ensures that we can use the most recent features and increases what we are allowed to express in our programs in addition to not tying our shaders to one particular vendor, since the code should be able to run on any OpenGL 2.0 compliant GPU.

Microsoft HLSL DirectX also has its own high-level shader language (briefly described below), and also specifies the DirectX *Shader Model*, which specifies what types of shader programs GPUs implementing the Shader Model must support by describing a low-level assembly-like language GPU drivers must be able to compile into shader instructions. There have been several versions of the Shader Model, with increasing requirements as to what the GPU must support. For instance, in Shader Model 2.0, branching instructions and precision were limited compared to the more recent Shader Model 3.0.

Microsoft's HLSL (*H*igh *L*evel *S*hader *L*anguage) is actually syntactically the same language as Cg, but includes some additional support accessing DirectX via the shader programs. It is essentially Microsoft's implementation of Cg, and is therefore bound to DirectX. For GPGPU applications there is no advantage in using HLSL instead of Cg.

can also happen inside the OpenGL driver by the use of the `GL_EXT_Cg_shader` OpenGL extension implemented by the NVIDIA drivers, which allows passing Cg programs directly to OpenGL. Thus in this case, OpenGL SL and Cg programs are probably compiled using a common back-end compiler inside the driver, and the Cg runtime does not need to do any compilation. Developers can also, if using this extension, pass Cg programs directly to OpenGL, at the cost of making programs less portable.

The OpenGL Shading Language OpenGL version 2.0 specifies a way to write shader programs by the means of a high-level C-like language that GPUs supporting the OpenGL standard must be able to execute. In OpenGL version 1.5, this was an optional extension to the OpenGL standard, but with OpenGL 2.0, support for shader programming was made mandatory and significantly improved. In OpenGL versions prior to 1.5, the only option available for writing shader programs was to write shader programs in a special assembly language. When programming shaders this way (using either the OpenGL SL or the low-level assembly language), the application passes a string containing the program source code to the vendor-provided OpenGL driver, which dynamically compiles the program into shader instructions and uploads it to the vertex and fragment processors.

The OpenGL shading language [16] is also similar to Cg and HLSL, and is a part of the OpenGL 2.0 standard. But Cg has so far been used more for GPGPU applications, and since Cg (since version 1.5) has a `glslv/glslf` profile target, it can output OpenGL SL compliant shader programs. OpenGL SL implementations were availble in late 2003, while Cg was released in 2002. Cg has, probably due to the head start, the massive support it has received from NVIDIA and the fact that it has a standalone compiler (enabling developers to quickly check if a program is syntactially correct without actually attempting to run the program), become more popular than OpenGL SL for GPGPU applications. OpenGL SL also lacks support for multidimensional arrays, non-square matrices, and some commonly used operators.

What language do we choose? All three shader languages are fairly similar — for a detailed comparison, see e.g. [17]. In this project, we have chosen to use NVIDIA's Cg language. The reasons are:

1. Cg is relatively mature and has the most features of all the shader languages,
2. Cg provides (contrary to OpenGL SL) a standalone compiler which enables developing shader programs without actually running the application,
3. Cg is the most used language for GPGPU applications,
4. Cg enables development of shader programs which are not tied to a particular 3D API, operating system platform or GPU vendor.

One potential disadvantage of Cg is that it is fully controlled by NVIDIA. However, since Cg can be used with GPUs from other vendors by compiling with non-NVIDIA specific profiles, this is of little importance.

Another big plus for Cg is that it is the *only* shader language which supports the Sony PlayStation 3, which will probably also be an interesting platform for GPGPU applications. [3]

GPGPU Languages

There are two main GPGPU languages: *BrookGPU* and *Sh*. These languages generate OpenGL SL or Cg code, which is again compiled to vertex and fragment processor instructions. We have chosen to not use these languages in this project, since they are relatively immature and do not currently deliver as good performance as Cg, but will probably become much more interesting in the future.

[3] PlayStation 3 is fully compliant with the embedded version of the OpenGL 2.0 standard *except* that it uses Cg instead of OpenGL SL.

BrookGPU BrookGPU [18] started as a research project at Stanford University's Graphics Lab. It is specifically designed for GPGPU applications, and is higher-level than Cg, which makes it easier to write GPGPU applications. BrookGPU comes with a compiler which translates BrookGPU code into Cg programs.

Sh Sh [19] is not really a programming language, but an integration of shader programming into C++. With Sh, shader programs are written directly in C++, and shaders can even share variables with a regular C++ application. Sh is intended for both graphics and GPGPU purposes.

2.3.3 General Programming on the GPU

Programming GPUs consists of writing the procedures that do computations on the vertex and fragment streams. As we have seen, the GPU is structured around a *stream computation model*, where data flows between individual *kernels* (vertex and fragment programs) which transform the stream. This stream computation model is the only model supported by GPUs.

NVIDIA engineer Mark Harris suggests in [20] the following mapping of computational concepts of traditional stateful CPU programming to the GPU stream programming model:

- *Rendering = Executing*. To actually run a fragment or vertex shader program which does computation, we simply pass drawing commands to the GPU. Passing drawing commands will start the rendering process. Most GPGPU applications — including those developed in this project — will compute something over a 2D grid. Hence, we simply tell the GPU (via for example OpenGL) to render a 2D square to start the calculations.
- *GPU textures = CPU arrays*. When we want to use an array, a texture can be used on the GPU. GPU textures are natively two-dimensional, but three-dimensional textures (and thus 3D arrays) will also be natively supported by future GPUs.
- *Fragment shader programs = Inner loops*. A program on the fragment processor is executed for all the fragments, and writes values back to texture memory. A fragment program can therefore be seen as the *inner loop* of computations. (The *outer loops* on a CPU would be a loop iterating over all the fragments. This loop is implicit when programming GPUs, since the GPU will automatically execute the fragment programs for all fragments.)
- *Rendering to texture memory = Feedback*. If a computation has several steps, we write the results of each step to texture memory. The next step can then use the result as its input. When all the steps are done, we read texture memory to get the result back to the CPU.
- *Vertex coordinates = Computational range*. The input vertices determine what pixels will be rasterized and therefore what coordinates the fragment programs will write to. Vertex coordinates therefore determine the range of the computation.
- *Texture coordinates = Computational domain*. The computational domain can be represented as a texture. Texture coordinates are passed along with each vertex. The GPU automatically interpolates the vertex texture coordinates for the fragments which will be rendered. GPGPU applications can take advantage of this, since it means that the input domain will automatically be shrinked or magnified to cover the output domain. On a CPU, such shrinking and magnifying would have to be done manually.

Although these mappings are fairly straightforward, special care must be taken when programming the GPU due to its specialized design.

2.3.4 Limitations

GPUs are able to execute calculations extremely quickly. But due to their specialized design, programming GPUs is more tricky than programming CPUs. When programming GPUs, the following must be considered [20]:

1. *Branching is usually not a good idea.* GPUs have absolutely no branch-prediction units, and since the SIMD fragment processor operates on several fragments at a time, if a branch is taken for some but not all fragments, both branches will be executed *for all fragments*. Furthermore, if-else instructions take up to six cycles. In six cycles, the fragment processor can do at least $4 \times 4 \times 6 = 96$ floating-point multiply-add operations (since there are four pipelines, each of which operates on four values, per cycle), and twice as much if we also take into account the second multiply-add unit which can do a multiply-add operation on the result of the first unit. Therefore, if an algorithm has a large number of branches, it may not be suitable for GPUs — and if computation can replace branching, it is almost always a good idea. In this project, we have replaced all branches with computations.

2. *GPU cache is different from CPU cache.* GPU cache is optimized for fetching textures, not general memory access. It is therefore small compared to the CPU cache. In addition, it is optimized for *two-dimensional locality* (i.e. for fetching 2D texture data), rather than 1D locality, which is what the CPU cache is optimized for. Furthermore, the cache is *read-only* — the only way to write values to texture memory is through the fragment programs.

3. *Random memory access is problematic.* A fragment program can only write to up to four fixed memory addresses in GPU texture memory on the GeForce GPUs, and each memory location can contain four 16 or 32-bit values. The addresses to which the fragment program can write are determined before the fragment program is run. Therefore, it is not possible for one instance of a fragment program to scatter data across memory. Also, GPUs are optimized for *sequential* memory access. This makes GPUs worse at random memory accesses than CPUs. Since many algorithms depend on quick random memory access time, they might need to be heavily adjusted or redesigned to run on the GPU.

4. *Floating-point precision.* GPUs often perform best when working with only 16-bit precision because it frees up more registers and allows more values to be processed per clock cycle, and increases the number of elements that are fetched from GPU memory in each cycle. The fragment processor has a normalization unit which only operates on 16-bit values. Another potential problem is that some GPUs use shortcuts when rounding floating-point values which gives slightly greater rounding error but quicker hardware. This might lead to slight differences between computation results on the CPU and GPU.

5. *No integers or booleans.* GPUs deal with floating-point numbers, not integers. This can be a problem, since there are some 32-bit integers which can not be *exactly* represented in 32-bit floating-point format. GPUs also do not currently provide any bitwise operators, but Cg has reserved symbols for such operators for the future, and integer support has been announced in the NVIDIA G80 GPU (see Section 2.4.2).

6. *Data must be uploaded and downloaded*. Computation results need to be uploaded and downloaded to and from the GPU. The time it takes to upload and download the data might be so large that we should rather use the CPU for computation. Thus the GPU is suitable only when very large amounts of operations are done on the data.

We will take a closer look at exactly what applications the GPU is suitable for towards the end of this chapter.

2.4 Current GPUs: Price and Performance

Let us now look at how GPUs actually stack up to CPUs. First, we look at some of the current GPUs available. Then we look at how GPUs compare to CPUs. Finally, we discuss what GPUs are suitable for GPGPU applications.

2.4.1 Current GPUs

The GPU market is dominated by two vendors: NVIDIA Corporation and ATI Technologies, which was recently acquired by the CPU maker AMD.

NVIDIA Corporation GPUs

NVIDIA's GPUs have been favored by the GPGPU community, since NVIDIA has actively supported GPGPU projects and provides the Cg programming language. NVIDIA has also contributed large amounts of research on GPGPU programming. NVIDIA produces two types of graphics cards — the GeForce and the Quadro — where the GeForce cards are intended for personal use and home entertainment and Quadro cards are intended for graphics professionals.

For GPGPU applications, the chips used in the GeForce 6 and Quadro FX 540, 1300, 1400, 3400 and 4400 series (NV40 series) or the GeForce 7 and Quadro FX 3500, 4500 and 5500 series (G70 series) are most suitable, since the vertex and fragment processors were dramatically improved from the previous GeForce chips. The G70 chip, which was launched in 2005 and further developed in 2006, is based on the NV40 chip from 2004 and has not been radically redesigned. New features include more fragment pipelines, more features in the fragment processors (the first FP shader unit is able to do an add after the multiply), and higher memory and internal clock speeds. The G70 is also suspected (though unconfirmed by NVIDIA) to have a larger and improved cache.

G70 comes in various versions. The cheaper cards have a G70 chip with fewer vertex and fragment processors, less memory bandwidth and higher latency than the more expensive cards. Inferior cards have lower numbers. For instance, the GeForce 7600 is inferior to the GeForce 7800. The same applies to the Quadro FX cards.

Graphics cards based on G70 are currently the most attractive ones to use for GPGPU applications. Information on some GeForce cards is shown in Table 2.1 on the facing page. Information on some Quadro FX cards is shown in Table 2.2 on the next page. For information on several other NVIDIA cards, see [21] and [22].

Although some of the Quadro cards seem inferior and overpriced in comparison to the GeForce cards, they have one distinct advantage: The ability to have more memory and

Graphics card	GF 6800 Ultra	GF 7600 GT	GF 7900 GS	GF 7950 GT
Chip	NV45	G73	G71	G71
Transistors	222 million	177 million	278 million	278 million
Core clock speed	400 MHz	560 MHz	450 MHz	550 MHz
Memory clock speed	1100 MHz	1400 MHz	1320 MHz	1400 MHz
Memory bandwidth	33.6 GB/s	22.4 GB/s	42.2 GB/s	44.8 GB/s
Interface	PCI-E x16	PCI-E x16	PCI-E x16	PCI-E x16
Pixel pipelines	16	12	20	24
Vertex processors	6	5	7	8
Released	2004	2006	2006	2006
Retail price, NOK, Dec 2006	549 (256 MB)	949 (256 MB)	1,749 (256 MB)	2,495 (512 MB)

Table 2.1: Properties of some recent GeForce graphics cards

Graphics card	Quadro FX 3500	Quadro FX 4500	Quadro FX 5500
Chip	G71	G70	G71
Transistors	278 million	302 million	278 million
Core clock speed	470 MHz	470 MHz	700 MHz
Memory clock speed	700 MHz	525 MHz	1000 MHz
Memory bandwidth	42.2 GB/s	33.6 GB/s	33.6 GB/s
Interface	PCI-E x16	PCI-E x16	PCI-E x16
Pixel pipelines	20	24	24
Vertex processors	7	8	8
Released	2006	2006	2006
Retail price, NOK, Dec 2006	7,595 (256 MB)	13,582 (512 MB)	23,980 (1024 MB)

Table 2.2: Properties of some recent Quadro FX graphics cards

(slightly) faster readback (download) times. Quadro FX cards have up to 1 GB of memory, while current GeForce cards have a maximum of 512 MB. This might be important for GPGPU applications. They are also advertised as having quicker download times, which is *very* important for GPGPU applications. But the main difference and the reason for the enormous price gap lies in the Quadro cards supporting advanced stereoscopic vision, certain high-quality video outputs and the ability to smooth (antialias) lines at a higher quality, which is useful for certain CAD applications. These features are irrelevant for GPGPU programs. There is no difference between the GPU and memory bus of the Quadro FX and GeForce cards (in fact, some GeForce cards use a quicker memory bus!)

Any GeForce 6 or 7 series GPU card or Quadro 3500, 4500 or 5500 card with at least 256 MB of RAM will probably be suitable for GPGPU usage. Performance will increase as the number of pixel pipelines increase, since the number of pixel pipelines determines how many pixels (values) can be rendered (computed) in parallel.

ATI Technologies GPUs

ATI Technologies GPUs have been less popular within the GPGPU community, since NVIDIA has been much more active supporting GPGPU programming. ATI's chips are named *Radeon* and come in both regular and *Pro* versions.

For GPGPU applications, GPUs should be of at least the X1000 series (having an R500 GPU), introduced in 2005. Before the X1000 series, ATI chips were not very suited for GPGPU applications. Since NVIDIA provides much better support for GPGPU applications than

ATI, it is hard to recommend an ATI GPU for such applications currently. This will, however, probably change in the future. ATI has recently begun to broaden its GPGPU support.

Information on some recent ATI GPUs is shown in Table 2.3.

Graphics card	X1600 Pro	X1800 GTO	X1950 XT
Chip	R515	R520	R580
Transistors	157 million	321 million	384 million
Core clock speed	500 MHz	500 MHz	625 MHz
Memory clock speed	780 MHz	1500 MHz	1800 MHz
Memory bandwidth	12.4 GB/s	48.0 GB/s	57.6 GB/s
Interface	PCI-E x16	PCI-E x16	PCI-E x16
Pixel pipelines	12	16	16[4]
Vertex processors	5	8	8
Released	2006	2006	2006
Retail price, NOK, Dec 2006	720 (256 MB)	1,749 (256 MB)	3,149 (256 MB)

Table 2.3: Properties of some recent ATI graphics cards

As seen from Table 2.3 and 2.1 on the preceding page, ATI and NVIDIA GPUs are fairly similar. The architecture is slightly different, with high-end NVIDIA GPUs having 24 pixel pipelines each capable of fetching texture data in parallel, while the ATI X1900 XT has 64 pixel pipelines where only 16 may do texture fetches simultaneously. In practice, performance will be similar for both GPUs.

One problem with ATI GPUs, and the reason we have opted for NVIDIA GPUs throughout this project, is drivers. Their Linux drivers have traditionally been of lower quality than NVIDIA's Linux drivers, and no drivers exist for FreeBSD or Solaris, which are frequently used as cluster operating systems.

2.4.2 Future GPUs

GPUs are evolving quickly. Large developments have happened, even during the limited time period of this project. We therefore include this section to describe the latest developments that have occured through this project.

In November 2006, NVIDIA launched the G80 GPU, which has over 600 million transistors. This GPU has a core 575 MHz clock speed, and a memory clock of 900 MHz. The memory bandwidth is up to 86.4 GB per second, representing a clear improvement compared to prior GPUs. The shader processors operate at 1350 MHz.

Next-generation GPUs, such as NVIDIA's GeForce 8 (released one month before the end of this project), and ATI's Radeon R800, feature programmable *geometry shaders*. These geometry shaders can do operations on entire primitives, such as triangles and squares. Whether these can give any advantage to general programming on the GPU and exactly how they will work is currently unknown.

These GPUs also integrate the vertex, geometry and fragment processors into a *unified shader*. Unified shaders solve two problems: First, the three processors will have similar capability (currently, fragment processors usually have more features than vertex processors) and second, applications can dynamically configure the unified shader according to the application needs — for example, an application which loads the fragment processors heavily but does not use the vertex processors can configure all the unified shaders to act as fragment processors. This leads to better utilization of the GPUs resources. The NVIDIA G80 has 128 such unified processors (which is about four times more processors than previous NVIDIA GPUs).

The GeForce 8 GPU has native support for integers, while prior GPUs can only emulate integers. This feature was specifically aimed at GPGPU projects, and should increase the number of applications suitable for the GPU. It is currently unknown whether new integer-specific operators, such as bit manupulation operators, are supported.

NVIDIA has also launched (but not, at the time of writing, released) the CUDA library, which only supports the GeForce 8 GPU. This allows programmers to use the GPU's computing power directly from the C programming language, and also permits for separate threads to run on the GPU. CUDA will include a BLAS and FFT library, implemented on the GPU, to allow developers to easily offload computations to the GPU. Debugging of GPU programs is also improved, and upload and download times are also said to be shorter. CUDA will likely make GPGPU applications far much easier to write.

ATI has also launched a GPU specifically made for computation. The product, marketed as a "dedicated stream processor", is essentially an R850 GPU with 1 GB of RAM.

All the recent developments indicate that GPUs will increasingly support general programming in the future.

2.4.3 GPUs versus CPUs

Consider Table 2.4, which compares the GPU and the CPU we will use for performance measurements in this project.

Characteristic	NVIDIA Quadro 3500	Intel Xeon 5160 Dual-Core
Transistors	278 million	291 million
Memory bandwidth	42.2 GB/s	6.4 GB/s (typical)
Clock speed	470 MHz	3000 MHz
L2 Cache	Unknown	4 MB
Power usage (load)	80 W	122 W
Retail price, NOK, Dec 2006	7,595	7,450

Table 2.4: Properties of a GPU and a CPU

The GPU has nearly the same transistor count as the CPU, even though the CPU much more general than the GPU. The difference lies in what the transistors are *used* for. On the GPU, most transistors are used for computation units, while on the CPU, most transistors are used for cache, branch prediction unit, out-of-order executions, prefetching strategies, and supporting large and complicated instruction sets. The cache makes the CPU very good at keeping the latency of random memory accesses fairly low, but the bandwidth suffers. Therefore, GPUs are much more able to deliver high performance in applications which spend most of their time doing computation. A GPU can typically have about six times or more the gigaflops rate of a CPU [23], and it uses less power! As can be seen from the table, the price of the GPU we use for our measurements is nearly equal to that of the CPU.

2.5 Conclusions: Tasks Suitable for GPUs

We conclude, as [23], that the GPU is suitable for:

- Applications which have a high *arithmetic intensity*, i.e. has a large ratio of operations per memory access.
- Applications where individual elements in the data stream are independent and are thus easily parallelizable.

The GPU is *not* suitable for:

- Applications having a random access pattern.
- Applications having a many complicated branches and inherent serial dependencies between individual computation elements.
- Applications which are not easily parallelizable.

Examples of applications well suited for the GPU were mentioned in the previous chapter. Examples of applications that are *not* suitable for the GPU are recursive problems, such as calculating the Fibonacci sequence through using the recursive formula $F_n = F_{n-1} + F_{n-2}\,(F_0 = F_1 = 1)$ (because computing one element depends on computing two other elements), and a lexical analyzer or a compiler (which has to do several random memory accesses since it uses several data structures and complicated branching).

In sum, we can say that the GPU reflects the trend where computation is less expensive than communication. The GPU is designed for computation, and its programming model is so simple that communication (random memory access) is kept to a minimum. The CPU, which must tolerate other, more complicated programming models and access patterns, must sacrifice computation logic for communication logic.

It turns out that some mathematical transforms used in compression methods are suitable for GPU implementation. We will look at the methods, which both are easily parallelizable and involve large amounts of floating-point computations, and their implementation on the GPU through the rest of this report.

Chapter 3

Data Compression and Previous Work

In this chapter, we will look at data compression methods and previous GPU implementations of algorithms related to data compression. First, in Section 3.1, we will give some background material on general data compression. Next, in Sections 3.2 and 3.3, we will in greater detail look at two mathematical transforms commonly used for image data compression, describe and discuss specific algorithms — some of which have been implemented both on the CPU and the GPU in this project — for calculating these transforms, and consider previous GPU implementations of the transforms.

3.1 Introduction to Data Compression

Data compression is the process of encoding data to a representation which uses less bits than the unencoded representation. This process is done by reducing or removing redundancy in the data. Redundancy reduction is, in turn, achieved by reducing correlation in the data, since all data redundancy in general can be viewed as correlation. Several ways to reduce data correlation and thus compress data have been devised through large amounts of research in the field since the 1970s, and consequently, a pleathora of useful *compression methods*, which convert an unencoded *input stream* to a compressed *output stream*, exist.

In this project work, we focus on *transform coding methods*. Transform coding methods convert data into a representation (using a mathematical transform, such as the Discrete Cosine Transform, which is a variant of the Discrete Fourier Transform, or other transforms) that makes it easier for the other methods to compress the data. Transform coding methods are particularly interesting for image, audio and other signal compression, since they are usually *lossy*, i.e. the methods lose some (hopefully unimportant) parts of the data in return for better compression ratios.[1]. In some cases, losing only 10% of the data can give 90% better compression ratios [24]. This fits signal data well, since signal data usually tolerate a slight loss of information without a significant loss in quality.

The GPU can best be used to assist computation in lossy transform coding methods, applied to images, audio and other signals, due to the following reasons:

1. The GPU has far greater floating-point capability than the CPU, but currently has limited support for integer operations. Signals are often represented using floating point values values, and are not overly sensitive to rounding errors. A slight rounding error

[1] The *compression ratio* R is a metric by which compression algorithms can be evaluated. R is defined as the compressed data size divided by the original data size.

in an image or audio signal will, for example, not degrade quality as much as it will for textual data.

2. Other compression methods use auxiliary, random-access data structures. For instance, text compression methods often use a hash table to store substrings. Implementing such a hash structure on the GPU would not be a good idea due to its very poor random memory access performance. Lossy transform methods, however, require little auxiliary data structures but in turn require more computations (which is what the GPU is good at). In short, the arithmetic intensity of lossy transform methods is usually higher than that of the other methods.
3. The transforms used in transform coding methods can usually be computed in a data-parallel manner. Computation of the transformed value of an input stream element is independent from the transformed value of other input stream elements at the same step. Thus, these methods are well suited for the GPU's data-parallel architecture.

We will now describe how some specific, commonly used transforms can be used to compress data and their implementation on the GPU. Note that a mathematical transform in itself does *not* compress data, but is usually the most computationally intensive part of the process. We will briefly describe how JPEG uses transform coding to compress images in the next section to see how the transform fits into the entire transform coding process.

3.2 The Discrete Cosine Transform (DCT)

The *discrete cosine transform* or *DCT* [24, 25], which is a variant of the more well-known *discrete Fourier transform* (*DFT*), is frequently used for compression. In this section, we will first define the DCT in Section 3.2.4, then describe how it can be used for data compression and describe some of its properties in Sections 3.2.2 and 3.2.3, and look at efficient methods for DCT computation in Section 3.2.4. Finally, we discuss previous GPU implementations of the DCT in Section 3.2.6.

3.2.1 Definition of the DCT

The DCT is used as the core transform in the successful JPEG image format and also the MP3 and Ogg Vorbis audio formats. The success of these formats can largely be attributed to the properties of the DCT. There are four basic variants of the DCT, named DCT-I through DCT-IV, where the DCT-II is the most frequently used and therefore is usually referred to as "the DCT". Since DCT-III is actually the inverse transform of DCT-II, it is usually referred to as "the IDCT".

There are many ways to define the DCT. The FFTW library [26] for calculating Fourier transforms (and DCTs) uses a DCT definition [27] which is different from what the JPEG standard and others use [24, 28]. One issue with the FFTW definition is that applying the inverse DCT to values transformed with the forward DCT will give the original values scaled by a factor of $2n$, where n is the number of sample values. Thus, to get the original values back from the values obtained from the DCT, we need to scale the values from the IDCT by $1/2n$. The JPEG DCT definition (and other definitions) incorporate the scaling factors into the transform equations. However, the FFTW definition of the DCT allows it to more straightforwardly be computed as a DFT. In this report and our implementations, we will use the JPEG definition of the DCT [28], since most DCT algorithms implement the JPEG

DCT. Furthermore, this definition is also used by IEEE Standard 1180 [29]. The definition is as follows:

Definition 1 *Given n real numbers (**sample values**) x_t for $t = 0, \ldots, n-1$, the **(forward) discrete cosine transform** of x_t, Y_f for $f = 0, \ldots, n-1$, is given by*

$$Y_f \quad = \quad \sqrt{\frac{2}{n}} C_f \sum_{t=0}^{n-1} x_t \cos\left[\frac{(2t+1)f\pi}{2n}\right] \tag{3.1}$$

*Similarly, the **inverse discrete cosine transform** of Y_f, x_t ($t = 0, \ldots, n-1$) is given by*

$$x_t \quad = \quad \sqrt{\frac{2}{n}} \sum_{f=0}^{n-1} C_f Y_f \cos\left[\frac{(2t+1)f\pi}{2n}\right] \tag{3.2}$$

C_f is given by

$$C_f \quad = \quad \begin{cases} 1/\sqrt{2} & \text{for } f = 0 \\ 1 & \text{for } f > 0 \end{cases} \tag{3.3}$$

■

Note that Definition 1 easily generalizes to k dimensions as follows: To calculate the k-dimensional DCT of a set of real values, first calculate the DCT along the first dimension, then apply the DCT along the second dimension to the DCT values calculated along the first dimension, and so on up to dimension k. Thus, the DCT is *separable*.

3.2.2 Using the DCT for Compression

Let us now look at how the DCT can be used to compress data. We shall primarily look at two-dimensional digital images, but the concepts apply equally to other types of signal data.

Consider Figure 3.1, which depicts how the DCT can be used for data compression. The JPEG standard compresses images in a similar manner [24].

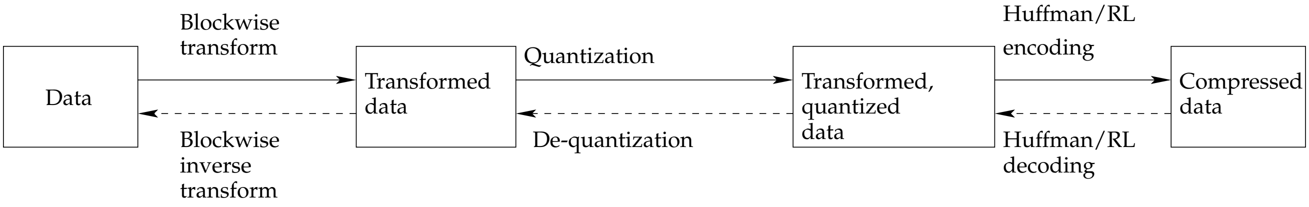

Figure 3.1: Data compression and decompression using the DCT. Filled lines represent compression steps, while dashed lines represent decompression steps.

The first step is to do a two-dimensional blockwise transform on the data. Common block sizes are 8 × 8, which is what JPEG uses, or 16 × 16. Large block sizes can give the decompressed image a more blocked appearence (i.e. block boundaries are more visible), but in turn can give better compression ratios. The JPEG 8 × 8 block size turns out to be a good tradeoff between compression ratios and image quality.

After the transform has been applied, the image is *quantized*. Quantization relies on the direct link between the high-frequency DCT coefficients and details in the image, and similarly a link between the low-frequency DCT coefficients and the slowly-varying parts of the image. This is due to the following: Computing the DCT on a data set is equivalent to computing a representation of the data set in terms of a set of cosine basis functions. This is a *frequency representation* of the data set. The high-frequency components (high f-values in Definition 1) give the amount of quickly varying information in the data set, which is the same as the details. Similarly, the low and middle-frequency components (low and middle f-values) give the amount of slowly and medium varying information in the data set. For example, in a picture of a black sky with small, bright stars, the high-frequency DCT coefficients will contain the information about the small, bright stars, while the low-frequency DCT coefficients will contain mostly information about the black sky.

Quantization involves scaling each transformed value, usually scaling higher-frequency components (located towards the lower right of the 8 × 8 2D DCT of the block) to numbers equal or near zero, while retaining the lower frequency components (located towards the upper left of the 8 × 8 2D DCT of the block). This ensures that very high-detail information is removed (since the quickly varying cosine basis functions are essentially left out of the representation), while the low- and mid-detail information is retained. See Figure 3.2.

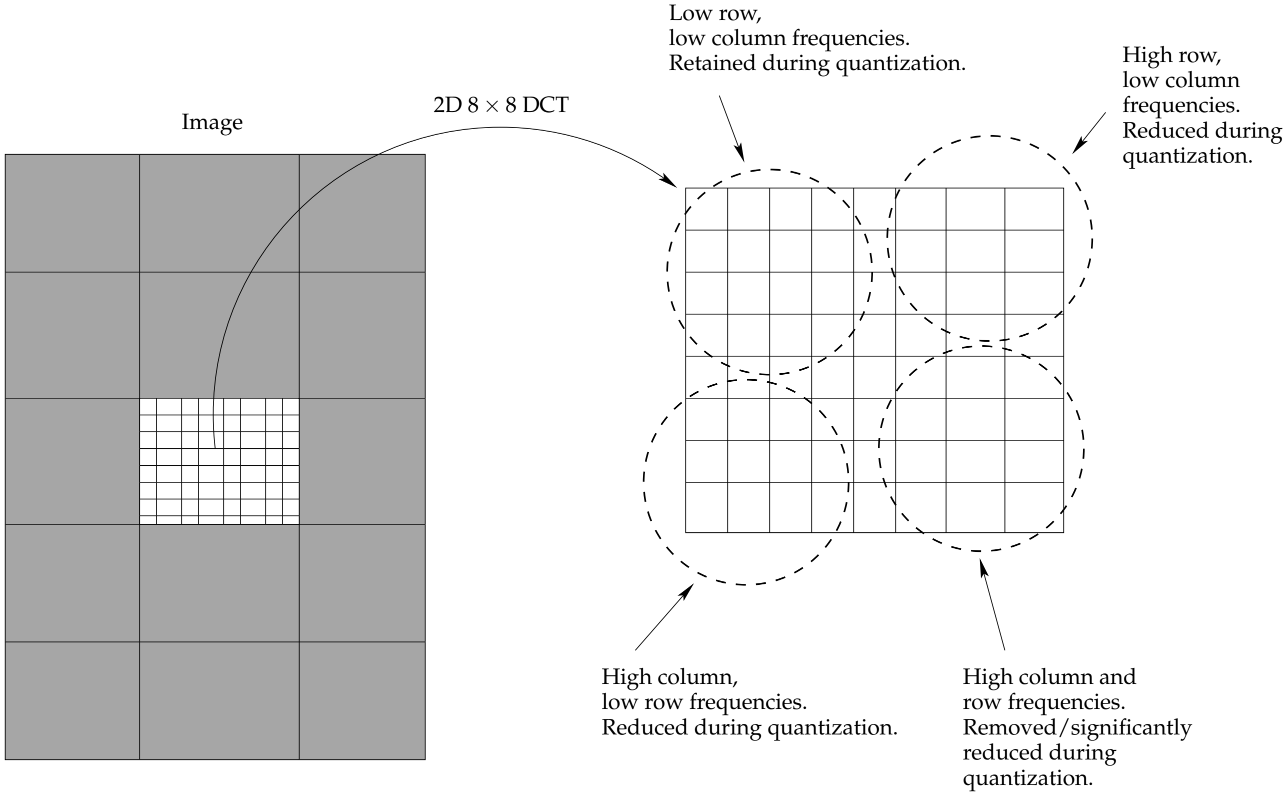

Figure 3.2: Quantization with the DCT

Removing very high-detail information in an image usually results in an insignificant loss of image quality, since in most images, most information is stored in the lower and middle frequencies. Consider Figures 3.3 through 3.6 on page 28. The figures show an image, taken by a digital camera[2], compressed at different JPEG quality levels. In Figure 3.4, a compression ratio of 0.35 (compared to Figure 3.3) is obtained, but visually, the quality difference

[2] The image is © Roger Midtstraum. Used with permission.

is unnoticable except for slight artifacts around certain edges (which are represented by the high-frequency coefficients). In Figures 3.5 and 3.6, we see, however, that as quantization increases (and as more high- and mid-frequency components are lost), details are lost. For example, the text on left building is not visible and the trees in the background are blurred in Figure 3.6.

There are various quantization schemes [24]. JPEG uses predefined quantization tables which are optimized for natural images. Another, simpler approach is to cut off all frequencies above a certain threshold.

In any event, the quantization process will produce blocks with *smaller values* and *large ranges of values equal to zero* — in fact, in JPEG, most of the block will consist of zero values. Two important observations can now be exploited:

1. Element $(0,0)$ of the block, or the *DC coefficient*, is proportional to the average value of the entire block. This follows from Definition 1. In fact, we merely need to divide this element by $4n^2$ ($n = 8$ for 8×8 blocks) to get the average value of all the elements in the block. But for most types of images, the average value and thus element $(0,0)$ is not likely to change dramatically between consecutive blocks.
2. For the other coefficients (the *AC coefficients*), there will be a few nonzero values, but they will likely have large ranges of zero-values in between. Furthermore, there will almost always be only zero values towards the lower-right section of the block.

These observations can be exploited by encoding all the DC coefficients in the image in the following way: Code only one DC coefficient and represent the other DC coefficients in the other blocks as differences. Since the differences are small, and since they are encoded along with the other coefficients which are also small compared to the DC coefficients, it is likely that there will be long sequences of the same symbol in the encoded output. Huffman encoding, which is a lossless compression method that works particularly well on sequences where few symbols from a large input alphabet occur frequently, and run-length encoding (RLE), which is another lossless compression method which efficiently encodes long strings of a single symbol, can be used in cojunction on the quantized transform values to obtain good compression ratios. RLE will compress the sequences and Huffman encoding will work well since there will be few different symbols. JPEG also uses some additional encoding tricks; see [24] for details.

To decompress the data, simply reverse the process: First do Huffman and RLE-decoding, then de-quantize, and finally apply the IDCT to each block. The result will, depending on the quantization scheme used, have some detail loss, but in many cases this loss will be hard to see.

As mentioned, computing the transform — in this case the DCT — is the most time consuming part of the compression process, and is also the part which is most suitable for GPU implementation. We investigate the implementation of the DCT throughout the next sections. First, we will look at some of the properties of the DCT.

3.2.3 DCT properties and problems

The DCT should be used for compression instead of the DFT or the DST (*discrete sine transform* — using sines instead of cosines in Defition 1). The DCT turns out to be superior to the DFT and DST for compression, since it is better at compacting energy at the low frequencies, while the DFT spreads the energy throughout a larger frequency spectrum. Thus, it is not efficient to merely cut off certain frequencies when using the DFT, since signal energy will

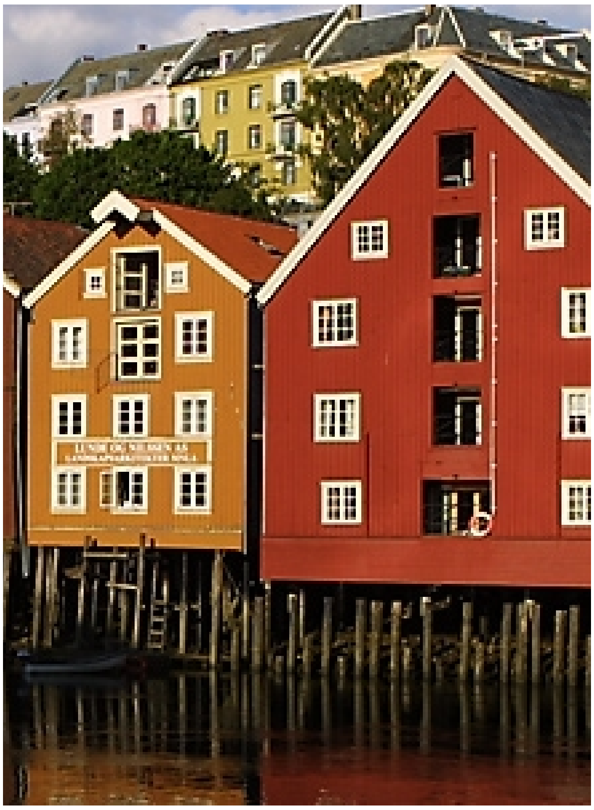

Figure 3.3: Original JPEG image, using almost no compression.

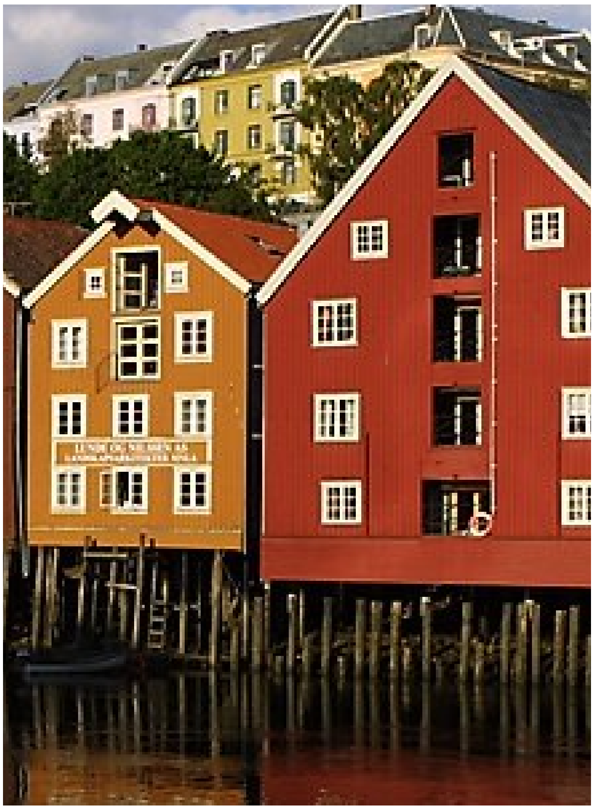

Figure 3.4: Image with JPEG quality 85. Compression ratio: 0.35

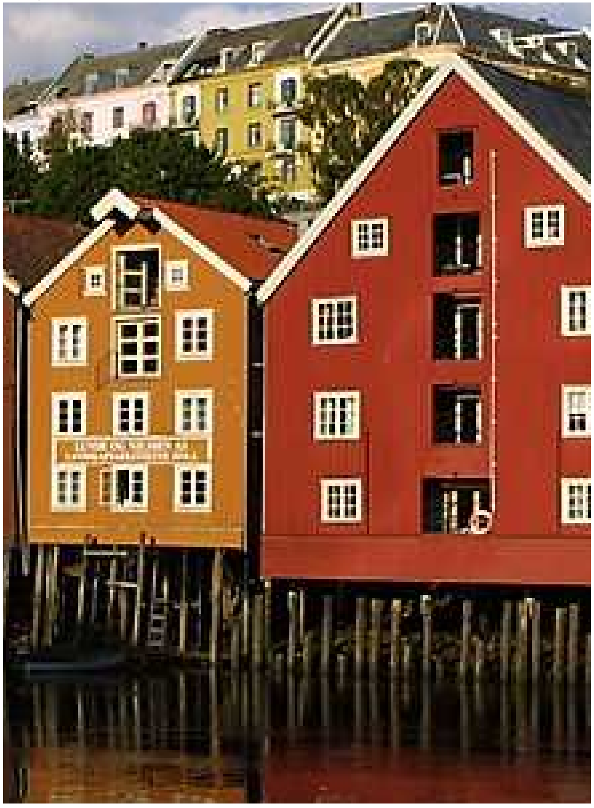

Figure 3.5: Image with JPEG quality 50. Compression ratio: 0.17

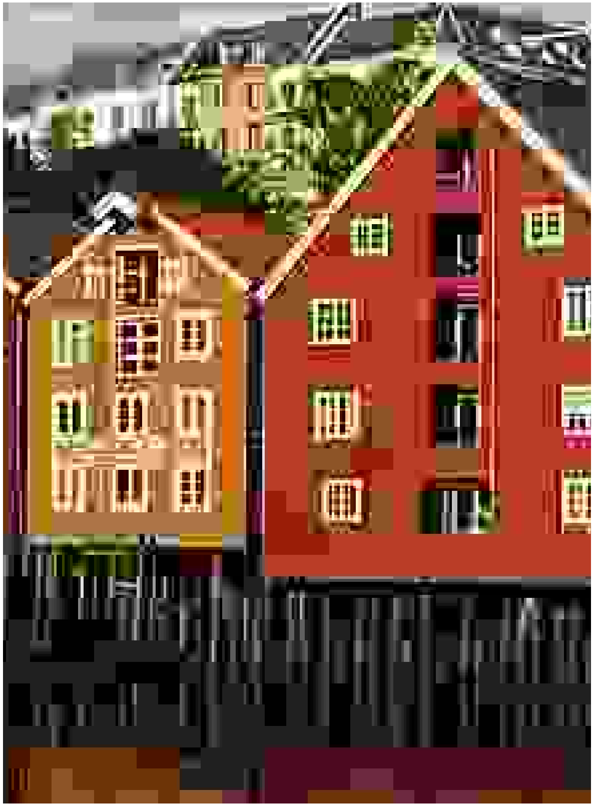

Figure 3.6: Image with JPEG quality 5. Compression ratio: 0.02

be spread over all frequency coefficients. But this is precisely what makes efficient encoding possible, since with the DCT method, there will be long runs of zeros and high-frequency values will be small. It has also been proven (see [30]) that the DCT is closer to the theoretically optimal Karhunen-Loève transform than the DFT. Therefore, the DCT is the preferred transform method. The DCT is also a *real transform*, and only uses real numbers, which is easier to encode than complex numbers. For further details on the DCT versus the DFT, see [31].

The DST is inferior to the DCT because it has no DC component. When we compute for $f = 0$ in Definition 1, all cosine terms will evaluate to 1 since $\cos(0) = 1$. If we used sines, then since $\sin(0) = 0$, the result would always be zero. Therefore, with the DCT, a constant signal will be represented as a single DC component, while the DST will represent the signal using a large number of AC coefficients. This is not good when, for instance, compressing an image consisting of large portions with similar color values — a property carried by most natural images. In this case, the DCT will give less quality reduction for the same compression ratio as the DFT or DST.

However, the DCT has certain problems, primarily related to the blocking effects. Since each block is transformed individually, the blocks' boundaries can become visible and severely degrade quality — particularly when using high quantization levels (i.e. small compression ratios). This is clearly seen in Figure 3.6. In addition, DCT methods usually emphasize low frequencies, while many types of data has most energy concentrated in the middle frequencies and carry almost no information in the low frequencies. We will these problems further in Section 3.3.

Also note that when we use a blocked DCT, it is not certain that we can get all the original data back even if we use no quantization. If we use $k \times k$ blocks on an $N \times N$ image where $k < N$, there might exist spatial frequencies in the image which can not be represented by the k^2 cosine basis functions. However, with $k = 8$, this turns out to be insignificant for most practical purposes [28].

3.2.4 Computing the DCT and the IDCT

It is apparent that we need efficient ways of calculating the 2D DCT in order to use it for compression. We will focus on doing a 2D, 8×8 DCT, which is the most frequently used DCT in data compression. There are several approaches to calculating the DCT. In this section, we will describe some of them.

By generalizing Definition 1 to two dimensions (as described in Section 3.2.4), we obtain the following expression for the 2D DCT Y_{ij} of an $n \times m$ data set p_{xy} $(x, i = 0, \ldots, n-1; y, j = 0, \ldots, m-1)$:

$$Y_{ij} = \frac{2}{\sqrt{nm}} C_i C_j \sum_{x=0}^{n-1} \sum_{y=0}^{m-1} p_{xy} \cos\left[\frac{(2y+1)j\pi}{2m}\right] \cos\left[\frac{(2x+1)i\pi}{2n}\right] \tag{3.4}$$

Similarly, we get the following expression for the IDCT:

$$p_{xy} = \frac{2}{\sqrt{nm}} \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} C_i C_j Y_{ij} \cos\left[\frac{(2y+1)j\pi}{2m}\right] \cos\left[\frac{(2x+1)i\pi}{2n}\right] \tag{3.5}$$

We will now look at some ways of calculating these expressions for 8 × 8 blocks, and compare their efficiency.

Direct computation

The straightforward and naive method of computing Equations 3.4 and 3.5 is to implement the formula directly, giving asymptotic running time $\Theta(n^2m^2)$ since there are $\Theta(nm)$ elements which each take $\Theta(nm)$ time to compute. In general, the number of multiplications used will be $n^2m^2 + nm$ (including final scalings by $2/\sqrt{nm}$), and the number of additions will be $nm(nm-1)$ (since adding nm terms requires $nm-1$ addition operations).

By exploiting the fact that the DCT, as the DFT, is separable, the number of operations used can be drastically reduced. In this case, we first compute a 1D DCT on each row of the data set, followed by a 1D DCT on each column of the data set. This reduces the asymptotic running time to $\Theta(nm^2 + n^2m) = \Theta(\max(n,m)^2 \min(n,m))$. The number of multiplications will be $nm^2 + n^2m + nm$, while the number of additions will be $n(m-1) + m(n-1)$.

However, this is not quick enough. Even when using separability, a DCT of an 8 × 8 block will require 1200 additions and multiplications, which is too much, considering that images often are of sizes above 1000 × 1000. Fortunately, there are better methods.

Using the DFT/FFT or FFT-like methods

Large amounts of research has gone into calculating the discrete Fourier transform using the Fast Fourier Transform (FFT) algorithm; see for example [32, 33, 34]. The first FFT implementation was done by Cooley and Tukey in 1965 [35], and quickly afterwards, research on the FFT algorithm surged [36]. The DCT can be calculated from the FFT in addition to a $\Theta(n)$ transformation on the input data [37]. Using this approach yields a time complexity of order $\Theta(N \log_2 N)$. The famous FFTW library [26] for uses this approach to calculate the DCT.

But even quicker DCT/IDCT computations can be obtained by using algorthms tailored to compute the DCT. These algorithms are usually derived from the FFT. We now describe how the three quickest known algorithms known for computing an 8 × 8 DCT work.

The AAN Algorithm The AAN algorithm (named after the inventors Arai, Agui and Nakajima) [38] computes a 1D 8-point DCT. It relies on the fact that we may obtain an N-point DCT may be obtained from a $2N$-point DFT. In particular, consider the definition of the K-point Discrete Fourier Transform $F(u)$ of an N-point sequence $s(x)$ of *real* numbers in Equation 3.6 below.

$$F(u) \quad = \quad \sum_{x=0}^{K-1} s(x) e^{-j2\pi ux/K} \tag{3.6}$$

Next, extend $s(x)$ symetrically about $(2N-1)/2$; that is, let $s(x) = s(2N-x-1)$ for $x = N, \ldots, 2N-1$. With this extension, [39] proves that if $F(u)$ is the $2N$-point DFT of $s(x)$, then for $u = 0, \ldots, N-1$ we have

$$\frac{\Re(F(u))}{2\cos\left(\frac{\pi u}{2N}\right)} = \sum_{x=0}^{N-1} s(x)\cos\left(\frac{(2x+1)\,u\pi}{2N}\right) \tag{3.7}$$

The right side of Equation 3.7 is the (unnormalized) DCT coefficient of Definition 1, while the left side is the real part of the $2N$-point DFT scaled by a constant. Thus this equation establishes that we may obtain an N-point DCT by scaling the real part of the first N coefficients of a $2N$-point DFT.

This implies that in order to calculate an N-point DCT, we need a $2N$-point DFT. An optimal form of a 16-point DFT has been given in [40]. But [38] noted that when computing the DCT, only the real parts of the coefficients are used, and the DFT will always be symmetrical, which implies that the final eight coefficients will be equal to the first eight coefficients. Due to this, parts of the computations in [40] can be avoided! This method of computing the 8-point DCT is known as the AAN algorithm, and it is one of the quickest methods of calculating the DCT. It is used by the popular JPEG library made by the Inpdendent JPEG Group[3].

We now describe precisely how the AAN algorithm works. Consider the graph in Figure 3.7 on the following page, which is a modified version of a similar flowgraph appearing in [28]. This flowgraph illustrates the AAN algorithm.

In Figure 3.7, the DCT is calculated by following the flowgraph from left to right. Nodes merging two edges (black circles) indicate addition, boxes indicate multiplication by the factor specified inside the box, and arrows indicate negation. As is evident by the figure, the AAN algorithm uses only 13 multiplications. Five of these — the a_i multiplications — are necessary to actually compute the DCT, while the other eight — the s_i multiplications — are just present for scaling and normalizing the outputs. In data compression applications, the quantization step can be incorporated into the final eight multiplications, thus incorporating the quantization phase into the transform calculation.

From Figure 3.7, equations for calculating the DCT and IDCT, which is obtained by simply following the flowgraph in the opposite direction[4], are easily derived. The a_i scale factors are[5]

$$a_1 = \frac{\sqrt{2}}{2} \tag{3.8}$$

$$a_2 = \sqrt{2}\cos\left(\frac{3\pi}{8}\right) \tag{3.9}$$

$$a_3 = a_1 = \frac{\sqrt{2}}{2} \tag{3.10}$$

$$a_4 = \sqrt{2}\cos\left(\frac{\pi}{8}\right) \tag{3.11}$$

$$a_5 = \cos\left(\frac{3\pi}{8}\right) \tag{3.12}$$

[3] URL: `http://www.jig.org`

[4] This is possible since the DCT is an orthogonal transform; i.e. the transform matrix is orthogonal. From this property, it can easily be proven that reversing the direction in simple flowgraphs (using only addition, negation and multiplication) is equivalent to multiplying with the inverse transform matrix. Orthogonality also provides good numerical stability.

[5] See [38] for a derivation.

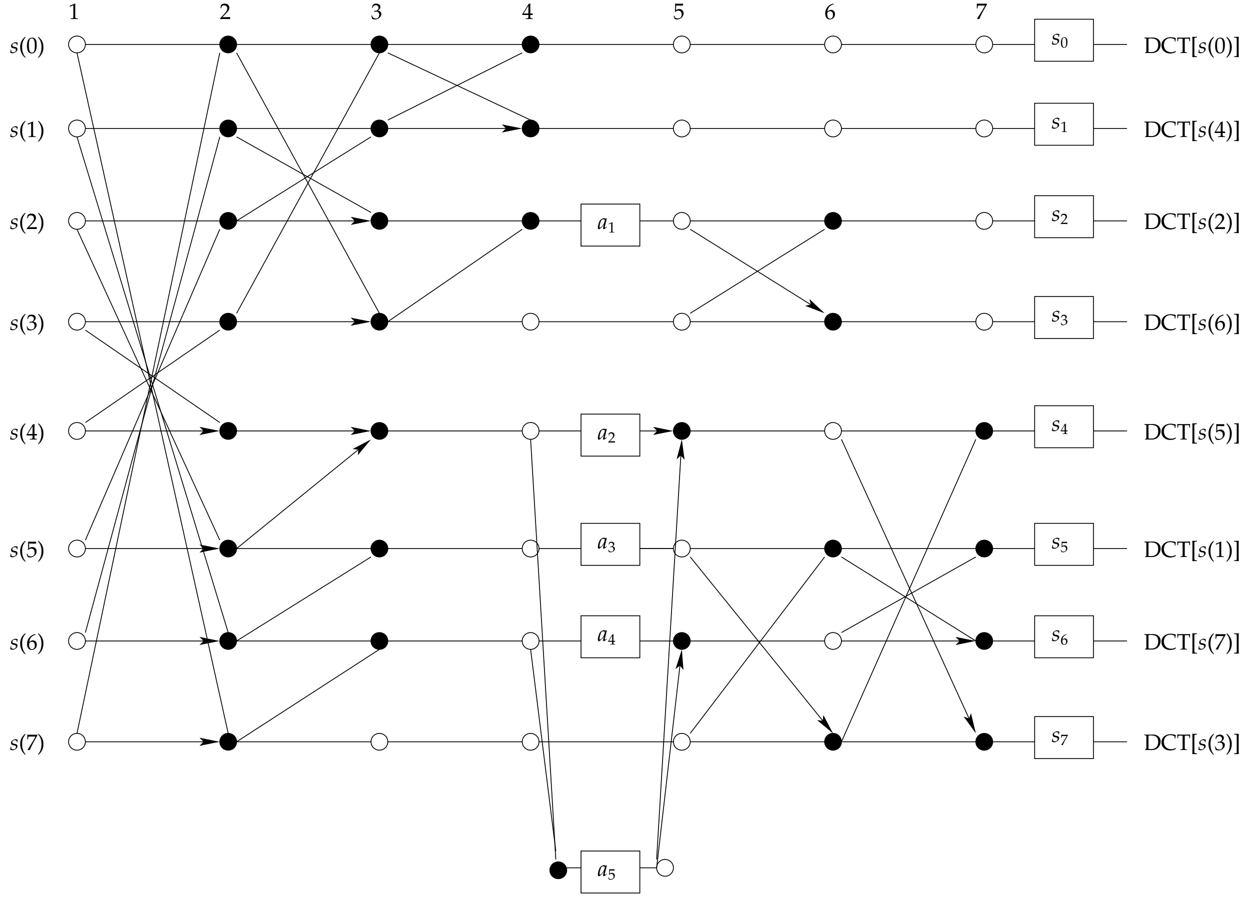

Figure 3.7: Flowgraph for the AAN DCT algorithm, based on Pennebaker and Mitchell's book [28] and Arai, Agui and Nakajima's original article [38]

The s_i factors, which are required to convert the DFT coefficients to DCT coefficients and to normalize the result, are derived as follows. By letting $N = 8$ and multiplying Equation 3.7 with the normalization factor $\sqrt{2/N}$ from Definition 1 we obtain

$$\frac{\sqrt{2}\Re(F(0))}{4} = \frac{1}{2\sqrt{2}} \sum_{x=0}^{N-1} s(x) \qquad (u = 0) \qquad (3.13)$$

$$\frac{\Re(F(u))}{4\cos\left(\frac{\pi u}{2N}\right)} = \frac{1}{2} \sum_{x=0}^{N-1} s(x) \cos\left(\frac{(2x+1)\,u\pi}{2N}\right) \qquad (u = 1,\ldots,7) \qquad (3.14)$$

The right sides of the two above equations are the DCT coefficients. Prior to the final scaling stage (i.e. subsequent to stage 7 in Figure 3.7), the calculated values are the real parts of the DFT coefficients; i.e. $\Re(F(u))$ in Equations 3.13 and 3.14. From this we conclude that $s_0 = \sqrt{2}/4$ and $s_i = 1/\left(4\cos(i\pi/16)\right)$ for $i = 1,\ldots,7$. These scaling factors can be precomputed.

The AAN algorithm has been implemented both on the CPU (both single and multi-core versions) and GPU in this project, and we will look closer at the implementation in the next chapter. Note that in order to apply the AAN algorithm of an 8 × 8 block, we use the separability property of the DCT discussed in Section 3.2.4.

The Feig-Linzer Algorithm for multiply-add architectures Feig and Linzer introduced an algorithm [41] for calculating the 1D DCT of 8 points, which is tailored to architectures having combined multiply-add (MADD) instructions (i.e. instructions calculating $AB + C$ for inputs A, B and C), which should suit recent GPUs well. Recall from the previous chapter (see Figure 2.4 on page 9) that recent NVIDIA GeForce 7 GPUs can perform several vectorized multiply-add instructions in parallel. The GPU is therefore a multiply-add architecture. This algorithm uses more multiplications than the AAN algorithm, but requires fewer additions. The total number of floating point operations required is therefore less than that of the AAN algorithm. This algorithm has been implemented both on the CPU and the GPU in this project.

We will briefly describe how this algorithm can be implemented; for further details, see [41]. The Feig-Linzer multiply-add algorithm expresses the 8-point 1D DCT $\mathbf{y} = [y_0\ y_1\ \cdots\ y_7]^T$ of $\mathbf{x} = [x_0\ \cdots\ x_7]^T$ as a set of linear equations

$$\mathbf{y} = \mathbf{C}\mathbf{x} \tag{3.15}$$

where $\mathbf{C}$ is a 8×8 matrix expressed in terms of the 8×8 orthogonal matrices $\mathbf{D}$, $\mathbf{A}_3$, $\mathbf{A}_2$, $\mathbf{A}_1$ and $\mathbf{A}_0$:

$$\mathbf{C} = \mathbf{D}\mathbf{A}_3\mathbf{A}_2\mathbf{A}_1\mathbf{A}_0 \tag{3.16}$$

$$\mathbf{A}_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad \mathbf{A}_1 = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \end{bmatrix},$$

$$\mathbf{A}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & c_4 \\ 0 & 0 & 0 & 0 & 0 & c_4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -c_4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -c_4 \end{bmatrix}, \quad \mathbf{A}_3 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & t_1 & 0 & 0 \\ 0 & 0 & 1 & t_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & t_5 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & t_5 & 1 \\ 0 & 0 & t_2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & t_1 & -1 & 0 & 0 \end{bmatrix},$$

$$\mathbf{D} = \frac{1}{2}\operatorname{diag}(c_4, c_1, c_2, c_5, c_4, c_5, c_2, c_1),$$

where $c_j = \cos(j\pi/16)$ and $t_j = \tan(j\pi/16)$. Multiplying a vector by $\mathbf{A}_0$, $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$ and $\mathbf{D}$ can be, respectively, done with 8 additions, 6 additions, 4 multiply-add operations, 8 multiply-add operations (of which two are only additions) and 8 multiplications. The multiplication of $\mathbf{A}_2$ can be reduced to two multiplications and four additions, but as we shall discuss in the next chapter, this might not be a good idea. A flowgraph of the Feig-Linzer multiply-add algorithm is shown in Figure 3.8, which was obtained by multiplying the matrices by a vector and writing down the resulting equations. Notice the similarity to Figure 3.7.

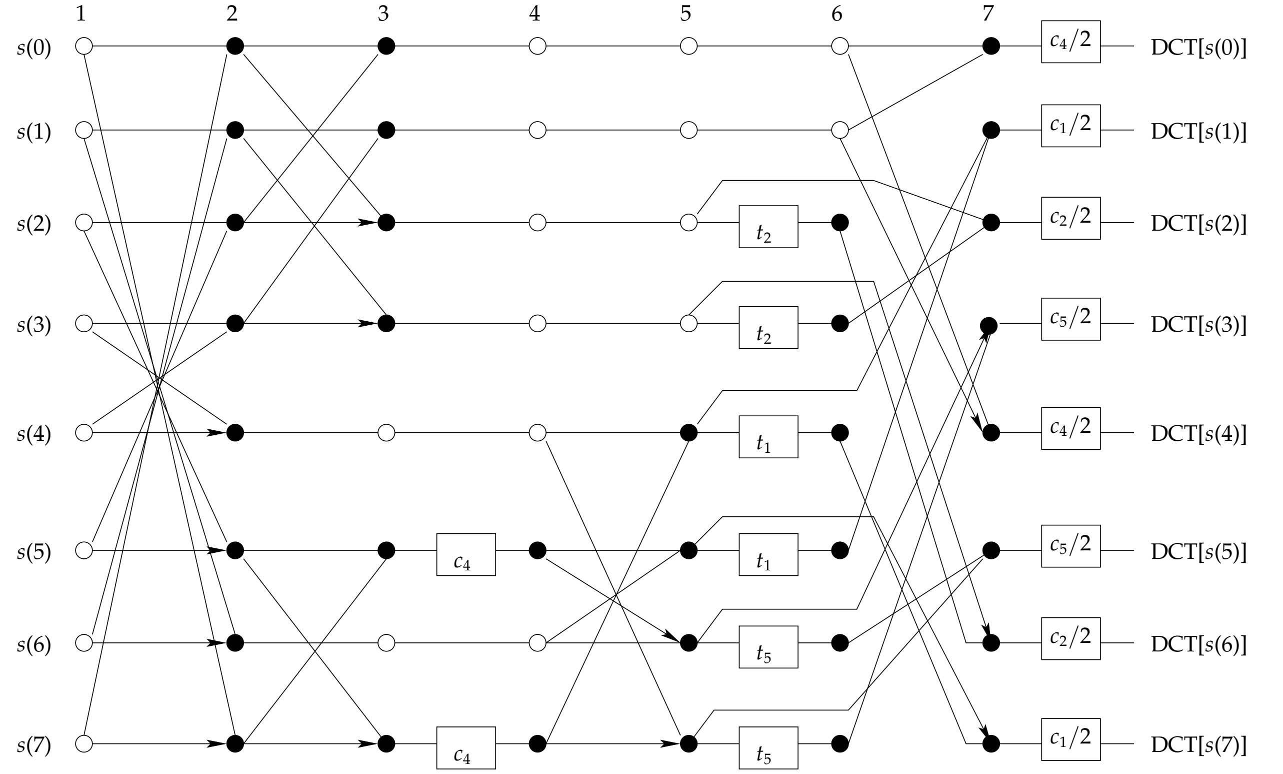

Figure 3.8: Flowgraph for the Feig-Linzer multiply-add DCT algorithm

To calculate the IDCT $\mathbf{x}$ of $\mathbf{y}$, we can calculate the product

$$\mathbf{x} = \mathbf{C^{-1}y} \tag{3.17}$$

$$\mathbf{C}^{-1} = \mathbf{A}_{0T}\mathbf{A}_1\mathbf{A}_{2T}\mathbf{A}_3^T\mathbf{D}, \tag{3.18}$$

where $\mathbf{D}$, $\mathbf{A}_1$ and $\mathbf{A}_3$ are as before and $\mathbf{A}_{0T}$ and $\mathbf{A}_{2T}$ are defined in [41] as

$$\mathbf{A}_{0T} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & c_4 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -c_4 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & c_4 \\ 0 & 1 & 0 & 0 & 0 & -c_4 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{A}_{2T} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \end{bmatrix}.$$

As with the AAN algorithm, the arithmetic complexity of the IDCT is the same as that of the DCT. Also, to apply the algorithm to a 2D 8 × 8 block, the separability property discussed in Section is used (as with the AAN algorithm).

This algorithm has been implemented on both the CPU and GPU in this project. We consider the implementation in the next chapter.

The Optimal Feig-Linzer Algorithm The AAN and Feig-Linzer multiply-add algorithm are not the optimal methods for 2D 8 × 8 DCTs in terms of arithmetic complexity. Since these algorithms only compute 1D 8-point DCTs, they must be implemented separably to calculate the DCT of a two-dimensional data set. It turns out that approaches that are tailored to two-dimensional inputs reduce the total arithmetic complexity. The algorithm with the lowest arithmetic complexity for calculating a 2D 8 × 8 DCT is the *optimal Feig-Linzer algorithm*[6] [42], which expresses the 2D transform as a series of tensor products. By manipulating the order in which the tensor products are calculated, several multiplications and additions are eliminated. This leads to a reduction in arithmetic complexity compared to the AAN and Feig-Linzer multiply-add algorithms. This approach has proven to improve performance in video compression applications employing the DCT [43].

But even though the optimal Feig-Linzer algorithm is the currently optimal algorithm in terms of arithmetic complexity, it might not be the optimal algorithm in practice. We discuss the reasons for this in Section 3.2.5 below.

Integer methods

Algorithms have been developed to calculate the DCT using integer computation [44]. Such algorithms usually introduce rounding errors in the result, and there is certainly no advantage (rather, there is a disadvantage) to using integer instead of floating point computation on the GPU. Since it is more fair to compare GPU floating point performance to CPU floating point performance and not CPU integer performance, and since floating point versions are most used in practice (e.g. in libraries for JPEG and MP3 encoding and decoding), we will not consider such algorithms further in this report and focus on floating point DCT algorithms.

3.2.5 Comparisons of DCT methods

Let us look at how the methods we discussed in the previous section compare. Consider Table 3.1 on the next page, which shows the approximate[7] arithmetic complexity of each of the DCT/IDCT algorithms we have discussed when applied to an 8 × 8 input data set. The numbers *include* any normalization operations. The table shows the total number of multiplications and additions used (floating-point operations), and also how many multiplications and additions may be fused together in one multiply-add instructions. These numbers are only theoretical; in practice, the compiler might decide not to use fused instructions for all multiplications followed by additions.

The FFTW library complexity was obtained by creating a plan for a 2D 8 × 8 DCT and using the `FFTW_MEASURE` parameter, which executes all several FFTW algorithms and selects the quickest algorithm.[8]

[6] Note that this is a completely different algorithm than the Feig-Linzer multiply-add algorithm.

[7] The real arithmetic complexity depends on the underlying architecture and compiler used.

[8] This was done on both an Intel Core 2 Duo 2.13 GHz and an Intel Xeon 5160 3.0 GHz, running Linux and FFTW 3.1.2.

Algorithm	Multipl.	Additions	Multiply-adds	FL ops.
Direct separable	976	0	112	1200
FFTW library	224	320	96	736
AAN separable	208	464	0	672
FL multiply-add	64	256	160	640
FL optimal	94	432	30	586

Table 3.1: Arithmetic complexity of methods for obtaining the DCT of an 8 × 8 input. Numbers for the optimal FL algorithm were obtained from [28].

Although, as the FFTW authors note in [34], arithmetic complexity might not be the most important factor in choosing what algorithm to use, it is still of some importance. In terms of arithmetic complexity, the optimal Feig-Linzer 2D DCT algorithm is clearly the winner, since it uses the least floating point operations[9]. The AAN algorithm and the Feig-Linzer multiply-add algorithms have fairly similar arithmetic complexity. AAN uses less multiplications than FL multiply-add, but in turn uses more additions. The Feig-Linzer multiply-add algorithm will likely have an advantage on the GPU since, as mentioned earlier, most multiplications can be fused with additions, which suits GPU architecture much better than performing separate multiplications and additions. The FFTW library uses more multiplications than AAN and FL multiply-add; this is probably due to FFTW using FFT subroutines directly to calculate the DCT, which requires the input to be modified prior to computation. It is also necessary to normalize the output from the FFTW library in order to obtain the DCT we defined in Definition 1. But even if we exclude the normalization operations from the FFTW library complexity in Table 3.1, the arithmetic complexity is still greater than the AAN and Feig-Linzer multiply-add algorithms (even if normalization operations are *not* excluded from the arithmetic complexity of these algorithms).

Most of the algorithms discussed can be implemented on the GPU. The AAN and Feig-Linzer multiply-add algorithms have been implemented on the GPU in this project, and we discuss the implementations in the next chapter. Although it would be interesting to implement the optimal Feig-Linzer DCT, it is not very well suited for GPU implementation. The problem is that an efficient implementation of the optimal Feig-Linzer DCT algorithm would have to consider the entire 8 × 8 block at once, while only 8 elements are considererd at a time when doing the AAN and Feig-Linzer multiply-add algorithms. It is currently not possible for a fragment processor to output 64 elements at one time, since each invocation of a fragment program can only output one pixel value (4 floating point values) to maximum 4 render targets, which gives a total of 16 values. Thus to implement the optimal Feig-Linzer DCT algorithm, blocks would have to be split up, and there would have to be branching instructions (or equivalent computations) in the fragment processor programs to determine what part of a block it is going to compute. This would destroy the algorithmical advantage of the optimal Feig-Linzer DCT algorithm.

The other algorithms only process 8 elements at a time, which makes branching unnecessary, since all 8 transform values can be written directly to GPU texture memory. In practice, the optimal Feig-Linzer algorithm is not usually used, since the other algorithms are usually just as quick in practice. For instance, the Independent JPEG Group's open source JPEG library uses the AAN algorithm. Finally, and most importantly, this algorithm is of no use when computing more advanced, superior transforms such as the LOT in the next section, since computing the LOT requires computing 1D DCTs.

[9] Note that multiplications and additions usually take the same amount of time on both GPUs and modern CPUs.

For these reasons, we have implemented the AAN and the Feig-Linzer multiply-add algorithms on the GPU, but not the optimal Feig-Linzer algorithm.

3.2.6 Previous DCT Implementations on the GPU

There have been two previously known DCT implementations on the GPU.

NVIDIA AAN Implementation

NVIDIA Corporation has available a DCT/IDCT implementation using the AAN algorithm[10]. However, it is not optimal (which is also stated in the source code) in three aspects:

1. *AAN implementation is not vectorized*. The NVIDIA code does not use any vectorized operations. Using vector instructions is highly important when using the GPU, since the GPU fragment processors *only* operate on four-component vectors. The GPU version of the AAN algorithm implemented in this project uses vector instructions to nearly the maximum possible extent.
2. *Does not use vertex processors*. The NVIDIA code only loads the fragment processors, and does not utilize the power of the vertex processors at all. There are not many uses for the vertex processors, since they deal with vertices instead of individual pixels, but the AAN version implemented in this project uses the vertex processors for a few calculations. Note that in the GeForce 8 series of GPUs, released when this report was in preparation, uses a unified shader architecture which uses just one processor core for both vertex, geometry and fragment processors. It is unclear whether using vertex programs will still be advantageous on this GPU.
3. *Does not use OpenGL Frame buffer objects (FBOs).* Previously, to read data from the GPU, so-called *pbuffers* had to be used. Pbuffers were not originally designed for GPGPU usage, and are also very slow and not fully supported on platforms other than Windows. The code in this project uses the relatively recent frame-buffer objects OpenGL extension, which is a method of transferring data to and from the GPU, which was created with GPGPU applications in mind. This method is quicker and more portable than pbuffers. (The reason the NVIDIA code uses pbuffers is likely because it is from 2004).

In addition, the Feig-Linzer multiply-algorithm should be superior to AAN on the GPU.

Microsoft Research Asia / Zheijiang University Implementation

The second DCT/IDCT implementation on the GPU has been done by researchers at Microsoft Research Asia and Zheijiang University in China, who in [45] argue that entirely new DCT/IDCT algorithms must be developed for the GPU, because current algorithms are optimized for the CPU's cache. The paper presents GPU DCT implementations based on plain matrix multiplication, which is essentially the same algorithm as the naive separable DCT algorithm (see Table 3.1). Microsoft Corporation has a patent pending on this work in U.S. Patent Application number 20060204119[11], where an elaboration on the choice of plain matrix multiplication can be found:

[10] Available from `http://download.developer.nvidia.com/developer/SDK/Individual_Samples/gpgpu_samples.html`.

[11] Note that the work in this project in no way uses methods from the patent application. We use entirely different methods, and have not considered implementing DCT (or the LOT) using plain matrix multiplication.

> *[0041] It is desirable to use fast algorithms or techniques instead of direct matrix multiplication for implementing a DCT/IDCT on a GPU. On the other hand, a GPU is generally considered a stream processor and its internal processing engine is highly pipelined. As a result, the programming model of GPUs may be somewhat limited, mainly due to the lack of random access writes. Moreover, each pixel (and correspondingly the internal graphics engine) can have only up to four channels, for example, which implies at most four coefficients can be determined a time. With a GPU, the cost of addition, multiplication, multiplication and addition (MAD) and dot product of up to 4-element vectors (dp4) is the same. Moreover, the GPU is intrinsically a SIMD (single instruction, multiple data) machine and the spatial data parallelism can be efficiently exploited. Based on these factors, it is desirable to implement the DCT/IDCT by direct matrix multiplication.*

The methods presented in the paper (and patent application) are unable to outperform the FFT-based SIMD-optimized methods on the single-core CPUs (see [45]). There is some room for improvements to their method. *First*, the naive separable DCT algorithm is slow, while the CPU algorithms which the paper compares the GPU algorithms with are FFT-methods. Although the 4-element dot product is used, which costs about the same as a multiply-add operation on the GPU, more instructions are still needed compared to the AAN or FL multiply-add algorithms. *Second*, and probably more importantly: When plain matrix multiplications are used, the fragment programs must in addition to input data also fetch matrix coefficients from texture memory. The coefficients are stored as separate textures. This increases the number of texture memory accesses *and* could decrease the locality of the memory accesses. This is unnecessary with the AAN or FL multiply-add methods implemented in this project. Thus, these methods should be able to outperform the methods in [45]. There is no reason why a naive DCT algorithm based on matrix multiplication should be quicker than FFT-based methods on the GPU — in fact, it has been demonstrated in [6] that the FFT (which AAN and FL multiply-add are based on) perform very well on the GPU.

As we shall see, we have successfully managed to implement quicker DCT algorithms — where each fragment processor processes 8 coefficients at a time — and more advanced transforms (the LOT) on the GPU while at the same time exploiting pipelining and the SIMD architecture of the GPU. However, note that our implementations are specialized for 8×8 DCTs, while the Microsoft implementation generalizes to other block sizes more easily.

3.3 The Lapped Orthogonal Transform (LOT)

As we mentioned in the previous section, the DCT has certain disadvantages. We will in this section describe the *lapped orthogonal transform* [46, 47, 30], or the LOT, which attempts to solve the blocking-problem of the DCT. The LOT and an efficient algorithm for LOT computation was invented by Henrique S. Malvar in the 1980s. It has been shown that the LOT gives both objectively and subjectively better compression results than using the DCT by itself [30]. As we did for the DCT in the previous section, we will first give a description of the LOT transform in Section 3.3.1. Then, we will describe some properties of the transform and a fast algorithm for computing the transform in Sections 3.3.2 and 3.3.3, respectively. Finally, we look at the arithmetic complexity of the fast LOT algorithm in Section 3.3.4.

3.3.1 Description of the LOT

The lapped orthogonal transform[12] is defined as follows [46, 30]. Consider $M > 1$ blocks of $N > 0$ sample values, i.e. MN sample values $\mathbf{x} = [x_0 \ \cdots \ x_{MN-1}]^T$. The transformed values $\mathbf{y} = [y_0 \ \cdots \ y_{MN-1}]^T$ may be written as a matrix-vector product

$$\begin{aligned} \mathbf{y} &= \mathbf{T}\mathbf{x} && (3.19) \\ \mathbf{T} &= \begin{bmatrix} \mathbf{P}_1 & & & \mathbf{0} \\ & \mathbf{P}_0 & & \\ & & \ddots & \\ \mathbf{0} & & & \mathbf{P}_2 \end{bmatrix}. && (3.20) \end{aligned}$$

$\mathbf{T}$ the $MN \times MN$ transform matrix containing the LOT basis functions (just as $\mathbf{C}$ is the transform matrix containing the DCT basis functions in Equation 3.15). $\mathbf{P}_0$ is an $L \times N$ matrix with the LOT basis functions for each block. L is the number of sample values in each block plus the number of overlapping sample values used, such that each block overlaps other blocks with $L - N$ sample values. $\mathbf{P}_1$ and $\mathbf{P}_2$ are matrices based on $\mathbf{P}_0$, but are adjusted since the first and last blocks only have one neighbouring block.

Now we choose $L = 2N$ so that each block overlaps neighbouring blocks by $2N - N = N$ sample values and $\mathbf{P}_0$ is of size $2N \times N$. One choice for $\mathbf{P}_0$ that works well is

$$\mathbf{P}_0 = \mathbf{P}\mathbf{Z} \qquad (3.21)$$

where

$$\mathbf{P} = \frac{1}{2}\begin{bmatrix} \mathbf{D}_e - \mathbf{D}_o & \mathbf{D}_e - \mathbf{D}_o \\ \mathbf{J}(\mathbf{D}_e - \mathbf{D}_o) & \mathbf{J}(\mathbf{D}_o - \mathbf{D}_e) \end{bmatrix}. \qquad (3.22)$$

$$(3.23)$$

Here, $\mathbf{D}_e$ and $\mathbf{D}_o$ are, respectively, matrices of size $N \times N/2$ containing the even and odd DCT basis functions for the current block. That is, if we write the transform in Definition 1 as a matrix-vector product, but separate the even and odd DCT coefficients, we get the $\mathbf{D}_e$ and $\mathbf{D}_o$. $\mathbf{J}$ is the $N \times N/2$ counteridentity or anti-identity matrix, i.e. a matrix containing all zero values except values equal to 1 along its second diagonal (i.e. the diagonal going from the last row in the first column to the first row in the last column).

It can be shown [46] that a near-optimal $\mathbf{Z}$ matrix is given by

$$\mathbf{Z} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \tilde{\mathbf{Z}} \end{bmatrix}, \qquad (3.24)$$

[12] We are really describing one specific LOT. In general, it is possible to define other LOTs which are not based on the DCT and do slightly more computations to further improve energy compaction properties. However, these are of limited practical interest since they have no quick algorithms and do not give any significant advantages. The LOT we describe is near-optimal and has a fast algorithm.

where $\mathbf{I}$ is an $N/2$ identity matrix and $\widetilde{\mathbf{Z}}$ is an $N/2 \times N/2$ matrix approximated by a sequence of $N/2 - 1$ two-dimensional rotations $\mathbf{R}_i$:

$$\widetilde{\mathbf{Z}} = \mathbf{R}_1\mathbf{R}_2\mathbf{R}_3 \cdots \mathbf{R}_{N/2-1}, \tag{3.25}$$

where each $\mathbf{R}_i$ is an $N/2 \times N/2$ matrix containing regular plane rotation matrices shifted along the main diagonal by $i - 1$ places; that is,

$$\mathbf{R}_i = \begin{bmatrix} \mathbf{I} & & & & \mathbf{0} \\ & \mathbf{Y}(\theta_i) & & & \\ & & 1 & & \\ & & & \ddots & \\ \mathbf{0} & & & & 1 \end{bmatrix} \tag{3.26}$$

where the identity matrix in the top-left corner of $\mathbf{R}_i$ is of order $i - 1$. Finally, $\mathbf{Y}(\theta_i)$ is the familiar 2×2 plane rotation matrix:

$$\mathbf{Y}(\theta_i) = \begin{bmatrix} \cos\theta_i & \sin\theta_i \\ -\sin\theta_i & \cos\theta_i \end{bmatrix}. \tag{3.27}$$

The LOT is now defined, except that we also need to define the angles θ_i and the matrices $\mathbf{P}_1$ and $\mathbf{P}_2$, which describe LOT computation along the first and last blocks. We will discuss both of these subjects in Section 3.3.3.

Fortunately, computing the LOT is not as complicated as it might seem, and a fast algorithm for it exists, which we will also describe in Section 3.3.3. This algorithm uses an N-point 1D DCT, which suits us well, since we have just derived several algorithms for computing 8-point DCTs.

Note that the LOT easily can be used for compression. Just replace the DCT/IDCT in Figure 3.1 on page 25 by the LOT/inverse LOT (*ILOT*, obtained by transposing the $\mathbf{T}$ matrix, since the transform is orthogonal). Also note that the LOT is, as the DCT, separable, so that it is possible to calculate the LOT for data sets of any dimension using a 1D LOT algorithm.

Now, let us look at some of the properties and extensions of the LOT.

3.3.2 LOT properties and extensions

The LOT significantly reduces blocking artifacts compared to the DCT. Consider Figure 3.9 below. In the figure, an arbitrary basis function is shown. When using the blocked DCT, the basis functions are chopped off at the block boundaries, since the length of the basis functions are equal to the length of each block. Therefore a blocking effect occurs, since the last samples of a block are unlikely to match the first samples of the next block. The LOT basis functions, however, extend into the neighbouring blocks and include the first samples of the neighbouring blocks, which gives a smoother transition between the block boundaries. With this approach, the tiles in the image in Figure 3.6 would be reduced. See [30] for comparisons of images compressed by the DCT and the LOT.

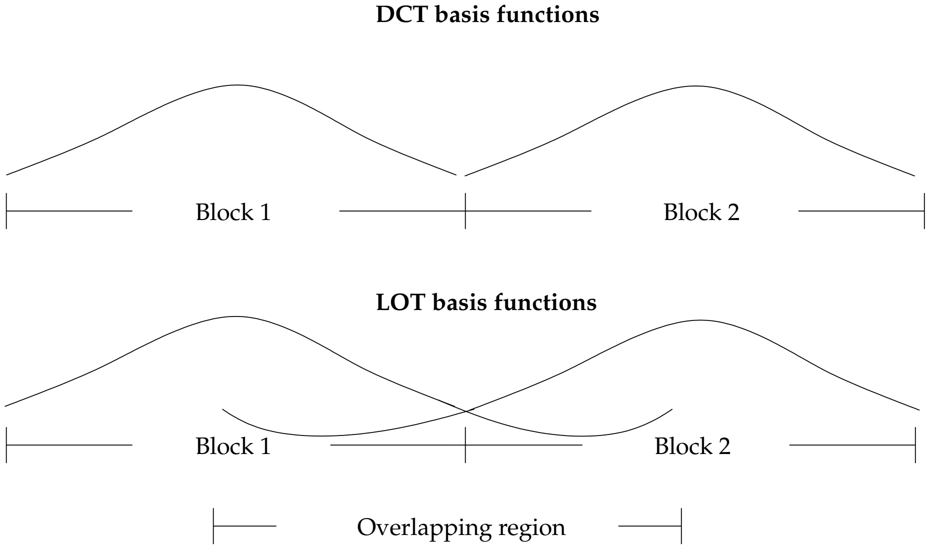

Figure 3.9: Illustration of the DCT and LOT basis functions at block boundaries, adapted from Malvar's book on the LOT [30]

There have been other attempts at solving the blocking problem of the DCT. One approach is to apply a smoothing filter before or after DCT computation, another approach is to use the infinite-length basis functions of the short-space Fourier transform (SSFT) [48]. These methods, however, either decrease attainable compression ratios or introduce new artifacts such as artificial rings in the image [30].

But there are several even more advanced methods based on the LOT, such as the *generalized lapped orthogonal transform* (GenLOT) [49] and *generalized unequal-length lapped orthogonal transform* (GULLOT) [50], which reduce the number of artifacts even further. These are among the most superior transform coding methods for image compression currently known. They are particularly well suited for seismological data [51, 52], which has greater requirements for accurate reconstruction than regular photographies. Applications running on clusters and executing calculations on such seismological data, which is often very large, will benefit from compressing the data with LOT-based methods when communicating with internal nodes and outside clients.

The GenLOT methods can be seen as a generalization of the LOT and the DCT. The DCT is the simplest GenLOT (an *order-1 GenLOT*), while the LOT is an *order-2 GenLOT*. Just as the LOT adds a step after the DCT, as we shall see in the next section, GenLOT methods of higher orders add several steps after computing the LOT to reduce artifacts occuring between a block and non-neighbouring blocks. Thus GenLOT corresponds to increasing the length of the basis functions in Figure 3.9 even further.

The GULLOT method solves problems related to spread of quantization error to neighbouring blocks when using GenLOT [50]. GULLOT is a generalization of GenLOT in which the basis functions may be on unequal length, as opposed to the GenLOT, LOT and DCT, where basis functions are always of fixed length. Thus GenLOT, LOT and DCT are all special cases of GULLOT. Thus, for instance, DCT may be viewed as a special case of an order-1 GULLOT where all the basis functions have the same length.

LOT-based methods are not the only solution to the blocking problem of the DCT. Using the discrete wavelet transform (DWT) reduces blocking problems and is used in the new JPEG-2000 standard. DWT-based methods have an advantage over DCT in that they not only give *what* frequencies occur in the image, but also detect precisely *where* in the image the frequencies are "located". This can be exploited so that high quantization levels are used in areas with no high frequencies, and lower quantization levels are used in areas with many details, thus reducing the blocking effects. But DWT-based methods have certain problems. Significant artifacts can still occur — for example, artificial rings may be introduced in the image. LOT-based methods have been proven to be superior for many types of images, both objectively (in terms of signal-to-noise ratio) and subjectively [53, 51].

Note that the theoretically optimal transform coding algorithm can be proven to be the Karhunen-Loève transform [30]. This transform, however, has no fast algorithm, and requires to be adjusted for the specific input sample values it is applied to. It has been shown that the methods discussed in this section are good approximations to this transform. The DCT is one good approximation, and the LOT is an even better approximation.

We will now look at an algorithm for computing the LOT using a block size of 8.

3.3.3 Computing the LOT and ILOT

Consider the flowgraph in Figure 3.10, which shows an implementation of the LOT described in Section 3.3.1 with $N = 8$. We will use the separability property of the LOT to compute the 2D 8×8 LOT of an image. The following algorithm, by Malvar [46, 30], is known as the Fast LOT algorithm, which computes a 1D LOT with a block size of 8. The algorithm described in this section is a *type-I fast LOT*. For larger block sizes (> 16), another algorithm, known as *type-II fast LOT* can be used.

In the figure, we show type-I fast LOT computation for two blocks in the middle of some data set $s(k)$. The flowgraph corresponds to multiplying the input data set by one of the $\mathbf{P}_0$ matrices). To obtain the inverse type-I fast LOT, we simply traverse the flowgraph from right to left instead of left to right — as with all other orthogonal transforms — and replace the DCT by the IDCT.

It is easy to show that the first part of the flowgraph, consisting of computing the DCT, executing the first merge or *butterfly* on the resulting DCT coefficients, and then executing the second butterfly and multiplying by $1/2$ corresponds to multiplying the input values by the $\mathbf{P}$ matrix of Section 3.3.1. Similarly, the final step is equivalent to multiplying by the $\mathbf{Z}$ matrix, since half of the first coefficients are multiplied by the identity matrix and the other half is multiplied by the $\widetilde{\mathbf{Z}}$ matrix. The flowgraph for multiplying by $\widetilde{\mathbf{Z}}$ — which is just a cascade of multiplications by $N/2 - 1$ (in this case, 3) planar rotation matrices — is shown in Figure 3.11. A single planar rotation matrix is shown in Figure 3.12.

The $\mathbf{Y}(\theta_i)$ rotation matrix in Figure 3.11 is shown as a flowgraph in Figure 3.12.

It can be proven [30] that approximate optimal angles for an 8-point LOT $\theta_1, \theta_2, \theta_3$ are given by

$$[\theta_1 \;\; \theta_2 \;\; \theta_3] \;\; = \;\; [0.13\pi \;\; 0.16\pi \;\; 0.13\pi]. \qquad (3.28)$$

The final question is how to compute the LOT for the entire data segment, i.e. how to define the $\mathbf{P}_1$ and $\mathbf{P}_2$ matrices of Section 3.3.1. There are several alternatives for extending the data set at the boundaries.

We have used a simple boundary extension scheme in our implementations. This scheme first extends the data set by $N = 8$ sample values both before the first and after the last

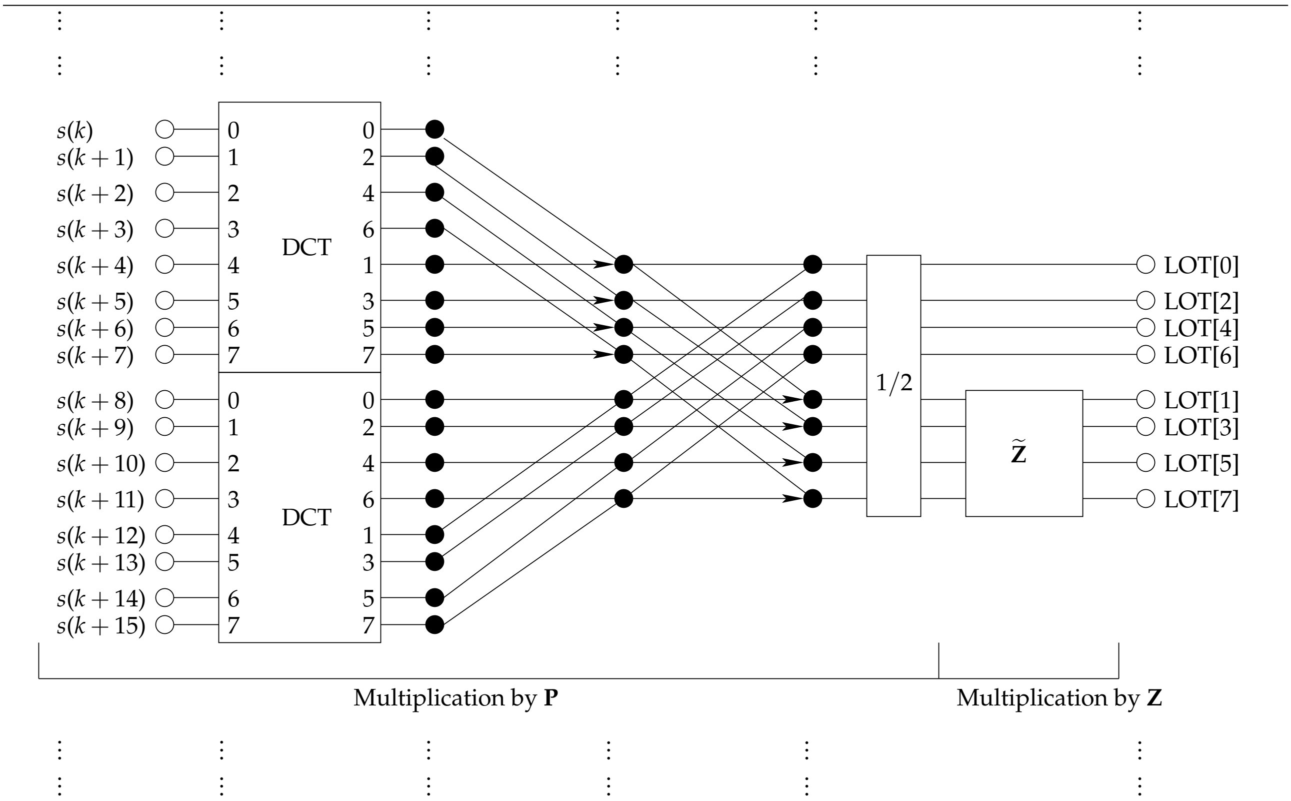

Figure 3.10: Flowgraph for the fast type-I LOT algorithm, with blocksize 8, adapted from Malvar's book on the LOT [30].

sample values. The new sample values are set to zero. When executing the transform, the DCT of the new first and last blocks will always evaluate to zero. Similarly, the first butterflies in Figure 3.10 for these blocks will also evaluate to zero. However, the second butterflies will introduce nonzero values in the, respectively, last and first halves of the new first and last blocks. These nonzero values are stored with the other transformed values, since not including them leads to artifacts along the boundaries. The image can then be restored without any artifacts by applying the inverse LOT transform.

Using this scheme implies that we need to store 8 additional values for each row and each column in an image in order to avoid artifacts around the image boundaries. For an image of size 1024 × 1024, these extra values lead to 1.5% extra values having to be stored after the transform. Therefore, the compression ratio will slightly increase. But the advantage of using this scheme is that the same operation can be done on each block. In other words, we do not need to treat the first and last blocks specially, as e.g. the scheme in [30] (implemented with no modifications) would require. Doing the same operation on all blocks and avoiding special cases is very advantageous on the GPU, since special cases often requires branching, which is slow.

In any event, boundary extensions are best suited to be done on the CPU, and the LOT implementations developed in this project can easily be used with certain more advanced schemes which would not require storing the extra values. For example, when using a symmetrical boundary extension, an application using our LOT implementation can extend the data set symmetrically before calling the LOT routines, and discard unneccessary extra values (which do not account for much of the total arithmetic complexity of the routine). In that way, storing the extra values is not necessary and only $8M$ values must be stored for an M-block data set (instead of $8(M+1)$ in our present implementation). Thus our

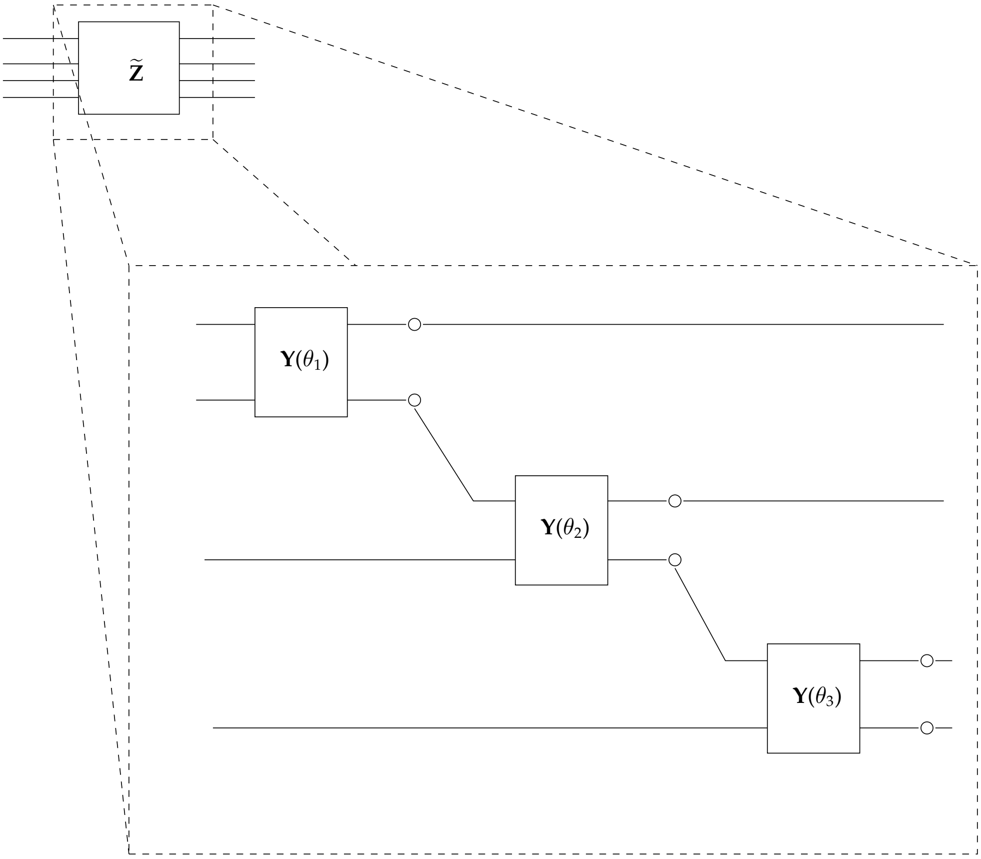

Figure 3.11: Flowgraph for the $\widetilde{\mathbf{Z}}$-block in Figure 3.10, based on a similar figure from Malvar's book and article [46, 30].

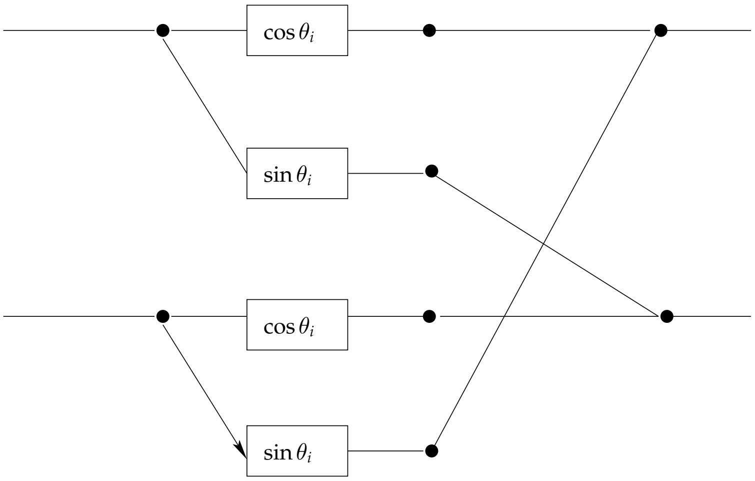

Figure 3.12: Flowgraph for the $\mathbf{Y}(\theta_i)$-matrix in Figure 3.11, based on a similar figure from Malvar's book and article [46, 30].

implementations can also be used with more advanced boundary extension schemes.

Figure 3.13 illustrates how the LOT is computed for an entire data set, along with our boundary extension scheme. Note that the section inside the dotted lines represents the operation in Figure 3.10. Also, notice that the first and last blocks are the extended blocks set to zero, and that the first and last halves of the first and last blocks, respectively, will always be zero, but not the other halves, which must be stored in order to avoid artifacts. But it is evident from the figure that setting the extra blocks to zero enables us to do exactly the same computation on all the blocks, which makes the LOT easier to implement on the GPU.

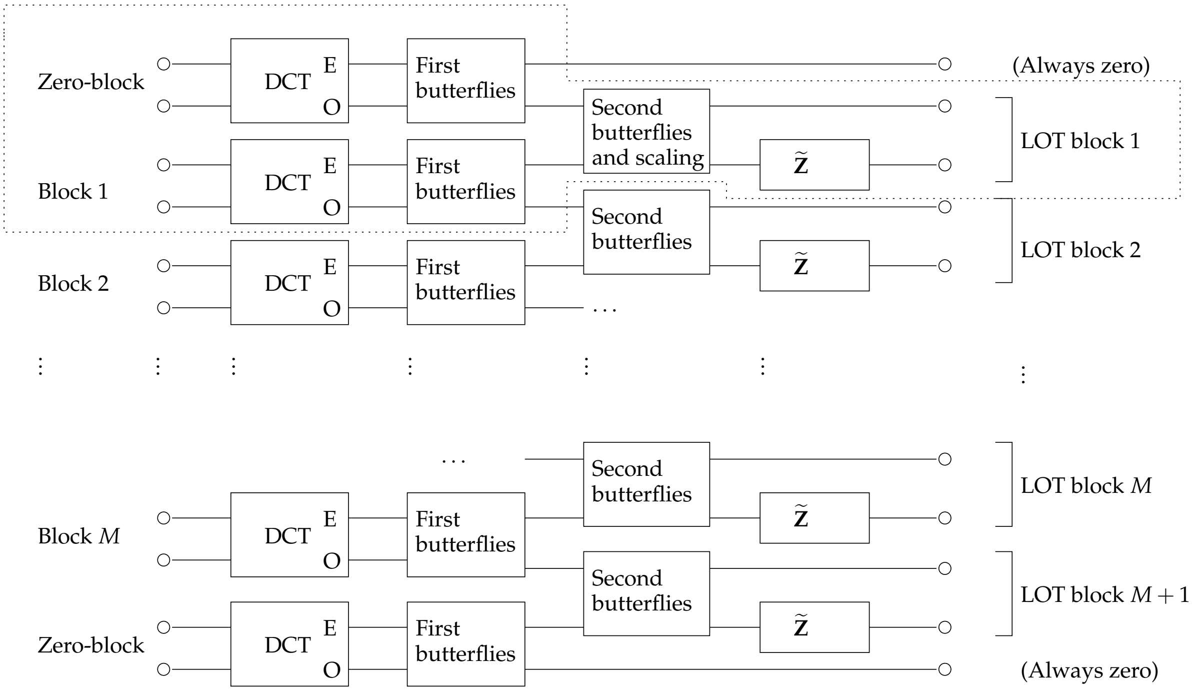

Figure 3.13: Computing the LOT for a data set, based on a similar figure in Malvar's book and article [30, 46]. The section inside dotted lines represent the steps of Figure 3.10. Note that the figure is just a sequence of several such steps.

3.3.4 Arithmetic Complexity of the LOT

The LOT transform has not received the same popularity as the DCT, primarily due to the LOT requiring more computations. Consider Table 3.2[13] which shows the arithmetic complexity of the DCT and the LOT of an 8 × 8 block when the DCT part of the LOT is implemented with the 1D 8-point DCT algorithm of the lowest arithmetic complexity on new machines (the Feig-Linzer multiply-add algorithm). The arithmetic complexity of the LOT for a single 8-point block is 8 multiplications for the 1/2 scalings, 6 multiplications for the $\widetilde{\mathbf{Z}}$-block, 16 additions/subtractions for the two butterflies, and 6 multiply-adds for the $\widetilde{\mathbf{Z}}$-block, in addition to the arithmetic complexity of the Feig-Linzer multiply-add DCT for 8 points.

Algorithm	**Multipl.**	**Additions/ subtractions**	**Multiply-adds**	**FL ops.**
FL multiply-add DCT	64	256	160	640
Fast LOT using FL multiply-add DCT	288	512	256	1312

Table 3.2: Arithmetic complexity of the DCT and the LOT on an 8 × 8 input

[13] A similar table for an 8-point 1D LOT occurs in [30]. However, that table assumes the AAN DCT algorithm is used, which is less optimal on machines where multiplications and additions take the same amount of time (which is true for all recent CPUs and GPUs). Also, the 1/2 scalings are not included.

From the table, we see that the LOT requires more than twice as much operations as the Feig-Linzer multiply-add DCT. This suggests that a GPU implementation will give good performance, and that the LOT is actually better suited for the GPU than the DCT, due to the higher arithmetic intensity than the DCT. Furthermore, all LOT steps except for the $\widetilde{\mathbf{Z}}$-block multiplication are suitable for vectorization (the first and second butterflies can be implemented using one vector instruction) — which also fits the GPU well.

The LOT described in the past sections has been implemented on both the CPU (both single and multi-core versions) and the GPU in this project. We look at the implementations in the next chapter.

3.3.5 Previous LOT Implementations on the GPU

No previous publically known implementations of the LOT on the GPU are known.

Chapter 4

Implementation

This chapter describes how the DCT and LOT transforms have been implemented on the CPU and GPU in this project. Implementing these transforms on the GPU requires special considerations, which we will discuss in this chapter. We look at CPU and GPU implementations in Sections 4.1 and 4.2, respectively.

A program has been developed for benchmarking various DCT and LOT algorithms. The program consists of about 16,000 lines of code, where 13000 lines are C code and 3000 lines are NVIDIA Cg code. An overview of the program is given in Appendix A.

The source code for the program developed can be found on the CD accompanying this report in the `dct` directory. We will show excerpts from the source code throughout this chapter and in the appendices.

4.1 CPU Implementations

The CPU implementations of the algorithms for the DCT/IDCT are in the files:

- `dct_cpu.c`: The direct separable DCT
- `dct_fftw.c`: The DCT through the FFTW library (both simple and advanced interface)
- `dct_cpu_aan.c`: The AAN algorithm
- `dct_cpu_flmad.c`: The FL multiply-add algorithm
- `dct_cpu_aan_openmp.c`: The AAN algorithm, OpenMP version
- `dct_cpu_flmad_openmp.c`: The FL multiply-add algorithm, OpenMP version

LOT/ILOT CPU implementations are in the files

- `lot_cpu.c`: Fast LOT
- `lot_cpu_openmp.c`: Fast LOT, OpenMP version

We will now briefly describe each implementation in turn.

4.1.1 Direct Separable DCT: `dct_cpu.c`

The direct separable DCT implementation is just an implementation of the equations in Definition 1 in the previous chapter. The algorithm first applies the equations to each row of the input block, and then to each column. This is done for all the blocks in the input. This algorithm is very slow, and is only included for completeness and checking the results of the other algorithms.

4.1.2 FFTW DCT: `dct_fftw.c`

The FFTW implementation is also quite simple, since the FFTW library does the actual computations.

FFTW uses *plans* to specify what computations are to be done, data set sizes, and what buffers should be used for input and output. There are two ways of using FFTW to compute a blocked DCT. Either, the application can create a plan for an 8 × 8 DCT, and split the input set into separate 8 × 8 blocks and run the FFTW plan on each block. This is referred to the "simple" interface of FFTW, and requires that the application copies blocks into and out of the buffers specified when creating the plan. This can induce some overhead. Alternatively, we may use FFTW's "advanced" interface, which allows for blocked transforms, to create an 8 × 8 blocked DCT of the entire data set. In that case, a single plan execution is required.

The steps for computing the DCT using FFTW can be summarized as follows.

1. Create an FFTW plan for the computation using `fftw_plan_r2r_2d` (simple interface) or `fftw_plan_many_r2r` (advanced interface). The transform we use is `FFTW_REDFT10` or `FFTW_REDFT01`, which is, respectively, the DCT or IDCT of the input. The time the planning func tion takes is not included in any timings, since it can use considerable amounts of time and does not actually compute any results. We also use the `FFTW_MEASURE` parameter, which executes several algorithms for calculating the specified transform and chooses the best one.
2. If executing the IDCT, scale inputs to FFTW according to Equations B.5 through B.8 in Section B.1 on page 83.
3. Execute the plan using `fftw_execute`. This does the actual computation.
4. If executing the forward DCT, scale outputs from FFTW according to Equations B.1 through B.4 in Section B.1 on page 83.
5. If the simple interface is used, steps 2–4 are done for all blocks. If not, they are only done once.

The number of multiplications, additions and fused multiplications and additions is also displayed. This number is not the same as the numbers in Table 3.1, since those numbers also include input/output scalings.

Our FFTW implementation supports using FFTW with threads, and enables them automatically if linked to a FFTW library compiled to support threads. We have, however, not benchmarked FFTW with threads, since often, FFTW will choose *not* to use threads even if thread usage is requested.

4.1.3 AAN DCT: `dct_cpu_aan.c`

The implementation of the AAN algorithm is based on Figure 3.7 on page 32. For each block in the input, we first apply the AAN to the rows and next to the columns. Explicit equations for the AAN DCT/IDCT are derived in [28], and our implementation is based directly on those equations. DCT and IDCT has been implemented. See Section **??** on page ?? for source code excerpts.

4.1.4 FL Multiply-Add DCT: `dct_cpu_flmad.c`

The implementation of the Feig-Linzer multiply-add algorithm is based on Figure 3.8 on page 34 and the matrix equations for the algorithm in the previous chapter. As with the AAN algorithm, we first apply Feig-Linzer multiply-add to each row and then to each column of all the blocks. Both DCT and IDCT has been implemented.

4.1.5 Fast LOT Implementation: `lot_cpu.c`

The fast LOT/ILOT implementations are based directly on Figures 3.10 on page 43 and 3.13 on page 45. The LOT is, as the DCT, a separable transform [30], so in order to compute an 8×8 2D LOT, we first compute the 8-point LOT in the x-direction and then the 8-point LOT in the y-direction. Similarly, computing the 2D 8×8 ILOT is done by computing a 1D ILOT in each direction.

The fast LOT requires a 1D DCT algorithm. We have chosen to use the Feig-Linzer multiply-add algorithm for DCT and IDCT due to its low computational complexity.

We described the boundary extension scheme in Section 3.3.3 on page 42. This is implemented as follows: When computing the forward LOT, if the initial input data set size is $N \times N$, the data set is extended to $(N + 16) \times (N + 16)$. The extra boundary values are initially set to zero. From Figure 3.13 on page 45, it is evident that the first *half* of the first extended block *and* the last half of the last extended block in each dimension will always be zero after the FLOT has been computed. The zero values are discarded after the forward LOT computation is done, so that only $(N + 8) \times (N + 8)$ values are stored. See Figure 4.1 on the following page.

Why do we allocate an $(N + 16) \times (N + 16)$ array when we only need an $(N + 8) \times (N + 8)$ array? The reason is that when executing the LOT on the GPU, letting the input data set be a set of full blocks (which it is not when we use an $(N + 8) \times (N + 8)$ array, since the first four values and the last four values will only be a "half" block), is advantageous. Allocating the extra blocks allows us to more easily apply exactly the same operation to all 8×8 blocks of the data set, which is very important on the GPU, since it does not deal with special cases along the boundaries very well.

While allocating an extra $(N + 16)^2 - (N + 8)^2 = 16N + 192$ values which will always be zero is not necessary on the CPU, the code that reads the input file is independent from the specific algorithm used, so we allocate the extended array on the CPU as well. But both on the CPU and GPU, we avoid doing the DCT and first butterfly on the first and last blocks in each dimension.

When computing the inverse LOT, we read in $(N + 8) \times (N + 8)$ values from a file and extend the data set to $(N + 16) \times (N + 16)$ for the same reasons as for the forward LOT (simpler addressing on the GPU). We also avoid computing the obviously redundant inverse first butterflies and IDCT on the first and last extended blocks. After computation, the extra values are cut off and the original $N \times N$ data set is produced as output.

4.1.6 OpenMP Versions: `*_openmp.c`

OpenMP versions of the AAN, Feig-Linzer multiply-add and fast LOT algorithms have also been developed[1].

[1] It might seem strange to include separate OpenMP versions of the algorithms, since with number of threads set to 1, these algorithms do exactly the same as the non-OpenMP versions with very little or no over-

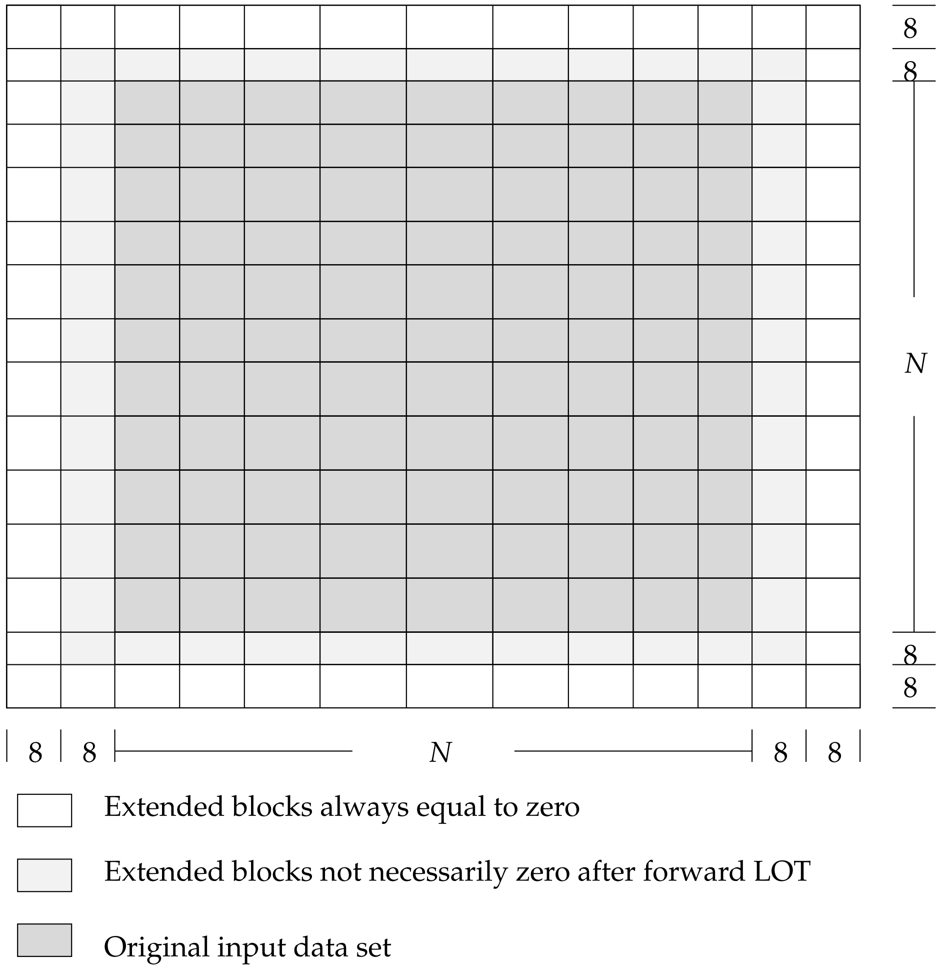
8
8
N
8
8
8 8 N 8 8
Extended blocks always equal to zero
Extended blocks not necessarily zero after forward LOT
Original input data set

Figure 4.1: LOT boundary extension implementation. All values except the first and last 8 values of each dimension (white boxes) are stored.

There are several choices as to how to parallelize the loops of the AAN and FL multiply-add algorithms. Consider Figure 4.2 below.

One way to use OpenMP, shown on the left side of Figure 4.2 is to compute each block sequentially and divide the computations of the rows and columns between the threads. This corresponds to parallelizing the inner loop (which iterates over all the row/columns in a single block and executes an 8-point transform on each row/column). This is inefficient, because OpenMP will execute a fork and a join at each iteration of the *outer* loop, which turns out to be more expensive than computing the entire 8×8 DCT or LOT transform using only one thread! Therefore, gains from parallelization are lost when using this scheme.

A better scheme is shown on the right side of Figure 4.2. This corresponds to parallelizing the outer loop (which iterates over all the blocks in the x-direction). This works more efficiently since OpenMP overhead is kept to a minimum. Listing 4.1 shows how this has been implemented in the OpenMP versions of the AAN, Feig-Linzer multiply-add and fast LOT algorithms.

```
#pragma omp parallel for private(x_start, y_start, /* other per-thread variables
    used in the transform */)
```

head. However, annoying warnings may occur when compiling the OpenMP versions on certain compilers not supporting OpenMP (about unrecognized `#pragma` directives). I have consequently provided implementations not using OpenMP

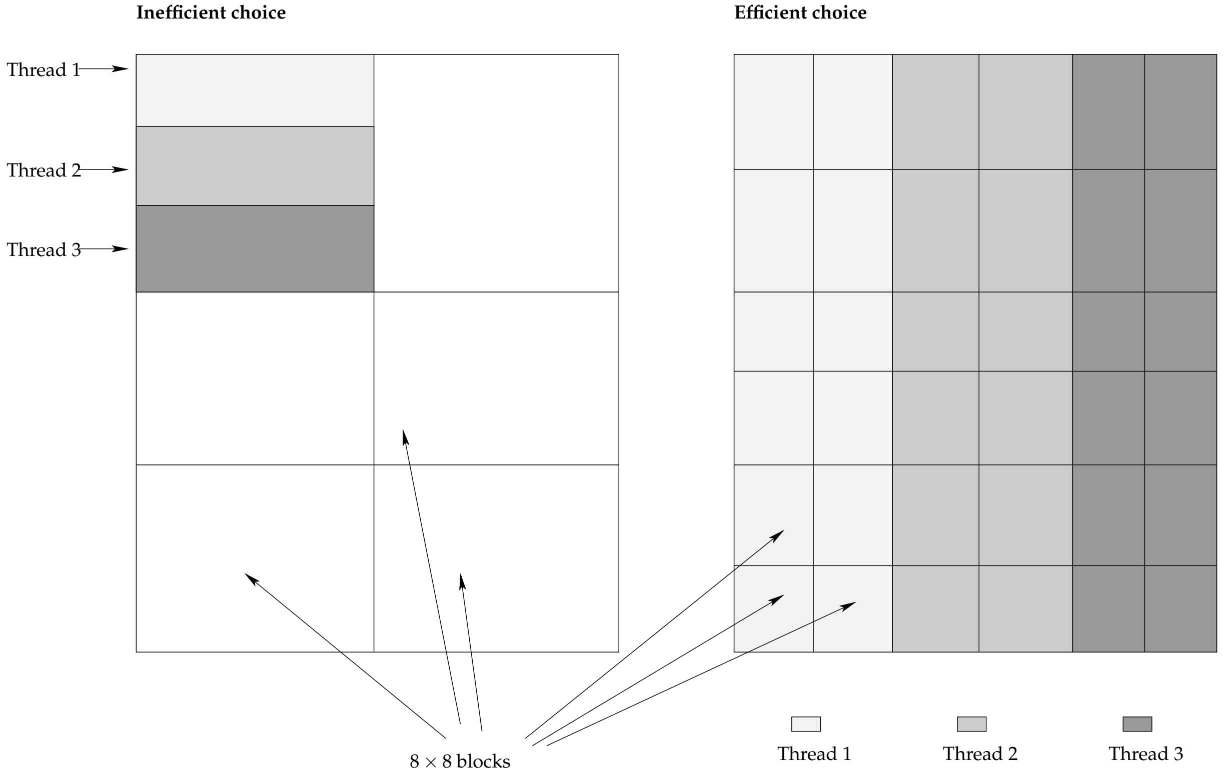

Figure 4.2: Different ways of using OpenMP when computing 2D transforms

```
for(x_start = 0; x_start < x_blocks; x_start++) {
  for(y_start = 0; y_start < y_blocks; y_start++) {
    /* Insert transform of a single block starting at
     * input[y_start * length * 8 + x_start * 8] here
     */
  }
}
```

Listing 4.1: OpenMP-parallelized transform loop

Note that the LOT is implemented by a sequence of such loops, since certain steps of the LOT must be done sequentially.

4.2 GPU Implementation

The implementations of the GPU versions of the DCT algorithms are in the files:

- `dct_gpu_aan.c` and `dct_gpu_aan.cg`: The AAN algorithm, GPU version
- `dct_gpu_flmad.c` and `dct_gpu_flmad.cg`: The FL multiply-add algorithm, GPU version
- `lot_gpu.c` and `lot_gpu.cg`: The AAN algorithm, GPU version

4.2.1 DCT on the GPU

We will now describe how the DCT has been implemented on the GPU. First, we will describe how OpenGL and Cg are initialized and how computation on the GPU is done. Next, we will describe the specific AAN and Feig-Linzer multiply-add implementations. The details are quite complicated, so we will only give an overview. Very specific details can be found in the appendix in Section B.2 on page 84.

Executing Transforms on the GPU

A coarse overview of how our GPU transform implementation works is shown in Figure 4.3.

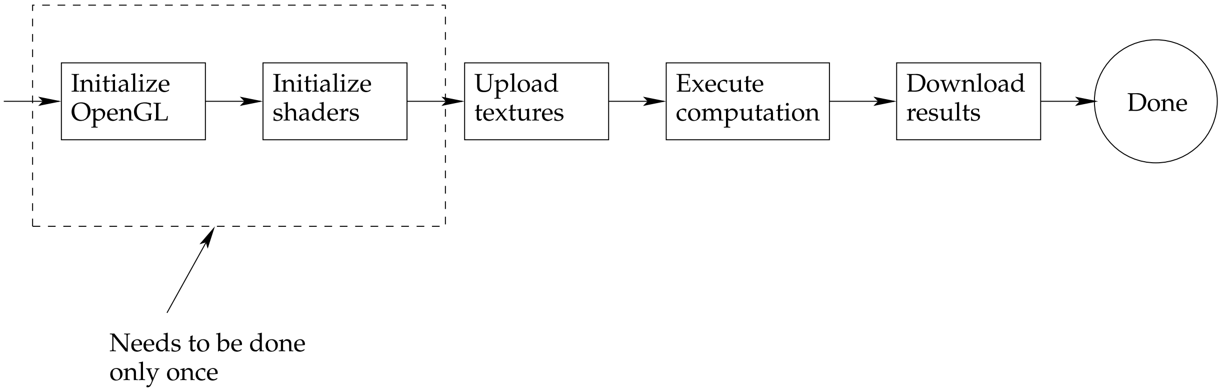

Figure 4.3: Computing transforms on the GPU

The steps in Figure 4.3 are detailed in Section B.2.1 on page 84. Summarized, they consist of:

1. *OpenGL Initialization.* This step involves creating an OpenGL coordinate system, describing how coordinates should be interpreted, and a *viewport*, which describes what part of the coordinate system should be viewed from the screen. We set the coordinate system up so that there is a one-to-one mapping between texture coordinates, OpenGL geometry coordinates, projection coordinates and viewport coordinates. This only has to be done once.

2. *Cg Initialization.* Next, our GPU fragment and vertex programs are compiled by the Cg runtime. This also only needs to be done once.

3. *Texture Upload/Initialization.* Next, the input data set is uploaded to the GPU's texture memory. For reasons described in Section B.2.1 on page 84, we use a single-channel texture, i.e. a texture containing only "red" and not red, green, blue and alpha channels, to represent the input data set. We also allocate two textures which are used for intermediate results. These textures are multi-channel, meaning that each texture element has four channels. The two multi-channel textures are bound to an OpenGL *framebuffer object* (FBO 0), and the single-channel input texture is bound to another FBO (FBO 1). Using FBOs allow us to easily render to GPU memory instead of the screen.

4. *Computation.* After the input data set has been uploaded to the GPU and the textures are initialized, computation starts. Starting the computation involves simply rendering a large square over the entire target texture. We do the computation in several steps (more on this below).

5. *Download.* After computation is done, we call OpenGL functions to read the texture from GPU memory to RAM.

The Computation Step

The actual GPU programs for both scattering and DCT/IDCT computation are located in the files `dct_gpu_aan.cg` (AAN algorithm) and `dct_gpu_flmad.cg` (Feig-Linzer multiply-add). Both 16-bit and 32-bit versions have been developed. Since the forward and inverse DCT are computed in a very similar manner, we will focus our description of forward DCT computation. The inverse DCT is simply a reversal of the steps, and is shown in the source code in the appendix. The method of computation is precisely the same whether we use AAN or Feig-Linzer multiply-add, except for the core transform algorithm. The techniques presented in this section are thus independent of the actual transform algorithm used, as long as it computes a 1D 8-point transform.

The computation step involves a large amount of technicalities. Everything is thoroughly documented in Section **??** on page ??. Here, we only provide a summary of what happens.

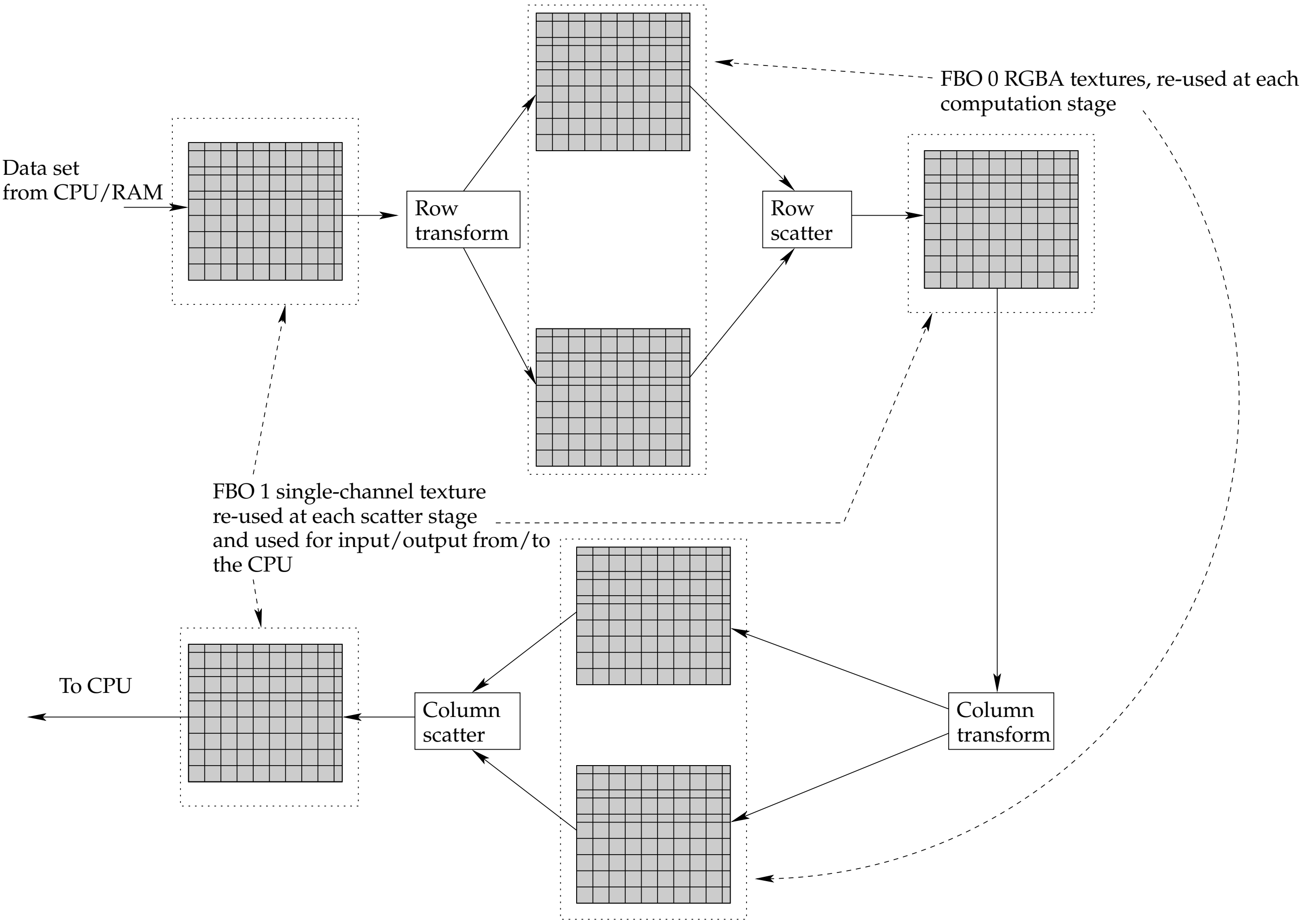
FBO 0 RGBA textures, re-used at each computation stage
Data set from CPU/RAM
Row transform
Row scatter
FBO 1 single-channel texture re-used at each scatter stage and used for input/output from/to the CPU
To CPU
Column scatter
Column transform

Figure 4.4: Computing the DCT on the GPU

Consider Figure 4.4. Computing the DCT on the GPU is done by both *vertex* and *fragment* programs. Recall from Chapter 2 that vertex programs can pass texture coordinates to the fragment programs, and the fragment programs can fetch textures, do computations and write the result to texture memory. Fragment programs do most of the job, but we have found that the vertex processors can also be used for a few small computations.

From Figure 4.4, it is evident that the computation is done in four steps: The steps of computing the *forward* DCT are:

1. *Row transform fragment program + passthrough vertex program.* First, we compute the row transform. This program reads 8 elements from a single-channel texture and outputs the first four transformed values in the first RGBA output texture, and the second four transformed values in the other RGBA texture. The vertex program does nothing; it is therefore a "passthrough" program.
2. *Row scatter fragment program + passthrough vertex program.* Next, we scatter the two RGBA textures onto the single original input texture. A passthrough vertex program is also employed in this case.
3. *Column transform fragment program + column vertex program.* Now, a fragment program computing the column DCT is executed. Each execution of the program produces 8 transform coefficients on two RGBA textures, similarly to the row transform fragment program. A vertex program executes some per-vertex values that are useful when computing the address in the single-channel texture where the fragment program will fetch the input values.
4. *Column scatter fragment program + column scatter vertex program.* Finally, the transformed values on the two RGBA textures are scattered on to a single texture, which is downloaded to the CPU. As with the transform program, a vertex program does some useful computations on the texture coordinates.

We provide summaries of the steps below. For very technical details, see Section B.2.2 on page 92.

Step 1. Row transform Recall from Chapter 2, where we claimed that fragment programs could only write to a single pre-determined texture memory location. However, each DCT program invocation computes *eight* transform values. This is precisely why we needed two RBGA textures: Each RBGA texture contains four channels, and since there are two textures, we can write eight values.

However, one problem occurs. We need a mapping between the single-channel texture and the RBGA texture. During the present step, one fragment program is invoked for each texture location in the RGBA textures. This texture program needs to fetch correct input values from the single-channel texture. Details on how this is done is shown in Section B.2.2 on page 92. The mapping between rows in the single-channel texture and the two RGBA textures is shown in Figure 4.5 on the facing page.

Row transform is done by a single fragment program.

Step 2. Row scatter After step 1, we are left with two RGBA textures, and the computed values are spread over the two RGBA textures. However, if we are now going to do a column transform or download the results to the CPU, we will have a problem. Consider a column program doing a column transform on the first column. When the column transform program would fetch the first value of the first column from the RGBA textures (see Figure 4.5 again), the program would automatically obtain *three* redundant values. This happens since there is no way to fetch a single channel from an RGBA textures. Four values are *always* fetched. In addition, the texture coordinate computations turns out to be complicated if a column transform is done over two RGBA textures — i.e. it is more difficult to find a column of values in the two RGBA textures than in a large single-channel texture (where a column is simply located along increasing y-coordinates).

This is why we use a *scatter* program. The row scatter program essentially computes the inverse mapping of the row transform program: The row transform program maps rows

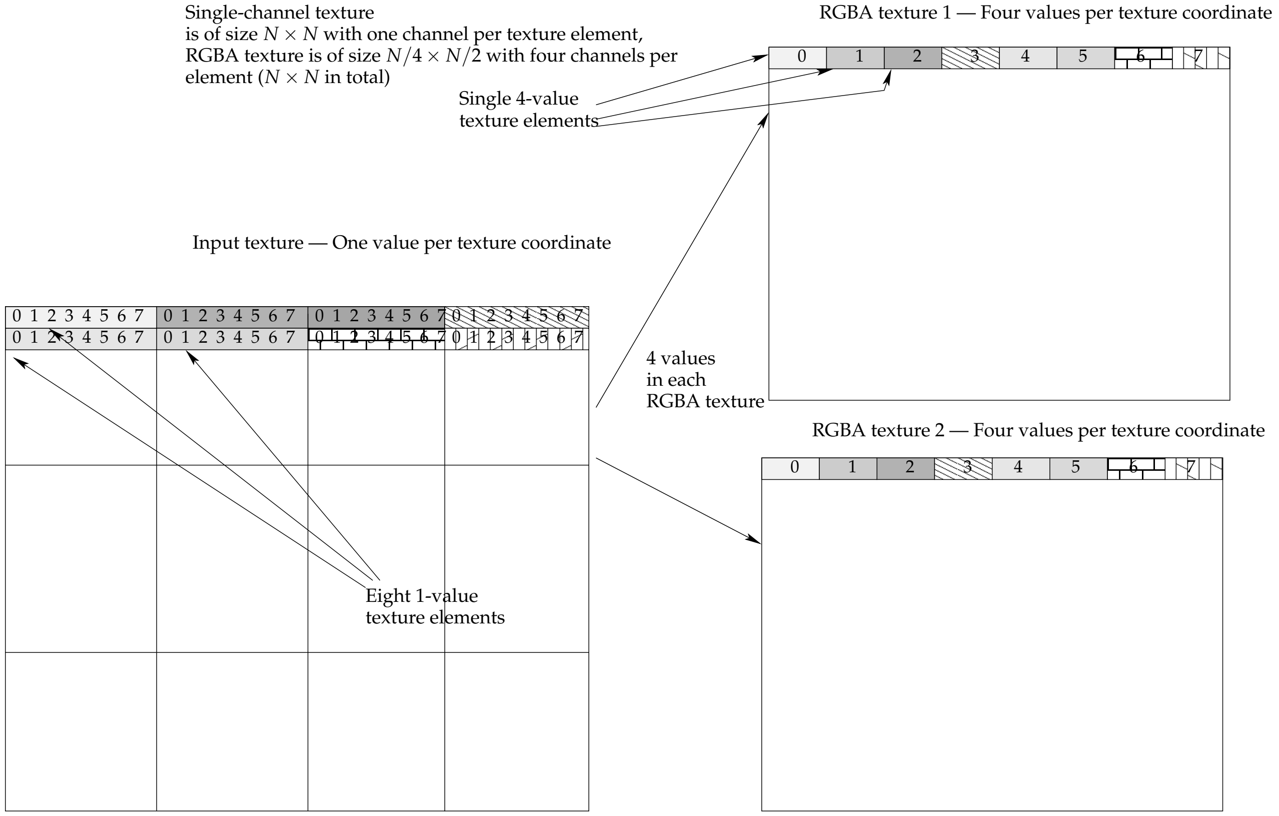
Single-channel texture
is of size $N \times N$ with one channel per texture element,
RGBA texture is of size $N/4 \times N/2$ with four channels per
element ($N \times N$ in total)
Single 4-value
texture elements
RGBA texture 1 — Four values per texture coordinate
0 1 2 3 4 5 6 7
Input texture — One value per texture coordinate
0 1 2 3 4 5 6 7
4 values
in each
RGBA texture
RGBA texture 2 — Four values per texture coordinate
Eight 1-value
texture elements

Figure 4.5: Row mapping between eight texture elements in a single-channel texture and two multi-channel textures

from the single-channel texture to locations in the two RGBA textures, while the row scatter program maps locations from the two RGBA textures back into correct locations the single-channel texture. This makes the column transform simpler. The row scatter is done by a single fragment program. As usual, we draw a large square over the target texture (in this case, the single-channel texture) to start computation.

Step 3. Column transform The column transform program is similar to the row transform program, but we now fetch values from columns instead of rows. Again, transformed values are written to the two RGBA textures. Figure 4.6 on the next page shows how the column transform program map columns from the single-channel texture to the two RGBA-textures.

The column transform program uses a vertex program in addition to the fragment program, since some parts of the texture coordinate computations (computations for mapping coordinates of input values in the single-channel texture to the RGBA textures) can be done by the vertex program. See Section B.2.2 on page 92 for the details.

Step 4. Column scatter Finally, we execute a column scatter in order to prepare the transformed values for download to the CPU. Note that if we do not do this step, the transformed values will be spread out over the two RGBA textures. In order for the CPU to get the values in a standard array, we need to execute a column scatter.

The column scatter is similar to the row scatter, but operates on the mapping of Figure 4.6 on the next page instead of Figure 4.5. See Section B.2.2 on page 92 for furhter information.

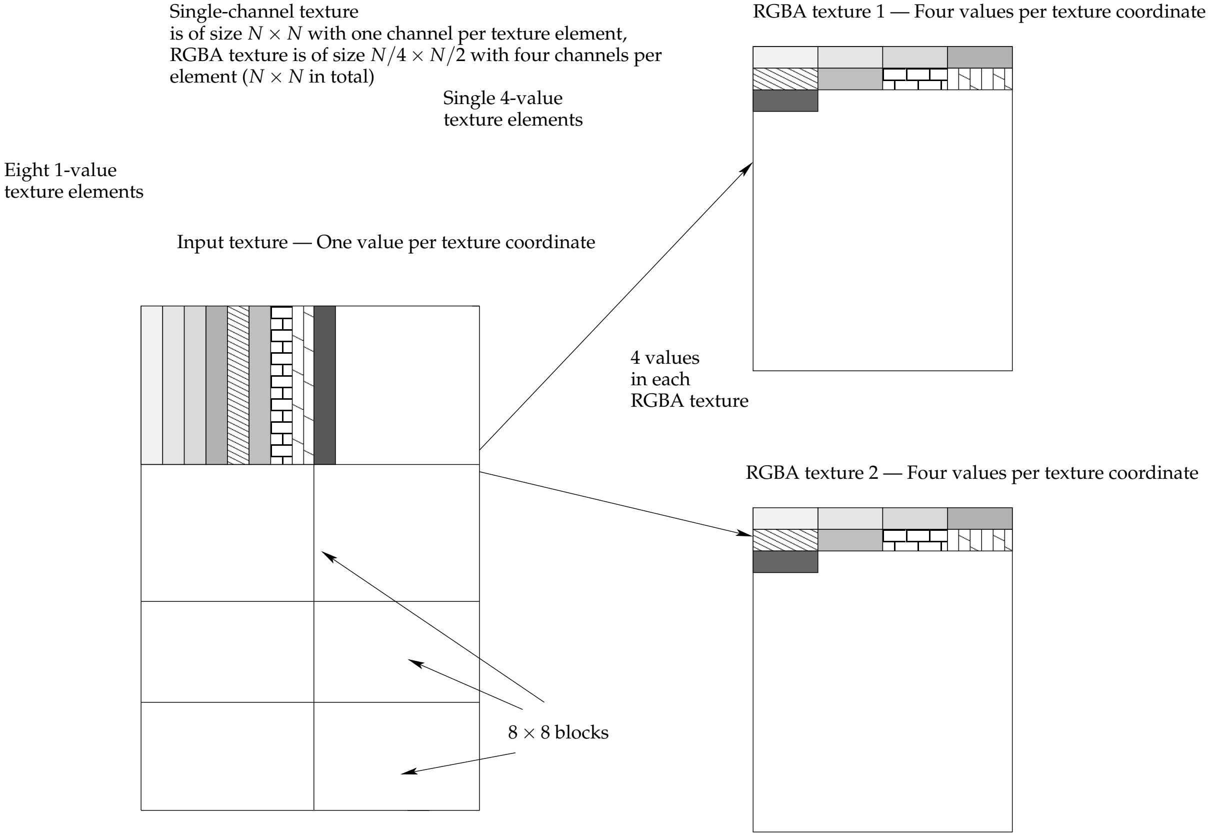
Single-channel texture
is of size $N \times N$ with one channel per texture element,
RGBA texture is of size $N/4 \times N/2$ with four channels per
element ($N \times N$ in total)
Single 4-value
texture elements
RGBA texture 1 — Four values per texture coordinate
Eight 1-value
texture elements
Input texture — One value per texture coordinate
4 values
in each
RGBA texture
RGBA texture 2 — Four values per texture coordinate
8 × 8 blocks

Figure 4.6: Column mapping between eight texture elements in a single-channel texture and two multi-channel textures

The Actual DCT Algorithms The row and column scattering is quite tedious. Now for the fun part. We have still not discussed the actual DCT algorithms. The forward Feig-Linzer DCT algorithms is shown in Listing 4.2 below. The AAN, inverse and 16-bit algorithms are similar, and can be found on the accompanying CD.

```
// Note that FLMAD_T1, FLMAD_T2, etc are predefined constants
void dct_gpu_fflmad( inout float4 in0_3 , inout float4 in4_7 )
{
  float4  vvec1 , vvec2;
  float4  uvec1 , uvec2;

  float4  mvec;

  vvec1 = in0_3.xywz + in4_7.wzxy;
  vvec2 = float4(in0_3.x, in0_3.y, in0_3.w, in4_7.y) − float4(in4_7.w, in4_7.z,
      in4_7.x, in0_3.z);

  uvec1 = vvec1.xyxy + float4(vvec1.z, vvec1.w, −vvec1.z, −vvec1.w);
  uvec2 = vvec2.xyzy + float4(0.0, vvec2.w, 0.0, −vvec2.w);

  vvec1 = uvec1;
  vvec2 = FLMAD_C4 * float4(uvec2.w, uvec2.y, −uvec2.y, −uvec2.w) + vvec2.xzzx;

  mvec = float4(1.0, FLMAD_T1, FLMAD_T2, FLMAD_T5);

  uvec1 = float4(vvec1.x, vvec2.y, vvec1.w, vvec2.w) * mvec + float4(vvec1.y,
      vvec2.x, vvec1.z, −vvec2.z);
  uvec2 = float4(vvec1.x, vvec2.z, vvec1.z, vvec2.x) * mvec.xwzy − float4(vvec1.
      y, -vvec2.w, vvec1.w, vvec2.y);
  mvec = float4(FLMAD_C1_2, FLMAD_C2_2, FLMAD_C4_2, FLMAD_C5_2);

  in0_3 = uvec1 * mvec.zxyw;
  in4_7 = uvec2 * mvec.zwyx;
}
```

Listing 4.2: 32-bit Feig-Linzer multiply-add GPU implementation

It is interesting to compare the vectorized GPU versions to the standard C versions of these algorithms. Notice, for instance, how easily four sums used in the AAN algorithm is computed by a single SIMD instruction

```
s = in0_3 + in4_7; /* calculates s07,s16,s25 and s34 */
```

Let us now, just for fun, look at the GPU assembly code of the `dct_gpu_fflmad` function. This can be generated by using NVIDIA's Cg compiler. The result is shown below in Listing 4.3. Note that this is not *exactly* the code that will be executed on the GPU, since Cg produces *pseudoassembly code*, but it is probably close. The instructions on the actual GPU may be slightly different.

```
MOVR R7.x, R2;
MOVR R7.y, R3.x;
MOVR R7.z, R4.x;
MOVR R6.z, R0.x;
MOVR R6.w, R1.x;
MOVR R7.w, R5.x;
MOVR R6.xy, R8;
ADDR R6, R6, -R7;
MOVR R7.w, R0.x;
MOVR R0.y, R1.x;
MOVR R0.z, R3.x;
MOVR R0.w, R2.x;
MOVR R0.x, R4;
MOVR R1.y, -R6.w;
MOVR R1.x, R6.w;
ADDR R3.xy, R6.y, R1;
MOVR R7.xy, R8;
MOVR R7.z, R5.x;
ADDR R0, R7.xywz, R0.wzxy;
MOVR R1.xy, R0.zwzw;
MOVR R1.w, -R0;
MOVR R1.z, -R0;
ADDR R2, R0.xyxy, R1;
MOVR R0.xy, R3.yxzw;
MOVR R0.w, -R3.y;
MOVR R0.z, -R3.x;
MADR R3, R0, c[6].x, R6.xzzx;
MOVR R1.y, R3.x;
MOVR R1.w, -R3.z;
MOVR R1.xz, R2.yyzw;
MOVR R0.xz, R2.xyww;
MOVR R0.yw, R3;
MADR R0, R0, c[7], R1;
MOVR R1.xz, R2;
MOVR R1.yw, R3.xzzx;
MOVR R4.y, -R3.w;
MOVR R4.w, R3.y;
MOVR R4.xz, R2.yyww;
MADR R1, R1, c[7].xwzy, -R4;
```

```
MULR  oCol , R0 , c[8];
MULR  oCol1 , R1 , c[8].xwzy;
```

Listing 4.3: GPU assembly code for the Feig-Linzer multiply-add DCT

There are certainly many `MOVR` instructions, since we need to do several swizzles (reorders) and construction of vector values in the code. However, using swizzles and vector operations should, according to NVIDIA and common sense, in general give better performance than doing some computations serially [54].

Notice that the `ADDR`, `MULR` and `MADR` are all vectorized. It is particularly interesting to see that multiply-add instructions (`MADR`) have been used three times. In the corresponding assembly output for the AAN algorithm, `MADR` is only used once, but it has many more addition (`ADDR`) instructions (8 vs. 4), and more multiply (`MULR`) instructions (8 vs. 4). Consequently, the Feig-Linzer multiply-add algorithm should, as we suspected in the previous chapter, be optimal for the GPU.

4.2.2 LOT on the GPU

That was DCT. Now let us look at the fast LOT implementation, which is an extension of the DCT implementation. We only describe forward LOT. The inverse LOT is just a reversal of all the steps.

Figure 4.7 shows how the fast LOT of Section 3.3.3 on page 42 has been implemented. For the inverse LOT, follow the graph in the opposite direction. We will focus on the forward LOT implementation in this section, since the inverse LOT is exactly the forward LOT done "backwards". Source code for both the forward and inverse LOT, 32-bit versions, is provided in Section **??** on page ?? (16-bit versions are on the CD).

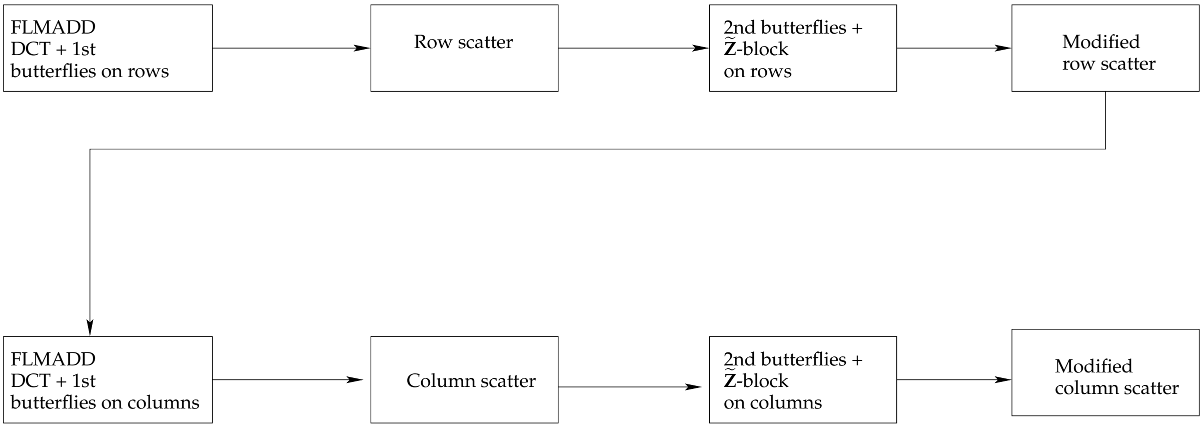
FLMADD DCT + 1st butterflies on rows
Row scatter
2nd butterflies + $\tilde{\mathbf{Z}}$-block on rows
Modified row scatter
FLMADD DCT + 1st butterflies on columns
Column scatter
2nd butterflies + $\tilde{\mathbf{Z}}$-block on columns
Modified column scatter

Figure 4.7: Fast LOT on the GPU

Consider Figure 3.10 on page 43. To compute the LOT on the GPU, we first compute the row DCT. We use the Feig-Linzer multiply-add algorithm, since, as we saw in the previous section, it is probably better than AAN on the GPU. Now, it is easy to incorporate the first butterflies of the LOT inside the DCT. Just add one line to the end of `dct_gpu_fflmad` and change variables a bit so that the four even and odd transform variables are located in separate variables, as in Listing 4.4 (see also Listing 4.2 on page 56):

```
void lot_gpu_fflmad_butterfly( inout float4 in0_3 , inout float4 in4_7 )
```

```
{
  // body is just as dct_gpu_fflmad , except that we keep
  // even coefficients in vvec1 and odd coefficients in vvec2
  // (instead of keeping 0-3 in one variable and 4-7 in another)

  /* Butterfly */

  in0_3 = vvec1 + vvec2;
  in4_7 = vvec1 - vvec2;
}
```

Listing 4.4: 32-bit AAN GPU implementation

That is all — two vectorized instructions implement the entire butterfly.

Next, we scatter the rows as usual, using precisely the same programs as with the DCT. Then, we need to execute the second butterfly, the 1/2 scaling and the $\widetilde{\mathbf{Z}}$-block. This is the "lapping" step. We can not do this step in the same rendering pass as the DCT + first butterfly step, since computing the second butterfly requires using the first butterfly results from *other blocks*. Since the GPU is a stream processor, the only way to obtain the results from the other blocks is by introducing a new rendering pass. However, we *can* do the $\widetilde{\mathbf{Z}}$-computation in the same pass as the second butterflies, since the $\widetilde{\mathbf{Z}}$-computation is independent of second butterfly computations in other blocks.

For the second butterfly along the rows, we use the passthrough vertex program (since no computations can be done in the vertex processor) and a fragment program. The fragment program is shown in Section B.2.3 on page 99. This program operates slightly differently than the DCT row programs, since it fetches only the *last four* values of the "current" block, and fetches the *first four* values from the *next* block, in accordance with Figure 3.10 on page 43.

But now, we must modify the row scatter program slightly, since the first four LOT coefficients of a block is stored as the first four values of the *previous* block. Hence, when we scatter the rows, we must fetch the first four block values from the previous block. This is a slight modification of the row scatter programs we used with the DCT. Section B.2.3 on page 99 shows the modified row scatter program.

After this, we can start processing the columns. We do the column transform and column scatter precisely as in the DCT case, but we use the modified DCT of Listing 4.4 so that the first butterflies are computed in the same step. Next, we compute the LOT for the columns. Finally, we compute the second butterflies on the columns and use a modified column scatter program similar to the modified row scatter program to fetch the first four coefficients of a column from the *previous* column. Then, the LOT is computed.

We have finally described the GPU transform implementations. In the next chapter, we will see how they stack up to the CPU implementations.

Chapter 5

Evaluation

In this chapter, we will consider the performance of the implementations described in the previous chapter, and look particularly at how the GPU versions compare to the CPU versions.

5.1 Test Machine and Methodology

We ran performance tests on the following machine:

- Quad-Core (2 Dual-Core) Intel Xeon 5160 CPUs, 3.0 GHz, 4 MB L2 cache
- 4 GB RAM
- Dual NVIDIA Quadro FX 3500, connected using PCI Express x16[1]

It should be noted that each Dual-Core Intel Xeon 5160 costs about the same as an NVIDIA Quadro 3500. Thus the total price of the Quad-Core system is about twice the price of the NVIDIA Quadro FX 3500.

Our tests were run on Red Hat Enterprise Linux version 4, kernel 2.6.9-34.ELsmp. We used NVIDIA Cg runtime version 1.5, NVIDIA OpenGL drivers version 1.0-8776, FFTW version 3.1.2 compiled with SSE2 support, and the Intel C++ Compiler for Linux version 9.1. All tests were run 40 times, and the results display the average value obtained from those 40 runs. The first timing of each (forward and inverse) run was discarded, to avoid timing initializations (particular GPU driver initializations) that are only done once. The input values were randomly generated in each run. The system was completely idle when the tests were run, and we ran the tests at normal UNIX priority levels.

When computing a transform, we can either use the forward or inverse transform. Both are equally important, but they should theoretically take precisely the same amount of time. We have used the following approach when benchmarking a transform algorithm: Half of the tests of an algorithm are run with the forward transform, and half of the tests are run with an inverse transform, and then, the average time is computed. Thus the times displayed in the measurements below is average time a forward or inverse DCT will take. Operations that only need to be done once per program execution, such as compiling the Cg shader programs and creating FFTW plans, are not measured. However, the upload and download time to/from the GPU *is* included in the performance timings.

[1] Note that only one card is used at a time in our measurements.

N	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
Algorithm							
cpu-naive-32	0.0017	0.0067	0.0269	0.1073	0.4529	1.8165	7.2778
cpu-fftw-simple-32	**<0.0001**	0.0002	0.0006	0.0028	0.0290	0.1274	0.5176
cpu-fftw-advanced-32	**<0.0001**	0.0002	0.0008	0.0032	0.0306	0.2038	0.8298
cpu-aan-32	**<0.0001**	0.0002	0.0007	0.0031	0.0310	0.1398	0.5842
cpu-flmad-32	**<0.0001**	0.0001	0.0004	0.0019	0.0295	0.1285	0.5319
cpu-aan-omp-2cpu-32	**<0.0001**	0.0001	0.0005	0.0021	0.0231	0.1065	0.4156
cpu-flmad-omp-2cpu-32	**<0.0001**	0.0001	0.0004	0.0015	0.0232	0.1046	0.4059
cpu-aan-omp-4cpu-32	**<0.0001**	**<0.0001**	**0.0003**	0.0013	0.0203	0.0789	**0.2978**
cpu-flmad-omp-4cpu-32	**<0.0001**	**<0.0001**	**0.0003**	**0.0011**	**0.0191**	**0.0741**	0.3112
gpu-aan-32	0.0106	0.0110	0.0123	0.0173	0.0381	0.1181	0.4371
gpu-flmad-32	0.0101	0.0105	0.0121	0.0173	0.0378	0.1178	0.4372

Table 5.1: 32-bit 8 × 8 DCT/IDCT wallclock times (seconds)

We focus on 32-bit computations, since 32 bits of precision is the only precision common to the CPU and GPU. There is no support in Cg nor the GPU for more than 32 bits of precision, and the CPU does not support 16-bit natively. However, we include some benchmarks of the 16-bit GPU and 64- and 128-bit CPU algorithms to get an idea of the performance impact of increasing precision.

We only consider data set sizes up to 4096 × 4096. The reason is that most GPUs, including the one we used for benchmarking, only support textures up to that size. Unfortunately, certain revisions of NVIDIA GeForce 6 and 7 GPUs have a hardware bug that makes the *last row and column* of the 4096 × 4096 texture unaccessable from fragment shader programs. We have tested the program on one such GPU, and the resulting transform values were corrupted. Therefore, be careful when using the program with 4096 × 4096 textures. The GPU in our benchmark machine did not have this problem.

5.2 DCT performance

In this section, we show benchmarks of and discuss the performance of the various DCT/IDCT algorithms.

5.2.1 Measurements

We will now show the results of the performance measurements. We will consider wallclock time, CPU time and download/upload time briefly.

Wallclock Time

Consider Table 5.1, which displays the wallclock time of the DCT/IDCT using 32 bits of precision. Boldfaced values indicate the quickest algorithm for a specific input set size. The interesting results are depicted graphically in Figures 5.1 on the next page.

For reference, we have also benchmarked 16-bit GPU versions and 64- and 128-bit CPU versions. The results are shown in Tables 5.2, 5.3 and 5.4, and illustrated in Figure **??**.

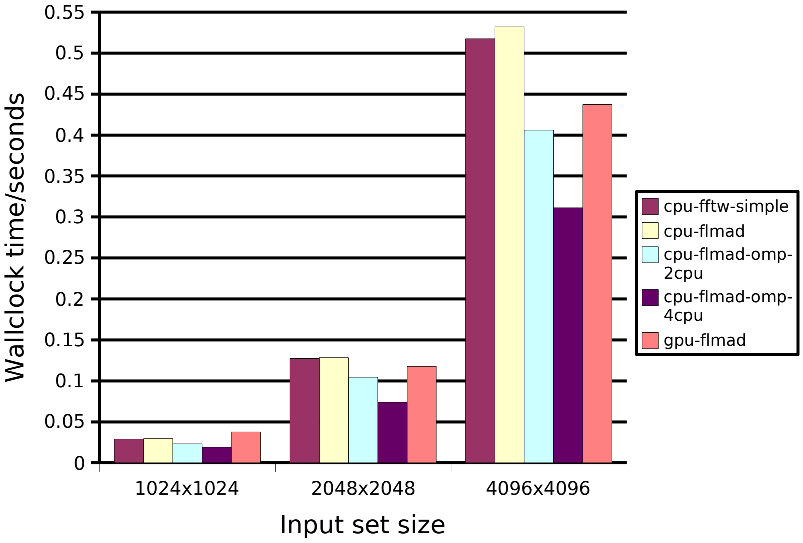

Figure 5.1: 32-bit DCT/IDCT performance. The naive and AAN algorithms are not shown, as they are inferior to other CPU-algorithms in this case, and AAN performs nearly equal to the Feig-Linzer multiply-add algorithm on the GPU.

N	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
Algorithm							
gpu-aan-16	**0.0106**	**0.0107**	**0.0120**	0.0161	**0.0319**	**0.0943**	**0.3451**
gpu-flmad-16	**0.0106**	0.0109	0.0121	**0.0160**	**0.0319**	0.0949	0.3441

Table 5.2: 16-bit 8 × 8 GPU DCT/IDCT wallclock times (seconds)

N	64	128	256	512	1024	2048	4096
Algorithm							
cpu-naive-64	0.0020	0.0081	0.0323	0.1311	0.5388	2.1634	8.6843
cpu-fftw-simple-64	**<0.0001**	0.0002	0.0007	0.0058	0.0360	0.1497	0.6119
cpu-fftw-advanced-64	**<0.0001**	0.0002	0.0008	0.0061	0.0565	0.2375	0.9655
cpu-aan-64	**<0.0001**	0.0003	0.0007	0.0066	0.0379	0.1671	0.6959
cpu-flmad-64	**<0.0001**	**0.0001**	**0.0004**	0.0049	0.0358	0.1525	0.6253
cpu-aan-omp-2cpu-64	**<0.0001**	0.0002	0.0008	0.0042	0.0298	0.1189	0.4692
cpu-flmad-omp-2cpu-64	**<0.0001**	0.0002	0.0005	0.0035	0.0265	0.1099	0.4734
cpu-aan-omp-4cpu-64	**<0.0001**	0.0002	0.0006	0.0041	0.0227	0.0939	0.3913
cpu-flmad-omp-4cpu-64	**<0.0001**	0.0002	**0.0004**	**0.0031**	**0.0210**	**0.0892**	**0.3821**

Table 5.3: 64-bit 8 × 8 CPU DCT/IDCT wallclock times (seconds)

N	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
Algorithm							
cpu-naive-128	0.0071	0.0282	0.1128	0.4537	1.8265	7.3090	29.3542
cpu-fftw-simple-128	0.0001	0.0003	0.0016	0.0109	0.0564	0.2394	1.0041
cpu-fftw-advanced-128	0.0001	0.0004	0.0019	0.0149	0.0927	0.3886	1.6001
cpu-aan-128	0.0001	0.0003	0.0017	0.0100	0.0481	0.1951	1.0308
cpu-flmad-128	0.0001	0.0003	0.0016	0.0100	0.0489	0.2000	0.8452
cpu-aan-omp-2cpu-128	0.0001	0.0003	0.0012	0.0058	0.0294	0.1305	0.6002
cpu-flmad-omp-2cpu-128	**<0.0001**	0.0003	**0.0009**	0.0064	0.0299	0.1322	0.5656
cpu-aan-omp-4cpu-128	0.0001	0.0003	**0.0009**	0.0065	**0.0250**	**0.0970**	**0.3970**
cpu-flmad-omp-4cpu-128	**<0.0001**	**0.0002**	**0.0009**	**0.0057**	0.0267	0.1022	0.4103

Table 5.4: 128-bit 8 × 8 CPU DCT/IDCT wallclock times (seconds)

N	64	128	256	512	1024	2048	4096
Algorithm							
gpu-aan-32	0.0010	0.0013	0.0030	0.0068	**0.0267**	**0.1062**	0.4240
gpu-flmad-32	0.0008	0.0015	0.0026	0.0069	0.0269	0.1071	**0.4208**
cpu-fftw-simple-32	**<0.0001**	0.0002	0.0006	0.0028	0.0290	0.1274	0.5176
cpu-flmad-32	**<0.0001**	**0.0001**	**0.0004**	**0.0019**	0.0294	0.1285	0.5319
cpu-aan-32	**<0.0001**	0.0002	0.0007	0.0031	0.0310	0.1398	0.5842
cpu-flmad-openmp-2cpu-32	**<0.0001**	0.0003	0.0008	0.0028	0.0460	0.2086	0.8104
cpu-flmad-openmp-4cpu-32	0.0001	0.0003	0.0012	0.0044	0.0737	0.2911	1.2374

Table 5.5: 8 × 8 DCT/IDCT CPU times (user + system)

CPU Time

Although the CPU seems to outperform the GPU when multiple cores are used in terms of wallclock time, CPU time used will in many cases be more important. CPU time measures CPU load, and using less CPU time on computing a transform means the ability to use CPU time on something else.

For the single-core CPU algorithms, the CPU time was measured to (and should — if the system is idle) be almost the same as the wallclock time[2]. For multi-core CPU algorithms, the CPU time is equal to the wallclock time times the number of cores used. Most of the CPU time used in the CPU algorithms is CPU user time. The CPU system time was measured to practically zero for all the CPU algorithms. This is natural, since the algorithms call no system calls while computing.

For the GPU, however, there is no simple connection between the CPU time and the wallclock time, and more CPU system time is used since the operating system is heavily involved in transferring commands and data to the GPU. The total CPU time for the 32-bit GPU DCT algorithms is, along with the CPU time numbers for some of the CPU algorithms, shown in Table 5.5. Figure 5.2 on the next page provides a graphical illustration.

Impact of Upload and Download

The time used for uploading and downloading data to/from the GPU is a bottleneck. The data transfer speed is usually limited by the RAM bandwidth since data is transferred from

[2] The resolution of the CPU time timer is lower than that of the wallclock time. CPU time results therefore may deviate slightly from the wallclock times.

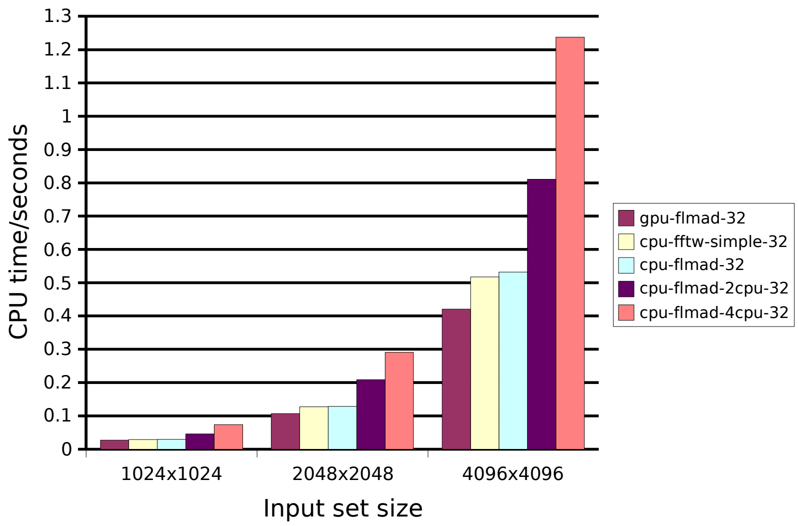

Figure 5.2: 32-bit DCT/IDCT CPU time usage

RAM to the north bridge to the GPU (with a high-bandwidth bus from the north bridge to the GPU).

The upload time is difficult to measure, since the GPU driver can move data to the GPU at *any time* prior to rendering. Thus data upload can happen immediately after we pass the texture to the GPU driver, or be deferred until we start passing rendering commands to the GPU. However, usually, upload times take longer than download times, since the latter requires less setup.

Download time is a bit easier to measure, but we can only provide rough estimates. During testing, we observed that downloading a 32-bit 4096×4096 takes approximately 0.15 seconds on our test machine (obtained by measuring the time taken by the `glReadPixels` function). Considering Table 5.1 on page 62 again, we see that this is a considerable amount of time compared to the total time, and we can expect that upload times take at least as long as the download times. Thus one estimate of the time actually used by the GPU for computation is $W - 0.15 \times 2$, where W is the wallclock time for computing the 4096×4096 data set. This gives an estimated upper bound of 0.1372 seconds for actually computing the entire 4096×4096 DCT, representing a speedup 2 over the best quad-core CPU algorithm.

5.2.2 Discussion

Consider Tables 5.6 and 5.7, which shows the speedups of the quickest GPU, dual and quad-core algorithms over the quickest single-core algorithm for each input set size.

In terms of *wallclock time*, the GPU algorithms are unable to outperform dual- and quad-core CPU algorithms. When using 16-bit precision, the GPU is able to outperform the 32-bit dual-core CPU implementation for input sizes greater than 2048×2048, but still not the quad-core implementation. However, in this case, the single-core DCT algorithm is outperformed for input sizes greater than or equal to 2048×2048.

In terms of *CPU time*, the GPU uses less CPU time than the quickest CPU algorithms for

N	64	128	256	512	1024	2048	4096
Algorithm							
Best 32-bit GPU	<0.00	0.01	0.05	0.16	0.77	1.08	1.18
Best 32-bit 1xCPU (base)	**1.00**	1.00	1.00	1.00	1.00	1.00	1.00
Best 32-bit 2xCPU	0.62	0.96	1.50	1.87	1.25	1.22	1.28
Best 32-bit 4xCPU	0.86	**1.30**	**2.00**	**2.55**	**1.52**	**1.72**	**1.66**
Best 16-bit GPU	<0.00	0.01	0.05	0.17	0.91	1.35	1.50
Best 64-bit CPU	0.56	0.46	1.50	0.90	1.38	1.43	1.35
Best 128-bit CPU	0.38	0.37	0.67	0.43	1.16	1.31	1.30

Table 5.6: Wallclock time speedups of GPU and multi-core DCT algorithms over the quickest single-core CPU algorithm. Boldface values indicate the best 32-bit algorithm.

N	64	128	256	512	1024	2048	4096
Algorithm							
Best 32-bit GPU	0.12	0.07	0.15	0.28	**1.10**	**1.21**	**1.26**
Best 32-bit 1xCPU (base)	**1.00**	**1.00**	**1.00**	**1.00**	1.00	1.00	1.00
Best 32-bit 2xCPU	**1.00**	0.33	0.50	0.68	0.64	0.62	0.66
Best 32-bit 4xCPU	**1.00**	0.33	0.33	0.43	0.40	0.44	0.43

Table 5.7: CPU time speedup of 32-bit DCT using GPU and multi-core algorithms over single-core CPU algorithm

input sizes greater than or equal to 1024 × 1024. The GPU implementations use considerably less CPU time than the dual- and quad-core implementations, and the savings increase as the input size increases.

Our findings can be summarized as follows. If CPU time must be saved (e.g. in order to use CPU time in other parts of an application), *the CPU algorithms will use from 26% (single-core) to about 300% (quad-core) more CPU time than the GPU versions for data sets of size* 4096 × 4096. In order for CPU time to be saved, the input set size must be $\geq$ 1024 × 1024. Wallclock time can be saved on single-core systems when using the GPU for DCTs of size greater than 2048 × 2048, resulting in a speedup of 1.08 to 1.18, depending on the data set size. If 16-bit precision is sufficient, wallclock time can also be saved on dual-core systems. However, using the GPU does not result in wallclock time savings on quad-core systems.

A couple of other interesting things are revealed in the measurements. First, the GPU AAN and Feig-Linzer multiply-add algorithms show nearly identical timings. This suggests, as we guessed, that the time of data upload, download and setup time completely dominates the computation time. This is a sign that the DCT is not of sufficient arithmetic complexity to be efficient on the GPU, since as arithmetic complexity increases, the importance of upload/download/setup times diminish.

Second, linear speedup is *not obtained* in the multi-core versions. Ideally, a quad-core algorithm should achieve a speedup of four, but a speedup of only 1.66 is achieved. One plausible reason is memory access contention.

Third, FFTW using the simple interface is apparently the quickest single-core CPU algorithm, although it is of higher arithmetic complexity than the AAN and Feig-Linzer multiply-add algorithms. One reasons for this might be that FFTW has been heavily SIMD-optimized, while our code has not. FFTW uses completely different methods than we have used, and FFTW is passed a 2D DCT, so that direct 2D FFT methods, which are very different from our 1D algorithms, might be used. However, it is a surpise that the advanced FFTW interface

N **Algorithm**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
gpu-flot	0.0085	0.0485	0.0568	0.0697	0.1255	**0.3412**	**1.2378**
cpu-flot-32	0.0001	0.0005	**0.0021**	0.0098	0.1348	1.2676	4.9811
cpu-flot-omp-2cpu-32	0.0004	0.0006	0.0023	0.0070	0.0988	1.0491	4.0143
cpu-flot-omp-4cpu-32	**0.0003**	**0.0004**	0.0027	**0.0055**	**0.0764**	0.7900	3.4243

Table 5.8: 32-bit CPU/GPU 8 × 8 LOT/ILOT wallclock times

uses longer time. The problem could be that threads are not properly used, or that our pre- and post-scaling code might run slower than the code in the simple FFTW interface. Note that our GPU algorithm outperforms single-core FFTW for input sizes greater than or equal to 2048 × 2048.

Fourth, considering Tables 5.1 and 5.2 on page 63, 16-bit does indeed reduce GPU computation time substantially, but only when the data set size is large. Again, for smaller data set sizes, download, upload and setup times completely dominate. However, with data sizes 2048 × 2048 and 4096 × 4096, the 32-bit GPU algorithms use about 25% more time than the 16-bit GPU algorithms.

Fifth, precision also matters on the CPU. Tables 5.1, 5.3 and 5.4 on page 64 show that using 64-bit and 128-bit precision instead of 32-bit precision gives an overhead of anywhere from 5 to 10% (64-bit) and from 10 to 30% (128-bit) for large input sizes.

5.3 LOT performance

In this section, we show benchmarks of and discuss the performance of LOT/ILOT on the CPU and GPU.

5.3.1 Measurements

We will now show the results of the performance measurements. As with the DCT, we will consider wallclock time, CPU time and download/upload time.

Wallclock Time

Table 5.8 shows the wallclock time of the LOT[3]. As with the DCT, boldfaced values indicate the quickest algorithm for a specific input size. We have also benchmarked 16-bit GPU versions and 64- and 128-bit CPU versions. The results are shown in Tables 5.9, 5.10 and 5.11, respectively. Figure 5.3 show parts of this information graphically for data set sizes $\geq 512 \times 512$.

CPU Time

As we mentioned in connection with the DCT results, CPU time may be equally important to wallclock time. Table 5.12 and Figure 5.4 shows the CPU time usage of the 32-bit LOT algorithms.

[3] Note that the data set sizes in the tables *include* the extended boundaries. Thus a 512 × 512 set is really a 496 × 496 data set.

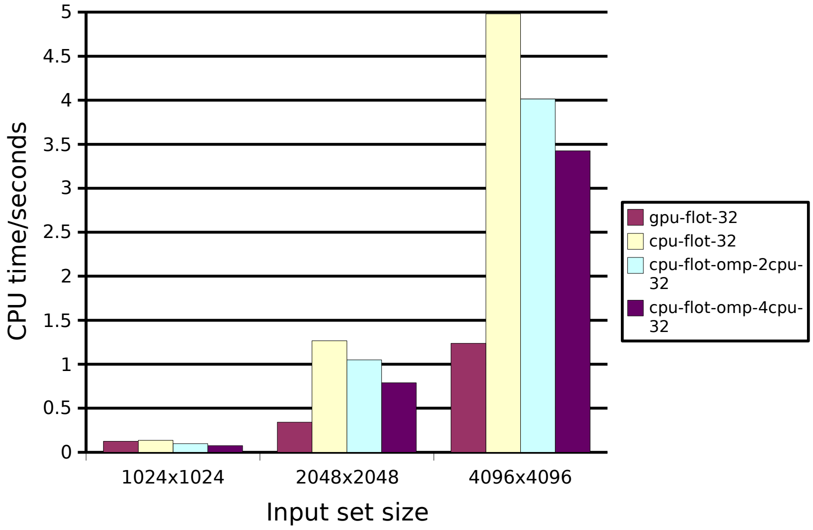

Figure 5.3: Fast LOT/ILOT performance using Feig-Linzer multiply-add for the DCT step. 16-, 32-, 64- and 128-bit algorithms are shown.

N **Algorithm**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
gpu-flot-16	**0.0928**	**0.0590**	**0.0574**	**0.0656**	**0.1039**	**0.2782**	**0.9226**

Table 5.9: 16-bit 8 × 8 GPU LOT/ILOT wallclock times (seconds)

N	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
Algorithm							
cpu-flot-64	**0.0001**	**0.0005**	**0.0023**	0.0182	0.2854	1.3138	5.6094
cpu-flot-omp-2cpu-64	0.0004	0.0009	0.0028	**0.0131**	0.2554	0.9973	4.0171
cpu-flot-omp-4cpu-64	0.0005	0.0008	0.0036	0.0362	**0.2290**	**0.9635**	**3.6690**

Table 5.10: 64-bit 8 × 8 CPU LOT/ILOT wallclock times (seconds)

N	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
Algorithm							
cpu-flot-128	**0.0002**	0.0012	0.0059	0.0397	0.4256	1.8402	7.6342
cpu-flot-omp-2cpu-128	**0.0002**	**0.0006**	**0.0032**	**0.0381**	0.3185	1.1834	4.8361
cpu-flot-omp-4cpu-128	0.0008	0.0039	0.0136	0.0651	**0.2848**	**1.0984**	**4.0254**

Table 5.11: 128-bit 8 × 8 CPU LOT/ILOT wallclock times (seconds)

Impact of Upload and Download

As for the DCT, we upload and download times take a large amount of the LOT computation time. However, since the LOT is a more complex transform, the relative impact of the download and upload times should be less than that for the DCT. Using the same estimation method as we did for the DCT gives an estimated upper bound on the GPU LOT computation time of a 4096 × 4096 data set of about $1.2378 - 0.15 \cdot 2 = 0.9378$, which is about 3.65 times quicker than the quad-core CPU!

5.3.2 Discussion

Consider Tables 5.13 and 5.14 on the next page, which shows the speedup obtained by using the GPU and multi-core algorithms over using the single-core algorithm in terms of wallclock and CPU time.

It seems that the GPU implementation finally outperforms the quad-core CPU. *The GPU LOT algorithm outperforms the 2 x dual-core CPUs, which are twice as expensive than the GPU we used, by a factor of 2.7 for a* 4096 × 4096 *data set*. When only two cores are used, the speedup is 2.5, and when a single core is used, the speedup is above 4! And if 16-bit precision is used, The GPU *outperforms all CPU algorithms in terms of wallclock time* if the data set size is $\geq$ 2048 × 2048, and the speedup over the single-core version is 5.4!

In terms of CPU time, the results are even better. In this case, the GPU outperforms *all CPU algorithms* if the input data set size is $\geq$ 1024 × 1024. For 4096 × 4096 data set sizes, the single- dual- and quad-core CPU algorithms use 4.2, 6.8 and 11.1 times more CPU time than the GPU algorithm, respectively.

N	64	128	256	512	1024	2048	4096
Algorithm							
gpu-flot-32	0.0028	0.0035	0.0110	0.0229	**0.0768**	**0.2985**	**1.1866**
cpu-flot-32	**0.0001**	**0.0005**	**0.0022**	**0.0098**	0.1348	1.2676	4.9811
cpu-flot-omp-2cpu-32	0.0004	0.0011	0.0044	0.0132	0.1970	2.0956	8.0198
cpu-flot-omp-4cpu-32	0.0017	0.0017	0.0112	0.0217	0.3025	3.0957	13.2131

Table 5.12: 32-bit 8 × 8 LOT/ILOT CPU times (seconds)

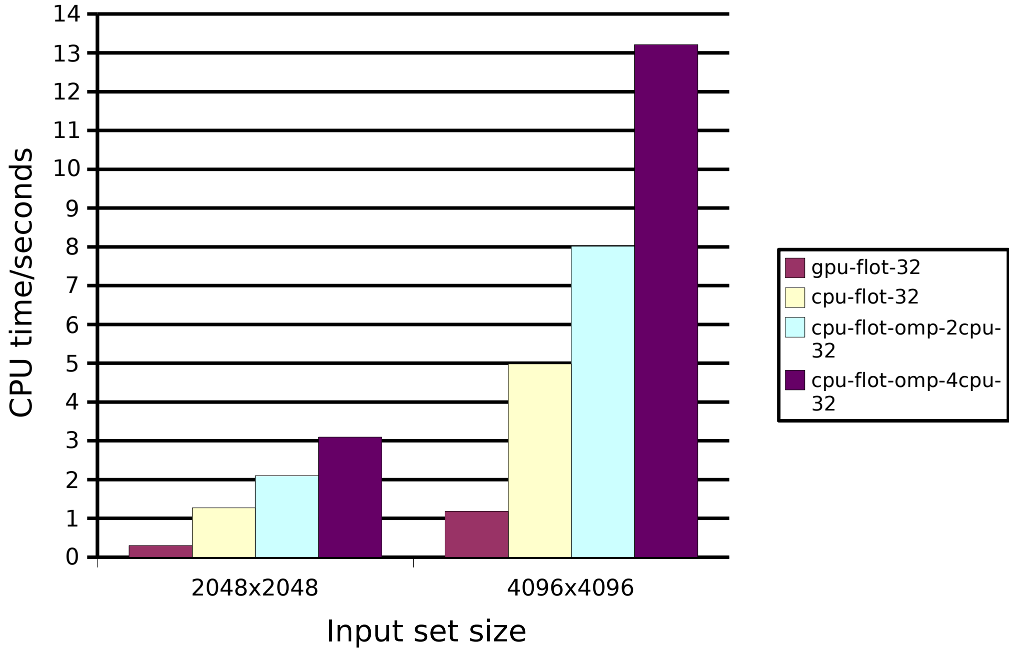

Figure 5.4: 32-bit 8 × 8 LOT/ILOT CPU time usage for large data set sizes (user + system)

N	64	128	256	512	1024	2048	4096
Algorithm							
gpu-flot-32	0.01	0.01	0.04	0.14	1.07	**3.71**	**4.02**
cpu-flot-32 (base)	**1.00**	1.00	**1.00**	1.00	1.00	1.00	1.00
cpu-flot-omp-2cpu-32	0.25	0.83	0.91	1.40	1.36	1.21	1.24
cpu-flot-omp-4cpu-32	0.33	**1.25**	0.78	**1.78**	**1.76**	1.60	1.45
gpu-flot-16	0.01	0.01	0.04	0.15	1.29	4.55	5.40
cpu-flot-omp-4cpu-64	0.20	0.63	0.58	0.27	0.59	1.32	1.36
cpu-flot-omp-4cpu-128	0.01	0.13	0.15	0.15	0.47	1.15	1.24

Table 5.13: Wallclock time speedups of using GPU and multi-core LOT algorithms over single-core CPU algorithm. Boldface values indicate the best 32-bit algorithm.

N	64	128	256	512	1024	2048	4096
Algorithm							
gpu-flot-32	0.04	0.14	0.20	0.43	**1.76**	**4.24**	**4.20**
cpu-flot-32 (base)	**1.00**	**1.00**	**1.00**	**1.00**	1.00	1.00	1.00
cpu-flot-omp-2cpu-32	0.25	0.45	0.50	0.74	0.68	0.60	0.62
cpu-flot-omp-4cpu-32	0.06	0.29	0.20	0.45	0.45	0.41	0.38

Table 5.14: CPU time speedup of 32-bit LOT using GPU and multi-core algorithms over single-core CPU algorithm

We can summarize as follows. If the data set size is $\geq 2048 \times 2048$ and wallclock time must be minimized, the GPU will give significant speedups over the single, dual or quad-core CPUs. However, the data set size must be $\geq 2048 \times 2048$. However, if only a single-core CPU is available, the GPU will still give a small speedup (1.07 32-bit, 1.29 16-bit) if the data set size is 1024×1024. The GPU also gives significant CPU time speedups over all CPU algorithms when the data set size is $\geq 1024 \times 1024$, so if wallclock time is of less importance, the GPU can be used to reduce CPU user time when the data set size is 1024×1024 on quad-core CPUs.

Unfortunately, the GPU does not outperform the CPU for smaller data set sizes. The problem is the same as with the DCT — namely, upload, download and setup times. However, contrary to the DCT, the GPU is able to outperform the quad-core CPU. This has one explanation: As arithmetic complexity increases, the bottleneck eventually becomes the actual computations, and not the upload, download and setup times. This is probably why the LOT, which has about twice the arithmetic complexity of the DCT, manages to outperform the quad-core CPU in terms of wallclock time, while the DCT does not. This suggests that even *more* complicated transforms, such as the GenLOT, will gain an even larger benefit from using the GPU.

Chapter 6

Conclusion

In this chapter, we summarize our findings and look at potential future work.

6.0.3 Findings

In this project, it was demonstrated that using the GPU for transform coding computations is indeed not only possible, but the GPU will for large data set sizes and provides a speedup of 2.77 over a CPU *costing twice as much* when using the LOT transform. Let's draw some conclusions.

Our first conclusion is that implementing algorithms on the GPU takes more time than it does on the CPU. On the GPU, few debugging tools exist, the programming model is limited, many elements of standard CPU programming languages are simply not available (bit-shifting is one example) or must be emulated (integers and booleans as floating-point), and implementing algorithms which need to write to several locations in memory can involve complicated calculations on texture coordinates. And worse, even if you *do* spend time implementing a GPU algorithm, it is not certain that any speedup will be obtained.

Our second conclusion is that upload and download of data to and from the GPU is the main bottleneck, *but as computations get large enough, the GPU wins*. Our AAN and Feig-Linzer multiply-add implementations on the GPU run nearly at exactly the same time. The execution time is (for relatively small data set sizes and for relatively simple computations) not bounded by whether or not fancy vectorized GPU code is used, but purely by upload and download times. However, and this is important, *as the data set size* or *as the computational complexity of an algorithm* increases, *computation time will eventually dominate.* That is why our LOT implementation outperformed the CPU implementation only for large data set sizes. Since the LOT does more computations than the DCT, more time is spent on computing, and the floating-point power of the CPU becomes the bottleneck for large data set sizes.

The second conclusion leads us to *our third conclusion*, which is that *even though* GPUs might be slower than the CPU in terms of wallclock time in some cases, it still offloads the CPU. As we saw in the previous chapter, using the GPU in many cases lead to reduced CPU time usage. This frees up the CPU to do other things, provided that the CPU program using the GPU for computations is multi-threaded.

We therefore conclude that the GPU *should be considered for use* for transform coding data compression if:

- Large data set sizes are used.

- The transform used is of high arithmetic complexity — such as LOT, GenLOT or GULLOT.
- Other parts of the application would gain an advantage of having more CPU time available.
- The application must reduce the wallclock time used on the transform coding algorithm to increase responsiveness.

This project has demonstrated that under these conditions, the GPU is very suitable for transform coding computations, since for large data set sizes and transforms of high arithmetic complexity, *the GPU saves both considerable amounts of CPU and wallclock time*. But we have also demonstrated that the GPU *should not be used* for transform coding data compression if:

- Only relatively small data set sizes, such as 128 × 128 and 256 × 256, are used.
- The transform does very small amounts of computation.
- You are not prepared to spend more time on developing the GPU algorithm than the CPU algorithm.
- CPU and wallclock time are not of critical importance to the application. This is often *not* the case in clustered applications.

6.0.4 Future Work

The GPU adventure has just started. Projections [55] estimate that in 8 years, GPU computing capability will increase by a factor of 40, GPU memory latency will be cut in half, and GPU memory bandwidth will increase by a factor of 8.

One of the most interesting things we found was that *more complicated transforms benefit more from using the GPU*. It would therefore be very interesting to implement transforms which do more computations than the LOT, for example GenLOT or GULLOT on the GPU. These transforms will likely obtain even better speedups over corresponding CPU algorithms.

Bibliography

[1] P. Micikevicius. GPU Computing for Protein Structure Prediction. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 43, pages 695–702. Addison-Wesley, Upper Saddle River, NJ, USA, 1st edition, 2005.

[2] J. Bolz, I. Farmer, E. Grinspun and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[3] J. Krüger and R. Westermann. GPU Computing for Protein Structure Prediction. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 44, pages 703–718. Addison-Wesley, Upper Saddle River, NJ, USA, 1st edition, 2005.

[4] C. Kolb and M. Pharr. Options Pricing on the GPU. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 45, pages 719–731. Addison-Wesley, Upper Saddle River, NJ, USA, 1st edition, 2005.

[5] X. Wei W. Li, Z. Fan and A. Kaufman. Flow Simulation with Complex Boundaries. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 47, pages 747–764. Addison-Wesley, 1st edition, 2005.

[6] T. Sumanaweera and D. Liu. Medical Image Reconstruction with the FFT. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 48, pages 765–784. Addison-Wesley, 1st edition, 2005.

[7] P. Kipfer and R. Westermann. Improved GPU Sorting. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 46, pages 733–746. Addison-Wesley, 1st edition, 2005.

[8] Wikipedia. Graphics processing unit — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Graphics_processing_unit&oldid=72792534`, 2006. [Online; accessed 31-Aug-2006].

[9] R. Fernando and M. J. Kilgard. *The Cg Tutorial*. Addison-Wesley, 1st edition, 2003.

[10] J. Krüger and R. Westermann. GPU simulation and rendering of volumetric effects for computer games and virtual environments. *Computer Graphics Forum*, 24(3), 2005.

[11] E. Kilgariff and R. Fernando. The GeForce 6 Series GPU Architecture. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 30, pages 471–491. Addison-Wesley, 1st edition, 2005.

[12] BeHardware. NVIDIA GeForce 7800 GTX (page 1: Introduction). `http://www.behardware.com/articles/574-1/nvidia-geforce-7800-gtx.html`, 2005. [Online; accessed 03-Sep-2006].

[13] D. Hearn and M. P. Baker. *Computer Graphics with OpenGL*. Prentice Hall, 3rd edition, 2004.

[14] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification (Version 2.1). `http://www.opengl.org/documentation/specs/version2.1/glspec21.pdf`, July 2006. [Online; accessed 04-09-2006].

[15] Microsoft Corporation. DirectX Software Development Kit. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx_sdk.asp`, 2006. [Online; accessed 04-09-2006].

[16] J. Kessenich. The OpenGL Shading Language. `http://www.opengl.org/registry/specs/ARB/GLSLangSpec.Full.1.20.6.pdf`, July 2006. [Online, accessed 04-09-2006].

[17] M. J. Kilgard. Cg and NVIDIA (Chapter 5 of SIGGRAPH 2006 course 3). `http://www.csee.umbc.edu/~olano/s2006c03/ch05.pdf`, 2006. [Online; accessed 26-September-2006].

[18] I. Buck et.al. BrookGPU. `http://graphics.stanford.edu/projects/brookgpu`, 2006. [Online, accessed 04-09-2006].

[19] RapidMind Inc. Sh: A high-level metaprogramming language for modern GPUs. `http://libsh.org`, 2006. [Online, accessed 04-09-2006].

[20] M. Harris. Mapping Computational Concepts to GPUs. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 31, pages 493–508. Addison-Wesley, Upper Saddle River, NJ, USA, 1st edition, 2005.

[21] Wikipedia. Comparison of NVIDIA Graphics Processing Units — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Comparison_of_NVIDIA_Graphics_Processing_Units&oldid=73077340`, 2006. [Online; accessed 4-September-2006].

[22] Wikipedia. NVIDIA Quadro — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=NVIDIA_Quadro&oldid=72555349`, 2006. [Online; accessed 4-September-2006].

[23] I. Buck. Taking the Plunge into GPU Computing. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 32, pages 509–519. Addison-Wesley, Upper Saddle River, NJ, USA, 1st edition, 2005.

[24] D. Salomon. *Data Compression: The Complete Reference*. Springer, 3rd edition, 2004.

[25] Wikipedia. Discrete cosine transform — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Discrete_cosine_transform&oldid=81725562`, 2006. [Online; accessed 22-October-2006].

[26] M. Frigo and S. G. Johnson. FFTW Home Page. `http://www.fftw.org`, 2006. [Online; accessed 22-October-2006].

[27] M. Frigo and S. G. Johnson. 1d Real-even DFTs (DCTs) — FFTW 3.1.2. `http://www.fftw.org/fftw3_doc/1d-Real_002deven-DFTs-_0028DCTs_0029.html`, 2006. [Online; accessed 22-October-2006].

[28] W. B. Pennebaker and J. L. Mitchell. *JPEG still image data compression standard*. Springer, 1st edition, 1993.

[29] IEEE Circuits and Systems Society. IEEE standard no. 1180 — specifications for implementation of 8 × 8 inverse discrete cosine transform. Technical report, 1991.

[30] H. S. Malvar. *Signal Processing with Lapped Transforms*. Artech House, 1st edition, 1992.

[31] G. Davis and A. Nosratinia. Wavelet-Based Image Coding: An Overview. *Applied and Computational Control, Signals, and Circuits*, 1(1), 1998.

[32] H. V. Sorenson, D.L. Jones, M. T. Heideman and C. S. Burrus. Real-valued fast Fourier transform algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(6):849–863, 1987.

[33] D. Takahashi. An extended split-radix FFT algorithm. *IEEE Signal Processing Letters*, 8(5):145–147, 2001.

[34] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech, and Signal Processing. ICASSP '98.*, 3:1381–1384, 1998.

[35] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Compitation*, 19:297–301, 1965.

[36] J. W. Cooley *et al.* The 1968 Arden House Workshop on Fast Fourier Transform Processing. *IEEE Transactions on Audio and Electroacoustics: Special issue on Fast Fourier Transform*, AU-17(2), 1969.

[37] J. Makhoul. A fast cosine transform in one and two dimensions. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(1):27–34, 1980.

[38] Y. Arai, T. Agui and M. Nakajima. A fast DCT-SQ scheme for images. *Transactions of the Institute of Electronics, Information and Communication Engineers*, E71(11):1095–1097, 1988.

[39] B. D. Tseng and W. C. Miller. On Computing the Discrete Cosine Transform. *IEEE Transactions on Computers*, C-27(10):966–968, 1978.

[40] S. Winograd. On Computing the Discrete Fourier Transform. *Mathematics of Computation*, 23(141):175–199, 1978.

[41] E. Feig and E. Linzer. Scaled DCT Algorithms for JPEG and MPEG Implementations on fused multiply/add architectures. *Proceedings of the SPIE Conference on Image Processing Algorithms and Techniques; San Jose, CA; (USA); 25 Feb.-1 Mar. 1991*, pages 458–467, 1991.

[42] E. Feig and E. Linzer. Discrete Cosine Transform Algorithms for Image Data Compression. *Electronic Imaging '90 East Proceedings*, pages 84–87, 1990.

[43] S.-L. Chang and T. Ogunfunmi. A fast DCT (Feig's algorithm) implementation and application in MPEG1 video compression. *Proceedings of the 38th Midwest Symposium on Circuits and Systems, 1995*, 2:961–964, 1995.

[44] Y. Jeong and I. Lee. Fast discrete cosine transform structure suitable for implementation with integer computation. *Optical Engineering*, 39(10):2672–2676, 2000.

[45] B. Fang, G. Shen, S. Li and H. Chen. Techniques for efficient DCT/IDCT implementation on generic GPU. *IEEE International Symposium on Circuits and Systems, ISCAS 2005, 23.-26 May 2005*, 2:1126–1129, 2005.

[46] H. S. Malvar and D. H. Staelin. The LOT: transform coding without blocking effects. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(4):553–559, 1989.

[47] P.M. Cassereau, D.H. Staelin and G. de Jager. Encoding of images based on a lapped orthogonal transform. *IEEE Transactions on Communications*, 37(2):189–193, 1989.

[48] B. Hinman, J. Bernstein and D. Staelin. Short-space Fourier transform image processing. *IEEE International Conference on Acoustics, Speech, and Signal Processing, 1984 (ICASSP'84)*, 9:166–169, 1984.

[49] R.L. de Queiroz, T.Q. Nguyen and K.R. Rao. The GenLOT: generalized linear-phase lapped orthogonal transform. *IEEE Transactions on Signal Processing*, 44(3):497–507, 1996.

[50] T. Nagai, M. Ikehara, M. Kaneko and A. Kurematsu. Generalized unequal length lapped orthogonal transform for subband image coding. *IEEE Transactions on Signal Processing*, 48(12):3365–3378, 2000.

[51] L. C. Duval and T. Q. Nguyen. Seismic data compression: a comparative study between GenLOT and wavelet compression. *Proceedings of SPIE conference on Wavelet Applications in Signal and Image Processing VII*, 3813(1):802–810, 1999.

[52] L. C. Duval and T. Nagai. Seismic data compression using GULLOTS. *Proceedings of 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'01)*, 3:1765–1768, 2001.

[53] S. Trautmann and T. Q. Nguyen. GenLOT — Design and Application for Transform-Based Image Coding. *1995 Conference Record of the Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, 1:41–45, 1995.

[54] NVIDIA Corporation. Cg Toolkit: User's Manual. A part of the Cg toolkit, available from `http://developer.nvidia.com/object/cg_toolkit.html`, September 2005. [Release 1.4].

[55] J. Owens. Streaming Architecture and Technology Trends. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 44, pages 457–470. Addison-Wesley, Upper Saddle River, NJ, USA, 1st edition, 2005.

Appendix A

Program Overview

The dct program is a front-end program for all the algorithms (including LOT algorithms) implemented in this project. Using this program, benchmarking different algorithms can be done very easily. The program reads an input file, applies a user-specified 8 × 8 DCT/IDCT or LOT/ILOT algorithm to the (square) input, and writes the result to a file — which can be /dev/stdout. The wallclock and CPU user and system time used for computing the transform is also displayed.

To run and compile the code, the following is recommended:

- Linux with kernel version 2.6, running on an x86-64 CPU, with standard UNIX development tools such as make
- Intel C Compiler version 9.1 (preferably), or the GNU C Compiler version 4.2 or more recent
- GNU C library version 2.4 or more recent
- A working X.org installation with NVIDIA GPU drivers version 1.0.8776 or more recent (available from http://www.nvidia.com). GLUT (GL Utility Toolkit) is also required.
- NVIDIA Cg toolkit 1.5 or more recent (available from http://developer.nvidia.com)
- FFTW library version 3.1.2 or more recent (available from http://www.fftw.org). Note that single (fftwf), double (regular fftw) and long double (fftwl) precision versions of FFTW must be installed. Also, threaded FFTW libraries should be installed.

All program code has been developed and tested on 64-bit x86 Linux kernel version 2.6. The code is developed for recent NVIDIA GPUs. Thus, in order to run the GPU programs, an NVIDIA NV40 or G70 GPU or more recent is required. Note that you must also use NVIDIA's proprietary OpenGL drivers, as the open source drivers are currently very limited and have no support for shader programs.

The program can be compiled using the make command, which produces an executable file named dct. For convenience, pre-compiled statically and dynamically linked binaries are also provided on the CD as dct-static and dct-dynamic, but there is no guarantee that these will work on all systems.

The syntax for running the program is:

```
./dct <algorithm> <input file> <output file> <forward|inverse>
        [threads]
```

where `algorithm` is the DCT or LOT algorithm used, `input/output file` is the input and output files, `forward|inverse` specifies whether the forward or inverse transform should be computed, and `threads` specifies the number of threads to be used — this argument is ignored for all algorithms except the OpenMP and FFTW algorithms, and defaults to 1.

The following DCT algorithms are available:

- `cpu-naive`: Naive, direct separable DCT on CPU
- `cpu-fftw-simple`: FFTW library DCT algorithm on CPU, using the "simple" interface of FFTW
- `cpu-fftw-advanced`: FFTW library DCT algorithm on CPU, using the "advanced" interface of FFTW
- `cpu-aan`: AAN DCT algorithm on CPU
- `cpu-flmad`: Feig-Linzer multiply-add DCT algorithm on CPU
- `cpu-aan-openmp`: AAN DCT algorithm on CPU, OpenMP version
- `cpu-flmad-openmp`: Feig-Linzer multiply-add DCT algorithm on CPU, OpenMP version
- `gpu-aan`: AAN DCT algorithm on GPU
- `gpu-flmad`: Feig-Linzer multiply-add DCT algorithm on GPU

The following algorithms are available for LOT computation:

- `cpu-flot`: Fast LOT on the CPU.
- `cpu-flot-openmp`: Fast LOT on the CPU, OpenMP version.
- `gpu-flot`: Fast LOT on the GPU

The program can use any of these algorithms to compute the forward or inverse DCT or LOT.

Input files which define 2D data sets must be passed to the program. Due to space restrictions on the CD, only small files are included. To generate larger input sets, a simple `generate` program, located in the `inputfiles` subdirectory of the `dct` directory, can be used. To compile the `generate` program, type `make` in the `inputfiles` subdirectory. This program is run as

```
./generate <data file> <size> <precision> <count> <lot|dct>
```

where `data file` specifies the output file, `size` specifies the size of the square input data set, and `precision` is the number of bits of precision (16, 32, 64 or 128 — but when using the GPU algorithms, 64 and 128-bit precision is not supported due to hardware limitations, and on the CPU, 16-bit is not supported — also due to hardware limitations). `count` specifies the number of data sets which should be written to the file (this is useful, for instance, if we want to run an algorithm 10 times and obtain the average computation time), and `lot|dct` specifies whether a LOT or DCT input data set must be generated[1]

The `dct` program will read the input files and write output files which can be read back by the program. Thus, if a forward DCT on a data set is read from `dct-in.txt` and written to `dct-out.txt`, the program can apply the IDCT to `dct-out.txt` (which would give the data in `dct-in.txt` as result). This also applies to a forward/inverse LOT.

[1] We must separate LOT from DCT data sets because the program will extend boundaries on LOT input sets. We will describe this more closely later in the chapter.

Data files are in plain text format, and several data sets can be defined in one file. A data set is specified in a data file as:

```
!<dct|lot>:<N>:<M>:<precision>
<element (0,0)> <element (0,1)> .. <element (0, M-1)>
...
<element (N-1,0)> <element (N-1,1)> .. <element (N-1, M-1)>
```

Thus, the data files start with a line with a signature (`!dct` or `!lot`), followed by the number of rows and columns in the data set (`<N>` and `<M>` — but only square data sets are supported), followed by how many bits of precision should be used in the calculation of the DCT of this data set[2]. Next, the actual data set is specified. Several data sets can be specified by including several such definitions to the same file.

[2] While the number of bits of precision be implemented as a command-line argument to the program, an advantage of including it in the input files is that it specifies how precise the input numbers are, which is useful if the numbers were previously calculated by the program.

Appendix B

Implementation Details

This chapter provides additional details on the implementations.

B.1 FFTW CPU Implementation Details

As we mentioned in Section 3.2.4, the FFTW computes a slightly different DCT than what we use. When FFTW computes the DCT, it does not include the C_f factor nor the $\sqrt{2/n}$ factor in Definition 1 in Section 3.2.4, *but* introduces a scale factor of 2 when computing the forward DCT. Consequently, inputs to or outputs from the FFTW DCT/IDCT routines must be scaled. When doing the forward transform, the following relations between the output values at (i, j) from the FFTW DCT in a block B, $F_{B,i,j}$, and the DCT values of Definition 1 in that block, $D_{B,i,j}$ can easily be deduced from Definition 1 and the FFTW library's DCT definition [27]:

$$D_{B,0,0} = \frac{4C_0^2}{64} F_{B,0,0} = \frac{1}{32} F_{B,0,0} \tag{B.1}$$

$$D_{B,i,0} = \frac{C_0/4}{4} F_{B,i,0} = \frac{C_0}{16} F_{B,i,0} \qquad (i = 1, \ldots, 7) \tag{B.2}$$

$$D_{B,0,j} = \frac{C_0/4}{4} F_{B,0,j} = \frac{C_0}{16} F_{B,0,j} \qquad (j = 1, \ldots, 7) \tag{B.3}$$

$$D_{B,i,j} = \frac{1/4}{4} F_{B,i,j} = \frac{1}{16} F_{B,i,j} \qquad (i, j = 1, \ldots, 7) \tag{B.4}$$

(Note that $C_0 = 1/\sqrt{2}$ was defined in Definition 1.) Similarly, it follows from the FFTW library's IDCT definition that the following scalings must be applied to an FFTW IDCT input block B with values $F_{B,i,j}$ to obtain the values $f'_{B,i,j}$, prior to FFTW plan execution (provided we did the scalings above when computing the forward DCT):

$$
\begin{aligned}
f'_{B,0,0} &= \frac{64}{4C_0^2}F_{B,0,0} = 32F_{B,0,0} && && \text{(B.5)}\\
f'_{B,i,0} &= \frac{4}{C_0/4}F_{B,i,0} = \frac{16}{C_0}F_{B,i,0} && (i = 1,\ldots,7) && \text{(B.6)}\\
f'_{B,0,j} &= \frac{4}{C_0/4}F_{B,0,j} = \frac{16}{C_0}F_{B,0,j} && (j = 1,\ldots,7) && \text{(B.7)}\\
f'_{B,i,j} &= \frac{4}{1/4}F_{B,i,j} = 16F_{B,i,j} && (i,j = 1,\ldots,7) && \text{(B.8)}
\end{aligned}
$$

Finally, IDCT values from FFTW must also be scaled (either prior to or after FFTW IDCT computation) by $(2n)^2 = 4n^2 = 256$ since FFTW scales the inverse values by $2n$ in each dimension. After this step, our original input data set is obtained.

We do these scalings in our code such that all algorithms calculate the same DCT values, so that performance comparisons are fair (since several floating-point operations are saved when using FFTW's definition of the DCT.)

By default, FFTW uses 64 bits of precision. When the input is 32 or 128 bits of precision, the code is the same, except for using, respectively, `fftwf_` (which uses `floats` instead of `doubles`) or `fftwl_` (`long doubles`) as a prefix to all FFTW functions and data structures instead of `fftw_`.

B.2 GPU Implementation Details

In this section, we provide further details on the GPU implementations.

B.2.1 The Steps to Compute the DCT Using the GPU

Step 1. OpenGL Initialization First, OpenGL is initialized. This involves creating an OpenGL coordinate system, describing how coordinates should be interpreted, and a *viewport*, which describes what part of the coordinate system should be viewed from the screen. Consider Listing B.1.

```
/* Set up coordinate system [0,0] x [W,H], parallel projection, creating
 * a 1:1 2D mapping between the "world" and the projection surface (pixels)
 */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0.0, dataIN->length_w, 0.0, dataIN->length_h, 0.0, 1.0);

/* Set up viewport to span all coordinates, creating a 1:1 mapping
 * between projection and viewport
 */
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0.0, 0.0, dataIN->length_w, dataIN->length_h);
```

Listing B.1: OpenGL initialization

This code initializes OpenGL for our purposes. First, the OpenGL projection matrix is set to use an *orthographic projection*, which is the simplest *parallel projection*. In parallel projections, parallel lines remain parallel independent of the depth value (unlike *perspective projections*, which are usually used for 3D animation and more realistic than parallel projection). Therefore, parallel projection ensures that we obtain a one-to-one mapping of the OpenGL (x, y) geometry coordinates and the input data set coordinates. After projection setup, the viewport is set up. We set a one-to-one mapping between viewport coordinates and projection coordinates, since that is precisely what we need, since we during computation will "render" all coordinates of the input data set. (For more information on OpenGL coordinate systems, consult [13].)

Step 2. Cg Initalization Next, the Cg subsystem is initialized. Cg library functions, such as `cgGLLoadProgramFromFile`, are executed, and the programs are loaded, compiled, and set up for execution on the vertex and fragment processors. As OpenGL initialization, this only has to be done once. After the programs are loaded, an application does not need to load the programs again at a later point of time in the execution, provided the OpenGL/Cg context is retained.

Steps 3, 4 and 5. Texture Upload, Computation and Download We discuss Steps 3, 4 and 5 together, as they are highly related.

How Textures are Used After Step 2, the input data set must be uploaded to the GPU. There are several ways to execute the upload, and some ways are quicker than others. Previously, using so-called *pbuffers* was the only way to upload data to the GPU. Using pbuffers was just a trick, since they were not intended for GPGPU usage, and were often not supported on all platforms. A better and more recent method is using *Framebuffer Objects (FBOs)*. FBOs are specifically designed for efficient data transfer between the GPU and the CPU, and is an extension to OpenGL, and are easy to use compared to pbuffers.[1]

To understand how we use framebuffer objects, we must describe how textures are used. The input data set, which is read from the input file, is uploaded as a texture to the GPU. Textures are normally used for applying a texture to a surface to make it look realistic or produce visual effects, but we use textures to represent the input data set, and we write to texture memory to store the result of the computation. Now, textures can have several properties:

- Textures can be 16-bit or 32-bit.
- Textures can have from one color channel up to four color channels (red, green, blue and alpha — RGBA)
- Textures can be rectangular or square

How should we use textures to do transform computation? We need to apply fragment programs to each pixel of the input texture, and write the result to an output texture. What format should the input and output textures be? We have several choices, and the choice we make can affect efficiency:

[1] Another technique, which was first proposed for GPGPU usage about two weeks prior to the end of this project, is using *Pixelbuffer Objects* (PBOs), which in some special cases (namely, when dealing with 16-bit data) can give better results than FBOs (but otherwise currently has about the same performance). We have used FBOs since they are more mature.

1. *Input texture is single-channel, output texture is single-channel (**single/single**).* In this case, the fragment program will be run once for each value in the output texture (since it is the output texture we are rendering to — it can be considered as our screen). Therefore, each fragment program execution will compute a single transform coefficient. This is not good, since we can not exploit SIMD parallelism at all! Fragment processors are capable of processing four floating-point values simultaneously, but with this scheme, we only compute *one* transform coefficient per execution![2]

2. *Input texture is single-channel, output texture is multi-channel (**single/multi**).* This scheme is better. Since the output texture is multi-channel (RGBA), each fragment program execution can write *four* values to the output texture. This means that four coefficients can be processed simultaneously by each fragment processor. But there is a problem: If we want to implement the fast LOT, Feig-Linzer multiply-add or AAN algorithms, we need to write *eight* values to the output texture, since these algorithms operate on 8-point blocks, and computing only four coefficients can be problematic. Furthermore, the fragment processors must "detect" whether or not they are going to compute the first four or the last four transform values. Fortunately, there is a solution: We use two multi-channel output textures! This technique is called *multiple rendering targets*, or MRT. Then, the fragment program writes the first four coefficients to the first texture, and the last four coefficients to the second texture.

3. *Input texture is multi-channel, output texture is single-channel (**multi/single**).* Unfortunately, the last method has a problem: Now, the transform coefficients are spread on two multi-channel textures. For the transform values to be useful, they must be merged into one large texture. Thus we need to develop code that *scatters* the values of *two* multi-channel textures into one large single-channel texture, which can be downloaded to the CPU or used for other computations[3]. (But it would be unwise to use this scheme when doing computation, for the same reasons as whith the single/single scheme.)

4. *Input texture is multi-channel, output texture is multi-channel (**multi/multi**).* This scheme is better than the single/single scheme, *but* requires superfluous memory accesses compared to using the single/multi and multi/single schemes combined. Consider the following case: Input values are stored along rows of the texture. The transform is applied, and the output is written directly to the multi-channel output texture. Everything is OK so far. But if we now do a *column* transform, then one fragment processor computing the coefficients for the first column requires all the "red" values of the column, while the fragment processor computing the coefficients for the second column requires all the "green" values, and so on. *But* when using multi-channel input textures, there is no way to just read one color channel from that texture. Either, the fragment processor must fetch all the RGBA values, or nothing. *This* leads to a problem: All the fragment programs will read $3 \times 8 = 24$ completely redundant values in *each* execution. Three fourths of the bandwidth used is not used for computation. Also, the column transform will be much slower than the row transform, since the row transform will only require one memory access per four values, while the column transform requires reading four values for *one* value. To a certain degree, we get the same problem with the multi/single scheme when we scatter, since single executions of fragment programs must read all four color channels when only one channel is required (but using the multi/single scheme must also be done at least once when using

[2] Using multiple rendering targets, this number would increase to four coefficients — but that is still not enough.

[3] The NVIDIA DCT implementation also uses scatters, but we use them in a slightly different way.

the multi/multi scheme in order compute a single, large texture at the end of computation). Since the multi/multi scheme leads to column transforms having an entirely different code than row transforms, and since they will likely be much slower than the row transforms, we have chosen to use the single/multi scheme combined with the multi/single scheme, which does not have these problems. The code turns out to be simpler since there is only one texture coordinate per input value when computing the transform, and very little difference, performance and code-wise, between the row and column transforms.

Summarizing our findings above, we have the following plan for executing the transform.

1. Upload the data set as a single-channel texture.
2. Execute fragment/vertex programs to do the computation. The results are written to two multi-channel output textures. Two multi-channel output textures are required since each fragment processor execution computs eight transform values simultaneously.
3. Execute another set of fragment/vertex programs which *as input* takes the two output textures from the previous textures, scatters[4] them, and writes the result back to the *original input texture* (which is single-channel).
4. This texture can be read by the CPU, or used in successive computations. For instance, to compute a 2D DCT, we first do the above steps for the DCT along the rows. Then, we change the fragment/vertex programs to compute the DCT along the columns, and use the single-channel texture containing the row transform values as input, and repeat the process (first a column transform is written to two RGBA textures, and are subsequently merged into one texture — which can be downloaded to RAM.)

Figure B.1 on the following page illustrates.

Notice that we need to “swap” the textures' role during the transform: Initially, the large single-channel texture is the input texture and the two RGBA textures are output textures. But at the scatter step, the two RGBA textures become input textures and the input texture becomes the output textures. This brings us back to where we started: FBOs can be used to easily “swap” the role of the textures, and also makes it easy to use MRT. Consider Figure B.1. We generate two FBOs. The first FBO is bound to the single-channel input texture of size $N \times N$, and the second FBO is bound to *two* RGBA textures of size $N/4 \times N/2$. Notice that the two RGBA textures can store $N \times N$ values, since each channel has four channels; thus we have $4 \cdot 2 \cdot (N/4 \times N/2) = N^2$. The reason we use $N/4$ values along the x-dimension is that the RGBA values are stored in the x-dimension, so each address value in the x-direction corresponds to *four* values in the y-direction. That is, the four channels of each texture element is stored along the x-direction (see Figure B.1). This leads to that the RGBA textures are not square, since $N/4 \times 4 = N$ values are stored along the x-direction, but $N/2$ values are stored along the y-direction. Practically, this has no significance, because our scatter programs take care of the job of mapping the elements of the two RGBA textures to the single-channel texture correctly.

Initializing FBOs Consider Listing B.2, extracted from the source code in Section **??** on page ??, which shows how texture and FBO initialization, upload and download is done. As is evident from the listing, using FBOs makes doing this very simple.

[4] We use the term *scatter* since the program will scatter packed RGBA values on to several single-channel values.

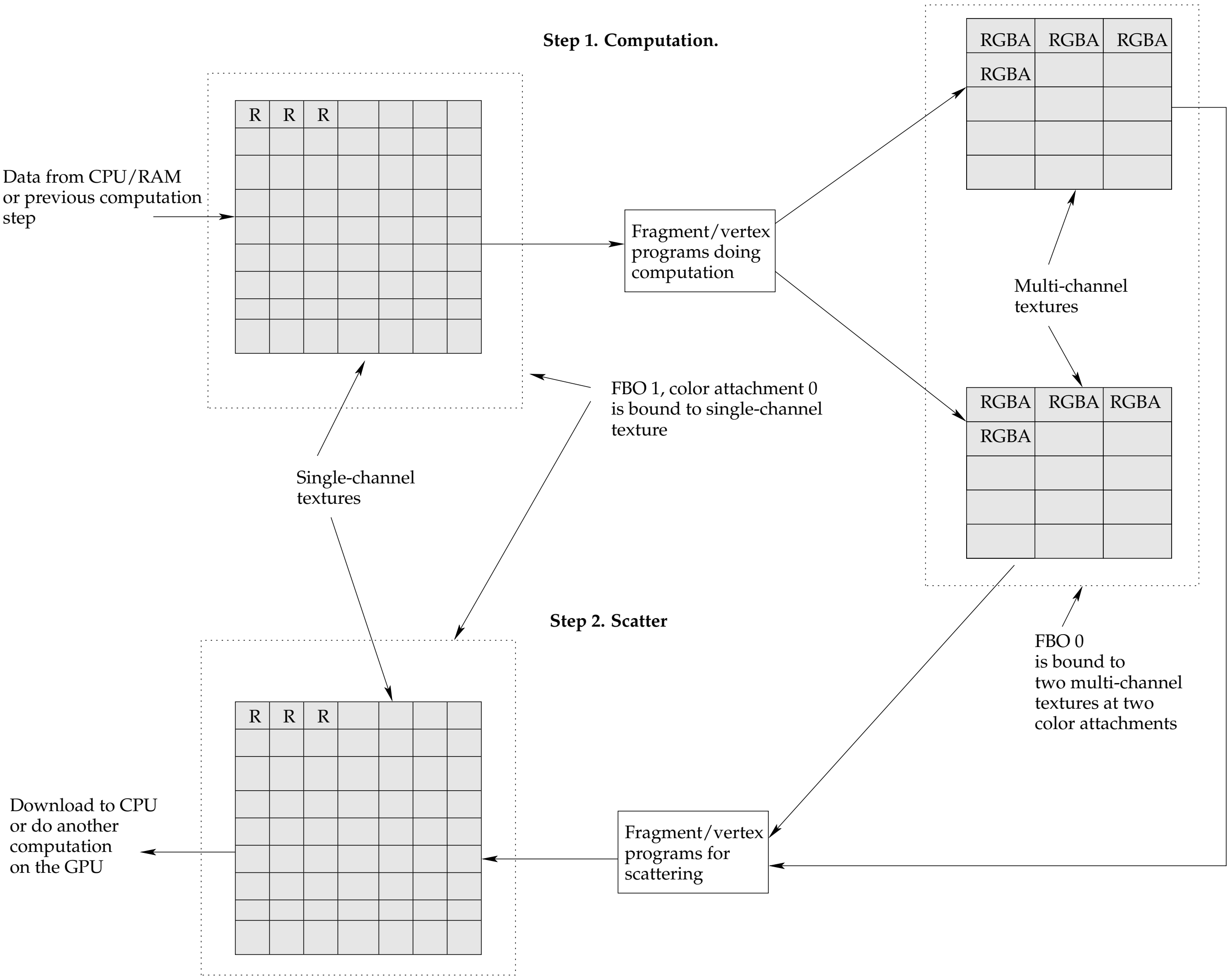
Step 1. Computation.
R R R
Data from CPU/RAM or previous computation step
Fragment/vertex programs doing computation
RGBA RGBA RGBA
RGBA
Multi-channel textures
RGBA RGBA RGBA
RGBA
FBO 1, color attachment 0 is bound to single-channel texture
Single-channel textures
Step 2. Scatter
FBO 0 is bound to two multi-channel textures at two color attachments
R R R
Download to CPU or do another computation on the GPU
Fragment/vertex programs for scattering

Figure B.1: Using textures for computation

```
/* ---- Initialize FBOs ---- */
static GLuint fbs[2];

/* Each texture associated with an FBO is associated with a color attachment.
 * In our case, the input texture is mapped to color attachment 0 on FBO 0,
 * and the two RGBA textures are mapped to color attachments 0 and 1 on FBO 1.
 */

GLenum drawbufs[] = { GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT };

/* Generate framebuffers */
glGenFramebuffersEXT( 2, &fbs[0] );

/* ---- Texture upload ---- */
/* Generate texture IDs */
glGenTextures( 1, &inout_texture ); /* Input texture */
glGenTextures( 2, out_textures );   /* (temporary) RGBA output textures */

/* Upload input texture to first color attachment on FBO 1.
   glBindFrameBufferEXT is required so that the glFramebufferTexture2D call
```

```
    below
 * applies to FBO 1
 */
glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fbs[1] );

/* glBindTexture is required so that subsequent calls apply to the correct
    texture */
glBindTexture( GL_TEXTURE_RECTANGLE_ARB, inout_texture );
/* We do not want any filtering, since we do not want values to be 'blurred'
    when we read them.
 * The following code disables filtering, and also disables unnecessary
     texture wrapping.
glTexParameteri( GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MIN_FILTER, GL_NEAREST )
    ;
glTexParameteri( GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MAG_FILTER, GL_NEAREST )
    ;
     glTexParameteri( GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_S,
         GL_CLAMP_TO_EDGE );
     glTexParameteri( GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_T,
         GL_CLAMP_TO_EDGE );

/* Specify the format and size of the texture. For the input texture we use
    GL_LUMINANCE (single-channel) with GL_FLOAT_R32_NV (internal GPU format).
 * For the RGBA textures we would use GL_RGBA with GL_FLOAT_RGBA32/16_NV
 */
if(dataIN->precision == 16)
  glTexImage2D( GL_TEXTURE_RECTANGLE_ARB, 0, GL_FLOAT_R16_NV, dataIN->length_w
      , dataIN->length_h, 0, GL_LUMINANCE, GL_FLOAT, NULL );
else if(dataIN->precision == 32)
  glTexImage2D( GL_TEXTURE_RECTANGLE_ARB, 0, GL_FLOAT_R32_NV, dataIN->length_w
      , dataIN->length_h, 0, GL_LUMINANCE, GL_FLOAT, NULL );


/* glTexSubImage2D actually uploads the data */
glTexSubImage2D( GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0, dataIN->length_w, dataIN->
    length_h, GL_LUMINANCE, GL_FLOAT, dataIN->d.f32_data);

/* glFramebufferTexture2DEXT links the
glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
    GL_TEXTURE_RECTANGLE_ARB, inout_texture, 0 );

// Repeat the steps above for the other two textures (but skip glTexSubImage2D
    --- see source code in the appendix

/*---- Texture download, after computation ----*/

/* Bind to the single-channel texture's FBO. */
glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fbs[1] );
/* Specify to read from attachment 0 of that FBO. */
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
/* Read */
glReadPixels( 0, 0, dataOUT->length_w, dataOUT->length_h, GL_LUMINANCE,
    GL_FLOAT, dataOUT->d.f32_data );
```

Listing B.2: FBO and texture operations

Executing the Computation One question remains: How do we actually execute the computation, and how do we load the vertex and fragment programs? First, the Cg programs must be loaded. This is done by the use of the Cg API, and is quite straight forward (see the source code in Section ??). The parameters of the Cg programs must also be set up. Parameters can either be *uniform*, which are constants, or *samplers*, which are objects the programs can fetch textures from. For example, when doing scattering, the size of the input data set is used to compute texture fetch coordinates. The size of the data set can be a uniform parameter, which is set from the CPU code using the function `cgGLSetParameter1f`. Then, at each program execution, the uniform parameter is available. Also, we must link the OpenGL textures to the Cg programs. This is done in a similar manner: We use the Cg API's functions to link the OpenGL textures to input samplers of the Cg programs using the `cgGLSetTextureParameter` function.

Several Cg programs can be loaded, but only one program may be bound to the vertex and fragment processors at a time. To specify the current active program, we use the `cgGLBindProgram` call.

Finally, we also need to actually start execution! This is done by simply rendering a square. Because there is a one-to-one relation between our viewport, texture coordinates and projection surface, we simply draw a large square covering the entire target texture. Thus when rendering in the scatter step to the single-channel texture, our square will be of size $N \times N$, and when rendering to the two RGBA textures during computation, our square will be of size $N/4 \times N/2$. The fragment programs are executed for each pixel in the textures, and the vertex programs are executed for each vertex in the square. To induce minimal overhead on the CPU when computing, we use *display lists* to store the textures. Display lists are sequences of OpenGL commands. We create one display list for generating a quadrilateral covering the input target texture (to which the scatter programs render), and another one for covering the RGBA textures (to which the computation programs render). Listing B.3 shows an outline of the code for computing the DCT (again, for details, see the source code or the appendix). Computing the LOT, as we will see, consists of more steps.

```
/* ---- Display list initialization (done only once) ---- */

GLint quadlists[2];
/* We use display lists to draw the quads, instead of doing the glVertex()
    calls when computing.
 * This makes the CPU part of the program run quicker.
 */

quadlists[0] = glGenLists(2);
quadlists[1] = quadlists[0] + 1;

/* Note that the texture coordinates specified are visible to the GPU shader
    programs. */

glNewList(quadlists[0], GL_COMPILE);
  glBegin(GL_QUADS);
    glTexCoord2f( 0.0, 0.0 );
    glVertex2f( 0.0, 0.0 );
    glTexCoord2f( dataIN->length_w/4.0, 0.0 );
    glVertex2f( dataIN->length_w/4.0, 0.0 );
    glTexCoord2f( dataIN->length_w/4.0, dataIN->length_h/2.0 );
    glVertex2f( dataIN->length_w/4.0, dataIN->length_h/2.0 );
    glTexCoord2f( 0.0, dataIN->length_h/2.0 );
    glVertex2f( 0.0, dataIN->length_h/2.0 );
  glEnd();
glEndList();

/* Dividing dataIN->length_w by 4 will become apparent in subsequent sections
```

```
     */

  glNewList(quadlists[1], GL_COMPILE);
    glBegin(GL_QUADS);
      glTexCoord2f( 0.0, 0.0 );
      glVertex2f( 0.0, 0.0 );
      glTexCoord2f( dataIN->length_w/4.0, 0.0 );
      glVertex2f( dataIN->length_w, 0.0 );
      glTexCoord2f( dataIN->length_w/4.0, dataIN->length_h );
      glVertex2f( dataIN->length_w, dataIN->length_h );
      glTexCoord2f( 0.0, dataIN->length_h );
      glVertex2f( 0.0, dataIN->length_h );
    glEnd();
  glEndList();

  /* ---- Actual computation of the DCT---- */

  GLenum drawbufs[] = { GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT };

  /* Swapping is done simply by (1) swapping FBOs, and (2) specifying the
      drawing buffers of the new FBO. Doing this gives off-screen rendering to
      texture.
   */

    /* Pass 1: DCT on rows */
    /* Loads a passthrough vertex program and DCT row program */
    cgGLBindProgram(cgprog_passthrough_v);
    cgGLBindProgram(cgprog_row_f);
    /* Set up to render to the two RGBA textures on FBO 0 (color attachments 0
        and 1) */
    glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fbs[0] );
    glDrawBuffers(2, drawbufs);
    /* Actually draw the square. */
    glCallList(quadlists[0]);

    /* Pass 2: Convert 2xRGBA -> 1xR */

    cgGLBindProgram(cgprog_passthrough_v);
    cgGLBindProgram(cgprog_rowscatter_f);
    /* Set up to render to the single channel texture on FBO 1 (color attachment
        0) */
    glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fbs[1] );
    glDrawBuffer( GL_COLOR_ATTACHMENT0_EXT );
    /* Actually draw the square covering the entire single-channel square. */
    glCallList(quadlists[1]);

    /* Pass 3: DCT on columns */

    cgGLBindProgram(cgprog_col_v);
    cgGLBindProgram(cgprog_col_f);
    /* Set up to render to the two RGBA textures on FBO 0 (color attachments 0
        and 1) */
    glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fbs[0] );
    glDrawBuffers(2, drawbufs);
    /* Actually draw the square. */
    glCallList(quadlists[0]);

    /* Pass 4: Convert 2xRGBA -> 1xR */

    cgGLBindProgram(cgprog_colscatter_v);
    cgGLBindProgram(cgprog_colscatter_f);
    /* Set up to render to the single channel texture on FBO 1 (color attachment
        0) */
    glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fbs[1] );
    glDrawBuffer( GL_COLOR_ATTACHMENT0_EXT );
    /* Actually draw the square. */
    glCallList(quadlists[1]);

    /* Download is now ready! */
```

Listing B.3: Computation and Scattering

An illustration of how we use the textures was provided in Section 4.2.1.

B.2.2 DCT Program Details

Step 1. Row Transform The row transform on the GPU is done by the passthrough vertex program and the row transform fragment program. These programs are listed in Listings B.4 and B.5, respectively.

```
void dct_gpu_flmad_passthrough_vertex( in  float2 tc      : TEXCOORD0,
          in  float4 pos      : POSITION,
          in  float4 color     : COLOR,

          out float2 tc0      : TEXCOORD0,
          out float4 pos0     : POSITION,
          out float4 color0    : COLOR,

          uniform float4x4 mvp)
{
  tc0 = tc;
  pos0 = mul(mvp, pos);
  color0 = color;
}
```

Listing B.4: Vertex processor passthrough program

Listing B.4 shows the vertex passthrough program. This is the simplest vertex program that is possible to write. It passes on the texture coordinate position and the color (which actually is irrelevant, but must be passed on). However, the position must be modified. Since we are using a vertex program, we must multiply the position by the model-view-projection (MVP) matrix, since if not, our viewpoint will not be located at the correct place (which is right in front of the data set). Hence we use the operation `mul(mvp,pos)`, which is a built-in Cg function to perform a matrix-vector multiplication. The transformed position is passed down to the rest of the pipeline.

Note that if we do *not* use vertex programs, the GPU will automatically multiply positions by the MVP matrix. But when we use vertex programs, we have to do it in the vertex shader program code.

Also notice that, unlike C, Cg has native vector (`float2`, `float4`) and matrix (`float4x4`) data types.

```
void dct_gpu_fflmad_row_fragment(
        in  float2 tc       : TEXCOORD0,

        out float4 out0     : COLOR0,
        out      float4 out1      : COLOR1,

        uniform samplerRECT in0,
        uniform float xSize,
        uniform half xSize_8)
{
  float v0, v1, v2, v3, odd_row;
  float4 dctin0_3, dctin4_7;

  /* xSize_8 is a uniform parameter set by the C code equal to the horizontal
     length divided by 8,
   * while xSize is just the horizontal input length
   */
  tc = floor(tc);
  odd_row = floor(tc.x/xSize_8);

  tc *= float2(8.0, 2.0);
  tc += float2(-odd_row * xSize, odd_row);


  /* Use vectorized addition to calculate texture coordinates
   * Also, the order of the computation is important: We calculate
   * the last four texture coordinates after the f1texRECTs have been issued
   * but before we need the values. This minimizes waste of time.
   */

  dctin0_3 = tc.xxxx + half4(0.0, 1.0, 2.0, 3.0);


  v0 = f1texRECT(in0, float2(dctin0_3.x, tc.y));
  v1 = f1texRECT(in0, float2(dctin0_3.y, tc.y));
  v2 = f1texRECT(in0, float2(dctin0_3.z, tc.y));
  v3 = f1texRECT(in0, float2(dctin0_3.w, tc.y));
  dctin4_7 = tc.xxxx + half4(4.0, 5.0, 6.0, 7.0);
  dctin0_3 = float4(v0, v1, v2, v3);
  v0 = f1texRECT(in0, float2(dctin4_7.x, tc.y));
  v1 = f1texRECT(in0, float2(dctin4_7.y, tc.y));
  v2 = f1texRECT(in0, float2(dctin4_7.z, tc.y));
  v3 = f1texRECT(in0, float2(dctin4_7.w, tc.y));
  dctin4_7 = float4(v0, v1, v2, v3);

  /* Actual FL MADD algorithm */
  dct_gpu_fflmad( dctin0_3, dctin4_7 );

  /* Note that dctin0_3 and dctin4_7 contain the DCT coefficients after calling
     dct_gpu_fflmad.
   * Cg supports call by reference.
   */
  out0 = dctin0_3;
  out1 = dctin4_7;
}
```

Listing B.5: Row DCT on the GPU — Fragment program

The fragment program in Listing B.5 executes the forward row transform on 8 points. Notice that this fragment program first does some computations on the texture coordinates. This is necessary, since the texture coordinates passed to the fragment program is the location of the current pixel in the *output* texture, that is, the RGBA texture. However, we are fetching data from the single-channel texture. We hence need a function to map from our texture coordinates (in the two RGBA textures) to eight input values in the single-channel texture.

The texture computations in the code make sure that one coordinate position in the two RGBA textures maps to eight distinct values on the same row in the input data set. This should be apparent if you consider Figure 4.5 on page 55, which provides an illustration.

In the figure, the x texture coordinate of the texture element is shown inside the element. Note that one coordinate in the single-channel texture refers to exactly one value, while one coordinate in the RGBA textures refers to four values. In the initial part of the code in Listing B.5, we map the RGBA texture coordinates to the single-channel texture coordinates. Then we fetch the corresponding row, compute the row DCT, and finally write out four values to the first RGBA texture and four values to the second texture.

After computing the texture coordinates, the program uses the internal `f1texRECT` Cg function to fetch the row starting at those coordinates, and then, the DCT is computed, and finally written to each RBGA texture. Note that nearly all operations are vectorized. In general, it pays off to vectorize Cg code as much as possible, as we will see later. We will also consider the actual FL MADD algorithm (the `dct_gpu_fflmad` function) later. The AAN implementation is similar, but calls `dct_gpu_faan` instead. Computing the inverse row transform is done in precisely the same way, but we call `dct_gpu_iflmad` instead of the forward transform function.

Step 2. Row scatter Row scatter also uses the passthrough vertex program, since no computations it does can be done by the vertex processor (more on this later). The fragment program simply computes a mapping from the single-channel texture coordinates to each of the two RGBA textures, and fetches the correct value from the RBGA textures and puts them into the single-channel texture at the correct location. Thus the program's job is to pick the right value from the two RGBA coordinates. The row scatter program is displayed in Listing B.6.

```
void dct_gpu_flmad_rowscatter_fragment( in  float2 tc      : TEXCOORD0,
          out float  out0      : COLOR0,
          uniform samplerRECT in0,
          uniform samplerRECT in1,
          uniform half xSize_8)
{

  half2 oddrowcol;
  half4 componentselect;
  float4 v0, v1;

  /* xSize_8 is the horizontal input set size divided by 8. This can be stored
   * in half precision since xSize_8 is always < 2048.
   */

  componentselect = (frac(tc.x) == half4(0.125, 0.375, 0.625, 0.875));
  oddrowcol = floor(fmod(tc, 2));
  tc = floor(tc - oddrowcol)/2.0;
  tc.x += oddrowcol.y * xSize_8;

  v0 = f4texRECT(in0, tc);
  v1 = f4texRECT(in1, tc);

  out0 = dot(lerp(v0, v1, oddrowcol.x), componentselect) ;
}
```

Listing B.6: Row scatter on the GPU — Fragment program

This program essentially does essentially the reverse of the previous program, in that the previous program computed the mapping function from the two RGBA textures to the single-channel texture, while this program computes the mapping from the single-channel texture to the two RGBA textures. This program illustrates a few important principles when programming the GPU: *Compute, don't branch* and *Use internal Cg functions*. The program first computes the mapping of the texture coordinates of the single-channel texture to the texture coordinates of the two RGBA textures. But we have a problem. The first problem is that we do not know what RGBA texture contains our value (there are two — which one do we choose?) And second, even if we knew which texture contained our value and the address of the element, the texture element at that address would have *four* components. How do we know what component we should pick?

Let us first look at the component selection problem — i.e. assume we have the address of the texture element in the RGBA texture, but do not know which component (R, G, B or A) we are going to store. Recall from Listing B.3 on page 90 that we divided the x texture coordinates by four. Normally, texture coordinates in the x-direction for the single channels are passed as:

```
[ 0.5 1.5 2.5 3.5 4.5 ... ]
```

by the GPU. But dividing by four in the `glTexCoord` call gives us the sequence

```
[ 0.125 0.375 0.625 0.875 1.125 ... ]
```

This is why we divided the texture coordinate by four in Listing B.3, and this gives us the key to choosing what component of the RGBA texture element we are interested in. Assume we know the location of the RGBA texture element. Then, if the fraction value, that is $x - \lfloor x \rfloor$, of our texture coordinate is 0.125, we choose the R component. If it is 0.375, choose the G component. And so on. In fact, this can be expressed as a dot product! While we *could* do something like[5]

```
  if(frac == 0.125) { out0 = textureElement.x; }
  else if(frac == 0.375) { out0 = textureElement.y; }
  else if(frac == 0.625) { out0 = textureElement.z; }
  else { out0 = textureElement.w; }
```

a dot product will be at least 12 times faster. I kid you not. Dot products take at most 1 cycle, while if-elseif-elseif-else takes at least 12 cycles.

Here is the idea, implemented in Listing B.6. We first do a vector comparison in the code above (we compare the fractional value of the x-coordinate to the four possible values in the `componentselect` variable). Then, if we have the four-component value in a variable `x`, the *same effect* as the above if-statements is obtained by simply computing the dot product

```
out0 = dot(textureElement, componentselect);
```

So that problem is certainly solved.

Let's now look at how we can obtain the texture coordinates in the RBGA texture. Consider Figure 4.5 on page 55 again. Let us look at an example and do the computation step by step. Suppose we have 16 elements in the x-direction and just two elements in the y-direction. The texture coordinates in the x-direction passed to the row scatter programs are[6]

```
[0.125 0.375 0.625 0.875|1.125 1.375 1.625 1.875|2.125 2.375 2.625 2.875|3.125 3.375 3.625 3.875]
[0.125 0.375 0.625 0.875|1.125 1.375 1.625 1.875|2.125 2.375 2.625 2.875|3.125 3.375 3.625 3.875]
```

[5] Note that the RGBA components are accessed in Cg by suffixing the vector value with `.x`, `.y`, `.z` and `.w`, respectively.

[6] Note that only one coordinate is passed in each execution. This is done automatically by the GPU.

Taking modulo 2 of these values and then computing the floor function gives us (this is the `oddrowcol` variable in the program)

```
(*)
[0.000 0.000 0.000 0.000|1.000 1.000 1.000 1.000|0.000 0.000 0.000 0.000|1.000 1.000 1.000 1.000]
[0.000 0.000 0.000 0.000|1.000 1.000 1.000 1.000|0.000 0.000 0.000 0.000|1.000 1.000 1.000 1.000]
```

for the x-components, and taking modulo 2 for the y-components (simply 0 for the first line and 1 for the second) gives

```
(**)
[0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000]
[1.000 1.000 1.000 1.000|1.000 1.000 1.000 1.000|1.000 1.000 1.000 1.000|1.000 1.000 1.000 1.000]
```

for the y-components. Thus `oddrowcol` just says whether the row and column coordinates are an odd number or not.

Now, subtracting the modulo 2 values from the original values, flooring and dividing by 2, componentwise, gives

```
(***)
[0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000|1.000 1.000 1.000 1.000|1.000 1.000 1.000 1.000]
[0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000|1.000 1.000 1.000 1.000|1.000 1.000 1.000 1.000]
```

for the x-components, and

```
(****)
[0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000]
```

for the y-components.

Finally, add the modulo 2 values ***for the y-component*** `(**)` multiplied by the input data set length divided by eight (in this case, 2) to `(***)` (***the x-component***) above. Then we get, finally, the texture coordinates

```
[0.000 0.000 0.000 0.000|0.000 0.000 0.000 0.000|2.000 2.000 2.000 2.000|2.000 2.000 2.000 2.000]
[1.000 1.000 1.000 1.000|1.000 1.000 1.000 1.000|3.000 3.000 3.000 3.000|3.000 3.000 3.000 3.000]
```

in the x-direction, and the texture coordinates in the y-direction are given by `(****)`.

Wasn't that an interesting pattern. It seems that we have *obtained the texture coordinates in the RGBA texture!* Just consider Figure 4.5 on page 55 again, and you will see that that is in fact the case. All we have to do now is to choose whether we are going to fetch from the first or second RGBA texture. Assume that we have fetched the four-component value from the first texture in value `v0` (using the `f4texRECT` Cg function) and the other in `v0`. Assume we set `oddrowcol.x` to `(*)` and `oddrowcol.y` to `(**)`. Then the correct output value is simply calculated by

```
out0 = dot( (1 - oddrowcol.x) * v0 + oddrowcol.x * v1), componentselect );
```

We may calculate the inner part by using the built-in Cg function `lerp` (linear interpolation), which gives us the last line in the fragment program in Listing B.6 on page 94.

Finding the correct mapping between the textures is quite tedious (and took quite a lot of time during the project).

Step 3. Column transform The column transform fragment program is shown in Listing B.8 below. But we also use a vertex program for certain computations! This program is listed in Listing B.7 below.

```
void dct_gpu_flmad_col_vertex(     in   float2 tc       : TEXCOORD0,
           in   float4 pos      : POSITION,
           in   float4 color     : COLOR,

           out  float2 tc0      : TEXCOORD0,
          out float2 tc1        : TEXCOORD1,
          out float4 pos0       : POSITION,
          out float4 color0       : COLOR,

          uniform float4x4 mvp,
          uniform half     xSize_4,
          uniform half   xSize_8)
{
  /* xSize_4 and xSize_8 are the horizontal input set size divided by 4 and 8,
     computed in CPU code */
  tc0 = tc;
  pos0 = mul(mvp, pos);
  color0 = color;
  tc1 = float2(tc.y/4.0, tc.y * xSize_4 - xSize_8);
}
```

Listing B.7: Column DCT on the GPU — Vertex program

```
void dct_gpu_fflmad_col_fragment( in  float2 tc          : TEXCOORD0,
        in  float2 tc1        : TEXCOORD1,

        out float4 out0       : COLOR0,
        out     float4 out1       : COLOR1,

        uniform samplerRECT in0,
        uniform float xSize)
{
  /* xSize is the horizontal length of the data set */


  float v0, v1, v2, v3, xaddr;
  float4 yaddr;
  float4 dctin0_3, dctin4_7;

  tc = floor(tc);
  xaddr = tc.x + fmod(tc1.y, xSize);
  yaddr = floor(tc1.x) * 8.0 + half4(0.0, 1.0, 2.0, 3.0);

  v0 = f1texRECT(in0, float2(xaddr, yaddr.x));
  v1 = f1texRECT(in0, float2(xaddr, yaddr.y));
  v2 = f1texRECT(in0, float2(xaddr, yaddr.z));
  v3 = f1texRECT(in0, float2(xaddr, yaddr.w));
  yaddr += 4.0;
  dctin0_3 = float4(v0, v1, v2, v3);
  v0 = f1texRECT(in0, float2(xaddr, yaddr.x));
  v1 = f1texRECT(in0, float2(xaddr, yaddr.y));
  v2 = f1texRECT(in0, float2(xaddr, yaddr.z));
  v3 = f1texRECT(in0, float2(xaddr, yaddr.w));
  dctin4_7 = float4(v0, v1, v2, v3);

  dct_gpu_fflmad( dctin0_3, dctin4_7 );

  out0 = dctin0_3;
  out1 = dctin4_7;
}
```

Listing B.8: Column DCT on the GPU — Fragment program

This is the first case in which we can use the vertex processor for something useful. Now, as with the row transform, we need a mapping from one texture coordinate in the two RGBA textures to a column of eight values in the R texture. This is where the vertex processor can do some computation. For each texture coordinate, we need the values `tc.y/4` and `tc.y * xSize_4 - xSize_8`. These are computed by the vertex program and passed to the fragment program in the texture coordinate parameter `tc1`.

One question immediately comes up. Can we move the other texture coordinate computations to the vertex processor as well? The answer is no. We can, for example, not compute the `floor` or `fmod` functions in the vertex processor, since the vertex program is run only *once per vertex*. So any texture coordinates it outputs is *linearly interpolated* over the entire quadrilateral. Thus non-linear functions such as will end up having a very wrong result. But we can move the scaling and addition computed prior to the nonlinear functions to the vertex processor, which saves us a few operations in the fragment processor.

Figure 4.6 on page 56 illustrates how columns in the single-channel texture map to the two RGBA textures. Since the RGBA are along rows, we "transpose" the columns into the two RGBA textures. The texture coordinate computations performed by the vertex and fragment programs compute the transformation from coordinates in the RGBA textures to columns inthe single-channel texture, as can easily seen by an example similar to that in the last section. (Like the other mapping functions, this one also took some time to perfect.)

Subsequently to the the texture coordinate computation, the fragment program fetches the eight column values from the single-channel texture, computes the DCT, and outputs the first and last four transform values in the first and last RGBA texture, respectively. Note that the DCT function called is the same function we used in the row transform. The AAN implementation is also exactly the same, but calls another DCT computation function. This is also the case when computing the inverse transform. Also, note that we are still using SIMD operations.

Step 4. Column scatter The column scatter is for the column transform program as the row scatter is for the row transform program. Given a position inside the single-channel output texture, the column scatter program's task is to store the correct value from one of the two RGBA textures at that location.

The column scatter fragment program is shown below, and is very similar to the row scatter program, except we now work on columns instead of rows. Also, we can use a vertex program for some computations. The column vertex and fragment programs are shown below in Listings B.9 and B.10.

```
void dct_gpu_flmad_colscatter_vertex( in  float2 tc      : TEXCOORD0,
          in  float4 pos      : POSITION,
          in  float4 color     : COLOR,

          out float2 tc0      : TEXCOORD0,
          out float2 tc1      : TEXCOORD1,
          out float2 tc2      : TEXCOORD2,
          out float4 pos0     : POSITION,
          out float4 color0    : COLOR,

          uniform float4x4 mvp,
          uniform half xSize_4)
{
  tc0 = tc;
  pos0 = mul(mvp, pos);
  color0 = color;
  tc1 = float2( tc.x * 4.0 / xSize_4, tc.y / 4.0 );
  tc2 = float2( tc.x * 4.0, tc.y / 8.0 );
}
```

Listing B.9: Column scatter on the GPU — Vertex program

```
void dct_gpu_flmad_colscatter_fragment( in  float2 tc     : TEXCOORD0,
           in  float2 tc1      : TEXCOORD1,
           in  float2 tc2      : TEXCOORD2,
           out float  out0     : COLOR0,
           uniform samplerRECT in0,
           uniform samplerRECT in1,
           uniform half xSize_4)
{

  half4 componentselect;
  half oddblock;

  float4 v0, v1;

  componentselect = (frac(tc1.y) == half4(0.125, 0.375, 0.625, 0.875));
  oddblock = fmod(floor(tc1.y), 2); /* oddblock = 0 => fetch from in0, else
     fetch from in1 */

  tc.y = floor( tc1.x ) + floor( tc2.y ) * 4.0;
  tc.x = floor( fmod(tc2.x, xSize_4) );

  v0 = f4texRECT(in0, tc);
  v1 = f4texRECT(in1, tc);


  out0 = dot(lerp(v0, v1, oddblock), componentselect);
}
```

Listing B.10: Column scatter on the GPU — Fragment program

The texture coordinate computations in these programs are essentially the inverse of the computations in the column computation programs. In Figure 4.6 on page 56, instead of mapping from the two RGBA textures to the single-channel texture, we now do the inverse mapping. It is easy to verify from the above programs that they compute the correct texture coordinates, just as we did when we dicussed the row scatter programs. Also note that we use the same technique for component selection as we did in that program, but we now work in the y-direction.

B.2.3 LOT Implementation Details

The GPU row second butterfly/$\widetilde{\mathbf{Z}}$-program is shown in Listing B.11.

```
#define LOT_COS_THETA1  0.9177546256839811411456038575494851  /* cos(0.13 * Pi)
     */
#define LOT_COS_THETA2  0.8763066800438635873081159039220626  /* cos(0.16 * Pi)
     */
#define LOT_COS_THETA3  0.9177546256839811411456038575494851  /* cos(0.13 * Pi)
     */
#define LOT_SIN_THETA1  0.3971478906347806137543773600194771  /* sin(0.13 * Pi)
     */
```

```
#define LOT_SIN_THETA2  0.4817536741017152749871915028721297  /* sin(0.16 * Pi)
      */
#define LOT_SIN_THETA3  0.3971478906347806137543773600194771  /* sin(0.13 * Pi)
      */



/* Fragment program for row 2nd butterfly steps and Z-blocks
 */

void lot_gpu_row_butterfly2Z_fragment(
        in   float2  tc        : TEXCOORD0,

        out  float4  out0      : COLOR0,
        out  float4  out1      : COLOR1,

        uniform samplerRECT in0,
        uniform float xSize,
        uniform half  xSize_8)
{
  half odd_col;
  float v0, v1, v2, v3, odd_row;
  float4 in0_3, in4_7;


  tc = floor(tc);
  odd_row = floor(tc.x/xSize_8);

  tc *= float2(8.0, 2.0);
  tc += float2(-odd_row * xSize, odd_row);



  /* Use vectorized addition to calculate texture coordinates
   * Also, the order of the computation is important: We calculate
   * the last four texture coordinates after the f1texRECTs have been issued
   * but before we need the values. This minimizes waste of time.
   */

  in4_7 = tc.xxxx + half4(4.0, 5.0, 6.0, 7.0);
  v0 = f1texRECT(in0, float2(in4_7.x, tc.y));
  v1 = f1texRECT(in0, float2(in4_7.y, tc.y));
  v2 = f1texRECT(in0, float2(in4_7.z, tc.y));
  v3 = f1texRECT(in0, float2(in4_7.w, tc.y));
  in0_3 = in4_7 + 4.0;
  in4_7 = float4(v0, v1, v2, v3);
  v0 = f1texRECT(in0, float2(in0_3.x, tc.y));
  v1 = f1texRECT(in0, float2(in0_3.y, tc.y));
  v2 = f1texRECT(in0, float2(in0_3.z, tc.y));
  v3 = f1texRECT(in0, float2(in0_3.w, tc.y));
  in0_3 = float4(v0, v1, v2, v3);


  // Butterfly - note that in0_3 refers to coeffs 0-3 of the NEXT block
  // When scattering in lot_gpu_butterfly2_row_scatter_fragment, we fix this up


  // odd_col checks that we are not out of bounds - if we are, we store zero.
  // this might be unnecessary but writing random values to memory is not clean
      , even on the GPU!
  odd_col = (tc.x/8.0 != xSize_8 - 1);

  out0 = odd_col * (in4_7 - in0_3) * 0.5;


  // The following code does the Z-blocks
  // Unfortunately, computing the Z-blocks involves some vector dependencies,
  // but we can at least calculate two values in each instruction
  // We use in0_3 as a temporary variable

  out1.xy = LOT_COS_THETA1 * out0.xy + LOT_SIN_THETA1 * float2(-out0.y, out0.x);
  out1.yz = LOT_COS_THETA2 * float2(out1.y, out0.z) + LOT_SIN_THETA2 * float2(-
      out0.z, out1.y);
  out1.zw = LOT_COS_THETA3 * float2(out1.z, out0.w) + LOT_SIN_THETA3 * float2(-
      out0.w, out1.z);

  out0 = out1;
  out1 = odd_col * (in4_7 + in0_3) * 0.5;
}
```

Listing B.11: The second butterflies and $\widetilde{\mathbf{Z}}$-block on the GPU — Fragment program

This program is similar to the fragment program for the row DCT, but there are a few differences. The texture coordinate computation is precisely the same. However, notice that we now fetch only the *last four* values (from the DCT + 1st butterfly) of our "current" block and the *first four* values of the next block. This should be evident from Figure 3.10 on page 43. The $\widetilde{\mathbf{Z}}$-part of the block is stored in the first RGBA texture, while the part only having a second butterfly and the 1/2 scaling is stored in the second RGBA texture. Also notice that the $\widetilde{\mathbf{Z}}$-block computation is difficult to parallelize, which is evident from Figure 3.11 on page 44, but we can at least compute two values simultaneously.

The row scatter program, which fetches the first four block values from the previous block, is shown in Listing B.12 below.

```
void lot_gpu_rowscatter_butterfly2Z_fragment(
          in   float2 tc       : TEXCOORD0,
          out  float  out0     : COLOR0,
          uniform samplerRECT in0,
          uniform samplerRECT in1,
          uniform half xSize_8)
{

  // First part omitted - precisely equal to dct_gpu_flmad_rowscatter_fragment

  v1 = f4texRECT(in1, tc);
  tc.x -= 1.0;
  v0 = f4texRECT(in0, tc);

  out0 = dot(lerp(v0, v1, oddrowcol.x), componentselect);
}
```

Listing B.12: Row scatter for second butterflies and $\widetilde{\mathbf{Z}}$-block on the GPU — Fragment program

In order to avoid cramming this report with too much code, and since the column programs are very similar to what we have previously seen, the column second butterfly/$\widetilde{\mathbf{Z}}$ and modified column scatter programs are not shown in the appendix, but is located on the CD. The program function names are `lot_gpu_col_butterfly2Z_fragment` and lot_gpu_colscatter_butterfly2Z_fragment (the fragment programs) and lot_gpu_colscatter_butterfly2Z_vertex (the vertex program for the column scatter). These programs do exactly the same as the row programs, but operate on columns instead. The column second butterfly/$\widetilde{\mathbf{Z}}$ computation program uses the same vertex program as the DCT column transform (implemented in lot_gpu_col_vertex, and does almost exactly the same as the program in Listing B.11, but operates on columns. As in that program, the first four LOT values of a block are shifted by four positions. The column scatter vertex and fragment programs are also simple modifications of the DCT column scatter programs. The only difference, as with the row scatter, is that the first four values of a column is fetched from the previous column (as was the case with the rows).

The C part of the GPU fast LOT is also similar to the C program for the GPU DCT C programs. This program can be found in Section ?? on page ??.