Solving Diffusion Curves on GPU

by

Jeff Warren, B.A. (Mod)

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

Interactive Entertainment Technology

University of Dublin, Trinity College

September 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Jeff Warren

September 13, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Jeff Warren

September 13, 2010

Acknowledgments

I wish to thank Daniel Sýkora for his input as a supervisor, both in proposing such an interesting idea and providing advice and help whenever possible.

I would also like to thank John Dingliana for kindly agreeing to proof read my report at short notice, and for his excellent mentoring throughout the course of the past year.

Jeff Warren

University of Dublin, Trinity College
September 2010

Solving Diffusion Curves on GPU

Jeff Warren
University of Dublin, Trinity College, 2010

Supervisor: Daniel Sýkora

Many tasks in computer graphics and vision produce a large sparse system of linear equations which typically requires a large amount of CPU time to be solved. Processing images which contain "diffusion curves" is one such example of this category of systems. Recently various GPU based solvers have been proposed allowing real-time processing and feedback for diffusion based images, however they have been closed systems which cannot be expanded and developed further. To mitigate this, we propose a linkable library which can be used by third party applications to easily abstract and solve diffusion curves, using available GPU hardware in a computer system. This allows applications which can provide feedback to artists working with large images, at unintrusive speeds. Both CPU and GPU based algorithms are provided, allowing support of legacy hardware. The library can also be compiled to run natively on 32 and 64 bit operating systems.

Using modest hardware, the users of such an application can edit and develop multi megapixel images at processing speeds in excess of 10 frames per second.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Diffusion curves are a vector based primitive used for describing smoothly shaded images or subsections of images [1]. Generating a final displayable image from a set of diffusion curves is a computationally expensive operation. This dissertation presents an accelerated library capable of solving such images at real-time speeds, and explores low level optimisation techniques in order to obtain the best performance possible on both CPU and GPU style platforms.

In recent years, there has been a rising interest in general purpose computing on non-CPU devices. Specifically, common GPU's available in standard computers have been successfully extended to enable high speed computation of non graphical workloads, though other experimental processors such as the Intel Larrabee and the IBM CELL Broadband Engine have also been developed. There are a number of challenges associated with developing applications suitable for this kind of hardware. Expressing the problem in a way that can be solved in a parallelised fashion is crucial - many data sets can be processed using algorithms which, while slower than their traditional linear single threaded counterparts, enable the reduction or elimination of data dependencies (which inhibit parallelism). As a result, processing the data can be often be completed faster on multiprocessor hardware.

Another problem is that of adapting to the "stream processing" paradigm. In traditional approaches to programming, much of the complexity of the processing unit is

hidden and abstracted away from the programmer - caches, memory hierarchies, even vectorisation can be ignored, often leading to a negligibly small impact on performance. Conversely, stream programming requires careful and direct manipulation of the data in order to achieve even barely acceptable performance - data and even code must be manually loaded and unloaded, sometimes asynchronously. This requires the programmer to have a low level understanding of their target device in order to achieve desirable results from their codebase. Stream processing is also still considered by many to be in its infancy - it has only become reasonably popular since the turn of the century. When targeting a GPU, for example, it is often unclear how future versions of the host hardware will develop. Linear scaling of performance cannot be assumed, especially with "problematic" algorithms which require a great deal of serialised execution.

This study presents a highly optimised solver for diffusion curves, with algorithms suited for both standard CPU style hardware, in addition to stream processors such as the GPU. While a version does not yet exist for the CELL Broadband Engine, a port from the GPU algorithm would be trivial.

Perceptual experiments carried out in the field of psychology have long suggested that the human biological image processing system does not simply measure intensity - it is highly adaptable, capable of managing to glean useful input in a wide range of brightness. Werner suggests that the brain measures local intensity differences rather than intensity itself [2], a phenomenon which can be demonstrated by many well-known optical illusions, including the one shown in Figure 1.1. As a result, some researchers find merit in the idea of creating images or artwork by working directly in the gradient domain, rather than the traditional approach of directly selecting colour intensities [3].

The greatest challenge when developing gradient domain image processing tools is that any local modification to the image has potentially global implications. Converging the image to a stable solution is computationally expensive and requires a large number of iterations. The complexity of the problem is directly proportional to the resolution of the image - an image with twice as many pixels will require at least twice as much computation to solve, in a best case scenario. Enabling constraints to interact with distant areas of the image also presents a bottleneck (this can largely be overcome by modifications to the algorithms, and is discussed later in the implementation).

Figure 1.1: A well known optical illusion demonstrating the gradient based method which the human visual system is suggested to use. The grey bar across the middle of the image has a constant intensity, however the varying shade in the background misleads the brain.

High resolution gradient domain images have previously taken several seconds (or even minutes for large rasters) to process. This has rendered the tools available for the task unintuitive to the artists, who require low latency feedback in order to unconstrain their creative expression. Many have expressed frustration at having to wait to see the effects of their modifications. Previous slow CPU based implementation evaluations have received a range of criticism, mainly centered around the long waits for feedback. As users of the program are accustomed to working with image manipulation applications whose tools provide mainly local modifications, it is essential to provide up to date results from a modification as quickly as possible in order to increase usability.

Previous studies have proposed accelerated, hierarchical integrators which calculate the stable state of the image at high speeds [3, 4]. However, while their software is readily available, the authors have not opted to release the source code, which means developing additional novel tools for allowing the artist to express the curve set is not currently possible. These versions are also implemented using shaders, which not only reduces portability, but can prevent the algorithm from achieving the peak performance possible by using stream processor interfaces. It also remains unclear how the shader versions will scale to improve on future hardware - the implementation we

present attempts to reduce this ambiguity. A set of open libraries are provided - not only could these be linked to by the existing artist tools provided by McCann et al, but additional brushes and other methods of input can easily be added by someone without an in-depth knowledge of the stream processing paradigm.

The presented work concentrates on understanding both the optimisation techniques relevant to accelerating diffusion based algorithms, in addition to the benefits an artist might enjoy by using a diffusion curve based tool set. A reusable library is provided, and the potential future applications it could have are explored and analysed. We also compare the library to other similar software for working with diffusion curves.

Chapter 2

Background

This section explores some of the aspects of hardware architecture with regard to optimisation of the codebase. An overview of CPU and GPU style platforms is provided, with some commentary on the various systems of branch prediction, caching, and parallelism available. The algorithms implemented are also explored at a high level.

While Moore's (revised) law states that computational processing power available doubles every 18 months [5] (and along with it, memory capacity), problem size expands to challenge these expanding resources. The solving of massive sparse linear systems, such as those expressing a set of diffusion curves, is a problem which has only recently become possible to solve in real-time.

November 2002 saw the release of the last Intel chip to obey Moore's law via serial clock speed increase [6]. With the 3GHz Pentium 4 launched, Intel found that transistor power leakage began to grow rapidly as they attempted to fit more transistors onto their chips [7]. The fastest retail Pentium 4 never exceeded 4GHz clock speed [8], and Intel, along with other processor manufacturers, were forced to explore alternative routes to obtain increased computational power in computer processors. Intel launched a hyper threaded (HT) CPU, which allowed for non-simultaneous multithreading [6]. By duplicating areas of the CPU which store architectural state (5% of the die area [9]), pipeline stalls due to branch misprediction, data dependencies, or cache misses would allow another thread to be scheduled in more quickly than a non-HT implementation would allow. Due to the low number of heavily multithreaded applications available

at the time, this approach flopped - the most processing power hungry applications (such as 3D gaming) had no support for multithreading. In addition, the overhead introduced by work splitting/synchronisation of data sets in non embarrassingly parallel algorithms often cancelled out any improvements shown by HT. Computer systems containing more than one independent processing unit were traditionally restricted to professional applications - hosting, clustering, supercomputing etc. The large cost of dual CPU systems, and lack of desktop applications designed to take advantage of them prevented them from entering the consumer market. However as home computer users became heavier multitaskers, and traditional approaches of simply raising clock speed to improve performance failed, dual and quad core CPUs appeared on the market, and quickly dropped in price to the point where it is now difficult to purchase a home computer **without** more than one execution unit.

2.1 Caches

Processing units execute instructions far more quickly than they can be fetched from main memory. A processor which had to wait for each instruction to be loaded from main memory would be massively underutilised. However, over the course of a program's execution, it can be observed that a very high percentage of memory accesses are to the same memory addresses, repeatedly. This is referred to as "temporal coherency". To take advantage of this behaviour, processing units contain a cache hierarchy, usually with 2-4 levels [7]. The further from the CPU the cache level is, the larger and slower to access it is likely to be.

Associativity is also important. Associativity level indicates the number of cache entries where a given memory location can be stored. With a 1-way cache (known as direct mapped) each memory location can only be cached in one location in the caching unit. Each entry in the cache (or, cache line) services many main memory addresses. This is usually determined using a LSB bit masking technique. If a program repeatedly accesses two memory locations which map to the same cache entry in a direct mapped cache, there will be a high number of cache misses, as the locations would need to be repeatedly read in from main memory. Increasing the degree of associativity improves cache performance, but is more costly to implement, as more locations need to have their address tags checked during each lookup. A policy needs to be introduced to

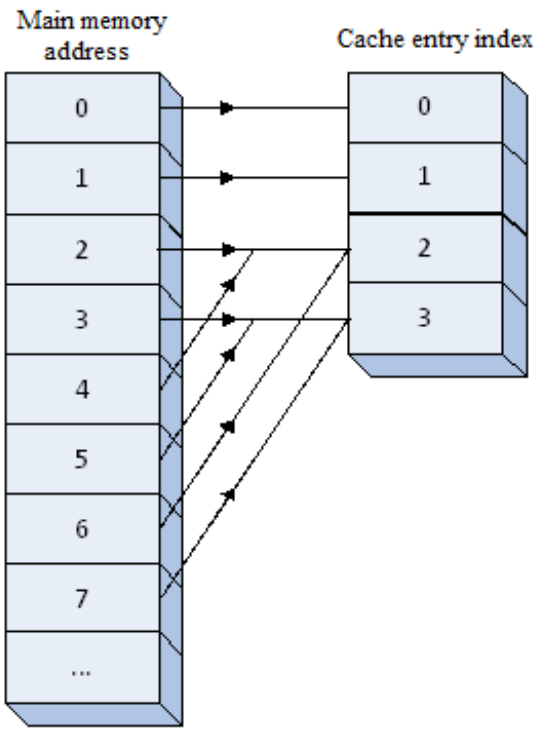

Figure 2.1: A direct mapped cache (associativity level of 1). Each address in main memory maps to one possible cache entry. This can cause cache thrashing/repeated misses if locations 0 and 5 are read alternately by the executing program. Direct mapped caches are however easier to implement in hardware than highly associative caches.

decide which of the possible cache entries will be overwritten with the new entry, e.g. Least Recently Used, pseudo Least Recently Used [10].

For multi-core systems, cache coherency protocol is necessary. In a single core computer, the CPU knows that the most up to date copy of a given memory address is going to be in the cache, or if it isn't cached, in main memory. When two CPU's are concurrently executing instructions, it is possible that a CPU can read a memory location which is out of date - because the other CPU could have just executed an instruction which modified the location. Cache coherency systems alleviate this problem, by maintaining the state of a cache line. One such example would be the MESI protocol [11]. A cache entry in the MESI system can be in 4 states.

- **Invalid**: the entry in this cache is not valid, as it has been written to by another

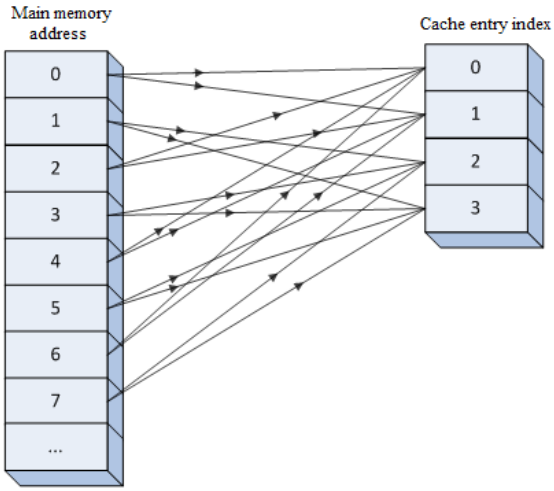

Figure 2.2: Cache with associativity level of 2. Each address in main memory maps to one of two possible cache entries. A replacement policy must be implemented for this to function (e.g. timestamping and Least Recently Used replacement). This makes better use of the cache space, since a program accessing locations 0 and 4 repeatedly will not result in as high a frequency of cache misses (the two pieces of data will likely be written into cache locations 0 and 1). Due to the extra storage needed for a replacement policy's data, this is more difficult to implement in hardware. Higher levels of associativity also introduce their own setback - in order to check for a cache hit or miss, extra cache tag entry comparators are required; each possible map-to cache location must be checked concurrently in hardware for each memory access.

CPU.

- **Exclusive**: this CPU is the only CPU to have a cache entry for this location. The entry matches the main memory's version (i.e. it is "clean").
- **Shared**: Main memory and other CPU's may have a copy of this location. All copies match, the entry is clean.
- **Modified**: Only the current CPU's cache has the latest version of this memory location. The entry is "dirty".

State transitions are initiated by observed bus reads/writes on the shared bus, and by internal processor reads/writes. The cache coherency protocol stores the state of each cache entry, and when a matching address is observed to be accessed by the

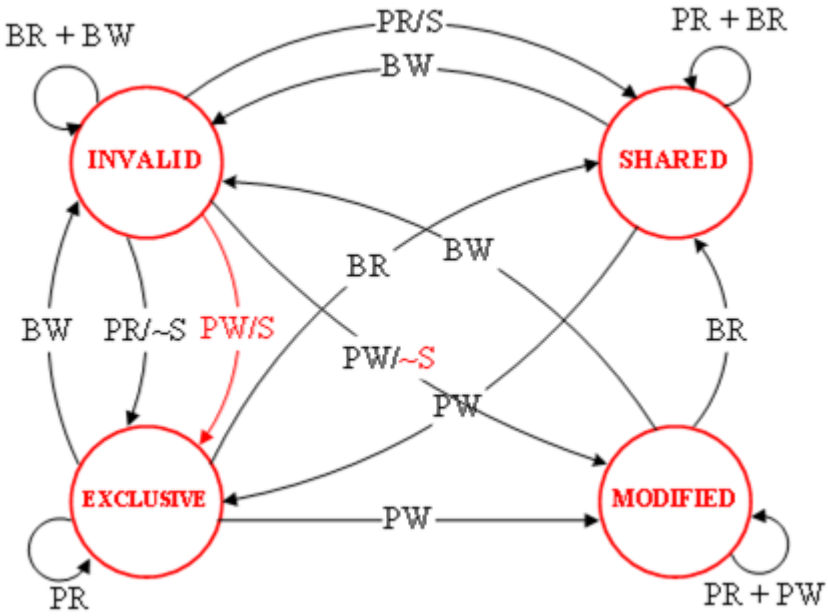

Figure 2.3: A state transition diagram for the MESI cache coherency protocol. For each cache entry, a 2 bit state representing which of the four states the cache line is in (Modified, Exclusive, Shared, Invalid). Transition is made between states via passive observation of the bus, and active reads/writes by the processor. BW and BR indicate writes and reads by other processors via observing the bus , respectively. PW and PR indicate active reads and writes to memory locations by the current processor. S and S indicate a shared/not shared operation.

processor, or another external processor the state is manipulated accordingly. Many protocols for this task require 4 states, which results in 2 bits to be stored for each cache entry. Protocols such as MESI allow for coherent caching to occur. This also enables different processors to cache different, potentially partially overlapping sets of memory locations. Fig. 2.3 shows a complete state transition diagram for the MESI protocol, though others are also in wide use (e.g. Firefly).

Repeated modification of memory addresses stored on the same cache line by different CPU's in a multiprocessor system causes repeated cache invalidation. It is important to note that the memory addresses (and associated variables) being modified by different processors need not be the same for this undesirable effect to occur; if two variables are compiled to exist in two aligned memory addresses, they may reside on the same cache line.

By understanding precisely how caches work, it is possible to tune algorithms such as those in the diffusion curve solver to perform optimally. In particular, multi datum cache lines provide for a speed increase if temporal locality can be exploited wherever possible.

2.2 CPU

General purpose CPUs tend to be feature packed - large caches, complex branch prediction and advanced pipelining make them very easy to achieve high performance on without needing to hand tune a codebase to target them.

SIMD

SIMD (Single Instruction Multiple Data) refers to computer processors with multiple Arithmetic Logic Units (ALUs) which execute the same instruction on multiple data simultaneously. The advantage provided by this architectural design is as follows: previously, a loop which needed to add 2 arrays (of length N) values into a destination array of the same length would have taken N iterations. This equates to N add operations in addition to N branches. SIMD extensions allow M data to be added to M data by four separate ALUs, meaning that N/M iterations of the loop are required. In practice this results in a massive speed-up for certain algorithms - particularly various kinds of multimedia processing.

Figure 2.4 shows a typical SIMD configuration - one instruction is executed on 4 pieces of data. Currently the majority of computer processors supporting SIMD extensions cater for 4 pieces of single precision data per operation (with some caveats). Some also provide support for manipulation of two double precision data together. There are plans to extend this to allow cooperation of 8 ALUs in the future.

In theory, SIMD can speed up the execution of an algorithm by a factor of 4. In some cases, even greater speed-ups may be attained. SIMD instructions take heavy advantage of the design of data caches. As most caches consist of sets of cache lines which contain multiple data, cache efficiency is raised. In addition to this, for SIMD implementations to yield any useful speed-up, operands must be loaded from aligned

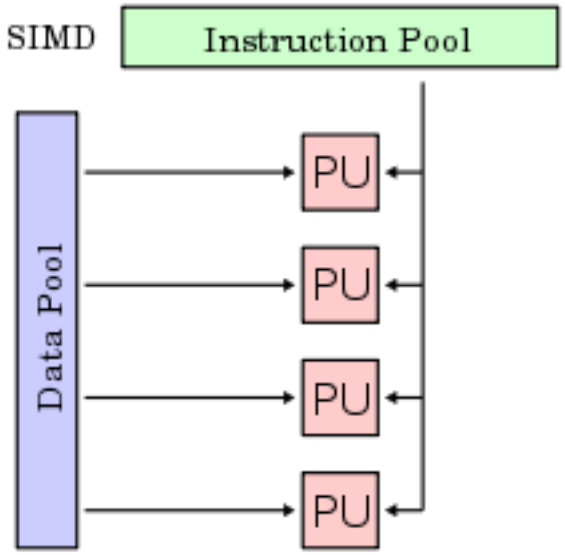

Figure 2.4: A 4-datum wide SIMD arrangement (Image released under GFDL).

data locations - i.e. for 4x4byte operands, the memory address of the first must be evenly divisible by 16. Cache lines are loaded in the same fashion.

However, SIMD also has disadvantages. Since flow control is restrictive, if different pieces of data in a block need to be treated differently, the algorithm needs to be executed strictly serially. The result is that only embarrassingly parallel algorithms have much to gain from SIMD; applications like parsing and branch-heavy decoding are unsuitable. Other disadvantages include the facts that since SIMD requires extra registers and additional ALUs, they require more floorspace on CPU dice, and also consume more power, resulting in chips being more expensive to manufacture and operate.

Stream processing such as that offered by CUDA represents a middle ground between strictly SISD processors and SIMD extensions - the thread warps are not unlike a SIMD instruction operation on multiple data, yet since they consist of collections of lightweight threads, there is additional flexibility in that per thread branching is possible.

2.2.1 Pthreads & Workload Scheduling

Pthreads, or POSIX threads, are a useful interface allowing programmers to create multi-threaded applications, capable of executing on many physical cores in a computer simultaneously [12]. This allows us to take advantage of the full amount of computational horsepower available, essential in applications such as the one presented. While Pthreads are primarily designed for UNIX systems, a port of the library to Win32 exists [13]. It was selected over the native Windows threading library to discourage lock-in - the diffusion library, in its current form, can easily be compiled for Windows, OSX, Linux and Unix based operating systems.

Parameters required for the worker threads must be passed in via a struct pointer, as indicated in Listing 2.1. While writing threading models yourself does allow for more fine tuning of exactly how the threads behave, the human effort required to rewrite the code and integrate it with C++ libraries is significant, and prone to error. OpenMP directives tend to be far quicker and more elegant to add, although the abstraction can lower performance in some cases.

When writing our own threaded versions of algorithms, a method for partitioning up the workload is necessary. There are three appropriate methods for implementing workload splitting for the purposes of this library.

- Static - The image is statically split into 4 equally sized areas, and a worker thread is assigned to each one.
- Work units - The image is split up into a large number of work units. Worker threads are then activated when work becomes available. They compete to complete allocated work units, until no more remain.
- Temporal - The image is tentatively split into 4 equally sized areas. An iteration is performed, and timings are measured by each. Using the time results from iteration N, the image is repartitioned to give more area to the threads that finished ahead of the slowest.

For problems where the work load tends to be evenly spaced, a static scheduler is sufficient. Static scheduling is also far easier to implement without introducing

Listing 2.1: A simple demonstration of POSIX threads.

```
1
2  #include <pthread.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <assert.h>
6
7  #define NUM_THREADS     5
8
9  void *TaskCode(void *argument)
10 {
11    int tid;
12
13    tid = *((int *) argument);
14    printf("Hello World! It's me, thread %d!\n", tid);
15
16    /* optionally: insert more useful stuff here */
17
18    return NULL;
19 }
20
21 int main (int argc, char *argv[])
22 {
23    pthread_t threads[NUM_THREADS];
24    int rc, i;
25
26    /* create all threads */
27    for (i=0; i<NUM_THREADS; i++) {
28       printf("In main: creating thread %d\n", i);
29       rc = pthread_create(&threads[i], NULL, TaskCode, (void *) &i);
30       assert(0 == rc);
31    }
32
33    /* wait for all threads to complete */
34    for (i=0; i<NUM_THREADS; i++) {
35       rc = pthread_join(threads[i], NULL);
36       assert(0 == rc);
37    }
38
39    exit(EXIT_SUCCESS);
40 }
```

situations where concurrency problems can happen - thread starvation, data collisions, and deadlock. Uneven workloads require work unit creation or temporal redistribution of work to maximise CPU usage.

The diffusion solver's iterations run in a constant time - there are no areas of the image which take significantly longer to process than others. Constraints are quicker - but in the average rasterised diffusion based image, the number of constrained pixels is a very low percentage of the total pixel count, so this becomes negligible. Using work units and temporal shifting also requires more locking and calculation, which does introduce another cost. A static scheduler for the diffusion portion of the calculation is provided, along with a basic work unit splitter.

2.2.2 OpenMP

OpenMP is a API which supports multi platform shared memory multiprocessing in C/C++ (and Fortran) [QUINN 2003]. It consists of a set of compiler directives and background libraries which allow for semi automatic parallelisation of a program. Clauses exist for:

- **data scoping** - indicating variables as shared, and creating per-thread private versions
- **synchronisation** - critical sections, which may only be executed by one thread at any one time, and atomic sections, which must be completely executed or not executed at all
- **scheduling** - division of a large work set into work chunks, for load balancing across many worker threads
- **conditional parallelism** - use of an if statement to only parallelise if certain conditions are met (e.g. a sufficiently large data set size)
- **initialisation, reduction** - management of initial values of per-thread private variable copies, and automatic combination of their incremental results

Using simple #pragma directives referring to loops and sections in a C/C++ program which can be parallelised, optimal or near-optimal speed-ups can be realised. The alternative of implementing pthreaded portions is time consuming and usually more error prone. It also creates a larger bulk of code which is then more difficult to maintain, something we wish to avoid in a library intended to be as open to fine tuning

Listing 2.2: An OpenMP enabled code example. A parallel set of threads execute the contents of the #pragma omp parallel code section, each with a private copy of th_id. The library function omp_get_thread_num() acquires the thread number, which is used to report to the user. A barrier clause prevents threads from advancing beyond a point until all have reached the barrier. Only a single thread is permitted to print the total thread count, also acquired from a library function call.

```
1
2
3  #include <omp.h>
4  #include <iostream>
5  int main (int argc, char *argv[])
6  {
7    int th_id, nthreads;
8  #pragma omp parallel private(th_id)
9  {
10   th_id = omp_get_thread_num();
11   std::cout << "Hello World from thread" << th_id << "\n";
12 #pragma omp barrier
13   if ( th_id == 0 )
14   {
15     nthreads = omp_get_num_threads();
16     std::cout << "There are " << nthreads << "threads\n";
17   }
18 }
19
20 return 0;
21 }
```

as possible. If an additional algorithm being added to the library requires not only a serial implementation, but a pthreaded optimised version too, the code base rapidly moves towards unmanageable. OpenMP is easy to control, in addition to these other advantages. Since the clauses are merely compiler directives, disabling the OpenMP compile flag results in the code being treated as serial. The advantage here is that the same codebase can be deployed and compiled on a non-OpenMP capable system without any special cases or extra programming allowances required. Extra scope begins and ends which mark OpenMP sections do not affect the program's structure and are ignored. Environment variables may need to be set in order to instruct OpenMP as to how many threads it should execute. This upper limit may be reached due to larger limits set by the #pragma directives. Though OpenMP is a very powerful library which enables high performance parallelism without major code changes, there are dis-

advantages. Only recent compilers support OpenMP, particularly versions which are not yet available in OS package distribution systems. Fortunately the implementation with the various versions of Visual Studio available is well established and reliable, and as Windows is the primary target platform of this project, OpenMP was considered stable and fast enough to use.

Listing 2.2 shows a code example of an OpenMP enabled program. A parallel section with a private variable is exemplified, in addition to two OpenMP library calls. Thread IDs can be used in order to force parts of the algorithms to only be executed by certain threads. Barriers ensure that a multi part algorithm in which the second portion requires the first to be completely executed in order to proceed can be parallelised safely.

2.3 GPU

Originally, graphics hardware was non-programmable - there was a fairly rigid static pipeline provided for generating 3D graphics, which was not very flexible. However, eventually programmable processors known as shaders were added, which allowed game programmers to add dynamic, more realistic effects to their games. This allowed a relaxation of many games seeming too similar - since effects are now programmed by different teams, in slightly different ways, products have a more unique feel to them. In addition to this, ways were found to exploit these programmable processors to execute algorithms which were not graphical in nature. While originally this relied on expressing algorithms as a set of shaders (a difficult task to complete and debug), soon additions were made to the devices to enable general purpose algorithms to run more easily [14]. Access to caches, and differentiating between constant, shared and register categories of memory explicitly also meant that programmers were then able to achieve throughput closer to the theoretical maximum of the target devices. NVIDIA pushed the "Compute Unified Device Architecture" (CUDA) which they themselves had developed [15]. Later, AMD/ATI along with NVIDIA promoted a more open framework, known as OpenCL, for programming algorithms targeting GPUs [16]. OpenCL implementations are however young, at time of writing the toolchains for devices have been available for many devices for less than six months. As OpenCL is young and has little to offer that CUDA does not, CUDA was selected as a platform for the GPU based

algorithms in the diffusion curve solver.

2.3.1 The CUDA Programming Library

CUDA presents the programmable vertex and pixel processors of the GPU using a single-program multiple-data (SPMD) model. The user passes code to the device in the form of a kernel, which is uploaded and executed. An extremely high number of lightweight threads are spawned by the kernel, and scheduled to run in parallel on the available hardware. This is similar to the SIMD constructs available on modern CPU's capable of vector processing, such as Intel's SSE and the PowerPC's AltiVec technologies. CUDA, however, allows for limited branching within the kernel, permitting different threads to diverge and execute independently. This should be avoided wherever possible, however, as it forces a fallback to serial execution of the divergent portions of an algorithm. CUDA also provides a more complete set of instructions than other vector capable processors - for example, exposing bilinear interpolation implemented in hardware.

CUDA organises its lightweight threads into user defined subsections. These execute in groups (when the scheduler permits) on partitions of the hardware. Algorithms are often bottlenecked by the fact that the latency associated with loading a memory location is high, compared to the kernel's execution time. CUDA effectively hides this by scheduling hundreds (or thousands) of loads, which allows the memory bandwidth bottleneck to be overcome. Threads block based on data availability, and are allowed to execute only when the data they are dependent upon becomes available.

The artificially unified shaders are used to execute a user's kernels in a stream processing fashion. The processors support single precision floating point numbers, and have also had integer support added to cater for GPGPU applications. In current generations of NVIDIA graphics cards, the streaming multiprocessors are clustered together in groups of 8, which are used to execute an instruction on a group of threads. This is known as a *warp*; a set of threads are *warped* from one state to the next. Warps are organised into *thread blocks*, which all run on the same set of streaming processors.

Each thread has access to a register file accessible by just that thread. Threads grouped into a thread block all have access to a piece of shared memory. Threads

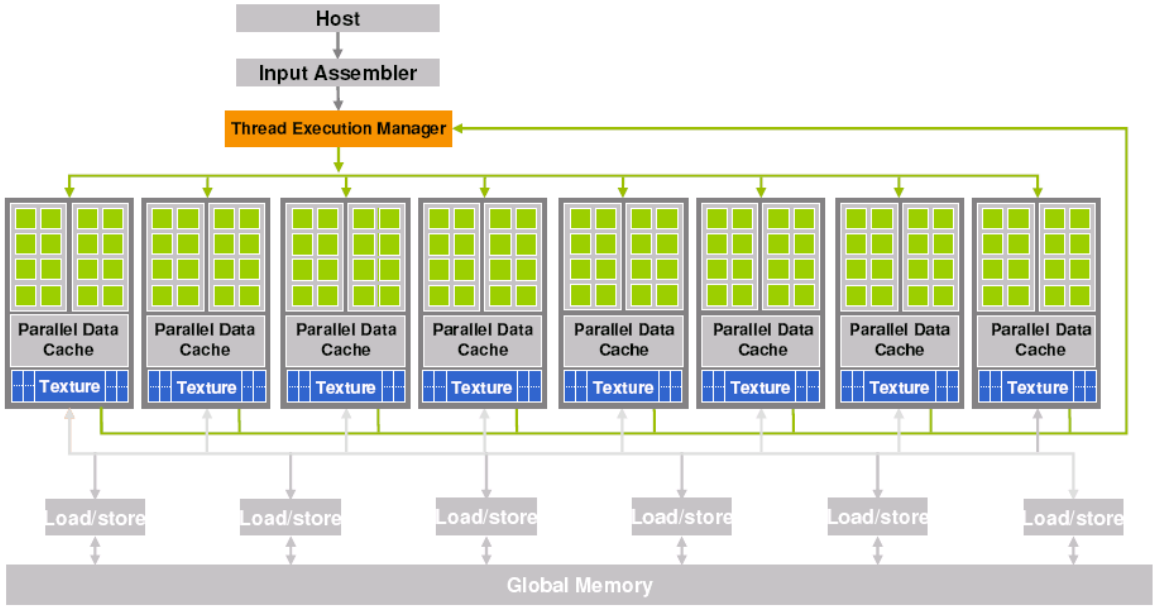

Figure 2.5: An overview of the GeForce 8800 implementation of the CUDA environment. 128 streaming processors are grouped into partitions of 8. Pairs of partitions share a local texture and data cache.

cannot access the shared memory owned by a different thread block. All threads can access global memory and exchange information via it, however it is significantly slower than accessing the local shared memory and register file. There is also a cached constant memory area, which cannot be modified at runtime. Register access is the fastest, however the register file only supports an extremely limited amount of space (32-64KB). Thus register conservation is important to avoid register spill.

Shared memory is the next fastest to the register file, yet is significantly slower and smaller. It is invaluable for any application which requires data sharing between threads in a given block.

Global memory comprises the rest of the memory, and is large and slow by comparison. The exact amount varies from device to device, and even from vendor to vendor. The 8800 family of GPUs typically have 320MB, however newer generations of graphics adapter have seen upwards of 1GB of high speed, GPU only memory. Global memory can be read and written to by any thread, and the latency in accessing it is the reason why GPU stream programming requires such vast amounts of threads to achieve

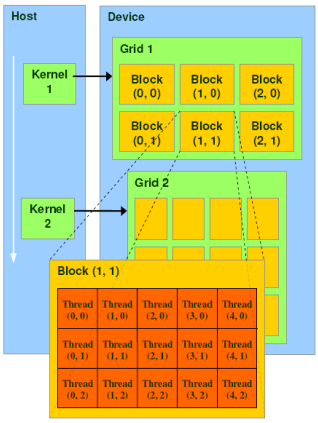

Figure 2.6: An overview of multiple kernel execution on a CUDA device. While multiple kernels can be loaded into memory, only one may be executed at any one time. If functionality of two components is required to be executed concurrently, it is necessary to combine functionality into a single kernel.

optimal throughput. When a thread accesses memory locations, it stalls pending the availability of the locations (i.e. the scheduler only allows the thread to proceed if all locations have been loaded). Thousands of pending operations allows saturation of both memory bus and streaming multiprocessor throughput [17].

Multiple kernels may be loaded into the memory of a single device at the same time, however only one can be executed at any given time. Sharing data between kernel runs is costly, hence functionality is, if possible, combined into a single kernel.

Accelerating an application using CUDA depends on identifying algorithmic bottlenecks and implementing the program model such as to reduce their effect. This involves careful structuring and ordering of user defined kernels to ensure that all the streaming processors are executing at peak performance. Strict management of data access patterns (to fit into the limits imposed by the memory hierarchy) are essential. For memory throughput intensive programs such as image registration, this latter issue of memory management is key to obtaining good speed-up.

2.4 CELL Broadband Engine

Sony's Cell Broadband Engine is another computer processor designed with the stream processing paradigm. A master processor, known as the Power Processor Element

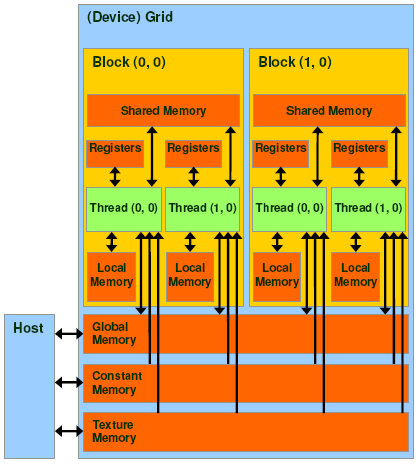

Figure 2.7: The CUDA memory hierarchy. Threads in a thread block each have their own private registers/local memory, and can share data via the shared local block memory. Global memory is uncached, but writable by all threads anywhere in the thread grid. Constant and texture memory are cached, yet not writable by the device during kernel execution, thus less useful for data throughput bottlenecked algorithms.

(PPE), uses a number of Synergistic Processor Elements (SPEs) to achieve computational throughput possible of rivaling a small cluster [18]. The PPE is capable of dual threaded execution (Simultaneous Multithreading), while both the PPE and the SPEs are capable of data level parallelism. The SPEs are SIMD only (any SISD code will be converted into SIMD by the compiler) and are fully managed by their PPE host threads.

IBM chose to match the SPE clock frequency to a high figure along with that of the PPE, reducing complex intercommunication problems. In order to achieve this while minimising floorspace on the die (thus minimising cost and maximising silicon yield), architectural complexity was reduced. Register renaming and highly efficient branch prediction were thus deprioritised. Branch misprediction thus causes a lengthly pipeline stall and associated performance penalty.

While a port of the GPU based diffusion curve solving algorithms to the CELL would have been quite direct, due to time constraints it could not be considered within the scope of this dissertation. It is also unlikely that the performance would have been able to meet or exceed that provided by a modern GPU - the CELL has been on the

market for five years without any performance enhancement or upgrades, and plans to build a better, 32 SPE version of the processor have been shelved by IBM.

2.5 Branch Prediction

A branch in a computer program is a conditional statement which allows for flow control in instruction sequences. A branch will, conditionally or unconditionally, indicate which is the next instruction that shall be executed. Conditional branches are of interest in the area of high performance code optimisation, as they can greatly affect the runtime of a set of instructions. if and while statements, and their derivatives are the high level language constructs which resolve down to branching. Conditional branching can cause delays in a processor's execution pipeline. When a CPU has a multi stage pipeline, many instructions can be at various stages of execution simultaneously. However, when we have a conditional branch in a program (i.e. an instruction which redirects the execution sequence based on a condition), the processor does not know which instruction will be executed next, causing a pipeline stall.

Modern processors perform speculative execution - the processor assumes that a branch will, or will not be taken, and begins to execute subsequent instructions [19]. If the processor's assumption regarding the branch was correct, a costly pipeline stall is avoided. If the processor was incorrect, the partially executed instructions are flushed from the pipeline, causing a stall.

Avoiding branches by writing algorithms in such a way that branch predictors are either encouraged to be accurate, or that some branches can be eliminated, can provide a noticeable speed-up. This is explored in the implementation of the diffusion curve solver. Notably, on GPU hardware, where we are dealing with less feature packed processors, branch predictors are not present. This means that branching should be avoided even more so than on CPU based code. In addition to this, stream processors suffer additional heavy penalties where flow control of grouped threads diverges. A branch statement which causes different threads to take different paths can require a full serialisation of each thread, although this has been improved upon since the first generation of GPGPU capable devices.

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) \varphi(x, y) = f(x, y)$$

Figure 2.8: Poisson's Equation - a partial differential equation which takes this form in a two dimensional Cartesian system

2.6 Diffusion Curves

As mentioned in the introduction, diffusion curves are a vector based primitive used for describing smoothly shaded images or subsections of images [1]. A diffusion image through a plane partitions it into two half-spaces, defining different (or sometimes the same) colour(s) on either side. The colour may vary along the curve - but more importantly, a blended transition between the curve and any other curves can be generated. If, along a row of pixels in one direction towards the edge of a rasterised diffusion image, there are no closer curves or curve subsections with a different colour intensity, we might expect that row to all take the value of the curve.

Given a set of diffusion curves, the final image is constructed by solving a Poisson equation whose constraints are specified by the set of gradients across all diffusion curves [1]. We also have the added advantage of being able to apply operations to diffusion curves usually associated with other vector based primitives - for example, keyframing, if the objective was to build an application which stored animation as a changing set of diffusion curve based images. Also, storing diffusion curves as pure vectors means that the resolution of the final image is not bounded - the curve set can be rasterised at many wildly different resolutions and then solved. Although this is not of immediate importance with regard to the library presented, it is worthwhile to remember. Our library merely solves an already rasterised set of pixel constraints - other functionality would be the responsibility of the host application.

If the constraints presented by being limited to expressing an image or animation in the gradient domain are acceptable, there are also other benefits. Storing an image or animation in vector format is extremely efficient - size is dependent more on the complexity of the image than its resolution/quality. Though most imaging systems currently favour raster based techniques, vector graphics still remain popular - Flash animations, and the SVG format (Scalable Vector Graphic) are two examples.

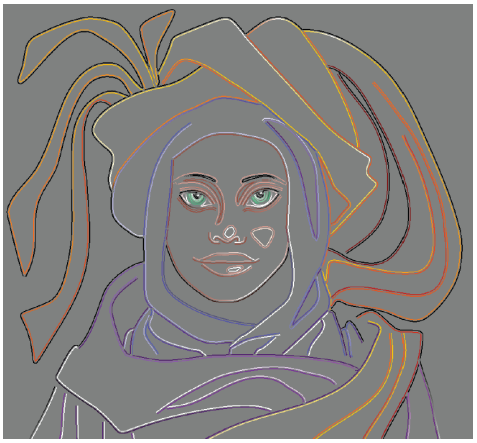

Figure 2.9: An example set of diffusion curves, sketched freehand.

Artists can create these images by simply sketching in freehand, or alternatively, by tracing lines over an existing image. An example of the former technique's curve set, and the resulting output, can be viewed in Figures 2.9 and 2.10 respectively. Libraries also exist which can extract features out of existing images, effectively automating the conversion of a standard image to a diffusion curve based image.

Despite their advantages, most software which supports creation and editing of vector graphics has limited or no support for adding colour gradients, which are desired by many artists. Realistic shadows, pleasing shading, and even the famous airbrush technique are based on colour diffusion - and can be represented by a set of diffusion curves. Tool support can be greatly improved - existing software can take extremely long time periods to allow the perfection of a diffusion vector based image by a human artist.

Orzan et al, in their research with artists, found that using diffusion based tools has a notable benefit [1]: an artist can sketch an image using black on white lines - colour can be added later. The colour can be easily changed at any point, without requiring extra human effort, and the colours at the side of each line can specified to be tightly knit, or have a large gap between them. Again, this can be tweaked for curves, or sets of curves, at diffuse time - the user can easily experiment with their image in ways that conventional raster based software simply cannot facilitate.

Figure 2.10: The final image after diffusion has been applied to the set of curves given in Fig. 2.9

In addition to this, the majority of significant colour variations in an image tend to be caused by hard edges [20]. Marr and Hildreth note that complex shading effects can be reconstructed using a number of edges, and that an entire image can be encoded with trivial loss using a set of edges [21]. Using established edge detection algorithms, existing images can be converted into vector based diffusion images, ready to be highly compressed in a format immune to further loss. They are highly desirable if future scaling may be required. This has been investigated by Orzan et al [1], though we focus on previously rasterised representations of diffusion curves.

Chapter 3

Previous Work

In this section we examine several existing relevant implementations of solvers which take advantage of the additional computational capacity provided by GPU devices.

Jeschke et. al. presented a new Laplacian solver for minimal surfaces [4] - i.e. surfaces which have a mean curvature of zero in all regions, excepting some fixed boundary conditions. Firstly, they provide a robust rasterization technique to transform continuous boundary values (diffusion curves) to a discrete domain. Secondly, and more relevant to this study, they propose a variable stencil size diffusion solver that solves the minimal surface problem. They detail a proof that their system will converge to a mathematically correct solution for a given set of input data, and demonstrate that it is at least as fast as commonly proposed multigrid solvers, but much simpler to implement. It also works for arbitrary image resolutions, as well as 8 bit data. Examples of robust diffusion curve rendering are provided, demonstrating where their our curve rasterisation and diffusion solver implementation eliminate the strobing artifacts present in previous methods, such as those of Orzan et al [1].

Given a set of boundary points, the associated minimal surface can be found by solving an equation which minimises the Laplacian of the solution (i.e. modifies the surface to have the required mean curvature of zero globally), while maintaining the defined boundary points. The diffusion solver is capable of converging simple sets of boundary volumes in an image in as few as eight iterations.

McCann & Pollard also completed some research, concentrating more on the area

of gradient domain painting (i.e. tools associated with manipulating images based on diffusion curves). Gradient domain painting allows artists to paint in the "gradient domain" - a line or curve can be drawn, and a gradient on either side can be resolved seamlessly into the rest of the image. This results in a novel style of output, and is very rapid to use. However, resolution of these complex systems is very computationally expensive, and hence a GPU based solver is used to allow for faster processing. Many iterations can be performed in real time, allowing an artist to see the brush strokes they make resolve and integrate into the rest of the image over many iterations. On a slower, CPU based system, the time interval would be so great that it would not appear as an animation. However, with the massive parallel power provided by a GPU, this can be processed in real-time on multi megapixel canvasses.

McCann & Pollard introduce a powerful, gradient painting brush and gradient clone tool, as well as an edge brush designed for edge selection and replay [3]. Their implementation on GPU enables an artist to manipulate the surface in a gradient-oriented fashion in real-time. These brushes, coupled with special blending modes, allow users to accomplish global lighting and contrast adjustments using only local image manipulations e.g. strengthening a given edge or removing a shadow boundary.

Chapter 4

Implementation

An overview of the algorithms used, in addition to the techniques used to accelerate the library, is given in this chapter. Details of how to link against the library inside Windows, along with a description of the example program provided with the codebase are also below.

We provide two versions of the library: one which implements Dirichlet boundary conditions, and another which implements Neumann boundary conditions. The difference between these two methods can be easily put: Dirichlet places constraints upon pixels, and Neumann places constraints upon boundaries between pixels.

As can be seen in Figure 4.1, certain pixels are given a value and constrained, so that they can only contribute to surrounding pixels, and will not vary across iterations. Discontinuities are introduced by rasterising two parallel (not necessarily straight) lines of constraints. There may or may not be a space between - this can be implemented by the user's software. A small gap is often used as it adds a pleasing blend between the two lines, while maintaining the discontinuity's visual effect. A Dirichlet based system is straightforward to process.

The alternative of Neumann is subtlely different. Since boundary conditions are implemented upon pixel edges rather than pixels themselves, we need a 4 bit mask for each pixel on the image, to mark the top, bottom, left, and right as constrained (or not). A fully constrained pixel (in the Dirichlet style) will have all four boundaries

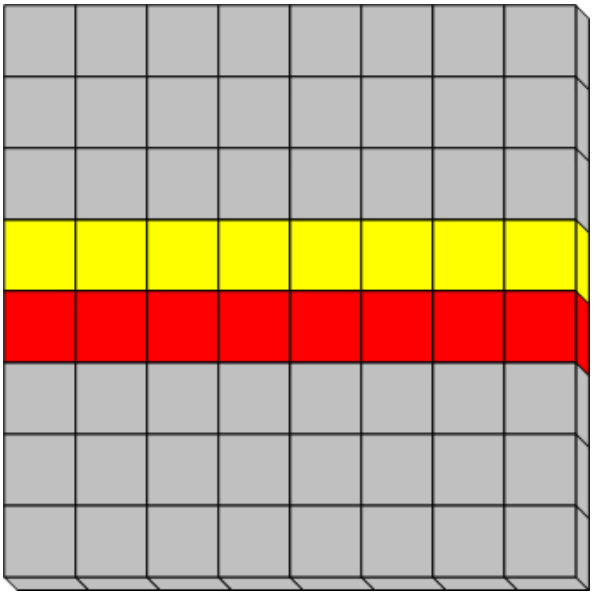

Figure 4.1: Dirichlet boundary conditions. Grey indicates unconstrained pixels.

constrained; i.e. an iteration cannot accept a contribution from any surrounding pixel. When processing a pixel at (x,y) we only consider the boundaries on that pixel - if a boundary is not set, we do not need to check the corresponding side of an adjoining pixel to see if it is bound. Boundaries are treated as one way. To introduce a discontinuity, a parallel row of pixels have the boundaries facing each other set. This allows more flexibility than the Dirchlet system - we can introduce a discontinuity without forcing either side to a specific colour value.

Figure 4.2 shows an example of Neumann constraints. A discontinuity near the base of the image is introduced, without a constrained colour near it. We would expect a resulting converged image to be fully red - however, the algorithm would need to "flow" the source colour around through the small unconstrained gap on the bottom right of the system, and along through to the bottom left. The notable disadvantage of Neumann boundary conditions is that there are a lot more conditions to check for every pixel, in every iteration. The overall algorithm, using Neumann boundary conditions, is noticeably slower than when using Dirichlet boundary conditions. The advantage offered is slim, and the added cost arguably outweighs this. As convergence speed is regarded as one of the more important portions of the research into this area, the

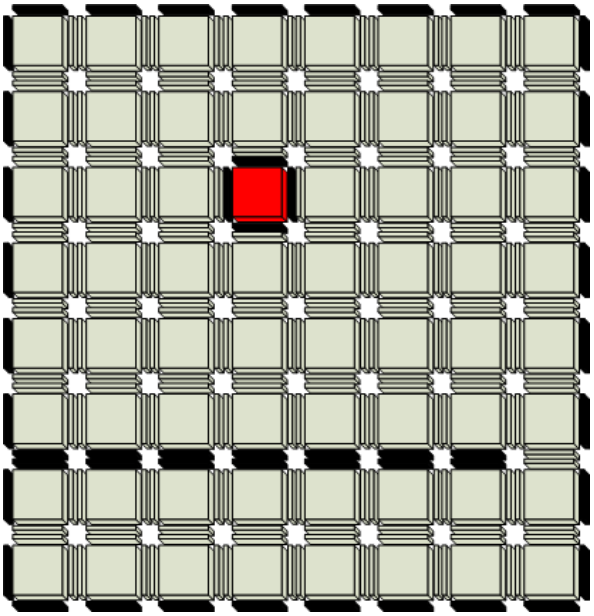

Figure 4.2: Neumann boundary condition example.

Dirichlet model is treated as the primary choice for implementation. The Neumann version of what is otherwise an identical codebase remains useful for the purposes of a performance comparison. During implementation, the changes made to switch from Dirichlet to Neumann conditions were extremely low down, in portions of the code which have been very precisely fine tuned by hand. For this reason, it was more sensible to extract a disjoint version of the library to cater for Neumann conditions; attempting to maintain a unified library added complexity, and reduced performance. Neumann boundaries proved to perform particularly poorly on the GPU, the reasoning behind this is explained in the implementation section.

4.1 Algorithms

There are three main techniques used to solve a set of diffusion curves. We assume that a target resolution has already been selected, and that the curves have been rasterised to a template image accordingly. Firstly we examine the most simple naive case, and then apply the other optimisation techniques.

4.1.1 Naive Algorithm

We attempt to solve for the steady state of the system, i.e. a heat equation at a time where the system has stabilised, and energy is no longer moving through it. If we are at iteration n in a diffuse operation, the value for a pixel at position x,y is given by the following equation:

$$4*(V[n+1]_{x,y}) + V[n]_{x-1,y} + V[n]_{x+1,y} + V[n]_{x,y-1} + V[n]_{x,y+1} = 0$$

At the boundary of an image, however, there is an issue - the out of bounds value cannot be used. We can solve this by omitting the contribution of that pixel and scaling the equation accordingly.

Effectively, we are using a first order integrator by forward Euler method. This means that the error introduced will be $O(dt^2)$. While a good second order integrator could converge the image with $O(dt^4)$ for only twice the work, we are only interested in the final result - thus the forward Euler method is acceptable.

The values of each colour channel associated with a given pixel must be processed individually. In addition to this, there is an unfortunate caveat: values must be calculated using floating point arithmetic. The loss of precision that results from using integer arithmetic for solving these equations introduces significant error, which causes the final converged result to be incorrect. Extra overhead is introduced by the requirement to split and merge the colour channels - however this is unavoidable for a high quality solution. The diffusion library uses an abstract class for representation of images: the *DiffusionImage* class. Concrete classes named *DiffusionImageInt* and *DiffusionImageFloat* are provided. The DiffusionImageInt class can be used to calculate these low quality results for the Naive diffuser algorithm, but has been superseded by the other, more highly optimised methods.

A basic naive function for a diffusion iteration for a single pixel is shown in Listing 4.1. Input images are stored by the library as an array of 32 bit integers. This is a useful method of storing them for several reasons: firstly, most hardware systems perform 32 bit data transfers. If we were to store a rasterised image as an array of single

characters, we could not guarantee that the compiler would transfer in this optimised fashion. The majority of compilers will be able to detect these circumstances with the correct combination of counters declared constant - however, explicitly forcing the processor to behave in this way is highly desirable in situations like this, where we want to guarantee a potential speed-up is being compiled.

A diffusion image may consist of simple monochrome pixels, or potentially 24/32 bit colour. 24 bit colour is most common - images with 32 bits (4 bytes) used to store each pixel generally use the fourth channel for storing transparency values. Generating a gradient of transparency is not commonly used by artists, thus we do not use this extra byte for each pixel. If transparency support is required at a future stage, it is a trivial addition - however it adds a 20% calculation overhead to each iteration.

The remaining byte is left unused. As such, the data is converted from a packed form, which would be the usual method of inputting it, to an unpacked, aligned version. This allows for easy 32 bit data transfers without the need for masking and shifting instructions. It makes the reintroduction of alpha channel processing a trivial matter, and there are also implications for cache performance and cache coherency protocols. In modern computer processors, particularly GPUs (but there is still an impact on CPUs, e.g. when using vector extensions) "aligned" memory transfers are preferable. Optimisations are built into the hardware which allows faster data transfer operations to be performed where the memory addresses in question are aligned to even, 32 bit boundaries. Specialist processors like the CELL Broadband Engine, furthermore, require 128 bit aligned boundaries. Where unaligned transfers are possible, a performance penalty may be introduced. Certain compilers will purposely place data at appropriately aligned boundaries (e.g. Visual Studio 2005 and later), but others demand compiler directives to be wrapped around declarations, and even then these may be ignored (certain versions of gcc/g++). This avoids the need for compiler specific directives which may be disobeyed, or disrupt other compilers.

There is a further implication which comes into play when using packed data on multicore systems. As described in the background session, multi-core processors need to use a cache coherency protocol, in order to prevent mismatched/out of date memory

locations from being cached (i.e. if processor 0 updates a memory location, and this is not flushed out to main memory before processor 1 reads it, processor 1 could read an out of date version from its own private cache). Cache entries are stored as lines of aligned memory addresses, often 16 or 32 bytes (4 or 8 single precision data). A modification of a location inside the cache will switch it away from being shared (other processors will see this via snooping on the bus). However, the entire cache line will be declared invalid, requiring it to be flushed back out to main memory if another processor core wishes to modify it (or at least into a slower, shared high level cache).

Packed data which does not consist of 4 byte groupings will, therefore, cause some pixels to straddle multiple cache lines. This will not have a very noticeable effect on a single threaded CPU implementation of the diffusion algorithm, however when the algorithm is load balanced across multiple CPUs, it will cause an increase in cache misses, causing a decrease in performance which outweighs the benefit of being able to keep more pixels inside the level 1 cache. Unpacked data has a larger advantage when applied to the GPU versions of the algorithms. There was also an interest in keeping both CPU and GPU versions of the algorithms closely knit. Different components of the CPU and GPU algorithms could be used together during development for testing, verification and debugging purposes. Using this pattern of development for this hybrid CPU/GPU based library proved invaluable for progression through the project's lifecycle.

Listing 4.1: The basic implementation of a single pixel diffusion iteration.

```
1
2  void DiffusionImageFloat::DiffuseFunc(int x, int y)
3  {
4    // if the pixel is a constrained pixel, simply copy it and return
5    if(constrained[y*IMAGE_WIDTH + x] == true)
6    {
7      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 1] = imageFloat[(y*IMAGE_WIDTH
           + x)*4 + 1];
8      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 2] = imageFloat[(y*IMAGE_WIDTH
           + x)*4 + 2];
9      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 3] = imageFloat[(y*IMAGE_WIDTH
           + x)*4 + 3];
10     return;
```

```
11    }
12
13    int pixcount = 0;
14    imageFloat2[(y*IMAGE_WIDTH + x)*4 + 1] = 0;
15    imageFloat2[(y*IMAGE_WIDTH + x)*4 + 2] = 0;
16    imageFloat2[(y*IMAGE_WIDTH + x)*4 + 3] = 0;
17
18    //left
19    if(x > 0)
20    {
21      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 1] += imageFloat[(y*
          IMAGE_WIDTH + x - 1)*4 + 1];
22      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 2] += imageFloat[(y*
          IMAGE_WIDTH + x - 1)*4 + 2];
23      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 3] += imageFloat[(y*
          IMAGE_WIDTH + x - 1)*4 + 3];
24      pixcount++;
25    }
26
27    //right
28    if(x < (int)IMAGE_WIDTH - 1)
29    {
30      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 1] += imageFloat[(y*
          IMAGE_WIDTH + x + 1)*4 + 1];
31      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 2] += imageFloat[(y*
          IMAGE_WIDTH + x + 1)*4 + 2];
32      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 3] += imageFloat[(y*
          IMAGE_WIDTH + x + 1)*4 + 3];
33      pixcount++;
34    }
35
36    //up
37    if(y > 0)
38    {
39      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 1] += imageFloat[((y-1)*
          IMAGE_WIDTH + x)*4 + 1];
40      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 2] += imageFloat[((y-1)*
          IMAGE_WIDTH + x)*4 + 2];
41      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 3] += imageFloat[((y-1)*
```

```
        IMAGE_WIDTH + x)*4 + 3];
42      pixcount++;
43    }
44
45    //down
46    if(y < (int)IMAGE_HEIGHT - 1)
47    {
48      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 1] += imageFloat[((y+1)*
            IMAGE_WIDTH + x)*4 + 1];
49      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 2] += imageFloat[((y+1)*
            IMAGE_WIDTH + x)*4 + 2];
50      imageFloat2[(y*IMAGE_WIDTH + x)*4 + 3] += imageFloat[((y+1)*
            IMAGE_WIDTH + x)*4 + 3];
51      pixcount++;
52    }
53
54
55    // calculate the averaged values and store.
56    imageFloat2[(y*IMAGE_WIDTH + x)*4 + 1] /= pixcount;
57    imageFloat2[(y*IMAGE_WIDTH + x)*4 + 2] /= pixcount;
58    imageFloat2[(y*IMAGE_WIDTH + x)*4 + 3] /= pixcount;
59
60  }
```

With the conversion to using per-channel single precision floating point numbers, another layer is added to the diffuser's process. Any application for manipulating the curves will still provide integer data - so, for every diffuse, the data set will need to be converted to floating point values, and then back to integers. This overhead comes into play when comparing performance with other implementations, and is discussed in the Evaluation section.

Listing 4.1 contains a basic naive implementation for a per-pixel diffusion using Dirichlet boundary conditions. Dirichlet boundary conditions place per-pixel constraints - if a discontinuity is required, we rasterise two sets of parallel constrained pixels appropriately. Surrounding pixels are averaged and written to a buffer, which is then swapped in when the entire iteration has been calculated. We use a disjoint buffer for two reasons. Firstly, if a pixel at (x,y) was calculated and written out, then

the input at pixel (x+1, y) would have a surrounding value from the current iteration as an input, rather than the last iteration. This would gradually introduce a sweeping and increasing error, from the top left of the image down to the bottom right (as this is the order in which all of these algorithms proceed through the diffusion curve based rasters).

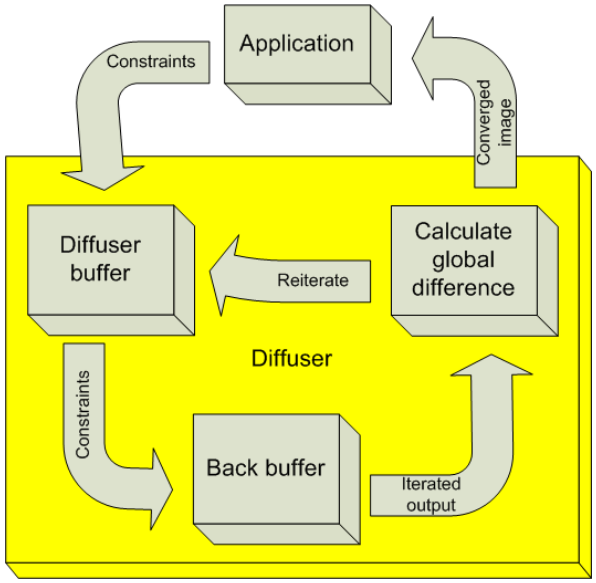

Figure 4.3: A flowchart detailing how the naive diffuser processes images on CPU

Secondly, unlike many implementations, the library presented features an adaptive feedback approach to converging the images. Rather than simply applying the algorithm for N iterations, where N is a number which appears to give reasonably high quality results across a variety of images with differing complexity, a difference calculation is performed. Since the iteration method used results in us having iteration N in the image buffer, and iteration N-1 in the back buffer, we can perform a per-pixel, per channel comparison, and reduce it to calculate a global difference estimation.

Using this reduction we can:

1. greatly increase the speed of diffusing images with a simple, easily processed set of constraints.

2. reprocess complex images rapidly as incremental constraints are added/removed/edited

by the artist.

3. vary the quality/speed trade off of the image by convergence threshold variation.

Parallelising a reduction is reasonably straightforward on fat core CPU-like processors, but provides something of a challenge on a stream processor such as a GPU.

4.1.2 Hierarchical pyramid

While the naive implementation is correct and will eventually come to a suitably converged solution, it has a large disadvantage. Pixels which are very far away from a constraint (yet are destined to converge to a 100% weighted value of said constraint) will take many, many iterations to converge. This means that an image using the method may take as much as minutes to converge. With relation to our method of calculating convergence, there is an additional issue: using the naive algorithm, rate of convergence tends to fluctuate. Since the library waits for the rate of change of the global difference to fall beneath a specific value, this makes it very difficult to select an ideal constant for the threshold.

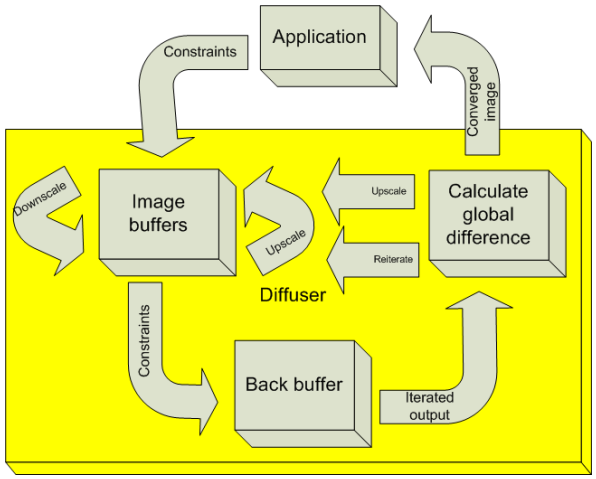

Figure 4.4: The diffusion library with the pyramid optimisation applied

We can greatly accelerate the algorithm by applying a classic hierarchical pyramid technique to the iteration process. Pyramid approaches to image processing algorithms are a classic method of acceleration, and have been used for many radically different

varieties of problem [22]. To apply the pyramid algorithm, we downscale the rasterised source set of constraints, each time reducing the resolution by a factor of 4. At the lowest resolution, we apply the naive diffusion algorithm, until we detect convergence. Once this occurs, the pixel values are upscaled back to the higher resolution version. This process continues until we are back at the full resolution.

Figure 4.4 shows the modifications to the system in order to accommodate the hierarchical pyramid of images. When a diffusion operation is invoked on an image, a stack is used to store the various levels of the pyramid. These downscaled versions are generated immediately - the constraint markers are left unchanged at each level during processing. Pixel values are overwritten each time a child level of the pyramid converges, however storing them is sensible as they are needed regardless to generate their own child levels - generating a complete image set, each from the base image would be equally or more expensive.

At the lowest resolution, our generated representation of the image will converge, and be upscaled back up to its parent once the global difference calculator's results indicate that it is time to do so. The sample application supports redrawing the various levels of the pyramid if required, to demonstrate the process. This process continues until the top level image reaches a final solution, at which point the data is passed back out to the host application for display and further manipulation by the user.

The major advantage of this is that a constraint is permitted to affect distant pixels a lot more quickly. In line with other image processing algorithms which use hierarchical pyramid techniques, this fashion of speed increase does not result in a different, incorrect/lower quality solution. Even if a low resolution image's pixel converges to a value which is majorly polluted when compared with what the final, full resolution components should be, this is not of great concern. When upscaled, the value will become closer and closer to the correct final value. In addition to this, as a polluted value arising from surrounding pixels will be an average of the correct values for its "sub pixels" once upscaled.

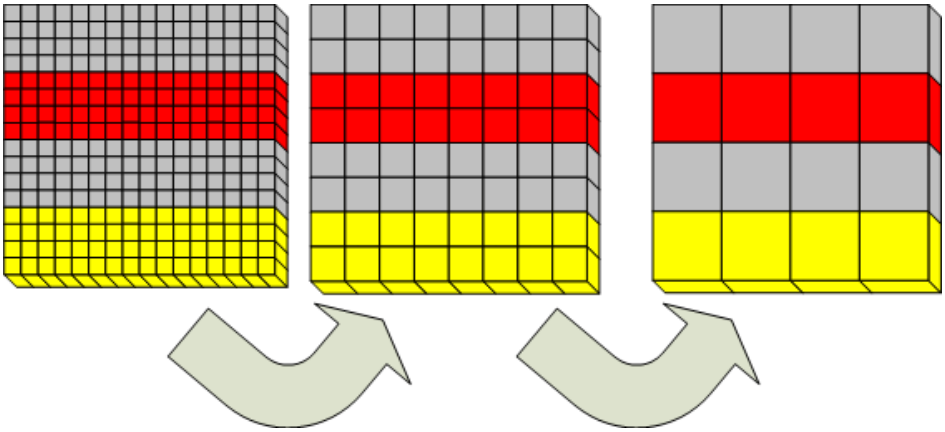

Figure 4.5: The basic downscaling concept is depicted.

Cautious and Greedy scaling approaches

Two approaches were experimented with for the downscaler. These will be referred to as the *cautious* and *greedy* approaches within this report.

These two techniques are for dealing with downscaled constraints. Downscaling by a resolution factor of 4 means we can simply use nearest neighbour for snapping the new pixel values to - useful, since we want the algorithm to be as computationally cheap as possible, and bilinear/trilinear/antrisophic filtering forces a trade off between slowdown and error introduction. However, constraints can cause a problem - how do we handle cases where a pair of constraints map to the same pixel downscaled? For regular pixels, simply averaging them will suffice - as already referenced, an average, when upscaled, will diffuse back to have the correct colour channel intensities, and more rapidly than if the pixels were omitted from the operation. Constraints are less straightforward to handle.

One option is to simply ignore constraints which cause this problem. The diffusion library will run very slowly if this approach is used, and thus it was eliminated before reaching the benchmarking section. As the image gets repeatedly downscaled, constraints become increasingly contented, leading to more and more of them being ignored. The constraints which are left behind will then give too much of a contribution to the pixels which do manage to have their value converged - when later upscaling to a higher resolution, they need to be fully reiterated, rendering the pyramid technique

worthless.

Another way of handling overlapping constraints is to average the contributing constraints, and/or unconstrained pixels. Unconstrained pixels would be considered from the second lowest resolution, as the lowest resolution would fill some upscaled proposed values to reiterate. Experiments showed that across the range of test images, the resulting values (for passing into the upscaler) were globally closer to the converged values of the next layer on the pyramid in every case.

Additional improvements were shown by using a "greedy" approach to downscaling. If any of the four parent pixels were constrained, the child pixel will take on the value of the parent alone, with no averaging. If two parent pixels of differing colour intensity are contributing, then an average is calculated. For the same reason as described above, averaging of two constraints provides a compromise - the child pixel's contribution to unconstrained pixels brings them closer to the final, top level converged solution. The implications and results of choosing a greedy approach are discussed in the evaluation section.

In cases where no parent pixels are constrained, then a simple blind average can be taken. This case is separated out as once it is identified, the calculation itself is branchless, allowing a speed-up, particularly on the GPU port of the algorithm.

The constrain status of a downsampled pixel is set to true if any parent pixel is constrained, in all of the techniques described above.

This project concentrated on the Dirichlet version of the library, however as mentioned, the Neumann boundary condition version is also provided. Downscaling constraints in a Neumann system is a far more complex operation than in a Dirichlet system. As mentioned above, the optimally performing solution for constraint downscaling was to simply constrain the child pixel if any of the 4 parent pixels was a constraint. Since Neumann constraints tend to form channels along which colour values are forced to flow, we need to treat the consolidation of these 16 parent constraints very carefully in order to generate sensible child constraints. The optimal method must be internally greedy, and externally cautious. The relevant constraints can best be visualised as a two thick set of corresponding side constraints, shaped as a + symbol,

with a one-thick enclosing border. As with Dirichlet downscaling, we take an average value from the colour values as appropriate, or override using constraints contained. Constraints in the + are all ignored - the grouping of four pixels is internally greedy. On each external side of the square of 4, we have two potential constraints. If either of the two on a given side of the parent square is set, then the corresponding child side must be set. Unfortunately this blocks some circumstances where a diffuse out this side would converge the system towards a final solution more rapidly, but other configurations would result in an unwanted colour leak which would diverge the current level of the pyramid's solution away from the required high level steady state.

Upscaling

The upscaling portion of the pyramid algorithm is more straightforward. The upscaler operates upon the source image, and the higher resolution parent image. It is important to note that an upscaled image is not generated directly from the low resolution source - while the nearest neighbour colour data could be generated without error, there would be a loss of resolution with regard to the constraint indicators. Upscaling from child image back up to parent image is implemented identically in Dirichlet and Neumann boundary systems, as the constraint values are effectively disregarded during the upscaling process.

4.1.3 Variable size stencil

Jeschke et al propose a *variable size stencil* to accelerate the convergence process [4]. The main reason for the slow convergence of the system is that a constrained colour cannot move quickly from one area of the image to another. Using the naive method, an image N pixels wide will take at least N-1 iterations for a constraint on one side to affect pixels at the far side. The hierarchical pyramid overcomes this, however the issues described above involving constraint downscaling make it inefficient in many cases.

Jeschke et al suggest a more efficient approach that uses a a pre-generated radius map, which operates in a similar fashion to the existing naive integrator. The difference can be seen in Figure 4.6. While the naive diffuser will only consider immediately

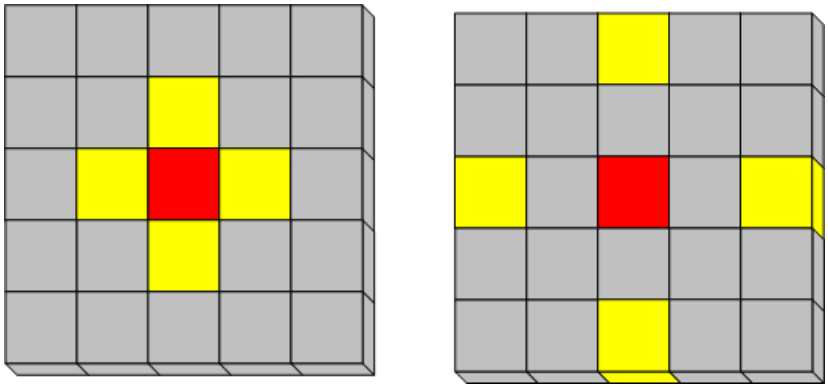

Figure 4.6: The original iteration (left) and the variable stencil iteration (right)

surrounding candidates, the variable stencil method allows us for sampling of further away pixels. Before the diffusion mapping begins, a distance map is used to calculate the how far from each pixel the samples used can be taken.

Generating this map requires a per pixel seek operation - the distance is different for every pixel, and not strictly related to the position of the pixel in the image space. A pixel can only look out as far in any direction as the distance to the closest constraint or edge to it. The distance traveled in each direction must be identical for Jeschke's proof of correctness to remain valid [4].

Another feature of this variation that Jeschke et al require is that the distance each pixel seeks must be reduced via one of a number of strategies, each iteration. This library has been unable to reproduce their results despite implementing their method as specified, even with any the given shrinking approaches. While the speed increase is substantial, our resulting images yield artifacts from an unknown cause. Thus we do not attempt to compare this with our other, fully functional algorithms.

Previous papers have not, however, noted that the above techniques could easily be integrated, and would potentially result in another speed increase. Jeschke et al report that their solver will converge to a pleasing solution in as few as 8 iterations. If we were to apply the variable stencil to the hierarchical pyramid algorithm (i.e. on each level of the pyramid) this could lead to some interesting potential speed increases - especially with large multi-megapixel images containing a lot of tightly packed constraints.

4.2 Example program

A non-functional requirement of the diffusion library was a complete separation and explicit boundary between the image processor itself and the frontend program. Thus a separate but lightweight example program was required to allow easy testing of the library with regard to quality of feedback, correct operation, and usability. For display, the well-known Simple DirectMedia Layer (SDL) was used. It is readily available in a 32 bit library, and it was also possible to compile a 64 bit library for use with this project. The SDL is also open source and cross platform, which further facilitates support of multiple architectures and operating systems.

The SDL allows us to draw a simple window on our screen, and write pixels into a screen buffer for display. Pixel values are progressively added (this can also be parallelised if required) and the screen is "flipped" to switch from the last displayed frame on screen to the next. As the name suggests, it is very easy to work with. SDL also provides input listeners, allowing for keyboard and mouse controls.

The L key is bound to (re)loading the image file in use from disk. Esc can be used to terminate the program elegantly. SDL's blocking threads are also used to enable waiting on the user. Otherwise, the program would have to a) be forced into spawning and terminating a worker thread, which in itself may contain sub threads, or b) use constant polling, which generally uses 100% of a CPU core. Both are sub optimum solutions. Without needing to write a more complex threading interface at the top level, the example program simply blocks waiting on user input. When the image is edited (i.e. some more constraints are added freehand by the artist) the diffuser is activated. It will spawn/unblock its own CPU threads, or initiate a GPU based algorithm, converge, then return to the program. The library itself is designed asynchronously; the frontend handles the diffuse as a roughly evenly timed set of equal work chunks. Again, the frontend can use this to poll for further user input, and append additional constraints to an already-in-progress diffusion operation. Designing the library in this way was intended to increase quality of feedback to the artist - rather than making a change to the image, and the entire application locking itself from further input until it is converged, additional curves can be added without the need to wait.

A set of worker functions exist in the library in order to handle mouse movement, mouse click, and keyboard operations. Application behaviour is defined within these functions. The application will wait for input (thus blocking the diffuser) until the user makes an action. If an action is already underway, the SDL input devices are polled for further input. This can be done cheaply while cooperating with the asynchronous nature of the diffusion library's operations. By combining polling and waiting techniques in this fashion, a lightweight application resulted which can be used to obtain pure benchmarks from the library without results being offset by frontend based overheads.

Loading pre-existing images is supported in addition to freehand sketching on a blank canvas. Several image formats can be imported, including JPEG, PNG, BMP, GIF and TIFF. The DevIL open source image support library is used to perform these tasks. Despite a non-ideal state machine style non Object Oriented interface, it provides reliable functionality for importing all of the mentioned image formats across many platforms. It is also one of the few free use libraries which provides compilable code for 32 and 64 bit operating systems - given the goal of allowing the whole project to run on both of these options, DevIL was the obvious choice.

4.3 Linking the Library

The libraries provided (Dirichlet and Neumann) can both be used in the same fashion by external, more fully featured programs than the example provided. This can be achieved in one of two ways: 1) the library code can simply be inserted into a project as-is, and 2) the diffuser can be linked to as a normal class within the project. However, this is not recommended; the code base provides a precompiled Dynamic Link Library (DLL) for each version, which can be used at runtime by external programs. This is preferable - as it is the purpose of a DLL to be shared on a system and used by potentially more than one program. In addition to this, compiled binaries of dependent programs are smaller, thus easier and cheaper to distribute, than if the codebase were to be directly integrated (this is equivalent to using a static library).

Newer versions of the library can also be distributed independently - which means

any tools developed for it would not need to be upgraded providing the interface remains the same (and there is no reason why it would not). A programmer can use the set of header files available, as well as the compile-time library which indicates information about the compiled code inside the DLL. The DLL can either be distributed with a final application, or installed separately on end users' computers. Since the different diffusion techniques all inherit from a common class and provide an identically behaving set of functions (as far as the application is concerned), a single abstract diffuser pointer can be inserted. Then, depending on the hardware available, the application can instantiate one of the high performance diffusion classes. Checking for GPU support can be done with the CUDA library, and if a suitably versioned compute able device is present in the system (computer version 1.0 is acceptable for the GPU diffusers) then an attempt at executing the GPU code can be made. As many users have suitable hardware, but do not install the necessary CUDA drivers, this may fail. If the CUDA device is not set up correctly, or one is not present, the application can then simply fall back to a single or multi-threaded CPU algorithm.

Chapter 5

Optimisation

This chapter focuses on exploring the methods used to accelerate and parallelise the algorithms, beyond the scope of the description in the implementation chapter. A simple pair of parallel lines representing a discontinuity was used during the optimisation process. This was acceptable as the improvements in runtime scaled similarly to those of the ladybird curve set used in the experimental results chapter.

5.1 CPU

While using SIMD extensions was a planned option for the CPU version, it was not implemented due to the time constraints of the development period. In addition to this, the main scope of the project was to provide highly optimised GPU versions of the algorithms - the CPU versions, while otherwise optimised, are mainly for comparison and present as a legacy fallback.

Two custom multithreaded versions of the algorithm were implemented using the Win32 Pthread library. A 64 bit binary of this library has been successfully compiled, but is not yet publicly available - thus while these algorithms will function correctly in a compiled 64 bit version of the diffusion library, they are disabled in the source code at present. Parameters shared between the threads, as well as private exclusive variable memory spaces, are allocated and packed into structs, which are then passed in as pthread creation parameters for later use. The threads lock and wait until the

diffusion library is allocated work, at which point the threads will be unlocked and can run their worker functions, re-locking upon completion.

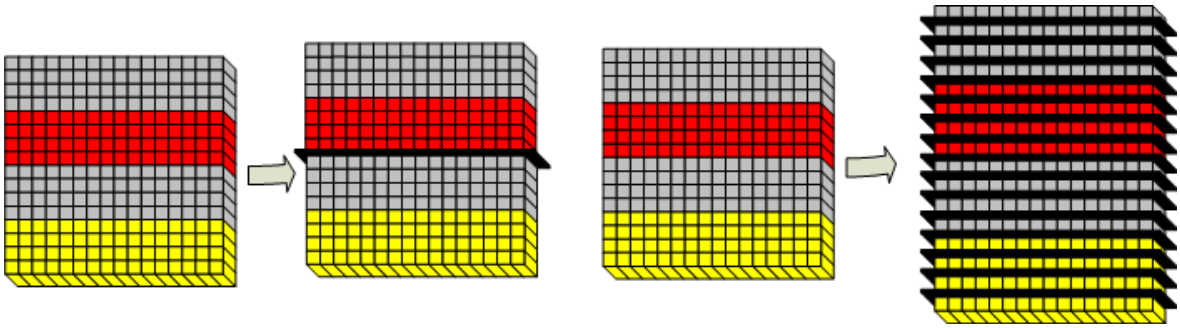

Figure 5.1: Static and work-unit based workload division

The first of these uses a static splitting technique. As each atomic unit of work (i.e. an iteration on a single pixel) takes roughly the same amount of processor time to complete, partitioning the image space into equal areas is a viable method of work load balancing. If an image has M rows, and the host machine has N processor cores, then we will create N threads - each with M/N rows from the image to process. A collection of master and slave mutexes are used to ensure that the threads remain synchronised correctly. A persistent pool of threads is also used - spawning and destroying threads creates some amount of overhead in the operating system's kernel, and for an optimised algorithm it is desirable to eliminate this.

The second custom implementation again uses the pthread interface, allowing theoretical 32 and 64 bit versions across Windows and Unix based operating systems to be compiled without changes to the source code. It divides the work up into a large number of disjoint work units, a quantity far greater than the number of threads. A thread pool takes work units from a mutually exclusive locked list, and processes each of them. When there are no work units left to process, the threads sleep and wait for the next batch of work. While a simple concept, ensuring that there is no possibility for thread starvation or or deadlock presents a small challenge.

A third method, referred to as "temporal load balancing" functions by performing

a tentative static split, and then adjusting boundaries on successive iterations based on how quickly the previous divisions terminated and finished. This takes advantage of the fact that uneven loads tend to take the same amount of processing time to calculate a given area on an image. This was not implemented as it was felt that the algorithm could not outperform either of the other two previously mentioned.

These custom workload dividers were implemented for the naive diffusion technique only, however they could easily be applied to the hierarchical pyramid versions contained within the library. A separate OpenMP based version was also added. This consisted simply of adding a *parallel for* directive to the main loop which iterates over each pixel. As OpenMP yielded similar/better results than the better of the two custom implementations, in addition to taking a tiny fraction of the time to add to the library, only an OpenMP version of the hierarchical pyramid algorithm was created.

5.2 GPU

A GPU kernel was defined, based on the CPU version of the naive algorithm. As detailed in the CUDA section of the background chapter, a single kernel is executed in a massively concurrent fashion by thousands of threads. These threads each have their own high speed register file, exclusive to them and inaccessible outside of the thread. The contents of these registers are lost unless they are saved back to device memory upon thread termination. Threads are grouped into thread blocks, which have access to a shared memory accessible only by threads in the block. Again, data stored in the shared memory is lost if it is not written back to device memory before every thread in the block terminates. It is slower than the register file, but significantly faster than main memory. Typically if a piece of memory needs to be accessed repeatedly by different threads in the same thread block, caching it in shared memory is worth considering. Shared memory is small (in the order of a few kilobytes on current CUDA devices) yet its low latency makes it worth using.

A lot of helper functions were necessary for calculating information such as the number of thread blocks, and the thread count of each thread block. While important to the library, their implementation is not worth exploring. CUDA .cu files contain C

code with extended functionality and syntax specific to CUDA. The NVIDIA C Compiler (nvcc) is invoked upon them, and will compile/translate the relevant portions of the code. When complete, nvcc will call your standard C compiler (in our current configuration, the Visual Studio 2008 C++ compiler).

The DiffusionImage class was altered so as to contain a pointer to memory on the GPU device for storing the image. Upon creation of an image object, this memory is allocated on the CUDA device - across many iterations, reallocating the memory can only introduce a performance penalty, thus it is avoided. Unfortunately nvcc does not support any kind of C++ or objectified input, so the GPU versions of the diffusion algorithms act as little more than intelligent wrappers for global functions. In porting the naive algorithm to the GPU, a number of steps were taken. Firstly, simply the diffusion code was ported. For each iteration, the image had to be loaded (already in float format) to the GPU's memory. The grid of threads would then execute, calculating the result and writing it back out to the GPU's DiffusionImage backbuffer. Upon completion, the back and current buffer pointers are swapped, as on the CPU. The results then had to be loaded back to main memory to perform the global difference calculation, which is a reduction operation.

The difference calculator function was then implemented directly on the GPU. This is beneficial for a number of reasons - not only can the GPU outperform the CPU if we make a highly hand tuned version of the summater, but the need to copy the new iteration back out to main system memory is eliminated. Now, for each diffusion operation, only two memory transfers are required - providing a measurable speed increase.

5.2.1 The parallel reducer

Typically summating a large number of values is a function suited to a serial processor. Implementation on a GPU is difficult; since to obtain high performance we must divide a workload over thousands of threads. However, the NVIDIA CUDA programming manual refers to an example of a similar technique - a massively parallel reduction. By adding a similar algorithm, an extremely high performance convergence calculation on

device was realised.

A fully unrolled, templated reduction is provided. Essentially we generate a number of similar kernels which can be used to consolidate values down by half until eventually there is a single value left, representing the total global difference for the iteration. Since the front and back image buffers are kept on the GPU device, there are no memory loads to contend with. Certain areas of memory have to be declared volatile to prevent the compiler from reordering store operations, which would disrupt the result and introduce error. For versions of the template which are for consolidating the final, small amounts, the shared memory allocated must be rounded up so as to not allow the threads to go out of bounds (empty values will be consolidated, but this is a moot point with regard to performance).

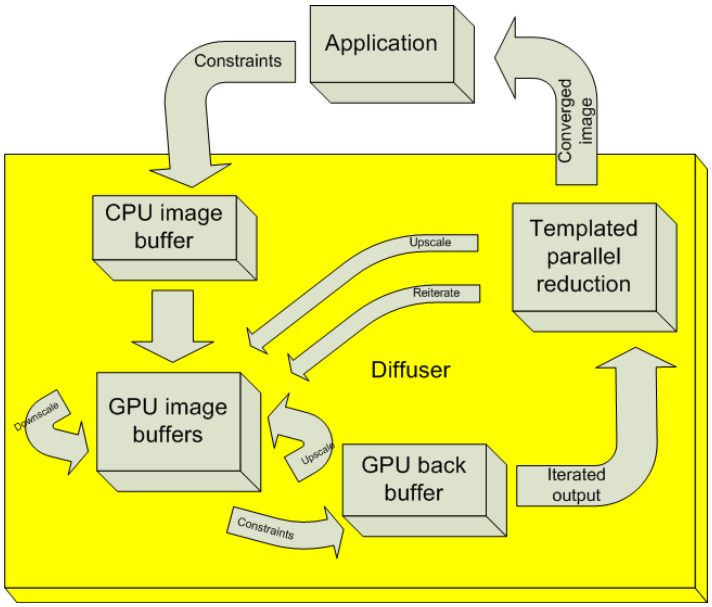

Figure 5.2: The hierarchical pyramid algorithm's behaviour, running on GPU

In order to further accelerate the process, consider that if we have an N*N resolution image, the number of values needing to be consolidated will be 3*N*N, taking into account the three channels. The naive diffuser is already optimised so as to make maximum use of fast register storage, avoiding the wait times for data to be streamed in from main memory where possible. The colour channels are stored here in register

- thus consolidating them into one floating point difference value per pixel and writing that out to a buffer (which is then passed to the templated reducer) we gain another speed increase. The solution NVIDIA provide has been modified to accept non power-of-two quantities of values. Experimental evidence suggests that this change does not cause a slowdown at all for the system sizes relevant to the diffusion image solver - in fact, when compared with the alternative of buffering the array with trailing 0 values, it performs favorably. The use of constants and templating in the code allows for a full unrolling of the code by the compiler, amounting to an efficiency increase to roughly 1.2x.

In addition to these kernels, two additional ones for upscaling and downscaling images within the GPU memory have been added. Investigating clever caching and unrolling strategies here is unfortunately not worthwhile; downscaling will always require 4 reads from, and one write to device memory. These memory transfers are unavoidable. Upscaling can of course place the value into a register before writing it out four times to memory, and this is used in code. In addition to these points, the Neumann version of the downscaler must be mentioned. Due to the fact that 16 input constraints are involved instead of 4 per output pixel are used to construct the downscaled constraints, a lot of potential branches are introduced. If we have N threads, and half do take a branch while the other half do not, these two sets must be executed serially - if the block of code inside the conditional took M time, then with the branch taken they could take up to 2*M time with an alternative branch. In a best case scenario, many threads will stall. This leads to the Neumann boundary condition downscaler performing quite poorly on a GPU.

The diffuser function was revisited. It is heavily bound by memory latency - by using shared memory inside thread blocks, it was possible to create a cache of repeatedly accessed values, avoiding duplicate loads from slow device memory.

Initially images were split into work units as in the top of Figure 5.3. Images were split into rows, and each row would be processed by one or more thread blocks, depending on the width (there is a hard limit of 512 threads per thread block in current CUDA devices). In this case, it becomes immediately obvious that pixels sampled to the left and right will overlap within the thread block, this data can be cached. However, by

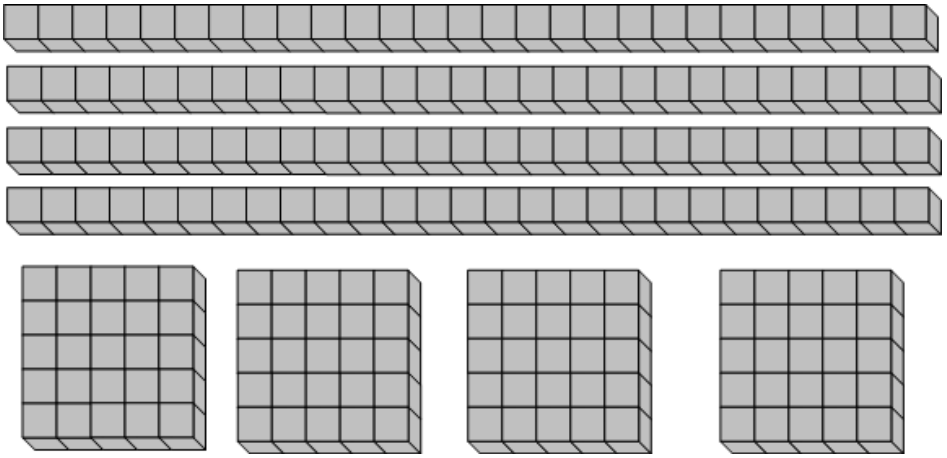

Figure 5.3: Thread organisation techniques

rearranging the area of the image processed by a thread block to be square, we maximise this overlap. 16x16 blocks (256 threads per block) yielded good performance, so these dimensions were hard coded into the kernel. A piece of shared memory of 18x18 size (a 1 pixel border is required to facilitate edge pixels) is introduced. Threads begin by loading one element to shared memory (two for border threads) and then stall until all data has been loaded. Then they proceed as before, except referencing the faster shared memory cache instead of slow device memory. Note that a __syncthreads() function might be here in other circumstances - however the final if statement in the diffusion function effectively introduces a synchronisation barrier.

Caching to shared memory in this fashion reduced the runtime of the entire diffusion process by 30%.

Chapter 6

Experimental results

When benchmarking the various combinations of diffusion algorithm and upscalers / downscalers, it was important to find a common ground with the resulting images. Due to the nature of the first downscaler function, for example, the speed of propagation of colour across large distances in the image was not as great as with the improved version.

Figure 6.1: The input curve raster used for benchmarking

Therefore, it will take more iterations for the image to converge towards a pleasing, acceptable solution - and more importantly, the convergence rate will be different. Our system depends on calculation of a global difference between each (overlapping) pair of successive iterations - this value is used to test for convergence. An ideal image using

a given algorithm combination requires a different convergence rate threshold to all other combinations.

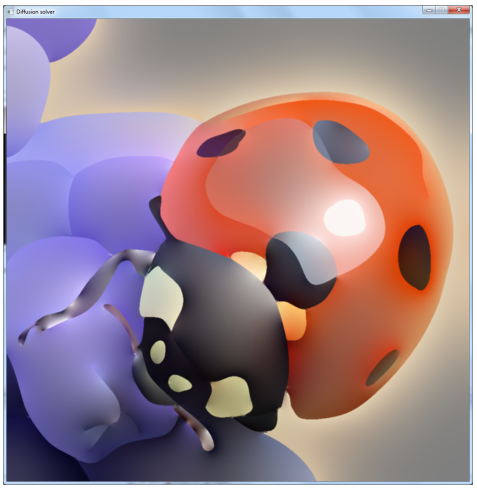

Figure 6.2: The gold standard output image generated whilst benchmarking

Approaching the problem by guessing values for the convergence threshold is unacceptable - we want to be able to globally vary output quality across all algorithms. To generate a set of suitable convergence thresholds, a "gold standard" of output was generated, suited to the example curve raster shown in Figure 6.1. Next, all benchmarks described below were run, effectively creating a search space by varying the convergence threshold in a sensible range. The reason for using this method may not be immediately clear - a seemingly easier method would be to simply iterate until the image falls below a certain global difference when directly compared with the gold standard - then this threshold could be used. However, this would not function well when using the hierarchical pyramid algorithm, because convergence thresholding is used at every level on the pyramid. We would need to generate a gold standard version for each level, and even then, differing thresholds on the various levels of the pyramid would potentially allow a faster convergence to an ideal solution.

Figure 6.3 shows an example of an incorrectly selected threshold - areas of the image which are not close to the constraints influencing them heavily remain uncoloured.

The results for this are radically different. As can be seen in Table 6.1, the naive

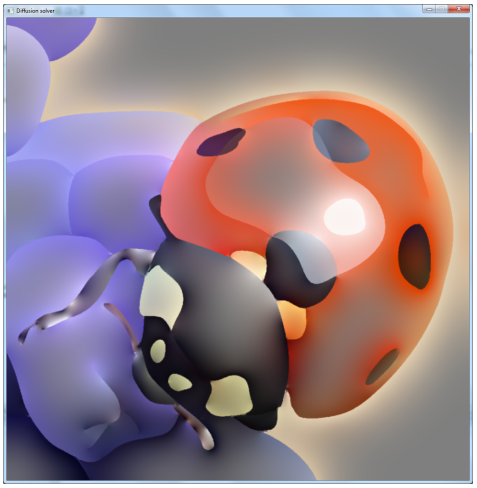

Figure 6.3: Halo artifacts in an incorrectly thresholded image

algorithm requires a far lower threshold - the further from constraints a pixel is, the slower it will be to be influenced by said constraints. Thus it follows on that the threshold for constraint convergence would be lower in order to produce a fully diffused image. With these figures established, benchmarks were run on the various algorithm combinations in the library. Each was run 20 times, and the average time was calculated. Although the runtime variation was trivial, we set aside the "warm-up" (see page 57) periods for CUDA and OpenMP. Firstly, experiments were executed upon the different algorithms with a fixed image input resolution.

The figures in Table 6.2 show that the OpenMP algorithm achieves the fastest performance, though it is quite close to the custom static load balancer. For this reason, the custom load balancers were not reused for testing with the hierarchical pyramid technique - OpenMP for the CPU version is sufficient, and significantly easier to implement.

The versions of the hierarchical pyramid algorithms were also benchmarked on both CPU and GPU for a 1MP image. These results can be seen in Table 6.3.

Using the hierarchical pyramid with the initial version of the downscaler yields a speed increase by a factor of almost 3. However, when we switch to the alternative version of the downscaler (which will set a child pixel to constrained if **any** parent pixel is constrained) the huge potential to increase speed is revealed. The improved

Algorithm	Acceptable threshold
Naive	0.2
Pyramid & Downscaler 1	1.0
Pyramid & Downscaler 2	50.0

Table 6.1: Acceptable Convergence Thresholds

Algorithm	Convergence Time
CPU Naive	706.760 sec
CPU Naive (Static load balancing)	329.842 sec
CPU Naive (Work unit load balancing)	333.211 sec
CPU Naive (OpenMP load balancing)	325.202 sec

Table 6.2: Naive load balancing methods on a 1MP image (4 threads)

downscaler allows the image to converge to a gold standard quality of solution with a speed increase of almost 60x, compared with the previous downscaler. When this was fully ported to GPU (i.e. with downscaler, upscaler, global difference calculation and hierarchical pyramid diffuser all done on GPU), the image is converged in 0.167 seconds on an NVIDIA Geforce 8800GT.

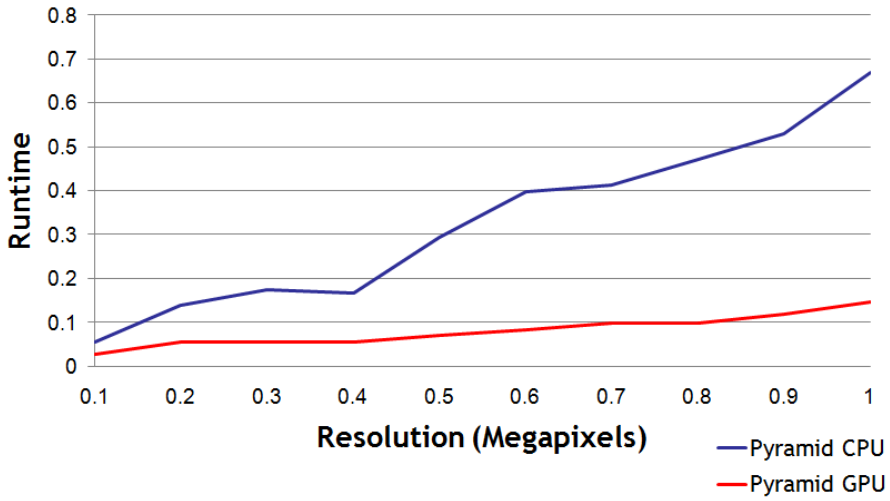

Figure 6.4: Hierarchical pyramid algorithms on CPU and GPU

Also provided in Figure 6.4 are the performance results of the CPU and GPU implementations of the hierarchical pyramid algorithm, with varying image resolution. Table 6.4 lists the results from the fastest CPU and GPU implementations of the naive

Algorithm	Convergence Time
CPU Hierarchical pyramid (OpenMP) with downscaler 1	117.347 sec
CPU Hierarchical pyramid (OpenMP) with downscaler 2	0.512 sec
GPU Hierarchical pyramid (8800GT) with downscaler 2	0.167 sec

Table 6.3: Load balanced CPU (4 threads) & GPU on a 1MP image

Algorithm	Convergence Time
CPU Naive (OpenMP)	325.202 sec
CPU Hierarchical pyramid (OpenMP) with downscaler 2	0.512 sec
GPU Naive	56.554 sec
GPU Hierarchical pyramid with downscaler 2	0.167 sec

Table 6.4: Load balanced CPU & GPU on a 1MP image

and hierarchical pyramid algorithms.

Graphics adapter	Convergence Time
NVIDIA GeForce 8800 GT	0.167 sec
NVIDIA GeForce GTX 460	0.080 sec

Table 6.5: Load balanced CPU & GPU on a 1MP image

Some experimentation with other GPUs was attempted, limited by availability. The results are shown in Table 6.5, and indicate that the careful structuring of the GPU based algorithms with regard to thread warp size, the library should scale to perform well on future GPUs, taking advantage of extra compute horsepower which may be available. Unlike many GPGPU implementations, our code should not require hand optimisation to get the best performance out of future GPUs.

Jeschke et al benchmark their implementation at roughly 20 frames per second [4]. We use their test input for our own benchmarks (see Figure 6.1), in order to be able to compare our results with theirs more directly. While ours does not quite match that, there are some hidden overheads which should be taken into account, which are not present in Jeschke's system. Firstly, for display (or potentially writing out to file) we load our converged image back off the GPU device into main memory. Jeschke et al do not do this - instead they render to a texture, and display it directly. CUDA memory transfers take what would be a significant portion of time in an algorithm which

completes one convergence in such a small amount of time. In addition to this, their benchmarks consist of moving a set of constraints slowly around within the boundaries of an image (demonstrated in the video accompanying their work). Since they are keeping the image on GPU, and reiterating with the starting point of a previously converged image which has simply been moved slightly, their image will be far quicker to converge. This is because large areas of the image will already have the correct convergence colour from the beginning of the frame - our benchmarks run freshly from a strict set of constraints only, with no previous iteration to work from. The results are based on a benchmark reflecting a flexible system - Jeschke et al were able to sacrifice some flexibility in favour of increased performance, as they were building a closed system suitable to their own tool set only.

Many GPU based algorithms suffer from the CUDA "warm-up" phenomenon . The first time a kernel is executed on the device, some driver optimisation is performed, calculating where thread blocks should be assigned on the GPU for future iterations [23]. This takes a significant amount of time - it can be in the order of seconds for our library, because due to the templated nature of the reduction algorithm, it effectively results in a large number of different kernels, all of which must be warmed up. However, this is not a problem for us - since programs using the library will be running the algorithms repeatedly, we can nullify this warm-up time by priming the device via a test call.

OpenMP also has a warm-up penalty - threads in OpenMP are not initialised until the first algorithm or portion of the code which uses them is executed. For long running programs, this can also add an overhead in the order of a few seconds to the runtime [24]. Therefore a warm-up call is also performed before recording benchmarking runtimes on the OpenMP based CPU versions. The custom schedulers initialise and suspend their worker threads when the library is loaded, so a warm-up is not necessary for them when testing.

Chapter 7

Conclusions & Future Work

The library presented by this work is capable of solving diffusion curve based images at high speeds, thus enabling real-time feedback to the artist. Similar projects have provided tools for editing these images, however most of these have focused partially or wholly on the tools for expressing the curves that make up the image. This library segregates the diffusion functionality into an easy to use dynamic link library, which can now be used by application developers to implement fully featured design tools without having to implement their own diffusion algorithms. Existing image manipulation software such as PhotoShop could also be extended using the library, via a plugin interface.

Currently, it is only possible to compile a 32 bit binary of the library using a 32 bit operating system, and a 64 bit binary using a 64 bit operating system. This is a limitation of the CUDA API and libraries provided - 32 and 64 bit support are mutually exclusive, and depend on the installed development environment. However if the NVIDIA C compiler and associated libraries are adapted to support cross compiling, the library and project setup will not require modification.

By taking advantage of available GPU hardware, and also supporting multi-core CPU hardware, we provide a robust library capable of executing on a variety of different computer architectures and operating systems. It will also scale to take advantage of future devices which provide greater numbers of stream processors, and be capable of diffusing even larger resolution rasters in the shortest time possible.

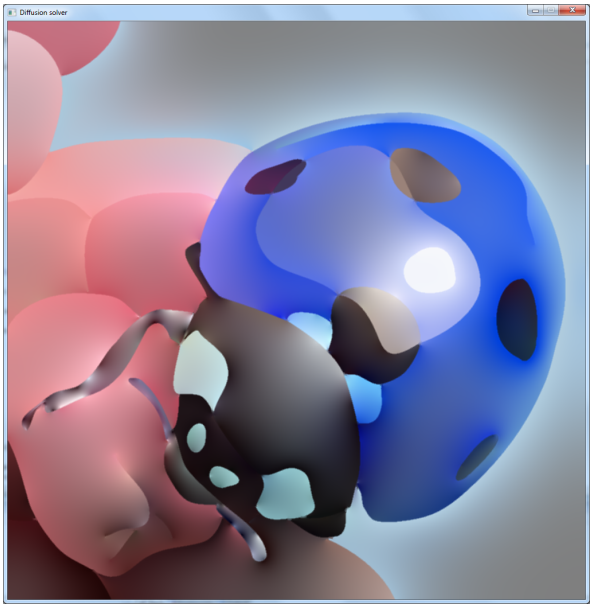

Figure 7.1: A modified version of the ladybird test image

Future work could include a port of the library to the OpenCL system, which would enable suitable ATI and Intel graphics adapters to run the algorithms, rather than just NVIDIA hardware. Support for systems containing multiple GPUs would also be possible - suitable hardware was not available for testing this during the development process. The library may also be capable of scheduling the workload over multiple GPUs which have different computational capabilities, while maximising device usage. The variable stencil could be repaired, providing increased performance.

Jeschke et al also apply some postprocessing effects to soften the sharp edges that can result from the diffusion process [4]. While the images produced by this library are appealing, the addition of a blur effect and other postprocessing options is also a potential area for future development. Our library in its current form has some simple options for remapping and clamping the values of already rasterised curves - some simple adjustments to an existing set of curves can be used to quickly produce alternative outputs. An example of this is shown in Figure 7.1.

As already mentioned, the goal of this project was to create a reusable library. A future project based around developing a new set of tools which could link against it

would be a worthwhile undertaking. The creation of a new set of tools for expressing images as diffusion curves could easily build on this existing work - while some tool sets already exist [3], it is noted that additional brushes, and higher level tools could be advantageous. This could also focus on the processing of existing images - expressing photographs or conventional cartoons is possible [4], but tools which allow tracing of features could greatly improve the artist's efficiency. Showing a preview of the current output and a per-pixel difference measurement alongside original image would have obvious advantages. Detecting curves in an existing image, and then allowing the user to edit these as vectors rather than as a raster would also be an interesting area to explore.

Bibliography

[1] A. Orzan, A. Bousseau, H. Winnemöller, P. Barla, J. Thollot, and D. Salesin, "Diffusion curves: a vector representation for smooth-shaded images," in *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, (New York, NY, USA), pp. 1–8, ACM, 2008.

[2] H. Werner, "Studies on contour: I. qualitative analyses," *The American Journal of Psychology*, vol. 47, no. 1, pp. 40–64, 1935.

[3] J. McCann and N. S. Pollard, "Real-time gradient-domain painting," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–7, 2008.

[4] S. Jeschke, D. Cline, and P. Wonka, "A gpu laplacian solver for diffusion curves and poisson image editing," *ACM Trans. Graph.*, vol. 28, no. 5, pp. 1–8, 2009.

[5] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[6] I. Corporation, "Intel pentium 4 processors 3.06 ghz product specification and information.," *Intel Processor finder*, vol. SL6SM–SL6S5, 2002.

[7] J. Warren, "Adaptive multi-core sorting generator," *Final Year Project*, 2009.

[8] I. Corporation, "Intel pentium 4 processors 3.8 ghz product specification and information," *Intel Processor finder*, vol. SL8Q9, 2006.

[9] D. Koufaty and D. T. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 56–65, 2003.

[10] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," *SIGMETRICS Perform. Eval. Rev.*, vol. 18, no. 1, pp. 134–142, 1990.

[11] I. Corporation, "Mesi protocol on l1 and l2 caches for write protect (wp) memory.," *Pentium Pro Family Developer's Manual*, vol. 1, 2004.

[12] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[13] M. May, "Pthread benefits and annoyances experiencied parallelizing a sparse grid based numerical library," tech. rep., Department of Computer Science, Technische Universität München, 1999.

[14] M. Bailey, "Using gpu shaders for visualization," *IEEE Computer Graphics and Applications*, vol. 29, pp. 96–100, 2009.

[15] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[16] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[17] P. Bui and J. Brockman, "Performance analysis of accelerated image registration using gpgpu," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, (New York, NY, USA), pp. 38–45, ACM, 2009.

[18] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

[19] D. W. Wall, "Speculative execution and instruction-level parallelism," tech. rep., WRL Technical Note TN-42, 1994.

[20] D. Marr and E. Hildreth, "Theory of edge detection," *Proceedings of the Royal Society of London. Series B, Biological Sciences*, vol. 207, no. 1167, pp. 187–217, 1980.

[21] J. J. Koenderink and A. J. van Doorn, "The internal representation of solid shape with respect to vision.," *Biol Cybern*, vol. 32, pp. 211–216, May 1979.

[22] S. Connelly and A. Rosenfeld, "A pyramid algorithm for fast curve extraction," *Computer Vision, Graphics, and Image Processing*, vol. 49, no. 3, pp. 332–345, 1990.

[23] NVIDIA, *NVIDIA CUDA Programming Guide 2.0*. 2008.

[24] J. Bull, J. Enright, and N. Ameer, "A microbenchmark suite for mixed-mode openmp/mpi," in *Evolving OpenMP in an Age of Extreme Parallelism* (M. Mller, B. de Supinski, and B. Chapman, eds.), vol. 5568 of *Lecture Notes in Computer Science*, pp. 118–131, Springer Berlin / Heidelberg, 2009.