Brute Force Deshredding Algorithm Using the GPU

Submitted in partial fulfilment of the requirements of the degree of Bachelor of Science (Honours) of Rhodes University

Simon Naudé

Grahamstown, South Africa
October 2014
Abstract

The graphics processing unit (GPU) has seen significant increase in performance over the past few years. Hence the interest in using GPUs for more general purposes has increased. The higher number of cores on a GPU allows it to outperform central processing units (CPUs). However, since in certain aspects instructions executed on the GPU must be executed in lock-step, generally the same instruction needs to be executed on multiple data sets. A deshredder program was created to test the performance gain where using the GPU to perform matching of shreds. Tests were performed using varying numbers of shreds to test how the transfer overhead between the CPU and GPU affects the timing, and how accurate a simple matching algorithm able to be executed on a GPU can be. The deshredder program showed a significant speedup when run on the GPU compared to with execution on a CPU, thereby confirming the hypothesis that a deshredding program could benefit from being executed on a GPU.
ACM Computing Classification System Classification

ACM Computing Classification System Classification (2012 version, valid through 2014)


D.1.3 [Programming Techniques] Concurrent programming.

F.1.2 [Modes of Computation] Parallelism and concurrency.

General terms: Design, Experimental, Measurements
Acknowledgements

I would like to thank my supervisor, Dr Bradshaw. Without her guidance, support, and positive attitude I would not be where I am now.

I would like to thank my family and friends for their continuous support.

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs, Genband, Easttel, Bright Ideas 39, THRIP and NRF SA (TP13070820716). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard.
# Contents

1 Introduction ........................................... 1
  1.1 Problem statement .................................... 2
  1.2 Organisation of the thesis ......................... 2

2 Background concepts and related work ................. 3
  2.1 Parallel concepts .................................... 3
    2.1.1 Flynn’s taxonomy .................................. 3
      2.1.1.1 Single instruction, single data .............. 3
      2.1.1.2 Single instruction, multiple data .......... 4
      2.1.1.3 Multiple instruction, single data .......... 4
      2.1.1.4 Multiple instruction, multiple data ....... 5
    2.1.2 Synchronization .................................. 5
    2.1.3 Amdahl’s law .................................... 5
  2.2 General processing on a graphics processing unit .... 6
    2.2.1 Graphics cards .................................... 6
    2.2.2 Development environments ....................... 7
      2.2.2.1 CUDA ........................................ 7
      2.2.2.2 OpenCL ...................................... 8
  2.3 GPU architecture .................................... 10
    2.3.1 Platform model .................................. 10
    2.3.2 Memory model .................................... 11
      2.3.2.1 Global memory ................................ 12
      2.3.2.2 Constant memory ................................ 12
      2.3.2.3 Local memory .................................. 12
      2.3.2.4 Private memory ................................ 13
      2.3.2.5 Transferring data between the CPU and GPU ... 13
    2.3.3 Effective use of GPU memory ..................... 13
  2.4 Deshredding algorithms ............................... 14
2.4.1 Straight line shredding .................................................. 16
2.4.2 Cross-shredding .............................................................. 16
2.4.3 Implemented solutions .................................................... 16
   2.4.3.1 DARPA shredder challenge ........................................ 17
   2.4.3.2 Straight-line De-Shredder ......................................... 17
   2.4.3.3 Cross-cut Deshredder .............................................. 18
   2.4.3.4 Deshredding using image feature matching .................... 20
2.5 Summary ................................................................. 23

3 Design and implementation .................................................. 24
  3.1 Overview of the system .................................................. 24
  3.2 Auxiliary programs ...................................................... 24
     3.2.1 Digital shredder .................................................... 24
     3.2.2 Image to text converter ......................................... 27
  3.3 DeShredder program ..................................................... 28
     3.3.1 Design of the sequential version ................................ 28
     3.3.2 Design of the parallel version .................................. 32
     3.3.3 Basic algorithm for the parallel deshredder ................... 35
     3.3.4 Limitations of the OpenCL deshredder ......................... 38
  3.4 Problems encountered ................................................... 39
  3.5 Summary ................................................................. 39

4 Results ............................................................................. 40
  4.1 Performance analysis .................................................... 40
  4.2 Experiments ............................................................... 40
  4.3 Final results ............................................................... 41
     4.3.1 Performance .......................................................... 42
     4.3.2 Accuracy .............................................................. 44
  4.4 Summary ................................................................. 46

5 Conclusion and future work ................................................ 47
  5.1 Summary of results ....................................................... 47
     5.1.1 Performance .......................................................... 47
     5.1.2 Accuracy .............................................................. 48
     5.1.3 Difficulty of implementation .................................... 48
  5.2 Future work ............................................................... 48

Appendices .......................................................................... 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Initial Kernel</td>
<td>55</td>
</tr>
<tr>
<td>B Final Kernel</td>
<td>56</td>
</tr>
<tr>
<td>C Shred example 1</td>
<td>57</td>
</tr>
<tr>
<td>D Shred example 2</td>
<td>58</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Memory specifications (sourced from [2, 25]) . . . . . . . . . . . . . . . . . . 12
4.1 Platforms used in the testing process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.2 Amount of time (s) to complete the matching process for kernel A . . . . . . . . 42
4.3 Amount of time (s) to complete the matching process for kernel B . . . . . . . . 43
4.4 Accuracy rating of the deshredder programs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
List of Figures

2.1 The four architectures defined by Flynn et al. (taken from [15]) . . . . . 4
2.2 Platform model (taken from [1]) ........................................... 10
2.3 Example of executing a branch instruction ................................ 11
2.4 Memory Model (taken from [1]) .............................................. 13
2.5 Example of a thread requesting a block of data (taken from [4]) ....... 14
2.6 Example of a thread requesting a block of data (taken from [4]) ....... 15
2.7 Example of the offset of a cross-cut shredder (taken from [6]) ......... 16
2.8 Example of a vertical time series (taken from [6]) ......................... 19
2.9 Example of a Luma time series match (taken from [6]) ................. 19
2.10 Example of the text blocks and shred model created using the horizontal projection histogram (taken from [18]) ......................... 21
2.11 Example of how the word length is calculated (taken from [18]) ....... 22

3.1 Complete processing during the use of the deshredder program ...... 25
3.2 Digital Shredder ................................................................. 26
3.3 Image-to-text converter ...................................................... 28
3.4 Matching algorithm for edge 1 and edge 2 ............................... 30
3.5 C# deshredder program .................................................... 31
3.6 Execution sequence of the OpenCL deshredder program ............... 32
3.7 OpenCL deshredder ............................................................ 36

4.1 Timing comparison between devices ...................................... 45
Chapter 1

Introduction

Parallel processing has been the focus of microprocessor development over the past few years. Companies such as Intel\(^1\) and Advanced Micro Devices\(^2\) (AMD) have moved away from increasing single thread performance, and have instead opted for increasing the total number of cores on their devices \([21]\). This trend is evident in Intel’s range of processors that have moved from single core Celeron processors to multicore Core i7 processors. In this development the clock speed has not increased substantially, but the number of processors has. The graphics processing unit (GPU) is another example of devices becoming more parallel orientated.

GPUs have increased substantially in performance over the years, allowing them to outperform central processing units (CPUs) in certain aspects \([9]\). Not only have GPUs started to outperform CPUs, but they have also become cheaper in terms of Flops\(^3\)/dollar \([13]\). This has created a growing interest in general purpose GPUs (GPGPUs) in an attempt to access the growing processing power of the GPU for non-graphical applications. A GPU however, has the limitation that it must be run in lock-step (where a group of cores execute the same instruction). This adds limitations to which processes can benefit from being executed on the GPU.

\(^{1}\)http://www.intel.co.za/content/www/za/en/homepage.html
\(^{2}\)http://www.amd.com/en-us
\(^{3}\)The number of floating-point operations per second, giving an accurate measure of the performance of the processor.
1.1 Problem statement

The intension of this research is to evaluate the performance gain obtained by using a GPU over a CPU, during the execution of a highly parallelizable program. It also considers the difficulty level of designing and implementing a program to be run on the GPU. For the purpose of this research a deshredder program was designed and implemented. The speed of the deshredding process on the GPU was tested against the speed of executing the process on the CPU.

The research objectives are as follows:

- Implement a deshredder program, that can be executed on multiple devices, including the CPU and GPU.
- Evaluate the performance of the parallel program compared with the sequential version by testing the time it takes to complete the matching process.
- Evaluate the difficulty and limitations of creating a parallel deshredder program using OpenCL

As a sub-objective, we also set out to investigate the accuracy of the parallel deshredder compared with the sequential one.

1.2 Organisation of the thesis

- Chapter 2 includes background information on GPUs and the deshredding process as well as past work done on deshredding.
- Chapter 3 describes how the deshredder program was designed and implemented.
- Chapter 4 presents the results obtained from tests preformed on both the sequential and parallel versions of the deshredder program.
- Chapter 5 discusses the results obtained in the context of the objectives set for this research, and identifies future work that can be done.
Chapter 2

Background concepts and related work

In this chapter, first some parallel concepts are discussed and explained. Then, an overview of the GPU architecture is presented, focusing on the process and memory models. Related work on deshredding is also discussed.

2.1 Parallel concepts

To fully understand the later chapters in this thesis we include a brief overview of some fundamental parallel processing concepts.

2.1.1 Flynn’s taxonomy

According to Flynn et al. [14] computer architectures commonly fall under one of the following models: SISD, SIMD, MISD, MIMD. Figure 2.1 depicts the four types of architectures defined by Flynn et al.

2.1.1.1 Single instruction, single data

Single instruction, single data (SISD) is the architecture used for sequential programs. A single processor element executes single instructions in sequence on a single item of data [15]. Since the execution is sequential, it can be traced from start to finish [7].
2.1. PARALLEL CONCEPTS

2.1.1.2 Single instruction, multiple data

The single instruction, multiple data (SIMD) architecture is used when multiple items of data need to be processed by the same instruction simultaneously [25]. This model is executed by two device architectures: array processors and vector processors [15]. Array processors include multiple processing elements that execute the same instruction on multiple items of data (executed in lock-step), while vector processors include a single processor element that executes the instruction sequentially on multiple pieces of data [15], which allows for the instruction to vary slightly depending on the data. GPUs follow SIMD architecture, in particular the array processor model.

2.1.1.3 Multiple instruction, single data

The multiple instruction, single data (MISD) architecture is not generally used in practice. This architecture allows for a single item of data to be processed by multiple in-
2.1. PARALLEL CONCEPTS

2.1.1.4 Multiple instruction, multiple data

Multiple instruction, multiple data (MIMD) architecture allows for each of the processing elements to act independently of the other processing elements, possibly executing different instructions on different items of data [15]. This architecture is used by modern multi-core CPUs [25].

2.1.2 Synchronization

Synchronization happens when two processors or devices need to be at the same point or contain the same data. For example, synchronization is used to ensure that two devices that store data in different physical locations are using the same data. Synchronization can also be used to ensure that all parallel processors have reached a certain point before any of them continues; this is achieved by using a barrier command where no processor will pass the barrier until all of them have reached the barrier. This is used for critical regions of code that require a previous process to have been completed before the critical code can be executed.

2.1.3 Amdahl’s law

To assess the efficiency of parallel tasks executed on a GPU instead of a CPU, Amdahl’s law is used [17]. Amdahl’s law defines speedup as:

\[ Speedup = \frac{\text{ExecutionTime}_{\text{without enhancement}}}{\text{ExecutionTime}_{\text{with enhancement}}} \]

where the \( \text{ExecutionTime}_{\text{without enhancement}} \) refers to running the algorithm on the CPU, and the \( \text{ExecutionTime}_{\text{with enhancement}} \) refers to running the algorithm on the GPU. A speedup of one signifies that the two algorithms ran in the same amount of time, and so no speedup was obtained. On the other hand, a speedup greater than one signifies that the GPU time is faster than the CPU time and so a definite speedup was obtained.
2.2 General processing on a graphics processing unit

There is no single architecture that is optimal for running all types of workloads, and most applications use a mix of workloads at runtime. A control-intensive application, where there are many branch instructions, would run best on a CPU, while a data-intensive application, where the same instruction needs to be applied to multiple items of data, would run far better on a GPU [4]. Owing to the fact that GPUs have to run in lock-step the performance obtained by running a control-intensive application on the GPU would be far lower than running it on the CPU. Because of this GPUs are mainly used for processes that work with the lock-step requirement.

2.2.1 Graphics cards

The processing power of a GPU is greater than that of a normal processor on a given die. However, it has the constraint whereby a group of cores must execute the same instruction (lock-step execution). This causes the GPU to be faster but less versatile, than other processors (such as the CPU) [9].

Recently, attention has been drawn towards using GPUs to accelerate non-graphic implementations; this recent interest is due to two aspects of the GPU: its price / performance ratio, and its evolution of speed [13]. The evolution of speed is mainly due to the gaming industry demanding better graphics, which in turn requires faster graphics cards. This demand has made the speed of a GPU double approximately twice a year [13]. Since there is such a high demand in the gaming industry, the price of the actual graphics card has remained relatively cheap in comparison to that of a CPU [13].

Rendering a graphical scene with a GPU is made up of a sequence of processing stages. These stages run in parallel and in a fixed order, which is known as the graphics hardware pipeline [13]. From a simple perspective there are four main stages in this process:

1. In stage one (vertex processing) the GPU obtains a 3D polygraph mesh, which is then converted into a 2D screen position render. During this, color and texture coordinates associated with the vertex are evaluated [13].

2. Within the second stage the vertices are grouped into triangles, and then scan-converted to generate a set of fragments. These fragments store the state information needed to update a pixel [13].
3. During the third stage (fragment processing) texture coordinates of the fragments are used to get colors from one or more textures. Mathematical operations are then performed to determine the ultimate color for the fragment [13].

4. The final stage is comprised of various tests (such as depth and alpha tests), which will determine whether the fragment should be used to update the pixel [13].

Although the GPU was designed to process graphics, because of its architecture, applications that do not require graphics rendering, but have certain characteristics can still be run on the GPU [16]. The characteristics of such applications can include any of the following:

- Computationally intensive. Since the GPU normally renders graphics, which need to be rendered in real time, the GPU can deliver a vast amount of computational performance [16].

- Highly parallel. The GPU is designed and optimized for parallel processing owing to the vast number of computing cores available, and so can add to the performance of parallel code [21].

- Throughput outweighs latency. The GPU architecture prioritizes throughput over latency. This is due to the difference between human perception and the speed of modern processors, where modern processors operate far faster than humans can perceive visual change. This allows for a higher degree of latency [21].

There are two major producers of GPUs, NVIDIA\(^1\) and AMD\(^2\) (Radeon), and each has its own programming model: NVIDIA uses CUDA, and AMD uses OpenCL. CUDA is only used on NVIDIA devices, while OpenCL can be run on either NVIDIA or AMD devices. Both of these are discussed in the next section.

### 2.2.2 Development environments

#### 2.2.2.1 CUDA

Developed by NVIDIA in 2006, Compute Unified Device Architecture (CUDA) introduced new components that were designed specifically for the GPU and solved several limitations

\(^1\)http://www.nvidia.co.uk/page/home.html

\(^2\)http://www.amd.com/en-us
of GPGPU [23]. CUDA introduced unified shader pipelines, which allows for each arithmetic logic unit (ALU) to be used to perform general-purpose computations. In order to create a GPU that can be used for more general purpose computing, NVIDIA had to build the ALUs in compliance with the IEEE standards for single-precision floating-point arithmetic [22, 27]. Once CUDA was developed the main issue NVIDIA faced was that there was no way of programming for CUDA without using Open Graphics Library³ (OpenGL) or DirectX⁴ which forced the programmer to disguise the program as a graphical task. To ensure people could access the power of CUDA, NVIDIA created the language CUDA C⁵ which was built on top of a restricted version of the C language with a few additional keywords that allowed the user to access features of CUDA [23].

2.2.2.2 OpenCL

The Open Computing Language (OpenCL) was released in 2008 by the Khronos Group [1], a nonprofit technology consortium [4], and has been managed by them ever since. OpenCL is classified as a heterogeneous framework, and so it supports application development that can be run across different devices manufactured by multiple vendors. This allows for an application to be run on a system that consists of GPUs and CPUs, and allows the programmer to utilize both these components [4].

OpenCL is coded in OpenCL C which was built on top of a modified version of the C99 language with additions for running data-parallel code. This allows code that is developed for one device to be run on any other hardware that is OpenCL compliant, creating device-independent applications [4]. The OpenCL architecture is split into four models: the platform model, the execution model, the memory model, and the programming model [19].

The platform model is comprised of a host connected to one or more OpenCL devices. The host is classified as any computer with a CPU running on a standard operating system while the OpenCL devices can be GPUs or a multi-core CPU [1]. Devices are made up of multiple compute cores, which contain multiple processing elements. Once the platform has been created the context needs to be created to manage the relevant resources including the OpenCL devices and command queue.

³https://www.opengl.org/
⁵https://developer.nvidia.com/how-to-cuda-c-cpp
The execution model is separated into two sections: kernel and host programs. A kernel is a unit of executable code that is run on an OpenCL device. When a kernel is queued, it is split into multiple work items, where each work item executes the same instruction on a different item of data. The kernel is executed over a defined n-dimensional index space [1], and from this each work item is given a global ID. So, for an image processing kernel, the kernel will be stored and executed in a defined 2-dimensional index space and the work item for pixel \( x = 80 \) and \( y = 93 \) would be given the global ID of \((80,93)\) [1]. Work items can also be grouped together to form work groups, which allows the work items to share local memory between work items within the same group. This gives the added benefit of being able to use synchronization on the work items within the group.

The host program is responsible for managing and executing kernels. This is done through the use of a context, which is a space where OpenCL devices that are in the same context can share objects such as programs, kernels and data buffers [11].

The memory model is split into four sections: global memory, constant memory, local memory, and private memory. Global memory is the section of memory that all work items have full access to for both the OpenCL device and the host device. This section of memory can only be declared by the host at runtime. Constant memory is part of global memory, and only allows read access to work items. The host however, has full access to the constant memory section. Local memory is located within a work group, and only work items within the respective work group have access to this section of memory. Private memory is a small section of memory that is only accessible to a single work item [1]. An extended version of this model includes the host memory, which contains the application data structures and the programs data, as well as the PCIe memory, which is part of the host memory but is accessible by the host and the OpenCL device. However, to modify this memory a synchronization state must be formed between the CPU and OpenCL device where the two devices will ensure that the information is the same on both devices [2]. Figure 2.4 displays how the memory model is designed.

The programming model simply maps the hardware threads to the GPU hardware [4].


2.3 GPU architecture

In this paper the Gigabyte Radeon HD 7970OC GPU was used. Hereafter we refer to this device as the HD7970. This section explains in some detail the architecture of this specific series of GPUs.

2.3.1 Platform model

The HD7970 is made up of multiple compute units, each of which contains multiple vectors and a scalar unit. The scalar unit handles branch instructions, read-only memory operations from the scalar level 1 cache, and other once-off operations that are independent. The scalar unit can execute one instruction per cycle [24].

There are four vectors in each compute unit, and each vector contains 16 stream processors or processing elements [2]. The processing elements execute instructions following the SIMD model of computation. Figure 2.2 displays how the platform model is designed.

![Platform model diagram](image)

Figure 2.2: Platform model (taken from [1])

The HD7970 contains 32 compute units each made up of 64 stream processors, giving the GPU a total of 2048 stream processors running at 925 MHz. This gives the GPU a maximum possible performance of 3.79 TFlops/s [2]. The GPU separates workloads into 64 work items, known as wavefronts, which are distributed to each compute unit for

---

6The number of floating-point operations per second, giving an accurate measure of the performance of the processor.
2.3. GPU ARCHITECTURE

processing. Due to the limitation of SIMD processing, the stream processors in a vector are all required to execute the same instruction at the same time. Different vectors in the same compute unit (and by extension, different wavefronts) however, can execute independent instructions. Processing elements on a GPU are allowed a certain degree of independence, and are able to process separate branch instructions. This is done by combining all possible branches and then processing them in sequence. This means that each stream processor will go through each branch instruction. In order to stop stream processors from executing branch instructions not relevant to their own work items, these can be disabled during execution and then re-enabled once the irrelevant section of the branch instruction has been completed.

As shown in Figure 2.3 all processors are active before the branch instruction, which states that the next code must only be run if \( y = 1 \). In this case only processors 0, 1, 3, and 5 have a \( y \) value of one and so are the only processors that should run this piece of code. Processors 2, 4, and 6 are disabled while the code is being run. After the if statement is complete processors 2, 4, and 6 are enabled and processors 0, 1, 3, and 5 are disabled for the else branch statement. Once the else statement has terminated all processors are enabled and continue with the remaining instructions.

2.3.2 Memory model

Much like the OpenCL framework the memory space in the HD7970 architecture is split into four sections: global memory, constant memory, local memory, and private memory.
2.3. GPU ARCHITECTURE

Each memory operates at a different speed and is accessible to different groups. Table 2.1 includes the size and access speeds for each type of memory available, and Figure 2.4 displays how the memory model is designed [2, 25].

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Size</th>
<th>Peak read bandwidth per stream core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private memory</td>
<td>256KB/CU - 8192KB/GPU</td>
<td>12 byte/cycle</td>
</tr>
<tr>
<td>Local memory</td>
<td>64KB/CU - 2048KB/GPU</td>
<td>8 byte/cycle</td>
</tr>
<tr>
<td>Global memory</td>
<td>3GB/GPU</td>
<td>(\sim 0.4) byte/cycle</td>
</tr>
<tr>
<td>Constant memory</td>
<td>48KB/GPU</td>
<td>4 byte/cycle</td>
</tr>
</tbody>
</table>

2.3.2.1 Global memory

Global memory is the largest section of memory, and all work items have full access to it. This section of memory can only be declared by the host at runtime. The HD7970 has a total of 3 GB of global memory. This memory is however also the slowest memory on the GPU with a peak read speed of approximately 0.4 bytes per cycle [2]. As such section of memory is generally used for the input and output of the program, and is the primary cause of any bottlenecks arising at runtime.

2.3.2.2 Constant memory

Constant memory does not reside in its own memory space; instead, it is in the same physical memory space as global memory. However, unlike global memory, constant memory is read-only to all work items and has a much faster maximum read time of 4 bytes/cycle. The host has full access to constant memory allowing it to write data that is commonly needed across all work items. It is also the smallest memory space as it only comprises of 48 KB for the entire GPU.

2.3.2.3 Local memory

In total the HD7970 has 2048 KB of local memory across the entire GPU, which is separated into 64 KB sections within each compute unit. This memory is only accessible by the work items that reside in the same compute unit/work group, allowing for faster sharing of memory between work items in the same compute unit without having to use the slower global memory. Local memory can serve one request per cycle (8b/s read...
2.3. GPU ARCHITECTURE

However, if all the work items request the same data, a broadcast occurs with the requested data at no extra cost [2, 25].

2.3.2.4 Private memory

The HD7970 has a total of 8192 KB of private memory (or registers) across the entire GPU. This memory is used as the private storage of a specific work item, and is only accessible by that work item. Each compute unit has a total of 256 KB of private memory, which is divided between each of the vectors. This memory is faster than all of the other memories with a peak read time of 12 bytes per cycle. Making correct use of this memory is vital to increasing the performance of the program being run on the GPU [2, 25].

2.3.2.5 Transferring data between the CPU and GPU

Moving data between the host and the device memory requires the use of the PCI Express bus, which has a maximum transfer rate of 16 GB/s. This is the slowest transfer time of all the memories, which can cause large performance loss.

Figure 2.4: Memory Model (taken from [1])

2.3.3 Effective use of GPU memory

Physical location and organization of memory within the GPU is important in order to obtain maximum speedup. This stems from the fact that when a memory request is made, a block of memory is returned [4]. If the data is sequentially organized in the memory
location, the thread can request multiple pieces of data, and the data can all be returned at once. Figure 2.5 shows how this process works.

If the data requested is not organized sequentially, the thread will need to request the data separately. This causes the device to return multiple, items which is less efficient. Figure 2.6 illustrates this process.

## 2.4 Deshredding algorithms

Paper shredding is a popular way of destroying critical pieces of information. Even today, many important documents are not stored electronically but instead stored as hard copies [8]. Shredding can be seen as a weak method of cryptography, where it separates paper into blocks of data and then rearranges the block. This method will secure the data but because no data is altered during this process, anyone with the patience and means will be able to access the information after some time [8].

A document can be shredded in multiple ways depending on the type of shredder, each of which has a different level of security. The security of the shredder is determined by the complexity of the shredding, where long strips of paper with a large width provide the lowest security and small strips of paper with a narrow width have a higher security level. According to Brassil [5] the US Department of Defense classifies a Class I secure shredder...
to be crosscut in nature with output pieces being no larger the 0.79375 mm wide and 12.7 mm long. This ensures that a single character 10 points in size will be vertically cut multiple times. He goes on to state that this does not ensure that the document cannot be reconstructed, but will at least deter attempts at reconstruction.

There are two popular methods for shredding a document, namely, straight line shredding and cross-cut shredding. Each provides a different level of security and satisfies a certain need of the user such as speed over security, as discussed in the next subsection of this paper.

The act of deshredding is to take a document that has previously been shredded and attempt to put it back together to view the information. This is often used by government officials to re-assemble possible evidence. Reconstruction of a shredded document however is a challenge, and requires a large amount of resources [6]. This challenge led to the DARPA shredder challenge in 2011 created by the US Defense Advanced Research Projects Agency (DARPA) [3]. The DARPA challenge was a contest where teams had to reconstruct multiple sets of documents, with the difficulty of reconstruction increasing with each document ranging from 224 pieces to over 6000 pieces to reassemble [3]. The teams then used the information obtained from the documents to answer specific questions and were awarded points for full or partial correct answers [3].
2.4. DESHREDDING ALGORITHMS

2.4.1 Straight line shredding

A straight line shredder is the most basic of shredders and so is also the easiest to reconstruct. The shredder only cuts along the papers length and commonly has widths ranging from 3.175 mm to 7.9375 mm. The mechanism makes use of two arrays of rotating ribbed metal bands, the width of which determines the width of the resulting strips of paper. This style of shredding is favored by those who wish to shred multiple documents quickly where the shredder is able to handle pages either stapled together or bound by paper clips [5].

2.4.2 Cross-shredding

A cross-cut shredder is a more secure method of shredding as it cuts both vertically and horizontally, which gives an added level of security. In most cross-cut shredders the horizontal cut will only occur every few inches on a page [5]. The dimensions of a cross cut shredded piece of paper range from 9.525 mm wide by 80.16875 mm long to 079375 mm wide by 4.7625 mm long [20]. The horizontal cuts with the cross cut shredder do not necessarily happen at the same time for each strip giving the added complexity of offset strips as shown in Figure 2.7 [6].

![Figure 2.7: Example of the offset of a cross-cut shredder (taken from [6])](image)

An extension to the cross-cut shredder is the confetti shredder, which includes crumpling the resultant shredded pieces [5].

2.4.3 Implemented solutions

Owing to the increased interest in this field, people from around the globe have attempted to create a solution to the deshredding problem. The DARPA challenge involved more than 9000 teams attempting to solve this problem [10].
2.4. DESHREDDING ALGORITHMS

2.4.3.1 DARPA shredder challenge

The DARPA challenge [3] forced teams to design different ways of reconstructing multiple shredded documents. The top team “All Your Shreds Are Belong To U.S” made use of a computer-vision algorithm, coded by the team, which found possible matches and asked the user to verify the match. The solution included a scoring system, and made use of a repeating set of yellow dots found on the pieces of paper to help with the matching algorithm to return the best possible matches and to save time by having as little human input as possible [6].

The second placed team, “Shroddon”, used a similar approach, but also included a text recognition aspect to the matching algorithm in an attempt to determine the next most probable character in the word using dictionaries.

The sixth placed team, “UCSD”, decided on a crowd sourcing approach, which proved effective for the first few documents, but owing to fraudulence by the users they were forced to implement a security feature, which would determine how often a user can move a piece depending on their previous success [3].

2.4.3.2 Straight-line De-Shredder

The De-Shredder, developed by Chan, Gillespie, and Leong [8], was designed to handle straight line shredding of a single page. The method used is a pixel matching algorithm that matches the pixels on the leftmost side of one strip with the rightmost side of another strip [8].

After scanning the strips on a red background, each strip is fed through a MatLab script, which, using a color threshold technique, forces each pixel on the strip to either a pure red, pure white, or pure black pixel. A MatLab function\(^7\) is used to find the centroid of the strips which in turn is used to rotate the strips so that they are aligned vertically [8]. These strips are finally processed by the De-Shredder program.

The De-Shredder program creates two arrays for each strip, one containing the leftmost pixels of a strip and the other containing the rightmost pixels of the strip. These pixels are defined by the rightmost/leftmost pixel that is not a red pixel. These arrays are used in the matching algorithm. Once the arrays have been set up, the program checks for matching pixels between the left edge array of one strip with the right edge array of

\(^7\)The MatLab function is called bwlabel: http://www.mathworks.com/help/images/ref/regionprops.html
another strip; if the colours between the two strips are a match the matching coefficient increases. The amount by which the matching coefficient increases depends on the colour of the matched pixel. If the color of the matched pixel is white the matching coefficient will increase less than if there are two matching black pixels [8].

After the initial matching algorithm has generated the matching coefficient, the De-Shredder shifts one of the pieces up or down a single pixel and generates another matching coefficient for the two altered pieces. This is repeated for ±0.25% of the total number of pixels along the strips edge, with the final matching coefficient being the largest coefficient for the two pieces [8]. This helps avoid errors that may occur owing to small errors in the scanning process or previous image processing. A matching coefficient for every possible pair is generated.

Next, the program requires that the user specify which strips are classified as edge strips, for use by the reconstruction part of the process. Once the edge strip is specified, the program shows the user the strip with the highest matching coefficient, to which the user responds by confirming whether the strip is a match. This process continues as the document is slowly built up starting from one of the edge pieces. The user can view the reconstructed document to see if it is correct and easy to read at any point during this process [8].

Chan et al. [8] states that the De-Shredder program works well in re-constructing single page documents, but is far from a practical solution. The program is unable to handle multiple pages, or pages that have been shredded using the cross-cut method. The program also relies on human input to accept matches, which can take time, increasing the total time taken for the document to be reconstructed.

2.4.3.3 Cross-cut Deshredder

Deshredder is a program developed by Butler, Charkraborty, and Ramakrishnan [6] designed to reconstruct shredded paper shred using a cross-cut shredder. Deshredder uses an array of methods to reconstruct the document such as: vertical edge time series, Luma time series, color targeting matching, and user interaction.

Deshredder takes as input all the scanned pieces on a single sheet. The program then separates all the pieces into single shreds, which are then straightened using a vertical edge time series. The vertical edge time series gives the distance from a perfectly straight vertical line to the leftmost or rightmost pixel for each row in the image. The average of
these distances are then minimized to find the optimum $\theta$ (see Figure 2.8 for an example of this process). When implemented on the first test of the DARPA challenge, this method produced results with only a 2% error, which came about owing to the shred not being completely separated from other pieces [6].

$$\theta^* = \arg\min_{\theta} (V(y) - E)^2$$

Figure 2.8: Example of a vertical time series (taken from [6])

A Luma time series shows the leftmost and rightmost pixel in each row of the image. This is used to find the largest peak values along the edges, which are then used to create a Chamfer distance distribution. This allows the program to locate features of a shredded piece, which will produce a Chamfer similarity value. Once the similarity value is produced the program uses the value to attempt to match two strips that have similar features (For cross shredding it may only be a partial match as the pieces do not fit squarely with each other.) Figure 2.9 shows the feature matching in this process. This method is not perfect as shredded documents are generally sharp and distinct, and so an almost match would fail just as much as a complete mismatch would [6].

Figure 2.9: Example of a Luma time series match (taken from [6])

Color targeting allows for human input during the matching process. The user selects a specific color he/she would like the matching algorithm to look at and the system
2.4. DESHREDDING ALGORITHMS

highlights all the colors that are similar to the selected color. The user is now able to use a threshold to vary how much the system should consider certain colors. This can help the system focus on features that the user deems more important [6]. The user then moves to the match selector section of the program. The user selects which piece he/she would like to attempt to match to which the program responds with the pieces that have the highest possibility of matching. Once the user has selected a match the program stores the match in the reconstruction palette, which allows the user so see all the matched pairs and allows the user to do multiple transforms on the strips to attempt to perfect the match [6].

2.4.3.4 Deshredding using image feature matching

The system developed by Lin and Fan-Chiang [18] uses a two stage approach: the first stage uses image-based techniques to create a matching algorithm, while the second uses graph-based algorithms to restitch the document. The system was developed for a straight-line shredder, and was tested using a single page that was both digitally ‘shredded’ and physically shredded [18]. The shredded document is scanned on a blue background, and then put through object segmentation and length normalization. An image processing technique known as morphological erosion is used to remove the noise on the boundaries of the shredded piece that may have occurred during the shredding process and/or the scanning process. This is done by passing a structuring element (small shape or template) over the image. The structural element is a small 2-dimensional binary grid consisting of a pattern of “1”s and “0”s, where one of the points is set as the origin point. This point can be anywhere in the grid depending on what the programmer wants to do with the structural element. As the element passes over the image, the system does a logical check to see if the element fits at each point in the image. Equation 1 [12] shows the rule used for erosion where s is the structural element, and f is the image. The element is positioned so that the origin is placed above pixel (x,y) and then the rule is applied to the pixel [12].

\[
g(x, y) = \begin{cases} 
1 & \text{if } s \text{ fits } f \\
0 & \text{otherwise}
\end{cases}
\]

Next the system uses a histogram obtained from the horizontal projection of the shred to align the text lines of each of the shredded pieces with every other piece, which is vital to the matching process [18].

The system identifies three special types of pieces found in a shredded document: blank
2.4. DESHREDDING ALGORITHMS

pieces, leftmost piece, and rightmost piece. The blank pieces, which are defined by not having any text visible on them, are removed as they are of no importance to the reconstruction process and would be difficult to near impossible to place correctly. The leftmost and rightmost pieces are identified as having no black pixels on the left/right section of the piece and having black pixel on the right/left section of the piece, respectively [18]. These pieces are then used as the starting and ending points in the reconstruction process.

Using the horizontal projection histogram created above, the system locates and creates text blocks within each shredded piece separated by blank blocks, as shown for the first six strips in Figure 2.10. The system creates a binary pattern for each strip. This is achieved by first creating a shred model (the final strip in Figure 2.10), which is a strip that contains every possible location for a text block using the locations of the text blocks in each shredded piece [18]. Each piece is compared to the shred model to create the binary pattern, where a ‘1’ is assigned to those sections that include a text block, and a ‘0’ is assigned to the sections that do not include a text block matching the shred model’s text block. Based on the binary codes created, the system is able to group and sort the shreds using binary operations [18].

![Figure 2.10: Example of the text blocks and shred model created using the horizontal projection histogram (taken from [18])](image)

Next the system gives the shredded pieces a similarity measure, where a higher score indicates a higher correlation between the two shreds. Lin et al. [18] used two methods, namely, shred coding discrepancy and average word length, for doing this.

**Shred code discrepancy** The shred coding discrepancy check uses the binary pattern created for each shred and checks these for differences. Each difference gives the pair an accumulative negative score, so two pieces with no differences will have a perfect score of zero. A smaller score between two shreds can also indicate spatial familiarity [18]. Since this gives a broad check for similarity, the information gained is not accurate enough to begin the reconstruction process.
Average word length  The average word length check is the second method used to create the similarity score between two pieces. As a unit of measure pixel count does not work for finding the word length in a reconstructed section owing to the fact that the character size may differ both between the documents and also between different text areas in the same document. As an alternative, Lin et al. [18] suggested a method where each shred is divided vertically into four blocks. These blocks are either part of a word or part of a blank space between the words, where the blank space between words will always take up a full partition. The word length is taken as the rounded integer strip width [18]. Figure 2.11 shows an example of the vertical dividing of the strips and how the word length is measured.

![Shredded Document Reconstruction](image)

Figure 2.11: Example of how the word length is calculated (taken from [18])

Once the system finds the word length, a negative score is accumulated based on the difference between the word length and the average word length. Since word length in a document varies, a high score simply implies that the shred permutation is reasonable [18].

The reconstruction section of the process is divided into two stages and uses a graph based scheme to sort them. In the first stage, a digraph is created based on the scores given by the binary pattern matching. The directed edges are weighted by the bit difference between the pair of binary codes [18]. The second stage then focuses on finding the shortest path of the digraph created in the first stage. Since the edge pieces are defined in the special selection stage at the beginning of the deshredding process, it is possible to find the shortest path between theses two points [18].
2.5 Summary

This section introduced some core concepts of parallel computing. An overview of the architecture of the HD7970 was discussed, focusing on the memory models of the GPU. The OpenCL and CUDA programming frameworks were introduced, giving more detail on the OpenCL framework.

An overview was presented of how shredding works and what types of shredders are available. The process of deshredding was also discussed. Multiple currently implemented solutions for the deshredding problem were also discussed in this chapter.
Chapter 3

Design and implementation

This section gives an overview of the design of the system. It further provides detailed explanations of the auxiliary programs used and created, how the deshredder program was designed and implemented, and the problems faced during the implementation process.

3.1 Overview of the system

A number of auxiliary programs were created to aid the deshredder program, this was done in C# using Visual Studio 2012. These include a digital shredder program and an image-to-text converter program. Figure 3.1 shows an overview of the process the user needs to follow to reconstruct a document using the deshredder program.

3.2 Auxiliary programs

3.2.1 Digital shredder

A digital shredder was created to assist in the testing of the deshredder program. Figure 3.2 shows the application interface for digital shredder program.

The user places an image of the page to be shred into the working folder of the shredder program (..\Debug\Files\). The shredder program detects these image files when it is started up, and displays the files in a list box. The user then selects the file s/he wishes
3.2. AUXILIARY PROGRAMS

The user can now select how many shreds s/he wants to cut the image into, and preview it using the ‘Preview’ button. Once the ‘Preview’ button is pressed the shredder program decides on how many columns will be in each shred (where the last shred may have fewer columns than the rest). Once the number of columns has been decided the program creates a new image with red lines inserted to demonstrate the boundaries of each of the shreds and displays the image to the user, who can then decide if the number of shreds is sufficient or if s/he wishes to change this.

Once the user is happy with the shreds, s/he clicks the ‘Shred’ button. This causes the shredder program to iteratively create several images, each containing the selected number of columns. The pixel values are thresholded to be either a black pixel or a white pixel. If the shred is made up of entirely white pixels it is discarded. Each non white image is saved in a separate folder in the working directory of the program (..\Debug\Shreds\), and is named with a preceding number signifying the shred number. These images are saved using the bitmap format because it is an uncompressed file format; this is essential as the pixel values are either 255 (white) or 0 (black), and compression of these files may attempt to alter the colours. Having other shades of black and white will force the need for another thresholding process once the image is input for matching.

Figure 3.1: Complete processing during the use of the deshredder program
Figure 3.2: Digital Shredder
3.2.2 Image to text converter

The converter program takes the shreds of a given file either digitally shredded or scanned, and converts the edge pixels into a text format, and saves multiple text files. This is to assist the OpenCL program to read in the images. With scanned images, the converter program will need to distinguish between pixels that are part of the shred, and pixels that are only present due to the scanning process. This is done by using a green background when scanning the shred, thereby giving each shred a green outline. The converter program will then only take the first and last non green pixels in each row of the shreds, which contain non green pixels to in case the top and bottom of the shred also have a green background, these will be saved as the edge pixels of the shred itself. This also allows for shreds to have a slightly jagged edge cut, as the program only takes the significant pixels. Figure 3.3 shows the the look of the converter program.

The user initially either scans in the shreds and places the images into the shred working directory as mentioned above, or uses the digital shredder program to create the shreds. If the user scans in shreds, they are also required to name the shreds using a common name with an incrementing preceding number to signify the shred number, and to make a file in the files working directory as mentioned above using the same common name. On startup the converter program detects files in the working directory, and the user can select the file s/he wishes to convert. This causes the converter program to find all files with the same common name and display them in a list box.

Once the user is happy with the selection, s/he can click the ‘Texify’ button. The program initially deletes all the text files in the text working directory (..\Debug\Text\) that include the same common name. Once it has deleted the old files the converter program reads in each image with the same common name. If the shred is made up entirely of white pixels the shred is discarded as it is not useful in the deshredding process. Otherwise the converter program finds the first and last non green pixel in each row and saves those values into a text file. The converter program also writes the lower number of rows between the two edges at the top of each text file, this is to account for the shred being slightly rotated and will ensure that the OpenCL deshredder program will not attempt to access invalid memory during the matching process. These text files are saved in the text working directory, with the same names as the shreds. Once each of the shreds has been processed, the converter program creates another text file using the common name of the shreds; this text file will include how many shreds there are in total.

The option to flip the selected shreds by $180^\circ$ is included in this section. This takes each
shred within the folder that has the same common name, duplicates it, and flips it by 180°. Flipping the shreds ensures that if the shred was initially scanned in upside down, it still has the chance to be correctly matched and reconstructed.

3.3 DeShredder program

As a proof of concept a C# program was written using Visual Studio 2012. This was done before creating the OpenCL program to ensure that a proper understanding of the matching algorithm was obtained before attempting to create a program that could run on a GPU.

3.3.1 Design of the sequential version

The C# program first requires the user to either manually scan in shreds on a green background or use the digital shredder program to create shreds. Manually scanning the shreds involves naming the shreds with a common name using an incrementing preceding number to signify the shred number, saving the images in the shred working folder as mentioned in section 3.2.1 as well as creating a file in the file working folder with the same common name. Figure 3.5 shows the look of the C# deshredder program.
The deshredder program detects the files in the working directory, and displays them in a list box for the user to select which file they wish to deshred. Once the user has selected a file, the program detects all the shreds that contain the same common name as the selected file and displays them in a list box. The user can then either choose to match all the shreds together, or match two specific shreds.

If the user wishes to match two specific shreds, they would select a shred from the list box and click the ‘Select 1’ button; this will cause the program to load the image and display it. The user can then either select a different image as the first shred, or select the second shred by selecting a shred and click the ‘Select 2’ button, which causes the program to load the image and display it. Once the user is happy with the selections, s/he presses the ‘DeShred’ button, which initially causes the program to threshold the shreds to ensure that they are made up of the correct pixel values, and then gets the non green pixels that make up the edges of the two shreds after which it sends the two selected shreds through the matching process and returns a matching coefficient and displays it.

If the user decides to deshred the entire document, s/he clicks the ‘DeShred All’ button. The deshredder program will run a threshold algorithm on the specified shreds to account for any scanned images and ensures that the pixels are either black, white, or green. An option is given for whether the shreds include flipped shreds, where there are two copies of each shred with half of them rotated by 180°. This is important for the ordering and reconstruction process of the deshredder program. The program now loads each of the images, and stores the non green edge pixels in memory. If the shred is made up entirely of white shreds, the program discards it as a white shred with no information does not need to be part of the deshredding process. This can be done as the edge pixels of a white shred would also be white, and so the two shreds that it would match, would be able to match each other since they would also have white edges. Once all the shreds have been loaded into memory, the deshredder matches each shred’s edge with every other shred’s edge in a brute force manner. As the matching progresses a matching coefficient is saved in a 2D array as shown in Figure 3.4. Once the matching is done the program displays each shred’s edge and the matching coefficient with each other shreds edge including itself. The accuracy of the matching algorithm can vary depending on the type of document shredded, therefore different values used for the within the matching algorithm (i.e. +15 for a black match, +1 for a white match, and -20 for a mismatch) may provide better results. Several text boxes allow the user to change the values used in the matching process including the amount used for the running total for both black pixel matches and white pixel matches.
3.3. DESHREDDER PROGRAM

With the 2D array completed the ordering process can begin. Since the deshredder attempts to do as much matching with as little human involvement as possible, the deshredder does not have a way of discerning edge pieces. Since there is no definite starting point for the ordering process, the deshredder program iterates through each shred as a starting point for the reconstruction process. The ordering process starts by adding a shred to an array, then using the 2D array of matching coefficients, the process finds the highest match for the current shred that is not included in the reconstructed array and adds the shred to the array as well as adding to a total matching coefficient variable. If flipped shreds are included, the ordering process will also ensure that the neither the flipped shred or the normal shred is contained in the array before adding the new shred. The process only matches the right edge of the shred with the left edge of the other shreds as if you match the right edge with another right edge the reconstruction of the page will have a flipped shred. Once all the shreds have been placed into the reconstructed array, the total matching coefficient variable is checked to see if it is larger than the current largest total matching coefficient; if it is, the current best array is replaced by the newly created array. After the algorithm terminates, the best reconstructed array is processed to attempt to reconstruct the image.
Figure 3.5: C# deshredder program
After the ordering process has found the best possible solution, the reconstruction process attempts to restitch the image using the solution. An image with the size: number of shreds * number of columns in the first shred (the first shred will always have the maximum number of columns when using the digital shredder) is created, which will either have the correct number of columns or more (if the final shred has fewer columns). The program then iterates through the reconstruction array, loads each shred into memory, and copies the pixel values into the correct location on the new image. Red lines are added in between each shred to help highlight the shreds, before displaying the image to the user. This process uses the original images, and therefore will include the green outline if the shreds were initially scanned in. A simple scoring algorithm is used to calculate how well the deshredder was able to reconstruct the shredded document. This can be done since all of the shreds have been saved in order, so the program will simply check how many consecutive numbered shreds appear in the final document. This gives a sense of accuracy for the deshredder even though accuracy was not the main focus of this study.

### 3.3.2 Design of the parallel version

Figure 3.6 shows an overview of the execution sequence of the OpenCL deshredder program.

![Figure 3.6: Execution sequence of the OpenCL deshredder program](image)

The OpenCL program was programmed in C++ and OpenCL using Visual Studio 2012. The user is required to first use the converter program mentioned above before running the OpenCL deshredder program. Figure 3.7 shows the look of the OpenCL deshredder program after it has completed the matching process.

Once the program is running, the user can select to run the program on the CPU or the
3.3. DESHREDDER PROGRAM

GPU, after which it prompts the user for the common name of the shreds they wish to deshred and how many iterations the user wants to run. The program then looks through its working directory for the file with that common name, and reads in the setup file that was created by the converter program to obtain the number of shreds. The program reads in each of the shred files and stores all of the pixel values in a 2D array. Once the program has read in the pixel values, it flattens the 2D array to prepare it for use in OpenCL as arrays in OpenCL must be a one-dimension array.

Once the data is ready, the program is required to initialize certain items required for running an OpenCL program. The platform must be set, and the device must be initialized. OpenCL allows for both GPU devices and CPU devices, where the OpenCL compiler will attempt to parallelize the code in the kernel by creating as many threads that the device can handle. This is done with the clGetDeviceID’s() method integrated in the OpenCL compiler. This method requires a valid platform (using the clGetPlatformIDs() method), the device type for the kernel to be run on, the number of devices requested, and a variable to store the information (using the cl_device_id type). Examples of the commands for creating a GPU and CPU device respectively, are as follows:

```c
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, numDevices,
               device, NULL);

clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, numDevices,
               device, NULL);
```

The number of devices is set to how many devices you want to discover. This allows the use of multiple devices, allowing the developer to select a secondary card or even use both devices for the same program. When both devices are used for the same program, the global id of the work item is shared uniquely between the devices. For example if device one sets a thread’s global id to one, device two will not set any global ids to one. This allows for the work load to be split between the two devices.

The context and command queue are the next two items that need to be initialized. The command queue is the set of instructions that need to be carried out. For example, the command queue could include: moving data to the device, running the kernel, and moving data from the device back to the CPU. This is done with the clEnqueue commands. Once the command queue is empty the program terminates. The context connects the device to the program, and the command queue.
Once the context and the command queue have been created, the program needs to be initialized using a kernel and then built. The building of the program is done by using the clBuildProgram() method, which returns an error if there is an issue with the compiling of the kernel. If this happens, the clGetProgramBuildInfo() method can be used to obtain more information about why it failed. Once the program has successfully been built, an instance of the kernel can be created.

Any parameters needed by the kernel must be set up before the program is run. This requires the creation of memory buffers for any arrays. For the deshredder program the input and output arrays need to be set up, where the input is the array of shred edges and the output is an array initialized to zero to store the matching coefficients. Memory is allocated using the clCreateBuffer() method and can be set to read only, write only or read and write. Once the buffers are created and linked to a local array using the clEnqueueWriteBuffer() method, the clSetKernelArg() method must be used to set the arguments of the kernel. The arguments of the kernel are the shred edges, the output array, the total number of shreds, and the number of pixels that makes up an edge.

Setting the work group sizes is the final task that needs to be done before the kernel is run. The device includes two work group sizes, the local work group size and the global work group size. Each work group size can have up to three work items. The global work group size is the total amount of work the device needs to do, i.e. the amount of threads the device will spawn. The method get_global_id() gets the thread ID, which can be used to determine which piece of information the thread should be manipulating. The local work group size determines how many threads are grouped together; each thread in the same group can access the respective group’s local memory. The method get_local_id() returns the thread’s local ID. The local work group size must be less than the global work group size, and cannot be greater than the device’s maximum local work group size, which can be obtained using:

```c
clGetDeviceInfo(device, CL_DEVICE_MAX_WORK_ITEM_SIZES,
               sizeof(maxItems), &maxItems, NULL);
```

Generally the local work group size will be set either to the device’s maximum work group size, or the same as the global work group size if it is less than the maximum work group size. In the case of the deshredder program, the global work group size is set to the number of pixels multiplied by the number of edges. The device’s maximum work group size is then obtained and checked to see if it is greater than the global work group size. If it is, the local work group size is set to the same number as the global work group size, otherwise it is set to the device’s maximum local work group size.
The device creates as many threads as it needs (depending on the global work size, and how many resources are available), and each thread runs the code in the kernel. The kernel simply matches a single pixel with each corresponding pixel on each of the shred edges, and adds to the matching coefficient in the corresponding location in the output array. Due to the number of threads that could requesting to get the current matching coefficient and the number of threads requesting to write to the matching coefficient at the same time, the value stored has the potential to become incorrect. For example if two threads receive the matching coefficient for a specific thread straight after each other, they will both add their matching score to it and attempt to save it, the second thread however would have obtained the matching coefficient without the first threads value added to it, and so when it save the value it will overwrite the value saved by the first thread. This causes the program to continuously give different results each time it is executed. The atomic \texttt{add()} function (as shown below) is used to fix this problem. The function locks the place in memory, reads the value from the memory location, adds the current match value to the matching coefficient, and saves the value back into the locked location in memory where it proceeds to unlock the memory location so that other threads can gain access to it. This will slow down the algorithm as threads will have to wait their turn before continuing, but it is necessary in order to obtain correct results. The results were tested against the C# deshredder program to ensure that they were correct.

\begin{verbatim}
atomic_add(&matchOut[currShred*shredCount+i], match);
\end{verbatim}

Once the program is complete, it returns the output array to the C++ section of the program, which displays each shred edge and its matching coefficient with each other shred edge.

Once the deshredder program has matched all the shreds, it writes the matching coefficient matrix to a text file. This text file will be used by the by the C# deshredder program when the user clicks the ‘’ button, which will attempt to load the correct text file depending on which file is selected in the list box. The C# program will attempt to reconstruct the document using the methods mentioned above.

### 3.3.3 Basic algorithm for the parallel deshredder

The algorithm is a simple brute force matching algorithm. It takes each shred edge, and matches each pixel with the corresponding pixel on every shred edge including itself. The C# algorithm does this sequentially as it is using a single thread, while the OpenCL algorithm does each pixel in parallel. Both algorithms use the same matching logic: if
3.3. DESHREDDER PROGRAM

Figure 3.7: OpenCL deshredder

the pixels are the same colour, it checks to see if that colour is black or white and adds a set amount to the matching coefficient. However if they are not the same colour then it decreases the matching coefficient by a set amount. The reasoning behind checking whether the pixel is black or white, is to give preference to black pixels as the page will be mostly made up of white pixels, and so finding two matching black pixels carries more weight than finding two matching white pixels.

The C# algorithm also includes a running total that increases every time it finds a matching pixel (more so when it finds a matching black pixel), and is reset to 0 when it finds a mismatch. This helps increase the accuracy of the deshredder program as shreds with more continuous matching pixels will have a higher matching coefficient. This is
not included in the OpenCL version of the deshredder program as keeping the running total requires a more sequential approach as it needs to know the amount to add from the previous pixels. Owing to the vast parallelization of the OpenCL code, this is not viable, as a thread near the bottom of the page may add the running total to the matching coefficient even if the pixel above may not yet have been calculated and may be a mismatch. This approach would also require the program to lock that resource using a method such as the atomic_add(), while a thread writes to it, and this has the potential of slowing the algorithm down as it will add a second lock to the program. The program already requires the threads to lock the matching coefficient array when adding to it.

The kernels used are given as Appendix A and Appendix B. Appendix A shows the initial kernel created; in this kernel the maximum number of threads the device creates is equal to the number of rows in the shreds. Each thread gets a pixel, which is dependent on its global ID, from the first shred edge, and matches that pixel with the corresponding pixels of each other shred edge. Once it has matched the pixel, the thread gets a pixel from the second shred edge and repeats the matching algorithm. This repeats until the thread has matched a pixel from each shred edge after which the thread despawns. This is a less efficient way of performing the matching algorithm as it forces each thread into a sequential algorithm where it must traverse two for loops.

```c
for (int currShred = 0; currShred < shredCount; currShred++){
    matchPixel = allShreds[currShred*pixelCount+thisPixel];
    for (int i = 0; i < shredCount; i++){
        ...
    }
}
```

This also has the possibility of not using the maximum number of resources the device may have to offer, and so can be improved upon.

Appendix B shows the final kernel created. This kernel creates a maximum thread count of \( \text{TheNumberOfEdges} \times \text{TheNumberOfRows} \) and therefore creates a thread for each pixel that needs to be matched. Each thread obtains a pixel by using the thread’s global ID to detect which shreds edge it should get the match pixel from as well as which pixel to use from that shred. As shown in the code snippet below:

```c
int GlobalID = get_global_id(0);
int matchPixel = 0;
```
int currShred = GlobalID / pixelCount;
int thisPixel = GlobalID - (currShred * pixelCount);
matchPixel = allShreds[currShred * pixelCount + thisPixel];

The kernel then matches that pixel with the corresponding pixel on each of the other shred edges. After the one pixel has been matched with the corresponding pixels, the thread despawns to allow the device to spawn a new thread. This kernel should be more efficient, as it creates more threads than the initial kernel, and reduces the amount of sequential code each thread needs to execute.

A further improvement could be made to the final kernel by attempting to remove the remaining “for loop” and doing some arithmetic with the global ID to obtain which two pixels the thread must match, where the global work group size is set to as many matches the algorithm would do in total. However, this may prove to me more detrimental as although it will greatly reduce the amount of work each thread will do, the overhead cost of creating a new thread may be greater than the cost of having a thread match more than one pixel.

3.3.4 Limitations of the OpenCL deshredder

The deshredder program has some limitations for it to work properly. The shreds must be from a straight-cut shredder and be cut down the entire length of the page; this means that pages shredded by a cross-cut shredder are not usable. Another limitation is that the deshredder program can only deal with pages consisting of black and white text, so it cannot handle colour images. The deshredder program can only handle single pages at the moment as it does not try to discern edge shreds.

Shreds that are cut horizontally instead of vertically will not get desired results with the deshredder program, as the matching algorithm is designed for vertical cut shreds and will suffer with horizontal cut shreds. This is because a shred that is horizontally cut could have the cuts between lines of text which will cause the deshredder program to not be able to correctly match/order the shreds since each shred could match with the same matching coefficient with each other shred. The shreds must also be processed before hand and rotated to the correct angle for better results.
3.4 Problems encountered

In the early stages of development the digital shredder program would save the shreds using the JPEG (Joint Photographic Experts Group) format, but due to the compression that occurs when saving a JPEG image, the image read by the deshredder program was not pure black and white. This was overcome by saving the images using the Bitmap format.

While developing the OpenCL version of the deshredder program, Visual Studio refused to allow the use of both OpenCL and OpenCV, where OpenCV would allow the program to load images directly. This was rectified by creating the converter program so that the OpenCL program could simply read in text files.

3.5 Summary

This chapter gave an overview of the design and implementation of the deshredder program (both the sequential version and parallel version), as well as the axillary programs created. The matching algorithm, as well as the different implementations of the kernel were discussed and explained. Finally some of the problems encountered were discussed.
Chapter 4

Results

This chapter first explains how the efficiency of the deshredder program was evaluated. Thereafter, performance and accuracy results are given for the experiments conducted.

4.1 Performance analysis

Amdahl’s law, as defined in Section 2.1.3, was used to determine the performance of the parallel version of the deshredder program.

For the OpenCL tests, a custom timer class was used for both the GPU and CPU to calculate the time taken from the point where the program called the kernel to the point where the command queue signaled that it was done executing. For the C# program the stopwatch class was used to measure the time taken from the point the matching process began until it had finished. Both kernels were tested to see if the increase in the number of threads and decrease in sequential code improved the efficiency of the code. All the results were recorded and compared.

4.2 Experiments

Experiments were conducted on both digitally shredded shreds, as well as scanned shreds. Due to time constraints, only a small number of scanned shreds was tested merely to ensure that the programs could handle manual shreds, while the bulk of the testing was done on digitally shredded shreds created by the digital shredder mentioned in Section
3.2.1. The number of shreds tested varied from a small number of shreds to a very large number. The smaller number of shreds highlighted the overhead of moving data onto the GPU compared with the sequential code running on the CPU. By increasing the number of shreds, a tipping point was found where the code executing on the GPU started to show a speedup since the advantage gained from executing the code on multiple threads exceeded the time taken to move the data onto the GPU. On the other hand, the larger numbers of shreds showed the true performance of the GPU executing highly parallel code.

To simulate very large test data without making the shreds only a single pixel wide, the tests used the same shreds multiple times. This allowed the algorithm to be run on a much larger sample size without the shreds becoming too narrow. It was possible to do this, as only time was being tested in this experiment and thus the accuracy and ability to reconstruct the document could be disregarded for these tests. Each test was repeated multiple times and the final execution time was recorded. This was done to account for natural speedup due to the operating system caching data after running the program more than once. The averages of the times, excluding the first run, was taken as the total time to complete. Speedup was obtained using Amdahl’s formula; that is dividing the CPU execution time (using the OpenCL code) by the Radeon HD7970 execution time as these are the two times most relevant given the objectives of this paper.

### 4.3 Final results

Testing was done on multiple devices as listed in Table 4.1, using both versions of the deshredder, i.e. the OpenCL and C# versions.

<table>
<thead>
<tr>
<th>Device</th>
<th>Number of compute units</th>
<th>Available memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gigabyte Radeon HD 7970 OC</td>
<td>2048</td>
<td>3</td>
</tr>
<tr>
<td>Gigabyte Radeon HD 7850 OC</td>
<td>1024</td>
<td>2</td>
</tr>
<tr>
<td>Intel(R) Core(TM) i7-3770 3.4GHz</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
4.3. FINAL RESULTS

Table 4.2: Amount of time (s) to complete the matching process for kernel A

<table>
<thead>
<tr>
<th>Total number of Edges</th>
<th>Intel i7-OpenCL</th>
<th>HD7970</th>
<th>HD7850</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.46</td>
<td>0.50</td>
<td>0.56</td>
<td>0.92</td>
</tr>
<tr>
<td>100</td>
<td>0.39</td>
<td>0.56</td>
<td>0.60</td>
<td>0.70</td>
</tr>
<tr>
<td>200</td>
<td>0.28</td>
<td>0.57</td>
<td>0.52</td>
<td>0.49</td>
</tr>
<tr>
<td>300</td>
<td>0.31</td>
<td>0.60</td>
<td>0.61</td>
<td>0.52</td>
</tr>
<tr>
<td>500</td>
<td>0.48</td>
<td>0.77</td>
<td>0.80</td>
<td>0.62</td>
</tr>
<tr>
<td>900</td>
<td>1.25</td>
<td>1.5</td>
<td>1.39</td>
<td>0.84</td>
</tr>
<tr>
<td>1000</td>
<td>1.47</td>
<td>1.75</td>
<td>1.70</td>
<td>0.83</td>
</tr>
<tr>
<td>1500</td>
<td>3.16</td>
<td>3.30</td>
<td>1.12</td>
<td>0.96</td>
</tr>
<tr>
<td>1560</td>
<td>3.39</td>
<td>3.09</td>
<td>3.01</td>
<td>1.10</td>
</tr>
<tr>
<td>2600</td>
<td>9.22</td>
<td>8.4</td>
<td>8.11</td>
<td>1.10</td>
</tr>
<tr>
<td>3000</td>
<td>12.28</td>
<td>11.14</td>
<td>10.11</td>
<td>1.10</td>
</tr>
</tbody>
</table>

4.3.1 Performance

Table 4.2 shows the time in seconds that the OpenCL deshredder program took to complete the matching process using kernel A (given in Appendix A) with regard to the number of shred edges it had to match. The speedup refers to the performance gained between the OpenCL deshredder program run on the Intel i7 CPU and the deshredder program run on the Radeon HD7970 GPU.

The speedup results of kernel A show that the CPU outperforms the GPU for the majority of the tests causing a speedup less than one until 1500 shred edges. This shows that the cost of transferring the data from the CPU to the GPU is greater than the benefits of using the GPU with this kernel. The GPU does however show some speedup starting at 1560 shred edges, but it still only provides a speedup of 1.10, and so is only 10% faster than the CPU. This shows that the cost of transferring the data can be overcome, but it still does not provide a very significant performance gain. This is due to the kernel including very sequential style code using the double for loop.

Since the number of threads the GPU spawns is equal to the number of pixels contained in each shred edge (which during the tests was approximately 870 pixels), the difference in time between the HD7970 GPU and the HD7850 GPU is never very large even though the HD7970 GPU can create double the number of threads the HD7850 can. The difference in time is solely a result of the clock speed of the HD7970 being greater than that of the HD7850.

As the page size increases the number of pixels contained on each shred edge also increases. This should show a better increase in performance between the CPU and GPU, and are
even a greater increase between the HD7970 and the HD7850 once the pixel count is greater than the maximum threads the HD7850 GPU can spawn.

<table>
<thead>
<tr>
<th>Total number of Edges</th>
<th>Intel i7-OpenCL</th>
<th>HD7970</th>
<th>HD785</th>
<th>Intel i7-C#</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.165</td>
<td>0.54</td>
<td>0.53</td>
<td>0.02</td>
<td>0.31</td>
</tr>
<tr>
<td>100</td>
<td>0.18</td>
<td>0.50</td>
<td>0.51</td>
<td>0.55</td>
<td>0.35</td>
</tr>
<tr>
<td>200</td>
<td>0.19</td>
<td>0.46</td>
<td>0.49</td>
<td>2.21</td>
<td>0.41</td>
</tr>
<tr>
<td>300</td>
<td>0.26</td>
<td>0.51</td>
<td>0.53</td>
<td>4.94</td>
<td>0.51</td>
</tr>
<tr>
<td>500</td>
<td>0.47</td>
<td>0.55</td>
<td>0.57</td>
<td>12.73</td>
<td>0.85</td>
</tr>
<tr>
<td>900</td>
<td>1.21</td>
<td>0.65</td>
<td>0.71</td>
<td>42.18</td>
<td>1.86</td>
</tr>
<tr>
<td>1000</td>
<td>1.46</td>
<td>0.69</td>
<td>0.76</td>
<td>51.47</td>
<td>2.12</td>
</tr>
<tr>
<td>1500</td>
<td>3.14</td>
<td>0.87</td>
<td>1.12</td>
<td>115</td>
<td>3.61</td>
</tr>
<tr>
<td>1560</td>
<td>3.38</td>
<td>1.05</td>
<td>1.74</td>
<td>124.25</td>
<td>3.22</td>
</tr>
<tr>
<td>2600</td>
<td>9.70</td>
<td>1.14</td>
<td>1.84</td>
<td>342.56</td>
<td>8.51</td>
</tr>
<tr>
<td>3000</td>
<td>13.20</td>
<td>1.49</td>
<td>2.38</td>
<td>455.50</td>
<td>8.86</td>
</tr>
<tr>
<td>5200</td>
<td>40.41</td>
<td>4.17</td>
<td>6.89</td>
<td>1357.16</td>
<td>9.96</td>
</tr>
</tbody>
</table>

Table 4.3 shows the time in seconds the OpenCL deshredder program took to complete the matching process using kernel B (given in Appendix B) with regard to the number of shred edges it had to match, and the time taken for the C# program to complete. The speedup once again refers to the performance gain between the deshredder program run on the CPU and the deshredder program run on the HD7970 GPU.

The C# deshredder program timings show how inefficient running this sort of algorithm is using a sequential approach compared with using a more parallel approach. The CPU is slower than the other devices, and the time required increases substantially as the number of shred edges increases.

Figure 4.1 and Table 4.3 show that for smaller numbers of shred edges, the CPU’s performance is able to once again outperform the GPU’s performance, as the cost of transferring the data is higher than the performance gain the GPU can provide. However, unlike the performance of kernel A, the GPU starts showing an increase in performance at around 900 shred edges. Taking 0.56 seconds less to run, and providing a speedup of 1.86, shows that it took almost half the time needed by the CPU. This can be accredited to the use of kernel B which makes better use of the number of threads the GPUs are able to create. The time taken for the smaller number of shreds shows how the cost of transferring the data to the GPU affects the time, where the time taken from 20 shred edges up to 300 shred edges are all around 0.51 seconds. The HD7850 was able to finish in a shorter time
4.3. FINAL RESULTS

even though it is a less powerful card purely as a result of the variation in the transferal
time of the data.

Figure 4.1 shows how the increase in shred edges also increases the time difference between
the HD7970OC and the HD7850OC GPUs. This is due to the HD7850OC having half
the maximum number of threads than the HD7970, and so shows how much better kernel
B uses the available resources.

4.3.2 Accuracy

Although accuracy was not an explicit objective of the research it was measured for
completeness. The accuracy of the deshredder program was determined by how many
shreds were in the correct order based on the previous shred number. That is, if shred
1 followed shred 0, points were added to the score, however if shred 2 followed shred 3
no points were added. The score was then tested against the maximum score possible for
the number of shreds being reconstructed, and turned into a percentage.

<table>
<thead>
<tr>
<th>Number of Shreds</th>
<th>Accuracy rating (%) C#</th>
<th>Accuracy rating (%) OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>57.14</td>
<td>57.14</td>
</tr>
<tr>
<td>13</td>
<td>25</td>
<td>16.67</td>
</tr>
<tr>
<td>26</td>
<td>20</td>
<td>28</td>
</tr>
<tr>
<td>33</td>
<td>18.75</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Shreds</th>
<th>Accuracy rating (%) C#</th>
<th>Accuracy rating (%) OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>13</td>
<td>100</td>
<td>97.67</td>
</tr>
<tr>
<td>26</td>
<td>84</td>
<td>88</td>
</tr>
<tr>
<td>33</td>
<td>53.13</td>
<td>87.5</td>
</tr>
</tbody>
</table>

Table 4.4 shows the accuracy percentage of the reconstruction process. This was tested
using two pages that were shredded into multiple shreds using the digital shredder. The
examples were taken from a screen shot with different zoom levels. Since page B contained
more detail than page A, it performed better during the reconstruction process overall.

The accuracy rating was taken for both the C# program and the OpenCL program as the
Figure 4.1: Timing comparison between devices

Number of shred edges

Time (s)

Time in seconds for the de-shredding matching process to complete.
C# program makes use of a running total variable during the matching process while the OpenCL code could not. The running total was introduced in an attempt to increase the accuracy of the matching process. With both page examples divided into three shreds, the OpenCL program only obtained 50% matching accuracy. This was due to the edges of the page matching so well together, and so the ordering algorithm placed the two together. The C# program however, managed a 100% accuracy as the running total variable caused the edges to match with a lower matching coefficient than the correct sides.

As the number of shreds increased, the accuracy decreased for page A as the shreds started to intersect more characters, and so more matches were being found with the shreds that were not a match. However, due to the good image quality of page B the accuracy only started declining when using 26 shreds. The accuracy of the OpenCL program was lower (except for 8 shreds on page A and page B) until 26 shreds for both pages were processed. This was purely due to not including the running total. With more then 26 shreds the accuracy of the OpenCL program is greater than that of the C# program.

The accuracy rating is very dependent on how many shreds are being reconstructed, and how the cuts of the page line up with the characters. However, the running total strategy does seem to cause the accuracy to decrease with higher numbers of shreds, while the accuracy without using the running total remains relatively high with the higher quality scan.

The matching algorithm used was a very basic algorithm, and so was not expected to give perfect results. A more robust algorithm could be used to obtain better results with regard to accuracy. This research focused on the speedup of the parallel deshredder program, while the accuracy of the algorithm was a secondary objective.

4.4 Summary

Based on the tests completed, it can be seen that once the workload becomes large enough the GPU is able to gain a significant speedup over the CPU. The size of the speedup was however, dependent on the kernel used as kernel A did perform as well as kernel B. The accuracy of the deshredder program was relatively good even when using such a simple matching algorithm. The running total helped increase accuracy with the lower number of shreds, however, it seemed to decrease the accuracy once the number of shreds increased. The quality of the shred was also shown to be very important for accuracy as the higher quality shreds faired better than the lower quality ones.
Chapter 5

Conclusion and future work

By using the increasing parallel power of a GPU this research set out to achieve the following: create a deshredder program using OpenCL, evaluate the performance of the deshredder program, and evaluate the difficulty of implementing a parallel deshredder program using OpenCL.

In this thesis a sequential deshredder system was created using C#, and a parallel version of the deshredder program was created using OpenCL. This fulfills the first objective of this research.

An additional sub-objective was to evaluate the accuracy of the parallel program.

5.1 Summary of results

With regard to the second and fourth objectives, a summary of the findings obtained are given in this section.

5.1.1 Performance

The parallel version of the deshredder program executed on the GPU clearly outperformed the CPU when given a large enough data set to work with, and when using the correct kernel, shown by the results in Figure 4.1 and Table 4.3. However, with a smaller number of shred edges, the cost of transferring the data to the GPU caused the execution to be slower than that of the CPU. This fulfills the second objective of this research.
It should be noted that the possible performance gain of the GPU is dependent on how the kernel is programmed and how well the available resources of the GPU are used. This is confirmed by the results in Table 4.2.

### 5.1.2 Accuracy

The accuracy of the deshredder program was relatively high given the simplicity of its matching algorithm, depending mostly on the quality of the scan rather than the number of shreds. The parallel deshredder program was able to reconstruct a document of 31 shreds with an 87% level of accuracy. The deshredder designed in this paper is not a perfect solution for the deshredding problem where a more advanced algorithm could be used to achieve better results. It is however, a good starting point for use by other deshredding methods. By using the processing power of the GPU the deshredder program can match an initial number of shreds, allowing the user to find shreds that belong together and using that information to decrease the number of shreds that would need to be used with other algorithms. This fulfills the fourth objective of this research.

### 5.1.3 Difficulty of implementation

Using a GPU as a GPGPU is not a simple task; it required a lot of effort in the design of the program to be run on the GPU. The programmer requires a good grasp of what kind of programs can and cannot be run on a GPU, since if the program requires many branch statements, the GPU will most likely reduce the performance of the program instead of increasing it. There is also complexity in debugging the program as there is no simple debugging tool for the GPU, and there are many areas for the program to go wrong. To address the third objective of this research, we conclude that from experience gained in this research, creating parallel programs to be executed on the GPU, is not a trivial task.

### 5.2 Future work

An improvement to the matching algorithm would be to not only match the corresponding pixels, but also to offset the shreds by a certain amount to account for errors caused by the shredding or scanning process. The program would start by offsetting a shred by a defined number of pixels, and then executing the matching algorithm. Once the matching
had been completed, the shred would be repeatedly moved down by one pixel until the offset was inversed from the original offset. The highest matching coefficient would then be used.

An extension to the deshredder program would be for it to support cross-cut shreds. In this case it would have to include matching the top and bottom of each shred and also considers white shreds.

Another extension would be for it to handle multiple pages and reconstruct them. This would require the program to decide which shreds are the most likely edge shreds, and attempt to reconstruct a page from one edge shred to the other. This could be done using human input selecting the edge shreds for the program. Allowing the matching of colour could be added, which would allow the deshredding of pictures. This would however, require the matching algorithm to be more complex as it would need to account for the fact that nearby pixels may not be the exact same colour, but a close variant of the neighboring pixels colour. This would add more conditions within the code, perhaps preventing it from being optimizable for execution on a GPU.

Human input could be included after the ordering process is completed to select which shreds were correctly matched, and which were incorrectly matched. The ordering process could then be run again with the new information to ensure certain shreds were no longer placed next to the previously identified incorrect shreds. This would become less viable as the number of shreds increased owing to the difficulty of deriving correct shreds from incorrect ones. Although the user would be able to obtain a correctly reconstructed document eventually, this method has the potential of drastically increasing the time taken for the process to be completed.

Using other matching processes such as character recognition and Luma time series would increase the accuracy of the deshredder program, however these processes would have to be run on the CPU as they require the algorithm to be more conditional and therefore would not gain performance when run on a GPU. However, the current deshredder program provides a good starting point for these other algorithms.

The use of CrossFireX, or Scalable Link Interface (SLI) in the case of NVIDIA GPUs, may have the ability to increase the performance. The increase in performance would be gained as the workload could be shared between the devices. However, the sharing of information would introduce the need for synchronization between the devices which increases the overhead of using the devices. This solution will provide a speedup for higher numbers of shreds, but would probably decrease the overall performance for smaller
numbers of shreds. The performance gain would not increase in a linear fashion with an increase in the number of devices because of the added overhead of synchronization of the data, and so may not provide the expected speedup [26].
References


REFERENCES


Appendix A

Initial Kernel

```c
const char *KernelSource =
   "__kernel void match(
   __global int* allShreds,
   __global int* matchOut,
   const unsigned int shredCount,
   const unsigned int pixelCount){
   int match = 0;
   int thisPixel = get_global_id(0);
   int matchPixel = 0;
   for (int currShred = 0; currShred < shredCount; currShred++){
      matchPixel = allShreds[currShred*pixelCount+thisPixel];
      for (int i = 0; i < shredCount; i++){
         match = 0;
         if (matchPixel == allShreds[i*pixelCount+thisPixel]){
            if (matchPixel == 0){
               match = match + 15;
            } else match = match + 1;
         } else match = match - 20;
         atomic_add(&matchOut[currShred*shredCount+i], match);
      }
   }
};";
```
Appendix B

Final Kernel

const char *KernelSource =
  __kernel void match(
    __global int* allShreds,
    __global int* matchOut,
    const unsigned int shredCount,
    const unsigned int pixelCount){
    int match = 0;
    int GlobalID = get_global_id(0);
    int matchPixel = 0;
    int currShred = GlobalID / pixelCount;
    int thisPixel = GlobalID - (currShred * pixelCount);
    matchPixel = allShreds[currShred * pixelCount + thisPixel];
    for (int i = 0; i < shredCount; i++){
        match = 0;
        if (matchPixel == allShreds[i * pixelCount + thisPixel]){
            if (matchPixel == 0){
                match = match + 15;
            } else match = match + 1;
        } else match = match - 20;
        atomic_add(&matchOut[currShred * shredCount + i], match);
    }
};
Appendix C

Shred example 1

and reconstruct them. This would require the program to decide which shreds are the
most likely edge shreds and attempt to reconstruct a page from one edge shred to the
other. This can be done using human input where they can select the edge shreds for the
program. Allowing the matching of colour could be added, this will allow the de-shredding
of pictures, this would require the matching algorithm to be more complex as you would
need to account for the fact that near by pixels may not be the exact same colour, but a
close variant of the pixels colour. This would add more conditions within the code and
so may not be optimized to be run on a GPU.

Human input could be included after the ordering process is completed where the user
would select which shreds were correctly matched, and which were incorrectly matched.
The ordering process could then be run again with the new information to ensure certain
shreds were no longer placed next to the incorrect shreds. This would become less viable as
the number of shreds increase due to the difficulty of deriving correct shreds from incorrect
ones, and although the user would be able to eventually get the document reconstructed
correctly this method has the potential to drastically increase the time taken for the
process to be completed.

Using other matching processes such as character recognition and Luma time series would
increase the accuracy of the de-shredder program however these processes would have to
be run on the CPU as they require more the algorithm to be more conditional and therefore
would not gain performance when run on a GPU. However, as the de-shredder program
is now, it can provide a good starting point for the other algorithms.

The use of CrossFireX or Scalable Link Interface (SLI) in the case of NVIDIA GPUs,
can have the ability to increase the performance. The increase in performance would be
gained as the workload would be shared between the devices, however the sharing of
information would introduce the need for synchronization between the devices which will
increase the overhead of using the devices. This solution will provide a speedup for the
higher number of shreds, however would probably decrease the overall performance for
the lower number of shreds. The performance gain would not increase in a linear fashion
as the number of devices increase due to the added overhead of the synchronization of
data and so may not provide the speedup expected [19].
Appendix D

Shred example 2

The user initially either scans in the shreds and places the images into the shred working directory as mentioned above, or uses the digital shredder program to create the shreds. If the user scans in shreds, they are also required to name the shreds using a common name with an incrementing preceding number to signify the shred number, and to make a file in the files working directory as mentioned above using the same common name. The converter program then detects files in the working directory, and the user can then select the file s/he wishes to convert. This causes the converter program to find all files with the same common name and display them in a list box.

Once the user is happy with the selection, s/he can click the ‘Texify’ button. The program initially deletes all the text files in the text working directory (..\Debug\Text\) that include the same common name. Once it has deleted the old files the converter program reads in each image with the same common name. If the shred is made up entirely of white pixels the shred is discarded as it is not useful in the de-shredding process. Otherwise the converter program finds the first and last non green pixel in each row and saves those values into a text file. The converter program also writes the lower number of rows between the two edges at the top of each text file, this is to account for the shred being slightly rotated and will ensure that the OpenCL de-shredder program will not attempt to access invalid memory during the matching process. These text files are saved in the text working directory, with the same names as the shreds. Once each of the shreds have