UNIVERSITEIT VAN AMSTERDAM
SYSTEM & NETWORK ENGINEERING

Distributed GPU Password Cracking

Research Project 1

Students

Aleksandar Kasabov
aleksandar.kasabov@os3.nl

Jochem van Kerkwijk
jochem.vankerkwijk@os3.nl

Supervisors

Marc Smeets (KPMG)
Smeets.Marc@kpmg.nl

Michiel van Veen (KPMG)
vanVeen.Michiel@kpmg.nl

May 20, 2011
Final version rev. 2

Abstract

This research project explores the possiblities of intergrating GPU processing power with a network cluster in order to achieve better performance with respect to password cracking. First a literature study is performed on the field of passwords in general, GPGPU computing and distributed computing through means of middleware. With these building blocks, combined with current single machine GPU password crackers, a comparison is made in order to answer the question on how GPU enabled distributed password cracking can be performed.

Contents

Chapter 1

Introduction

GPUs appear on the consumers market driven by the computer games industry. They are adapted to perform great amount of computations in a parallel fashion, required for graphic rendering. However, in recent years manufacturers try to extend their market share by targeting their GPU products for more general purpose problems, such as scientific simulations, weather forecasting, financial computations but also cryptography. These are all computational tasks which are inherently suitable for parallel computing.

This is extremely useful when it comes to the case of password cracking. The rise of stronger passwords have had the effect of alternative cracking techniques. Because the brute force technique is embarrassingly parallel, the additional processing power can be put to use effectively. The more processing power available, the faster one can perform such an attack.

GPGPU computing is generally a new topic. With respect to password cracking there are researches[1, 14] that have shown that password cracking performance can be enhanced by using GPUs. However, most of those are treated in environments with shared memory models, such as seen in single machines. Another problem is a limit on the amount of GPUs that can be placed within a single machine. In order to scale this processing power to another level, this concept must be projected upon a distributed environment.

KPMG has a distributed CPU cluster which is used for password cracking of common password hashing algorithms. It consists of 30 computers and is operated by a message passing interface (MPI) version of John The Ripper.

This project is established to explore the possibilities of using GPUs into a cluster to achieve distributed GPU password cracking. Hence, the following research questions are investigated in chapter 3.

- Which is the best password cracking method for a distributed GPU implementation?

- Which is the best architecture for distributed GPU computing based on predefined common criteria.

To achieve these goals the following sub-questions are researched in chapter 2.

- What is password cracking?
- What is a GPU and what application programming interfaces (API) exist for GPUs?
- What is distributed computing?
- What architectures for distributed computing exist?

Chapter 2

Theory

This chapter elaborates on the foundation which is needed to answer the research question on how to distribute password cracking on Graphics Processing Units (GPUs). The first part of this chapter explains general ideas on passwords and focuses on the common types of password attacks. GPUs can achieve greater computer speeds than CPUs for some given problems. The second section gives a description on GPUs and their Application Programming Interfaces (APIs). Because our main goal is to distribute the GPU power the last section is dedicated to main concepts of distributed computing and the available implementations available to allow for this.

2.1 Passwords

Computer passwords are sequences of characters and are used for the authentication of users. This section will describe the theory of passwords which will give some insights and will aid in understanding the process of cracking passwords.

2.1.1 Storage

To verify and acknowledge the correctness of a password, the system in question or a third party authenticator needs to reply yes or no. Whether or not this information is stored in the system or at the offside authenticator one might imagine that this is a weak point, as the password needs to be stored in order to answer the question if the user is authorised to use or enter the system. If the system is compromised, be it digital or physical, the passwords are available in plain text if no countermeasures are applied. Encrypting the passwords, or hashing the passwords, offers a solution here. [5]

As our main focus does not encompass cryptography nor handling eavesdropping we will disregard these aspects. What is related to this research is hashing.

2.1.2 Hashing

Hashing is the act in which a one-way function processes, or digests, a certain input of variable size and returns a deterministic fixed size output. This has the advantage that the password does not have to be stored in clear-text on the systems end and can be verified through a reference, not by using the actual password. Various hashing types exists, such as MD5, SHA and LM.

However, hashing does not solve every problem with respect to securing passwords. One of the main problems is that hashing does not prevent interception or eavesdropping using a communication channel, which in turn could be replayed in order to spoof authentication. To solve this, other secure channels and cryptographic methods should be applied. Encrypting the messages for example with the *Advanced Encryption Standard* (AES) in combination with a safe key exchange protocol, for example the algorithm proposed by *Rivest, Shamir and Adleman* (RSA), is an example of this.[5] However, as this is not part of our research scope it will not be discussed in further detail.

Collisions are also an important point of consideration as this is the main weakness of hash functions. It is possible for two distinct input strings to end up having the same hash, which can be explained through the pigeonhole-principle[16]. As there is a fixed bucket size for the hash function and an infinite amount of input possibilities there will exist inputs that will end up in the same bucket. This weakness can be exploited when cracking passwords. Before diving into

the available attack types we will first discuss what actually defines a password "strong".

2.1.3 Strength and key space

The strength of a password lies in the effort it takes in order to crack it. In order for a cracking attempt to take more time, one can increase the iterations needed. To do this, a more complex algorithm can be used or a bigger keyspace can be introduced. This results in more difficulty to recover the password.

Shannon defined a metric called entropy[13] which identifies the "informational value", or also coined information content, in a numerical sense. With this metric it is possible to define the best possible encoding for a certain input, making this metric very appropriate for password strength comparison.

Equation 2.1 shows Shannons mathematical representation of this metric. $I(X)$ is defined as the informational value of input data which is interpreted as a random variable and can be related to a probability mass function p. The base of the logarithm b can be freely chosen depending on the desired output.

$$H(X) = E(I(X)) = \sum_{i=1}^{n} p(x_i)I(x_i) = -\sum_{i=1}^{n} p(x_i)log_b p(x_i) \tag{2.1}$$

To make this idea more relevant to password cracking it can be applied to calculating the entropy of a string. Given a random string that consists of bits and a uniform probability mass function the entropy of that string can be calculated. The calculation for this can be derived from the general entropy equation and can be seen in equation 2.2 [2]. In this equation, b is defined as the size of the character set and l as the length of the password.

$$H = log_2(b^l) = l * log_2(b) = l * \frac{log(b)}{log(2)} \tag{2.2}$$

To give some examples for comparing password strength, a couple of character sets are given with their derived entropy based on a password length l. The table below shows that the bigger the key space of a character set is, the more bits are needed to represent that information. Also, the entropy of a password grows linearly as more characters are added to the password.

Type	Key space	Entropy (bits)
Arabic numerals	10^l	$l * 3.22$
Case insensitive Latin alphabet	26^l	$l * 4.70$
Case insensitive alphanumeric	36^l	$l * 5.17$
Case sensitive Latin alphabet	52^l	$l * 5.70$
Case sensitive alphanumeric	62^l	$l * 5.95$
All ASCII printable characters	94^l	$l * 6.55$

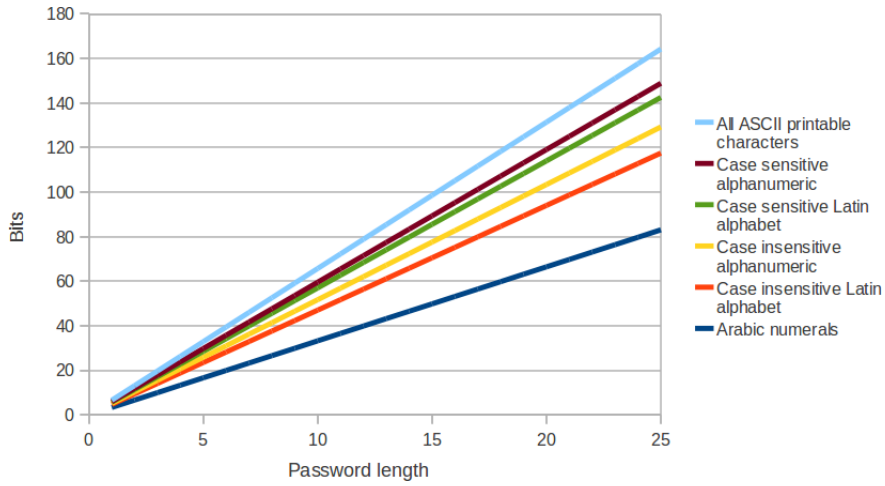

Figure 2.1: Entropy per character set

Figure 2.1 shows the table in a graph, which enables to derive the length of a password through the desired strength in bits and the character set in question.

There is however a difference between theoretical strength, as mentioned in the entropy table overview, and the password strength that is seen in reality.

People have troubles remembering a total random uniformly distributed password. They tend to go for passwords that are easy to remember, such as words and sentences as seen in languages, which represents other statistic properties and probability distributions regarding probability and downsizes the key space of possible passwords.[2]

2.1.4 Attack methods

In the previous section a definition of entropy is given as a metric to password strength. When cracking a password there is a trade-off between time and success rate. Where time performance is gained success rates drop, which will follow from the explanation of available attack methods that can be applied to recover passwords.

Brute force

As there is no such thing as an infinite password, the key space of a password is limited. By merely trying every possible item in the key space one is bound to fit the password that is searched for. As the size of the key space can be increased to an immense size, time and computational power are the main bottlenecks of this method.

If the key space is really big, one can imagine that it might be infeasible to wait for the time it takes to try every combination in the key space. Luckily there are other options to reduce this key space.

Dictionary Attack

The dictionary attack is based on the power of pre knowledge. If it is for example known that the targeted password shows linguistic properties, it is possible to downsize the key space by removing non-linguistic sequences. This has the advantage that less possibilities have to be tried and thus enhances the performance of the cracking speed.

Downsides to this method are the success rate and the reading of the dictionary. The key space is significant smaller than that of a brute force attack, but possibly does not hold the password due to wrong assumptions of a priori knowledge which heavily influences the quality of the dictionary and hence the success rates. The success rates can however be increased by applying so called smart dictionaries.

Smart dictionaries are regular dictionaries that apply a certain amount of rewrite rules. These rewrite rules use the words found in the dictionary and apply certain substitutions and permutations to them. An example to this is to rewrite the word with capitals, numbers and special characters; `monkey` by `MOnk#y`.

The dictionary method also introduces different delays with respect to the brute force method, namely I/O latency which is the biggest bottleneck in this method.

Pre computation

The normal iteration of password input requires a hash and verify step. By pre computing hashes, it is possible to do the checking in less time because the hashing function can be skipped. This is useless against so called salted hashes, which introduce a static additional factor during hashing (which would require a precomputed table for every salt).

A good example of such an attack is one that uses *Rainbow Tables*. This is an optimized pre computation table which does not store a tuple for every input string and output hash, but creates so called chains which group hashes together and store this more efficiently. It is also able to crack and compare password hashes blazingly fast.[11]

Main downsides to pre computation is the size of the resulting tables. As it is unknown what character set is used, as well as what salt is used, the optimized Rainbow tables require a large dataset to be available. Technically this could be loaded into memory for small character sets such as alpha numerical but is not very feasible for the “special” character sets. This leads to the same bottleneck as in the dictionary attack, which is latency due to I/O operations.

2.2 Graphics Processing Unit and Application Programming Interfaces for GPUs

The previous section described what password cracking is and what types of password cracking methods exist. This section outlines what a Graphic Processing Unit (GPU) is and how it applies to password cracking. A comparison to the way how a CPU works will also be presented. Finally, the section will list currently available interfaces for programming applications (APIs) using a computer's GPU.

2.2.1 Overview of Graphics Processing Unit (GPU)

Graphic Processing Units (GPUs) emerge in the early nineties[8] when the need for 3D processing in computer games unfolded. NVIDIA is the manufacturer who first[10, 7] coined the term "GPU". A GPU is a dedicated processor in a computer's video card developed to accelerate graphics rendering. This way it decreases the CPU load in a computer system. A GPU is optimized for doing floating-point calculations in a parallel fashion. It consists of several dozens of processors (etched on a multiprocessor chip), each of which is able to fork up to 1024 concurrent threads. Each processor has allocated memory, shared among the threads it creates. The memory component (Dynamic Random Access Memory - DRAM) is integrated in the graphics card and independent from the main Random Access Memory (RAM) memory in a computer system. Figure 2.2 represents the internal components of processing units which are the same for CPUs and GPUs.

Figure 2.2: Internal components of processing units - CPU and GPU

The *Arithmetic Logic Unit (ALU)* is the electronic circuitry which executes all arithmetic, logical and shift operations. The *Control Unit* is a digital component which manages stored program instructions and dispatches them to the ALU

for processing. Hence, the Control Unit needs to communicate with both the ALU and the memory. There are actually two types of memory on a graphics card - DRAM (Dynamic Read Access Memory) and cache. They are both used for storing program instructions and data and can be refreshed dynamically by the Control Unit. The cache memory is local to each ALU and stores the most recently accessed memory locations.

GPUs are designed for highly parallel computation for solving computational intensive tasks such as graphics rendering. It is achieved by allocating more transistors for arithmetic operations (data processing) than memory operations (data caching and flow control). This can easily be observed in figure 2.2 knowing that the transistors reside in the ALUs - there are many more ALUs in GPUs compared to one ALU in a CPU. That is why GPUs are explicitly suited to perform data-parallel computations with high arithmetic intensity[1]. The GPU executes each program on many data elements in parallel. Therefore no sophisticated flow control among the separate threads is required and memory access latency can be "hidden" with arithmetic calculations by other concurrent threads instead by using big data caches.

GPU computation is based on data parallelism - each data element of a task is processed by an individual thread. This is opposite to task parallelism used with CPUs where each task is assigned to a separate thread. Each application that needs to process a large data set can use a data-parallel programming model to speed up computation with GPUs. This approach is used for computing image projections on a large set of pixels and vertices in 3D rendering. Video encoding/decoding, image scaling, pattern recognition and physics simulation are also applicable for data-parallel programming model. Last but not least, password cracking is especially suited for parallel processing because the input data load can uniformly be distributed over a group of concurrent threads. Then they can all concurrently and independently perform the same hash function on different set of data until a match with the searched hash is produced.

2.2.2 How does GPU compare to CPU

GPUs can achieve greater computing speeds than traditional CPUs for some specific problems. GPU performance has doubled every 6 months since the mid-1990s, compared to the rate of CPU performance which doubles every 18 months on average (Moore's law)[3]. It is interesting to see what is the key architectural difference in their design and to compare how fast exactly each of the processing unit is.

[1] Arithmetic intensity - the ratio of arithmetic operations to memory operations

Processor architectures

Computer architectures were defined during the 70's by Michael Flynn[4] and remains pretty much unchanged since then. Modern graphic cards extend Flynn's classification by introducing the Single Instruction Multiple Threads processing model. In order to compare it to the rest of architectures, we should first take a look at them.

As we illustrated earlier, a single-core CPU contains one ALU, hence, it processes one instruction per processor cycle on a single input data. Such a computer architecture is called Single Instruction Single Data (SISD), it processes an input data sequentially and exploits no data parallelism (only task parallelism). Some CPUs can also perform Single Instruction Multiple Data (SIMD) instructions based on the supported instruction set. As the name hints, SIMD allows for a processor to process multiple data elements (data stream[2]) with one instruction. As one might suggest, SIMD appears with the first computer games as it speeds up image processing. With the advent of multi-core processors, the Multiple Instruction Multiple Data computer architecture started to be used. It allows a multiprocessor (with several cores) to process multiple instructions on multiple data elements simultaneously. Thus, processes which are forked on separate cores run asynchronously.

GPUs are making use of the SIMD architecture but exploit it to a new scale with their thousands of concurrent threads. In 2008 the Nvidia Tesla graphics card introduced[7] a new execution model for its multiprocessors - Single Input Multiple Threads (SIMT). It is based on SIMD and allows a multiprocessor (also called *stream multiprocessor* - SM) to create, manage, schedule and execute threads in groups of 32, all controlled by a single instruction as illustrated on figure 2.3[7].

Threads in a group (called a *warp*) start together but some can execute or idle, and diverge depending on conditional code paths. Thus, threads within a warp can be coordinated for data-parallel code or can be ignored for thread-parallelism. Nevertheless, threads in separate warps run independently on separate multiprocessors, no matter whether they are executing common or disjoint paths. Even though GPUs can schedule thousands of threads concurrently, that would not contribute to their high computing power if multiprocessors could not switch between them quickly. For instance, a thread switch on a typical CPU takes hundreds of clock cycles whereas a GPU can switch several threads per single clock cycle. In a CPU the (multitasking) operating system takes care for thread switching, replacing the thread context in the CPU registers each time a new thread is scheduled. In a GPU, threads are switched by the GPU itself. Apart from that, a GPU has more than one thread register; they are actually more than the number of processing cores. This allows for faster thread switching as registers should not be cleared and can simply be allocated for all the active threads. This is known as *hardware multithreading*.

[2] A stream is used to notate a collection of data which can be operated on in parallel.

The highly-parallel computing techniques in GPUs have their downsides as well. The main difficulty in achieving high computing performance is the fact that all processor cores must be kept busy at all times to allow for the full performance of a GPU. Threads must not diverge through conditionals but follow the same execution path. From the developer's point of view, the SIMT behavior can even ignored. However, if the code does not diverge threads in a warp noticeable performance improvements can be realized.

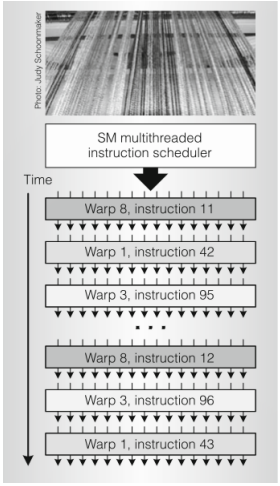

Figure 2.3: SIMT warp scheduling

Speed

Computation units are measured by the number of floating-point number operations which they can perform per second. This can be expressed through the amount of Giga Floating-Point Operations per Second (GFLOPS). Manufacturers publish this information in the technical specification of their processor. Previous research has already shown that contemporary GPUs can compute floating points 17[1] times faster than modern CPUs. This is mainly to the fact that they use more processor cores as we explained in the previous subsection. Figure 2.4[9, 1] helps to illustrate the computational advantage of GPUs over CPUs.

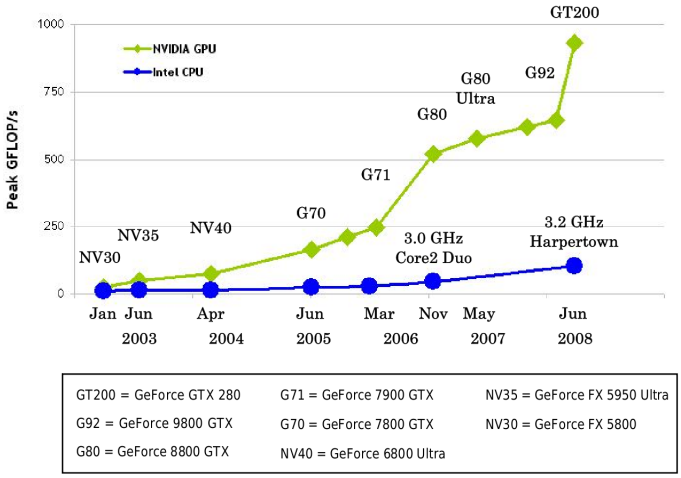

Figure 2.4: Peak performance in GFLOPS for some CPUs and GPUs

2.2.3 GPGPU API's

A GPU's sole purpose used to be rendering graphics. However, in 2003, graphic card manufacturers such as NVIDIA and ATI try to change that fact and publish application programming interfaces for their graphic products. This way application developers can directly target the GPU in computer systems to perform more general calculations, not related to image-processing only. The computing society calls this programming General Purpose on Graphic Processing Unit (GPGPU).The following list presents GPU APIs that are currently available which allow for general purpose programming.

Compute Unified Device Architecture (CUDA) CUDA is a parallel programming model and software environment which allows general purpose programming for CUDA-enabled graphics card from NVIDIA. CUDA abstracts the graphics hardware on three levels - thread groups, shared memories and synchronization. These can easily be controlled by the developer with a minimal learning curve through extensions to the C language.

Stream ATI introduced a special hardware architecture called FireStream to support the stream computing paradigm which enabled AMD to compete in GPGPU computing. Stream is a software development kit that enables GPGPU computing on AMDs compatible products. It is somewhat comparable to NVIDIAs CUDA, but it is not just a single library and compiler. For example, it also includes Brook+,AMD Core Math Library (ACML), AMD Performance Library (APL) and Accelerated Video Transcoding or (AVT)[17]

BrookGPU BrookGPU project provides a compiler and runtime environment for the Brook stream language (an extension of ANSI C). The project allows general purpose programming for GPU devices and requires DirectX or OpenGL (for NVIDIA/ATI GPUs) or Close To Metal (included in the Stream SDK from ATI). However, the project was discontinued in 2007.

DirectCompute DirectCompute allows general-purpose programming for GPUs, compatible with DirectX 10 (and future 11), for Microsoft Windows Vista and Windows 7. Programs with DirectCompute are written in the High Level Shader Language (HLSL).

Open Computing Language (OpenCL) OpenCL is an initiative by the Khronos consortium, who are also responsible for projects such as OpenGL. OpenCL is a new industry standard computing library that tries not only to combine GPU power to the computational resources, but any hardware accelerator available in a system. Major vendors such as NVIDA and ATI are already complying to this standard which brings heterogeneous computing to a new level, as there is no need for seperate or multiple interfaces like CUDA or Stream. Added, OpenCL also supports IBM Cell and Field-programmable gate array (FPGA) processors.

OpenCL is only available as a library, which has the downside that the developer is in charge of handling kernel specific parameters. [6]. This could also be a positive point, as there is no hassle with compiler specific peculiarities. This is for example the case when CUDA specific function calls are handled through a non-CUDA enabled compiler.

OpenCL is relatively new but it is quickly being adopted - there exist many online tutorials, developers' support from both NVIDIA and ATI and attempts for integration into various computational software (including Mathematica[3] and MATLAB). OpenCL includes a C-like language (C99) for developing GPGPU applications but there are also wrappers available for Java and Python.

OpenCL still does not fully support fine-grained parallelism and thread synchronization on shared data structures is left to be implemented by the developer through events.

2.3 Distributed computing

After we looked into the parallel computation design of GPUs in the previous section, we need to research how a computational task is actually distributed among remote nodes. This will help us identify the bottlenecks in distributed computing, which will be the foundations for our research criteria on some distributed architectures in subsection 3.2.2.

2.3.1 Overview

Distributed systems - a collection of independent computers that appears to its users as a single coherent system.[15]

Distributed computing has been researched for some decades already. It is a broad and dynamic field of study with lots of new innovations. Distributed computing becomes even more popular when frequency scaling was prevented by thermal constraints. Chips running on higher frequency heat up to the point that it is economically inefficient to cool them down. Hence, instead of manufacturing faster computing units, people started integrating units together. Distributed systems did not only increase the processing efficiency but also made computing more profitable. However, this introduced the problem of distributing processes which, initially, were computed in a sequential order. It turns out that distributing a computational task is not a straight-forward process and eventually depends on the nature of each particular computing problem.

[3] http://www.wolfram.com/mathematica/

Intercommunication among the nodes is based on message passing. It is a communication paradigm which allows processes to reliably transfer data structures, RPC calls or process signals. Both synchronous and asynchronous communication is supported. Message passing is also used for process synchronization in parallel computing environments which lays the foundations of transactions in higher-level programming languages. The most prominent example of message passing is the Message Passing Interface in High Performance Computing (HPC). Generally, an HPC cluster is considered to perform at least a 1 TFLOPS. MPI is a standardized specification for interprocess message exchange which ensures for portability of MPI code. We give more details on MPI in section 3.2.3.

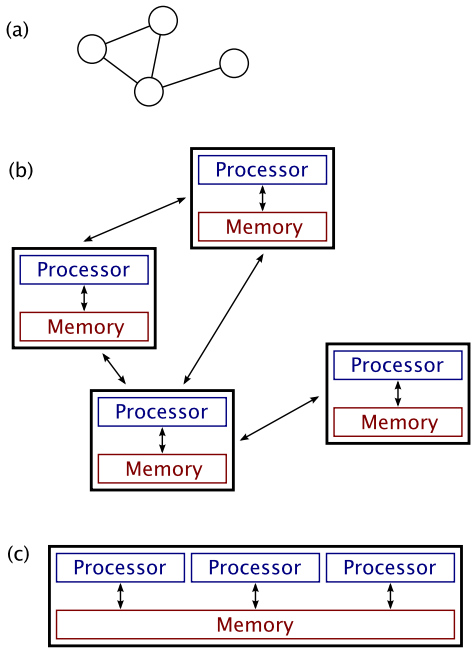

Figure 2.5: Distributed (a, b) vs parallel (c) computing [18]

No clear distinction between distributed and parallel computing exists. Distributed computing is performed by separate machines without shared memory which allows for loose coupling and interaction among the processing units. Thus, intercommunication is based on message passing. Parallel computing is performed by separate computational units placed on a single bus and sharing access to a memory unit, as depicted on figure 2.5.

Large-scale distributed systems often lack support for heterogeneous computing units. Middleware helps solving this issue. It resides on the lowest level in the application stack and implements communication (including group communication) and resource-sharing features. Therefore middleware masks heterogeneity

and provides a convenient programming model to application developers.

2.3.2 Models for distributed computing

Distributed systems have existed for decades and their variety has increased. Yet, depending on the function which each distributed node performs, we distinguish two types of distributed system architectures - centralized and decentralized. Both of them can also be combined to achieve hybrid solutions.

Centralized model The centralized model separates the computational nodes in functionality. Generally, this implies of one node, acting as a *server* and providing service for the others - the *clients*. A typical example of a centralized model is client workstations retrieving email messages from a central mail server. Centralized systems are easy to implement because the distinction of functional components in a computer cluster facilitate further software development and network management. However, such systems do not scale well simply because the central component becomes a single point of failure if the system load is increased and overloaded.

Distributed model The distributed model splits up network nodes in their physical location. This means each node performs the same logical functions, such as in peer-to-peer networks.

2.3.3 Bottlenecks in distributed applications

The parallel computational nature of distributed applications makes them intricate to develop. Developers must take care of all challenges of procedural (sequential) programming but also new ones applied to distributed computing - scheduling, interprocess-communication, synchronization and remote debugging. Additionally, many programmers make some false assumptions when building a distributed application for the first time which cost them painful learning experiences. These assumptions were summarized by Peter Deutsch (Sun Microsystems) in the 90's.

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology does not change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

Previous work of Rotem-Gal-Oz[12] provides a profound explanation of each of them in a contemporary context. Some of these assumptions apply to our

research project. The frameworks which we list in the following section must be appropriate to perform in a heterogeneous environment, supporting various types of GPUs from various manufacturers. Added, the GPUs nodes can be connected through an Ethernet network which makes proper scheduling the password cracking process difficult. Thus, the architectures must consider that bandwith and latency is variable.

2.4 Software frameworks for distributed computing

This section enumerates the software frameworks which are publicly available-frameworks. A brief description and a list of features derived from the manufacturers' specifications is also given for each of the frameworks. In the following section we will look further into each of these frameworks and how specifically they can be used for distributed GPU password cracking.

BOINC The Berkley Open Infrastructure for Network Computing is a free middleware system for high throughput computing (HTC) which is available as open source. BOINC is designed for heterogeneous systems with distributed memory. It was developed at the University of California in 2002 and is funded by the National Science Foundation. The BOINC framework supports both NVIDIA and ATI graphic cards. However, the GPU support is available in terms of API functions which developers can use in their applications to identify the presence of GPUs in each BOINC node. That means that the GPU application itself must still be programmed manually, by making use of one of the GPGPU APIs. BOINC documentation is available but is very unstructured.

PVM The Parallel Virtual Machine (PVM) is a software tool for distributed computing in heterogeneous clusters. It is developed at the University of Tennessee and released as open source. It has been compiled for the Unix OS as well as the Windows OS. It supports systems with local and distributed memory. PVM libraries are available for C, C++ and Fortran and provide broadcast and multicast routines. The online documentation seems scattered and not frequently updated. Implementations of PVM with GPU support could not be found.

Unified Parallel C Unified Parallel C (UPC) is designed for high performance computing on parallel machines. It is an extension of the C programming language and is developed by UC Berkley. It is suitable for distributed systems with both shared and non-shared memory. The software is regularly updated and documented. UPC targets mostly large-scale scientific projects and a project with UPC has even won a high-performance com-

puting (HPC) competition[4]. There are articles on the developer's website which mention integration of UPC with CUDA and OpenCL. Yet, the information is limited and vague.

Condor Condor is a high-throughput computing software framework for coarse-grained parallelism of computationally intensive tasks. It is released as open source and supports multiple operating systems - Unix, Linux, Mac OS, FreeBSD and Windows. Condor is developed at the University of Wisconsin-Madison and is mainly used in academic environments and supercomputing. Condor can be used on both dedicated or idle machines (known as "cycle scavenging"). For example, NASA has a cluster with 350 Sun and SGI workstations in an office environment. Condor's website states that there is support for GPUs (CUDA only) but it has not been updated since 2009 and seems obsolete.

2.5 Summary

This chapter laid the foundation knowledge for passwords, password cracking and distributed computing. The first section explained how passwords can be stored in non-clear text using hashing functions, how password strength can be expressed through means of Shannon's definition of entropy and how a hash value can be attacked to retrieve the original password.

The second section looked into the internals of a GPU unit and its parallel computations design. Additionally, the GPU was compared with a CPU. Their processor architectures were described and the computational speed advantage of GPUs was admitted.

Further on, section three looked at the theory on distributed computing. This pinpointed common bottlenecks in building distributed clusters and will help us define criteria for evaluating available distributed architectures which can be found in section 3.2.2. Finally, the chapter gave a brief introduction on several popular distributed frameworks. They all allow process distribution over an Ethernet network. Nonetheless, they cannot target a GPU unit unless it is capable of performing *general computing*, altering the denomination of a GPU to a GPGPU.

The following chapter will research into how a framework can be integrated with such a GPGPU and, based on that, propose an efficient implementation architecture for building a GPU-enabled password cracking cluster.

[4] http://www.hpcchallenge.org/

Chapter 3

Research

In order to harness the computational power of GPUs for password cracking, the most efficient password cracking method must be identified. In the previous section we discussed the characteristics of a graphic processing unit and how it can be used for cryptographic hash functions which will help us make an informed choice for the most efficient method for password cracking in the following subsection. In the second subsection, we will identify possible methods and tools for distributing the password cracking method over a cluster of GPUs (in combinations with both shared and non-shared memory).

3.1 Password cracking methods in a distributed GPU environment

In section 2.1.4 the available attack types are mentioned. In section 2.2.1 we explained the parallel computing design of the GPU. It has a lot of processor cores (about 480 for a NVIDIA GTX295[1]) with cores having small shared memory of 16 KB ([7], page 43). Having all that in mind, we can assume that not every attack type is feasible for a GPU environment.

To select the "best" attack method for a distributed GPU solution we need to define this first. Password cracking is a time-expensive procedure with the trade-off to the success rate. Therefore, applying faster hardware and a priori information helps here.

Due to the fact that GPUs have little cache memories, it is impossible to store big datasets in there, hence being reliable on slow storage elsewhere. To define the "best" attack method in the speculated environment, I/O intensive depending tasks will not be faster if we introduce the usage of a GPU, as communication over various GPU and CPU buses introduce significant delay.

If we take the bottlenecks of each attack type into account, one notices that only computationally intensive methods will be feasible in this setting.

To fully use the power of a GPU, the best available attack type will be brute force attacks. Dictionary attacks are considered to be possible on GPUs but have less net profit.

3.2 Distributed architectures for GPU password cracking

We already have shown that GPUs speed up the the solving of computationally intensive problems. We now take a look at scaling that GPU power in distributed environments. Current motherboards are relative low in price but are limited to the amount of graphic cards they can fit. Apart from residing on a common motherboard, GPUs can be controlled via host CPUs, clustered in an ethernet network. However, a system with distributed memory, unlike the RAM memory on motherboard, requires different criteria for best computational efficiency and high data throughput. In the following subsection we present the approach, we took during this project, in defining the scope for researched architectures and software frameworks. In subsection 3.2.2 we define what a good distributed architecture is. Right after that, we present our findings.

3.2.1 Approach and scope

We will tackle the criteria required for combining GPU power with a distributed system. After those criteria are listed and defined, an evaluation will be done over the available architectures. Their list grows highly when we include distributed architectures, current distributed software frameworks and tools, or combinations of those. Therefore, we distinguish three general approaches for building a GPU password cracking cluster.

- process distribution for CPUs and GPUs is implemented by the software framework
- process distribution can be implemented by a company engineer using the distributed architecture (or specification) with combination of GPU APIs
- current tools for password cracking which support GPUs

Due to time-wise project constraints we cannot look into all architectures currently in the market. Added, our project objectives are to research the general approaches for building a distributed GPU environment for password cracking. Therefore, we will select one particular distributed software product for each

of the enumerated implementation approaches. That selection will not only be based on the architecture's popularity (amount of users) and maturity, but also how it practically applies to the current CPU cluster at KPMG, which we presented in section 1. It consists of 30 machines, which are connected through a dedicated LAN network and are running an MPI patch [1] of John The Ripper.

3.2.2 Criteria

Various factors define how a distributed GPU cluster for password cracking can be optimized for efficient performance and ease of management. In this subsection we outline these factors on which we will base our architectures evaluation.

Distributing key space

In order to split the key space we need more context on this. If the brute force attack is performed the key space can be easily be split up and send to client nodes. Because all the possible keys are tried, a linear (but pipelined) iteration can be performed. The key space can be represented by an iteration function that toggles over all the possibilities and an offset can be specified in order to determine where in this linear task a node should start. This yields minimal communication as not every item in the key space needs to be sent separately.

If a dictionary attack is performed, there is no way to represent the key space through a function, as this is language dependent and can only be related to through means of statistics, for example the expectation rate of bigrams, trigrams etc. This information needs to be exchanged and divided before iteration of the key space can begin. Not every node needs to be aware of the entire key space, but looking from an abstract level, the entire key space needs to be divided explicitly over all the nodes. This can be all at once, if it is known a priori that nodes will not leave or join the network. Otherwise, a part of the key space needs to be kept in memory and assigned to joining nodes.

Besides methods of distribution we will also assess the possibilities in which middleware can aid in distributing these workloads to various nodes.

Scheduling and queueing/Distributing the work load

In the previous subsection we described how to split key space. There are two noted environments, namely a static and a dynamic, with regard to the availability of client nodes. In a static environment, like a cluster or LAN, the

[2] http://www.bit-tech.net/hardware/motherboards/2010/12/31/msi-big-bang-marshal-preview/1

[1] http://www.bindshell.net/tools/johntheripper

key space can be split up in a trivial matter. The cluster can be assumed to be homogeneous and no congestion is expected. In such an ideal case, the workload can be scheduled through a round-robin way if all the machines deliver the same performance.

However, the situation described above is not often the case. The cluster might be running on a local LAN, but heterogeneous systems are usually preferred. Due to possible performance differences among the nodes, the workload can not be distributed uniformly. Hence, the computational power of a node needs to be determined by benchmarking or profiling of processors and GPUs in order to achieve good performance in such systems.

The benchmarking can be ran over intervals to check if the system is a) capable of running a specified workload with respect to the time taken by the rest of the nodes b) see if the node is available for work at all. A downside to this approach is that the load on the node will increase for mere testing purposes, which could effectively be used for password cracking.

Profiling could outweigh the previous mentioned downside. When talking about profiling, a system is able to report during query, or through a table created a priori stating the capabilities of a node. This can easily outperform benchmarking, as a single lookup needs to be executed. The downside to this is that the table needs to be updated when new hardware is added and hence can get outdated in which an unknown state is created in the system.

However, if it is possible to retrieve such information, the scheduler can easily distribute the load according to the capabilities or processing power of each member of the cluster.

To optimize the batch size of each job, benchmarks need to be performed as well. Such hardware profiling can ensure that the time it takes to transfer a job to a client node is smaller than the time it takes for the node to compute the job.

Recovery and error handling

It is not a trivial task to detect if a node is down, restoring or merely not reacting by the fact of network congestion. Lets skip the part of detection, and look at this problem the other way around.

How should a node react if it crashes and recovers. A trivial solution to handling such events could be to request the status of the node sending a query and let a general coordinator resent the last job that got transmitted. This does require a state full server at the coordinators end, and also introduces a single point of failure; if the coordinator crashes the whole distributed system would become unavailable.

Another point of interest is integrity and consistency of the data. Redundancy

could be applied in the encoding, going as far as self restoring codes (Hamming, Reed Solomon). This however comes at a computational cost.

A different solution to error control is not to worry about this at all, and let lower level protocols handle such events. TCP is able to detect packet loss and solves this through retransmission. Something which is pretty cheap in a solid local LAN environment.

As we are communicating over an assumed unreliable network, should we worry about errors at all as it is not that trivial in noticing if a node down or not. How do you guarantee that the results are correct and consistent.

Supported hash types and extensibility

The current environment does offer various GPU tools for password cracking. Unfortunately these are often proprietary, unstable or are only dedicated to a single or a limited set of hash types. It is important to look into the extensibility options for development which these tools offer. Their features will have to be adjusted in the future in order to support newly emerging hashing algorithms.

API, Documentation & Support

The architectures which can be used to achieve distributed computing on GPUs all have a different set of API functions that they offer. Some of these functions include reliable interprocess communication, others provide thread synchronization and some other target directly the GPU computational power by incorporating necessary GPU interfaces. We should look into all those supported features for developers because a password cracking cluster would need to be changed periodically. New hash types can be introduced, together with code optimization and administration features. Therefore, it is essential that the deployed software is well maintained by its creators - documentation must be sheer and concise and provide plenty of tutorials. If further development support is needed, the releaser must be available for contact, directly or through a volunteer community.

3.2.3 Distributed architectures evaluation

We discussed the problems seen in general with password cracking and distributed systems and how to tackle them. We will use this knowledge as criteria to formulate an answer to the question what the best distributed architecture, as listed in section 2.4, for GPU password cracking is. Do we need middleware at all, as it has somewhat been proven to be done through a simple C++ server/client model [20].

We outlined three approaches for tackling our defined criteria, which we described respectively in 3.2.1 and 3.2.2. In the following list we have chosen two distributed architectures - BOINC and MPI - which best represent the first two approaches, defined in section 3.2.1. BOINC is the most mature project (since 1995) with great user base (∼300 000) as we mentioned in 3.2.3. MPI is a specification for message passing. It is standardized which ensures that MPI calls on different implementations (such as MPICH, OpenMPI, pyMPI and MPI.NET) would behave the same on any architecture (performance might differ, though)[19]. This leads to great portability of the MPI code. Based on this, we will focus our evaluation on MPI in combination with OpenCL to target our second approach, defined in 3.2.1. We also evaluate some current tools (IGHASHGPU, oclHashCat, Distributed Hash Cracker and ElcomSoft Distributed Password Recovery) for distributed GPU password cracking to cover the third approach.

BOINC

Berkley Open Infrastructure for Network Computing is a middleware system for grid computing. It is released under the GNU LGPL license and was originally developed in 1995 at the Space Sciences Laboratory at the University of California, Berkeley. Its initial goal was to support various research projects[2] mainly through the computers of thousand volunteers in an unreliable network, such as the internet. Therefore, the task of distributing enormous volumes of data periodically and reliably becomes intricate and is known as high throughput computing (HTC). The latest version of BOINC (6.10 since June 2010, according to the release notes [3] introduces support for ATI graphic cards (NVIDIA cards were supported already in 6.6). By employing the raw computational power of the graphic cards, BOINC allows combination of HTC and high performance computing (HPC). This makes BOINC a suitable solution for wide range of arithmetic intensive tasks which can be distributed. However, we should note that BOINC cannot actually "port" CPU-an application written for a GPU. Developers who want to target GPUs for their project must first build a GPU version of the application using the CUDA interface. BOINC can then detect whether clients have a supported graphics card and push the GPU version of the application. Recent ports of BOINC in Java and Android try to target ubiquitous mobile devices as well.

Apart from targeting volunteer user-base, BOINC allows companies to employ their internal infrastructure as well. An organization can set up their own BOINC server [4] which will dispatch jobs to all idle desktop stations in the office. Thus, a dedicated password cracking cluster is not mandatory. By default, BOINC performs an integrity check on job results from public computing re-

[2] http://boinc.berkeley.edu/wiki/Publications_by_BOINC_projects
[3] http://boinc.berkeley.edu/w/?title=Release_Notes&action=history
[4] http://boinc.berkeley.edu/trac/wiki/DesktopGrid

sources. This includes dispatching the same job to multiple clients and then verifying the results (e.g. taking the mean average). Such a check is not performed when BOINC is used with internal resources.

This page [5] lists all software prerequisites for building and running a BOINC server. Figure 3.1 illustrates the internal components of the BOINC architecture which offers developers a wide range of ready-to-use functions.

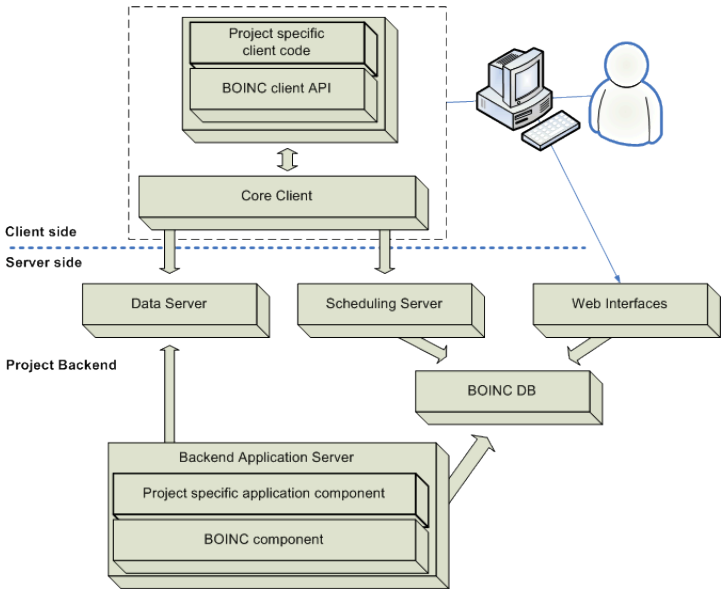

Figure 3.1: BOINC architecture [6]

Main components include:

- *Web backend* which facilitates user account administration and allows administrators to view log results. However, the web site does not allow for project administration which still have to be done manually (in terms of bash scripts).
- *Scheduler server* which takes care of distributing the workload to idle nodes
- *Data server* handling data channels to clients for input and output files
- Relational *database server* to persist processed data and store user accounts

BOINC has a broad API[7] to assist developers in building cross-platform distributed applications:

[5] http://boinc.berkeley.edu/trac/wiki/SoftwarePrereqsUnix
[7] http://boinc.berkeley.edu/trac/wiki/BasicApi

- File system I/O wrappers
- Checkpointing
- Functions for critical operations (which cannot be interrupted or killed)
- Progress status of the performed computation
- Timing information for benchmarking

Apart from all its advantages, a BOINC server for a custom computational project requires time and technical skills to be set up which makes BOINC not an out-of-the-box solution for password cracking. A *BOINC project* consists of a custom application which is spread along client nodes along with an input file (workunit) for processing. When an input file which contains arguments to the aforementioned application is processed, a new one is dispatched by the *scheduler server* and this repeats until all client nodes have processed all workunits. Even though the BOINC API supports functions for managing the work load (workunits) to client nodes, the BOINC API lacks functions for managing individual projects.

MPI

The *Message Passing Interface* (MPI) is a library which allows processes to exchange messages. It targets HPC applications with distributed (non-shared) memory and includes functions for both point-to-point and broadcast (through process grouping) message passing. It is available for all major operating systems and can be used with C, C++ and FORTRAN. MPI is standardized which ensures for code portability across current implementations, which include LAM/MPI (currently OpenMPI) and MPICH. To these we also add many implementations for major programming languages - Python, Perl, Java and .NET. The MPI library is used for parallel programming in computer clusters. When two processes want to exchange data over the network, they must exchange messages because the processes do not share memory. MPI manages this time-consuming operation. Therefore, it is not suitable for thread synchronization with shared memory. Open Multi-Processing (OpenMP) supports fine-grained parallelism and some combinations with MPI currently exist[19]. However, brute force password cracking can be implemented using only coarse-grain parallelism as it is known as "embarrassingly parallel". MPI master would split up the searched key space into partitions and send these to client nodes which will process them independently. An MPI implementation can be compiled together with CUDA and Stream to support one or multiple GPU cards on client nodes [8]. However, the Stream drivers seem still inmature and developers

[8] http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#simpleMultiGPU

are having trouble with the documentation and support from AMD[9].

OpenCL for multiple GPUs

OpenCL is a runtime which also allows to develop parallel applications for GPUs. It is designed in close relation to OpenGL. They are both released by the Kronos group[10]. OpenCL is volunteer-based open project and any organization which wants to contribute can contact the Khronos group to add design ideas or support for its computing products. Support for OpenCL from main video card manufacturers NVIDIA and ATI is growing constantly and has converged to the point that all their new products are currently OpenCL-compatible. OpenCL is also supported on IBM Cell and Field-Programmable Gate Arrays (FPGAs). Additionally, because of its open source nature, integration projects to target new devices, such as Playstation 3[11] and Nintendo Wii, already exist.

The last stable specification of OpenCL (version 1.1 released only 6 months ago) supports copy/move of memory on multiple devices but does not provide thread synchronization. Thus, developers must manually monitor (through the use of events) if shared memory blocks are accessed by other kernels[12]. However, NVIDIA and ATI implementations of OpenCL do not handle multiple GPUs well and developers are still having trouble using OpenCL for targeting multiple devices on one motherboard [13]. Nevertheless, the software is in early stage and this feature is expected in the near future.

CLara is the only project which tries to employ OpenCL for distributed computing over an ethernet network but no stable release is available yet (current version is 0.0.1). Its homepage is unreachable, apart from the source code which can be downloaded from sourceforge.net[14].

3.2.4 Custom tools implementing distributed GPU password cracking

In order to target our third approach (as defined in section 3.2.1) for building a GPU password cracking cluster, we looked at some current software tools for password cracking that are emerging on the commercial market, as well as being released as an open source. Previous research of other UvA SNE students[1] have listed several custom tools available for GPU cracking. The tools which are still available and actively developed are presented in the following list.

[9] http://forums.amd.com/devforum/messageview.cfm?catid=390&threadid=143851&STARTPAGE=2&FTVAR_FORUMVIEWTMP=Linear

[10] http://www.khronos.org/opencl/

[11] http://sites.google.com/site/openclps3/

[12] Kernel - application running on a GPU

[13] http://forums.amd.com/devforum/messageview.cfm?catid=390&threadid=129284&STARTPAGE=2&FTVAR_FORUMVIEWTMP=Linear

[14] http://sourceforge.net/projects/clara/

IGHASHGPU[15] is a tool able to use GPU devices for cracking a variety of hash types, under which MD5, SHA1 and Oracle 11g. The developer claims that it is the fastest MD5 and SHA1 hash cracker available. As far as MD5, this is supported by the benchmarks done by the other UvA SNE students.

The tool is free for private usage, but is still closed source which limits the extension of other hash types. Also, the tool does not offer a solution for it to be used in a distributed environment. This necessarily does not have to be a problem, as it is possible to pass arguments to the program which controls which key space should be iterated (lower and upper bounds).

oclHashCat[16] is the OpenCL version of HashCat. HashCat is a closed source though free tool which is able to coordinate up to 16 GPUs for cracking various password types.

There is no support for running this tool in a distributed environment. Also, key space control seems to be more difficult if compared to IGHASHGPU.

Zonenbergs Distributed Hash Cracker(DHC)[20] is an academic project for MD5 hash cracking which incorporates both GPU processing, as well as a possibility to split up the problem over a distributed environment.

To achieve GPU acceleration CUDA is invoked. As for the distributed system there is a centralized server model applied. Communication is done through sockets. Additional hashtypes are supported by having the algorithms available through modules.

The project itself is open source, unfortunately the source code is nowhere to be found.

ElcomSoft Distributed Password Recovery A commercial product which offer the possibilities to crack hash types, as well as specific application support for recovering password in a distributed CPU and GPU environment. The developers claim that the product scales linearly up to 10.000 nodes without performance drops. On a single machine, up to 64 CPUs are supported and 32 GPUs are supported.

A downside to this product are the high purchase price, which increases if more nodes are required for the cluster - €599 (max 20 clients) up to €4.999 (max 2500 clients). Also, this is proprietary software which does not allow alternation or improvement of the product (e.g. optimization and support for additional hash types).

The variety of GPU-enabled password cracking tools is not huge. IGHASHGPU, oclHashCat and ElcomSoft Distributed Password Recovery support multiple hash types, IGHASHGPU and oclHashCat are free but do not offer source code and ElcomSoft Distributed Password Recovery supports distributed GPUs over ethernet networks. Only ElcomSoft Distributed Password Recovery combines distributed GPU support for multiple hash type password cracking but it is not

free, nor open source because it is proprietary. Its license is not cheap and the lack of source code prevents customers from extending its functionality.

3.2.5 Results

During the comparison we found out that there is no single solution for achieving GPU accelerated password cracking in a distributed manner. This is due to the fact that efforts are currently focused on maximum performance from a single machine.

Hence, we need to look further into ways of integrating several frameworks into one application, able to crack an unlimited number of hash types in a distributed environment. To achieve this multiple paths are possible.

On top of the hardware level of GPUs a layer of communication is needed. Here the APIs play their role. With these hooks single machine GPU applications are created. However, these are not distributed yet and require different hooks. By going another level up the distribution can be implemented through means of middleware.

Figure 3.2 shows the possible integration paths of various technologies mentioned in this report and also shows how they relate to each other. With this information the elements can be combined in order to achieve distributed GPU cracking. Outliers in this case are Zonenbergs DHC and Elcomsoft, as it seems that they are directly tied to the hardware level. This is because of uncertainty in the applications architecture due to respectively the lack of available source code and proprietary nature.

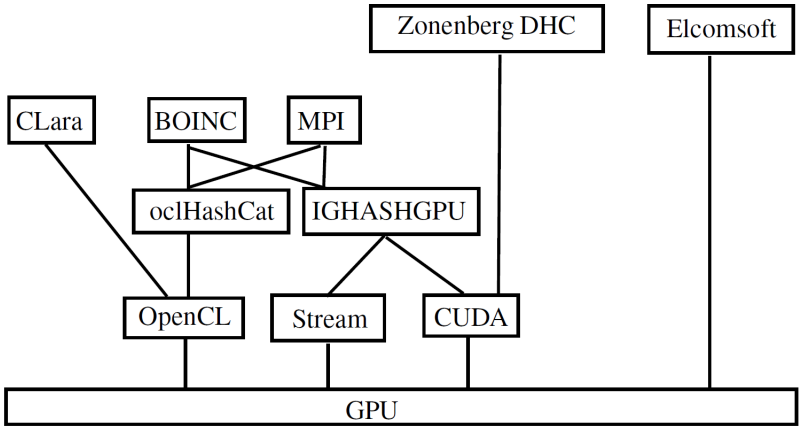

Figure 3.2: Flow chart of possible solutions

The following subchapters present our evaluation on the researched frameworks and tools based on the research criteria, which was defined in subsection 3.2.2. That is also summarized in table 3.1 through a grading from 0 (--) to 5 (++). The grouped columns represent the three implementation approaches which were defined in 3.2.1. Namely, BOINC, MPI and CLara support distribution but not GPU. IGHASHGPU and oclHashCat supports GPU processing, however

not distribution and are built for single machines only. The last group, which supports GPU as well as distribution consists of Zonenberg's Distributed Hash Cracker and Elcomsoft's Distributed Password Recovery.

	BOINC	MPI	CLara	IGHASH GPU	oclHash Cat	DHC	Elcomsoft
Distributing key space	+	-	-	--	--	++	++
GPU support	+	-	-	++	++	++	++
Recovery & error handling	+	+/-	+	--	--	?	?
Different hash types (extensible)	C	C	C	+	+	--	+/-
API, Documentation & support	--	+	--	-	-	--	+/-
KPMG cluster	+	++	+	+	+	+	--

Table 3.1: C = custom application development required ; ? = unknown

Distributing key space

From the current tools, the ones which support single machines score lowest due to their non distributive nature. The all-in-one solutions are graded highest as they comply to distributing the problem in theory, as well as distributing it through the practical networking side.

From the middleware frameworks, BOINC scores positive because it takes care of reliably dispatching jobs (workunits) to client nodes and processing the computational results. However, developers must still program their own password cracking applications and use the CUDA/Stream wrapper interfaces to target GPUs.

GPU support

BOINC claimed to have GPU support. However, this is to the sense of GPU support on the middleware layer. It allows for selecting GPU capable machines for a task or project. With the other middleware systems this is not available.

The single machine tools score highest as this is their main focus, as well as the all in one solutions.

Recovery & error handling

With the middleware solution error handling is handled by the middleware itself, like with BOINC which offers some monitoring capabilities, or through means of a developer who has to implement this themselves. As recovery over the distributed is not applicable to the single machine tools they score lowest. It is unknown how this criteria is handled because of unclear documentation and proprietary properties.

Supported hash types and extensibility

Hash types are not offered by the middleware layers and need to be coded by hand. The single machine tools offer a fairly good range of hashtypes and hence score highest. DHC only implemented MD5, however did give the option to include other hash types. Elcomsoft is a proprietary solution and hence can not be extended. It does offer a good amount of hashtypes and interoperability with well know products such as Microsoft Office, PGP implementations and Adobe Acrobat PDF.

API, Documentation & support

BOINC is released as an open source software which allows developers to tweak into its workings when needed.

BOINC project administration turns out difficult. Hence, security auditors have to navigate through the graphical user interface (GUI) of the *BOINC Manager* application in order to manually add a new project for each password which they want to crack. It is possible to automate this process by executing long bash scripts but this prompts for wide set of technical skills and makes projects difficult to manage. The BOINC documentation is also not very clear about that. There is a great load of BOINC documentation available but it looks very unstructured and hectic (separate domain names, dead links and pages and contributed and hosts by users). Added, the interprocess communication of the KPMG cluster is based on MPI which cannot be integrated with a BOINC project. All this makes us conclude that BOINC is not a viable solution.

MPI is the most standardized one and hence offers general support through the community where a lot of documentation is available.

The single machine solutions offer no more than just a manual page on how to use the application, not the inner workings. Zonenbergs DHC documentation is only retrievable from the article written about it. The source code is not

available and the website is no longer accessible. Elcomsoft is a commercial product which does come with some documentation on its usage. Once again, due to its proprietary nature they do not specify architectural properties.

KPMG cluster

With regards to the available environment an incorporation with MPI would be fairly easy, as this is already available on site. The other tools could be worked with by means of scripting and programming. Due to the fact that Elcomsoft is proprietary it is graded lowest.

3.3 Summary

In this chapter we identified that brute force cracking is the most feasible way for cracking passwords on a GPU. Compared to the other attack types (section 2.1.4), it avoids the bottlenecks from I/O operations and small shared GPU memory.

After that, we covered that there are a lot of possibilities when it comes down to matching GPU applications, interfaces and distribution middleware. We summarized the number of possibilities by distinguishing three major approaches to building a GPU password cracking cluster, namely 1) by developing a new application for GPU password cracking on 1 host using the GPGPU APIs together with an existing distributed middleware which distributes the application to network hosts 2) by developing not only a new application for GPU password cracking using GPGPU APIs, but also developing the distributed features and 3) by using available software applications for password cracking.

These groups were then identified with current frameworks and tools from section 2.4. The research showed that no current tools support distributed password cracking or can be extended to support multiple hash types. Further on, the research had to evaluate which of the other two approaches is more feasible. Due to time-constraints, we chose one available distributed framework to represent a solution for each of the two approaches. This allowed us to perform a comparison based upon the criteria that was defined in section 3.2.2.

The results of the comparison are graded on a scale from `--` to `++`, after which each criteria is separately discussed. The research group recommended the second approach as most feasible for building a distributed GPU password cracking cluster which can support wide variety of hash types. The presented technologies - MPI (for distribution) and OpenCL (for GPU interaction) - provide ease of development and high password-cracking efficiency.

Chapter 4

Conclusion

In chapter 1 we formulated the research questions. In chapter 2 background theory was discussed. In this chapter we drew conclusions based on the theory as well as our research, which was presented in chapter 3.

4.1 Password cracking on GPU

Efficient password cracking is a trade-off between success rate and speed. In section 2.1.4 we described the possible types of password cracking. In section 2.2.1 we have shown that GPUs are most efficient at arithmetically intensive tasks that require little data input. This is because of limited bandwidth available to the memory unit on a GPU. Thus, we concluded that brute force is the most feasible approach towards GPU password cracking. It is an "embarrassingly parallel" problem which makes it suitable for the GPU multi-core design. Fine-grained parallelism (thread-level synchronization) is redundant in the scenario of bruteforce cracking, which simplifies implementations for password cracking GPU clusters.

4.2 Distributed implementations for GPU password cracking

Section 3.2.4 presented current tools that support GPU password cracking. With their help, companies can set up a GPU password cracking cluster with minimal efforts. However, during the research it was discovered that only *Elcom-Soft Distributed Password Recovery* and *Zonenbergs Distributed Hash Cracker* supports password cracking in a distributed cluster. It supports one hash type by default, but is extendible with modules. However, development has come to

a total halt some time ago. ElcomSoft supports a wide range of hash types but since it is released as a closed source, it prevents developers to extend the list of supported hash types. This makes it not a suitable solution for the long term.

Section 3.2.5 proves that using current available password cracking applications alone is not sufficient for building a distributed password cracking cluster. This implies the need of a custom-developed application which can be built on top of a middleware. In section 3.2.1 we defined the approaches which developers can take in order to build a password cracking cluster.

The BOINC framework was presented to target the first approach. It is a mature distributed software framework that provides GPU interfaces for BOINC developers and has the biggest user base with more than 300 000 active volunteers. However, section 3.2.3 points out that its documentation is chaotic, making the development of a BOINC project a time consuming process. Additionally, it cannot support MPI in our bruteforce password cracking scenario, which prevents its integration with the current John the Ripper cluster that is based upon MPI.

We consider the combination of MPI with OpenCL to be the best practical approach for setting up a GPU cluster for password recovery. The MPI specification (and its framework implementations) provides functions for coarse-grained parallelism which is exactly what brute force password cracking requires. Developers using MPI are left with the flexibility to add application functionality for cracking passwords of various hash types and invoke OpenCL functions to target GPUs. Bearing in mind that password cracking functionality is already available at KPMG's cluster by John The Ripper, OpenCL can be combined with the current MPI patch of John The Ripper to achieve distributed GPU password cracking. Password types which are currently not supported by John The Ripper will have to be manually implemented by the developer. Yet, this is the most feasible approach which would require least time to implement.

The added value of using OpenCL instead of using the rest of the GPGPU APIs, which are listed in subsection 2.2.3, is that with OpenCL developers can target not only NVIDIA or ATI cards independently through one single interface, but it also offers support for IBM Cell and FPGA processors. Furthermore, OpenCL supports multiple GPU cards on one host machine, just like CUDA and Stream. The available high-level programming interface in C allows for integration with the source code of John The Ripper. A sample approach for an implementation might look as following:

- Administrators will communicate with the MPI master (coordinator) node through a web-interface in order to place their cracking tasks (upload password hashes for cracking).
- The autonomous cluster coordinator first contacts each client to determine if it is idle or still busy with cracking a password.

- The coordinator contacts the free nodes again for profiling. It will try to estimate the computing capabilities of each client machine. It is expressed by the amount of passwords which a node can brute-force in a certain time frame, making use of its on-board CPU and GPU processor cores, together with the node's network latency.
- The coordinator splits up the key space of possible combinations for each given password based on the nodes' measured performance. This allows a balance of the workload based upon the cracking power available at each node, not based upon the amount of nodes. Additionally, a part of the key space is reserved for nodes which unexpectedly finish earlier than others, or new nodes join the network after finishing a cracking task. Each set of combinations which a node must exhaust is called a *job*.
- The coordinator dispatches jobs to their destined client nodes and awaits for a response (job completed, but did, or did not find a match). When a password is successfully guessed by a client, the coordinator registers the results and stores them for later review. Until then it keeps waiting for new job requests. Network nodes will finish working on their jobs at random times and keeping them busy with new jobs is the key to GPU usage optimization. As mentioned earlier in subsection 3.2.2 this is facilitated by *profiling*.
- Once a job is received on each machine, the local scheduler will further split up the password job into input streams for the active thread warps, as section 2.2.2 explained. This key space split is based on hardware profiling when multiple GPU cards are available on-board, so that faster cards would get more time-consuming thread tasks. The communication with the actual GPU cores is realised by using the OpenCL GPU API.

4.3 Future work

It has to be noted that if at some point in time John The Ripper is considered obsolete and a new GPU password cracking application needs to be developed from scratch, CLara seems to be a promising solution. It combines OpenCL functions for GPU interfaces, and functions for process communication in Ethernet/IP networks which minimizes the efforts in building a GPU cluster. Added, the use of OpenCL for heterogeneous systems is incomparable to any other framework or GPGPU API to target ubiquitous client devices. This makes it a feasible solution for a large-scale password cracking GPU cluster. However, we need to stress that, due to time constraints we could not dive into CLara's source code and our judgement is based on our online research.

Chapter 5

Acknowledgements

We would like to thank our supervisors for giving us the right insights and helping us finish the project. Additionally, we appreciate the help of Marcus Bakker and Martijn Sprengers who helped us grasp in the foundations of password cracking and GPU programming through their reports and numerous open discussions. The report[14] of Martijn, which is unpublished at this moment, was valuable to understanding CUDA programming and GPU internals in the context of parallel computing.

Bibliography

[1] Bakker, M., and van der Jagt, R. GPU-based password cracking. 2010. Cited on pages 6, 16, 23, and 31.

[2] Burr, W. E., of Standards, N. I., and (U.S.), T. *Electronic authentication guideline [electronic resource] : recommendations of the National Institute of Standards and Technology / William E. Burr, Donna F. Dodson, W. Timothy Polk*, version 1.0.2. ed. U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, Gaithersburg, MD :, 2006. Cited on pages 10 and 11.

[3] Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2004), SC '04, IEEE Computer Society, pp. 47–. Cited on page 14.

[4] Flynn, M. J. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on C-21*, 9 (09 1972), 948 –960. Cited on page 15.

[5] Kaufman, C., Perlman, R., and Spenciner, M. *Network Security: PRIVATE Communication in a PUBLIC World*, 2nd ed. Prenctice Hall, 2002. ISBN-10: 0130460192. Cited on page 9.

[6] Kindratenko, V., Enos, J., Shi, G., Showerman, M., Arnold, G., Stone, J., Phillips, J., and Hwu, W. GPU clusters for high-performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on* (2009), IEEE, pp. 1–8. Cited on page 18.

[7] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE 28*, 2 (4 2008), 39 –55. Cited on pages 13, 15, and 23.

[8] Luebke, D., and Humphreys, G. How GPUs work. *Computer 40* (2007), 96–100. Cited on page 13.

[9] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008. Cited on page 16.

[10] NVIDIA. *Whitepaper - NVIDIA´s Next Generation CUDA Compute Architecture: Fermi*. NVIDIA, 2009. Cited on page 13.

[11] Oechslin, P. Making a faster cryptanalytic time-memory trade-off. *Advances in Cryptology-CRYPTO 2003* (2003), 617–630. Cited on page 12.

[12] Rotem-Gal-Oz, A. Fallacies of distributed computing explained. *URL http://www.rgoarchitects.com/Files/fallacies. pdf* (2006). Cited on page 20.

[13] Shannon, C. Prediction and entropy of printed English. *Bell System Technical Journal 30*, 1 (1951), 50–64. Cited on page 10.

[14] Sprengers, M. On the Security of Password Hashing Schemes regarding Advances in Graphic Processing Units. 2011. Cited on pages 6 and 41.

[15] Tanenbaum, A. S. *Computer Networks, 4th Edition*. Prentice Hall, 2002. Cited on page 18.

[16] Trybulec, W. Pigeon hole principle. *Journal of Formalized Mathematics 2* (1990). Cited on page 9.

[17] Wikipedia. Amd firestream — wikipedia, the free encyclopedia, 2010. [Online; accessed 29-January-2011]. Cited on page 17.

[18] Wikipedia. Distributed computing — wikipedia, the free encyclopedia, 2011. [Online; accessed 21-January-2011]. Cited on page 19.

[19] Yang, C.-T., Huang, C.-L., and Lin, C.-F. Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. *Computer Physics Communications 182*, 1 (2011), 266 – 269. Computer Physics Communications Special Edition for Conference on Computational Physics Kaohsiung, Taiwan, Dec 15-19, 2009. Cited on pages 28 and 30.

[20] Zonenberg, A. Distributed hash cracker: A cross-platform gpu-accelerated password recovery system. *Rensselaer Polytechnic Institute* (2009). Cited on pages 27 and 32.