Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOXÍAS DA INFORMACIÓN

Paralelización en CUDA de varios modos de ataque a algoritmos de hashing

Estudante: Pedro Fernández-Arruti Gallego
Dirección: Diego Andrade Canosa
Dirección: Basilio Fraguela Rodríguez

A Coruña, agosto de 2020.

A mis padres y a mi hermano.

Agradecimientos

A mis padres por apoyarme siempre y darme la oportunidad de estudiar lo que me gusta. Además agradecer que hayan leído este trabajo varias veces sin entender nada. A mi hermano por ser lo más importante que tengo. A Rubén y Pablo por ser lo mejor que me ha dado Coruña y estar ahí siempre. Y por último a Alex, Alicia, Brais, Carlos, Gabri, Martina, Mencía, Rita, Sara y Uxía por ser los de siempre, animarme cuando lo necesito y hacerme ver lo que valgo.

Resumen

Los algoritmos de hashing se utilizan habitualmente en el ámbito de la ciberseguridad para encriptar información secreta, por ejemplo, se usan para la encriptación de todo tipo de contraseñas. Por este motivo, los atacantes de sistemas informáticos suelen idear y poner en práctica medios para desencriptar códigos hash. En todos ellos es necesario utilizar recursos computacionales de altas prestaciones, pues una forma habitual de ataque consiste en encriptar varias contraseñas candidatas obteniendo su código hash correspondiente, para compararlo con el código hash objeto del ataque. Hashcat es una herramienta de código abierto que implementa una gran variedad de estos ataques. Para hacer un uso eficiente de los recursos computacionales disponibles en una plataforma dada, Hashcat utiliza principalmente la plataforma de paralelización OpenCL. Esta plataforma es un estándar que permite explotar las capacidades paralelas de todo tipo de procesadores multinúcleo y de tarjetas gráficas. Sin embargo, el soporte de OpenCL por parte de muchos candidatos ha decaído en los últimos años, siendo en algunos casos inexistente y en otros deficiente.

Esta falta de soporte de OpenCL es especialmente relevante en las tarjetas gráficas Nvidia, donde el propio fabricante ofrece una alternativa propietaria llamada CUDA. En este trabajo exploraremos las posibilidades de reemplazar el uso de OpenCL por el uso de CUDA en hashcat. Estudiaremos hasta qué punto podemos mejorar el rendimiento de la herramienta utilizando CUDA. Para ello deberemos realizar tres tareas fundamentales: (1) Una tarea de ingeniería inversa para entender cómo funciona Hashcat internamente y el grado de soporte que existe para la tecnología CUDA. Esta tarea es necesaria debido a la escasa documentación que la aplicación aporta para sus desarrolladores. (2) Una tarea de análisis de rendimiento de la herramienta Hashcat, y (3) efectuar una propuesta de mejora del soporte de CUDA dentro de la aplicación, estudiando el efecto que esta mejora puede tener en el rendimiento.

Abstract

Hashing algorithms are commonly used in the field of cybersecurity to encrypt secret information, for example, they are used to encrypt all kinds of passwords. For this reason, computer system attackers often devise and implement means to decrypt hash codes. In all of them it is necessary to use high-performance computing resources, since a common form of attack is to encrypt several candidate passwords obtaining their corresponding hash code, to compare it with the hash code that is the object of the attack. Hashcat is an open source tool that implements a wide variety of these attacks. To make efficient use of the computational resources available on a given platform, Hashcat primarily uses the OpenCL parallelization

platform. This platform is a standard that allows you to exploit the parallel capabilities of all types of multi-core processors and graphics cards. However, the support of OpenCL by many candidates has declined in recent years, being in some cases non-existent and in others poor.

This lack of OpenCL support is especially relevant on Nvidia graphics cards, where the manufacturer itself offers a proprietary alternative called CUDA. In this work we will explore the possibilities of replacing the use of OpenCL with the use of CUDA in hashcat. We will study how we can improve the performance of the tool using CUDA. For this we will have to carry out three fundamental tasks: (1) A reverse engineering task to understand how Hashcat works internally and the degree of support that exists for CUDA technology. This task is necessary due to the scarce documentation that the application provides for its developers. (2) A performance analysis task of the Hashcat tool, and (3) make a proposal to improve CUDA support within the application, studying the effect that this improvement may have on performance.

Palabras clave:

- CPU
- GPU
- Hash
- CUDA
- OpenCL

Keywords:

- CPU
- GPU
- Hash
- CUDA
- OpenCL

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introdución

La ciberseguridad es un elemento clave de la sociedad de la información. Dentro de ella, el almacenamiento seguro de contraseñas o la comprobación de la integridad de distintos ficheros son muy importantes. La generación de códigos hash aborda el problema de cómo comunicar contraseñas y almacenarlas de forma segura y cifrada. Sin embargo, existe una amplia variedad de herramientas y ataques enfocados a obtener la contraseña asociada a un código hash. Una de estas herramientas es Hashcat, que implementa un gran número de ataques a los códigos hash más conocidos. Esta aplicación utiliza técnicas de computación de altas prestaciones (HPC) para realizar este proceso de forma mucho más rápida. El uso de estas técnicas, que explotan las capacidades de las arquitecturas modernas, ponen en jaque la seguridad de los métodos de hashing. En Hashcat, hace años se adoptó de forma universal OpenCL para la paralelización tanto sobre procesadores multinúcleo como sobre tarjeta gráficas. Sin embargo, en los últimos años, el soporte de OpenCL en ciertas plataformas como las tarjetas gráficas de Nvidia es cada vez peor, en favor de la plataforma propietaria del fabricante CUDA. Este trabajo trata de responder a la pregunta de qué ganancia del rendimiento se puede obtener utilizando CUDA en la aplicación de hashing Hashcat en vez de OpenCL.

1.1 Objetivos

En esta breve sección se enumeran uno a uno los objetivos principales de este trabajo:

- Estudiar a fondo cómo funciona la herramienta Hashcat
- Evaluar el rendimiento de OpenCL en distintas ejecuciones de la herramienta
- Aprender y entender el funcionamiento de CUDA
- Estudiar la forma en la que implementar paralelización con CUDA de la forma más eficiente posible

- Efectuar una propuesta de mejora del soporte de CUDA en Hashcat
- Realizar un análisis de rendimiento en el que se compare nuestra implementación frente a las que ofrece Hashcat en su repositorio oficial

1.2 Organización del proyecto

Para la realización de la memoria de este proyecto se divide el trabajo en 6 capítulos:

Capítulo 1: En este capítulo se hace una breve introducción al trabajo y se explican los objetivos del mismo.

Capítulo 2: En este capítulo hay una primera sección en la que se explican una serie de conceptos necesarios para comprender el trabajo, y un segundo apartado en el que se realiza un estudio de alternativas, sobre el que nos basamos para tomar una serie de decisiones a lo largo del proyecto. Por último, hay una sección dedicada al aprendizaje de CUDA.

Capítulo 3: En este capítulo se explica, tanto la metodología utilizada a lo largo de todo el proyecto, como la planificación del mismo, detallando esta última, fase a fase.

Capítulo 4: En este capítulo se explica la parte de desarrollo del proyecto. En primer lugar, lo que se hace es explicar el estudio de la aplicación Hashcat, en el que se ve a fondo cómo funciona la herramienta por dentro. Además, se dedica una sección a la implementación de mejoras en la paralelización de CUDA.

Capítulo 5: En este capítulo se explican todos los experimentos y pruebas de rendimiento realizadas a lo largo de todo el proyecto. En estas que se van comprobando posibles mejoras. Los datos obtenidos en este capítulo son los que se nombran en el capítulo anterior y se utilizan en un futuro para llegar a una conclusión.

Capítulo 6: En este capítulo se realizan las conclusiones finales de este proyecto, ya que se corresponde con el último capítulo de la memoria.

Capítulo 2

Fundamentos teóricos y tecnológicos

Este capítulo se utiliza para exponer los fundamentos teóricos y tecnológicos que se usan a lo largo de todo el proyecto. Esto implica explicar una serie de conceptos importantes, describir el estudio de alternativas realizado y finalmente nombrar la opción que se utiliza en el desarrollo del trabajo, resultante del estudio anterior.

En primer lugar, en la sección 2.1, se explican una serie de conceptos clave que son necesarios para entender este trabajo. Posteriormente, en la sección 2.2 se presenta un estudio de alternativas llevado a cabo en determinados puntos del proyecto, con el objetivo de tomar alguna decisión teniendo en cuenta tanto las ventajas como los inconvenientes de cada opción. Una vez explicadas las distintas alternativas disponibles junto con sus pros y contras, se indica cuál es la opción utilizada finalmente en el desarrollo del trabajo siendo esta escogida en base a las características explicadas anteriormente. Por último, en la sección 2.3, se describe la plataforma CUDA, la cual se usa en este trabajo como alternativa a OpenCL. Además, también se realiza una descripción a fondo de las funciones y distintas variables que nos ofrece esta tecnología para utilizar paralelismo en su código, y por lo tanto mejorar el rendimiento de cada ejecución.

2.1 Conceptos previos

A continuación, se explican varios términos esenciales utilizados a lo largo de toda la memoria a fin de facilitar la compresión de la misma.

2.1.1 CPU

La CPU es la unidad central de procesamiento (por sus siglas en inglés CPU - Central Processing Unit). Esta, forma parte del hardware de un equipo y es la encargada de procesar

todas las instrucciones que se ejecutan en dicho equipo. La CPU trabaja con las peticiones del sistema operativo y a su vez con las instrucciones de todas las aplicaciones que se ejecuten en el dispositivo en el que esté instalada. "Se encarga de procesar todas las instrucciones del dispositivo, leyendo las órdenes y requisitos del sistema operativo, así como las instrucciones de cada uno de los componentes y las aplicaciones." [1].

2.1.2 GPU

Por otro lado también existe la GPU, que es la unidad de procesamiento gráfico (por sus siglas en inglés GPU - Graphics Processing Unit). La función principal de este dispositivo hardware es procesar las operaciones gráficas y en punto flotante. Si comparamos su estructura con la de la CPU podemos ver que tiene muchas más unidades de procesamiento, pero más simples [2], con lo que las GPUs son capaces de llevar a cabo un número mucho mayor de tareas simples en determinado tiempo en comparación con lo que haría la CPU. Gracias al uso de la GPU se consigue reducir la carga de trabajo de la CPU, puesto que la libera de parte del trabajo de procesamiento. En la figura 2.1 podemos ver de forma muy general la diferencia entre la estructura de la CPU y de la GPU.

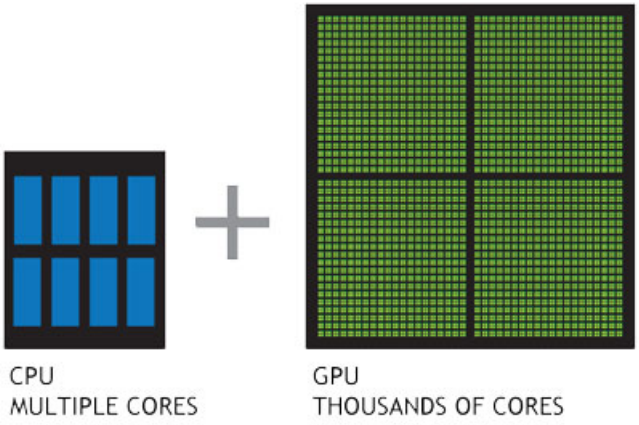


Figura 2.1: Diferencias de la CPU frente a la GPU.

2.1.3 Hash

Según la página oficial de Kaspersky un hash es una función criptográfica, es decir, es un algoritmo que transforma cualquier bloque de datos en una nueva serie de caracteres con una longitud fija [3]. Este es un término muy importante en este trabajo debido a que, como se explica más adelante, Hashcat es una herramienta que es capaz de recuperar contraseñas a través de su hash. Para entender de una forma muy sencilla lo que es una función hash utilizamos la figura 2.2, en la que se observa que para entradas distintas y de distinto tamaño si

aplicamos el mismo algoritmo de hash obtenemos 3 cadenas distintas pero del mismo tamaño. Una característica muy importante de las funciones de hash es que tienen como resultado un conjunto de caracteres de longitud fija, es decir, siempre que se aplique la misma función de hash a cualquier conjunto de caracteres va a dar como resultado una cadena de la misma longitud. Esto tiene varias razones, entre ellas nos encontramos con que los hashes normalmente codifican una entrada muy grande, por lo que es útil para trabajar con ellos que su resultado sea de una longitud menor. Esto último hace que, en muchos casos, se lleve a cabo una compresión con pérdidas, lo que imposibilita el hecho de reconstruir la entrada a partir del hash obtenido. Además, al tener una longitud fija lo que ocurre es que todas las operaciones que se realicen están trabajando con datos del mismo tamaño, por lo que esto hace que los algoritmos de hash se optimicen al máximo para ese número concreto de caracteres.

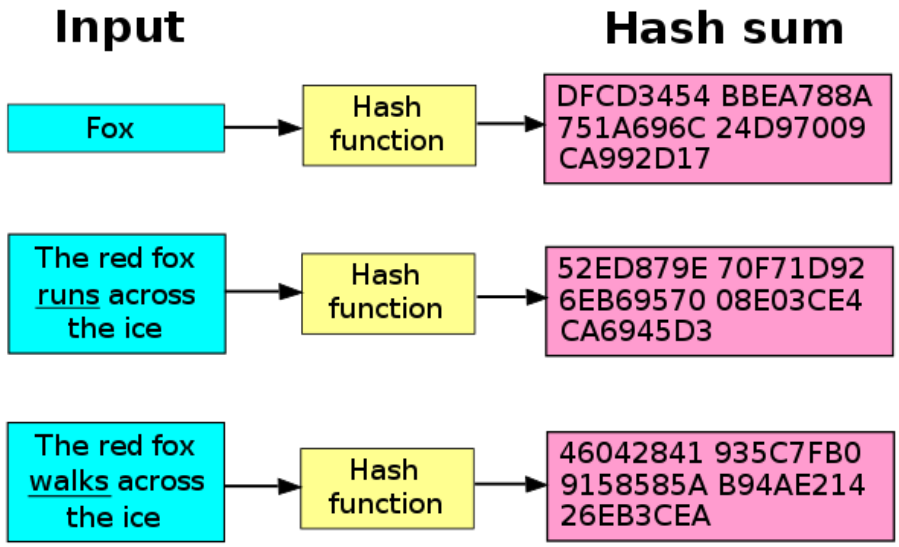


Figura 2.2: Esquema hash.

Entre las múltiples funcionalidades de los algoritmos de hash podemos destacar algunas de ellas. En primer lugar, la más utilizada consiste en la gestión de contraseñas e identificadores. Lo que se hace en un sistema para lograr una mayor seguridad es no guardar la contraseña de los distintos usuarios directamente en él, sino que se guarda el hash de cada una de ellas. Esto es así debido a que, cuando un usuario trata de acceder al sistema se calcula la función hash de los caracteres introducidos por este, en el campo de la contraseña, y se compara dicho hash con el que tiene guardado el sistema. En el caso de que ambos coincidan la autenticación del usuario es correcta. Esto hace incrementar la seguridad del sistema. Hacer esto de esta forma tiene un motivo, y es que calcular un hash de una serie de caracteres supone una operación rápida y sencilla, pero para realizar el proceso inverso necesitamos llevar a cabo una serie de

operaciones muy complejas y con una carga computacional muy elevada. Por culpa de esto, es muy complicado, computacionalmente hablando, conseguir una contraseña a partir de su hash. Otro uso muy común de los algoritmos de hash es aquel relacionado con la integridad de los mensajes, ya que cada documento genera su propio hash, y en el caso de que este haya sido modificado, su nuevo hash será diferente al calculado inicialmente. De esta forma, sabremos si un fichero ha sido modificado después de su creación.

Entre los algoritmos de hash más populares en la actualidad podemos destacar los siguientes: MD5, SHA-1, SHA-2, SHA-3, RIPEMD, NTLM, BLAKE2, etc.

A pesar de la gran dificultad computacional que supone conseguir una contraseña a partir de su hash, en la actualidad existen métodos que nos permiten realizar ataques con los que se consigue dicho propósito. Un ataque a un hash se basa en que un atacante tiene acceso a un hash del cual quiere averiguar su procedencia, es decir, quiere obtener los caracteres a los que aplicándole determinado algoritmo de hash dan como resultado ese hash en concreto. Normalmente, dichos caracteres pertenecen a una contraseña. Para ello, lo que se hace es crear una serie de contraseñas candidatas, a las cuales se les aplica la misma función de hash que se le aplicó al original, y el resultado de cada una de ellas se comparará en un futuro con el hash original. En el caso de que ambos hashes coincidan, el atacante sabrá a qué conjunto de caracteres corresponde dicho hash, por lo que habrá conseguido la contraseña de su víctima. En este tipo de ataques es importante generar las mejores contraseñas candidatas posibles, ya que son estas las que se comparan con la original. Para generar dichas contraseñas candidatas existen distintos métodos, entre ellos podemos destacar la técnica de fuerza bruta o el ataque por diccionario. Muchas de estas técnicas las implementa Hashcat, por lo que se explican más adelante.

2.1.4 CUDA

CUDA (Compute Unified Device Architecture) es una tecnología desarrollada por Nvidia y que consiste tanto en una plataforma de computación en paralelo como en un modelo de programación pensado para la computación en GPUs [4]. Con esta tecnología se consigue que los desarrolladores puedan explotar paralelismo en aplicaciones de la forma más eficiente posible mediante el uso de las GPUs, ya que estas se encargan de las partes de la ejecución que necesitan una computación más intensiva, por lo que gracias a CUDA se logran aprovechar dichas unidades de procesamiento gráfico al máximo.

En la sección 2.3 de este capítulo se explica cómo se ha llevado a cabo el aprendizaje de esta tecnología. La fase de aprendizaje de CUDA antes de entrar de lleno en su uso es muy importante, debido a que con un conocimiento previo es mucho más sencillo y eficiente la utilización de cualquier lenguaje, herramienta o tecnología.

2.1.5 OpenCL

OpenCL (Open Computing Language) es un API que permite una computación heterogénea ejecutable en una gran diversidad de dispositivos, proporcionando así gran portabilidad. En el caso de las GPUs de Nvidia, OpenCL opera sobre la tecnología de CUDA [5]. Esta tecnología fue creada inicialmente por Apple y posteriormente desarrollada por un conjunto de empresas. Este entorno está formado por un lenguaje y una interfaz de programación, que usado de forma conjunta permite la creación de código que utilice paralelismo y ejecutable tanto en CPUs, como en GPUs y otros tipos de dispositivos.

2.1.6 Hashcat

Hashcat es una de las herramientas de recuperación de contraseñas más rápidas que existen en el mundo y fue creada por Jens Steube (como se puede observar en el propio github de la herramienta [6]), un desarrollador de software alemán especializado en el cracking de contraseñas. Hashcat es un software de código abierto, por lo que es muy sencillo acceder a él.

Actualmente existe una versión (5.1.0) que combina dos que en un pasado se encontraban de forma separada, por un lado el Hashcat basado en CPU, el cual ahora se llama hashcat-legacy, y por otro lado el Hashcat basado en GPU llamado oclHashcat [7]. Esto nos permite ignorar, en un principio, las diferencias entre los ataques basados en la CPU y los basados en las unidades de procesamiento gráfico, usando una única herramienta que opera sobre ambos tipos de dispositivos.

En general, si se utiliza la GPU para crackear contraseñas con los algoritmos que soporta la herramienta, se va a tardar menos tiempo que si lo hacemos utilizando la CPU, por lo que normalmente se suele hacer uso de dichas unidades de procesamiento gráfico para realizar estos ataques. A continuación se observa qué partes de Hashcat están implementadas en GPU y cuáles no. Además se identifican los usos más comunes que se le dan a este software.

Partes de Hashcat implementadas en GPU

Antes de entrar de lleno en las implementaciones de Hashcat para GPU es importante explicar cómo fue evolucionando la versión de este software dedicado a dichas unidades de procesamiento gráfico (ya que, como se explicó anteriormente, en un principio la herramienta Hashcat se dividía en la parte basada en CPU y la basada en GPU), por lo que a partir de ahora vamos a hablar principalmente de esa parte del software. Si nos remontamos al principio de Hashcat, su implementación para GPUs funcionaba muy bien con los algoritmos rápidos, pero no era eficiente cuando trataba con algoritmos más modernos [8]. Esto dio lugar a que se continuase utilizando Hashcat basado en CPU, por lo que los desarrolladores de la herramienta

fueron buscando soluciones para cambiar esto, ya que sabían que se le podía sacar mucho más partido a la herramienta usando unidades de procesamiento gráfico en vez de CPUs. Con el tiempo consiguieron sacar a la luz una versión que combinaba varias versiones anteriores de su software basado en GPUs, dando mejores resultados. Con esto, se consiguió que los algoritmos más modernos (algoritmos lentos) se pudiesen crackear de manera más eficiente con el software basado en GPU que con el basado en CPU. Además, al incorporar un motor centrado en reglas se consigue que los ataques simples que utilizan diccionarios sean eficientes en la GPU, ya que sin este motor llevaba demasiado tiempo transferir los datos del diccionario a la memoria de la GPU, por lo que se utilizaba la CPU para estos casos [8]. A partir de este momento se comenzó a utilizar el software basado en GPUs para las ocasiones en las que se buscaba hacer un crackeo de hashes de forma más seria, buscando la mayor eficiencia posible.

A la hora de centrarse en la implementación de Hashcat para unidades de procesamiento gráfico, se puede observar que soporta un gran número de algoritmos para su uso, encontrándose entre ellos: md5, sha1, joomla, hmac, smf, NTLM, sha256, sha512, WPA, IKE, Kerberos, algoritmos de Oracle, Cisco, MySQL, Grub 2 y un largo etcétera, todos ellos para GPUs tanto de Nvidia como de AMD. Además de escoger el tipo de algoritmo de hash, también es posible elegir, entre otros, el modo de ataque. Según la wiki de la herramienta [8] soporta los siguientes modos:

- **Ataque por fuerza bruta**: se trata de ir probando todas las combinaciones posibles de caracteres hasta dar con la contraseña. Mediante este método se prueba de forma exhaustiva todo el conjunto de contraseñas posibles, dentro del marco de soluciones. Al estar probando un número tan grande de contraseñas candidatas, el tiempo de ejecución de un ataque de este tipo suele ser muy elevado.

- **Dictionary Attack**: también conocido como Wordlist attack. Este método, prueba una a una cada palabra de un diccionario, comprobando si el hash de alguna de ellas se corresponde con el de la original. Dichos diccionarios son un conjunto de palabras combinadas con símbolos y números, que suelen utilizarse como contraseña.

- **Combinator attack**: combina dos diccionarios de palabras concatenando cada palabra de uno de ellos con todas las palabras del otro. Con esto, estamos consiguiendo aumentar el rango de soluciones posibles, con lo que tendremos un mayor número de posibilidades de realizar el ataque con éxito. Una opción dentro de este tipo de ataque es trabajar con dos diccionarios iguales, de esta forma dicho diccionario se combina consigo mismo.

- **Fingerprint Attack**: este consiste en la generación automática de reglas y en un proceso de búsqueda de patrones para obtener las contraseñas candidatas. Este método funciona mejor en los casos en los que se utilice un dispositivo de tipo GPU al realizar el ataque.

- **Hybrid Attack**: es una variante del ataque llamado Combinator, explicado anteriormente. La diferencia entre los dos es que en este, en vez de trabajar con dos diccionarios, lo que se hace es concatenar todas las palabras de un diccionario con el resultado de un ataque por fuerza bruta.
- **Mask Attack**: método de ataque similar al ataque por fuerza bruta, pero algo más específico, por lo que consigue reducir el espacio de claves candidatas, consiguiendo así un ataque más eficiente.
- **Permutation Attack**: es otra variante del ataque por diccionario. En este caso existe un diccionario, del cual para cada palabra se generan todas las permutaciones posibles de sí misma. De esta forma se consigue ampliar de forma drástica el número de contraseñas candidatas.
- **Rule-based Attack**: es uno de los más complejos ya que es una especie de lenguaje diseñado para generar claves candidatas. Esta gran complejidad hace que sea uno de los más eficientes.

Sin embargo, en su versión actual en la que junta las versiones basadas en CPU y GPU solamente soporta los ataques de tipo Brute-Force attack, Combinator attack, Dictionary attack, Hybrid attack, Mask attack, Rule-based attack y Toggle-Case attack (este último consiste en generar distintas claves cambiando entre mayúsculas y minúsculas las distintas letras de cada palabra). A pesar de esta información que se muestra en la página oficial de Hashcat, la herramienta usada desde el terminal de linux nos proporciona los siguientes modos de ataque con sus respectivos códigos: 0-Straight (ataque por diccionario), 1-Combination, 3-Brute-force, 6-Hybrid Wordlist + Mask y 7-Hybrid Mask+ Wordlist.

Por otro lado, si se observa la parte del software actual de Hashcat que se basa en GPUs, también se encuentra alguna limitación, como es el caso de que la mayor parte de los algoritmos tienen un número máximo de caracteres igual a 55 para la longitud de las contraseñas, y además cuando los algoritmos usan unicode esta cifra baja a 27 caracteres [8].

Como se puede ver, el uso de la parte de la herramienta que se basa en las unidades de procesamiento gráfico supone en la mayor parte de los casos un gran número de ventajas frente al uso del software con CPUs, pero a su vez también aparecen algunas pequeñas limitaciones.

Usos más comunes

Como ya se ha comentado anteriormente, Hashcat tiene como función principal la recuperación de contraseñas a partir del hash de las mismas, pudiendo hackear las más complejas de una forma sencilla. Este hecho da pie a que miles de atacantes usen este software para atacar víctimas de las cuales pueden obtener el acceso a los ficheros en los que guardan los

hashes de sus contraseñas, por lo que si los atacantes disponen de un equipo adecuado no les resulta muy complejo conseguir su objetivo.

Por otro lado, esta herramienta también tiene usos para beneficio propio sin que afecte a terceros. Un ejemplo de ello es el siguiente: un usuario normal podría guardar en un fichero cifrado los hashes de sus contraseñas más importantes (de forma que a ese archivo solo pueda acceder él, guardándolo en un dispositivo hardware diferente como puede ser un pendrive). En el caso de pérdida de alguna de sus contraseñas puede acceder a este archivo descifrándolo y posteriormente usar Hashcat para obtener su contraseña a partir del hash. Esto es bastante seguro ya que se están guardando las contraseñas hasheadas y a su vez se está cifrando el archivo en el que se guardan. Además, para mejorar dicha seguridad se guardaría físicamente en lugares distintos, para de esta forma asegurar que el único que tenga acceso a las contraseñas sea el propietario de las mismas.

2.2 Estudio de alternativas

A lo largo del desarrollo de un proyecto de este estilo es muy común encontrarse con distintas alternativas con las que se puede trabajar para llegar a un mismo objetivo. Es muy importante estudiar todas las opciones para tomar la mejor decisión posible. Por esta razón, a continuación se explican todas las alternativas y opciones que se fueron teniendo a lo largo del desarrollo de este proyecto, explicando para cada una de ellas sus ventajas y desventajas. Además, se detalla cuál de las alternativas ha sido la escogida finalmente para llevar a cabo el trabajo.

2.2.1 Herramientas de análisis de rendimiento

En primer lugar, se describen las herramientas que se han encontrado para realizar distintos análisis de rendimiento y de uso de las funciones de Hashcat, con las cuales se hace un estudio a fondo de la aplicación. En este caso, se barajan dos opciones y ambas se acaban utilizando debido a que, a la hora de hacer un análisis, es importante tener distintos puntos de vista mediante herramientas que funcionan de forma diferente. En este caso vamos a hablar de Grpof e Intel Paralel Studio.

Gprof

Gprof es una herramienta que permite realizar un análisis de una ejecución de un código, de forma que se pueden obtener estadísticas de tiempo y uso de CPU/GPU de cada una de las funciones utilizadas, así como un árbol de funciones en el cual se pueden observar las llamadas que hacen y las veces que se ejecuta cada una de ellas [9]. Esta herramienta está

pensada para entornos Unix. Esto es un punto a favor ya que, como se explica en la sección 2.2.2, en este trabajo se utiliza un entorno Linux.

Una de sus desventajas es que sus datos no son exactos al 100%, debido a que emplea un muestreo estadístico, por lo que siempre existe un pequeño error. Esto no es un problema si queremos aplicar esta herramienta para ver un resumen de cuáles son las funciones que más utiliza Hashcat y una aproximación del porcentaje de uso de CPU de cada una de ellas, como ocurre en nuestro caso [10].

Intel Paralel Studio: Intel Vtune

En este subapartado, tenemos una segunda herramienta que fue creada por Intel y permite analizar ejecuciones de código paralelo a través de sus distintas funcionalidades. Intel Paralel Studio no solamente está pensada para realizar análisis, sino que fue desarrollada con el propósito de crear y modernizar código además de aumentar su rendimiento [11]. Como desventaja, en un primer momento nos encontramos que no es una herramienta free source, sino que hay que pagar una licencia. Pero indagando un poco en su página web se puede observar que tiene una licencia de estudiante a la cual tenemos acceso desde cualquier cuenta de la UDC. Esto, que en un principio podría ser un inconveniente, se convierte en una ventaja, ya que al ser una herramienta de pago está continuamente actualizada y proporciona un soporte continuo.

2.2.2 Equipos en los que se realizan las pruebas de rendimiento

Para la realización de este proyecto se utilizan dos plataformas hardware a lo largo de las distintas fases del trabajo. Para aquellas que no están relacionadas con el rendimiento se opta por la utilización del portátil personal descrito a continuación, debido a la mayor accesibilidad del mismo y una mayor flexibilidad a la hora de hacer pruebas e instalaciones. Este es un portátil ASUS, el cual dispone de una tarjeta gráfica GeForce GTX 1050 dedicada de NVIDIA, una CPU Intel Core i-7-8550U y 16GB de RAM. El sistema operativo con el que se trabaja en él es un Linux Ubuntu 18.04LTS. Por otro lado se ha dudado entre la utilización de dicho ordenador personal o un clúster del grupo de arquitectura de computadores de la Universidad de A Coruña a la hora de llevar a cabo las tareas de análisis de rendimiento, ya que para ambos casos se encuentran ventajas y desventajas que se explicarán a continuación. Antes de ver los puntos a favor y en contra de cada uno de ellos se nombran las características principales de dicho clúster llamado Pluton. Este está formado por un nodo frontend, el cual es el punto de entrada desde el exterior, y 20 nodos de cómputo divididos en 2 cabinas. En resumen tiene 20 nodos de cómputo para un total de 336 cores físicos, 1.4 TB de memoria, 17 aceleradores NVIDIA Tesla, 3 aceleradores Intel Xeon Phi y 1 acelerador AMD FirePro. El sistema operativo utilizado en dicho clúster es CentOS Linux release 7.7.1908.

Ordenador portátil

Existe la posibilidad de llevar a cabo todas las tareas del trabajo con el ordenador personal que dispone de una tarjeta gráfica de NVIDIA, por lo que se podría probar en él el funcionamiento de la tecnología CUDA, ya que es precisamente en los productos de este fabricante en los que se quiere ver si se mejora el rendimiento gracias a dicha tecnología. Para esta primera opción se encuentran más inconvenientes que ventajas:

- Ventajas: como se acaba de comentar, una ventaja muy importante es que el dispositivo utiliza hardware de NVIDIA, por lo que para ver si las traducciones dan lugar a un mayor rendimiento podría valer en principio. Por otra parte también existe la ventaja de que al ser un ordenador personal se pueden hacer todo tipo de instalaciones en él sin ninguna restricción. Por último, otro punto a favor es que se puede usar este dispositivo en cualquier momento, sin tener que estar pendientes de tareas de mantenimiento, disponibilidad de los recursos u otros contextos en los que no podamos tener acceso al clúster. En este caso, al estar refiriéndonos a las pruebas de rendimiento puntuales tampoco es necesaria una gran disponibilidad, por lo que esta última ventaja no es demasiado importante.
- Desventajas: nos podemos encontrar con temas relacionados con la capacidad de cómputo, ya que esta es mucho menor en el ordenador personal que en el clúster. Además al utilizar una sesión personal no nos encontramos con un entorno preparado específicamente para correr aplicaciones que requieran mucha carga de la CPU y es muy probable que existan diversos procesos paralelos que utilicen la CPU a la vez que Hashcat, lo cual haría que las pruebas de rendimiento no sean tan exactas.

Clúster Pluton

Por otro lado nos encontramos con el clúster Pluton, que pertenece al grupo de arquitectura de computadores de la Universidad de A Coruña, el cual, en un principio, está mucho más capacitado para realizar las pruebas de rendimiento.

- Ventajas: el clúster tiene una capacidad de cómputo mucho mayor que la del portátil, por lo que es más sencillo que ejecute aplicaciones que precisen de muchos recursos y un gran porcentaje de la CPU, como es el caso de Hashcat. Además el uso de este clúster se hace mediante sesiones interactivas que se abren en el momento que un usuario quiera y tienen una duración limitada. Por ello se parte de un entorno ”limpio”, en el que no hay prácticamente procesos por detrás que estén consumiendo CPU. Esto último favorece a las tareas relacionadas con el análisis de rendimiento y estudios de tiempos de ejecución. Además, este clúster también utiliza hardware del fabricante NVIDIA, de

manera que se puede utilizar para correr Hashcat una vez que haya sido implementado con CUDA.

- Desventajas: al estar hablando de tareas puntuales (ya que las pruebas de rendimiento se realizan en puntos determinados del desarrollo de este trabajo) no vamos a tener en cuenta la disponibilidad del clúster o los momentos puntuales en los que no se tenga acceso al mismo. La desventaja más clara es que no es un clúster de uso público y hay que solicitar una cuenta para poder comenzar a trabajar en él, lo cual no es algo tan sencillo como utilizar un ordenador personal.

Una vez comparadas las opciones, se llega a la conclusión de que todas las tareas de análisis de rendimiento tienen más fiabilidad en el clúster Pluton, ya que está más preparado para este tipo de pruebas. Las restantes tareas se realizan mayoritariamente en el ordenador personal descrito anteriormente, como ya se ha comentado. Durante el desarrollo de las pruebas de rendimiento no se utilizan todas las plataformas disponibles en el clúster, sino que sólo se usan las GPUs de Nvidia llamadas Tesla Kepler K20m y Tesla Kepler K40c.

Además, para la realización de este trabajo se utilizan distintas herramientas software que se explican a lo largo de este documento según sean necesarias y que son las siguientes: aplicación Hashcat, herramientas de análisis de rendimiento (Gprof e Intel Vtune Profiler) y herramientas para llevar a cabo técnicas de depuración (GDB desde Microsoft Visual Studio Code).

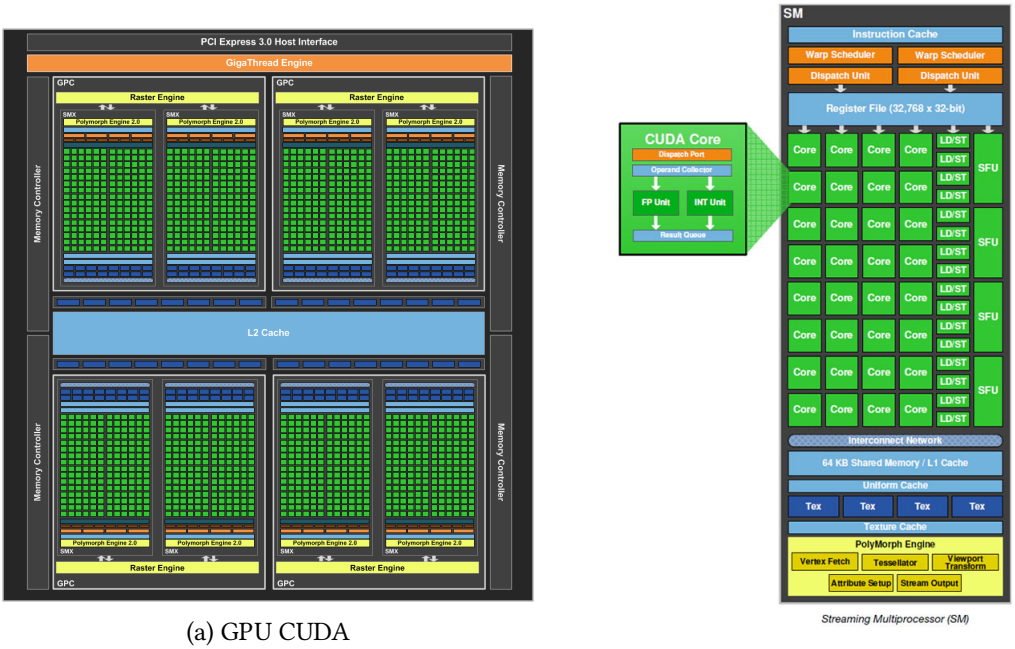
2.3 Aprendizaje en CUDA

A lo largo de esta breve sección se hace un pequeño resumen de la forma en la que se ha realizado un aprendizaje del lenguaje CUDA. Este aprendizaje es muy importante, debido a que a lo largo del desarrollo de este proyecto se intenta mejorar el código de Hashcat mediante algunas de las funcionalidades que nos ofrece dicha tecnología. Estas mejoras están enfocadas a dispositivos con hardware de NVIDIA, ya que existen numerosas opciones e implementaciones parecidas, pero que en OpenCL se realizan de forma menos eficiente. Una primera idea es conseguir dichas mejoras gracias a la ejecución de código paralelo con CUDA. Por lo tanto, son muy importantes las distintas funcionalidades que esta tecnología nos aporta a la hora de querer llevar a cabo una paralelización en el código mediante el uso de varios threads paralelos en la ejecución. Por otro lado, es conveniente realizar una buena fase de aprendizaje, ya que si no se hace esto, el proceso de desarrollo es mucho más lento, debido al desconocimiento de la tecnología utilizada.

En primer lugar se realizan una serie de tutoriales que se encuentran en la propia página de NVIDIA [12], en los cuales se tratan distintos temas importantes como pueden ser la

asignación de memoria de la forma más eficiente posible, el uso de threads, la gestión de los bloques, etc. Una vez que los tutoriales anteriores se completan y entienden de forma clara, se crean una serie de ficheros en los que ir probando las distintas funcionalidades de CUDA aprendidas anteriormente. Esto se hace con el objetivo de familiarizarse con la tecnología. Además de todo esto y a lo largo de todo el proceso de la implementación en CUDA, se hacen diferentes consultas a la documentación oficial de Nvidia para resolver dudas en temas más concretos que no han sido explicados en los tutoriales nombrados anteriormente.

Antes de explicar alguna parte concreta del aprendizaje de esta tecnología, es importante tener claros una serie de conceptos acerca de la estructura de una GPU de CUDA. Dichas GPUs están formadas por varias unidades de ejecución, conocidas como SM (Streaming Multiprocessors), las cuales a su vez están formadas por varios núcleos de cómputo llamados núcleos CUDA o SP (Streaming Processors). Estos últimos son los que se encargan de ejecutar todas las instrucciones. Esta estructura es la que permite el uso de paralelización en los programas escritos en CUDA de forma sencilla y muy eficiente. En la figura 2.3 se puede observar un esquema de la estructura explicada anteriormente, en la que se ve cómo una GPU de CUDA se divide en SMs y estos a su ves en SPs.



(a) GPU CUDA

(b) Streaming Multiprocessors (SM)

Figura 2.3: Estructura de una GPU CUDA.

Uno de los apartados más importantes en la fase de aprendizaje es aquel que está relacionado con la paralelización mediante el uso de varios threads paralelos en tiempo de ejecución. Con ello, conseguimos que un código que anteriormente se corría en un solo hilo de ejecu-

ción pase a ejecutarse a la vez en varios threads, de forma que se reduce mucho el tiempo de ejecución al repartirse la carga computacional entre los distintos hilos. Esta práctica se puede hacer de varias formas a la hora de llamar a una función de un kernel: desde utilizar algunas funciones que nos ofrece CUDA, como puede ser el caso de cuLaunchKernel(), a la cual le pasamos como parámetro una serie de variables con las que indicamos el número de threads que se utilizarán y distinta información acerca de la paralelización, hasta indicar estos datos directamente en la llamada a dicha función mediante los símbolos "«< »>". De esta última forma lo que hacemos es señalar en la propia llamada al kernel el número de bloques que se utilizan y el tamaño de cada uno de ellos (esta es la manera más habitual de utilizar la paralelización en CUDA). La forma en la que se hace una llamada de este tipo se puede observar en el listado 2.1.

```
1 funcion_kernel<<<numBlocks,blockSize>>>(param1, param2, param3)
```

Listado 2.1: Llamada a un kernel utilizando paralelización en CUDA.

Una vez comentadas algunas de las formas que nos permiten indicarle al kernel de CUDA cómo queremos que se lleve a cabo la paralelización, es importante saber cómo funciona esto por dentro y la forma de aprovecharnos de ello. Antes de comenzar con un ejemplo es importante tener claro el significado de algunos conceptos que se utilizan más adelante. En primer lugar tenemos el concepto de bloque, el cual hace referencia a un conjunto de hilos que trabajan en paralelo. En el caso de que tengamos un tamaño de bloque de 1024, significa que tenemos 1024 threads que trabajan de forma paralela. Por otro lado existe el concepto de grid (también conocido como cuadrícula o malla), referido al número de bloques que tenemos en total y en los que se agrupan los distintos hilos de ejecución. Por lo que si tenemos un tamaño de grid igual a 128, tenemos 128 bloques de threads, y por lo tanto, los hilos de ejecución se dividen en 128 grupos. Otro concepto importante en CUDA es el de dimensión, que se refiere a la posibilidad de trabajar con arrays de una o más dimensiones. Un uso común de arrays de varias dimensiones se da en el caso de procesado de imágenes o en el de resolución de ecuaciones diferenciales. A pesar de estas distintas opciones en cuanto al término de dimensión, se comprueba que Hashcat sólo trabaja con una dimensión (X). Aún así, es importante saber que tanto el concepto de bloque, como el de cuadrícula están presentes en las 3 dimensiones existentes (X, Y y Z).

Un ejemplo claro para ver la forma en la que podemos aprovechar la paralelización de CUDA es el caso en el que tengamos un bucle dentro del kernel que se va a ejecutar n veces. De esta forma, en una ejecución sin paralelización un thread hace las n iteraciones del bucle, pero aprovechándonos de las funcionalidades que nos ofrece CUDA podemos hacer que todos

los hilos de ejecución que se utilicen se repartan las iteraciones del bucle, de manera que se puedan hacer varias iteraciones a la vez. La identificación de cada hilo se realiza mediante una serie de variables disponibles en los kernels de Nvidia, que son blockIdx.x, blockDim.x, threadIdx.x y gridDim.x. En primer lugar, blockIdx.x hace referencia al índice del bloque de threads actual, blockDim.x se refiere al tamaño de cada bloque de threads, threadIdx.x al índice actual dentro de un bloque de threads en concreto y por último gridDim.x contiene el número bloques de threads totales. Una vez que entendemos bien estas variables podemos llegar a la conclusión de que el índice del bucle puede pasar a ser igual a blockIdx.x * blockDim.x + threadIdx.x y que el bucle va a ir avanzando en pasos de blockDim.x * gridDim.x unidades. Esto se entiende mejor con un ejemplo gráfico, por lo que a continuación, en la figura 2.4 podemos observar un esquema tras el que veremos un ejemplo del bucle antes y después de aplicar esta funcionalidad de CUDA. (Todo esto está explicado en el tutorial de Nvidia nombrado anteriormente [12]).

La figura 2.4 representa la forma en la que se hace el reparto de trabajo en CUDA. En este caso en concreto se está trabajando con un tamaño de grid igual a 4096. Con esto sabemos que la cuadrícula se divide en 4096 bloques de threads. Además, vemos que el tamaño de bloque es igual a 256, por lo que cada uno de los 4096 bloques nombrados anteriormente está compuesto por 256 threads. Como se observa en la parte de abajo de la figura, podemos acceder a una posición específica mediante la fórmula: blockIdx.x * blockDim.x + threadIdx.x. En el caso de la figura se está accediendo a la posición 515 (marcada en naranja), por lo que para ello tenemos que acceder al bloque número 2 y dentro de este a la posición local 3.

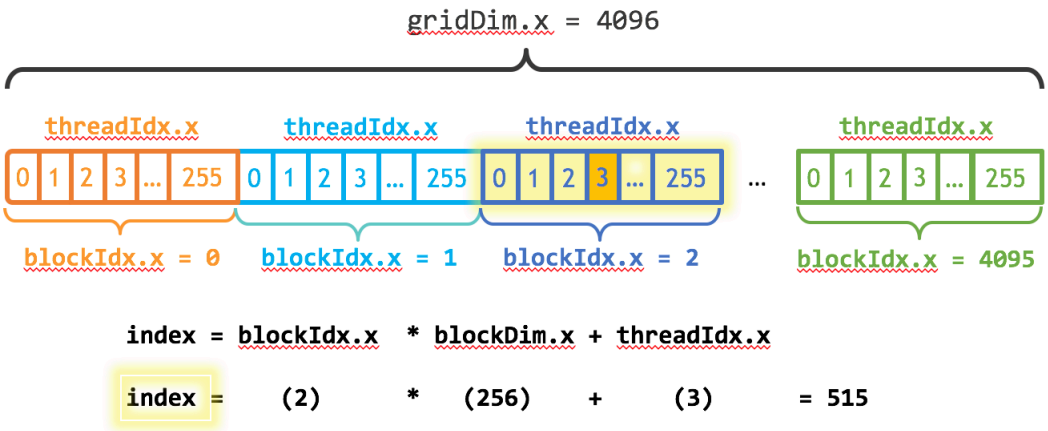


Figura 2.4: Esquema del funcionamiento de los bloques en CUDA.

Si aprovechamos este tipo de reparto de trabajo que nos ofrece CUDA, podemos hacer que un código que, en un principio, no está aprovechando al máximo la paralelización, pase a

ejecutarse con varios hilos de ejecución paralelos y con un reparto de trabajo equitativo. De esa forma se consigue aumentar la eficiencia de cada ejecución.

Partiendo del código mostrado en el listado 2.2, en el cual tenemos un bucle con n iteraciones, vemos que lo que se está haciendo es que cada thread está resolviendo varias iteraciones del bucle antes de que otros threads entren en él. Con esto, se consigue un reparto de trabajo desequilibrado, ya que hay una serie de hilos que hacen la mayor parte del trabajo y otros que no hacen nada.

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

Listado 2.2: Código del bucle antes de la mejora.

Para solucionar este problema lo que podemos hacer es lo que se muestra en el listado 2.3, en el cual se ve que se modifica tanto el índice como el avance del bucle. Con esta modificación se consigue que un thread entre en el bucle y haga la operación correspondiente a su posición en el vector, y en el caso de que el vector sea de un tamaño mayor que el número de elementos de trabajo (blockDimx.x * gridDim.x), hace otra operación. Es decir, se hace un reparto cíclico, en el que un thread no hace una segunda operación hasta que el resto de threads no hayan hecho la primera. De esta forma se obtiene un reparto de trabajo equitativo y como consecuencia, una mayor eficiencia.

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

Listado 2.3: Código del bucle depués de la mejora.

Capítulo 3

Metodología y planificación

En este capítulo se explica detalladamente la metodología utilizada en el proyecto, así como la planificación acordada para su desarrollo, de forma que se describan de la mejor manera posible las distintas tareas llevadas a cabo durante todas las fases de elaboración de este trabajo. Por último se dedicará una sección a la explicación del coste total del proyecto.

3.1 Metodología

Para la realización de este proyecto se utiliza la metodología SCRUM, la cual consiste en un marco de trabajo con el que se busca una mayor colaboración en los equipos que van a desarrollar productos complejos [13]. No es un conjunto de reglas que haya que cumplir, sino que es una serie de buenas prácticas que nos ayudan a mejorar la colaboración entre los miembros de un equipo de trabajo.

Esta metodología surge en los años 80 a partir de un análisis realizado por Ikujiro Nonaka y Takeuchi acerca de la forma en la que las empresas más importantes de la época desarrollaban sus productos. El nombre SCRUM traducido al español significa ”melé”, que es una formación de rugby basada en el trabajo en equipo, cuyo objetivo es conseguir un avance con la pelota partiendo de un juego en parado. Nonaka y Takeuchi vieron semejanzas entre la metodología que habían analizado y esta táctica de rugby, por lo que la nombraron de la misma forma (SCRUM) [14].

Es importante destacar que SCRUM utiliza una estrategia de desarrollo en la que se avanza de forma incremental y en cada una de sus iteraciones se pasa por todas las fases del desarrollo, por lo que a diferencia de otros tipos de estrategias, en SCRUM nos encontramos con una estructura cíclica en la que a cada ciclo se le llama Sprint. Cada uno de estos Sprints se corresponde con un periodo de tiempo (normalmente 1 o 2 semanas) en el cual se desarrolla el trabajo necesario para hacer la entrega de una iteración. Esto se puede observar en la figura 3.1.

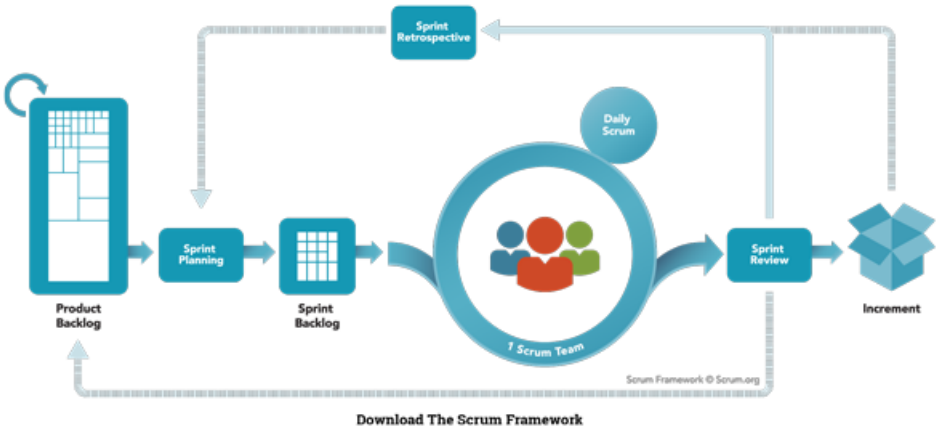


Figura 3.1: Diagrama SCRUM.

Gracias a este diagrama se puede explicar de forma sencilla el funcionamiento de la metodología de SCRUM, en la cual se puede observar que el ciclo de SCRUM parte de una iteración determinada del producto, a excepción del primer periodo, en el que no existe una base de la que partir. Lo primero que se hace en el ciclo es determinar los objetivos de este Sprint para tener establecidas desde un principio las tareas que se van a llevar a cabo. Como consecuencia, se consigue que surjan las menores desviaciones de la línea general del Sprint posibles, con lo que se trabaja con tareas más específicas y por lo tanto más controladas. Una vez que se comience a trabajar en una iteración del producto, se hace una reunión diaria en la que cada miembro del equipo explica todo lo que ha hecho desde el día anterior y lo que tiene pensado hacer ese día. Esto se hace para que todo el grupo de trabajo esté informado y en el caso de que surjan problemas, solucionarlos con la mayor brevedad posible. Una vez se acabe el periodo de trabajo de cada Sprint se hará una reunión final para ver si se han logrado los objetivos y cómo afrontar la siguiente iteración.

Cuando se termina cada ciclo de trabajo se vuelve a comenzar uno nuevo de forma que en cada uno de ellos se va pasando por todas las fases de desarrollo. En el caso particular de este proyecto, lo que se hace en cada Sprint es una reunión inicial con los directores del TFG para establecer unos objetivos de cara a cada ciclo de trabajo, y por regla general cada uno de ellos dura entre 1 y 2 semanas. Además de esto, las reuniones diarias de todo el equipo de trabajo se ven reflejadas como una revisión diaria de las distintas fases dentro del Sprint de forma que cada día se identifican los posibles problemas, posibles soluciones y se llevan a cabo las que mejor se adapten a cada contexto.

En resumen, esta metodología proporciona una serie de ventajas de las que nos podemos aprovechar. Entre ellas nos encontramos con una mayor facilidad a la hora de hacer un planning y establecer los tiempo de trabajo para cada parte del proyecto, una mayor tolerancia a

cambios y, entre otras, reducir los riesgos. En la sección 3.2, se explican de forma más detallada todas las fases del proyecto gracias al uso de la herramienta Microsoft Project, la cual nos permite planificar tarea a tarea nuestro proyecto y realizar varios diagramas, como es el caso del diagrama de Gantt.

3.2 Planificación

Como se comenta en la sección anterior, se lleva a cabo una planificación siguiendo la metodología de SCRUM, lo cual implica que nos basamos en una serie de periodos de 1-2 semanas en los que se hacen diversas reuniones para establecer objetivos y ver los avances. En primer lugar, se identifican una serie de fases en las que se divide el proyecto en un contexto general y cada una de ellas se explica de forma individual a lo largo de este documento. Dichas fases son las siguientes: en un primer lugar se realiza un estudio de Hashcat en la sección 4.1, para de esta forma ver cómo funciona e identificar las partes en las que podemos mejorar el código utilizando la paralelización de CUDA. En dicha fase se estudia de forma muy profunda el código de la aplicación, para que en un futuro resulte mucho más sencillo trabajar con el código de la herramienta. Una vez hecho esto se lleva a cabo un aprendizaje de la tecnología CUDA, con el objetivo de poder realizar futuras implementaciones de forma sencilla utilizando las funcionalidades que nos ofrece dicha tecnología. Este aprendizaje se explica en la sección 2.3. A continuación y gracias a lo visto anteriormente, se implementan dichas mejoras en la herramienta, haciendo uso de CUDA. Esto último está explicado en la sección 4.2. Una vez hecho esto y para finalizar, se realizan varios estudios de rendimiento de la herramienta Hashcat antes y después de la implementación de las mejoras, con el objetivo de llegar a una conclusión y determinar si realmente vale la pena implementarlas.

Para llevar a cabo una planificación más precisa se utiliza la herramienta Microsoft Project, que no es open source, por lo que normalmente es necesario comprar una licencia, pero en este caso no es necesario ya que está disponible en las máquinas virtuales de la facultad, a las cuales tenemos acceso mediante nuestra cuenta de la UDC. Microsoft Project permite representar de forma muy visual cada una de las tareas que se realizan a lo largo de todo el proyecto y la fecha en la que tienen lugar, así como muchas otras características que pueden ser de gran utilidad. Como resultado final de nuestra planificación se obtienen los resultados mostrados en las figuras 3.2 y 3.3, que se explican más detenidamente a continuación.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Sprint 1	**13 días**	**mar 04/02/20**	**jue 20/02/20**		
Búsqueda de información Hashcat	4 días	mar 04/02/20	dom 09/02/20		Pedro
Creación informe hashcat	5 días	lun 10/02/20	vie 14/02/20	2	Pedro
Creación documento en Overleaf	2 días	sáb 15/02/20	dom 16/02/20		Pedro
Instalación y pruebas con Hashcat	2 días	lun 10/02/20	mar 11/02/20	2	Pedro
Documentación en overleaf	1 día	mié 19/02/20	mié 19/02/20	3;4;5	Pedro
Reunión Resumen Sprint 1	0 días	jue 20/02/20	jue 20/02/20	6	Pedro;Director TFG
Sprint 2	**12 días**	**vie 21/02/20**	**vie 06/03/20**	1	
Instalación y pruebas con Gprof	2 días	vie 21/02/20	lun 24/02/20		Pedro
Búsqueda de información sobre compilación hashcat	4 días	mar 25/02/20	vie 28/02/20		Pedro
Creación de análisis de hashcat con Gprof	3 días	sáb 29/02/20	mar 03/03/20	10;9	Pedro
Documentación en overleaf	2 días	mié 04/03/20	jue 05/03/20	11	Pedro
Reunión Resumen Sprint 2	0 días	vie 06/03/20	vie 06/03/20	12	Director TFG;Pedro
Sprint 3	**9 días**	**sáb 07/03/20**	**mié 18/03/20**	8	
Instalación y pruebas Intel Vtune	3 días	sáb 07/03/20	mar 10/03/20		Pedro
Creación de análisis de hashcat con Vtune	3 días	mié 11/03/20	dom 15/03/20	15	Pedro
Documentación en overleaf	2 días	lun 16/03/20	mar 17/03/20	16	Pedro
Reunión Resumen Sprint 3	0 días	mié 18/03/20	mié 18/03/20	17	Director TFG;Pedro
Sprint 4	**6 días?**	**jue 19/03/20**	**jue 26/03/20**	**14**	
Estudio del análisis de Vtune	1 día?	vie 20/03/20	sáb 21/03/20		Pedro
Identificación de funciones hotspot	2 días	dom 22/03/20	lun 23/03/20	20	Pedro
Estudio del árbol de llamadas	2 días	mar 24/03/20	mié 25/03/20	21	Pedro
Reunión Resumen Sprint 4	0 días	jue 26/03/20	jue 26/03/20	22	Director TFG;Pedro
Sprint 5	**9 días**	**vie 27/03/20**	**jue 09/04/20**	**19**	
Ejecución y estudio de hashcat en modo debug	3 días	vie 27/03/20	mar 31/03/20		Pedro
Estudio de las funciones que llaman al kernel	3 días	mié 01/04/20	vie 03/04/20	25	Pedro
Documentación en overleaf	3 días	lun 06/04/20	mié 08/04/20	26	Pedro
Reunión Resumen Sprint 5	0 días	jue 09/04/20	jue 09/04/20	27	Director TFG;Pedro
Sprint 6	**10 días?**	**jue 09/04/20**	**mié 22/04/20**	**24**	
Estudio de los ficheros del kernel	3 días	jue 09/04/20	sáb 11/04/20		Pedro
Estudio de los hashes más utilizados en Hashcat	2 días	dom 12/04/20	lun 13/04/20		Pedro
Aprendizaje en CUDA	4 días	mar 14/04/20	vie 17/04/20		Pedro
Traducción de los kernels elegidos a CUDA de forma literal	2 días	sáb 18/04/20	dom 19/04/20	30;31	Pedro
Documentación en overleaf	2 días	lun 20/04/20	mar 21/04/20	33	Pedro
Reunión Resumen Sprint 6	0 días	mié 22/04/20	mié 22/04/20	34	Director TFG;Pedro
Sprint 7	**10 días**	**mié 22/04/20**	**mié 06/05/20**	**29**	
Estudio de cómo hashcat distingue originalmente los dispositivos OpenCL frente a los CUDA	2 días	mié 22/04/20	jue 23/04/20		Pedro
Compilación de los kernels traducidos a CUDA	2 días	vie 24/04/20	dom 26/04/20		Pedro
Estudio de la compilación de los kernels en la herramienta Hashcat	3 días	lun 27/04/20	mié 29/04/20	37	Pedro
Integración de la compilación de los kernels CUDA en Hashcat	3 días	jue 30/04/20	dom 03/05/20	38;39	Pedro
Documentación en overleaf	2 días	lun 04/05/20	mar 05/05/20	40	Pedro
Reunión Resumen Sprint 7	0 días	mié 06/05/20	mié 06/05/20	41	Director TFG;Pedro
Sprint 8	**9 días?**	**jue 07/05/20**	**mar 19/05/20**	**36**	
Primeras pruebas de funcionamiento y rendimiento	3 días	jue 07/05/20	dom 10/05/20		Pedro
Estudio de Hashcat relacionado con la paralelización	3 días	lun 11/05/20	mié 13/05/20	44	Pedro
Implementación de paralelización en la llamada al kernel CUDA	3 días	jue 14/05/20	sáb 16/05/20	45	Pedro
Documentación en Overleaf	2 días	dom 17/05/20	lun 18/05/20	46	Pedro
Reunión Resumen Sprint 8	0 días	mar 19/05/20	mar 19/05/20	47	Director TFG;Pedro
Sprint 9	**10 días?**	**mié 20/05/20**	**mar 02/06/20**	**43**	
Segundas pruebas de funcionamiento y rendimiento	3 días	mié 20/05/20	vie 22/05/20		Pedro
Creación de 5 nuevos kernels en CUDA	2 días	sáb 23/05/20	lun 25/05/20	50	Pedro
Integración de los nuevos kernels en Hashcat	2 días	mar 26/05/20	mié 27/05/20	51	Pedro
Pruebas de rendimiento de los nuevos kernels	3 días	jue 28/05/20	sáb 30/05/20	52	Pedro
Documentación en Overleaf	2 días	dom 31/05/20	lun 01/06/20	53	Pedro
Reunión Resumen Sprint 9	0 días	mar 02/06/20	mar 02/06/20	54	Director TFG;Pedro
Sprint 10	**10 días?**	**mié 03/06/20**	**mar 16/06/20**	**49**	
Estudio de la posibilidad de una nueva mejora en el rendimiento	3 días	mié 03/06/20	vie 05/06/20		Pedro
Implementación de dicha mejora	2 días	sáb 06/06/20	lun 08/06/20	57	Pedro
Análisis de rendimiento de la última mejora	3 días	mar 09/06/20	jue 11/06/20	58	Pedro
Elaboración de las conclusiones del trabajo	2 días	vie 12/06/20	dom 14/06/20	59	Pedro
Documentación en Overleaf	2 días	lun 15/06/20	mar 16/06/20	60	Pedro
Reunión Resumen Sprint 10	0 días	mar 16/06/20	mar 16/06/20	61	Director TFG;Pedro

Figura 3.2: Tareas Project. Planificación completa.

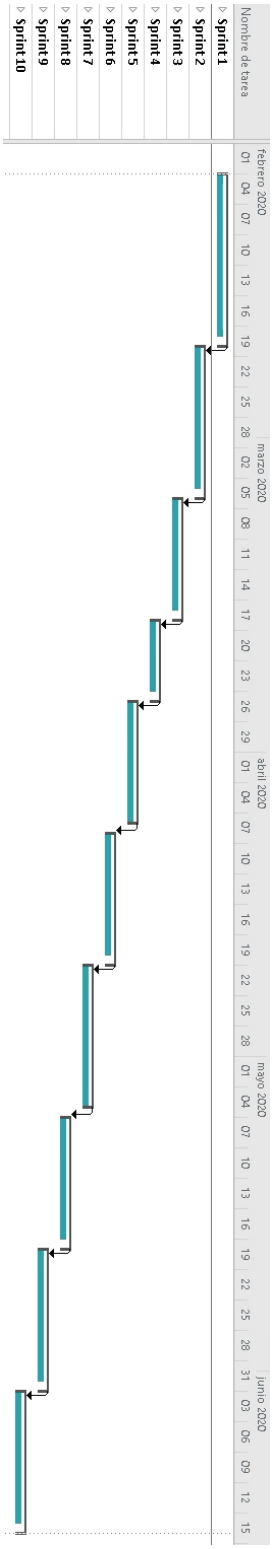


Figura 3.3: Diagrama de Gantt. Planificación completa.

Como se puede ver, hay dos figuras. En primer lugar tenemos la tabla de tareas (3.2), en la cual se describen todas ellas al detalle, y que nos sirve para obtener la segunda figura. En este caso, en la tabla indicamos el nombre de cada Sprint, su duración, su fecha de inicio, su fecha de finalización y las tareas de las que depende. En cuanto a las tareas de cada Sprint, indicamos su nombre, su duración, su fecha de inicio y de fin y su dependencia con otras, al igual que en el caso de los Sprints, pero además también señalamos qué persona la realizó concretamente. Si hablamos de la segunda figura (3.3), podemos ver que se trata del diagrama de Gantt, el cual se construye a partir de la tabla explicada anteriormente y que es una forma muy representativa de ver la temporalidad de cada tarea con respeto al resto (en este caso sólo se muestran los sprints, con el objetivo de facilitar la visualización de la imagen). Para una mejor comprensión se explican uno a uno los Sprints, indicando sus objetivos y tareas principales, mostrando para cada uno de ellos sus figuras correspondientes, con la finalidad de que se entiendan lo mejor posible.

3.2.1 Sprint 1

Nos encontramos ante el primer ciclo del proyecto, por lo tanto estamos en un punto en el cual no se parte de ninguna iteración anterior, sino que se comienza con un proyecto en blanco, aunque ya con los objetivos generales claros y bien determinados. Este primer Sprint comienza el día 4 de febrero de 2020 y termina el día 20 de ese mismo mes, por lo que tiene una duración aproximada de 2 semanas. En esta fase se definen 6 tareas, incluyendo la reunión de resumen del Sprint programada para el último día. En primer lugar, contamos con una tarea que consiste en realizar una búsqueda de información de la herramienta Hashcat para tener una idea general sobre el uso de dicha aplicación. Tras esa búsqueda de información se hace informe recogiendo los puntos claves sobre la herramienta, que se refleja en este documento en la sección 2.1, en la cual se explican los conceptos clave que hay que tener en cuenta para entender este proyecto. A continuación, se crea el documento en la herramienta Overleaf a través de la plantilla oficial de la facultad, con el objetivo de poder comenzar con la realización de la memoria del TFG. Una vez hecho esto, se comienza con la instalación y las primeras pruebas de uso de Hashcat. Tanto el primer análisis de Hashcat, como sus pruebas de uso, son muy importantes para que en un futuro se tenga un mínimo conocimiento de la herramienta y se pueda trabajar de forma más eficiente con ella. Por último y como se hace en gran parte de los Sprints, se documenta todo lo realizado en esta fase en el documento de Overleaf y se hace una reunión final, en la cual se ve si se alcanzaron los objetivos y se definen unos nuevos para la siguiente fase. Las figuras correspondientes a este Sprint son la figura 3.4 y la 3.5.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Sprint 1	**13 días**	**mar 04/02/20**	**jue 20/02/20**		
Búsqueda de información Hashcat	4 días	mar 04/02/20	dom 09/02/20		Pedro
Creación informe hashcat	3 días	lun 10/02/20	mié 12/02/20	2	Pedro
Creación documento en Overleaf	2 días	jue 13/02/20	vie 14/02/20		Pedro
Instalación y pruebas con Hashcat	2 días	sáb 15/02/20	dom 16/02/20	2	Pedro
Documentación en overleaf	3 días	lun 17/02/20	mié 19/02/20	3;4;5	Pedro
Reunión Resumen Sprint 1	0 días	jue 20/02/20	jue 20/02/20	6	Pedro;Directores TFG

Figura 3.4: Tareas Sprint 1.

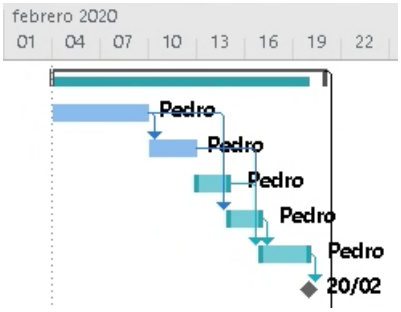


Figura 3.5: Diagrama de Gantt Sprint 1.

3.2.2 Sprint 2

Una vez acabado el Sprint 1 y establecidos los objetivos para la siguiente fase, se crean 5 nuevas tareas que se llevan a cabo a lo largo del Sprint 2. Este ciclo comienza el día 21 de febrero y termina el 6 de marzo, con lo que tiene una duración de 2 semanas. A lo largo de este periodo de tiempo se realizan tareas como la instalación y pruebas de una de las herramientas utilizadas para el análisis de rendimiento y uso de CPU (llamada Gprof), la búsqueda de información sobre la compilación de Hashcat para ver cómo se está haciendo por debajo, haciendo uso del modo DEBUG cuando sea necesario, y por último, se hace un análisis de Hashcat mediante Gprof. También se realizan las tareas de documentación en Overleaf y la reunión resumen en la que se determinan los objetivos de la siguiente fase. Las figuras relacionadas con esta fase son la figura 3.6 y la 3.7.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Sprint 2	**12 días**	**vie 21/02/20**	**vie 06/03/20**	1	
Instalación y pruebas con Gprof	2 días	vie 21/02/20	lun 24/02/20		Pedro
Búsqueda de información sobre compilación hashcat	4 días	mar 25/02/20	vie 28/02/20		Pedro
Creación de análisis de hashcat con Gprof	3 días	sáb 29/02/20	mar 03/03/20	10;9	Pedro
Documentación en overleaf	2 días	mié 04/03/20	jue 05/03/20	11	Pedro
Reunión Resumen Sprint 2	0 días	vie 06/03/20	vie 06/03/20	12	Pedro;Directores TFG

Figura 3.6: Tareas Sprint 2.

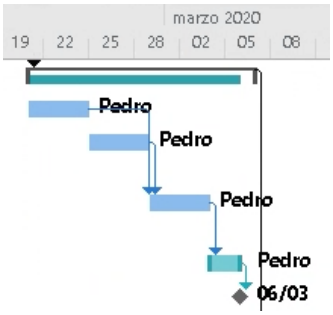


Figura 3.7: Diagrama de Gantt Sprint 2.

3.2.3 Sprint 3

El día 7 de marzo de 2020 se comienza con el Sprint número 3, el cual dura hasta el día 18 de ese mismo mes y en el que se definen 4 tareas principales. En primer lugar se realiza la instalación y las pruebas de la segunda herramienta destinada al análisis de rendimiento de Hashcat, llamada Intel Vtune. Una vez hecho esto, se continúa con la creación de un nuevo análisis de la aplicación, pero esta vez se hace con Intel Vtune en vez de hacerlo con Gprof. Por último se llevan a cabo las 2 tareas con las que se suelen cerrar el resto de Sprints, que son la documentación de lo realizado a lo largo de todo el Sprint en Overleaf y la reunión del último día, en la que se hace un resumen de lo hecho a lo largo de estas 2 semanas y en la que se establecen los objetivos del siguiente ciclo de trabajo.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Sprint 3	**9 días**	**sáb 07/03/20**	**mié 18/03/20**	8	
Instalación y pruebas Intel Vtune	3 días	sáb 07/03/20	mar 10/03/20		Pedro
Creación de análisis de hashcat con Vtune	3 días	mié 11/03/20	dom 15/03/20	15	Pedro
Documentación en overleaf	2 días	lun 16/03/20	mar 17/03/20	16	Pedro
Reunión Resumen Sprint 3	0 días	mié 18/03/20	mié 18/03/20	17	Pedro;Directores TFG

Figura 3.8: Tareas Sprint 3.

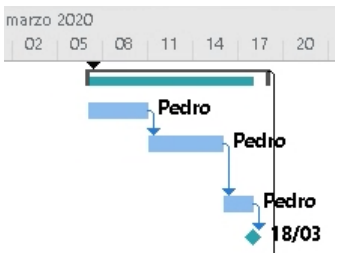


Figura 3.9: Diagrama de Gantt Sprint 3.

3.2.4 Sprint 4

Una vez que haya terminado el tercer ciclo de trabajo se comienza con el Sprint 4, el cual está comprendido entre el 19 de marzo y el 26 de marzo, con lo que tiene una duración de una semana. A lo largo de este periodo se definen 4 tareas, la primera consiste en realizar un estudio del análisis creado a través de la herramienta de Intel en el Sprint anterior, centrado en la identificación de las funciones hotspot de Hashcat, es decir, las funciones en las que más tiempo se pasa la CPU. Esto se hace con el objetivo de tener un conocimiento más profundo de la herramienta y obtener una conclusión sobre qué parte del código de Hashcat implementar con CUDA para obtener un mayor beneficio. La siguiente tarea que se lleva a cabo es el estudio del árbol de llamadas, el cual nos va a ayudar a ver la forma en la que Hashcat hace llamadas a los kernels en su código y entender un poco mejor la estructura de la aplicación. Por último, se hace la reunión de resumen del Sprint y se identican los objetivos de la siguiente fase.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Sprint 4	**6 días?**	**jue 19/03/20**	**jue 26/03/20**	**14**	
Estudio del análisis de Vtune	2 días?	jue 19/03/20	vie 20/03/20		Pedro
Identificación de funciones hotspot	2 días	sáb 21/03/20	lun 23/03/20	20	Pedro
Estudio del árbol de llamadas	2 días	mar 24/03/20	mié 25/03/20	21	Pedro
Reunión Resumen Sprint 4	0 días	jue 26/03/20	jue 26/03/20	22	Pedro;Directores TFG

Figura 3.10: Tareas Sprint 4.

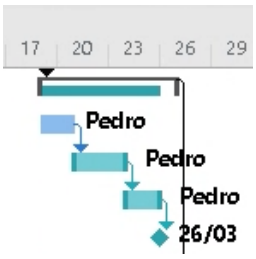


Figura 3.11: Diagrama de Gantt Sprint 4.

3.2.5 Sprint 5

Una vez realizada la reunión de resumen del Sprint 4 podemos continuar con el siguiente, comienza el día 27 de marzo de 2020, termina el 9 de abril del mismo año y está formado por 4 tareas principales. En primer lugar, se ejecuta Hashcat en modo DEBUG para así entender el árbol de llamadas mejor y conocer la forma en la que Hashcat trabaja internamente. Además, esto nos ayuda a realizar la siguiente tarea, la cual consiste en encontrar y estudiar aquellas funciones encargadas de llamar a los ficheros kernel, ya que estos son los que aprovechan la paralelización de CUDA. Por último se documenta lo realizado en esta fase en Overleaf y

se hace la reunión resumen del Sprint con el objetivo de ver si se han logrado los objetivos establecidos.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Sprint 5	**8 días**	**vie 27/03/20**	**mié 08/04/20**	**19**	
Ejecución y estudio de hashcat en modo debug	3 días	vie 27/03/20	mar 31/03/20		Pedro
Estudio de las funciones que llaman al kernel	3 días	mié 01/04/20	vie 03/04/20	25	Pedro
Documentación en overleaf	3 días	sáb 04/04/20	mar 07/04/20	26	Pedro
Reunión Resumen Sprint 5	0 días	mié 08/04/20	mié 08/04/20	27	Pedro;Directores TFG

Figura 3.12: Tareas Sprint 5.

Figura 3.13: Diagrama de Gantt Sprint 5.

3.2.6 Sprint 6

El día 9 de abril comienza el sexto Sprint del proyecto, con una duración aproximada de 2 semanas, por lo que acaba el día 22 del mismo mes. A lo largo de este ciclo de trabajo se desarrollan 5 tareas. En primer lugar se hace un estudio a fondo de los ficheros de los kernels, en los que se comprueba cómo funcionan realmente por dentro. Esto se hace con el objetivo de ver cuál es la forma más eficiente de trabajar con código paralelo a la hora de llamarlos. Al finalizar dicha tarea se lleva a cabo un aprendizaje a fondo de la tecnología CUDA. En este punto del proyecto, se hace un primer análisis de rendimiento en el que se ve cuál es la forma más eficiente de ejecutar un ataque con el código original de la herramienta. Para ello, se realizan ejecuciones desde distintos tipos de dispositivos (tanto CPUs como GPUs), comparando así sus resultados. Al finalizar esa tarea se lleva a cabo la documentación en Overleaf y la reunión de resumen pertinente.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Sprint 6	**10 días?**	**jue 09/04/20**	**mié 22/04/20**	**24**	
Estudio de los ficheros del kernel	3 días	jue 09/04/20	sáb 11/04/20		Pedro
Aprendizaje en CUDA	4 días	dom 12/04/20	mié 15/04/20		Pedro
Primer análisis de rendimiento	3 días	jue 16/04/20	dom 19/04/20	30	Pedro
Documentación en overleaf	1 día	lun 20/04/20	lun 20/04/20	32	Pedro
Reunión Resumen Sprint 6	0 días	mar 21/04/20	mar 21/04/20	33	Pedro;Directores TFG

Figura 3.14: Tareas Sprint 6.

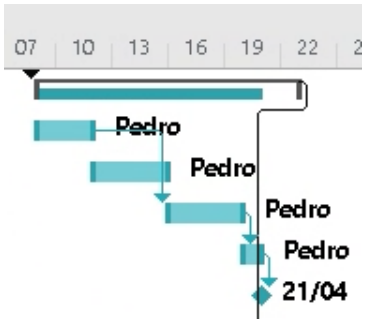


Figura 3.15: Diagrama de Gantt Sprint 6.

3.2.7 Sprint 7

Una vez que se acaba la reunión de resumen del Sprint 6 comienza el Sprint 7, el cual se corresponde con el periodo de tiempo situado entre el día 22 de abril y el día 6 de mayo, por lo que tiene una duración de 2 semanas. Durante estas dos semanas se realizan 5 tareas (contando tanto la tarea de documentación como la de resumen). En primer lugar se comienza con un estudio cuyo objetivo es ver si en el código original de Hashcat se está haciendo algún tipo de diferenciación entre quién realiza la ejecución, si un dispositivo OpenCL o uno CUDA. De esta forma sabremos si es necesario implementar algunas funciones extra propias de CUDA, que son imprescindibles para realizar una llamada a un kernel compatible con esta tecnología. En segundo lugar, se intenta mejorar la llamada a los kernels, de forma que esta aproveche al máximo las funcionalidades relacionadas con la paralelización que nos ofrece CUDA. Una vez hecho esto, se traduce un kernel de forma literal a CUDA. Cuando se acaben las tareas anteriores se lleva a cabo la documentación y la reunión de resumen del Sprint con los tutores del TFG.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Sprint 7	**10 días**	**mié 22/04/20**	**mié 06/05/20**	**29**	
Estudio de cómo hashcat distingue originalmente los dispositivos OpenCL frente a los CUDA	4 días	mié 22/04/20	sáb 25/04/20		Pedro
Implementación de una primera mejora utilizando paralelización	2 días	dom 26/04/20	lun 27/04/20	36	Pedro
Traducción de un kernel a CUDA	3 días	mar 28/04/20	jue 30/04/20	36	Pedro
Documentación en overleaf	3 días	vie 01/05/20	mar 05/05/20	38	Pedro
Reunión Resumen Sprint 7	0 días	mié 06/05/20	mié 06/05/20	39	Pedro;Directores TFG

Figura 3.16: Tareas Sprint 7.

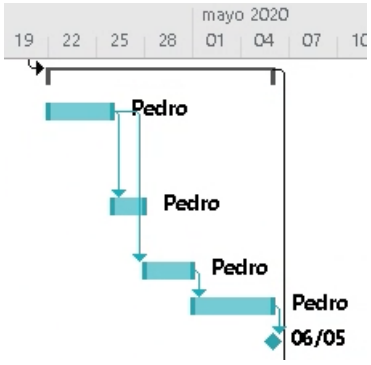


Figura 3.17: Diagrama de Gantt Sprint 7.

3.2.8 Sprint 8

El Sprint 8 comienza el día 7 de mayo de 2020 y termina el día 19 de ese mismo mes, por lo que tiene una duración de un poco menos de dos semanas. Como se puede ver en la figura 3.18, a lo largo de este periodo de tiempo se desarrollan 5 tareas. En primer lugar se hace un análisis de rendimiento en el que se comprueba si el cambio realizado en la fase anterior da como resultado una mejora en el rendimiento de la aplicación. Esto se hace comparando ejecuciones del código original frente al mejorado. A continuación se realiza un análisis de funcionamiento y rendimiento para comprobar que la mejora no sólo funciona para el kernel utilizado en el anterior análisis, sino que favorece a cualquiera de los definidos por la herramienta. Con esto, se comprueba si lo anterior está bien hecho y se corrigen los posibles errores. Una vez hecho esto se hace un nuevo estudio de la herramienta en el que se busca una forma de implementar una nueva mejora que consiga aumentar el rendimiento de la aplicación. Una vez acabada esta tarea se documenta todo el proceso de este Sprint en Overleaf y se hace la reunión final con los directores del TFG.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Sprint 8	**9 días?**	**jue 07/05/20**	**mar 19/05/20**	**35**	
Análisis de rendimiento de la mejora en el primer kernel	3 días	jue 07/05/20	dom 10/05/20		Pedro
Análisis de rendimiento para el resto de los kernels	3 días	lun 11/05/20	mié 13/05/20	42	Pedro
Estudio de Hashcat para implementar una nueva mejora	3 días	jue 14/05/20	sáb 16/05/20	43	Pedro
Documentación en Overleaf	2 días	dom 17/05/20	lun 18/05/20	44	Pedro
Reunión Resumen Sprint 8	0 días	mar 19/05/20	mar 19/05/20	45	Pedro;Directores TFG

Figura 3.18: Tareas Sprint 8.

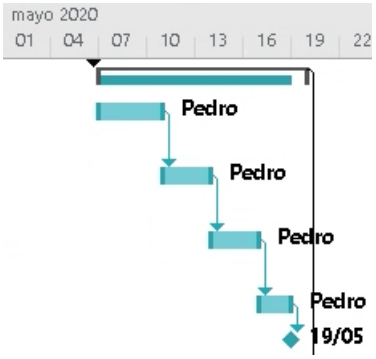


Figura 3.19: Diagrama de Gantt Sprint 8.

3.2.9 Sprint 9

Una vez revisados los objetivos del Sprint 8 se comienza con el Sprint número 9, el cual tiene una duración aproximada de dos semanas. El primer día de este periodo se corresponde con el 20 de mayo de 2020 y el último día es el 2 de junio de 2020. En primer lugar, se implementa una nueva mejora gracias al último estudio de la herramienta realizado en la fase anterior. A continuación, se efectúan una serie de pruebas para comprobar el buen funcionamiento y el posible aumento del rendimiento fruto de esta nueva mejora. En el momento en el que se haya comprobado el resultado de la última mejora, se procede a realizar un nuevo análisis de rendimiento. Este análisis consiste en ejecutar un gran número de ataques desde un dispositivo GPU con CUDA. En este lo que se hace es probar diferentes funcionalidades de la herramienta, con el objetivo de ver si la mejora afecta a toda la herramienta, o simplemente a una funcionalidad específica. Cuando finalice la prueba de rendimiento se documenta todo el proceso realizado en este Sprint en Overleaf y se lleva a cabo la reunión resumen correspondiente a este periodo de tiempo.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Sprint 9	**10 días?**	**mié 20/05/20**	**mar 02/06/20**	**41**	
Implementación de la nueva mejora	3 días	mié 20/05/20	vie 22/05/20		Pedro
Análisis de rendimiento de la nueva mejora	2 días	sáb 23/05/20	lun 25/05/20	48	Pedro
Análisis total de la herramienta para una GPU concreta	5 días	mar 26/05/20	sáb 30/05/20	49	Pedro
Documentación en Overleaf	1 día	lun 01/06/20	lun 01/06/20	50	Pedro
Reunión Resumen Sprint 9	0 días	mar 02/06/20	mar 02/06/20	51	Pedro;Directores TFG

Figura 3.20: Tareas Sprint 9.

Figura 3.21: Diagrama de Gantt Sprint 9.

3.2.10 Sprint 10

El día 3 de junio de 2020 comienza el último Sprint de este proyecto, el cual va a terminar el día 16 del mismo mes, con lo que tiene una duración de dos semanas. A lo largo de este Sprint se definen 5 tareas. En primer lugar, se hace un nuevo estudio del rendimiento de Hashcat, pero esta vez desde una plataforma distinta. Esto se hace debido a que todos los análisis anteriores se han hecho desde el mismo dispositivo GPU, por lo que en este punto es importante saber si la mejora solo se da en ese caso o si también afecta al resto de GPUs que soportan CUDA. A continuación, se elaboran las conclusiones finales del trabajo y se documentan en Overleaf. En este punto, se realiza una reunión con los directores del TFG, para ver si se han cumplido los objetivos generales del proyecto. A su vez, se hacen correcciones que se resuelven en la siguiente tarea. Por último, se hace una última reunión con los directores para finalizar el proyecto y comprobar que las correcciones han sido solucionadas. Con este último periodo de dos semanas se acaba el proyecto y se prepara todo lo necesario para llevar a cabo la entrega del mismo.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Sprint 10	**10 días?**	**mié 03/06/20**	**mar 16/06/20**	**47**	
Análisis total de la herramienta con una segunda GPU	4 días	mié 03/06/20	dom 07/06/20		Pedro
Elaboración de las conclusiones del trabajo	4 días	lun 08/06/20	jue 11/06/20	54	Pedro
Reunión con los directores del TFG	0 días	vie 12/06/20	vie 12/06/20	55	Pedro;Directores TFG
Corrección de fallos	2 días	sáb 13/06/20	lun 15/06/20	56	Pedro
Reunión final del proyecto	0 días	mar 16/06/20	mar 16/06/20	57	Pedro;Directores TFG

Figura 3.22: Tareas Sprint 10.

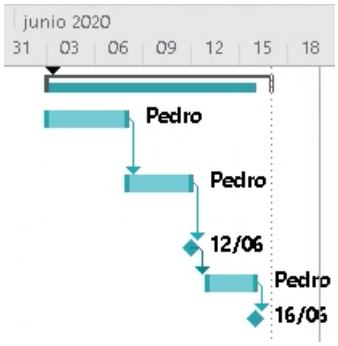


Figura 3.23: Diagrama de Gantt Sprint 10.

3.3 Coste del proyecto

A lo largo de esta sección se explican las estimaciones del coste del proyecto, teniendo en cuenta tanto los costes humanos como los costes de los distintos recursos utilizados a lo largo de todo el proceso.

En primer lugar, comenzamos explicando y desglosando los costes humanos. Para ello vamos a tener en cuenta que se ha trabajado con un programador, con un sueldo aproximado de 25€/hora y con dos directores de proyecto, uno de ellos catedrático y otro titular. El sueldo medio del director titular es de aproximadamente 35€/hora y el del catedrático de 40€/hora. Para representar el resultado de forma clara se ha creado la tabla 3.1, la cual recoge el coste humano total de este proyecto. Las horas totales trabajadas por parte de cada una de las personas implicadas en este trabajo han sido extraídas de la planificación.

	€/hora	**Horas trabajadas**	**Coste total**
Programador	25	480	12000€
Director titulado	35	20	700€
Director catedrático	40	10	400€
Total	-	**510**	**13100€**

Tabla 3.1: Costes humanos del proyecto.

En segundo lugar, es importante tener en cuenta los costes indirectos del proyecto, relacionados con los equipos y las herramientas utilizadas. Para calcular el coste de los equipos se hará una operación muy sencilla, en la que a partir del precio de estos y del tiempo en el que han sido utilizadon, se calculará el coste de los mismos. El tiempo de uso del ordenador portátil ha sido de 4 meses, mientras que el tiempo de uso del clúster ha sido de 1 mes. El coste de haber utilizado el portátil es de 66.67€, ya que su precio original es de 800€, su tiempo medio de vida es de 4 años y su uso en el proyecto ha sido de 4 meses (1/3 * 800€/4años = 66.67). Por otro lado, el coste de haber utilizado el clúster es también de 66.67€, ya que el precio del nodo utilizado es de 4000€ aproximadamente, su tiempo de vida es de 5 años y su tiempo de uso en el proyecto es de 1 mes (1/12 * 4000€/5años = 66.67).

Por último, si no tuviésemos acceso a la herramienta de Microsoft Project mediante las máquinas de la universidad, su coste durante los 4 meses que se utilizó sería de 101.2€, ya que su precio mensual es de 25.30€.

Con todos los datos explicados anteriormente, obtenemos la tabla 3.2, en la que se ve reflejado el coste total del proyecto, el cual es igual a **13334,54€**.

	Euros
Costes humanos	13100
Costes indirectos	234,54
Total	**13334,54**

Tabla 3.2: Costes totales del proyecto.

.

Capítulo 4

Desarrollo

Como ya se ha explicado anteriormente, a lo largo de la elaboración de este proyecto se han seguido una serie de fases: estudio de la aplicación Hashcat, aprendizaje de CUDA (explicada en la sección 2.3), estudio del soporte actual para CUDA con vistas a mejorarlo, si fuera posible, y estudio del rendimiento. Este apartado explica de forma detallada tanto la fase de estudio de la herramienta, como la de implementación y mejora de CUDA.

En la sección 4.1, se realiza un estudio a fondo de Hashcat, con el objetivo de localizar qué partes implementar en CUDA y qué partes del código mejorar, además de conocer con detalle el funcionamiento de la herramienta. Para realizar dicho estudio ha sido necesario realizar un proceso de ingeniería inversa, debido a que la documentación de Hashcat para desarrolladores de la herramienta es bastante deficiente, y tampoco se ha tenido acceso directo al equipo de desarrollo de la aplicación. La sección 4.2 explica fase a fase todo el proceso de implementación y mejora de CUDA en la herramienta y los distintos cambios realizados en el código.

4.1 Estudio de la aplicación Hashcat

En primer lugar, es necesario llevar a cabo un estudio a fondo de la aplicación para conocer tanto sus funcionalidades, como sus usos más comunes y sus características, y en un paso posterior ver cómo funciona por dentro y cómo está desarrollado su código. Para comenzar se busca información y se crea un informe recogido en la sección 2.1. Este estudio se realiza siguiendo la documentación de la página oficial de Hashcat [15] y contrastando información de su wiki [7], de su repositorio en Github [6], y de la última versión de la aplicación, ya que hay partes que no están actualizadas en la documentación pero sí en la herramienta. Una vez hecho este estudio inicial, se considera que, para conocer Hashcat más a fondo, lo más adecuado es realizar una serie de pruebas en las que se usen varias funcionalidades de las que nos ofrece. Para ello, el primer paso es instalar la herramienta compilando el código fuente de

su repositorio GitHub oficial [6]. Una vez hecho esto, con ayuda del manual de la aplicación y de lo estudiado en el paso anterior, se realizan una serie de ejecuciones de ejemplo entre las que se destacan las siguientes:

Ataque por fuerza bruta a un hash en concreto: Para este caso se puede observar el comando utilizado en el listado 4.1, y posteriormente, su resultado en la figura 4.1. Esta ejecución se corresponde con un ataque por fuerza bruta (opción -a 3) al hash 502ff82f7f1f8218dd41201fe4353687 en md5 (opción -m 0). En este caso sabemos que la contraseña es de 4 caracteres, que pueden ser letras en minúsculas o mayúsculas, números, caracteres en hexadecimal en minúsculas o mayúsculas y símbolos (opción ?a?a?a?a). En la mayor parte de las ejecuciones realizadas para hacer ataques se utiliza la opción –force. Esto se hace debido a que en la última versión de la herramienta si no se utiliza esta opción en algunos dispositivos se produce a un error de ejecución. Este error lo causa el driver de OpenCL, por lo que Hashcat nos avisa de que la compilación del kernel puede producir falsos negativos y que es recomendable utilizar dicha opción. La opción -d1 hace referencia al dispositivo que queremos utilizar en esta ejecución, el cual en este caso es el número 1, correspondiente a un dispositivo concreto de tipo OpenCL. Hashcat también dispone de una opción para escoger entre dispositivos de tipo CPU (-D1) y dispositivos de tipo GPU (-D2).

```
1 ./hashcat -m 0 -a 3 502ff82f7f1f8218dd41201fe4353687 ?a?a?a?a
    --force -d1 -D2
```

Listado 4.1: Ataque por fuerza bruta.

En la figura 4.1 podemos observar que el ataque por fuerza bruta ha sido un éxito, ya que nos muestra que el hash indicado pertenece a la palabra ”luis”.

```
Watchdog: Temperature abort trigger set to 90c

Host memory required for this attack: 151 MB

502ff82f7f1f8218dd41201fe4353687:luis

Session..........: hashcat
Status...........: Cracked
Hash.Name........: MD5
Hash.Target......: 502ff82f7f1f8218dd41201fe4353687
Time.Started.....: Sat May  2 13:42:42 2020, (0 secs)
Time.Estimated...: Sat May  2 13:42:42 2020, (0 secs)
Guess.Mask.......: ?a?a?a?a [4]
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  1448.9 MH/s (11.94ms) @ Accel:64 Loops:95 Thr:1024 Vec:1
Recovered........: 1/1 (100.00%) Digests
Progress.........: 31129600/81450625 (38.22%)
Rejected.........: 0/31129600 (0.00%)
Restore.Point....: 0/857375 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-95 Iteration:0-95
Candidates.#1....: sari ->  p8x
Hardware.Mon.#1..: Temp: 52c Util: 18% Core:1784MHz Mem:3504MHz Bus:4

Started: Sat May  2 13:42:41 2020
Stopped: Sat May  2 13:42:43 2020
```

Figura 4.1: Ataque de fuerza bruta con Hashcat.

Ataque por diccionario: En segundo lugar efectuamos un ataque por diccionario basado en reglas, en el cual se utiliza el diccionario rockyou.txt. Este ataque ya no va dirigido a un único hash, sino que en la ejecución se pasa como entrada un fichero en el que hay un gran número de hashes, por lo que si el ataque es correcto, en la salida deberíamos encontrar un gran número de hashes y a continuación la palabra a la que pertenecen. El comando utilizado es el que se muestra en el listado 4.2. En este caso se utilizan nuevas opciones que se explican a continuación. En primer lugar se usa el parámetro -O, para que la herramienta utilice los kernels de tipo optimizado y la ejecución sea más rápida. Dichos kernels de tipo optimizado sirven para que el ataque se realice de forma más rápida, pero tienen la desventaja de que no se pueden utilizar para contraseña de más de 256 caracteres. La opción -m 0 sirve para indicar que los hashes son de tipo MD5 y la opción -r para añadir un fichero en el que se definen una serie de reglas que se aplican a cada palabra del diccionario al hacer el ataque. Además, como se ha comentado anteriormente, no se indica en el comando el hash que queremos resolver, sino que se le pasa una ruta en la cual se encuentra un fichero con un gran número de hashes (../bfield.hash/bfield.hash).

```
./hashcat -O -m 0 ../bfield.hash/bfield.hash ../rockyou.txt -r
    rules/rockyou-30000.rule --force
```

Listado 4.2: Ataque por diccionario.

La salida del comando anterior se puede ver en la figura 4.2, en la cual observamos una parte de los pares hash-contraseña que se han obtenido y un resumen de la ejecución.

```
7baade55f8a221879a7deb4603bf6198:01029dd4
7362090312bced282c6f685765f69012:0127101877
7a171fe4ed1a1c8712cf06a5bec36c8e:2299667733
1fd76b0d781e725a368ea64e19f7ca09:o0435533
5c6370ba3a5253a0d7964e7cab23db62:2428561230

Session..........: hashcat
Status...........: Exhausted
Hash.Name........: MD5
Hash.Target......: ../bfield.hash/bfield.hash
Time.Started.....: Sat May  2 17:46:30 2020, (7 mins, 21 secs)
Time.Estimated...: Sat May  2 17:53:51 2020, (0 secs)
Guess.Base.......: File (../rockyou.txt)
Guess.Mod........: Rules (rules/rockyou-30000.rule)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:   977.8 MH/s (3.10ms) @ Accel:64 Loops:256 Thr:1024 Vec:1
Speed.#2.........: 53548.9 kH/s (459.68ms) @ Accel:512 Loops:256 Thr:8 Vec:4
Speed.#*.........:  1031.4 MH/s
Recovered........: 295342/423623 (69.72%) Digests
Remaining........: 128281 (30.28%) Digests
Recovered/Time...: CUR:2159,N/A,N/A AVG:40215,2412928,57910279 (Min,Hour,Day)
Progress.........: 430331520000/430331520000 (100.00%)
Rejected.........: 92820000/430331520000 (0.02%)
Restore.Point....: 12913472/14344384 (90.02%)
Restore.Sub.#1...: Salt:0 Amplifier:29952-30000 Iteration:0-256
Restore.Sub.#2...: Salt:0 Amplifier:29952-30000 Iteration:0-256
Candidates.#1....: $HEX[6262756464793836] -> $HEX[042a0369c2a156616d6f7321033832]
Candidates.#2....: b412159286 -> 2121156152482
Hardware.Mon.#1..: Temp: 85c Util: 40% Core:1695MHz Mem:3504MHz Bus:4
Hardware.Mon.#2..: N/A

Started: Sat May  2 17:46:20 2020
Stopped: Sat May  2 17:53:52 2020
```

Figura 4.2: Ataque por diccionario con Hashcat.

Benchmark de la herramienta: Por último, es importante tener en cuenta la opción que nos ofrece Hashcat relacionada con la ejecución de un benchmark en el que podemos hacer una prueba de todos los hashes disponibles para un ataque de tipo fuerza bruta. Un benchmark es una prueba de una aplicación en la que se mide el rendimiento del sistema completo o de alguno de sus componentes. Hashcat nos da la opción de correr un benchmark genérico de toda la herramienta, que nos muestra los tiempos de cada tipo de hash, pero también nos permite ejecutar el benchmark de un algoritmo de hash en concreto, por lo que si estamos trabajando con uno en específico esta opción puede ser muy útil. El benchmark que ofrece Hashcat permite comprobar el rendimiento para cada tipo hash, realizando un ataque por fuerza bruta desde cada dispositivo disponible. Además, por defecto utiliza los kernels de tipo optimizado. En este caso se muestra un ejemplo de la ejecución del benchmark completo de la herramienta, en el que se van mostrando uno a uno todos los tiempos de ejecución de cada tipo de hash. El comando a utilizar en este caso es el mostrado en el listado 4.3. La opción -b le indica a la aplicación que queremos correr el benchmark de la herramienta.

```
1 ./hashcat -b --force
```

Listado 4.3: Benchmark de la herramienta.

Una parte de la salida se puede ver en la figura 4.3, en la que, como se ha comentado anteriormente, se va indicando para cada tipo de algoritmo de hash el tiempo que tardó el benchmark en ejecutarse. Además esta ejecución la repite para todos los dispositivos disponibles en el equipo en el que se ejecute la aplicación.

```
Benchmark relevant options:
===========================
* --force
* --optimized-kernel-enable

Hashmode: 0 - MD5

Speed.#1.........:  6195.4 MH/s (53.68ms) @ Accel:64 Loops:1024 Thr:1024 Vec:8
Speed.#2.........:   439.8 MH/s (56.47ms) @ Accel:512 Loops:256 Thr:8 Vec:4
Speed.#*.........:  6635.3 MH/s

Hashmode: 100 - SHA1

Speed.#1.........:  2036.8 MH/s (81.96ms) @ Accel:32 Loops:1024 Thr:1024 Vec:1
Speed.#2.........:   138.7 MH/s (89.70ms) @ Accel:256 Loops:256 Thr:8 Vec:4
Speed.#*.........:  2175.5 MH/s

Hashmode: 1400 - SHA2-256

Speed.#1.........:   749.4 MH/s (55.56ms) @ Accel:8 Loops:1024 Thr:1024 Vec:1
Speed.#2.........: 55819.2 kH/s (55.82ms) @ Accel:32 Loops:512 Thr:8 Vec:4
Speed.#*.........:   805.2 MH/s
```

Figura 4.3: Ejecución del benchmark de Hashcat.

A continuación, usamos herramientas de depuración y profiling para entender el funcionamiento interno de la aplicación y analizar el rendimiento de sus distintos componentes.

Es importante explicar que los diferentes análisis que se realizan se hacen sobre ejecuciones del benchmark de Hashcat, ya que si se hacen sobre una ejecución de un ataque en concreto se deja de analizar buena parte del código. Esto no ocurre si se analiza la ejecución del benchmark, ya que este está creado para probar todos los modos que ofrece la herramienta, por lo que se va a cubrir una gran parte de todo el código de la aplicación. Las herramientas de profiling utilizadas para realizar este trabajo son Gprof e Intel Vtune.

Para comenzar, se trabaja primero con la herramienta Gprof, la cual es muy sencilla de entender y de ejecutar. Para poder utilizarla en linux se puede instalar directamente mediante apt y para usarla en un código determinado es necesario, en primer lugar, cambiar la forma en la que se compila, ya que hay que añadir una opción específica de Gprof. Para compilar el código con dicha opción es necesario editar el Makefile de Hashcat, que se encuentra en la carpeta src y simplemente añadir la opción "-pg". De esta forma cuando volvamos a compilarlo se utiliza dicha opción y a partir de ese momento se crea un archivo llamado gmon.out cada vez que ejecutemos Hashcat. Este archivo se trata de un fichero que contiene información sobre la ejecución del comando que se quiere analizar, el cual se puede procesar con gprof para generar el informe de profiling. La forma en la que procesar el fichero gmon.out para obtener el informe en un formato cómodo y fácil de visualizar es ejecutar el comando mostrado en el listado 4.4.

```
1 gprof hashcat gmon.out > out_gprof.txt
```

Listado 4.4: Obtención del informe en formato texto.

Lo que se consigue con el comando anterior es generar un informe .txt de la última ejecución de la aplicación, el cual nos va a dar mucha información del rendimiento de los distintos componentes de la herramienta. Si abrimos el archivo .txt generado se puede ver que lo primero que tenemos es una tabla con cada una de las funciones utilizadas por Hashcat, en la que se muestra el tiempo de CPU consumido por cada una de ellas, el porcentaje de uso de CPU de cada función con respecto al resto e información acerca de las llamadas que se hacen a cada una de ellas. Esta información es muy útil para comprobar cuál es la parte del código en la que pasa más tiempo el dispositivo que lleve a cabo la ejecución y con esto podemos saber qué parte de la herramienta tenemos que modificar para mejorar el rendimiento. En este caso y como ya se ha explicado anteriormente se está trabajando con una ejecución del benchmark de la aplicación, con lo cual el resultado obtenido es el que mejor refleja un uso general de Hashcat.

La tabla 4.1 contiene una lista de las funciones de la aplicación que más tiempo consumen CPU, junto con una breve descripción de cada una de ellas.

Función	% CPU	Descripción de la función
LzmaDec_DecodeReal2	31.18%	Tareas de compresión y decomprensión sin pérdidas
sp_setup_tbl	29.98%	Tareas de inicialización de Hashcat
sp_comp_val	19.52%	Tareas de comparación de valores
sp_tbl_to_css	8.33%	Tareas de conversión de tablas de Hashcat a ccs
byte_swap_64	4.19%	Tareas de intercambio de Bytes

Tabla 4.1: Análisis Gprof. Funciones más usadas

A primera vista podemos ver que estas funciones no están relacionadas directamente con las funcionalidades esenciales de Hashcat, sino que realizan tareas auxiliares, como puede ser la compresión y decompresión o la inicialización de la aplicación. Esto nos indica que es necesario realizar un análisis con otra herramienta de este tipo, la cual nos proporcione unos resultados más completos en los que podamos desglosar cada función de forma más visual.

Una vez que se nos muestran para todas las funciones utilizadas sus datos correspondientes, lo siguiente que aparece es un árbol de llamadas, es decir, otra tabla que refleja la forma en la que unas funciones van llamando a otras, indicando además cuantas veces ha sido llamada cada una de ellas. Esta parte del informe se puede ver resumida en la tabla 4.2, en la cual se muestran los datos más importantes del mismo. Dicha tabla es muy importante debido a que, gracias a ella, se puede ver para cada función quién la llamó y a quién llamó esta, además del número de veces que lo hizo, por lo que nos puede ayudar a tener una primera idea de la forma en la que Hashcat va llamando a las distintas funciones hasta llegar a los kernels. Con este resumen se puede comprobar cómo va avanzando el árbol de funciones en el inicio de la ejecución. La primera función en ejecutarse es el main() de la herramienta, el cual llama a hashcat_session_execute() y esta última a outer_loop(). Si seguimos observando este informe y continuamos por el árbol de llamadas podemos observar que finalmente llegamos a la función generate_source_kernel_filename(), que es la encargada de escoger qué fichero de kernel se va a llamar en un futuro.

Función	% Tiempo	Función que la llama
hashcat_session_execute	74.5%	main
outer_loop	74.5%	hashcat_session_execute
main	74.5%	-
mask_ctx_init	62.6%	outer_loop
sp_setup_tbl	62.6%	mask_ctx_init

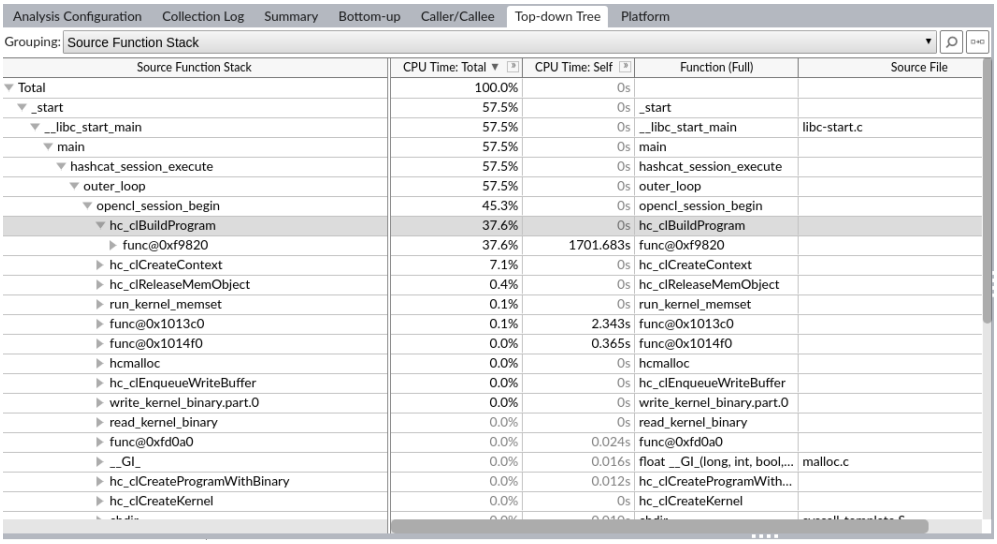
Tabla 4.2: Análisis Gprof. Árbol de llamadas en el inicio de la ejecución.

Como complemento al uso de Gprof, el profiling de la aplicación lo hemos enriquecido con el uso de la herramienta Intel Vtune.

Para poder utilizar Intel Vtune es necesario descargarla desde la página oficial [11], en la cual existe una licencia para estudiante a la que tenemos acceso desde la cuenta de la UDC. Una vez que tenemos la herramienta podemos optar por utilizar la versión del terminal o una versión con interfaz gráfica. En un principio se realiza el análisis de la misma ejecución del benchmark de la herramienta que en el caso de Gprof y, a continuación, se analizan otras ejecuciones para comparar si en todos los casos se obtienen conclusiones parecidas.

Para comenzar y obtener un resultado más claro y visual se realizan los primeros análisis con la interfaz gráfica de Vtune. Por otro lado, para que esta ejecución fuese exactamente la misma que la de Gprof lo que se hace es ejecutar el benchmark directamente desde Intel Vtune, ya que de esta forma también se genera el archivo gmon.out que se ha utilizado en el análisis de Gprof.

En un principio se utilizan las pestañas de ”Caller/Callee” y ”Top-down Tree” para ver el árbol de llamadas de la ejecución de Hashcat y el porcentaje de uso de CPU de cada función, y la pestaña ”Platform” para ver el nivel de paralelismo. De esta forma se puede comparar el resultado de ambas herramientas y llegar a una conclusión. Una vez estudiado el resultado proporcionado por Intel-Vtune se comprueba que este es muy parecido al que nos ofrece Gprof, ya que se puede ver que las funciones que más porcentaje de CPU utilizan son las mismas que en el caso anterior (LzmaDec_DecodeReal2, sp_setup_tbl, sp_comp_val, sp_tbl_to_css y byte_swap_64). Indagando un poco más en el árbol de llamadas se observa cómo las funciones que de verdad utilizan más porcentaje de CPU son funciones genéricas como pueden ser hc_clBuildProgram, run_kernel, hc_clWaitForEvents, hc_clEnqueueNDRangeKernel, ..., es decir, funciones que no son propias del funcionamiento de Hashcat, desempeñan tareas como cargar el kernel, construir el programa, etc. Esto se puede comprobar en la figura 4.4.



Source Function Stack	CPU Time: Total	CPU Time: Self	Function (Full)	Source File
Total	100.0%	0s		
_start	57.5%	0s	_start	
__libc_start_main	57.5%	0s	__libc_start_main	libc-start.c
main	57.5%	0s	main	
hashcat_session_execute	57.5%	0s	hashcat_session_execute	
outer_loop	57.5%	0s	outer_loop	
opencl_session_begin	45.3%	0s	opencl_session_begin	
hc_clBuildProgram	37.6%	0s	hc_clBuildProgram	
func@0xf9820	37.6%	1701.683s	func@0xf9820	
hc_clCreateContext	7.1%	0s	hc_clCreateContext	
hc_clReleaseMemObject	0.4%	0s	hc_clReleaseMemObject	
run_kernel_memset	0.1%	0s	run_kernel_memset	
func@0x1013c0	0.1%	2.343s	func@0x1013c0	
func@0x1014f0	0.0%	0.365s	func@0x1014f0	
hcmalloc	0.0%	0s	hcmalloc	
hc_clEnqueueWriteBuffer	0.0%	0s	hc_clEnqueueWriteBuffer	
write_kernel_binary.part.0	0.0%	0s	write_kernel_binary.part.0	
read_kernel_binary	0.0%	0s	read_kernel_binary	
func@0xfd0a0	0.0%	0.024s	func@0xfd0a0	
__GI_	0.0%	0.016s	float __GI_(long, int, bool,...	malloc.c
hc_clCreateProgramWithBinary	0.0%	0.012s	hc_clCreateProgramWith...	
hc_clCreateKernel	0.0%	0s	hc_clCreateKernel	

Figura 4.4: Resultado Intel-Vtune. Árbol de llamadas

Una vez realizados estos primeros análisis se llega a una primera conclusión y es que las funciones que más porcentaje de CPU utilizan son aquellas que están relacionadas con llamadas al kernel, inicialización del kernel, etc., por lo que podemos ir deduciendo que es muy probable que lo que más beneficios nos va a dar en un futuro es mejorar el proceso de llamada a los kernels. Con esto, además, se llega a la conclusión de que es necesario un estudio más a fondo del funcionamiento de la aplicación. Como ya se ha comentado al inicio de este capítulo, en este punto es necesario llevar a cabo un proceso de ingeniería inversa, debido a que la documentación de Hashcat es muy escasa y no se ha tenido acceso directo al equipo de desarrollo de la aplicación.

Mediante el uso de puntos de parada (breakpoints) utilizando el modo DEBUG y el estudio de la guía de desarrolladores de Hashcat [16], se puede comprobar que lo que ocurre en la herramienta es que, según el tipo de algoritmo de hash al que se quiera atacar, Hashcat utiliza un kernel u otro. Dichos kernels se encuentran en la carpeta OpenCL y todos ellos son archivos .cl, lo que quiere decir que son archivos propios de OpenCL. Además, siguiendo la guía y el repositorio se puede observar que para cada tipo de función hash existen por regla general 6 kernels distintos y el formato de estos archivos es el siguiente: OpenCL/mXXXXX_a[0|1|3]-[pure|optimized].cl, siendo 0, 1 y 3 distintos tipos de ataques, mientras que pure y optimized son las formas en las que se puede optimizar (o no) el código de un kernel. Los de tipo puro son aquellos que se utilizan para contraseñas de más de 256 caracteres de longitud, por lo que en la mayor parte de los casos se utiliza el de tipo optimized. Es importante aclarar que solo hay kernels que se corresponden con los ataques de tipo 0 (Straight), 1 (Combination) y 3 (Brute-force), ya que los ataques de tipo 6 (Hybrid Wordlist + Mask) y 7 (Hybrid Mask + Wordlist) utilizan el mismo fichero de kernel que los de tipo 1 [16]. En la figura 4.5 se puede ver un ejemplo de uso del modo DEBUG, en el cual se ve el nombre del kernel al que se está llamando (m00000_s04).

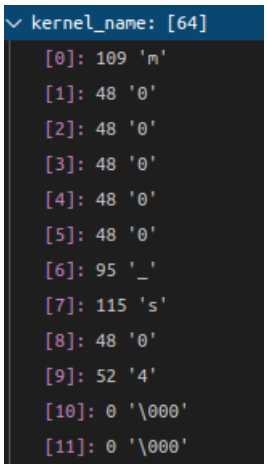


Figura 4.5: Nombre del kernel utilizando GDB.

Una vez entendida la forma en la que Hashcat clasifica los kernels, se ha comprobado que, a pesar de que todos los ficheros de kernel tienen extensión .cl (propia de OpenCL), la herramienta es compatible con el uso de la tecnología CUDA, ya que a lo largo de todo el código se puede observar la existencia de distintas funciones del API de CUDA. Dichas funciones son propias de tareas como llamar a una función del kernel determinada, esperar a que un evento se complete, reservar memoria, liberar memoria, etc, como pueden ser las funciones cuEventRecord(), cuLaunchKernel(), cuEventSynchronize(), cuMemAlloc(), cuMemFree(), cuMemcpyDtoH() , etc. Algún ejemplo del código de Hashcat en el que se observan funciones propias de CUDA puede verse en la figura 4.6.

```
if (device_param->is_cuda == true)
{
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_bitmap_s1_a,    bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_bitmap_s1_b,    bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_bitmap_s1_c,    bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_bitmap_s1_d,    bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_bitmap_s2_a,    bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_bitmap_s2_b,    bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_bitmap_s2_c,    bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_bitmap_s2_d,    bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_plain_bufs,     size_plains)             == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_digests_buf,    size_digests)            == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_digests_shown,  size_shown)              == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_salt_bufs,      size_salts)              == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_result,         size_results)            == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_extra0_buf,     size_extra_buffer / 4)   == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_extra1_buf,     size_extra_buffer / 4)   == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_extra2_buf,     size_extra_buffer / 4)   == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_extra3_buf,     size_extra_buffer / 4)   == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_st_digests_buf, size_st_digests)         == -1) return -1;
  if (hc_cuMemAlloc (hashcat_ctx, &device_param->cuda_d_st_salts_buf,   size_st_salts)           == -1) return -1;

  if (hc_cuMemcpyHtoD (hashcat_ctx, device_param->cuda_d_bitmap_s1_a, bitmap_ctx->bitmap_s1_a, bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemcpyHtoD (hashcat_ctx, device_param->cuda_d_bitmap_s1_b, bitmap_ctx->bitmap_s1_b, bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemcpyHtoD (hashcat_ctx, device_param->cuda_d_bitmap_s1_c, bitmap_ctx->bitmap_s1_c, bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemcpyHtoD (hashcat_ctx, device_param->cuda_d_bitmap_s1_d, bitmap_ctx->bitmap_s1_d, bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemcpyHtoD (hashcat_ctx, device_param->cuda_d_bitmap_s2_a, bitmap_ctx->bitmap_s2_a, bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemcpyHtoD (hashcat_ctx, device_param->cuda_d_bitmap_s2_b, bitmap_ctx->bitmap_s2_b, bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemcpyHtoD (hashcat_ctx, device_param->cuda_d_bitmap_s2_c, bitmap_ctx->bitmap_s2_c, bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemcpyHtoD (hashcat_ctx, device_param->cuda_d_bitmap_s2_d, bitmap_ctx->bitmap_s2_d, bitmap_ctx->bitmap_size) == -1) return -1;
  if (hc_cuMemcpyHtoD (hashcat_ctx, device_param->cuda_d_digests_buf, hashes->digests_buf,     size_digests)            == -1) return -1;
  if (hc_cuMemcpyHtoD (hashcat_ctx, device_param->cuda_d_salt_bufs,   hashes->salts_buf,       size_salts)              == -1) return -1;
```

Figura 4.6: Ejemplo de alguna función propia de CUDA en el código de Hashcat.

.

En este punto se comprueba mediante el modo DEBUG, que al ejecutar la herramienta desde un dispositivo que utilice la tecnología de CUDA, se está haciendo uso de las funciones propias del API de CUDA y de los kernels de tipo OpenCL. Esto es sorprendente, ya que no es posible llamar a un kernel de tipo OpenCL mediante funciones propias de CUDA, además desde un principio se pensaba que esto no ocurría, ya que en la documentación de la herramienta se explica que todo el código ha sido portado de CUDA a OpenCL [17]. La única opción posible para que esto ocurra es que cada fichero de kernel esté pensado y desarrollado de forma que se pueda ejecutar desde CUDA haciendo algunos cambios en él en tiempo de ejecución.

Para conocer la forma en la que Hashcat llama a kernels de tipo OpenCL desde código CUDA, se realiza un estudio acerca del código de uno de los kernels propios de Hashcat. En este caso se utiliza el archivo llamado m01400_a3-optimized.cl situado en la carpeta OpenCL.

En este se ve que en algunas partes de su código se utilizan funciones propias de OpenCL, como puede ser el caso de get_global_id() o get_local_id(). Esto se puede comprobar en la figura 4.7.

```
37   const u64 gid = get_global_id (0);
38   const u64 lid = get_local_id (0);
39
```

Figura 4.7: Código OpenCL en el fichero del kernel.

Estas funciones nos permiten obtener el elemento actual de trabajo, tanto local como global. Lo que ocurre en este caso es que al ser código propio de OpenCL, no se puede llamar desde la función cuLaunchKernel(), debido a que, al ser esta propia de CUDA, no es capaz de ejecutar código perteneciente a OpenCL. Por otro lado, en el API de CUDA también existe una forma mediante la cual obtener el elemento de trabajo actual. El código análogo de CUDA para estas funciones se puede observar en el listado 4.5.

```
1 blockIdx.x * blockDim.x + threadIdx.x
```

Listado 4.5: Análogo de la función get_global_id() de OpenCL en CUDA.

La traducción a CUDA del caso anterior, el cual se corresponde con la función de OpenCL ”get_global_id()”, se puede verificar en la tabla 4.3, en la cual vemos cómo se hace la traducción de este tipo de funciones entre CUDA y OpenCL.

CUDA	OpenCL
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id()
blockIdx * blockDim + threadIdx	get_global_id()
gridDim * blockDim	get_global_size()

Tabla 4.3: Tabla para traducir funciones propias de OpenCL a CUDA.

Es este punto en el que se comprueba que los kernels utilizan funciones propias de OpenCL para hacer determinadas tareas. Con esto, se llega a la conclusión de que, para que CUDA sea capaz de llamar a las funciones de los kernels, es necesario que en algún punto del código de Hashcat se redefinan dichas funciones propias de OpenCL. Una vez más, mediante el uso del modo DEBUG, se ha llegado a dicha parte del código, en la cual se está definiendo, entre otras, la función get_global_id() para los casos en los que la herramienta se ejecute utilizando la tecnología CUDA. Esta definición la podemos ver en la figura 4.8.

```
DECLSPEC size_t get_global_id  (const u32 dimindx __attribute__((unused)))
{
  return (blockIdx.x * blockDim.x) + threadIdx.x;
}

DECLSPEC size_t get_local_id (const u32 dimindx __attribute__((unused)))
{
  return threadIdx.x;
}
```

Figura 4.8: Definición de funciones usadas en el kernel para que sean compatibles con CUDA.

Esta definición se encuentra en el fichero inc_platform.cl, el cual está situado dentro de la carpeta OpenCL. A su vez, todas estas funciones que necesitan una redefinición para que puedan ser llamadas desde CUDA, se encuentran dentro de un if por el que sólo se pasa en el caso en el que se esté utilizando la tecnología de CUDA. Por otro lado, si la aplicación está siendo ejecutada con la tecnología de OpenCL, no se llevan a cabo estas definiciones que afectan al código del kernel, y como consecuencia se utilizan las funciones propias de OpenCL.

En este punto del trabajo se concluye con la parte relacionada con el estudio de la herramienta Hashcat, obteniendo como resultado una conclusión muy diferente a la que se pensaba en un inicio. A pesar de que en un principio se creyese que Hashcat había portado todo su código CUDA a OpenCL, como se explica en sus foros, esto no es así. Lo que está ocurriendo es que se le está dando una mayor importancia a OpenCL, como se puede ver en sus propios kernels. Con esta conclusión nos damos cuenta de que, al ser una herramienta más centrada en OpenCL que en CUDA, es muy probable que no se estén aprovechando al máximo todas las funcionalidades que ofrece esta última tecnología. Esto se comprobará en la sección 4.2.

4.2 Implementación en CUDA

Una vez que se estudia a fondo la herramienta y se realiza la fase de aprendizaje de CUDA, es momento de comenzar la fase de implementación y mejora de dicha tecnología. Esta etapa consiste en un pequeño estudio de los kernels de OpenCL y su compatibilidad con CUDA, y en una serie de modificaciones en el código de la herramienta con las que se consigue variar el reparto de trabajo de la misma.

En primer lugar se ha comprobado que los kernels que utiliza Hashcat son muy genéricos, por lo que haciendo alguna modificación en tiempo de ejecución se consigue que sean compatibles tanto para la tecnología de OpenCL como para CUDA. Como se ha explicado anteriormente, lo que se hace para que ambas tecnologías sean compatibles es redefinir algunas funciones utilizadas en los kernels en tiempo de ejecución, utilizando definiciones propias de CUDA cuando se utiliza un dispositivo CUDA. Al hacer estos cambios en tiempo de ejecución es posible que se pierda un poco de rendimiento.

Para comprobar que esta pérdida es mínima y que no es necesario tenerla en cuenta, se

traduce un kernel sencillo de OpenCL a CUDA y se hace una prueba de rendimiento. De esta forma, en vez de realizar cambios en el kernel de OpenCL en tiempo de ejecución, se usa un kernel con extensión .cu que es la traducción literal del kernel con extensión .cl. Lo que se hace para traducir ese kernel de forma literal a CUDA es sustituir las funciones que se redefinen en tiempo de ejecución por funciones propias de CUDA. Esta prueba está reflejada en la sección 5.1 y da como resultado la tabla 4.4, correspondiendo la primera fila con la ejecución del código original de Hashcat desde una CPU OpenCL y la segunda fila con esa misma ejecución, pero desde una GPU con OpenCL. Las dos últimas filas pertenecen a ejecuciones realizadas desde una GPU utilizando la tecnología de CUDA. En primer lugar se ejecuta la herramienta con el kernel original escrito en un archivo con extensión .cl, y en segundo lugar se hace la misma ejecución pero utilizando el kernel con extensión .cu traducido por nosotros. Todas las especificaciones de los dispositivos utilizados se nombran en el capítulo 5. Para entender mejor los datos de la tabla es importante tener en cuenta que, MH/s significa un millón de hashes probados por segundo y que dichos datos son resultado de la media de varias ejecuciones iguales, y no de una ejecución aislada.

Kernel	Dispositivo	Tiempo	Velocidad
.cl	CPU OpenCL	88.53 ms	11.8 MH/s
.cl	GPU OpenCL	76.93 ms	816.6 MH/s
.cl	GPU CUDA	69.12 ms	908.6 MH/s
.cu	GPU CUDA	69.29 ms	906.3 MH/s

Tabla 4.4: Resumen del primer análisis de rendimiento.

Como se puede ver en los datos de la tabla, no compensa traducir los kernels a CUDA, ya que el tiempo es prácticamente igual al conseguido con la ejecución del kernel con extensión .cl. De hecho en este caso el rendimiento es un poco peor, aunque la diferencia entre ambos es mínima. Una vez visto esto, se llega a la conclusión de que la mejor opción es correr la herramienta con la tecnología de CUDA utilizando los kernels originales de la aplicación. Con estos datos se comprueba que los kernels creados por los desarrolladores de Hashcat están muy optimizados, pero al darle más importancia a OpenCL que a CUDA, es muy posible que no se haya hecho un estudio a fondo relacionado con la elección del reparto de la carga de trabajo y, por lo tanto, este puede ser un punto clave para mejorar el rendimiento de la aplicación. El reparto de la carga de trabajo se refiere a la cantidad de procesamiento que se le asigna a cada hilo de ejecución en un momento dado

Con esto último se ha llegado a la conclusión de que es necesario hacer distintas pruebas de rendimiento modificando el reparto de trabajo de la herramienta, y para ello podemos modificar tanto el tamaño de grid utilizado por CUDA en las llamadas a los kernels, como el tamaño de bloque. Esta idea de realizar modificaciones en el tamaño del grid surge por la

dependencia directa que existe entre el tamaño de cuadrícula (grid) y el uso efectivo del hardware disponible en el equipo que ejecuta la herramienta. Es decir, modificando el tamaño de bloque y de grid podemos conseguir un mejor reparto de la carga de trabajo, y como consecuencia, conseguimos un uso más efectivo de los recursos disponibles. Además, también dependen del tamaño de grid otros conceptos relacionados con el reparto de trabajo, como puede ser la ocupación, que hace referencia al número de recursos de la GPU utilizados frente a los disponibles, y el porcentaje máximo de utilización de cada una de las unidades de SM (Streaming Multiprocessors; véase sección 2.3). Una vez que se prueben varios tamaños de cuadrícula y de bloque y se encuentre la mejor combinación, se conseguirá un mejor reparto de trabajo, dando como resultado un mejor rendimiento general de la aplicación.

Para lograr el aumento de rendimiento de Hashcat y aprovechar al máximo todas las funcionalidades que nos ofrece CUDA, se divide esta sección en 2 apartados, correspondiendo cada uno de ellos con una nueva mejora. Esta división se hace con el objetivo de presentar gradualmente el trabajo realizado y aplicarle una segunda mejora al código en el momento en el ya se haya cumplido el objetivo principal del proyecto.

4.2.1 Ajuste del tamaño de grid y de bloque en CUDA.

Como hemos visto tras el primer análisis de rendimiento, se comprueba que la opción más eficiente hasta el momento es ejecutar el código original de la herramienta desde un dispositivo GPU con CUDA. Además, una vez estudiado el comportamiento de la herramienta con su código original, se concluye que es muy posible que el rendimiento de esta mejore si somos capaces de aprovechar al máximo las funcionalidades relacionadas con la paralelización que nos ofrece CUDA.

Para comenzar, se estudia el comportamiento de Hashcat con respecto a la llamada al kernel, con el objetivo de comprobar cuál es la forma en la que la herramienta utiliza CUDA para llamar al kernel utilizando varios hilos de ejecución de forma paralela. Como se ha comentado anteriormente en la sección 2.3, existen dos opciones para realizar una llamada a un kernel mediante esta tecnología. Se puede llamar directamente a la función del kernel indicándole el número de bloques y el tamaño de bloque que queremos utilizar mediante los símbolos «<,»>, o por otro lado existe la posibilidad de invocar el kernel haciendo uso de la función cuLaunchKernel(), a la cual le podemos indicar vía parámetros el tamaño de la cuadrícula (grid) y el tamaño de bloque para cada dimensión (X, Y y Z).

Gracias a los estudios hechos anteriormente, sabemos que Hashcast utiliza esta última opción para llamar a los kernels, ya que en el fichero backend.c, podemos localizar la llamada a la función cuLaunchKernel(). Este es un buen ejemplo con el que comprobar que el estudio a fondo de la aplicación es una fase indispensable para el desarrollo de este proyecto, ya que en el caso de no haber realizado dicho estudio, se perdería mucho tiempo en encontrar el

fichero y la función en la que se llama a los kernels. Para ser más concretos, se comprueba que dicha llamada se encuentra dentro de una función con el nombre de hc_cuLaunchKernel(), la cual a su vez se llama desde la función run_kernel(). El uso de cuLaunchKernel() lo podemos observar en la figura 4.9.

```
int hc_cuLaunchKernel (hashcat_ctx_t *hashcat_ctx, CUfunction f, unsigned int gridDimX, unsi
{
  backend_ctx_t *backend_ctx = hashcat_ctx->backend_ctx;

  CUDA_PTR *cuda = (CUDA_PTR *) backend_ctx->cuda;

  const CUresult CU_err = cuda->cuLaunchKernel (f, gridDimX, gridDimY, gridDimZ, blockDimX,

  if (CU_err != CUDA_SUCCESS)
  {
    const char *pStr = NULL;

    if (cuda->cuGetErrorString (CU_err, &pStr) == CUDA_SUCCESS)
    {
      event_log_error (hashcat_ctx, "cuLaunchKernel(): %s", pStr);
    }
    else
    {
      event_log_error (hashcat_ctx, "cuLaunchKernel(): %d", CU_err);
    }

    return -1;
  }

  return 0;
}
```

Figura 4.9: Llamada al kernel con la función cuLaunchKernel().

Una vez localizada la llamada a la función del kernel, se utiliza el modo DEBUG para comprobar los distintos parámetros que se le están pasando a dicha función en el código original, con el objetivo de comprobar qué tipo de paralelización se está utilizando en los dispositivos que utilizan la tecnología de CUDA y cómo se está realizando el reparto de la carga de trabajo. Al ver dichos parámetros se comprueba que Hashcat funciona con una sola dimensión en sus kernels. Esto es debido a que sus programadores decidieron que la mejor opción era trabajar con estructuras de datos de una sola dimensión en sus kernels. Es por esta razón por la cual podemos ver que los valores de las variables correspondientes a las dimensiones Y y Z están igualados a 1. Por otro, al observar los parámetros también se comprueba que no se está aprovechando la paralelización al máximo. El valor de estos parámetros es el mostrado en la figura 4.10 y se puede observar poniendo un punto de parada en la llamada a los kernels. Si nos fijamos en dichos valores podemos deducir que, para la paralelización, se está utilizando una cuadrícula de tamaño igual a 5 y un tamaño de bloque de 1024. Con esto vemos que todos los threads disponibles se agrupan únicamente en 5 bloques, y que por lo tanto, se está trabajando con 5120 (5 * 1024) hilos que se ejecutan de forma paralela. Esto se podría mejo-

rar mediante la realización de una prueba de rendimiento en la que se vayan variando estos valores, de forma que se consiga la mejor combinación posible para el tamaño de bloque y de grid, con el objetivo de aumentar el rendimiento lo máximo posible.

Figura 4.10: Parámetros de llamada al kernel en el código original.

De esta forma estamos viendo que el código original de Hashcat está utilizando paralelismo en la llamada al kernel, pero no lo está haciendo de la forma más eficiente posible, ya que sólo están utilizando 5 bloques de threads. Con esto llegamos a la conclusión de que cambiando algún parámetro y haciendo que la herramienta aproveche mejor la paralelización, es posible que se consiga una ejecución mucho más eficiente y que por lo tanto, el tiempo de ejecución del benchmark y de cualquier ataque realizado con una GPU que use CUDA disminuya bastante. Antes de hacer un análisis en el que se prueben varios valores para estas variables, con el objetivo de encontrar la forma más eficiente en la que ejecutar la herramienta, es importante recordar que, en este caso, sólo podemos cambiar el valor de las variables gridDimX y blockDimX. Esto ocurre debido a que, como se ha comentado anteriormente, Hashcat sólo utiliza una dimensión, por lo que si modificamos cualquiera de las otras, correspondientes a las dimensiones Y y Z, se producirá un error de ejecución.

Para llegar a la solución más eficiente posible, se realiza un segundo análisis de rendimiento en el que se prueban varios valores para las variables gridDimX y blockDimX. El objetivo de dicho análisis es encontrar una combinación de valores con la que mejorar el reparto de la carga de trabajo y disminuir el tiempo de ejecución. Este análisis se explica en la sección 5.2, en la cual se observa que para conseguir la mejor combinación de tamaño de grid y de bloque se prueban un gran número de valores para cada una de las variables. Como resumen de las mejores ejecuciones obtenemos la tabla 4.5. La forma en la que implementar la mejora y el motivo de la misma se explican a continuación.

Dispositivo	gridDimX	blockDimX	Tiempo	Velocidad
GPU Nvidia Tesla Kepler K40c	5	1024	69.29 ms	906.3 MH/s
GPU Nvidia Tesla Kepler K40c	128	1024	10.65 ms	5858.1 MH/s
GPU Nvidia Tesla Kepler K40c	1024	128	9.60 ms	6485.0 MH/s

Tabla 4.5: Datos de las mejores ejecución del segundo análisis.

Una vez hecho el segundo análisis de rendimiento, podemos comprobar que se consigue bajar el tiempo de ejecución de 69.29ms a 9.60ms, lo cual implica una mejora bastante grande. Además, no conseguimos una única configuración que mejore el rendimiento, sino que también obtenemos una segunda combinación de valores en la que se disminuye el tiempo de ejecución a 10.65ms. Ambas mejoras consiguen unos resultados muy parecidos, pero como es obvio, a partir de ahora se trabaja con la configuración que nos ha dado un menor tiempo de ejecución.

Una vez concluido este análisis y obtenidos los distintos resultados, se ha realizado un estudio de varias ejecuciones, con la herramienta NVIDIA Nsight Compute, el cual se explica a fondo en el apartado 5.2. Este estudio nos permite comprobar cómo se cumple lo explicado anteriormente, ya que al encontrar una mejor combinación para el tamaño de grid y de bloque conseguimos un mejor reparto de la carga de trabajo y, como consecuencia, un aumento del uso efectivo del hardware disponible. Esta es la causa principal por la que se consigue aumentar el rendimiento de la herramienta.

En este punto, para hacer efectiva esta mejora, debemos implementar este cambio en el código original de Hashcat. Para ello, se asignan los nuevos valores para las variables gridDimX y blockDimX justo antes de la llamada a la función cuLaunchKernel(), dentro de la función run_kernel, ya que es desde esta desde donde se le pasan los parámetros relacionados con la paralelización al kernel. Una vez añadidas estas líneas de código, si ejecutamos el modo DEBUG podemos comprobar que cada vez que se llama al kernel de cualquier algoritmo de hash se está haciendo con el valor de gridDimX establecido a 1024 y el de blockDimX a 128. Estos nuevos valores se pueden observar en la figura 4.11.

```
gridDimX: 1024
gridDimY: 1
gridDimZ: 1
blockDimX: 128
blockDimY: 1
blockDimZ: 1
```

Figura 4.11: Valores actualizados para aprovechar el paralelismo al máximo.

La mejora obtenida en esta sección se consigue gracias a que se aprovecha al máximo la paralelización de CUDA, es decir, en el código original de Hashcat se estaba utilizando una cuadrícula de tamaño igual a 5, lo que quiere decir que se estaba trabajando únicamente con 5 bloques. A pesar de esto, sí que se estaba utilizando algo de paralelización, ya que se estaban utilizando un total de 5120 threads de forma paralela. Por lo tanto, se estaban usando 5 bloques de 1024 threads que se ejecutaban de forma paralela en la llamada al kernel. Al cambiar el tamaño del grid, pasando de 5 bloques en el código original, a 1024 bloques en

el nuestro y mantener un tamaño de bloque de 128 se consigue que la cuadrícula (grid) esté formada por 1024 bloques de hilos que pueden trabajar de forma independiente, y por lo tanto de forma paralela. A su vez, dentro de cada uno de esos bloques hay 128 threads que cooperan entre ellos y trabajan también de forma paralela.

Antes de verificar esta mejora para otros kernels, es importante realizar una prueba de funcionamiento, en la que se compruebe que una vez modificado de forma manual el tamaño de cuadrícula y de bloque, la herramienta sigue funcionando correctamente. Esta prueba también se ha realizado en la sección 5.2, y en ella se ha ejecutado un ataque con el código de la herramienta modificado, dando como resultado la contraseña perteneciente al hash objetivo. Esta prueba es necesaria debido a que algunas aplicaciones están pensadas únicamente para trabajar con una configuración del grid específica, por lo que no sería raro que al cambiar esta de forma manual, la herramienta dejase de funcionar. A pesar de esta posibilidad, se ha comprobado que la herramienta sigue funcionando correctamente y es capaz de resolver los ataques que se ejecuten.

Llegados a este punto, es necesario comprobar que esta mejora funciona para cualquier tipo de kernel. Para ello se hace un tercer análisis de rendimiento en el que se comparan los tiempos de 4 nuevos kernels utilizando el código original, y posteriormente usando nuestro código mejorado, siendo este último el modificado para trabajar con un tamaño de grid de 1024 y un tamaño de bloque de 128. Para esto se escogen los kernels que se corresponden con los algoritmos de hash más utilizados por los usuarios en la actualidad, los cuales son los siguientes: md5, NTLM, RIPEMD-160 y SHA-3.

Este tercer análisis de rendimiento está explicado en la sección 5.3, y como resumen de los datos obtenidos elaboramos la tabla 4.6. El término de ”código mejorado” en dicha tabla hace referencia al código modificado para conseguir un mejor reparto de la carga de trabajo, utilizando 1024 bloques de tamaño igual a 128.

Código	Dispositivo	Hash	Tiempo	Velocidad
Original	GPU Nvidia Tesla Kepler K40c	MD5	54.32 ms	4621.4 MH/s
Mejorado	GPU Nvidia Tesla Kepler K40c	MD5	32.88 ms	30527.3 MH/s
Original	GPU Nvidia Tesla Kepler K40c	NTLM	59.96 ms	8375.5 MH/s
Mejorado	GPU Nvidia Tesla Kepler K40c	NTLM	18.55 ms	53996.9 MH/s
Original	GPU Nvidia Tesla Kepler K40c	RIPEMD-160	48.04 ms	1306.0 MH/s
Mejorado	GPU Nvidia Tesla Kepler K40c	RIPEMD-160	7.08 ms	8769.8 MH/s
Original	GPU Nvidia Tesla Kepler K40c	SHA3-256	66.03 ms	237.8 MH/s
Mejorado	GPU Nvidia Tesla Kepler K40c	SHA3-256	10.35 ms	1508.0 MH/s

Tabla 4.6: Resultado de las ejecuciones de los nuevos kernels comparándolos con los originales.

Como se puede ver en la tabla 4.6, la mejora de rendimiento no es igual en todos los casos, ya que en algunos se nota un poco más que en otros. Aún así, se puede observar una

gran mejora en todos, y en su totalidad, se puede comprobar que existe una diferencia notable tanto en la velocidad como en el tiempo, entre su ejecución realizada con el código original y su posterior ejecución con el código modificado.

Una vez implementadas las mejoras relacionadas con la paralelización del código CUDA y comprobadas mediante un análisis de rendimiento en 5 kernels pertenecientes a 5 algoritmos de hash distintos (MD5, SHA2-256, RIPEMD-160, NTLM y SHA3-256), podemos sacar una conclusión general de lo realizado hasta el momento. Con esto, comprobamos que habiendo estudiado la forma en la que Hashcat utiliza la paralelización de su código CUDA, es posible aprovechar las funciones que nos ofrece dicha tecnología para aumentar el rendimiento en el 100% de los kernels analizados.

A pesar de conseguir una gran mejora en el rendimiento de la aplicación, se sigue intentando mejorar algo más estos datos. Para ello se realiza un nuevo estudio en el que se investiga un poco más el funcionamiento de Hashcat. Este proceso se explica en la sección 4.2.2.

4.2.2 Nuevos ajustes relacionados con la carga de trabajo de Hashcat

Una vez que se logra el objetivo de este proyecto, consiguiendo una gran mejora de rendimiento trabajando con la paralelización de CUDA, se descubre una funcionalidad que ofrece Hashcat desde el año 2016 que puede mejorar aún más el rendimiento de la aplicación. Esta funcionalidad consiste en un motor de autoconfiguración que se activa cada vez que hacemos una ejecución de la herramienta, pero que se puede desactivar o configurar manualmente, basándonos en las características de nuestro dispositivo GPU/CPU. Leyendo y entendiendo el artículo en el que se explica el funcionamiento de esta característica [18], se llega a la conclusión de que, modificando los valores que toma este motor a la hora de trabajar con nuestra GPU en concreto, se pueden conseguir nuevas mejoras de rendimiento.

Antes de comenzar con estos nuevos cambios es importante explicar qué es el motor de autoconfiguración y cómo funciona. Se trata de un sistema que busca de forma automática la mejor distribución de la carga de trabajo entre los hilos disponibles para cada ataque que se realice. Para cambiar la forma en la que trabaja este sistema de forma sencilla y que cualquier usuario pueda variar un poco su funcionamiento, se puede hacer uso de la opción ”w” a la hora de ejecutar la herramienta, habiendo 4 valores posibles: 1, 2, 3 o 4. Cuanto menor sea dicho valor, menos recursos utilizará y por lo tanto, podremos ejecutar otras aplicaciones mientras se realiza el ataque. Por otro lado, si el valor de w es 4, el ataque utilizará todos los recursos disponibles y no podremos hacer otras tareas en el dispositivo en el que se ejecute el ataque.

Además de la forma sencilla de jugar con el motor de autoconfiguración, también existe una forma pensada para los desarrolladores o para gente que esté dispuesta a tocar el código de la herramienta. Esta consiste en una base de datos en la que se establece, para cada dispositivo, el ancho del vector que se quiere utilizar para las operaciones vectoriales y los valores

para las variables Kernel-Accel y Kernel-Loops. Kernel-Accel hace referencia al concepto de OpenCL de elemento de trabajo, por lo que al cambiar este valor estamos cambiando la configuración del bucle externo (outerloop). Por otro lado, Kernel-Loops dependiendo del tipo de algoritmo de hash se refiere al número de iteraciones por elemento de trabajo (en algoritmos de hash lentos) o al número de mutaciones por elemento de trabajo (en algoritmos de hash rápidos), con lo cual, al cambiar este valor estamos cambiando la configuración del bucle interno (innerloop) [7]. En la figura 4.12 podemos ver un ejemplo de un kernel en el cual se puede comprobar que existe un bucle al comienzo del kernel (bucle externo) y un bucle en el centro del kernel (bucle interno), la diferencia entre ambos se explica en la guía para desarrolladores de Hashcat [16]. Es importante recalcar que ninguna de estas dos últimas variables se corresponde con el concepto de hilo, como bien se explica en el foro de la herramienta [18].

```
KERNEL_FQ void m01400_mxx (KERN_ATTR_VECTOR ())
{
  /**
   * modifier
   */

  const u64 lid = get_local_id (0);
  const u64 gid = get_global_id (0);

  if (gid >= gid_max) return;

  /**
   * base
   */

  const u32 pw_len = pws[gid].pw_len;

  u32x w[64] = { 0 };

  for (u32 i = 0, idx = 0; i < pw_len; i += 4, idx += 1)
  {
    w[idx] = pws[gid].i[idx];
  }

  /**
   * loop
   */

  u32x w0l = w[0];

  for (u32 il_pos = 0; il_pos < il_cnt; il_pos += VECT_SIZE)
  {
    const u32x w0r = words_buf_r[il_pos / VECT_SIZE];

    const u32x w0 = w0l | w0r;

    w[0] = w0;

    sha256_ctx_vector_t ctx;

    sha256_init_vector (&ctx);

    sha256_update_vector (&ctx, w, pw_len);

    sha256_final_vector (&ctx);

    const u32x r0 = ctx.h[DGST_R0];
    const u32x r1 = ctx.h[DGST_R1];
    const u32x r2 = ctx.h[DGST_R2];
    const u32x r3 = ctx.h[DGST_R3];

    COMPARE_M_SIMD (r0, r1, r2, r3);
  }
}
```

Figura 4.12: Código de un kernel en el que se ve tanto el bucle externo como el interno.

La base de datos de motor de autoconfiguración es un fichero en el cual, cada entrada está en una línea, y los distintos parámetros están separados por espacios o tabulaciones. Los parámetros a utilizar son los siguientes: Nombre del dispositivo (nombre que le da Hashcat sustituyendo espacios por "_"), modo de ataque (número que Hashcat relaciona con cada ataque, por ejemplo: si el ataque es de tipo fuerza bruta, se igualará a 3), tipo de función hash (número que Hashcat le da a cada tipo de algoritmo de hash, por ejemplo: si el algoritmo de hash es SHA2-256, le corresponde el valor 1400), ancho del vector (puede ser igual a 1, 2, 4, 8 o N, siendo este último el valor que OpenCL cree más conveniente), Kernel-Accel (puede tomar como valor máximo 1024 y además existen los valores A y M, significando autoajuste y máximo, respectivamente) y Kernel-Loops (puede tener los mismos valores que la variable anterior). Por otro lado, en este mismo fichero también se pueden definir "Alias", con lo que a varios dispositivos se les puede referenciar con un mismo nombre, y de esta forma simplificar la base de datos.

Lo primero que necesitamos para la funcionalidad dedicada a los desarrolladores es encontrar la base de datos dentro del repositorio de la herramienta, ya que en el informe en el que se explica su uso [18] no se indica ni su ubicación ni su nombre. Para realizar esta búsqueda se utiliza la herramienta "Eclipse", ya que permite buscar en un directorio de trabajo según el contenido de todos los ficheros de forma muy visual (se podría haber utilizado cualquier otra herramienta que permitiese hacer lo mismo). Como resultado obtenemos que la base de datos se encuentra en el directorio principal de la herramienta, y que el fichero se llama "hashcat.hctune".

Una vez encontrado dicho archivo, se comprueba que el formato el que se describe en el artículo nombrado anteriormente, y se comienza a probar su funcionamiento. En primer lugar, se verifica si el dispositivo utilizado en las pruebas de rendimiento (llamado Tesla_K40) tiene algún Alias con el cual trabajar. Como se puede observar en la figura 4.13, dicho dispositivo se relaciona con el Alias "ALIAS_nv_real_simd".

```
#Device                                                 Alias
#Name                                                   Name

Tesla_C2050                                             ALIAS_nv_real_simd
Tesla_C2050/C2070                                       ALIAS_nv_real_simd
Tesla_C2070                                             ALIAS_nv_real_simd
Tesla_C2075                                             ALIAS_nv_real_simd
Tesla_K10                                               ALIAS_nv_real_simd
Tesla_K20                                               ALIAS_nv_real_simd
Tesla_K40                                               ALIAS_nv_real_simd
Tesla_K80                                               ALIAS_nv_real_simd
Tesla_M20xx                                             ALIAS_nv_real_simd
```

Figura 4.13: Alias de los dispositivos Tesla en la base de datos.

Posteriormente, ya sabiendo con qué nombre referirnos a nuestro dispositivo, pensamos cuáles eran los mejores valores para mejorar el rendimiento al máximo teniendo en cuenta los valores que estamos utilizando para llevar a cabo la paralelización del código (en nuestro caso se está trabajando con 1024 bloques y con un tamaño de bloque de 128, como se ha explicado en la sección 4.2.1). En primer lugar, los valores que son claros desde un principio, ya que se refieren al nombre de dispositivo, tipo de ataque y tipo de hash, son los siguientes: Device_name = ALIAS_nv_real_simd, Attack_mode = 3 y Hash_type = 1400 y los valores con los que podemos jugar son los pertenecientes a las variables Vector_width, Kernel_Accel y Kernel_Loops. En este punto se decide realizar un cuarto análisis de rendimiento, para ver qué valores hacen que la herramienta sea más eficiente. Todo ello está explicado en la sección 5.4. Como resumen, se consigue disminuir el tiempo de ejecución de 9.60ms a 0.04ms en la ejecución del benchmark perteneciente al algortimo de hash SHA2-256. Para ello, se cambian los valores de las variables nombradas anteriormente, dándole a las 3 (Vector_width, Kernel_Accel y Kernel_Loops) un valor de 1.

Lo que se logra con esto es que todos los threads que se utilizan en la paralelización explicada anteriormente, tengan un reparto de trabajo similar, con lo cual alcanzamos un mejor rendimiento. Al poner a 1 tanto el tamaño del vector de las operaciones secuenciales, como los pasos de los bucles interno y externo, cada elemento de trabajo hace, en principio, una sola iteración en los bucles, y en el momento en el que todos hacen la suya se vuelve a empezar desde el elemento de trabajo número 0, por lo que se consigue un reparto de la carga de trabajo equitativo y cíclico. Gracias a este reparto de trabajo equitativo y cíclico se consigue aprovechar la coalescencia en las tarjetas gráficas. Esta es una técnica con la cual se logra que varios hilos consecutivos soliciten zonas de la memoria global contiguas. Al utilizar esta técnica se reduce el número de accesos a memoria, ya que lo que hace la tarjeta gráfica es acceder a varias posiciones consecutivas de memoria en vez de realizar un acceso para cada hilo de ejecución. Esto explica que, una vez implementada la mejora, se consiga un gran avance tanto en el tiempo, como en la velocidad de ejecución. Esto ocurre debido a que se está haciendo que todos los hilos de ejecución realicen un número de operaciones similar, mejorando así el rendimiento. Para hacer efectivo este cambio es necesario actualizar los valores en la base de datos. Para ello se añaden la líneas mostradas en la figura 4.14. Una de ellas se corresponde con el alias del clúster Pluton y la otra con el alias del ordenador portátil.

```
#### Pruebas para el kernel CUDA 1400
ALIAS_nv_sm50_or_higher                 3       1400    1       1       1
ALIAS_nv_real_simd                      3       1400    1       1       1
```

Figura 4.14: Linea añadida en la base de datos.

En este punto, es importante recordar que esta segunda mejora conseguida mediante el

uso del motor de autoconfiguración, se implementa sobre el código modificado en la sección 4.2.1, con el cual conseguimos un aumento de rendimiento variando la configuración del grid. Por lo tanto, en esta sección, se está consiguiendo una segunda mejora que se aplica sobre la primera.

Una vez llegados a este punto, se comprobó que las mejoras aplicadas hasta ahora funcionan en algunos casos en particular, por lo que es necesario probarlas en ejecuciones con distintos parámetros, para de esta forma confirmar si dichas mejoras afectan a la gran parte de los casos en los que se usa la herramienta. Para ello se realiza un último análisis de rendimiento, ejecutando distintos ataques reales, variando el dispositivo con el que se llaman, el tipo de hash, la complejidad y número de contraseñas a las que se ataca, y el tipo de ataque. De esta forma, pasamos de analizar ejecuciones del benchmark de la herramienta, a analizar ataques reales, con lo cual veremos cuál es la ganancia real conseguida con las modificaciones explicadas anteriormente. Este quinto análisis de rendimiento está explicado en la sección 5.5.

Como conclusión se comprueba que las mejoras implementadas a lo largo de este capítulo funcionan para el 100% de los casos probados. Además, se corrobora que cuanto más tiempo duren las ejecuciones de Hashcat, más notaremos la mejora. Un ejemplo de esto se muestra en la tabla 4.7, en la que se observa la gran diferencia que hay al ejecutar un ataque por diccionario antes y después de haber implementado las mejoras.

	Tiempo CUDA código original	**Velocidad CUDA código original**	**Tiempo CUDA código mejorado**	**Velocidad CUDA código mejorado**
GPU Tesla Kepler K40c	1 h 6 min	103.4 MH/s	14 min 43 s	104.2 MH/s
GPU Tesla Kepler K20m	41 min 2 s	158.4 MH/s	29 min 8 s	159.1 MH/s

Tabla 4.7: Datos del ataques por diccionario dirigido a hashes de tipo SHA3-256.

Capítulo 5

Experimentos

En este capítulo se describe y muestra el proceso de pruebas y análisis de rendimiento de las distintas fases relacionadas con la mejora del código CUDA de la herramienta Hashcat. Para esto, se realizan varias ejecuciones que se van explicando a lo largo del capítulo. Con los datos obtenidos en dichas ejecuciones se elaboran tablas para mostrar los resultados de la forma más visual posible. Como ya se ha comentado en la sección 2.2.2, las ejecuciones relacionadas con pruebas de rendimiento se realizan en el clúster Pluton utilizando la GPU de Nvidia Tesla Kepler K40c. Además, una vez que se implementen todas las mejoras y se llega al final del proyecto se comprueban los tiempos con varias ejecuciones realizadas con otra GPU de Nvidia llamada Tesla Kepler K20m, con el objetivo de confirmar que dichas mejoras se cumplen en distintas plataformas.

En la sección 5.1 se hace un primer análisis en el que se estudia el rendimiento del código original de Hahscat, comparando los datos obtenidos al ejecutar la herramienta desde dispositivos de distinto tipo (CPUs, GPUs con OpenCL y GPUs con CUDA). Además se estudia el rendimiento de un kernel traducido a CUDA al inicio de la sección 4.2.

En la sección 5.2 se realiza un análisis de rendimiento en el cual se varían los tamaños de grid y de bloque en la llamada al kernel desde CUDA. Con esto se busca obtener la mejor combinación, con el objetivo de conseguir una mejora en el rendimiento de la herramienta. Además, se utiliza la herramienta NVIDIA Nsight Compute para comprobar la razón real por la que se consigue dicha mejora.

En la sección 5.3 se comprueba que la mejora conseguida en la sección 4.2.1 se aplica a todos los kernels y no sólo al utilizado hasta el momento. Es importante hacer esta prueba para verificar que la memoria es de ámbito general y no sólo afecta a un caso concreto.

En la sección 5.4 lo que se hace es una serie de ejecuciones variando la configuración del motor de autotunning, con el objetivo de lograr la configuración más óptima posible para el dispositivo que se está utilizando. Además, en este caso también se utiliza la herramienta de NVIDIA llamada Nsight Compute para verificar que se está consiguiendo un mejor uso

efectivo del hardware disponible.

Por último, en la sección 5.5 se realizan un gran número de ejecuciones de ataques reales con nuestra modificación de Hashcat, con el objetivo de analizar las mejoras de rendimiento obtenidas frente a la ejecución de esos mismos ataques con el código original de la herramienta. Estas ejecuciones se han realizado variando las características de cada ataque y los dispositivos desde los que se ejecutan, para así comprobar el aumento de rendimiento en el mayor número de casos posibles.

5.1 Análisis de rendimiento del código original de Hashcat

Como se comenta al inicio de la sección 4.2, se hace un primer análisis de rendimiento con el objetivo de ver los tiempos de ejecución de Hashcat con su código original. Además, se verifica el tiempo de ejecución de una traducción literal de un kernel a CUDA, ya que es importante comprobar si se consigue una mejora. Pero como ya se ha explicado, es muy posible que la diferencia entre los kernels originales y el kernel traducido sea mínima. Este análisis inicial se corresponde con una serie de pruebas preliminares en las que se utiliza el benchmark de la herramienta, por lo que los tiempos de ejecución son muy bajos.

Para este primer análisis de rendimiento se han realizado varias ejecuciones de un benchmark de la herramienta, que se corresponde con el kernel perteneciente al algoritmo de hash llamado SHA2-256. Para comprobar las diferencias en el tiempo de ejecución de Hashcat en distintos dispositivos, se han realizado 4 ejecuciones distintas. En primer lugar, se ha utilizado una CPU OpenCL, la cual se encuentra en el clúster Pluton y es una Intel Xeon E5-2650v2 Ivy Bridge-EP. A continuación, se han hecho pruebas en la GPU de NVIDIA nombrada al inicio del capítulo, programándose tanto con OpenCL como con CUDA, con el objetivo de ver si existe una diferencia clara entre ambas tecnologías. Por último, se ha hecho una segunda ejecución mediante la GPU con CUDA, pero esta vez utilizando el kernel traducido a CUDA en la sección 4.2. Como se ha explicado anteriormente, es posible que no exista prácticamente ninguna mejora a la hora de llevar a cabo esta última ejecución, ya que el cambio realizado es mínimo. Lo que se hace para utilizar nuestro kernel es cambiar momentáneamente en el código de la herramienta el nombre del fichero del kernel al que se llama, sustituyendo ”m01400_a3-optimized.cl” por ”m01400_a3-optimized.cu”.

Para realizar estas 4 ejecuciones se ha utilizado el comando mostrado en el listado 5.1, variando el valor de las opciones -d y -D, con el objetivo de utilizar los distintos dispositivos nombrados anteriormente. Estas dos opciones que nos ofrece Hashcat han sido explicadas en la sección 4.1.

```
1 ./hashcat -m 1400 -b --force
```

Listado 5.1: Ejecución del benchmark del kernel de SHA2-256.

El resultado de estas 4 ejecuciones de la herramienta se puede ver reflejado en la tabla 5.1. En esta, podemos ver cómo los casos más eficientes son aquellos que se ejecutan desde una GPU utilizando la tecnología CUDA, ya que su tiempo de ejecución es bastante menor a los conseguidos utilizando OpenCL. Además, también se puede comprobar cómo ejecutar la herramienta mediante una GPU es mucho más eficiente que si se hace con una CPU. Por último, en esta tabla resumen, podemos ver cómo los tiempos de las dos ejecuciones realizadas con la GPU con CUDA son muy similares, e incluso es algo menor el de la ejecución con el kernel original de Hashcat. Con esto, se ha comprobado que no compensa traducir los kernels a CUDA, ya que se consigue un resultado idéntico al logrado con los kernels originales (aquellos escritos en ficheros con extensión .cl).

Kernel	Dispositivo	Tiempo	Velocidad
.cl	CPU OpenCL	88.53 ms	11.8 MH/s
.cl	GPU OpenCL	76.93 ms	816.6 MH/s
.cl	GPU CUDA	69.12 ms	908.6 MH/s
.cu	GPU CUDA	69.29 ms	906.3 MH/s

Tabla 5.1: Resumen del primer análisis de rendimiento.

5.2 Modificación del tamaño de cuadrícula y de bloque

En este segundo análisis de rendimiento se realizan una serie de ejecuciones del benchmark utilizado anteriormente en la sección 5.1, cambiando los valores de las variables gridDimX y blockDimX en la llamada al kernel desde la función cuLaunchKernel() perteneciente al API de CUDA. Todas se llevan a cabo utilizando CUDA en la GPU de Nvidia Tesla Kepler K40c, disponible en el clúster Pluton. Es importante recordar que las ejecuciones del benchmark nos proporcionan tiempos de ejecución muy pequeños, pero a su vez son las ejecuciones que mejor reflejan el comportamiento general de la herramienta. Esta última característica es muy importante debido a que a la hora de comprobar una mejora, es preferible que se aplique a un contexto lo más general posible, en vez de a un caso particular.

Las pruebas realizadas en esta sección se hacen con el objetivo de encontrar la configuración más eficiente a la hora de correr el código de forma paralela. Más adelante se hacen pruebas de rendimiento sobre casos reales, en los cuales los tiempos de ejecución son más elevados y se puede comprobar si las mejoras conseguidas para el benchmark se cumplen también para usos normales de la herramienta.

Antes de comenzar directamente con el análisis es importante recordar el mejor tiempo

conseguido hasta el momento, teniendo en cuenta que este se consigue ejecutando el código original de Hashcat en una GPU con CUDA. Además, como se ha explicado en la sección 4.2.1, hay que tener en cuenta que hasta ahora se estaba usando un valor igual a 5 para la variable gridDimX y un valor de 1024 para blockDimX. Los datos obtenidos hasta el momento son los que muestra la tabla 5.2.

Dispositivo	gridDimX	blockDimX	Tiempo	Velocidad
GPU Nvidia Tesla Kepler K40c	5	1024	69.29 ms	906.3 MH/s

Tabla 5.2: Datos de la mejor ejecución del primer análisis.

Partiendo de estos datos se realiza una ejecución del benchmark del kernel 1400, perteneciente al algoritmo de hash SHA2-256, que es el que se ha utilizado en el análisis anterior. Si se logra una mejora para este hash, en un futuro se comprobará para varios kernels más, con el objetivo de ver si se consigue la misma mejora en todos ellos. El comando que se utiliza a lo largo de esta sección es el mismo que el utilizado en el análisis anterior, forzando que sea que la GPU con CUDA la que ejecute el benchmark. Dicho comando se muestra en el listado 5.2.

```
1 ./hashcat -m 1400 -b --force -D2 -d1
```

Listado 5.2: Ejecución del benchmark del código original con GPU CUDA.

Un resumen de los resultados obtenidos en algunas de las ejecuciones realizadas, cambiando los valores de las variables gridDimX y blockDimX, está reflejado en la tabla 5.3.

Dispositivo	gridDimX	blockDimX	Tiempo	Velocidad
GPU Nvidia Tesla Kepler K40c	5	1024	69.29 ms	906.3 MH/s
GPU Nvidia Tesla Kepler K40c	128	1024	10.65 ms	5858.1 MH/s
GPU Nvidia Tesla Kepler K40c	256	1024	69.26 ms	906.7 MH/s
GPU Nvidia Tesla Kepler K40c	512	512	73.97 ms	850.14 MH/s
GPU Nvidia Tesla Kepler K40c	1024	128	9.60 ms	6485.0 MH/s
GPU Nvidia Tesla Kepler K40c	1024	256	77.58 ms	809.6 MH/s

Tabla 5.3: Datos obtenidos al cambiar el valor de las variables gridDimX y blockDimX.

Como resultado, podemos ver que después de haber probado varios valores para cada una de las variables, llegamos a la conclusión de que la mejor combinación es usar un valor de 1024 para gridDimX y de 128 para blockDimX. Con estos cambios, conseguimos bajar el tiempo de ejecución del benchmark del algoritmo de hash SHA2-256, de un tiempo anterior de 69.29ms y una velocidad de 906.3MH/s a un tiempo de 9.60ms y una velocidad de 6485.0MH/s.

Siendo kH/s mil hashes por segundo y MH/s un millón de hashes por segundo. En conclusión, aprovechando las funcionalidades de CUDA relacionadas con la paralelización se obtiene un tiempo de ejecución 7 veces menor al conseguido con el código original de la herramienta.

Una vez que se ha implementado la mejora conseguida anteriormente es necesario comprobar que la aplicación sigue funcionando. Para ello ejecutamos un ataque a un hash de este tipo. En el caso de que este ataque se resuelva con éxito, podremos decir que la implementación de esta mejora es correcta. Es importante decir, que antes de hacer cualquier prueba de un ataque se borra el fichero en el que se almacenan las contraseñas crackeadas en un pasado. El motivo es que el caso de no hacerlo, Hashcat nos indica que ese par hash-contraseña ya lo tiene almacenado y por lo tanto no se ejecuta el ataque. Para esta prueba de funcionamiento se lanza el comando mostrado en el listado 5.3.

```
./hashcat -m 1400 -O -a 3
   b221d9dbb083a7f33428d7c2a3c3198ae925614d70210e28716ccaa7cd4ddb79
   ?a?a?a?a -D2 -d1
```

Listado 5.3: Ataque para probar el correcto funcionamiento después de implementar la mejora.

Como resultado se obtiene una ejecución exitosa en la que a partir del hash: ”b221d9dbb083a7f33428d7c2a3c3198ae925614d70210e28716ccaa7cd4ddb79” se recupera la contraseña ”hola”. Con esto comprobamos que el ataque es correcto y por lo tanto, los cambios realizados en el código de Hashcat para conseguir la mejora, también.

Antes de continuar con la prueba de esta mejora en otros kernels, es importante realizar un análisis a fondo de una ejecución, con el objetivo de conocer la razón por la que se está consiguiendo este aumento de rendimiento. Para ello, NVIDIA ofrece una herramienta llamada Nsight Compute, la cual se trata de un profiler que nos proporciona distintas métricas, análisis y gráficas que nos ayudan a estudiar con detalle las distintas llamadas al kernel realizadas a lo largo de cada ejecución. Esta herramienta está pensada para las nuevas versiones de CUDA, por lo que sustituye al anterior profiler de NVIDIA, llamado NVIDIA Visual Profiler.

En este punto se llevan a cabo varios análisis, en los cuales se comparara una ejecución realizada con el código original de Hashcat frente a una ejecución realizada con el código mejorado. Para llegar a una conclusión final vamos a hacer uso de algunos términos que utiliza la herramienta, los cuales se explican a continuación:

- SOL Memory [%]: hace referencia al porcentaje de carga de trabajo utilizado por cualquier sección del sistema de memoria de la GPU destinado a cómputo en la ejecución, frente a su máximo rendimiento posible.

- SOL SM [%]: hace referencia al porcentaje máximo de utilización de cualquier subunidad de SM (streaming multiprocessor) en la ejecución.
- Achieved Occupancy: hace referencia al porcentaje de warps activos frente a los warps máximos por SM en cada ciclo activo, siendo un warp un conjunto de 32 threads dentro de un bloque de threads. Todos los hilos dentro de un warp ejecutan la misma instrucción para conseguir una mayor eficiencia, ya que de esta forma todos los SP (CUDA cores) en un multiprocesador pueden ejecutar la misma instrucción en paralelo.

Estas 3 métricas son importantes debido a que hacen referencia al uso efectivo del hardware disponible, por lo que observándolas podemos ver si se está aprovechando al máximo la GPU utilizada, o si por el contrario, se está malgastando la mayor parte de la GPU. En resumen, cuanto mayor sea el valor de cada una de ellas mejor, ya que significará que estamos aprovechando de una mejor forma los recursos hardware que tenemos disponibles.

Una vez estudiado el significado de algunas de las métricas utilizadas en Nsight, se han realizado dos ejecuciones de un ataque por fuerza bruta que refleja un uso normal de Hashcat. Esto se hace para poder estudiar el comportamiento de nuestra mejora en un caso de uso normal de la herramienta. Es importante decir que todos los análisis realizados con Nsight se han llevado a cabo en el ordenador personal descrito en la sección 2.2.2, debido a que esta herramienta necesita permisos de superusuario, y en el clúster no disponemos de ese tipo de permisos.

El resultado de los análisis con Nsight se puede observar en la tabla 5.4, en la cual podemos ver, para cada ejecución, el valor de las 3 métricas explicadas anteriormente. En ella vemos que, para la ejecución del ataque realizado con el código original de Hashcat, obtenemos un menor porcentaje para cada una de las métricas estudiadas. Con esto, comprobamos que se está haciendo un peor aprovechamiento de las capacidades de la GPU en este primer caso, mientras que a la hora de ejecutar la herramienta con el código mejorado (fila 2), se consigue un mayor uso de dichas capacidades. Esta es la razón principal por la que, una vez implementada la mejora, se consigue un mayor rendimiento al compararlo con el código original.

Código	SOL Memory	SOL SM	Achieved Occupancy
Código original de Hashcat	29.59%	10.17%	37.16%
Código después de la mejora	36.13%	16.42%	44.28%

Tabla 5.4: Resultado del análisis realizado con la herramienta NVIDIA Nsight Compute.

Con esta herramienta, también comprobamos que la modificación realizada en el código de Hashcat para conseguir la mejora, cambia el tamaño de la cuadrícula y de bloque de forma correcta. En el análisis de la ejecución con el código original podemos ver que se están utilizando 5120 threads, ya que el tamaño del grid es igual a 5 y el de bloque es igual a 1024,

mientras que en la ejecución del código modificado, se utilizan 131.072 threads, ya que el grid tiene un tamaño de 1024 y los bloques de 128.

5.3 Prueba de la mejora en distintos kernels

En este tercer análisis de rendimiento vemos el tiempo y la velocidad de ejecución de los kernels escogidos (MD5, SHA2-256, RIPEMD-160, NTLM y SHA3-256), comparando los datos obtenidos en una primera ejecución con el código original de Hashcat, frente a los datos obtenidos en una segunda ejecución utilizando el código mejorado. La forma en la que se lleva a cabo este tercer análisis de rendimiento es igual al de la sección 5.2, ya que el objetivo es comprobar si el resultado de la mejora es común en todos los kernels. Para hacerlo de la forma más clara posible, existe un apartado para cada uno de los algoritmos de hash escogidos, en los que, además de explicar todo el proceso, se muestran una serie de tablas con un resumen de los resultados.

Todas las ejecuciones realizadas en este análisis de rendimiento, se llevan a cabo con CUDA en la GPU de Nvidia Tesla Kepler K40c, disponible en el clúster Pluton. Además, es importante recordar que todas las ejecuciones que hagan referencia al código ”mejorado” se realizan con un tamaño de grid igual a 1024 y un tamaño de bloque de 128.

5.3.1 MD5

En primer lugar vamos a comenzar con el kernel correspondiente al algoritmo de hash md5 (el cual utiliza el fichero m00000_a3-optimized.cl). Para ello, se realiza una ejecución de su benchmark con el código original de la herramienta. El comando utilizado se muestra en el listado 5.4. Con dicha ejecución se obtiene un tiempo de 54.32ms y una velocidad de 4621.4MH/s. Si todo sale como se espera estos datos se mejorarán en la siguiente ejecución.

```
1 ./hashcat -m 0 -b --force -D2 -d1
```

Listado 5.4: Ejecución del benchmark de MD5 con la GPU CUDA.

Una vez que tenemos un primer resultado, se repite dicha ejecución utilizando el código que hemos mejorado en la sección 4.2.1. Para esta ejecución se vuelve a utilizar el comando del listado 5.4. Como resultado, comprobamos que el tiempo baja notablemente a 32.88ms y la velocidad aumenta a 30527.3 MH/s. Con estos datos observamos que se cumple lo esperado y se obtiene una mejora notable al utilizar el código en el que se aprovecha al máximo la paralelización de CUDA.

En la tabla 5.5 vemos el resumen de las ejecuciones en las que se utiliza el kernel de MD5. En este kernel en concreto la mejora no es tan grande como en el caso de SHA2-256, que

fue el que se utilizó en un principio para explicar todo el proceso de uso de la paralelización que ofrece CUDA. Aún así la diferencia entre los tiempos de ejecución es notable y se va a agradecer bastante a la hora de realizar un ataque, ya que estos suelen durar más que el benchmark y cuanto más larga sea la ejecución más se va a percibir la mejora.

Código	Dispositivo	Hash	Tiempo	Velocidad
Original	GPU Nvidia Tesla Kepler K40c	MD5	54.32 ms	4621.4 MH/s
Mejorado	GPU Nvidia Tesla Kepler K40c	MD5	32.88 ms	30527.3 MH/s

Tabla 5.5: Resultado de las ejecuciones del kernel MD5.

Estas mismas comparaciones se llevan a cabo con los 3 kernels restantes, por lo que para estos casos simplemente se muestran los resultados y las tablas pertinentes.

5.3.2 NTLM

Para la realización de las 2 ejecuciones utilizadas para probar el kernel que se usa al atacar a un hash de tipo NTLM, se utiliza el comando que se indica en el listado 5.5, el cual corre el benchmark de dicho kernel.

```
./hashcat -m 1000 -b --force -D2 -d1
```

Listado 5.5: Ejecución del benchmark de NTLM con la GPU CUDA.

Como podemos observar en la tabla 5.6, en el caso del kernel correspondiente al algoritmo de hash NTLM vuelve a ocurrir algo parecido a lo descrito en la primera función hash que se analizó (SHA2-156), en el cual la diferencia entre la ejecución del código original y la segunda, era muy elevada. Cuando se ejecuta el benchmark estamos viendo que su tiempo es algo más de 3 veces mejor en el caso de estar utilizando la mejora implementada en el código de CUDA, por lo que cuando este kernel se utilice en un ataque real se va a notar mucho la diferencia.

Código	Dispositivo	Hash	Tiempo	Velocidad
Original	GPU Nvidia Tesla Kepler K40c	NTLM	59.96 ms	8375.5 MH/s
Mejorado	GPU Nvidia Tesla Kepler K40c	NTLM	18.55 ms	53996.9 MH/s

Tabla 5.6: Resultado de las ejecuciones del kernel NTLM.

5.3.3 RIPEMD-160

A la hora de hablar del kernel de RIPEMD-160, en la tabla 5.7 se observa cómo hasta el momento, este es el caso en el que hay una mayor mejora de rendimiento, ya que el tiempo de ejecución es aproximadamente siete veces más bajo cuando se utiliza el código mejorado frente al tiempo obtenido utilizando el código original de la herramienta. Que un ataque a un hash o a una serie de hashes sea 7 veces más rápido de lo que era anteriormente supone un cambio muy notable a la hora de llevar a cabo un ataque a este hash en concreto.

Código	Dispositivo	Hash	Tiempo	Velocidad
Original	GPU Nvidia Tesla Kepler K40c	RIPEMD-160	48.04 ms	1306.0 MH/s
Mejorado	GPU Nvidia Tesla Kepler K40c	RIPEMD-160	7.08 ms	8769.8 MH/s

Tabla 5.7: Resultado de las ejecuciones del kernel RIPEMD-160.

El comando utilizado para ejecutar el benchmark del kernel perteneciente al algoritmo de hash llamado RIPEMD-160, se puede observar en el listado 5.6.

```
./hashcat -m 6000 -b --force -D2 -d1
```

Listado 5.6: Ejecución del benchmark de RIPEMD-160 con la GPU CUDA.

5.3.4 SHA3-256

Por último se realiza el análisis del kernel correspondiente al algoritmo de hash SHA3-256. Aquí vemos una mejora muy similar a la anterior, en la cual el tiempo de ejecución es aproximadamente 6 veces más bajo. Por ello las conclusiones que obtenemos para este kernel son muy similares a las obtenidas en el caso anterior, es decir, que esta diferencia se nota mucho siempre que se realice un ataque largo a un hash de este tipo. Los datos obtenidos se pueden ver en la tabla 5.8 y el comando utilizado en este caso se muestra en el listado 5.7.

Código	Dispositivo	Hash	Tiempo	Velocidad
Original	GPU Nvidia Tesla Kepler K40c	SHA3-256	66.03 ms	237.8 MH/s
Mejorado	GPU Nvidia Tesla Kepler K40c	SHA3-256	10.35 ms	1508.0 MH/s

Tabla 5.8: Resultado de las ejecuciones del kernel SHA3-256.

```
./hashcat -m 6000 -b --force -D2 -d1
```

Listado 5.7: Ejecución del benchmark de SHA3-256 con la GPU CUDA.

En resumen, podemos concluir que para los cinco kernels en los que se ha probado la nueva mejora en la paralelización del código CUDA, se obtiene una gran mejora de rendimiento, la

cual se va a percibir a la hora de realizar ejecuciones largas relacionadas con estos hashes, siendo esto lo que se buscaba desde un principio. Las mejoras no son exactamente iguales en todos los kernels, ya que Hashcat calcula para cada uno de ellos un tamaño de cuadrícula y de bloque distinto, utilizando una función llamada kernel_power() definida en el fichero src/backend.c. Por lo tanto, con este análisis se ha comprobado que dicha función no está pensada para lograr la mayor eficiencia posible, ya que con la nueva configuración planteada (un tamaño de 1024 para el grid y un tamaño de 128 para los bloques) se ha conseguido que todos los benchmarks de los kernels analizados tengan un tiempo de ejecución notablemente menor.

5.4 Configuración del motor de autoconfiguración

En esta sección se prueba la nueva posible mejora de la herramienta, modificando los valores de la base de datos del motor de autoconfiguración. Lo que se pretende es probar distintas configuraciones para la base de datos de dicho motor, con el objetivo de encontrar una configuración que permita aumentar la mejora conseguida a lo largo del desarrollo del proyecto. Para ello, se utiliza una función hash de ejemplo con el que realizar las pruebas, en este caso es el llamado SHA2-256, al cual le pertenece el número 1400 en Hashcat. En un futuro se comprobará si esta mejora funciona para otros hashes, otras plataformas y distintos tipos de ataque. Igual que ocurre en los análisis anteriores, se utiliza la GPU Tesla Kepler K40c de NVIDIA, mediante la tecnología de CUDA.

Antes de nada, vamos a recordar cuáles eran los datos de la ejecución con mejor rendimiento conseguida hasta el momento del benchmark de este hash. Estos datos se corresponden con un tiempo de ejecución igual a 9.60 ms y una velocidad de 6485.0 MH/s. El objetivo de este último análisis es disminuir ese tiempo al máximo.

Para ejecutar el benchmark se usa el mismo comando que anteriormente, el cual se corresponde con el indicado en el listado 5.8.

```
1 ./hashcat -m 1400 -b --force -D2 -d1
```

Listado 5.8: Ejecución del benchmark de SHA2-256 con la GPU CUDA.

Como resultado se obtienen los datos reflejados en la tabla 5.9, en la que podemos ver algunas de las mejores configuraciones encontradas para la base de datos del motor de autoconfiguración. En dicha tabla, se muestran tanto los tiempos obtenidos al implementar distintas configuraciones en la base de datos del código mejorado en la sección 4.2.1, como los tiempos conseguidos al implementar las mismas configuraciones en la base de datos del código original. Con esto, verificamos que obtenemos un tiempo de ejecución menor al utilizar

este motor de autoconfiguración, y además confirmamos que esta mejora de rendimiento es más elevada en el caso de que trabajemos con el código que se ha ido modificando a lo largo de todo el proyecto, en vez de si utilizamos el código original de Hashcat. Al observar la tabla podemos ver cómo se consigue una gran mejora, ya que el tiempo de ejecución del benchmark disminuye de 9.60ms a 0.04ms, en el mejor de los casos. El significado de las variables Vector_with, Kernel_Accel y Kernel_Loops está explicado en la sección 4.2.2.

Código	Vector_width	Kernel_Accel	Kernel_Loops	Tiempo	Velocidad
Original	N	A	A	69.06 ms	908.9 MH/s
Mejorado	N	A	A	9.61 ms	6473.8 MH/s
Original	1	64	64	68.43 ms	917.8 MH/s
Mejorado	1	64	64	9.59 ms	6441.4 MH/s
Original	1	16	32	8.73 ms	892.2 MH/s
Mejorado	1	16	32	5.36 ms	1443.8 MH/s
Original	1	32	16	8.84 ms	881.6 MH/s
Mejorado	1	32	16	2.73 ms	2795.6 MH/s
Original	1	4	8	4.53 ms	459.7 MH/s
Mejorado	1	4	8	0.68 ms	648.5 MH/s
Original	1	1	1	3.97 ms	102.3 MH/s
Mejorado	1	1	1	0.04 ms	140.2 MH/s

Tabla 5.9: Resultados obtenidos al modificar la configuración del motor de autoconfiguración.

Una vez visto esto para el algoritmo de hash SHA2-256, se va comprueba que la herramienta sigue funcionando correctamente. Para ello se realiza un ataque a un hash de ese tipo. Además, en un futuro se comprobará que esta mejora es eficaz para distintos tipos de ataques, distintos algoritmos de hash y en distintas plataformas. El comando utilizado para realizar dicho ataque se puede observar en el listado 5.9.

```
./hashcat -m 1400 -O -a 3
    b221d9dbb083a7f33428d7c2a3c3198ae925614d70210e28716ccaa7cd4ddb79
    ?a?a?a?a -D2 -d1
```

Listado 5.9: Ejecución de un ataque a un algoritmo de hash de tipo SHA2-256 con la GPU CUDA.

Como resultado a este ataque se obtiene una ejecución correcta en la que se relaciona el hash introducido con la contraseña ”hola”. Con esto comprobamos que los cambios realizados hasta ahora son correctos y la herramienta sigue funcionando perfectamente.

Por último, y para comprobar el porqué de esta nueva mejora, se ha vuelto a analizar un ataque real con la herramienta NVIDIA Nsight Compute, obteniendo una tabla similar a la de la sección 5.2. En este caso, podemos ver el resultado en la tabla 5.10, en la cual se muestran tanto los datos de la ejecución del ataque con el código original, como los datos

obtenidos al realizar el ataque con el código de la primera mejora, como los datos obtenidos con los cambios en el motor de autoconfiguración conseguidos en esta sección. En ella, se puede comprobar que la mejora obtenida en esta sección es debida a que se está consiguiendo un mayor aprovechamiento de la GPU, gracias al uso de una mejor configuración tanto en el motor de autoconfiguración como en los tamaños de cuadrícula y de bloque utilizados en las llamadas a los kernels.

Código	SOL Memory	SOL SM	Achieved Occupancy
Código original de Hashcat	29.59%	10.17%	37.16%
Código con el grid modificado a mano	36.13%	16.42%	44.28%
Código con el grid modificado a mano + ajuste del motor de autoconfiguración	46.28%	25.63%	57.92%

Tabla 5.10: Resultado del 2º análisis realizado con la herramienta NVIDIA Nsight Compute.

5.5 Pruebas de rendimiento en ataques reales con diferentes configuraciones

En este último análisis de rendimiento se realizan ejecuciones cambiando distintos parámetros, con el objetivo de comprobar que las mejoras implementadas en este proyecto son eficaces en la mayor parte de los casos. Para ello vamos a hacer distintas tablas con los datos obtenidos en los distintos ataques. Para hacerlo de una forma más ordenada se ha dedicado un apartado para cada plataforma de trabajo, usando aquellas que se han nombrado anteriormente, y que se encuentran disponibles en el clúster Plutón.

Para probar a fondo el rendimiento de cada una de las plataformas, se comparan los tiempos y velocidades de ejecución obtenidos al realizar los ataques desde la GPU con CUDA con el código original, desde la misma GPU con CUDA con el código mejorado, y por último desde dicha GPU, pero utilizando OpenCL con el código original de la herramienta. Es importante aclarar que, a partir de este momento, el término de ”código mejorado” hace referencia al código después de ajustar manualmente el tamaño de grid a 1024 y el de bloque a 128, y después de configurar el motor de autoconfiguración cómo se explica en la sección 4.2.2. Con estas ejecuciones conseguimos ver la diferencia entre los 3 casos. Además, dentro cada apartado correspondiente a cada una de las plataformas, existe una tabla para cada tipo de algoritmo de hash utilizado (MD5, SHA2-256, RIPEMD-160, NTLM y SHA3-256). Finalmente, en cada tabla tenemos varias filas, para cada una de ellas se han ido cambiando los parámetros de ejecución. En primer lugar, se realiza un ataque por fuerza bruta al hash de una contraseña

sencilla (”hola”), en segundo lugar se ataca al hash de una contraseña más compleja (”as5e@”), y finalmente se realiza un ataque a 2 hashes a la vez, pertenecientes a contraseñas complejas (”%Bh3f” y ”rG3&q”). No se han realizado pruebas con contraseñas más largas, ya que el tiempo estimado de su ataque es de más de 3 horas, por lo que no sería viable hacer tantas pruebas de ese estilo. Para finalizar, se realiza un ataque extra, en el que se comprueba el funcionamiento de estas mejoras en un ataque que no sea por fuerza bruta. En este caso se trata de un ataque por diccionario basado en reglas. Este diccionario es el conocido como ”rockyou” y el conjunto de reglas utilizado es uno pensado para este diccionario en concreto. Además, en este último caso no se realiza un ataque sobre uno o dos hashes, sino que se ataca a exactamente 548686 hashes a la vez. Esto es posible gracias a la creación un diccionario para cada tipo de algoritmo de hash nombrado anteriormente, mediante un programa sencillo escrito en Python creado exclusivamente para hacer esta última prueba.

A partir de este momento, y con el objetivo de lograr una correcta visualización de las tablas, se utilizan los términos explicados a continuación:

- **Caso 1**: Ejecución del ataque por fuerza bruta realizado al hash perteneciente a la contraseña ”hola”.
- **Caso 2**: Ejecución del ataque por fuerza bruta realizado al hash perteneciente a la contraseña ”as5e@”.
- **Caso 3**: Ejecución del ataque por fuerza bruta realizado a los hashes pertenecientes a las contraseñas ”%Bh3f” y ”rG3&q”.
- **Caso 4**: Ejecución del ataque por diccionario utilizando reglas. El diccionario utilizado se llama rockyou y el ataque se realiza con el objetivo de resolver 548686 hashes.

5.5.1 Plataforma Tesla Kepler K40c de Nvidia

Para comenzar a realizar las distintas pruebas de rendimiento, se utiliza la GPU de Nvidia Tesla Kepler K40c. A lo largo de esta sección se explican uno a uno los resultados que se obtienen en cada tabla, y al final se hace un pequeño resumen en el que se explica si la mejora afecta a todos los casos o no.

MD5

Comenzamos con la tabla 5.11, la cual muestra los datos obtenidos después de realizar todas las ejecuciones explicadas anteriormente. En este caso se trata de hashes de tipo MD5. Como resultado vemos algo lógico, ya que podemos observar que para todas las ejecuciones realizadas, el tiempo de ejecución hecho con CUDA y con nuestro código mejorado es el más

bajo. Además podemos comprobar que, como se explicó anteriormente, cuanto más tiempo dura un ataque, más se va a notar la mejora. Aunque esto ya se pueda observar con los 3 primeros casos, si vemos el cuarto notamos que la diferencia es de 4 minutos con respecto al código original de Hashcat, y de 6 minutos con la ejecución de OpenCL. Si esto lo comprobamos en un ataque que dure varias horas, es muy posible que se consiga reducir el tiempo de ejecución más de una hora.

	Tiempo CUDA código original	**Velocidad CUDA código original**	**Tiempo CUDA código mejorado**	**Velocidad CUDA código mejorado**	**Tiempo OpenCL**	**Velocidad OpenCL**
Caso 1	0.04 ms	129.5 MH/s	0.02 ms	153.5 MH/s	0.04 ms	113.0 MH/s
Caso 2	27 s	139.7 MH/s	21 s	159.1 MH/s	27 s	123.0 MH/s
Caso 3	40 s	142.5 MH/s	34 s	163.9 MH/s	44 s	126.6 MH/s
Caso 4	6 min 7 s	1394.20 MH/s	2 min 18 s	7213.83 MH/s	8 min 3 s	765.2 MH/s

Tabla 5.11: Datos de los ataques a hashes de tipo MD5 en la GPU Tesla Kepler K40c.

SHA2-256

En la tabla 5.12, vemos el resultado de las ejecuciones de los ataques por fuerza bruta a hashes de tipo SHA-25. En este caso, la diferencia que hay entre cada tipo de código, es notable, ya que cuanto más largas sean las ejecuciones, la diferencia entre sus tiempos de ejecución va siendo mayor. Para este tipo de algoritmo de hash vemos que, a la hora de realizar ataques largos, la diferencia del tiempo de ejecución entre los distintos casos es muy elevada. En este caso en concreto estamos reduciendo más de 17 minutos el tiempo de ejecución si lo comparamos con la ejecución con OpenCL y 15 minutos si lo comparamos con la ejecución del código original con CUDA. Con estos datos verificamos que las mejoras implementadas en la fase de desarrollo son muy efectivas.

	Tiempo CUDA código original	**Velocidad CUDA código original**	**Tiempo CUDA código mejorado**	**Velocidad CUDA código mejorado**	**Tiempo OpenCL**	**Velocidad OpenCL**
Caso 1	0.04 ms	120.4 MH/s	0.04 ms	124.0 MH/s	0.06 ms	101.2MH/s
Caso 2	26 s	129.3 MH/s	24 s	138.4 MH/s	31 s	108.2 MH/s
Caso 3	45 s	126.0 MH/s	41 s	135.4 MH/s	50 s	112.0 MH/s
Caso 4	18 min 32 s	291.3 MH/s	3 min 55 s	2001.4 MH/s	21 min 15 s	265.0 MH/s

Tabla 5.12: Datos de los ataques a hashes de tipo SHA2-256 en la GPU Tesla Kepler K40c.

NTLM

Para el caso de NTLM, la situación es similar a la descrita en los apartados anteriores. En este, igual que en el resto, se puede comprobar cómo el dispositivo que más tiempo tarda en realizar los ataques, es el que usa OpenCL. Esto confirma los resultados del análisis de rendimiento explicado en la sección 5.1, en el cual se comprobaba que las ejecuciones realizadas con CUDA tardaban menos que aquellas hechas con OpenCL. Los resultados de las ejecuciones correspondientes al algoritmo de hash NTLM se reflejan en la tabla 5.13.

	Tiempo CUDA código original	**Velocidad CUDA código original**	**Tiempo CUDA código mejorado**	**Velocidad CUDA código mejorado**	**Tiempo OpenCL**	**Velocidad OpenCL**
Caso 1	0.03 ms	139.6 MH/s	0.02 ms	166.4 MH/s	1 s	124.2 MH/s
Caso 2	19 s	140.3 MH/s	16 s	168.3 MH/s	22 s	124.8 MH/s
Caso 3	46 s	143.3 MH/s	40 s	166.3 MH/s	53 s	126.3 MH/s
Caso 4	6 min 33 s	1611.6 MH/s	2 min 14 s	8449.2 MH/s	7 min 5 s	850.6 MH/s

Tabla 5.13: Datos de los ataques a hashes de tipo NTLM en la GPU Tesla Kepler K40c.

RIPEMD-160

En el caso del algoritmo de hash RIPEMD-160 ocurre exactamente lo mismo que en el resto de los casos estudiados. Esto lo observamos en la tabla 5.14, en la cual se ve que las ejecuciones con más rendimiento son aquellas que se corresponden con el código mejorado de CUDA, y las que menos rendimiento alcanzan son las que se corresponden con las ejecutadas por la GPU que usa OpenCL. Además, vemos que cuanto más larga es la ejecución, más notable es la mejora.

	Tiempo CUDA código original	**Velocidad CUDA código original**	**Tiempo CUDA código mejorado**	**Velocidad CUDA código mejorado**	**Tiempo OpenCL**	**Velocidad OpenCL**
Caso 1	0.04 ms	128.5 MH/s	0.03 ms	137.3 MH/s	0.04 ms	116.7MH/s
Caso 2	24 s	140.7 MH/s	22 s	152.3 MH/s	26 s	126.2 MH/s
Caso 3	41 s	138.8 MH/s	36 s	155.1 MH/s	45 s	123.7 MH/s
Caso 4	10 min 43 s	434.6 MH/s	3 min 5 s	3925.9 MH/s	12 min 23 s	369.1 MH/s

Tabla 5.14: Datos de los ataques a hashes de tipo RIPEMD-160 en la GPU Tesla Kepler K40c.

SHA3-256

Por último, tenemos la tabla 5.15, cuyos datos se obtienen después de realizar las ejecuciones de los ataques a hashes de tipo SHA2-256. Estas ejecuciones actúan de la misma forma que las anteriores, por lo que la conclusión es la misma. Para esta función hash en concreto, se puede observar que en general los tiempos de ejecución son mayores que en el resto de casos, pero esto es algo propio de este tipo de hash. Gracias a esta característica podemos ver que en los casos en los que los ataques sean de más de una hora con el código original, con nuestra mejora se consigue reducir el tiempo de ejecución más de 50 minutos, consiguiendo así un aumento notable en el rendimiento de la aplicación.

	Tiempo CUDA código original	**Velocidad CUDA código original**	**Tiempo CUDA código mejorado**	**Velocidad CUDA código mejorado**	**Tiempo OpenCL**	**Velocidad OpenCL**
Caso 1	0.11 ms	77.8 MH/s	0.1 ms	85.8 MH/s	0. 1 ms	80.5 MH/s
Caso 2	1 m 8 s	81.8 MH/s	37 s	89.8 MH/s	40 s	84.4 MH/s
Caso 3	1 m 9 s	82.0 MH/s	1 min 2 s	87.97 MH/s	1 min 6 s	85.3 MH/s
Caso 4	1 h 6 min	103.4 MH/s	14 min 43 s	104.2 MH/s	1 h 7 min	103.2 MH/s

Tabla 5.15: Datos de los ataques a hashes de tipo SHA3-256 en la GPU Tesla Kepler K40c.

Una vez realizados todos los ataques y analizados sus resultados, podemos llegar a la conclusión de que las mejoras implementadas en la fase de desarrollo se cumplen para el 100% de los casos estudiados en el dispositivo Tesla Kepler K40c. Visto esto, podemos llegar a la conclusión de que las mejoras afectan a la gran parte de los casos de uso de Hascat, pero para asegurarnos de que esta afirmación es cierta en otros dispositivos, se van a repetir todos los ataques anteriores en otra plataforma. El análisis realizado en otro dispositivo y sus conclusiones están explicadas en la sección 5.5.2.

5.5.2 Plataforma Tesla Kepler K20m de Nvidia

Como se ha explicado anteriormente, en este apartado se ejecutan los mismos ataques que en la sección 5.5.1, pero en otro dispositivo distinto. Esto se hace con el objetivo de comprobar que las mejoras realizadas en el código de Hashcat no sólo funcionan para el dispositivo utilizado en el resto de análisis de rendimiento. Para llevar a cabo todas las ejecuciones de esta sección, se utiliza la GPU de Nvidia Tesla Kepler K20m, la cual también está disponible en el clúster Pluton.

Para no repetir las mismas conclusiones en cada una de las tablas, lo que se hace en esta sección es poner todas ellas juntas, y finalmente llegar a una conclusión común. Las tablas correspondientes a este análisis de rendimiento son la 5.16, la 5.17, la 5.18, la 5.19 y la 5.20, las cuales se corresponden con las evaluaciones sobre los algoritmos MD5, SHA2-256, NTLM,

RIPEMD-160 y SHA3-356, respectivamente..

	Tiempo CUDA código original	Velocidad CUDA código original	Tiempo CUDA código mejorado	Velocidad CUDA código mejorado	Tiempo OpenCL	Velocidad OpenCL
Caso 1	0.03 ms	108.5 MH/s	0.02 ms	132.8 MH/s	0.04 ms	103.0 MH/s
Caso 2	27 s	122.0 MH/s	24 s	140.2 MH/s	31 s	109.2 MH/s
Caso 3	46 s	122.9 MH/s	40 s	139.7 MH/s	52 s	108.9 MH/s
Caso 4	8 min 14 s	1155.7 MH/s	2 min 21 s	6649.5 MH/s	8 min 12 s	1183.1 MH/s

Tabla 5.16: Datos de los ataques a hashes de tipo MD5 en la GPU Tesla Kepler K20m.

	Tiempo CUDA código original	Velocidad CUDA código original	Tiempo CUDA código mejorado	Velocidad CUDA código mejorado	Tiempo OpenCL	Velocidad OpenCL
Caso 1	0.05 ms	102.0 MH/s	0.03 ms	116.4 MH/s	0.05 ms	88.6 MH/s
Caso 2	31 s	109.9 MH/s	28 s	118.9 MH/s	34 s	98.9 MH/s
Caso 3	53 s	107.8 MH/s	48 s	116.7 MH/s	58 s	97.0 MH/s
Caso 4	16 min 56 s	478.1 MH/s	4 min 17 s	2226.9 MH/s	23 min 33 s	333.8 MH/s

Tabla 5.17: Datos de los ataques a hashes de tipo SHA2-256 en la GPU Tesla Kepler K20m.

	Tiempo CUDA código original	Velocidad CUDA código original	Tiempo CUDA código mejorado	Velocidad CUDA código mejorado	Tiempo OpenCL	Velocidad OpenCL
Caso 1	0.03 ms	124.5 MH/s	0.02 ms	145.0 MH/s	0.04 ms	109.3 MH/s
Caso 2	22 s	123.5 MH/s	18 s	143.7 MH/s	25 s	108.4 MH/s
Caso 3	54 s	124.3 MH/s	46 s	140.9 MH/s	1 min 0 s	110.3 MH/s
Caso 4	7 min 11 s	1383.1 MH/s	2 min 11 s	7848.0 MH/s	7 min 14 s	1362.2 MH/s

Tabla 5.18: Datos de los ataques a hashes de tipo NTLM en la GPU Tesla Kepler K20m.

	Tiempo CUDA código original	Velocidad CUDA código original	Tiempo CUDA código mejorado	Velocidad CUDA código mejorado	Tiempo OpenCL	Velocidad OpenCL
Caso 1	0.04 ms	112.3 MH/s	0.03 ms	119.9 MH/s	0.04 ms	102.7 MH/s
Caso 2	28 s	120.1 MH/s	26 s	129.5 MH/s	31 s	107.3 MH/s
Caso 3	46 s	121.4 MH/s	43 s	131.6 MH/s	52 s	105.9 MH/s
Caso 4	12 min 7 s	704.9 MH/s	3 min 17 s	3514.0 MH/s	13 min 39 s	610.8 MH/s

Tabla 5.19: Datos de los ataques a hashes de tipo RIPEMD-160 en la GPU Tesla Kepler K20m.

	Tiempo CUDA código original	Velocidad CUDA código original	Tiempo CUDA código mejorado	Velocidad CUDA código mejorado	Tiempo OpenCL	Velocidad OpenCL
Caso 1	0.12 ms	66.1 MH/s	0.1 ms	71.7 MH/s	0.11 ms	67.6 MH/s
Caso 2	1 min 21 s	70.2 MH/s	44 s	76.5 MH/s	48s	70.1 MH/s
Caso 3	1 min 22 s	70.6 MH/s	1 min 13 s	77.0 MH/s	1 min 18 s	72.1 MH/s
Caso 4	41 min 2 s	158.4 MH/s	29 min 8 s	159.1 MH/s	46 min 3 s	158.1 MH/s

Tabla 5.20: Datos de los ataques a hashes de tipo SHA3-256 en la GPU Tesla Kepler K20m.

Una vez vistos y estudiados todos los datos de cada una de de las ejecuciones anteriores, realizadas con un dispositivo diferente al utilizado en el resto de análisis de rendimiento, se puede llegar a la conclusión de que las mejoras realizadas en la fase de desarrollo funcionan en la gran mayoría de ataques que permite hacer la herramienta Hashcat. Tanto es así, que para el 100% de los casos analizados en este proyecto, el tiempo de ejecución al utilizar nuestro código CUDA mejorado, ha sido menor que los tiempos obtenidos al ejecutar exactamente los mismos ataques con el código original de la herramienta. Además, también se demuestra que ejecutar la herramienta mediante una GPU de NVIDIA con OpenCL es más lento que si lo hacemos usando CUDA.

Por último, es importante recalcar, que cuanto más larga sea la ejecución de un ataque, más notable será la mejora de rendimiento. Esto se puede comprobar si observamos el caso 4 de cada una de las tablas mostradas en esta sección, en las cuales podemos observar que para ataques de varios minutos se consigue reducir el tiempo de manera considerable, llegando a conseguir en algún caso una mejora de más de 50 minutos.

Capítulo 6

Conclusiones

Una vez terminadas todas las fases planeadas para este proyecto y realizados todos los análisis de rendimiento pertinentes, se puede dar este trabajo como finalizado y llegar a una conclusión final.

Antes de nada, es importante recordar los objetivos principales del trabajo. En primer lugar, se buscaba la realización de un estudio a fondo de la herramienta Hashcat, con el propósito de conocer cómo funciona por dentro. Este primer objetivo estaba pensado para que en un futuro, conociendo la forma en la que está implementada la herramienta, fuese más sencillo utilizar la paralelización de CUDA con la intención de mejorar el rendimiento general de la aplicación.

Una vez realizado dicho estudio y entendido el funcionamiento de Hashcat, se realizan una serie de cambios en el código de la aplicación. Estos nuevos cambios nos sirven para cumplir el objetivo principal de nuestro proyecto, ya que con la nueva implementación explicada a lo largo de esta memoria, se consigue aumentar el rendimiento de la aplicación en un 100% de los casos analizados. Es importante destacar que no sólo se han probado estos cambios en un dispositivo y para una funcionalidad específica de la herramienta, sino que se han hecho una gran cantidad de pruebas de rendimiento en las que se han modificado los parámetros de cada ataque, tratando de cubrir el mayor número de casos posibles de los que nos ofrece la aplicación.

En este punto vamos a tener en cuenta una serie de datos calculados con los resultados de los experimentos realizados a lo largo del proyecto. Dichos datos son: la mejora máxima, la mejora mínima y la mejora media. Cuando hablamos de "**mejora**" nos referimos a la diferencia de tiempo entre la ejecución con el código original y la ejecución con nuestro código modificado después de haber implementado todos los cambios explicados en el capítulo 4. Por ejemplo, si tenemos una ejecución en la que su tiempo con el código original es de 10 minutos y 43 segundos (643 segundos) y su tiempo con el código modificado de 3 minutos y 5 segundos (185 segundos), tendríamos una mejora de 643 - 185 = 458 segundos (7 minutos

y 38 segundos), ya que hemos conseguido disminuir 458 segundos el tiempo de ejecución en ese caso en concreto. Las ejecuciones utilizadas para llevar a cabo estos cálculos son aquellas mostradas en la secciones 5.5, ya que se corresponden a ataques reales medidos después de haber implementado todas las mejoras explicadas anteriormente. Además, se ha decidido utilizar únicamente aquellas ejecuciones con duración mayor o igual a 5 minutos y menor o igual a 1 hora y 15 minutos en su medición realizada con el código original de Hashcat. Esto se hace para evitar aquellas ejecuciones que son demasiado cortas y que por este motivo no reflejan una mejora tan notable.

A la hora de hablar de la mejora mínima conseguida, nos encontramos ante una mejora de 3 minutos y 49 segundos, dada en el ataque por diccionario a hashes de tipo MD5 en la GPU Tesla Kepler K40c. En este caso partíamos de un tiempo de ejecución de 6 minutos y 7 segundos obtenido con el código original de la herramienta y un tiempo de 2 minutos y 18 segundos para la ejecución realizada con el código modificado. Por otro lado, la máxima mejora obtenida ha sido de 51 minutos y 17 segundos, la cual se corresponde con la ejecución del ataque por diccionario a hashes de tipo SHA3-256 en la GPU Tesla Kepler K40c. En este caso tenemos un tiempo de 1 hora y 6 minutos con el código original y un tiempo de 14 minutos y 43 segundos con el código modificado. Por último, la mejora media obtenida de todas las ejecuciones analizadas es de 12 minutos y 35 segundos. Esta es una gran mejora, debido a que la media de tiempo de ejecución con el código original de estos ataques es de 20 minutos. Con esto obtenemos que si un ataque tarda 20 minutos en ejecutarse con el código original de Hashcat, vamos a obtener una mejora de 12 minutos y 35 segundos de media al utilizar nuestro código mejorado, lo cual supone un aumento de rendimiento bastante notable. Estos datos explicados anteriormente están reflejados en la tabla 6.1.

Mejora media	**Mayor mejora**	**Menor mejora**
12 min 35 s	51 min 17 s	3 min 49 s

Tabla 6.1: Mejoras conseguidas al utilizar nuestro código modificado frente al código original.

Además de la máxima mejora obtenida, correspondiente a un ataque realizado a hashes de tipo SHA3-256, las siguientes mejoras más altas conseguidas han sido aquellas relacionadas con ataques a hashes de tipo SHA2-256 y SHA3-256. Esto es debido a que, por la forma en la que están escritos sus kernels, el reparto de trabajo que hemos escogido para todas las llamadas a kernels de Hashcat es algo más beneficioso para estos tipos de hashes. Además, también se puede comprobar que cuanto más largos sean los ataques, más notable será la mejora. Es decir, en una ejecución corta, la diferencia entre los tiempos de ejecución con el código original y con el modificado, va a ser mínima, mientras que si realizamos ataques cuyo tiempo de ejecución con el código original sea de más de un día, conseguiremos reducir este

tiempo varias horas, lo cual hará más notable la mejora.

Los motivos principales por los cuales se consigue disminuir el tiempo de ejecución en el 100% de los casos probados están relacionados con el reparto de la carga de trabajo, ya que como se ha explicado en el capítulo 4, Hashcat no está aprovechando al máximo las funcionalidades de CUDA, por lo que no está consiguiendo un reparto de trabajo lo más óptimo posible. Con las mejoras explicadas e implementadas a lo largo del proyecto se consigue mejorar este aspecto de la herramienta, dando como resultado una gran mejora en el rendimiento de Hashcat.

Otro punto importante a tener en cuenta una vez acabado el proyecto, es el hecho de que en un principio se pensaba que Hashcat ya no utilizaba código CUDA para nada, debido a que en sus foros oficiales se explica que la herramienta ha sido portada en su totalidad a OpenCL. Fue en la mitad del estudio de la aplicación cuando nos dimos cuenta de que la herramienta sí que utiliza CUDA en su versión más actual. A pesar de esto, se ha comprobado que no se estaban aprovechando al máximo las funcionalidades de paralelización que nos ofrece dicha tecnología, por lo que este ha sido un punto clave para lograr el objetivo principal del proyecto.

Con esto conseguimos que, si disponemos de una GPU de Nvidia y necesitamos recuperar una contraseña a partir de un hash, lo hagamos de una forma mucho más rápida y eficiente.

Como última conclusión, también se demuestra que el uso de la tecnología CUDA en dispositivos de la marca Nvidia en Hashcat es más eficiente que el uso de OpenCL, a pesar de que en los foros de Hashcat [17] se diga que con OpenCL no se pierde rendimiento si las GPUs son modernas. Esto no es así, porque a lo largo de este trabajo se han utilizado 3 GPUs modernas distintas (Tesla Kepler K40c, Tesla Kepler K20m en el clúster y GeForce GTX 1050 en el ordenador portátil) y en todas ellas se estaba perdiendo rendimiento antes de haber implementado la mejora. Un punto a favor del uso de OpenCL es que permite la utilización de dispositivos que no sean GPUs, como es el caso de las CPUs. Aún así es importante explicar que no era necesario dicho cambio de forma tan drástica, si no que lo que realmente compensaba y se podría haber hecho era codificar la herramienta de forma que se aprovechen al máximo tanto las ventajas de OpenCL como de CUDA.

6.1 Líneas futuras

En esta breve sección se explican una serie de fases con las que se puede continuar este trabajo en un futuro, las cuales podrían aumentar algo más la mejora de rendimiento conseguida a lo largo de todo el proyecto.

En primer lugar, es importante recordar que Hashcat utiliza código genérico en sus kernels para facilitar la compatibilidad entre OpenCL y CUDA, redefiniendo alguna función en tiempo de ejecución. Con esto es posible que se esté perdiendo algo de rendimiento, ya que no se están

utilizando algunas características que ofrece CUDA debido a que no son compatibles con el código de OpenCL. Una primera opción sería la realización de un estudio con el objetivo de identificar las características de CUDA que mejoren el rendimiento de los kernels y su posterior implementación.

Una vez hecho esto, se podría continuar haciendo un estudio de varias estrategias de paralelización existentes, viendo cual de ellas puede mejorar el rendimiento general de Hashcat. Una vez hecho este estudio se haría un análisis de rendimiento con distintas estrategias de paralelización, viendo cual de ellas es la más apropiada para Hashcat, consiguiendo así una nueva mejora de rendimiento.

Apéndices

Apéndice A

Acceso al código mejorado de Hashcat y ficheros importantes

A lo largo de este capítulo se explica brevemente cómo acceder al código resultante de la realización de este proyecto y se nombran los ficheros más importantes del proceso de desarrollo de este trabajo.

En primer lugar si se quiere acceder al código de la herramienta con las mejoras explicadas en este documento lo único que hay que hacer es clonar el siguiente repositorio público de GitHub: `https://github.com/pedroarruti/Hashcat-CUDA`

Una vez clonada la herramienta ya podemos hacer uso de la misma. Es importante recordar que todas las mejoras que se llevan a cabo a lo largo de este trabajo han sido pensadas para dispositivos de tipo GPU con CUDA del fabricante Nvidia, por lo que para aprovechar estas mejoras se debe correr el código desde un dispositivo de ese tipo.

Para poder iniciar la herramienta es necesario ejecutar el comando ”make”, que nos creará el ejecutable llamado ”hashcat” con el que se realizan los distintos ataques.

Los ficheros más importantes nombrados en este documento son los siguientes:

- hashcat.hctune: base de datos del motor de autoconfiguración
- src/Makefile: Makefile de toda la herramienta
- src/main.c: main de la herramienta
- src/backend.c: fichero en donde se trabaja con los kernels
- OpenCL/m00000_a3-optimized.cl: kernel de MD5
- OpenCL/m01000_a3-optimized.cl: kernel de NTLM
- OpenCL/m01400_a3-optimized.cl: kernel de SHA-2 256

- OpenCL/m01400_a3-optimized.cu: kernel CUDA de SHA-2 256
- OpenCL/m06000_a3-optimized.cl: kernel de RIPEMD-160
- OpenCL/m17400_a3-optimized.cl: kernel de SHA-3 256

Otro archivo importante, pero que no está en el repositorio, es el llamado hashcat.potfile, que es aquel en el que se guardan los pares hash-contraseña una vez que hayamos ejecutado un ataque y nos haya producido una salida exitosa. Este fichero tiene la funcionalidad de almacenar todas las contraseñas que ya hemos crackeado, con el objetivo de que si en un futuro intentamos crackear dicho hash de nuevo, nos indicará que está almacenado en este archivo y no se realizará el ataque. Una vez que se cree lo podemos encontrar en el directorio principal de la herramienta.

Lista de acrónimos

GPU *Graphics Processing Unit.*

CPU *Central Processing Unit.*

CUDA *Compute Unified Device Architecture.*

OpenCL *Open Computing Language.*

API *Application Programming Interface.*

GDB *GNU Debugger.*

NVCC *NVIDIA CUDA Compiler.*

NVRTC *NVIDIA Runtime Compilation.*

TFG *Trabajo de Fin de Grado.*

UDC *Universidad de A Coruña.*

Glosario

Thread También conocido como hilo de ejecución, es la unidad básica de ejecución de un programa y está formado por una serie de instrucciones.

Clúster Conjunto de ordenadores interconectados que trabajan en conjunto de forma que el usuario los ve como una única máquina y que está diseñado para ofrecer alta capacidad de cómputo.

Kernel Software fundamental que constituye el puente entre las aplicaciones y el procesamiento de datos realizado a nivel de hardware.

Crackear contraseñas Proceso de recuperación de una o más contraseñas que están almacenadas en un equipo. Este proceso se puede realizar con distintos tipos de ataques y el resultado de estos se compara con el hash de la contraseña real, al cual se tiene acceso.

Benchmark También conocido como prueba de rendimiento, es una técnica utilizada para medir el rendimiento de un sistema o uno de sus componentes.

Bibliografía

[1] Xataka, "Cpu: qué es, cómo es y para qué sirve," 2020. [En línea]. Disponible en: https://www.xataka.com/basics/cpu-que-como-sirve

[2] ——, "Las gpu como pasado, presente y futuro de la computación," 2014. [En línea]. Disponible en: https://www.xataka.com/componentes/las-gpu-como-pasado-presente-y-futuro-de-la-computacion

[3] A. G. Konheim, *Hashing in Computer Science: Fifty Years of Slicing and Dicing*, 1st ed. Wiley-Interscience, 2010.

[4] J. Cheng, M. Grossman, and T. Mckercher, *Professional CUDA® C Programming*, 1st ed. John Wiley & Sons, Inc, 2014.

[5] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*, 1st ed. Addison-Wesley, 2011.

[6] Hashcat, "Hashcat-github," 2016. [En línea]. Disponible en: https://github.com/hashcat/hashcat

[7] ——, "Hashcat-wiki hashcat," 2012. [En línea]. Disponible en: https://hashcat.net/wiki/doku.php?id=hashcat

[8] ——, "Hashcat-wiki oclhashcat," 2012. [En línea]. Disponible en: https://hashcat.net/wiki/doku.php?id=oclhashcat

[9] Sourceware, "Introduction to profiling," 2020. [En línea]. Disponible en: https://sourceware.org/binutils/docs/gprof/Introduction.html#Introduction

[10] S. L. G. P. B. K. M. K. McKusick, "Gprof: a call graph execution profiler," 2000. [En línea]. Disponible en: https://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf

[11] Intel, "Intel parallel studio xe," 2020. [En línea]. Disponible en: https://software.intel.com/en-us/parallel-studio-xe

[12] Nvidia, "An even easier introduction to cuda," 2017. [En línea]. Disponible en: https://devblogs.nvidia.com/even-easier-introduction-cuda/

[13] K. S. Rubin, *Essential Scrum: A Practical Guide to the Most Popular Agile Process*, 1st ed. Addison-Wesley, 2012.

[14] S. Kneafsey, "A short history of scrum," 2020. [En línea]. Disponible en: https://www.thescrummaster.co.uk/scrum/short-history-scrum/

[15] Hashcat, "Hashcat," 2016. [En línea]. Disponible en: https://hashcat.net/hashcat/

[16] ——, "Hashcat development guide," 2020. [En línea]. Disponible en: https://github.com/hashcat/hashcat/blob/master/docs/hashcat-plugin-development-guide.md

[17] Atom, "Cuda?" 2017. [En línea]. Disponible en: https://hashcat.net/forum/thread-6712.html

[18] ——, "The autotune engine," 2016. [En línea]. Disponible en: https://hashcat.net/forum/thread-5184.html