eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Ingeniería de Computadores

Trabajo de Fin de Grado

Implementación eficiente en GPGPUs de la prueba de trabajo RandomX de la criptomoneda Monero

Autor

Julen Suárez

2020

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Ingeniería de Computadores

Trabajo de Fin de Grado

Implementación eficiente en GPGPUs de la prueba de trabajo RandomX de la criptomoneda Monero

Autor

Julen Suárez

Director

Jose A. Pascual

Resumen

Las criptomonedas son monedas digitales o virtuales que utilizan criptografía para garantizar su seguridad. La mayoría de ellas se basan en sistemas distribuidos y recurren a una *blockchain* para almacenar información relacionada con la plataforma, como la creación de nuevos tokens o las transacciones realizadas. De esta manera se garantiza que dicha información no puede ser modificada y se asegura la anonimidad de las transacciones. En todas estas plataformas, los tokens se crean y se transfieren a través de la resolución de problemas matemáticos muy costosos computacionalmente. A estos problemas se les denomina prueba de trabajo o Proof of Work (PoW).

Cada plataforma de criptomonedas tiene características propias que las distingue del resto, pero en este proyecto nos centraremos en Monero. La principal característica de esta criptomoneda es su capacidad de disociar las transacciones de las direcciones de los emisores y de los receptores. Además, Monero implementa una prueba de trabajo específicamente diseñada para ser ejecutada de manera eficiente en CPUs y penalizando su rendimiento en ASICs. La PoW se denomina RandomX y a pesar de no estar diseñada para GPGPUS existe una implementación realizada en CUDA. El objetivo del proyecto consiste en analizar Monero y RandomX en profundidad y en investigar posibles mejoras que incrementen el rendimiento de la PoW en GPGPUs.

Índice general

Índice de figuras

Índice de tablas

1. CAPÍTULO

Introducción

Desde el año 2009, cuando se creó Bitcoin [Nakamoto, 2008], el número de criptomedas ha crecido de manera sustancial. Se puede definir una criptomoneda como una moneda digital o virtual que utiliza criptografía para garantizar su seguridad. El uso de estas técnicas hacen prácticamente imposible la falsificación o el *doble gasto* (double spending). En la practica, estos sistemas de criptomonedas permiten la realización de pagos seguros online a través de *tokens* que son almacenados en libros de contabilidad (ledgers) internos del sistema. En la actualidad existen más de 5563 criptomonedas diferentes con una capitalización de más de 271 billones de dolares.

La mayoría de las criptomonedas se basan en sistemas distribuidos y utilizan la tecnología *blockchain* para almacenar la información relativa, tanto a la creación de nuevos tokens, como a las transacciones realizadas y así garantizar que dicha información no puede ser modificada, y la anonimidad de las transacciones realizadas. Una característica fundamental de este tipo de sistemas de criptomonedas es que no son gestionadas por ninguna autoridad central haciéndolos inmunes a la manipulación o a la ingerencia de algún gobierno.

En todas las criptomonedas, los tokens nuevos son generados resolviendo problemas matemáticos que son computacionalmente costosos. Estos problemas son utilizados para verificar las transacciones realizadas en el sistema y como recompensa, se generan nuevos tokens que son asignados a quien lo haya resuelto. Este proceso se conoce prueba de trabajo o proof of work (PoW) o, coloquialmente, como minado de bloques y, por poner un ejemplo, en Bitcoin consiste en encontrar un número (*nonce*) al que tras aplicarle una

función de *hashing* (SHA256 [Gilbert and Handschuh, 2004]), se obtenga una secuencia que cumple un cierto criterio, en este caso, un determinado número de ceros en el inicio.

Cada criptomoneda posee diferentes características que las hacen diferentes al resto, pero en este trabajo nos centraremos en un subconjunto de ellas que garantizan la completa anonimidad de las transacciones. En general, se puede decir que todas las criptomonedas garantizan la anonimidad de las transacciones utilizando criptografía pública (una transacción se ve como un pago emitido por una dirección generada a partir de una clave publica). Sin embargo todas las transacciones son fácilmente rastreables hasta una dirección (clave publica) que podría ser asociada con alguien en el *mundo real*. Con el objetivo de solucionar esta *debilidad* surgieron un conjunto de criptomonedas en las cuales es imposible rastrear las transacciones hasta una única dirección.

Dentro de ese subconjunto, Monero es la criptomoneda más utilizada y sus token se denominan *XMR*. Como se ha descrito anteriormente, Monero es una de las pocas plataformas que tiene una blockchain privada y que oculta las direcciones tanto del emisor como del receptor cuando se realizan transacciones. Sin embargo, aunque la completa anonimidad sea uno de sus principales atractivos, hay otro que la está haciendo cada vez más popular. En los últimos años, el minado de bloques se ha convertido en una actividad muy popular debido a que permite conseguir tokens verificando las transacciones que se producen. Esto ha provocado el desarrollo de hardware específico (ASICs) que permiten resolver los problemas de la PoW de una manera mucho más eficiente, en términos de energía consumida y tiempo de cómputo, que en una CPU convencional o que en una GPGPU (o GPU). Esto ha provocado que el minado de algunas criptomonedas quede fuera del alcance de la mayoría debido a que no dispone de ese hardware específico.

Con el objetivo de evitar este tipo de minado y permitir que cualquiera pueda minar XMRs, Monero implementa desde Noviembre de 2019, RandomX. Este algoritmo PoW "anti-ASIC" está específicamente diseñado para ser ejecutado en procesadores de proposito general (Intel, AMD, ARM, Power) y penaliza severamente su ejecución en otro tipo de arquitecturas hardware como pueden ser las FPGAs, los ASICs o incluso las GPUs.

RandomX se ha diseñado teniendo en cuenta características específicas como son la jerarquía de memoria o la predicción de ramas entre otras presentes en procesadores modernos y utiliza de manera muy intensa el algoritmo criptográfico AES implementado de manera nativa en hardware en la mayoría de los procesadores. De forma general, RandomX genera programas virtuales que deben ser ejecutados con el objetivo de obtener como salida un hash que cumple un criterio de dificultad impuesto por la red. A pesar de este enfoque

orientado a CPUs, existe una implementación (oficial) de RandomX para ser ejecutada sobre GPUs de NVIDIA utilizando CUDA.

El objetivo de este proyecto es analizar en profundizad el funcionamiento de RandomX y analizar el rendimiento que se consigue cuando se ejecuta sobre GPUs, en particular, la implementación sobre CUDA (existe otra sobre OpenCL). Así mismo, se comparará el rendimiento conseguido con la ejecución sobre una CPU convencional y se estudiarán e implementarán posibles mejoras que aprovechen la arquitectura especifica de las GPUs.

2. CAPÍTULO

Documento de Objetivos del Proyecto

El objetivo de este proyecto es el estudio de la plataforma de criptomonedas Monero y de su prueba de trabajo RandomX, así como el estudio y optimización de una implementación open source de RandomX realizada en CUDA para GPUs.

En un primer paso, se deberá realizar la planificación del proyecto, dividiéndolo en paquetes de trabajo y tareas. Así mismo, se deberá diseñar un plan de riesgo ante cualquier posible contingencia.

A continuación, se deberá estudiar y analizar el funcionamiento de Monero, incluyendo los mecanismos que utiliza para garantizar la anonimidad de las transacciones. El objeto de este estudio es entender cómo funciona esta criptomoneda, y en concreto, su prueba de trabajo denominada RandomX.

A pesar de que RandomX está específicamente diseñado para su ejecución eficiente en CPUs, en este proyecto nos centraremos en el estudio de una implementación CUDA para GPUs. Este estudio requerirá un paso previo, en el que se analice la plataforma CUDA sobre la que está realizada la implementación.

Una vez adquirido el conocimiento necesario sobre la plataforma CUDA, se procederá al análisis del código de la implementación existente y al desarrollo de técnicas que posibiliten la mejora de su rendimiento.

A continuación se enumeran de forma concreta las fases de este proyecto:

1. Planificación del proyecto.

2. Estudio de la criptomoneda Monero.
3. Estudio de la prueba de trabajo RandomX.
4. Estudio de la plataforma CUDA.
5. Análisis del código de la implementación CUDA de RandomX.
6. Propuesta de técnicas de mejora del rendimiento de RandomX en GPUs.

3. CAPÍTULO

Monero

En este capítulo, se describirán en profundidad las características más relevantes de la criptomoneda Monero. En particular, se describirán las herramientas necesarias para la creación de transacciones no rastreables y cómo se almacenan dichas transacciones. Así mismo, se estudiará la escalabilidad de Monero con respecto a Bitcoin y se introducirá la prueba de trabajo RandomX.

3.1. Introducción

Según la documentación oficial, Monero es una criptomoneda segura, privada e imposible de rastrear que fue desarrollada en 2014 por un grupo de 7 desarrolladores, de los cuales 5 aún permanecen anónimos. Monero implementa, como el resto de criptomonedas, una cadena de bloques (*blockchain*) para registrar las transacciones realizadas. Una cadena de bloques, como se puede ver en la Figura 3.1, se puede describir como un conjunto de nodos que contienen información en el que cada uno de ellos se encuentra asociado con el anterior de manera unívoca.

En el caso de las criptomonedas, la información que se almacena en los bloques es la relativa a todas las transacciones realizadas. Así mismo, cada bloque está identificado por un *hash* (que se obtiene en el proceso de minado) y contiene en uno de los campos el *hash* del bloque anterior, estableciéndose de este modo una cadena.

En Monero, al contrario de lo que ocurre en otras criptomonedas, los bloques no tienen

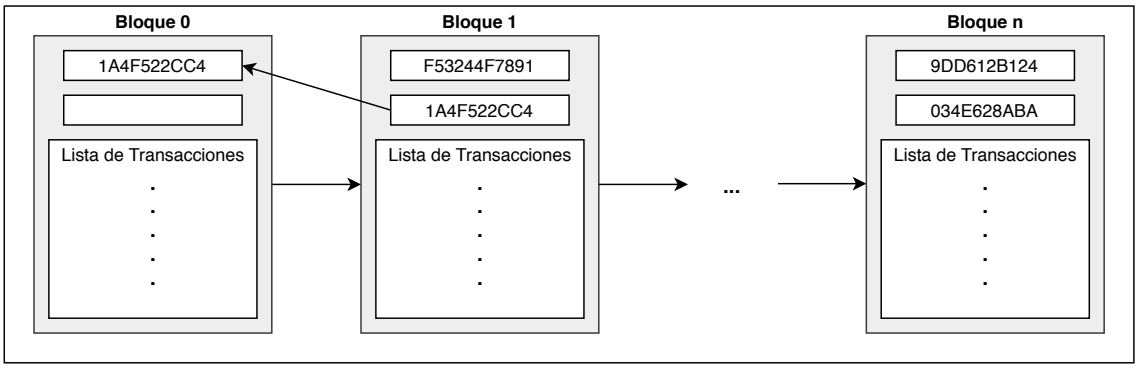

Figura 3.1: Ejemplo de una cadena de bloques (blockchain).

un tamaño predeterminado. Esto favorece la escalabilidad de la plataforma, debido a que el número de transacciones que se pueden validar por unidad de tiempo (en Monero se genera un nuevo bloque cada 2 minutos) no está limitado por el tamaño del bloque. Por ejemplo, en Bitcoin el tamaño máximo de bloque es 1MB lo que limita el número de transacciones que se pueden validar por unidad de tiempo, en este caso, 10 minutos.

El no tener un tamaño de bloque fijo favorece la escalabilidad (en términos de transacciones verificables por unidad de tiempo) pero plantea un problema: cuanto mayor es el número de transacciones del bloque a minar, mayor es la comisión que consigue el que lo haga. Por este motivo, los mineros podrían generar bloques enormes para aumentar los beneficios obtenidos. Para solucionarlo, Monero puso en marcha un sistema de penalización en el que la recompensa disminuye cuadráticamente con respecto a la diferencia de tamaño del bloque a minar y la media de tamaño de los últimos 100 bloques minados.

Como se ha comentado anteriormente, esta criptomoneda garantiza que las transacciones sean totalmente anónimas, o al menos, que no se puedan rastrear hasta un origen. Dicha característica ha suscitado un notable interés entre los usuarios que desean ocultar su identidad y permite dar el mismo valor a todos los tokens (fungibilidad). Un ejemplo podría ser el de tokens usados con fines ilegales, que se podrían devaluar debido a su origen ilegal. En el caso de Monero, esto no puede ocurrir debido a que no es posible conocer el origen de las transacciones.

Otra de las características principales que diferencia a Monero es la prueba de trabajo (PoW) o proceso de minado. En la actualidad, lo que sucede con otras criptomonedas es que el proceso de minado se concentra en puntos muy concretos del mundo donde se encuentran granjas de minado. Estas granjas se pueden describir como grandes agrupaciones de hardware específico (ASICs) para minar criptomonedas en donde se concentra la creación de la mayoría de nuevos tokens, haciendo imposible el acceso al proceso de

minado a usuarios “normales”. Con el objetivo de evitar esta situación, en el desarrollo de Monero se dio prioridad a garantizar la “igualdad” del proceso de minado, o dicho de otro modo, a evitar el minado de bloques utilizando hardware específico.

3.2. Privacidad

Monero implementa un nivel de anonimato que no se puede encontrar en otras criptomonedas. Para ello, se basa en CryptoNote [van Saberhagen, 2013], un protocolo que proporciona un alto grado de privacidad, permitiendo disociar los pagos de los emisores y los receptores de las transacciones. En particular, se tienen que cumplir las dos siguientes propiedades para conseguir un sistema con dichas características:

- No relacionabilidad (*unlinkability*): Para cualquier par de transacciones realizadas, es imposible saber si fueron emitidas hacia la misma persona garantizando así la **privacidad del receptor**.
- No trazabilidad (*untraceability*): Para cada transacción entrante, todos los posibles emisores son equiprobables garantizando así la **privacidad del emisor**.

En las siguientes subsecciones se describirá cómo Monero consigue implementar estas dos propiedades.

3.2.1. Pagos no relacionables (Privacidad del receptor)

En otras criptomonedas, el proceso para que un usuario reciba pagos se realiza de la siguiente manera: (1) un usuario publica su dirección, la cual es generada a partir de su clave pública y (2) los usuarios que quieren realizarle pagos los envían a esa dirección (firmando las transacciones utilizando la clave pública asociada). Como se puede observar, utilizando este método las transacciones pueden ser rastreadas hacia esa dirección.

Una solución, como se puede ver en la Figura 3.2, es que el usuario receptor genere múltiples direcciones (claves públicas) y nunca las asocie con su identidad (seudónimo). En la práctica, esto plantea ciertos inconvenientes debido a la necesidad de mantener múltiples direcciones-identidades. Para solucionar este problema, en Monero se utilizan claves públicas (direcciones) de **un solo uso** las cuales están asociadas con una única identidad. A estas direcciones de un solo uso se las conoce como *stealth addresses*.

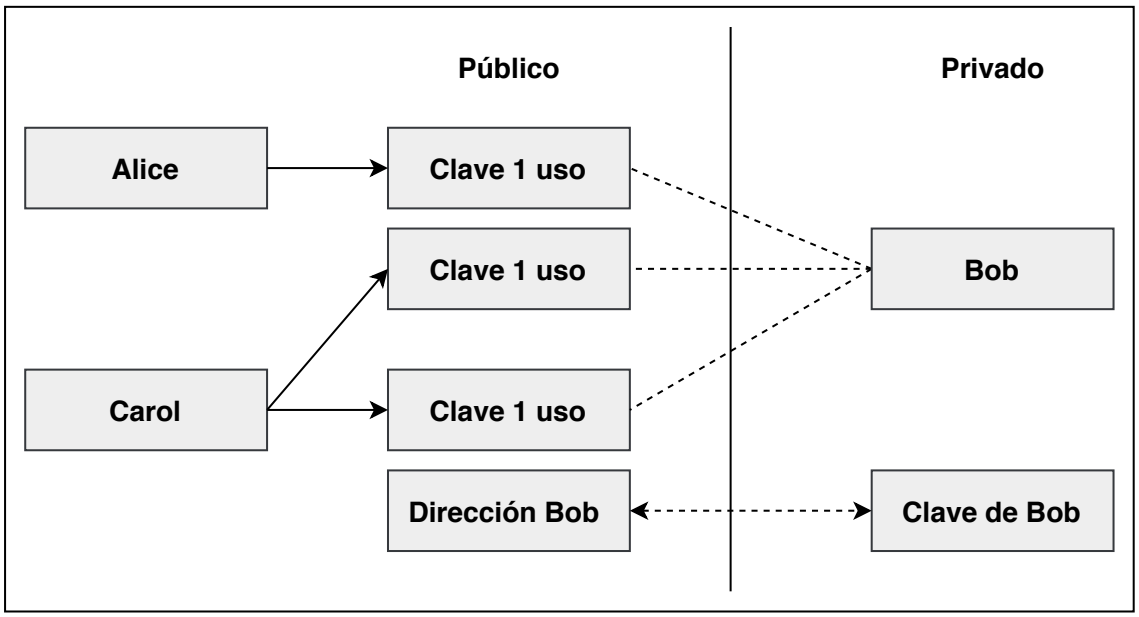

Figura 3.2: Direcciones (claves publicas) de un solo uso.

El proceso para que Alice emita un pago a Bob utilizando claves públicas de un solo uso, como se puede ver en la Figura 3.3, es el siguiente:

- Alice quiere hacer un pago a Bob, de quien ya conoce su dirección y en consecuencia la clave pública de Bob (A,B).
- Alice genera un valor random r y una clave pública de un solo uso con P = $H_s(rA)G+B$, donde H_s es una función de hashing y G es un parámetro de una curva elíptica.
- Alice usa P como clave de destino y empaqueta R = rG. Hay que señalar que Alice puede crear otras transacciones dirigidas a Bob con la misma clave (A,B).
- Alice envía la transacción a la red Monero.
- Bob recorre la blockchain en busca de transacciones de las que es destinatario utilizando su clave privada (a,b). Si las encuentra, es porque previamente habrá generado P' = $H_s(aR)G+B$, y se cumple que P'= P y que aR = arG =rA (ver Figura 3.4).
- Ahora Bob puede generar la clave privada de un solo uso mediante x = $H_s(aR)+b$. Cuando quiera usar lo tokens contenidos en dicha transacción, simplemente deberá firmar la nueva transacción con x.

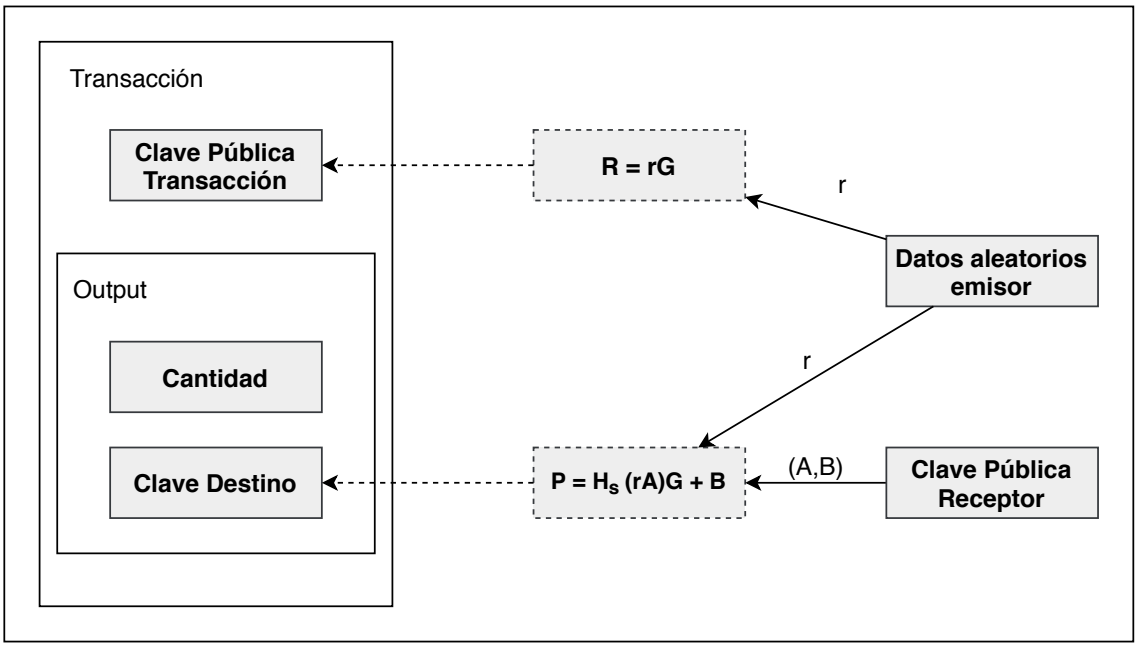

Figura 3.3: Transacción saliente desde el punto de vista del emisor.

De esta manera las transacciones realizadas hacia Bob son anónimas porque no se puede asociar las claves públicas de un solo uso con su dirección.

3.2.2. Ring Signatures (Privacidad del emisor)

Una *ring signature* o firma en anillo es un tipo de firma digital que permite conseguir *no relacionabilidad*. En otras criptomonedas, lo habitual es que firme el emisor real con su dirección (clave pública), lo que permite asociar las transacciones con sus emisores. Por el contrario, el uso de *ring signatures* no permite relacionar esa asociación, haciendo imposible relacionar una transacción con un emisor.

La forma en la que se garantiza esta propiedad es bastante simple, como se puede ver en la Figura 3.5. Básicamente, los usuarios generan firmas digitales que pueden ser verificadas por un conjunto de claves públicas (anillo) y no, como es habitual, frente a una única clave pública. De esta forma, la identidad del usuario que ha firmado no se puede distinguir de la identidad de otros usuarios que se encuentran en el anillo debido a que cualquiera de las claves públicas verifica la firma digital. Esto tiene como objetivo la anonimización de la transacción, es decir, de esta manera no se puede saber quién es el emisor de dicha transacción. Una descripción formal se puede encontrar en la *cryptonote* original [van Saberhagen, 2013].

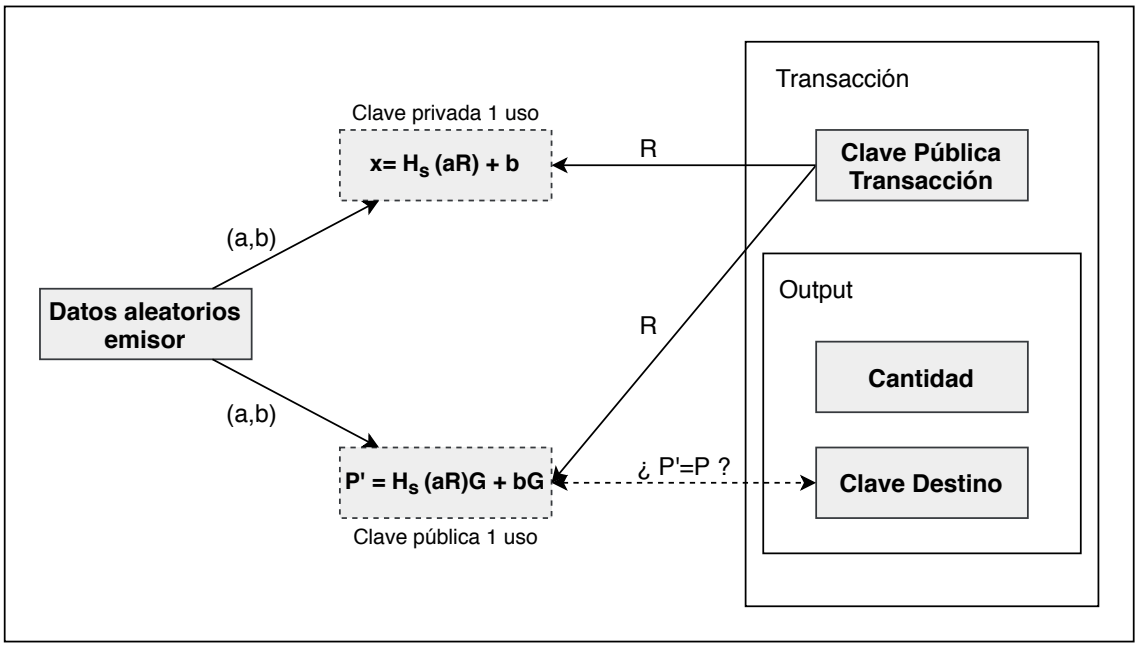

Figura 3.4: Transacción entrante desde el punto de vista del receptor.

3.2.3. Privacidad de las transacciones

Otro de los objetivos que persigue Monero es ocultar la cantidad de tokens (XMR) que se emiten o reciben en cada transacción. Para ello, Monero divide el valor total de la transacción en valores más pequeños y, en vez de generar una sola transacción, genera múltiples transacciones de valor inferior cuya suma es el valor total de la transacción real. Con esto se pretende ocultar la cantidad real de dicha transacción.

3.2.4. Proceso de una transacción

Ahora que disponemos de todos los elementos que proporcionan anonimidad a las transacciones en Monero, vamos a describir como se realiza una de ellas. En Monero, todos los usuarios disponen de una dirección pública (monedero o *wallet*). Aún así, los fondos de esa cartera no están asociados a la dirección pública, como es el caso de Bitcoin. En el caso de esta última, si alguien conociera la dirección pública podría saber, por ejemplo, el saldo de ese monedero.

Antes de describir el proceso vamos a definir las claves que utilizaremos:

- Public View Key: Necesaria, junto a la Public Spend Key, para generar la dirección de un solo uso a la que se transfieren los tokens.

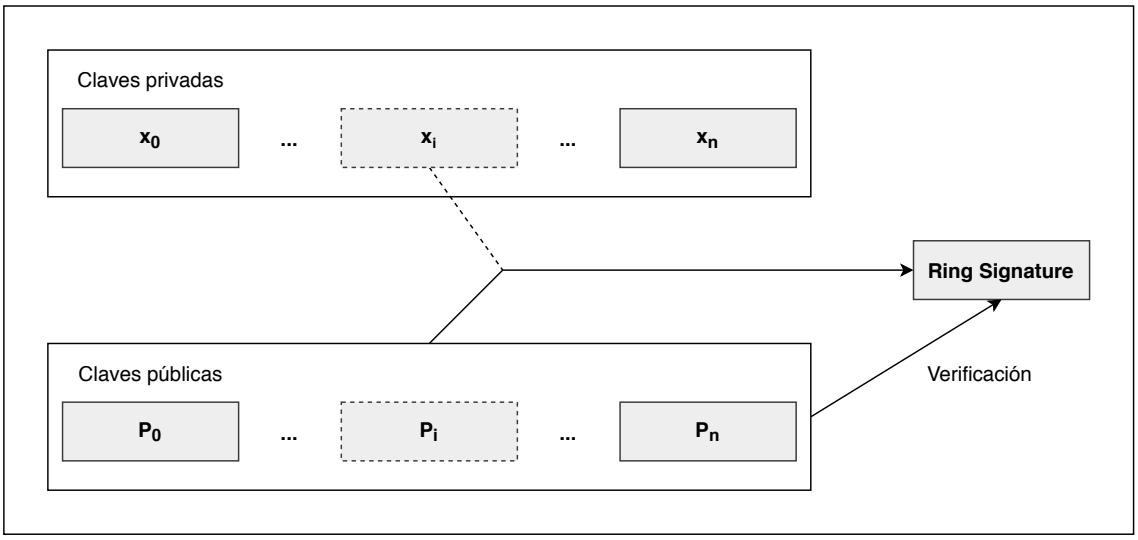

Figura 3.5: Proceso de creación de una firma en anillo (*ring signature*).

- Private View Key: Con esta clave el receptor puede encontrar su transacción en la blockchain.
- Public Spend Key: Es utilizada junto a la Public View Key para la generación de la dirección de una transacción.
- Private Spend Key: Necesaria para gastar tokens del monedero, se usa para firmar transacciones.

Cuando se envían fondos a la dirección pública de un usuario, en realidad el emisor está usando la "Public Spend Key" y "Public View Key" del destinatario para crear un *container* de un solo uso con los fondos a transferir. Una vez la transacción ha sido emitida (anunciada a toda la red de Monero), eventualmente será minada dentro de un bloque y agregada a la blockchain. Esto supone que en la blockchain no existe un registro que contenga la dirección pública real del destinatario. De la misma forma, en la blockchain tampoco aparecerá la dirección del emisor, debido a cómo se firman las transacciones. En su lugar, solamente aparecerá la *stealth address* (dirección publica de un solo uso) del destinatario.

Ahora la pregunta es clara: ¿cómo obtiene el destinatario los fondos de su monedero? Pues, en este momento entra en juego la "View Key Privada". Mediante el escaneo de toda la blockchain y usando esa clave, el destinatario es capaz de reconocer las transacciones que le corresponden. Como solo él conoce la clave privada, es el único capaz de ver las transacciones dirigidas a él. Por este motivo, cuando se lanza un monedero de Monero,

es necesario escanear toda la cadena de bloques. De la misma forma, una vez encontrada la transacción que le corresponde a un usuario, ese mismo usuario usa la "Private Spend Key" para poder gastar los fondos de ese *container*.

Para finalizar, si el destinatario de una transacción cualquiera T_1 usa los tokens en otra transacción T_2, el emisor de T_1 podría relacionar al emisor de la nueva transacción T_2 con el destinatario de T_1. Para evitar este conocimiento, en Monero se utilizan las "Ring Confidential Transactions" descritas anteriormente.

3.3. Prueba de trabajo

La Prueba de Trabajo es el procedimiento utilizado para verificar las transacciones que se realizan en la red. Por lo general, después de verificar que todas las transacciones a incluir en el nuevo bloque son válidas, implica resolver un problema matemático complejo y, a cambio, el primero que lo resuelve recibe una recompensa establecida por la red. Una vez el problema ha sido resuelto (el bloque ha sido minado), otros usuarios de la red deben verificar que el minado del bloque se ha realizado de manera correcta y, en caso de ser así, el nuevo bloque se añade a la blockchain.

El proceso de minado de un bloque se considerará válido siempre que se cumpla $\frac{B}{R} \geq D$ donde B es la máxima dificultad de la red (2^{256-1}), D es la dificultad actual de la red y R es el hash de salida de la prueba de trabajo.

Como se ha comentado anteriormente, Monero ha cambiado su PoW recientemente con el objetivo de hacerla resistente a los ASICs. Debido a la complejidad de RandomX y a que su estudio es uno de los principales objetivos de este trabajo, el algoritmo será descrito en profundidad en el capítulo 5.

4. CAPÍTULO

Cifrado basado en AES

Debido a que RandomX utiliza de manera intensiva el cifrado AES para generar la prueba de trabajo, en este capítulo se procederá a describir el algoritmo de cifrado Advanced Encryption Standard (AES) y las primitivas hardware y extensiones vectoriales que existen en los procesadores Intel que permiten su aceleración.

4.1. Introducción

AES [Intel, 2020b] es un estándar de cifrado simétrico adoptado por el gobierno de los Estados Unidos de América a principios de 2001. Este algoritmo se utiliza en múltiples aplicaciones y es habitualmente utilizado para, por ejemplo, cifrar las comunicaciones que circulan por una red utilizando el protocolo *https*. AES utiliza un cifrado de bloques simétrico que encripta/desencripta datos en varias etapas operando sobre una matriz de 4×4 bytes a la que se llama *state*, es decir, cifra un bloque fijo de 128 bits (16 bytes) de texto plano en varias rondas para producir, finalmente, el texto cifrado. El número de rondas depende de la longitud de la clave utilizada y varía entre 10, 12 o 14 rondas en función de si la clave es de 128, 192 o 256 bits. Lo salida obtenida en cada ronda es la entrada de la inmediatamente posterior.

En la **primera etapa** (ver Figura 4.1) se realiza una sustitución, en la cual, cada byte de la matriz *state* es sustituido por otro correspondiente a una tabla de sustitución de 8 bits. En criptografía, una tabla de sustitución es un componente esencial, se usa para ocultar la

relación entre la clave y el texto a cifrar. De esta manera se consigue que el cifrado no sea lineal.

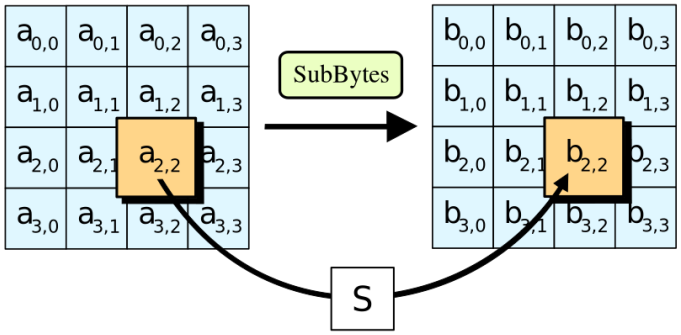

Figura 4.1: SubBytes

A continuación, en una **segunda etapa** los bytes de cada fila de la matriz se rotan cíclicamente hacia la izquierda, como se puede ver en la Figura 4.2. De esta manera, se pretende evitar que las columnas se encripten independientemente, en caso de que eso se sucediera, AES se degenera en 4 bloques cifrados independientes.

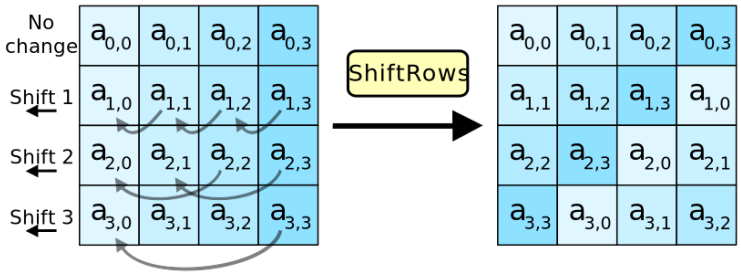

Figura 4.2: ShiftRows

Posteriormente, en la **tercera etapa** se realiza un mezclado de las columnas de la matriz *state*. Lo que se hace es combinar los 4 bytes en cada columna mediante una transformación lineal, como se muestra en la Figura 4.3. Lo que se consigue con este paso combinado con los dos anteriores es la difusión del cifrado. Esto significa que al cambiar un bit en el texto sin cifrar, deberían cambiarse la mayor cantidad posible de bits en el texto cifrado.

En la **etapa final** (ver Figura 4.4), cada byte de la matriz *state* se combina con una subclave. Por cada ronda, esta subclave se deriva de la clave de cifrado. El resultado se obtiene aplicando un XOR a cada byte del state con el correspondiente byte de la subclave.

Es posible mejorar el rendimiento del cifrado combinando los pasos de SubBytes y ShiftRows con el de MixColumns, transformándolos en una secuencia de búsqueda en las tablas. Ello requiere tablas de 32 bits con 4 entradas de 256 bits (en total 4096 bytes).

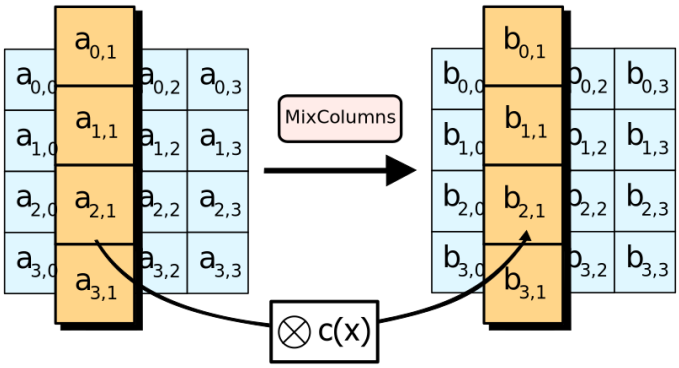

Figura 4.3: MixColumns

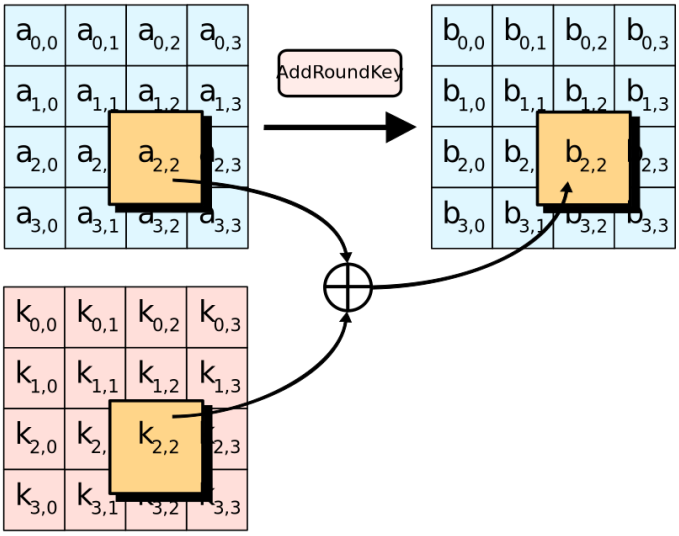

Figura 4.4: AddRoundKey

Una ronda se puede realizar con 16 operaciones de búsqueda en la tabla y operaciones de XOR de 32 bits, seguidamente se realizan 4 operaciones XOR de 32 bits en el paso AddRoundKey. Todo ello usando un direccionamiento al byte, se hace posible el combinarlos pasos de SubBytes, ShiftRows y MixColumns en una sola ronda.

4.2. Primitivas vectoriales

Las primitivas vectoriales Advanced Vector Extensions 2 (AVX2) son una evolución de las primitivas AVX que proporcionan soporte hardware para realizar operaciones vectoriales (SIMD) de 256 bits (en el caso de AVX las operaciones eran de 128 bits). AVX2 supone un gran salto en comparación a AVX ofreciendo nuevas instrucciones para realizar permutaciones y broadcast, instrucciones de cambio variable en vectores, e instrucciones

de búsqueda no contigua en memoria. Debido al éxito de AVX2, Intel ha desarrollado un nuevo paquete de instrucciones, AVX-512, que permiten realizar operaciones sobre 512 bits.

AVX-512 permite realizar 32 operaciones de coma flotante de precisión doble y 64 de precisión simple por ciclo de reloj en vectores de 512 bits. Así mismo, permite realizar operaciones con ocho enteros de 64 bits y dieciséis de 32 bits utilizando dos unidades combinadas de multiplicación y suma de 512 bits (FMA), duplicando el número de registros y el ancho (512 bits) de los registros y de las unidades FMA.

4.3. Aceleración hardware del algoritmo AES (AES-NI)

Intel Integrated Performance Primitives (Intel IPP) es la denominación oficial de un conjunto de primitivas hardware desarrolladas por Intel. IPP está compuesto por una extensa librería de funciones específicas optimizadas para diferentes arquitecturas Intel y proporciona a los desarrolladores un API para utilizar de manera sencilla instrucciones de tipo Single Instruction Multi Data (SIMD) que pueden ayudar a mejorar el rendimiento de ciertas aplicaciones, por ejemplo, aquellas que utilizan criptografía o compresión de datos.

En 2010, la nueva generación de procesadores Intel Core incluyó por primera vez un conjunto de instrucciones denominadas Intel Advanced Encryption Standard (AES) New Instructions (AES-NI) [Intel, 2020a]. Estas instrucciones fueron diseñadas para implementar las etapas más complejas y computacionalmente más costosas del algoritmo AES usando una implementación vectorial en hardware, permitiendo de esta manera acelerar la ejecución de este algoritmo. AES-NI está compuesto por 6 instrucciones que se corresponden con las diferentes etapas del algoritmo AES. En particular, 4 instrucciones se usan para mejorar el rendimiento del encriptado/desencriptado en cada ronda, mientras que las 2 restantes se utilizan en la etapa de generación de claves.

- AESENC: Ejecuta una ronda de encriptación
- AESENCLAST: Instrucción para la última ronda de encriptación
- AESDEC: Instrucción para una ronda de desencriptado
- AESDECLAST: Instrucción para la última ronda de desencriptado

- AESKEYGENASSIST: Instrucción usada para generar claves para las rondas de encriptación.
- AESIMC: Instrucción que convierte las claves de encriptación a un formato válido para el desencriptado.

AES-NI utiliza por defecto 128 bits, pero existen versiones más recientes capaces de cifrar/descifrar bloques de 256 y 512 bits. En las Figuras 4.1 y 4.2 se describen los algoritmos utilizados por las funciones *AESENC* y *AESDEC* para cifrar y descifrar bloques de 512 bits.

Código fuente 4.1: Algoritmo de la instrucción AESENC de 512 bits

```
FOR j := 0 to 3
    i := j*128
    a[i+127:i] := ShiftRows(a[i+127:i])
    a[i+127:i] := SubBytes(a[i+127:i])
    a[i+127:i] := MixColumns(a[i+127:i])
    dst[i+127:i] := a[i+127:i] XOR RoundKey[i+127:i]
ENDFOR
dst[MAX:512] := 0
```

Código fuente 4.2: Algoritmo de la instrucción AESDEC de 512 bits

```
FOR j := 0 to 3
    i := j*128
    a[i+127:i] := InvShiftRows(a[i+127:i])
    a[i+127:i] := InvSubBytes(a[i+127:i])
    a[i+127:i] := InvMixColumns(a[i+127:i])
    dst[i+127:i] := a[i+127:i] XOR RoundKey[i+127:i]
ENDFOR
dst[MAX:512] := 0
```

AES-NI puede conseguir un rendimiento entre 3 y 10 veces superior al obtenido utilizando implementaciones software en CPUs. Además, añade un nivel extra de seguridad ya que se minimiza el riesgo de sufrir ataques de tipo *side channel* recientemente descubiertos, debido a que las instrucciones de encriptado y desencriptado se realizan íntegramente en hardware.

5. CAPÍTULO

RandomX

En este capítulo se describirá en detalle la prueba de trabajo RandomX [Tevador, 2020] utilizada por la criptomoneda Monero. En particular se analizará en profundidad las técnicas utilizadas para hacerlo resistente a la ejecución en ASICs.

5.1. Introducción

Como se ha comentado anteriormente, uno de los objetivos del desarrollo de Monero fue hacer del sistema de minado un sistema justo. Más concretamente, lo que se pretendía es que la criptomoneda estuviera distribuida lo más equitativamente posible entre todos los usuarios de la plataforma. La forma de conseguirlo es evitar que dispositivos específicos hardware (ASICs) fueran capaces de resolver la prueba de trabajo. Para ello se ha desarrollado RandomX.

RandomX es un algoritmo cuya característica principal es la de generar y ejecutar programas que explotan características específicas de un dispositivo hardware, en este caso, CPUs de propósito general evitando así su implementación eficiente en dispositivo especializados. En particular, RandomX es capaz de generar programas específicos para procesadores Intel, AMD, ARM y Power de IBM.

El motivo por el que se decidió utilizar CPUs y no GPUs es que las CPUs están más extendidas y, en genera, mantienen un subconjunto de instrucciones comunes entre diferentes arquitecturas haciendo menos compleja la tarea de crear RandomX. Además los

juegos de instrucciones de las CPUs están bien documentados en comparación con los de las GPUs. A pesar de ello, existe una implementación que se ejecuta sobre GPUs de NVIDIA [SChernykh, 2020] y que sera analizada en detalle en el capítulo 7.

5.2. Diseño de RandomX

La principal idea detrás del diseño de RandomX es que la prueba a resolver debe ser dinámica aprovechando el hecho de que las CPUs aceptan dos tipos de entrada: (1) datos y (2) código que determina que hacer con los datos. Otras PoW de otras criptomonedas únicamente se basan en los datos permitiendo la resolución del problema en cualquier dispositivo debido a que la secuencia de operaciones a realizar sobre ellos es fija.

El algoritmo RandomX consiste en los siguientes 4 pasos:

1. Generación un programa aleatorio: El modo más rápido de generar un programa aleatorio es utilizar un generador sin lógica (logic-less) el cual rellena un *buffer* con datos aleatorios. A su vez, es necesario diseñar un lenguaje de programación sin sintaxis en el cual cualquier secuencia de bits aleatorios representa un programa valido.

2. Traducción de ese programa a lenguaje máquina de la CPU destino: Una vez creado el programa, este se tiene que traducir al lenguaje nativo e la arquitectura. Este paso se debe hacer de manera muy rápida por lo que las instrucciones de RandomX deben ser similares a las de la arquitectura destino.

3. Ejecución del programa en la CPU: La ejecución del programa debe utilizar el mayor número de características especificas de las CPUs (anti-ASIC). Entre ellas se pueden destacar la caches multi-nivel el controlador de memoria, la ejecución especulativa y fuera de orden y superescalar, etc. (ver Seccion **??**)

4. Transformación del resultado del programa en un valor criptográfico seguro: La salida del programa sera una hash que tiene que verificar cierta condición de la red Monero. En un entorno CPU es imprescindible usar funciones criptográficas rápidas como Blake2b (más rápida que SHA-3) y AES que debido a las implementaciones hardware ofrece una gran velocidad.

5.3. Arquitectura

Antes de describir en detalle cada paso del algoritmo, en esta sección se explicaran el modelo de maquina virtual (VM) que utiliza RandomX así como el formato de las instrucciones y las estructuras que utiliza para representar la jerarquía de memoria de una CPU.

5.3.1. Modelo de VM

En la Figura 5.1 se muestra el modelo de VM utilizado por RandomX. Los programas serán generados siguiendo este modelo y para ello se ha diseñado un juego de instrucciones específico.

Como se puede ver en la imagen la arquitectura de la VM es muy similar al de una CPU de propósito general. A continuación, se describe cada elemento funcional que la compone:

- Unidad de control: Unidad encargada de buscar las instrucciones, decodificarlas (interpretación) y ejecutarlas.
- Fichero de registros: Representa los registros de la VM. En este caso se disponen de 8 registros enteros de 64 bits y de 12 registros de coma flotante de 128 bits.
- Unidad aritmética lógica: Unidad encargado de realizar la aritmética lógica y de enteros.
- Unidad de coma flotante: Unidad encargada de realizar la aritmética de coma flotante.
- Buffer de programa: Memoria donde se almacena el programa.
- Scratchpads (3 niveles): Representa los 3 niveles de memoria cache presentes en los procesadores modernos.
- Dataset (solo lectura): Representa la memoria principal pero en este caso solo se pueden hacer accesos de lectura.
- Unidad de prefetching: Se encarga de adelantar lecturas realizadas sobre el Dataset.

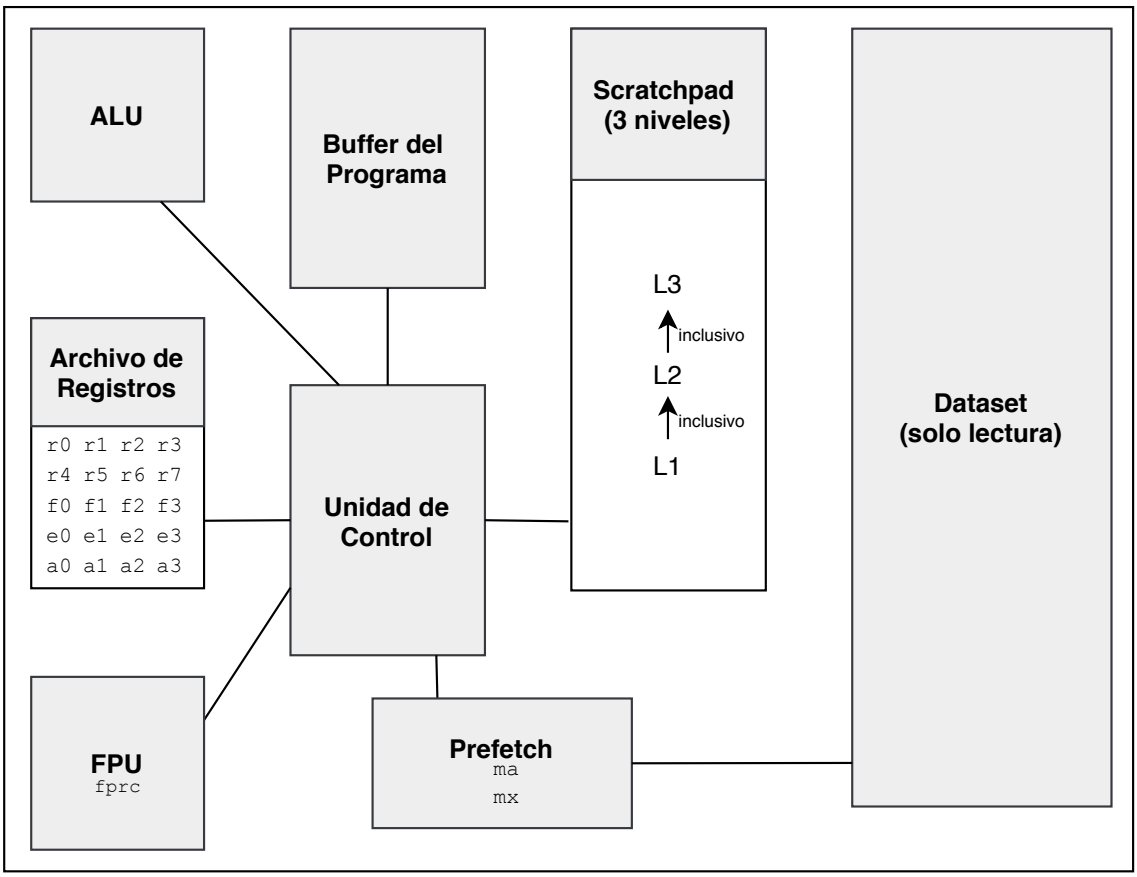

Figura 5.1: Componentes de la máquina virtual de RandomX.

5.3.2. Instrucciones

La maquina virtual ejecuta un set de instrucciones especial diseñado de tal manera que cualquier palabra de 8 bytes sea una instrucción válida y cualquier secuencia de instrucciones sea un programa válido. Al no haber reglas de sintaxis, generar un programa es tan sencillo como llenar el buffer del programa con datos aleatorios. Cada instrucción tiene una longitud de 64 bits como se muestra en la Figura 5.2 y cada instrucción de RandomX se traduce entre 1 y 7 instrucciones x86 siendo la media 1,8 instrucciones.

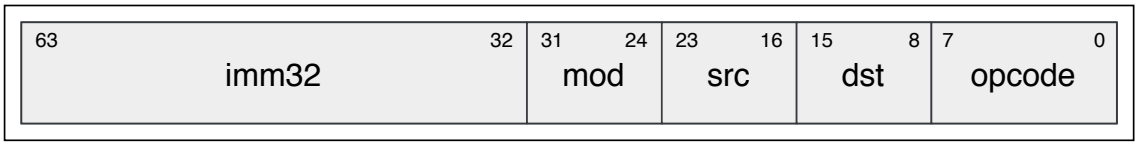

Figura 5.2: Representacion de una instrucción RandomX.

En total existen 256 opcodes distribuidos entre 29 instrucciones distintas. Cada instrucción puede ser codificada usando múltiples opcodes. En cuanto a los registros de destino,

solo son usados los bits 0-2 para codificar un registro. Por otra parte, el flag src codifica el registro del operando de origen, en los que en ocasiones se usa un valor constante cuando el destino y el origen codifican el mismo registro. Finalmente, el flag *mod.mem* selecciona entre el nivel L1 y L2 del Scratchpad cuando se lee o se escribe en memoria. Únicamente en 2 casos se utiliza el Scratchpad L3:

- Cuando se realiza una lectura y el destino y el origen es el mismo registro.
- Cuando una escritura *mod.cond* es 14 o 15.

5.3.3. Registros

Se usan 8 registros de enteros y 12 de coma flotante. Estos son, exactamente, el máximo número de registros que se pueden asignar físicamente en las arquitecturas x86-64. Si se usaran más registros pondría en seria desventaja a la arquitectura x86 que debería usar memoria para contrarrestarlo.

5.3.4. Operaciones de enteros

RandomX usa todas las primitivas de operaciones de enteros de las que disponen las CPUs. Sin embargo, para obtener rendimiento alto en CPUs y el destino debe ser siempre un registro de enteros (grupo R). Entre las instrucciones se encuentran: IADD_RS, ISUB_R o IMUL_R.

5.3.5. Operaciones de coma flotante

En RandomX se utilizan operaciones de coma flotante de doble precisión, soportadas por la mayoría de CPUs y que necesitan un hardware complejo para llevarlas a cabo. Por el mismo motivo se incluyen operaciones que realizan instrucciones vectoriales de 128 bits.

5.3.6. Instrucciones de control

Hay 2 instrucciones de control: CFROUND y CBRANCH. La primera de ellas, calcula un valor de 2 bits mediante la rotación a derecha del registro origen y cogiendo los 2 bits menos significativos. El resultado se guarda en el registro *fprc*, lo que provoca un

cambio en el modo de redondeo en las operaciones de coma flotante. Por otro lado, la segunda instrucción añade un valor inmediato al registro destino y después, prepara un salto condicional en el buffer del programa basado en el valor del registro destino.

Las CPUs modernas gastan una enorme cantidad de energía y esfuerzo gestionando las branches. Debido a este motivo, si la CPU especula y falla, se obtendría como resultado un gasto de tiempo y energía innecesario. Es por ello, que se minimiza lo máximo posible las posibilidades de que esto ocurra. Lamentablemente, esto no es sencillo. Incluir reglas para las branches podría provocar que se convirtieran en estáticas, lo que supondría un trabajo de optimización muy sencillo para hardware especializado y, por supuesto, una enorme ventaja para ellos.

5.3.7. Store

Solo hay una instrucción explícita para valores enteros: ISTORE. Esta instrucción guarda el valor del registro origen en la posición de memoria calculada a partir del valor del registro de destino. El registro de origen y el destino pueden ser el mismo.

5.3.8. SuperscalarHash

SuperscalarHash es una función de difusión personalizada que fue diseñada para consumir la mayor cantidad de energía usando únicamente la ALU de las CPUs. El input y el output de esta función son 8 registros de enteros de 64 bits cada uno. En este caso, el output sirve para la construcción del Dataset.

5.3.9. Paralelismo

Se usan varias técnicas para favorecer el rendimiento de la CPU. RandomX beneficia el hecho de tener múltiples unidades de ejecución (ejecución superescalar) y el poder ejecutar las instrucciones de manera desordenada (*out-of-order execution*).

5.3.10. Scratchpad

El denominado *Scratchpad* es simplemente un conjunto de datos que se divide entre las cachés del procesador. La mayoría de instrucciones acceden a la caché L1 o L2 debido a

su baja latencia comparada con L3.

El Scratchpad se modifica continuamente en tiempo de ejecución. A la finalización de cada iteración, todos los valores de registro, equivalentes a dos bloques de 64 bytes cada uno, se almacenan en L3. Además, después de la instrucción ISTORE, la cual realiza de media 16 escrituras por programa, se realizan 2 accesos aleatorios a memoria L3. Esto solo sucede una vez en cada iteración.

5.3.11. Dataset

Al contrario de lo que sucede con el Scratchpad, el Dataset es un conjunto de datos de solo lectura que RandomX guarda en memoria. Los accesos de lectura al Dataset son lo únicos que utilizan los controladores de memoria. El Dataset ocupa alrededor de 2080 MB, del orden de 8 veces más grande que la capacidad de las cachés. RandomX lee del Dataset una vez por iteración (16384 veces por hash).

El Dataset es construido a partir de una clave K que se pasa como parámetro y debe ser recalculado cada vez que cambia dicha clave. Con la idea de que la verificación de la Proof of Work pueda realizarse utilizando menos memoria, el Dataset se construye en 2 pasos valiéndose de una estructura intermedia llamada caché. La caché se construye a partir de K y el proceso se inicia llenando la caché usando la función blake2b, a la que se le pasa K como parámetro. Una vez construida la caché, cada 64 bytes del Dataset se generan independientemente. Se aplica la función SuperscalarHash junto a un XOR a datos seleccionados aleatoriamente de la caché. En la Figura 5.3 se puede ver como se construye de una forma gráfica.

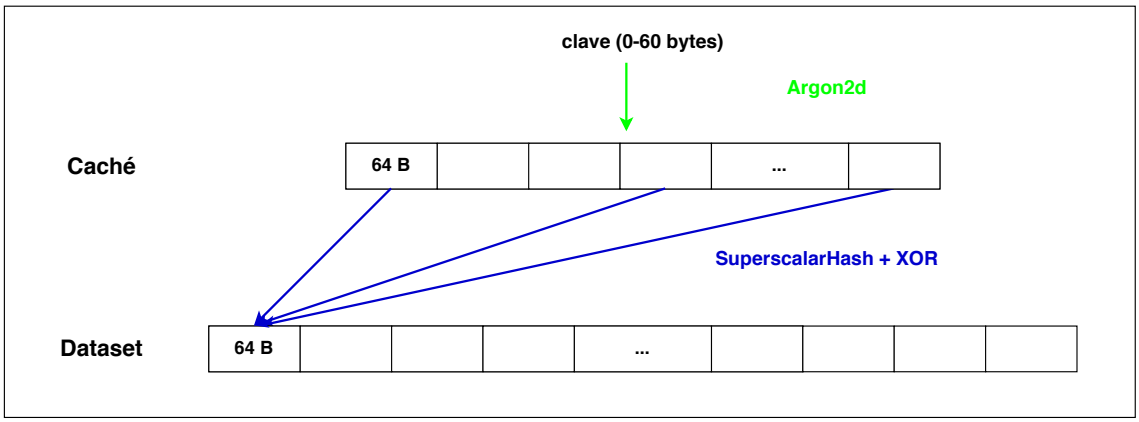

Figura 5.3: Construcción del Dataset

Por ello, en el proceso de verificación (*lightmode*) que solo requiere comprobar una hash

el Dataset no es generado y en su lugar se ejecuta la función *SuperScalarHash* de manera dinámica y se genera el valor que correspondería en el Dataset. Esto permite el uso de hardware con menos recursos para verificar bloques minados.

5.4. Ejemplo de Programa Random X

A continuación se muestra un ejemplo de programa RandomX generado aleatoriamente (16 instrucciones). En el se pueden ver múltiples instrucciones de aritmética de enteros, lógicas y de coma flotante. Así mismo se puede ver una instrucción de salto condicional y dos instrucción de escritura en el Scratchpad (L2 y L3).

Código fuente 5.1: Ejemplo de programa RandomX (16 instrucciones)

```
1  CBRANCH  r3,   7
2  FSUB_R   f3, a3
3  IMUL_R   r2, r4
4  IMULH_R  r3, r1
5  IXOR_R   r4, r7
6  IMUL_R   r1, r6
7  IMUL_RCP r1
8  ISUB_R   r2, r4
9  ISTORE   L2[r7], r4
10 ISTORE   L3[r6], r7
11 INEG_R   r0
12 FSUB_R   f0, a3
13 IMUL_R   r6, r6
14 FSUB_R   f2, a3
15 IROR_R   r7, r3
16 IXOR_R   r0, r7
```

5.5. Ejecución y generación de programas RandomX

La ejecución de un programa RandomX implica la ejecución de 8 programas encadenados en el que la entrada que recibe un programa se corresponde con la salida del anterior (Figura 5.4).

El motivo de ejecutarlos de esta manera es impedir que se analicen los programas y se pueda decidir no ejecutarlo si no es lo suficientemente favorable para el hardware del que se dispone.. Por ejemplo, podríamos disponer de un ASIC en el que la ejecución de divisiones no es eficiente. En caso de detectar un programa en el que hay varias podríamos

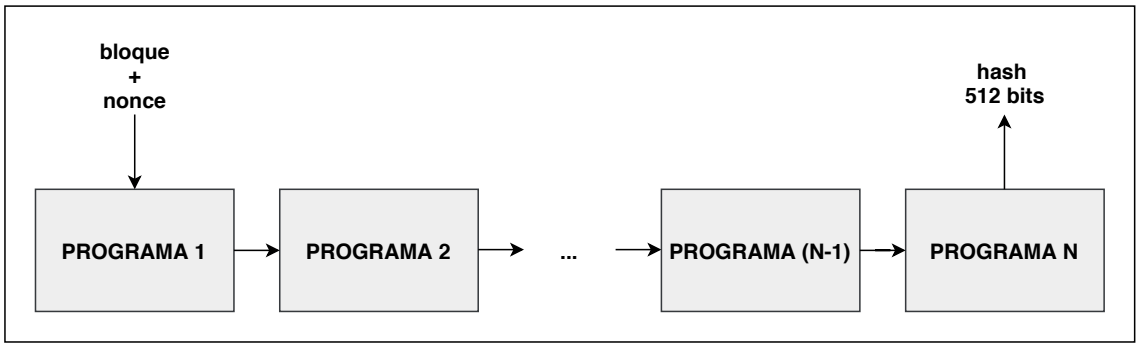

Figura 5.4: Secuencia de programas

decidir no ejecutarlo y proceder con el siguiente programa. Con el uso de la cadena, no compensa de tener la ejecución una vez analizado el programa debido a la energía y tiempo gastados para calcular los anteriores.

Dichos programas constan de 256 instrucciones que se ejecutan 2048 veces. Son exactamente 256 porque de esta manera hay suficientes instrucciones para generar múltiples programas y se da espacio para que haya branches. Por otra parte, el número de instrucciones es suficientemente bajo de tal manera que las CPUs de alto rendimiento pueden ejecutar una iteración en el tiempo que acceden a los datos en memoria. Las instrucciones se generan aleatoriamente y se transformarán más tarde en código ensamblador de la arquitectura correspondiente a la máquina utilizada.

5.6. Algoritmo RandomX en profundidad

En esta sección, describiremos paso a paso el algoritmo ejecutado por RandomX. Una representación gráfica de este se puede observar en la Figura 5.5.

El algoritmo RandomX recibe dos string K y H como entrada y produce como salida un hash R de 256 bits. En el proceso de minado, K se corresponde con un campo del bloque de Monero a minar y H se corresponde con el *nonce* que queremos evaluar. El valor R de salida es un hash que se comparará con la dificultad establecida por la red Monero. En caso de que cumpla esta restricción, el bloque se considerará minado y se enviará a la red para su verificación (ver sección 3.3).

1. El Dataset se inicializa usando el valor de la clave K.

2. se calcula una semilla (S) de 64 bytes usando la función Hash512 y la entrada H.

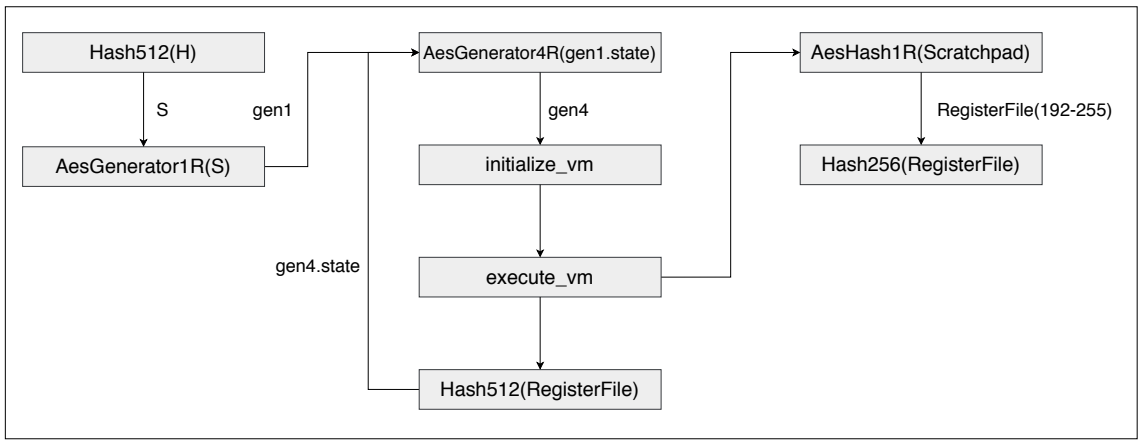

Figura 5.5: Secuencia de pasos del algoritmo RandomX.

3. La semilla S generada en el paso anterior se usa para crear un generador (gen1) mediante la función AesGenerator1R(S).

4. Se llena el Scratchpad de forma aleatoria usando el generador gen1.

5. Una vez lleno el Scratchpad, es momento de calcular gen4, para ello, AesGenerator4R utiliza gen1, tal y como se puede ver en la figura.

6. Tras realizar los pasos anteriores se programa la MV con bytes aleatorios (programas) usando gen4.

7. Se ejecuta la MV.

8. Se calcula una semilla de 64 bytes aplicando la funcion Hash512 al archivo de registros (ver parte izquierda de la Figura 5.6). Después se le asigna al estado de gen4 la semilla recién creada.

9. Los pasos 5–8 se repiten 7 veces.

10. En la octava iteración, el paso 8 no se realiza. En su lugar, se aplica función AesHash1Rsobre el Scratchpad y el resultado se guarda en los bits 192 al 255 del archivo de registro (ver parte derecha de la Figura 5.6).

11. Finalmente, el resultado es calculado aplicando Hash256 al archivo de registro.

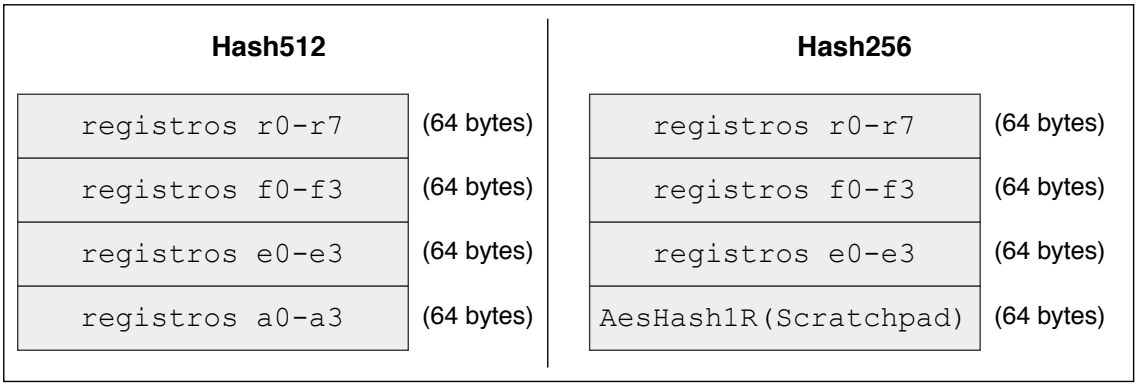

Figura 5.6: Representación del estado del archivo de registro durante ejecución de RandomX.

6. CAPÍTULO

CUDA

En este capítulo se describirá de forma general qué es CUDA y para qué se utiliza, y se profundizará en aquellos aspectos relevantes para este proyecto que serán utilizados en el capítulo 7, donde se analiza la implementación de RandomX. Así mismo, se describirá la arquitectura de la GPU utilizada.

6.1. Introducción

Compute Unified Device Architecture[1] (CUDA) [CUDA, 2011] se refiere a una plataforma de computación paralela para GPUs de NVIDIA que está compuesta por las herramientas de desarrollo necesarias para crear software para este tipo de arquitecturas hardware.

El objetivo del desarrollo usando CUDA es aprovechar los beneficios que se pueden obtener al utilizar el alto nivel de paralelismo que ofrecen las actuales tarjetas gráficas. Este paralelismo se obtiene mediante el uso de miles de hilos ejecutados en paralelo sobre los miles de cores disponibles. Hay problemas que se adaptan mejor que otros a la ejecución en este tipo de arquitecturas, en concreto, aquellos formados por tareas masivamente paralelas sin dependencias entre ellas (embarrassingly parallel).

Antes de describir el modelo de computación CUDA, es importante conocer en profun-

[1] Arquitectura Unificada de Dispositivos de Cómputo

didad el hardware de las GPUs. Para ello, pasaremos a describir la GPU que se utilizara más adelante, en los capítulos 7 y 8.

6.2. Hardware

Para el desarrollo de este proyecto se ha utilizado una NVIDIA Quadro P4000 con 1792 cores CUDA y 8 GB de memoria GDDR5 que posee un rendimiento pico teórico de 5.3 TFLOPS consumiendo tan solo 105W como máximo. La arquitectura de esta GPU se denomina Pascal, en concreto GP104, y es la primera que integra la interconexión bidireccional de alta velocidad de NVIDIA NVLink. Esta tecnología proporciona una interconexión directa de GPU a GPU proporcionando un mayor ancho de banda y una mejor escalabilidad para sistemas multi-GPU.

En la Figura 6.1 se puede ver una representación gráfica de la organización interna de los componentes presentes en la arquitectura Pascal. En ella se pueden observar 4 Graphics Processing Clusters (GPC) cada uno de ellos compuesto por 5 streaming multiprocessor (SM) para formar un total de 20 SMs. Cada uno de estos multiprocesadores está compuesto por cuatro bloques de 32 CUDA cores (en verde) que son los encargados de ejecutar un hilo cada uno.

Respecto a la memoria, la arquitectura Pascal GP104 dispone de 48KB de cache L1 y 96 KB de memoria compartida por SM y una memoria cache L2 de 2048 KB compartida por todos los SMs. Por otra parte, dispone de 8GB de memoria global y un bus de 256 bit lo que resulta en un ancho de banda de 243GB/s. La comunicación con la CPU para realizar transferencias hacia y desde la memoria de la GPU se realiza utilizando el bus PCI Express (versión 3.0).

6.3. Abstracciones de programación en CUDA

La programación de aplicaciones en CUDA utiliza una abstracción que imita la organización interna del hardware. En este tipo de aplicaciones, se definen kernels que se corresponden con el software que se ejecutará en la GPU. Hay que destacar que las GPUs se pueden ver como coprocesadores, por lo que siempre habrá un proceso en la CPU que se encargara de las transferencias de datos, del lanzamiento de los kernel y de la gestión de la ejecución.

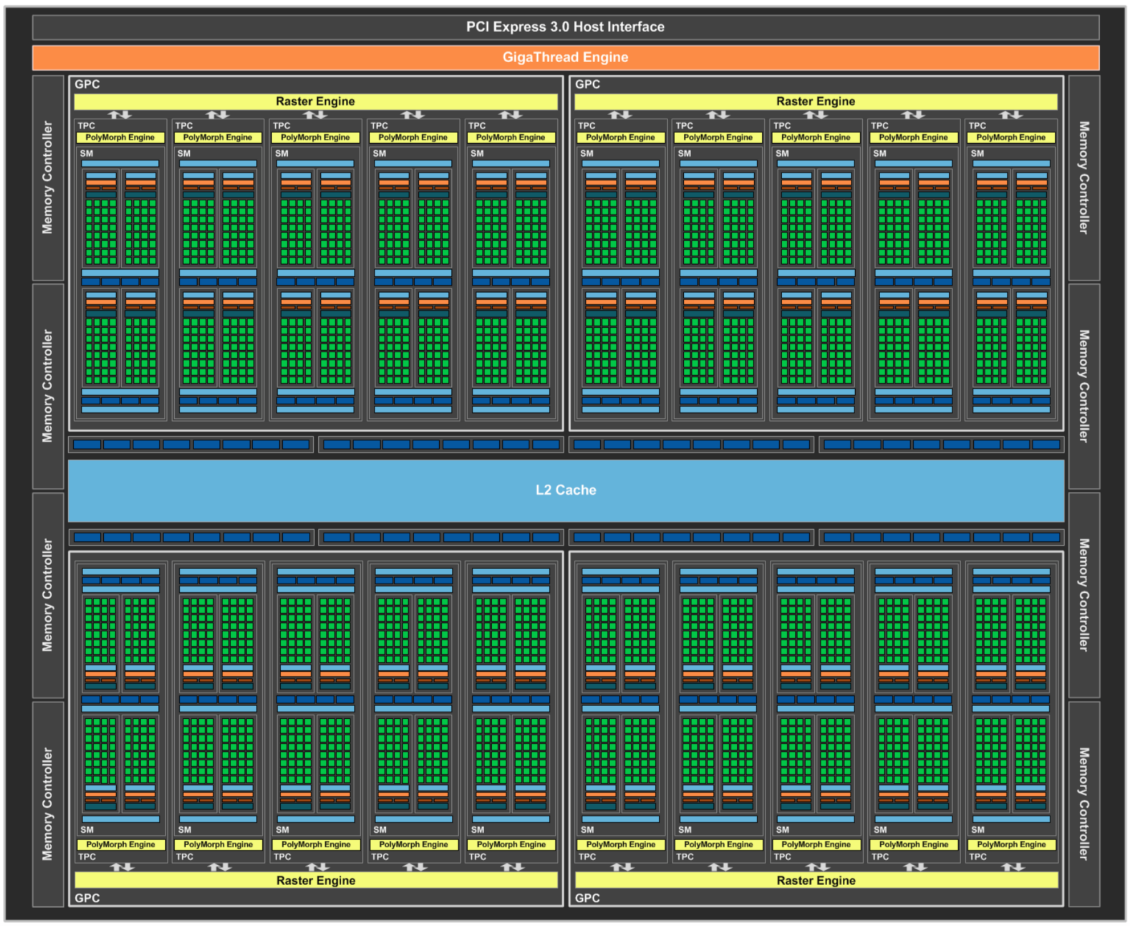

Figura 6.1: Organización interna de la arquitectura Pascal. [Fuente: NVIDIA]

A continuación se describen los elementos principales de un programa (kernel) desarrollado en CUDA:

- Thread [Wikipedia, 2020]: Es la unidad de ejecución y está compuesta (o debería) por un número pequeño de instrucciones. En la práctica se ejecutará un hilo en cada CUDA core.

- Bloque: Este término en CUDA se refiere a un grupo de threads que pueden ser ejecutados en serie o paralelo para conseguir un procesamiento y mapeado de datos eficiente. El número de threads que compone un bloque varía según la arquitectura y en la actualidad tiene un valor de 1024.

- Grid: Conjunto de bloques que contienen el mismo número de threads. A pesar de que el límite de threads en un bloque está limitado a 1024, se puede formar un grid para aumentar el número de bloques que operan en paralelo.

Durante la ejecución, el número de bloques está almacenado en la variable blockDim y cada bloque se identifica mediante la variable *blockIdx*. Dentro de cada bloque, los hilos están identificado con un identificador único, al que se accede con la variable *threadIdx*. Estas variables, se utilizan para identificar cada hilo y acceder a los datos de manera paralela usando como índices de, por ejemplo, un vector.

Cuando se lanza un kernel a ejecución, hay que indicar el número de bloques y el número de threads por bloque que lo formarán. Una vez en ejecución, se denomina warp a un conjunto de 32 threads dentro de un bloque en el que todos ejecutan la misma instrucción. Los bloques son seleccionados en serie por un SM y una vez que un warp es lanzado en el SM, todos los demás warps esperan hasta que la ejecución finaliza. Un nuevo bloque no será lanzado en el SM hasta que no existan un número suficiente de registros libres para todos los warps del nuevo bloque y hasta que no se tenga memoria compartida suficiente para el nuevo bloque.

La comunicación entre threads en ejecución solo se puede realizar si pertenecen al mismo bloque, mediante memoria compartida. En caso de necesitar comunicar threads de distintos bloques, es necesario utilizar la memoria global, con la penalización, en términos de rendimiento, que ello conlleva (unas 100 veces más lenta).

6.4. Streams

Un *stream* es una secuencia de operaciones que se ejecutan en la GPU en el orden establecido en el código. Las operaciones dentro del mismo stream se ejecutarán en secuencialmente, mientras que las operaciones ejecutadas en diferentes streams pueden ser intercaladas e incluso, si es posible, ejecutadas concurrentemente.

Todas las ejecuciones de kernels y las transferencias de datos en CUDA se ejecutan dentro de un stream. Por defecto, y si no se especifica nada, será en el stream 0 o *null stream*. El stream por defecto tiene una particularidad con respecto a los demás: es un stream de sincronización de operaciones, o dicho de otro modo, no se lanzará ninguna operación nueva hasta que todas las operaciones dentro de dicho stream hayan finalizado.

El código mostrado a continuación es un ejemplo de ejecución en el stream 0:

Código fuente 6.1: Ejecución en el stream por defecto.

```
1 cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
2 increment<<<1,N>>>(d_a)
```

```
3 cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Todas esas operaciones se realizan en el mismo stream (en el 0 por defecto) y serán ejecutadas de manera secuencial. En caso de querer ejecutar dichas operaciones en un stream diferente, los pasos son los siguientes: (1) declarar y crear el nuevo stream, (2) sustituir la función *memcpy* (síncrona) por la función *cudaMemcpyAsync* (asíncrona) especificando el identificador del stream creado y (3) indicar en el kernel el stream que se utilizará. En la siguiente imagen se puede ver el código modificado.

Código fuente 6.2: Ejemplo de uso de un stream.

```
1  // Se define el stream
2  cudaStream_t stream1;
3  cudaError_t result;
4  // Se crea el stream
5  result = cudaStreamCreate(&stream1)
6  // Función de transferencia de datos
7  result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1)
8  //Ejecución del kernel en el stream1
9  increment<<<1,N,0,stream1>>>(d_a)
10 // Se destruye el stream
11 result = cudaStreamDestroy(stream1)
```

Es importante destacar que el objetivo principal de los streams es solapar las ejecuciones de los kernels con las transferencias de datos. Es decir, como se muestra en la Figura 6.2 (versión asíncrona 2), el objetivo es que los datos del kernel que se ejecutará inmediatamente después se transfieran mientras se está ejecutando otro kernel. De esta manera, cuando acabe la ejecución del kernel, el siguiente ya puede ser lanzado sin esperar a que los datos que necesite se transfieran. A continuación se muestra el código que utiliza esta técnica:

Código fuente 6.3: Versión asíncrona usando multiples streams.

```
1 for (int i = 0; i < nStreams; ++i) {
2   int offset = i * streamSize;
3   cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice, stream[i]);
4   kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
5   cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost, stream[i]);
6 }
```

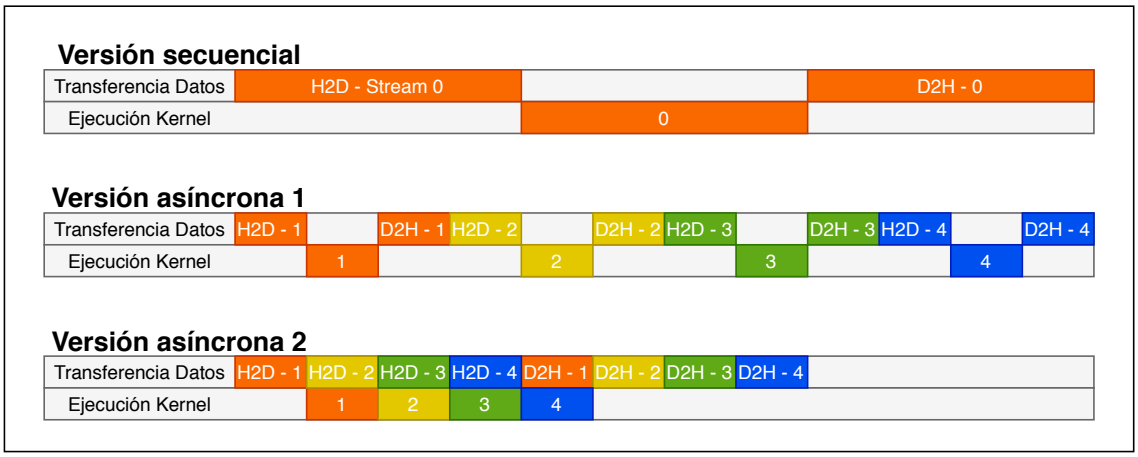

Figura 6.2: Streams en CUDA

6.4.1. Profiling de kernels CUDA

NVIDIA proporciona junto con el entorno CUDA múltiples herramientas que permiten analizar el rendimiento de los programas desarrollados (profiling). En particular, en este proyecto hemos usado el software Nsight Compute que permite, no solo medir los tiempos de ejecución de cada kernel y función, sino también extraer métricas de la propia GPU como: ocupación teórica y real de los SMs, número de registros utilizados, rendimiento de los warps, etc. Gracias a este software, podremos analizar lo eficiente que es la implantación de RandomX realizada sobre CUDA y localizar posibles puntos a mejorar.

7. CAPÍTULO

Análisis de la implementación CUDA de RandomX

En este capítulo, se realizará un estudio de la implementación realizada en CUDA de RandomX. Así mismo, se realizará un análisis del rendimiento y se comparará con la versión que se ejecuta sobre una CPU.

7.1. Análisis del código

La implementación CUDA de RandomX está compuesta por 6 kernels que se ejecutan en la GPU y varias funciones que se ejecutan en CPU. Básicamente, la implementación comienza generando el Dataset (ver Sección 5) y transfiriéndolo a la memoria de la GPU. Tras ello, todas las estructuras de datos necesarias para la ejecución en la GPU son generadas y se pasa el control a un bucle encargado de lanzar los kernels como se puede ver en la Figura 7.1.

Cada kernel se corresponde con una función de la implementación CPU de RandomX, sin embargo, en este caso la implementación es masivamente paralela utilizando múltiples hilos CUDA para calcular múltiples *nonces* a la vez. Como veremos más adelante, este número de hilos está únicamente limitado por la cantidad de memoria global de la que disponga la GPU.

A continuación se describirán la principales funciones que se ejecutan en la versión CUDA. Para conocer lo que hace cada una en profundidad consultar la Sección 6. La primera función es la encargada de inicializar el Dataset y transferirlo a la memoria de la GPU:

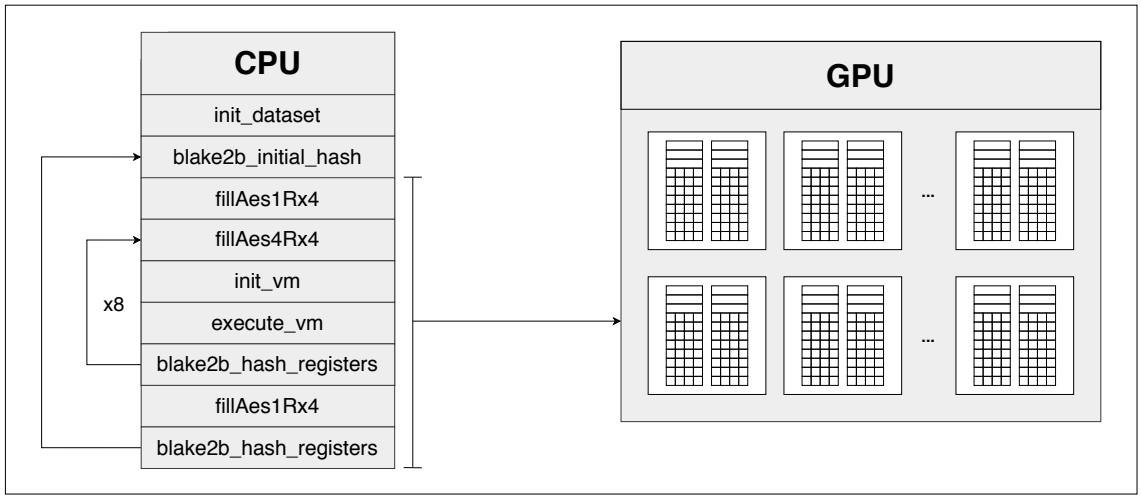

Figura 7.1: Funciones RandomX CUDA

- *init_dataset*: Función de inicialización del *buffer* de solo lectura denominado Dataset. Este último, se construye desde la memoria caché usando la función SuperscalarHash.

A continuación, para cada *nonce* que se quiere probar, se ejecutan la dos siguientes funciones. Hay que destacar que, como son kernels CUDA, cada función lanzara múltiples kernels en paralelo por lo que se evaluaran múltiples *nonces* a la vez:

- *blake2b_initial_hash*: Esta función genera un hash aleatorio de 64 bytes que se usara una semilla (S) que se usará como entrada de la siguiente función algoritmo.
- *fillAes1Rx4*: Esta función realiza una ronda del algoritmo AES usando como clave la semilla generada en el paso anterior. La salida del algoritmo se utiliza para inicializar el *Scratchpad* y se guarda el estado final (gen1).

Estos dos pasos se realizan antes de generar los 8 programas aleatorios que se generaran y ejecutarán en cada iteración del siguiente bucle:

- *fillAes4Rx4*: Esta función lleva a cabo 4 rondas del algoritmo AES utilizando como estado inicial el estado gen1. El estado de salida de esta función (gen4) será utilizado en el siguiente paso.
- *init_vm*: En este punto del proceso se inicializa la máquina virtual utilizando gen4 como generador para programar la maquina virtual, es decir, se generaran los bytes aleatorios que serán interpretados como programas.

- *execute_vm*: Ejecución de la máquina virtual programada en el paso anterior, es decir, del programa RandomX. Ésta es la parte esencial del algoritmo, en donde se realiza el trabajo computacionalmente más costoso. Esta función no tiene una salida explicita, pero modifica el conjunto de registros que será utilizado en el siguiente paso.

- *blake2b_hash_registers*: Al final de cada iteración se calcula el hash del conjunto de registros modificado en el paso anterior. Este hash se utilizará como estado inicial de la función *fillAes4Rx4* en la siguiente iteración.

La ultima iteracion del bucle cambia ligeramente y en vez de ejecutar la función *blake2b_hash_registers* utilizará las dos siguientes con el objetivo de incluir en el calculo del hash final, no solo el estado de los registros, sino también el estado del *Scratchpad*.

- *fillAes1Rx4*: Esta función realiza una ronda del algoritmo AES usando como clave el contenido del *Scratchpad*. La salida será utilizada para modificar los últimos 64 bytes del banco de registros.

- *blake2b_hash_registers*: En la ultima iteración se calcula el hash del conjunto de registros modificado en el paso anterior con el hash del *Scratchpad*. Este hash es el resultado final y habrá que verificar si cumple con la restricción de dificultad que impone la red. En caso afirmativo habremos logrado minar un bloque.

Como se ha comentado al principio de esta sección, cada función se corresponde con un kernel CUDA por lo que para ejecutarlos habrá que determinar el número de bloques y de hilos que se utilizarán. En la tabla 7.1 se detallan estos valores, así como otros relacionados con el rendimiento que serán analizados en la siguiente subsección. En cualquier caso, el número de kernels que se pueden lanzar de manera simultánea viene determinado por la cantidad de *Scratchpads* que se pueden reservar en memoria a la vez.

7.2. Métricas

Utilizando la herramienta Nsight Compute hemos realizado un análisis del código. Esto ha permitido medir el rendimiento de cada kernel en la GPU en términos de tiempo de ejecución y uso de recursos. La idea detrás de este análisis es identificar las funciones más pesadas y aquellas que no aprovechan de manera óptima los recursos de la GPU.

7.2.1. Warps/SM

En una GPU cualquier programa que quiera obtener el máximo rendimiento debería ser capaz de maximizar el uso de los recursos que se proporcionan. Para medir esta ocupación se pueden utilizar diferentes métricas, entre ellas warps/SM que mide, en media, la cantidad de warps que son ejecutados por SM. Esta métrica hay que compararla con el número de warps teóricos, que representa el número máximo de warps que puede ejecutar la GPU en condiciones óptimas de utilización.

Debido a ello, determinar el número de bloques y threads con el que se lanzarán los kernels es importante para conseguir tener en ejecución el máximo número de warps posible. Además, otro factor importante es el tamaño de los bloques que se asignan a cada SM. Si la ocupación de los SMs es inferior al 100 %, puede ser debido que los bloques no tienen suficientes warps (threads) para alcanzar ese 100 % de ocupación. Una posible solución podría ser el aumentar el tamaño de bloque. Por ejemplo:

- 16 bloques, 32 threads, 64 warps activos, 1 warps/bloque ->Máximo 16 warps activos (25 % de ocupación)
- 4 bloques, 128 threads, 64 warps activos, 4 warps/bloque ->Máximo 64 warps activos (100 % de ocupación)

7.2.2. Registros/SM

Otra métrica a tener en cuenta es el número de registros por SM. Cada SM tiene un número de registros compartidos entre todos los threads activos y puede convertirse en un factor que limita el número de bloques activos. Esto puede significar que el número de registros por thread que ha asignado el compilador debe ser reducido para aumentar la ocupación. Sin embargo, esta mejora en términos de ocupación puede suponer más accesos a memoria y, al final, degradar el rendimiento. Debido a ello, el efecto de lo que de primeras parecen ser optimizaciones debe ser evaluado cuidadosamente.

7.2.3. Memoria compartida/SM y sincronización de threads

Otra métrica que puede tener impacto en la ocupación es el uso de la memoria compartida/SM, memoria que se usa para comunicar threads activos de un mismo bloque. Este

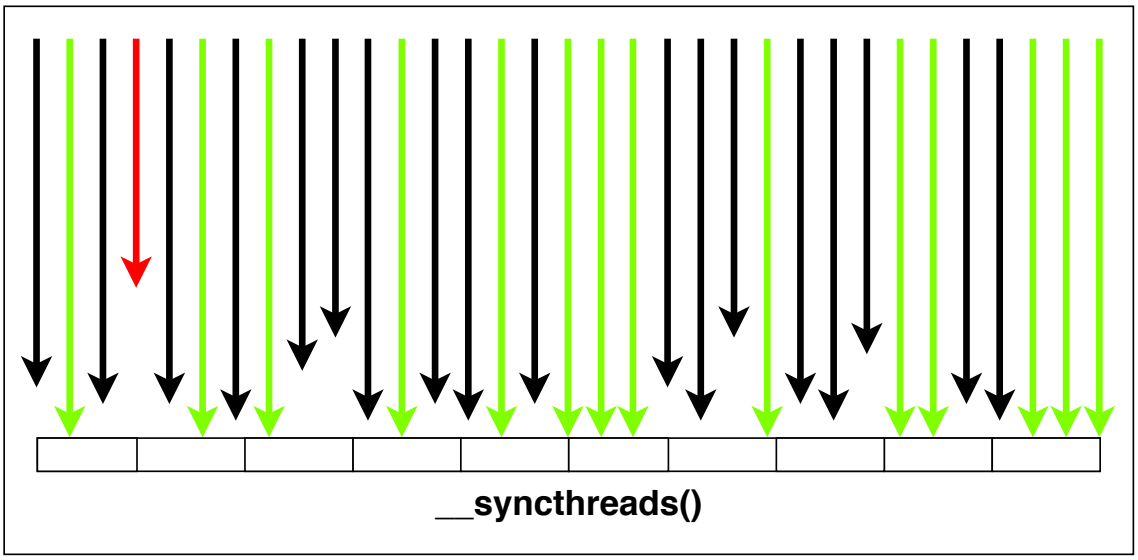

Figura 7.2: Representacion del efecto del uso de la función *__syncthreads()*

caso es similar al anterior, reducir la cantidad de memoria que necesita cada thread puede aumentar la ocupación utilizando más threads. De la misma forma que en el caso anterior, se debería encontrar un equilibrio entre el uso el tamaño de bloque y el uso de la memoria compartida porque utilizar más threads y menos memoria por thread podría suponer aumentar el número de accesos a memoria global.

Como se acaba de ver, los threads de un bloque pueden compartir datos usando la memoria compartida y acceder a ellos de manera coordinada. Sin embargo, en muchos casos es necesario utilizar barreras para sincronizar la ejecución y así garantizar que todos han terminado de escribir sobre esos datos compartidos. Para ello, CUDA dispone de las funciones *__syncthreads()* para sincronizar todos los threads de un bloque y *__syncwarp()* para sincronizar los threads de un warp. Como se puede ver en la Figura 7.2 el uso de estas funciones puede suponer un cuello de botella porque todos los threads deberán esperar al más lento y, dependiendo de las funciones que se ejecuten, este hecho puede suponer un *overhead* importante.

7.2.4. Análisis del *profiling*

En la tabla 7.1 se han representado los resultados obtenidos tras realizar el *profiling*. El análisis se realizará teniendo en cuenta la ocupación conseguida por cada función, analizando en cada caso los posibles motivos.

Como se puede ver las funciones *blake2b_initial_hash* y *blake2b_hash_registers* obtienen

KERNEL	BLOQUES	THREADS	REG/THREAD	WARPS	WARPS REAL	ms
blake2b_initial_hash	64	32	62	32	6.39	0,06
fillAes1Rx4	64	1024	32	48	26,10	25,80
fillAes4Rx4	64	1024	40	48	26	0,10
init_vm	512	32	50	12	10,78	9,25
execute_vm	1024	16	52	24	21,46	595,39
blake2b_hash_registers	64	32	96	20	6,45	0,06
TOTAL ITERACIÓN						630,66

Tabla 7.1: Profiling de la ejecucion de RandomX en una GPU (parametro density=2048)

un número muy bajo de warps/SM, 6,39 y 6,45 con respecto al máximo teórico, 32 y 20 respectivamente. Un análisis de ambas funciones revela que el motivo se puede deber a que acceden de manera muy intensa a memoria global para general los hashes iniciales, intermedio en cada iteración y finales. De todas formas, como se puede ver, el tiempo de ejecución de ambas funciones es 0,06 ms, por lo que su impacto en el tiempo de ejecución total, 630,66 ms, es despreciable.

En el caso de las funciones *fillAes1Rx4* y *fillAes4Rx4* el número de warps/SM es aproximadamente la mitad del máximo teórico, 26 frente a 48. Esto es debido a que al inicio de la ejecución todo los kernels de un bloque llenan una estructura de forma coordinada en memoria compartida y tienen que sincronizarse para garantizar que todos lo han hecho. En este caso podría ser interesante optimizar la función *fillAes1Rx4* que necesita una media de 25,80 ms para finalizar. Sin embargo, hay que tener en cuenta que esta función se ejecuta fuera del bucle por lo que únicamente se ejecuta una vez por programa.

Con respecto a las funciones *init_vm* y *execute_vm* ambas consiguen un valores de warps/SM muy cercanos al teórico por lo que mejorar esta métrica no es necesario.

Analicemos ahora el número de registros por thread. Como se puede observar en la tabla ningún thread utiliza más de 96 registros. Debido a que la GPU utilizada proporciona 255 registros a cada thread, esto no supone un problema que afecte a la ocupación.

7.2.5. Análisis temporal

Como se puede observar en la tabla 7.2, la duración de algunos de los kernels es despreciable con respecto al tiempo total. En concreto las funciones *blake2b_initial_hash*, *fillAes4Rx4* y *blake2b_hash_registers*. Como se ha comentado anteriormente, las funciones más interesantes para optimizar son aquellas que requieren un mayor tiempo de ejecución, en este caso, *execute_vm*. Con una duración de alrededor de 595 ms, supone

	CPU (AES Software)	CPU (AES Hardware)	GPU
blake2b_initial_hash	0,001	0,001	0,06
fillAes1Rx4	1,287	0,106	25,80
fillAes4Rx4 init_vm execute_vm blake2b_hash_registers	2,313	2,251	604,8
TOTAL ITERACIÓN	3,6	2,36	630,66

Tabla 7.2: Tiempo de ejecución de un programa RandomX en CPU con y sin AES hardware en GPU. Los tiempos están medidos en ms.

el 94 % del tiempo de cada iteración. Otras como *init_vm* y *fillAes1Rx4* no tienen gran interés por diferentes motivos. En el caso de la primera, la duración es relativamente corta y únicamente se ejecuta una vez por iteración en la fase de inicialización y en el caso de la segunda, como se ha visto anteriormente, la utilización de los recursos es casi óptima y el número de accesos a memoria muy bajo (ver siguiente sección).

A continuación, se va a realizar una comparación de tiempos de ejecución entre las versiones de GPU y de CPU, arquitectura para la que RandomX está específicamente diseñado. Los tiempos se corresponden con la ejecución de un programa RandomX y en el caso de de la CPU, se han ejecutado dos versiones, una que utiliza una implementación del algoritmo AES en software y otra que aprovecha las extensiones AES-NI de Intel para ser acelerada en hardware. La CPU utilizada ha sido un Xeon 6130 Gold.

Hay que señalar que los tiempos de CPU han sido medidos usando únicamente un core (1 hilo hardware) y se corresponden con el calculo de una única hash. Por su parte la GPU ejecuta múltiples kernels en paralelo (2048 en este caso). Como se puede observar, el tiempo requerido por la GPU es menor que el tiempo requerido por cualquiera de las dos versiones ejecutadas en CPU (630,6 ms frente a 3,6 * 2048 ms y 2,36 * 2048 ms). Cabe destacar también que el uso de las funciones AES-NI permiten reducir la ejecución de cada programa en más de un segundo.

Como conclusión de los resultados se puede decir que la implementación en GPU se aprovecha del paralelismo para mejorar los tiempos obtenidos en una CPU. Sin embargo, hay que tener en cuenta que la CPU utilizada dispone de 64 hilos hardware por lo que la ejecución con múltiples hilos de RandomX sería mucho más rápida que la ejecución en la GPU.

7.3. Accesos a memoria

Los accesos a memoria global de la GPU pueden suponer un cuello de botella para el rendimiento si los comparamos con los accesos a memoria compartida (del orden de 100 veces más lentos). El algoritmo RandomX, a parte de necesitar recursos de cómputo, necesita utilizar grandes cantidades de memoria. En concreto, la estructura del Dataset necesita 2080 MB y cada Scratchpad, que en la practica se corresponde con un hilo, necesita 2 MB. Debido a esto, ambas estructuras se almacenan en la memoria global de la GPU no pudiendo ser acomodadas en la memoria compartida.

La cantidad de memoria requerida para almacenar el Dataset (estructura compartida por todos lo hilos) impide el ejecutar más hilos en paralelo debido a que no queda memoria libre para crear más Scratchpads. En RandomX se realizan 2048 lecturas de 64 bytes al Dataset por iteración, haciendo un total de 16384 accesos por ejecución de un programa. Este número hay que multiplicarlo por el número de hilos que se ejecutan de manera concurrente, por lo que la presión ejercida sobre la memoria puede ser muy elevada.

En el siguiente capítulo, se evaluará el impacto que tiene el colocar el Dataset en memoria RAM (CPU) obligando a la GPU a utilizar el bus PCI Express para realizar las lectura. A cambio, se libraran 2080 MB de la memoria de la GPU permitiendo aumentar el nivel de paralelismo gracias a la ejecución de más de 1000 threads CUDA adicionales (2 MB * 1040 = 2080 MB).

8. CAPÍTULO

Mejoras en la implementación

Después de analizar en profundidad el funcionamiento de RandomX y CUDA, en este capítulo se describirán las mejoras que se han probado. Primero, se comenzara analizando dos intentos que no supusieron un aumento del rendimiento, A continuación se mostrara una tercera técnica en la que se obtuvieron algunas mejoras finalizando con una que era inicialmente el objetivo del proyecto pero que debido a circunstancias externas no se pudo realizar (ver Sección 9.5).

8.1. Pruebas realizadas

Después de realizar el profiling de utilización de la GPU se pudo comprobar como algunos de los kernels (warps) no eran capaces de utilizar de manera eficiente la GPU. Como primera modificación se intento mejorar el rendimiento de estos kernels siguiendo las recomendaciones de NVIDIA que básicamente sugieran aumentar el número de bloques y el tamano de estos (número de threads). Tras realizar algunas pruebas con los kernels *blake2b* se consiguieron mejoras marginales. En el caso de los kernel *fillAes* los bloques ya utilizaban el máximo de threads por bloque permitidos por lo que no se pudo modificar ese parametro.

Como segunda prueba, la intención era sacar la ejecución de del kernel fillAes4Rx1 de la GPU y generar los Scratchpads en memoria principal (ver Figura 8.1). Para ello se iba a aprovechar las instrucciones AES-NI implementadas en hardware para acelerar su ejecución. Debido a que CUDA utiliza un modelo de computación híbrido, ya existe un proceso

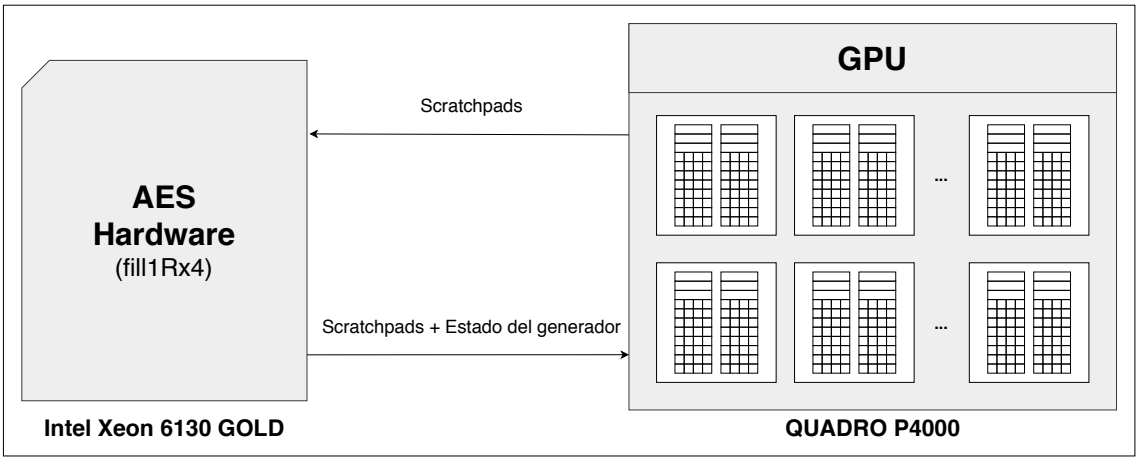

Figura 8.1: Descripción gráfica de la prueba de realización de fillAes4Rx4 en el hardware específico de la CPU.

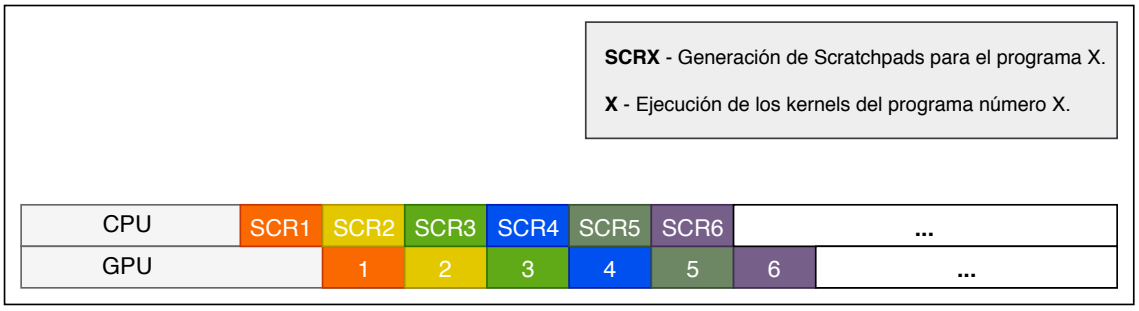

Figura 8.2: Descripción gráfica de la prueba de intercalado de generación de Scratchpads y ejecución de programas.

en la CPU gestionando las transferencias de datos y la ejecución de los kernels. Además, debido a que cada core de la CPU dispone de las instrucciones hardware necesarias, esto no debería suponer un aumento excesivo del consumo de energía ni de ciclos de CPU.

A pesar de que la implementación AES hardware es capaz de procesar hasta 10 GB/s utilizando instrucciones vectoriales, el calculo de los Scratchpads debería hacerse de manera secuencial. De todas formas, utilizando 512 bits la ejecución de la función *fill1Rx4* sobre 2912 Scratchpads de 2 MB cada uno tardaba menos de 500 ms.

Como se muestra en la Figura 8.2, una vez rellenados los Scratchpads en la CPU ahora hay que transferirlos a la GPU. Para ello se utilizaron streams que permiten solapar el calculo y las comunicaciones. El problema con esta propuesta es que la GPU no dispone de suficiente memoria para almacenar los Scratchpads generados en la CPU por lo que tiene que esperar hasta que memoria este disponible. Es en es momento cuando la transferencia se realiza haciendo despreciable la mejora conseguida usando AES hardware.

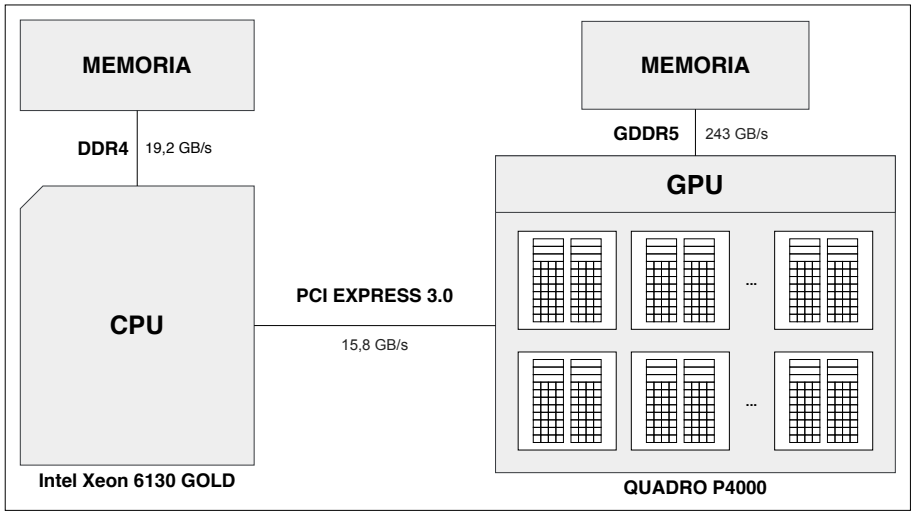

Figura 8.3: Representación de los ancho de banda de la memoria RAM, de la memoria de la GPU y del bus PCI Express 3.0.

8.2. Balanceo de accesos a memoria

La ejecución del algoritmo de RandomX es costoso en términos de memoria requerida. Solo el Dataset consume 2080 MB a los que hay que sumar 2MB por Scratchpad utilizado. Teniendo en cuenta que el número de Scratchpads que es posible utilizar en la GPU usada en este proyecto (8 GB de RAM) es de 2912, se requieren casi 6 GB para mantenerlos. En el análisis de RandomX se ha determinado que se realizan 16384 accesos para realizar lecturas de 64 bytes por programa ejecutado. Teniendo en cuenta que la GPU dispone de 1792 cores, se podrían llegar a producir en un corto periodo de tiempo 29360128 de accesos, lo que podría saturar el bus de acceso a la memoria de la GPU.

En la Figura 8.3 se ha representado la arquitectura de memoria de todo el sistema. Como se puede ver el ancho de banda de la memoria de la GPU (GDDR5) es de 243 GB/S en comparación con los 15,8 GB/s del bus PCI Express 3.0. Esto indica que acceder a la memoria principal del sistema puede tener un impacto negativo en el rendimiento.

Sin embargo, en RandomX, el colocar el Dataset en memoria principal supondría liberar memoria de la GPU, lo que permitiría ubicar más Scratchpads en la misma y en consecuencia aumentar el número de hilos lanzados. Este aumento del paralelismo también implica que se van a producir más accesos al Dataset, por lo que es posible que el aumento de rendimiento debido a la ejecución concurrente de más hilos pueda verse afectado negativamente por los accesos a memoria principal a través del bus PCI Express.

Una posible solución para reducir el número de accesos a memoria principal sería ubicar

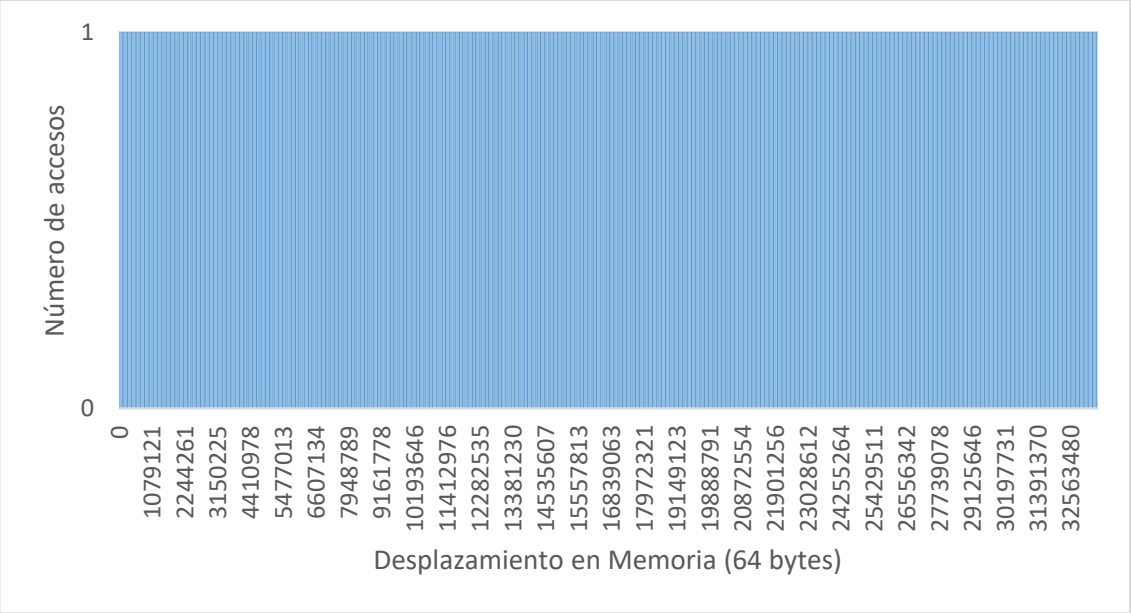

Figura 8.4: Distribución de los accesos al Dataset. Cada acceso realiza una lectura de 64 bytes.

parte del Dataset en la memoria de la GPU y parte en la memoria del sistema. Para ello deberíamos ser capaces de interceptar los accesos al Dataset y redirigirlos a la memoria correspondiente. Para dividir de manera eficiente el Dataset sería necesario conocer el patrón de accesos que RandomX realiza sobre él. Para ello, hemos modificado el código de RandomX monitorizando las direcciones a las que se accede.

En la Figura 8.4 se han representado los accesos al Dataset (lecturas de 64 bytes) de un kernel. Como se puede observar el patrón de acceso es completamente uniforme y en ningún caso se accede dos veces a la misma posición. Si nos centramos en una región más reducida del Dataset, (ver Figura 8.5) se puede observar el mismo comportamiento. Debido a esto, es imposible optimizar el acceso a ciertas áreas, porque todas son usadas de manera similar.

A pesar de ello, se podría dividir el Dataset en dos partes de diferentes tamaños y colocar cada una de ellas en una memoria diferente. Modificando la forma en la que RandomX accede al Dataset podríamos balancear los accesos. De esta manera, podríamos liberar parte de la memoria de la GPU para ejecutar más hilos (Scratchpads) y a la vez reduciríamos el número de accesos a la memoria principal del sistema.

El código encargado de calcular el desplazamiento dentro del Dataset está representado en código fuente 8.1. En la instrucción número 3 es donde se calcula el desplazamiento con respecto a la dirección base del Dataset. Es en este punto donde se pueden interceptar

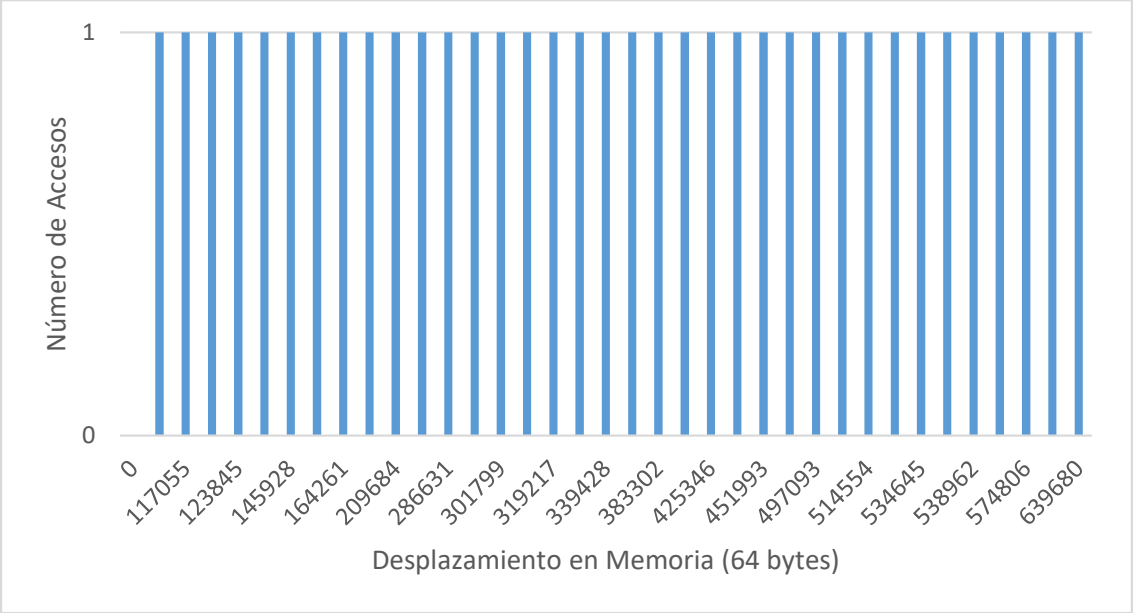

Figura 8.5: Distribución de los accesos a las primeros 640.000 posiciones de memoria.

los accesos al Dataset y dirigirlos hacia la memoria en la cual queramos hacer la lectura.

Código fuente 8.1: Snippet en el que se realizan los accesos al Dataset.

```
1 mx ^= *readReg2 ^ *readReg3;
2 mx &= CacheLineAlignMask;
3 const uint64_t next_r = *r ^ *(const uint64_t*)(dataset + ma + sub * 8);
4 *r = next_r;
5 *p1 = next_r;
6 *p0 = bit_cast<uint64_t>(f[0]) ^ bit_cast<uint64_t>(e[0]);
7 uint32_t tmp = ma;
8 ma = mx;
9 mx = tmp;
```

Con el objetivo de evaluar el impacto en el rendimiento del paralelismo frente a los accesos a memoria principal del sistema, hemos desarrollado e implementado las estrategias siguientes:

- GPU: El Dataset completo se encuentra en la memoria de la GPU.
- 75–25: El 75 % del Dataset reside en la GPU y el 25 % restante en la memoria principal.
- 50–50: Se divide el Dataset en dos partes iguales, una en la GPU y otra en la CPU.

MEMORIA	2912	**3200**	3456	3712	3968
GPU	393,79	-	-	-	-
75-25	389,92	**465,27**	-	-	-
50-50	384,02	**457,46**	405,41	-	-
25-75	380,15	**452,00**	400,56	386,58	-
CPU	376,26	**448,30**	396,99	382,10	368,26

Tabla 8.1: Scratchpads/hashes por segundo utilizando las estrategias de uso de la memoria.

- 25–75: El 25 % del Dataset reside en la GPU y el 75 % restante en la memoria principal.
- CPU: El Dataset en su totalidad reside en la memoria de la CPU.

Para realizar los experimentos se ha lanzado número diferente de kernels dependiendo del número máximo de Scratchpads simultáneos. Como se puede ver en la Tabla 8.1, este número varia dependiendo de la estrategia utilizada. Por ejemplo, la estrategia GPU puede procesar a la vez como máximo 2912 Scratchpads, pero si relajamos la cantidad de memoria de la GPU que utilizamos (estrategia 75-25) podemos procesar 288 Scratchpads mas de forma simultanea. El mayor número de Scratchpads que se pueden procesar a la vez se consigue utilizando la estrategia CPU en la cual se dispone de toda la memoria de la GPU para los Scratchpads (3968).

Si nos fijamos en los resultados (número Scrachpads/hashes por segundo) el rendimiento mas alto se produce cuando se ejecutan 3200 kernels (Scratchpads) de manera simultanea. Estos resultados se obtienen para todas las estrategias demostrando que liberar “presion” de la memoria de la GPU afecta positivamente al rendimiento. Además, se puede ver que manteniendo un numero alto de Scratchpads en la GPU (accesos locales) utilizando la estrategia *75-25* se obtiene el mayor rendimiento. Esto tiene sentido, debido a que solo la cuarta parte de las lecturas se realizarán usando el bus PCI Express.

Un resultado inesperado se produce al aumentar el numero de Scratchpads. Como se puede ver, al aumentar dicho numero a 3456, 3712 y 3968 se empieza a reducir el rendimiento obteniéndose el peor resultado cuando no se utiliza la memoria de la GPU.

Por poner en perspectiva los resultados obtenidos en la Figura 8.6 se ha representado los Scrachpads/hashes que cada estrategia es capaz de procesar en dos minutos. Este resultado es interesante debido a que el tiempo medio de publicación de un nuevo bloque en Monero es de dos minutos. El valor máximo es obtenido por la estrategia 75-25 utilizando 3200

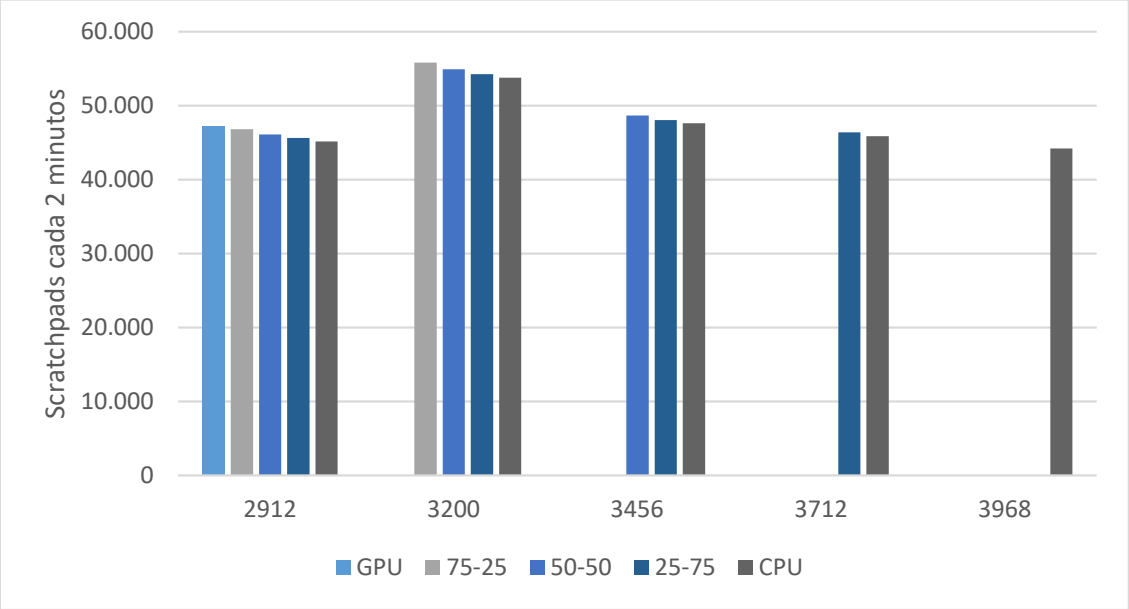

Figura 8.6: Scratchpads/hashes cada 2 minutos utilizando las estrategias de uso de la memoria.

Scratchpads, 55833 Scratchpads/hashes cada 2 minutos frente a la versión original que consigue 47255.

Como conclusión final ha quedado claro de la importancia que pueden tener los accesos a memoria en la GPU. Para RandomX en particular, que hace un uso intensivo de la memoria, el balancear esos accesos con la memoria principal del sistema, aumentando el paralelismo, proporciona un aumento del rendimiento importante.

8.3. Proyecto inicial

Inicialmente la idea era realizar el proyecto sobre otro hardware, en concreto, dos NVIDIA RTX-2080TI con 11GB de memoria principal interconectadas mediante una conexión NVLink pero no se pudo llevar a cabo debido a un imprevisto (ver Sección 9.5).

El uso de estas GPUs era muy interesante debido a la conexión NVLink, una tecnología de NVIDIA descrita anteriormente. Mas concretamente, el proyecto consistía en analizar el impacto que produce ejecutar RandomX en estas dos GPUs interconectadas por una conexión de alta velocidad. Con un ancho de banda muy superior al del PCI Express, dicha conexión hubiera permitido dividir el Dataset entre las memorias principales de las dos tarjetas, lo que aumentaría notablemente el número de programas que se pueden

lanzar al mismo tiempo, y habría permitida la comunicación entre ambas placas con una latencia muy baja.

Además, estas dos tarjetas disponen de hardware específico para realizar operaciones vectoriales lo que habría permitido vectorizar ciertas operaciones, como por ejemplo las funciones que utilizan AES. Este proyecto queda como trabajo futuro.

9. CAPÍTULO

Planificación

En este capítulo se describirá la planificación realizada al inicio del proyecto describiendo las tareas a realizar y una planificación de tiempos. Así mismo, se ha añadido una sección de imprevistos en la que se describirán algunas circunstancias que han obligado a reorientar ciertas partes del proyecto planificado originalmente.

9.1. Descripción de tareas

Para visualizar de una forma clara las fases que componen las fases este proyecto se ha dividido el trabajo a realizar en diferentes paquetes de trabajo, que a su vez, contendrán las tareas a realizar en cada uno de ellos.

- **Paquete de Diseño de Solución (DS)**
  - **DS.T1 Monero**: Análisis del funcionamiento de la plataforma.
  - **DS.T2 CUDA**: Estudio y pruebas de CUDA.
  - **DS.T3 Randomx**: Análisis en profundidad del algoritmo RandomX y sus posibles mejoras.
- **Paquete de Trabajo Entorno (E)**
  - **E.T1 Preparación del entorno de trabajo**: Obtención de acceso al hardware e instalación de dependencias en la máquina (CUDA).

- **E.T2 Minero.** Descarga y puesta en marcha del software de minado en la máquina.

- **Paquete de CUDA (C)**
  - **C.T1 Análisis de tiempo**: Medición de tiempos de las distintas fases del minado para determinar los puntos de mayor carga de trabajo.
  - **C.T2 Análisis de código**: Análisis en profundidad de las funciones más costosas para identificar estrategias de mejora.
  - **C.T3 Implementación**: Implementación de las mejoras para optimizar el código.
  - **C.T4 Pruebas**: Evaluación de la implementación.

- **Paquete de Planificación (P)**
  - **P.T1**: Tareas relacionadas con la identificación de requisitos, toma de decisiones iniciales, análisis del proyecto y resolución de dudas.
  - **P.T2.** Planificación inicial orientada a la preparación del entorno de desarrollo del proyecto.
  - **P.T3.** Actualización, si fuera necesaria, de la planificación.

- **Paquete de Seguimiento y Control (SC)**
  - **SC.T1**: Recogida de información relevante sobre el desarrollo del proyecto.
  - **SC.T2**: Contraste de la información con lo planificado, identificación de desviaciones significativas y de posibles riesgos.
  - **SC.T3**: Garantizar las condiciones para el éxito del proyecto.

- **Paquete de Informe (I)**
  - **I.T1**: Realización del informe del proyecto.

9.2. Periodo de desarrollo de las tareas

En la Figura 9.1 se ha representado el periodo de tiempo durante el cual se realizara cada una de las tareas. Como se puede apreciar las tareas relacionadas con el seguimiento y control del proyecto se extienden desde el inicio hasta la finalizacion del proyecto.

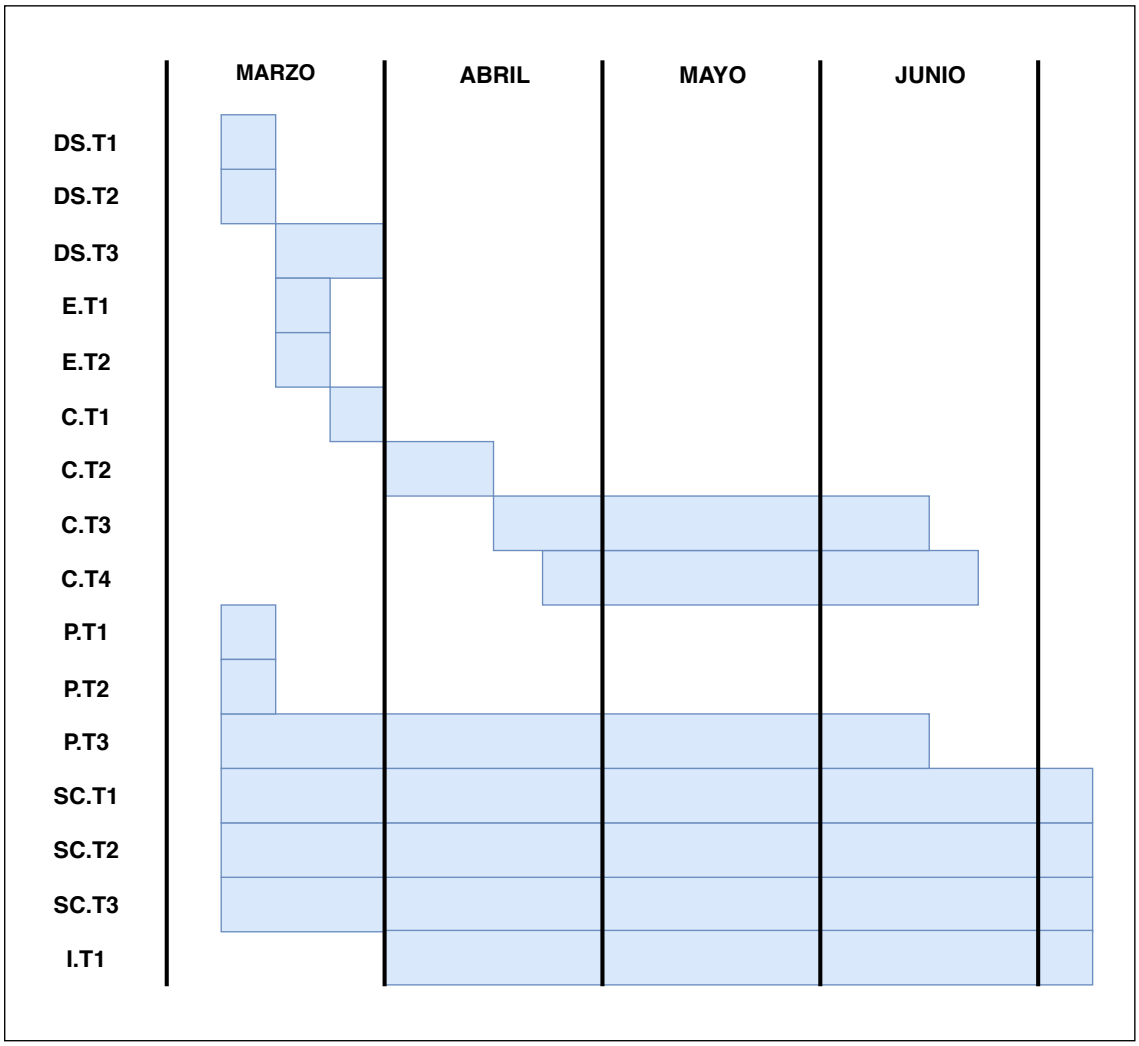

Figura 9.1: Periodos de realización de las tareas.

9.3. Estimación de dedicación a cada una de las tareas

Una vez explicadas las tareas que componen cada fase del proyecto, pasaremos a representar en la Tabla 9.1 la información referente al coste temporal estimado de cada una de ellas y la información sobre el tiempo real invertido en cada tarea, identificando el paquete de trabajo al que pertenecen.

El modo en el que se estimaron las horas representadas se explicara en la siguiente subsección. Quiero destacar que el tiempo que sera necesario para la realización de la presentación no ha sido incluido en esta tabla.

TAREAS	HORAS PLANIFICADAS	HORAS REALES
Diseño de Solución	14	24
Monero	4	3
CUDA	6	6
RandomX	4	15
Entorno	2	2
Preparación del Entorno	1	1
Minero	1	1
CUDA	185	210
Análisis de Tiempo	5	8
Análisis de Código	70	105
Implementación	60	55
Pruebas	50	42
Planificación	14	10
P.T1	6	4
P.T2	6	4
P.T3	2	2
Seguimiento y Control	14	18
SC.T1	10	12
SC.T2	2	3
SC.T3	2	3
Informe	40	50
I.T1	40	50
TOTAL	269	314

Tabla 9.1: Tabla de desviaciones.

9.4. Plan de riesgo

En proyectos de una duración tan larga y con tantas tareas es necesario contar con un plan de riesgo para prevenir posibles desviaciones debidas a factores imprevisibles a priori. El desarrollo de este punto nos proporcionará una mayor flexibilidad para abordar circunstancias no previstas. Por lo tanto, para poder reaccionar ante retrasos y problemas que seguramente ocurrirán, se pone en marcha el siguiente plan de riesgo, el cual se tendrá en cuenta lo largo del ciclo de vida del proyecto.

Debido a la inexperiencia a la hora de realizar un proyecto la estimación de los costes temporales exactos que se aproximen a la realidad es imposible. Por ello, se ha decidido hacer una estimación del tiempo y añadir un tiempo extra a cada paquete de trabajo. De esta manera, el coste temporal de los paquetes que se estima van a requerir de un mayor

tiempo (más de 25 horas) será extendido con un mayor tiempo extra que el que se añada a las estimaciones para paquetes más cortos.

9.5. Análisis de las desviaciones

Como todos sabemos, la pandemia del COVID-19 ha afectado de manera importante al desarrollo de las actividades cotidianas. Es inevitable que el estado de alarma en el que se ha encontrado el país durante varios meses haya afectado al desarrollo de este proyecto. De hecho, ha tenido un gran impacto. La idea inicial consistía en la realización de este estudio en un hardware específico que nos iba a permitir evaluar ciertas técnicas. Por circunstancias desconocidas ese hardware sufrió un fallo y debido a su localización física (el edificio Korta de la UPV-EHU) no ha podido ser puesto en marcha de nuevo. Es por eso, que hubo que desarrollar un plan de contingencia y realizar el proyecto en otra tarjeta gráfica la cual carecía de ciertas características necesarias para llevar a cabo la idea inicial.

10. CAPÍTULO

Conclusiones

Para finalizar el documento se presentaran algunas conclusiones extraídas de su realización así como algunas lineas de trabajo futuro que han quedado abiertas.

Respecto al apartado técnico, ha supuesto un reto el comprender el funcionamiento, tanto de Monero, como de RandomX. Una de las partes más exigentes sobre Monero ha sido entender el uso que hace de la criptografía para garantizar la anonimidad, debido sobre todo a la notación matemática que se utiliza. Por otro lado, RandomX es una prueba de trabajo extremadamente compleja y para entenderla ha sido necesario el estudio de múltiples áreas, desde la arquitectura de computadores para entender como RandomX evita su ejecución eficiente en ASICs has el uso del algoritmo AES y su implementación AES-NI.

Además de todo ello, entender el funcionamiento de la implementación CUDA ha sido un reto debido a su complejidad y al uso avanzado de C++ que se hace. Debido a las circunstancias expuestas, el objetivo del proyecto tuvo que ser modificado lo cual también supuso un reto para adaptarse a la arquitectura de la GPU y a la investigación de posibles mejoras.

A pesar de la dificultad, creo que ha merecido la pena porque me ha permitida aprender multitud de nuevos conceptos sobre GPUs, criptografía, CPUs y programación en general. Además, me ha servido para enfrentarme a un reto "super" avanzado desde cero habiendo logrado superarlo. Estoy seguro de que todo esto me servirá en el futuro para desarrollar de manera más eficiente mi futuro profesional.

Respecto a la planificación del proyecto en si, he aprendido la necesidad de saber gestio-

nar los riesgos que pueden surgir en cualquier momento. Siempre he considerado importante realizar una planificación de cualquier objetivo que quiera cumplir pero después de realizar este proyecto creo que es algo imprescindible.

Debido al cambio de objetivos del proyecto, como posible trabajo futuro queda la idea original del proyecto en la que se planeaba acelerar RandomX utilizando el hardware vectorial, así como disminuir el impacto que tienen los accesos a memoria utilizando los enlaces de alta velocidad NVLink presentes en la GPUs modernas.

Bibliografía

[CUDA, 2011] CUDA (2011). Cuda c/c++ basics. https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf. Visitado: 2020-06-12.

[Gilbert and Handschuh, 2004] Gilbert, H. and Handschuh, H. (2004). Security analysis of sha-256 and sisters. In Matsui, M. and Zuccherato, R. J., editors, *Selected Areas in Cryptography*, pages 175–193, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Intel, 2020a] Intel (2020a). Intel® advanced encryption standard (aes) new instructions set. https://software.intel.com/content/www/us/en/develop/articles/intel-advanced-encryption-standard-instructions-aes-ni.html. Visitado: 2020-06-18.

[Intel, 2020b] Intel (2020b). Intel® advanced encryption standard instructions (aes-ni). https://software.intel.com/content/www/us/en/develop/articles/intel-advanced-encryption-standard-instructions-aes-ni.html. Visitado: 2020-06-21.

[Nakamoto, 2008] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

[SChernykh, 2020] SChernykh (2020). Randomx_cuda. https://github.com/SChernykh/RandomX_CUDA. Visitado: 2020-06-21.

[Tevador, 2020] Tevador (2020). Randomx/master. https://github.com/tevador/RandomX/tree/master/src. Visitado: 2020-06-21.

[van Saberhagen, 2013] van Saberhagen, N. (2013). Cryptonote v 2.0.

[Wikipedia, 2020] Wikipedia (2020). Thread block (cuda programming). https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming). Visitado: 2020-06-10.