Paralelizando la máquina Enigma

Grado de Ingeniería Informática

Ingeniería de computadores
Facultad de Informática de Barcelona
Universidad Politécnica de Cataluña

Autor: Iván Reolid Torres
Director: Agustín Fernández Jiménez
(Departamento de Arquitectura de Computadores)
Data de defensa: 28/06/2022

Resumen

A principios de la segunda guerra mundial, surgió el mayor método de encriptación hasta entonces. Este, se realizaba grácias a una máquina que había sido diseñada por los alemanes para fines del mercado, y que el ejército alemán la mejoró para encriptar sus mensajes durante el periodo bélico, la máquina Enigma.

Durante este tiempo, un grupo de científicos ingleses estuvo trabajando junto con Alaran Turing para poder descencriptar estos mensajes. Utilizar fuerza bruta para este fin era una práctica impensable, debido a la complejidad que tenía la máquina Enigma (su total de combinaciones iniciales posible era de 158 962 555 217 826 360 000). Por este motivo, tuvieron que buscar otra alternativa y la encontraron, el método de contradicciones. A partir de una hipótesis inicial (una combinación de la enigma inicial), llegar a una contradicción para poder descartar todas las combinaciones que han ido apareciendo hasta ese momento. Con esto, se llegaba finalmente a una correcta.

Este método de contradicciones fue llevado a un nivel superior cuando encontraron fallos en las criptografías de los militares nazis (repeticiones de patrones en sus mensajes) y una funcionalidad reducida que tenía la máquina Enigma: una letra no podía encriptarse por ella misma.

Este proyecto trata de plasmar y simular todo el trabajo que realizaron mediante código y *hardware* actual, para dar una visión de lo importante que puede llegar a ser evitar la fuerza bruta cuando se puede. Se realizará un código de fuerza bruta para compararlo con el método de contradicciones.

Resum

A principis de la segona guerra mundial, va sorgir el mètode més gran d'encriptació fins llavors. Aquest es realitza gràcies a una màquina que havia estat dissenyada pels alemanys com a únic objectiu el mercat, i què l'exèrcit alemany la va millorar per encriptar els seus missatges durant el període bèl·lic, la màquina Enigma.

Durant aquest temps, un grup de científics anglesos va estar treballant juntament amb Alaran Turing per poder descencriptar aquests missatges. Utilitzar força bruta per a aquest fi era una pràctica impensable, a causa de la complexitat que tenia la màquina Enigma (el seu total de combinacions inicials possible era de 158.962.555.217.826.360.000). Per això, van haver de buscar una altra alternativa i la van trobar, el mètode de contradiccions. A partir d'una hipòtesi inicial (una combinació de l'enigma inicial), arribar a una contradicció per poder descartar totes les combinacions que han aparegut fins aleshores. Amb això, arriba finalment a una de correcta.

Aquest mètode de contradiccions va ser portat a un nivell superior quan van trobar mancances en les criptografies dels militars nazis (repeticions de patrons als seus missatges) i una funcionalitat reduïda que tenia la màquina Enigma: una lletra no podia encriptar-se per ella mateixa.

Aquest projecte tracta de mostrar i simular tota la feina que van realitzar mitjançant codi i maquinari actual, per donar una visió de com pot ser d'important evitar la força bruta quan es pot. Es farà un codi de força bruta per comparar-lo amb el mètode de contradiccions.

Abstract

At the beginning of the second world war, the greatest encryption method until then emerged. This was done thanks to a machine that had been designed by the Germans for market purposes, and that the German army improved to encrypt their messages during the war period. This machine was called Enigma.

During this time, a group of English scientists was working together with Alaran Turing to decrypt these messages. Using brute force for this purpose was an unthinkable practice, due to the complexity of the Enigma machine (its total number of possible initial combinations was 158,962,555,217,826,360,000). For this reason, they had to look for another alternative and they found it, the method of contradictions. Starting from an initial hypothesis (a combination of the initial enigma), arrive at a contradiction in order to rule out all the combinations that have been appearing up to that moment. With this, a correct one was finally reached.

This method of contradictions was taken to a higher level when they found flaws in the Nazi military's cryptography (pattern repetitions in their messages) and a reduced functionality that the Enigma machine had: a letter could not be encrypted by itself.

This project tries to capture and simulate all the work they did using current code and hardware, to give a vision of how important it can be to avoid brute force when possible. A brute force code will be performed to compare it with the contradictions method.

Índice de contenidos

Índice de Figuras

Índice de tablas

1. Encriptación histórica

1.1 Necesidad de encriptar los mensajes

A lo largo de la historia, el hombre ha estado en continuas confrontaciones con el hombre, y por lo tanto, ha sentido la necesidad de enviar mensajes a sus aliados sin que estos mensajes, si eran interceptados por el enemigo, pudieran ser comprensibles.

Hagamos un repaso a cómo ha ido evolucionando la criptografía a lo largo de la historia hasta el punto histórico donde está ambientado este proyecto.

1.2 Orígenes de la criptografía, criptografía clásica

Se calcula que el cifrado de mensajes se lleva practicando desde hace más de 4.000 años[1] y que, precisamente, el origen de dicha palabra proviene del griego *krypto*, que significa "oculto" y *graphos*, escribir. Por lo tanto, podría ser traducido como escritura oculta.

En la Biblia, hay referencias al Atbash, utilizado para cifrar mensajes y remontado al año 600 a.C. Este método de encriptado era un tipo de cifrado por sustitución[2], es decir, un método de cifrado por el que unidades de texto plano son sustituidas con texto cifrado siguiente un sistema regular[3]; las "unidades" pueden ser de una sola letra (el caso más común), pares de letras, tríos de letras, mezclas de lo anterior… Por lo que el receptor realiza una sustitución inversa para descifrar el texto.

En el caso particular del Atbash, esta sustitución se hacía entre la primera letra y la última, la segunda y la penúltima y así sucesivamente. Un ejemplo de cómo quedaría el alfabeto español en una tabla Atbash se muestra en la Figura 1.

Original	a	b	c	d	e	f	g	h	i	j	k	l	m	n	ñ	o	p	q	r	s	t	u	v	w	x	y	z
Clave	Z	Y	X	W	V	U	T	S	R	Q	P	O	Ñ	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Figura 1. Alfabeto español en una tabla Atbash[2]

Los espartanos también son estudiados por haber utilizado la criptografía para proteger sus mensajes. Concretamente, utilizaban una técnica conocida como cifrado por transposición. Esta técnica se basaba en cambiar la posición de las unidades de texto plano siguiendo un esquema bien definido[4]. Estas unidades de texto podían ser de una sola letra (el caso más común), pares de letras, tríos de letras, mezclas de lo anterior… Como era el tipo de cifrado más utilizado en la criptografía clásica, y por lo tanto, al tener que hacer los cálculos por medios muy básicos, normalmente el algoritmo se basaba en un diseño geométrico o en el uso de artilugios mecánicos. Este tipo de algoritmo es de clave simétrica, ya que tanto emisor como receptor necesitan saber dicha clave para realizar su función.

En el caso de los espartanos, utilizaban la escítala, Figura 2, como artilugio mecánico. Era un bastón de un diámetro determinado que se utilizaba para enroscar en torno a sí, una tira de cuero u otro material, en el que estaba escrito el mensaje. Al enrollarse alrededor del bastón, aparecía el mensaje correctamente escrito. Por lo tanto, y como hemos mencionado, el diámetro exacto de dicha escítala debía de ser conocido por ambos bandos.

Figura 2. Escítala espartana[5]

Por último, de la Antigua Roma procede el tipo de cifrado más conocido, el cifrado César, atribuido al mismo Julio César. Este tipo de cifrado pertenecía a los tipo de cifrado por sustitución[6] ya mencionados con la peculiaridad de que una letra en el texto original era reemplazada por otra letra que se encontraba a un número fijo de posiciones más adelante en el alfabeto. Por ejemplo, con un desplazamiento de 3, y además, según el historiador romado Suetonio[1], el más utilizado por Julio César, la A sería sustituida por la D, la B por la E, etc.

1.3 Criptografía en la Edad Media

En esta época es donde la criptografía tuvo su gran revolución, ya que en el siglo IX, Al-Kindi, un científico árabe de muchas disciplinas[7], reconocido por ser de los primeros que hicieron traducir al árabe la obra de Aristóteles y sobretodo, por sentar una de las bases

fundamentales para romper mensajes cifrados gracias al estudio del Corán; el análisis de frecuencia.

Se basaba en analizar patrones en los mensajes cifrados para localizar repeticiones y buscar la correlación con la probabilidad de que aparezcan determinadas letras en un texto escrito en un idioma concreto.

Ibn al-Durayhim y Ahmad al-Qalqashandi, dos matemáticos y criptologos árabes, también profundizaron en los análisis de frecuencia y trabajaron en el desarrollo de códigos más robustos al aplicar múltiples sustituciones a cada letra de un mensaje (rompiendo así los patrones que podían hacer que un código se rompiese).

1.4 Criptografía en el Renacimiento

Una de las figuras clave de esta época en esta disciplina fue Leon Battista Alberti[1], secretario personal de los tres Papas en los Estados Pontificios.

Al igual que los nombrados recientemente, Ibn y Ahmad, Alberti trabajaría en el cifrado polialfabético y desarrollaría un sistema de codificación mecánico (basado en discos) conocido como cifrado de Alberti. Está formado por dos discos, uno fijo y otro móvil. El disco fijo tiene un diámetro que es superior en un noveno al del disco móvil[8], tal y como se muestra en la Figura 3. El disco fijo está dividido en 24 sectores, repartidos en 20 para todas las letras del alfabeto ordenadas (excepto la J, U, W e Y, ya que en latín no existían), y los 4 sobrantes con números del 1 al 4 para que luego, estos sectores coincidan con los sectores del disco móvil. El disco móvil tendrá también 24 sectores, que coincide con el número total de carácteres con el que consta el alfabeto latino, siendo el vigesimocuarto “et”[8]. En este caso, los caracteres no están ordenados.

Figura 3. Discos del cifrado de Alberti[9]

Tanto emisor como receptor, tienen que compartir un “libro de códigos”[9], el cual, indicará que letra del disco móvil será utilizada como índice respecto al disco fijo para cada giro del disco móvil. Vamos a ver a continuación los dos métodos de encriptación que existían.

Partiendo de querer encriptar el siguiente mensaje: LAGVERRASIFARA, que significa la guerra se hará… tenemos que existe la “R” doble, la cual, traerá problemas por como se distribuyen los discos. Aquí tenemos dos opciones, eliminarla o introducir un carácter nulo.

Para el método de introducción del carácter nulo, el texto que queremos encriptar quedará tal que así: LAGVER2RASIFARA, habiendo introducido un número entre las dos “R” como carácter nulo.

En este caso, tenemos que indicar los cambios de giro para el disco móvil de alguna forma diferente a un número, puesto que como hemos dicho, este simboliza un carácter nulo. Se utiliza por ejemplo el carácter “_”, por lo que el texto a encriptar queda: _LAGVER2RA_SIFARA.

Ahora, mediante el libro de códigos, ambas partes deben de ser conocedoras de los índices. Por ejemplo, antes del primer giro, situaremos la g con la A, después del primer giro la g con la Q, y así sucesivamente. Por lo que en este método, el índice es una letra del disco móvil (la g). A parte, deben de ser conocedoras del orden en que aparecen los carácteres en el disco móvil. En la Figura 4 vemos el estado inicial para este ejemplo.

```
ABCDEFGILMNOPQRSTVXZ1234 Anillo fijo
gklnprtuz&xysomqihfdbace Anillo móvil
```

Figura 4. Estado inicial de los discos[9]

El primer carácter “_” se encriptará como la letra perteneciente a nuestro índice (“A”), ya que como dijimos, señala el cambio de giro del disco móvil. Después, a partir de la disposición de la Figura 4, podemos seguir encriptando hasta encontrar otro “_”, tal y como se muestra en la Figura 5 y donde, se encriptará con la siguiente letra para nuestro índice (“Q), tal y como está en el libro de códigos.

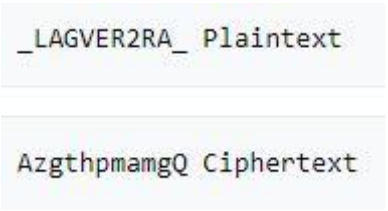


Figura 5. Encriptado de giro a giro mediante introducción de carácter nulo[10]

Una vez en este punto, se hará el giro en el disco móvil, haciendo coincidir la “Q” con la “g”, tal como vemos en la Figura 6, y podremos seguir encriptando con el resultado final de la Figura 7.

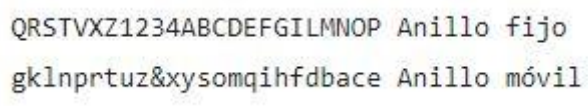


Figura 6. Nuevo estado de los discos después de un giro[9]

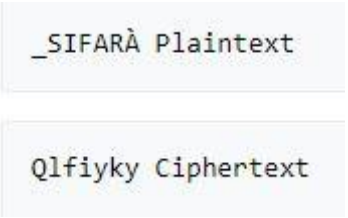


Figura 7. Encriptación final mediante introducción de carácter nulo[10]

Cabe destacar que tanto la “A” como la “Q” que salen en la encriptación serán ignoradas, ya que se sabe que es solo el indicador de un giro.

Para el segundo método, el de eliminación de la letra “R” repetida, indicaremos con un número cuando hay un giro del disco móvil, por ejemplo con el 3: LAGVERA3SIFARA.

En este método, el índice ya no es una letra del disco móvil, sino una letra del disco fijo. Por ejemplo, vamos a ubicar una posición inicial en que a la letra “A”, le pertenece la letra “m”, tal y como vemos en la Figura 8.

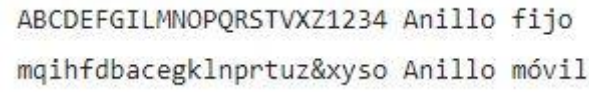


Figura 8. Estado inicial de los discos[9]

A partir de aquí, podemos empezar a encriptar hasta llegar al número, Figura 9. Una vez llegados a este punto, situaremos la letra que corresponde al número (3 y por lo tanto, “s”) a nuestro índice, que es la “A”, Figura 10.

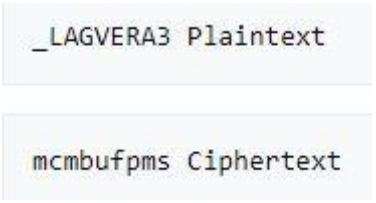


Figura 9. Encriptado de giro a giro mediante eliminación[10]

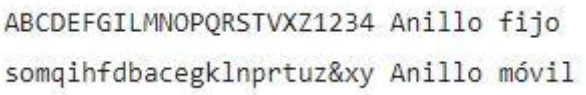


Figura 10. Nuevo estado de los discos[9]

Y ya solo quedará encriptar lo que falta de texto, Figura 11.

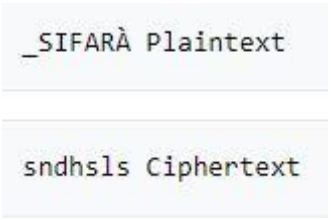


Figura 11. Encriptación final mediante eliminación[10]

Por lo tanto, el método de cifrado de Alberti presenta una clara ventaja: no es posible atacarlo con el análisis de frecuencias comentado anteriormente. Además, tiene un extra en la seguridad ya que para poder desencriptar un mensaje el receptor necesita de un disco exactamente igual al que sirvió para crear el cifrado. Esto también es un problema, ya que si alguien no intencional posee de un disco igual, descifrará el mensaje con facilidad.

Por último, mencionar que el cifrado de Alberti es el primero que utiliza un cifrado polialfabético. Esto quiere decir que a lo largo del mensaje se cambia de alfabeto, como ya hemos visto con los giros del disco móvil. Esta idea, será la explotada por los siguientes tipos de cifrado e incluso, por el cifrado que se analiza en este proyecto.

En esta época también tenemos el Manuscrito de Voynich, conocido como “el gran reto de los descifradores de códigos”. Aún no se ha podido descifrar[1].

En el siglo XVI, Francia vería nacer a otra de las figuras claves de la criptografía, Blaise de Vigenere,, que en 1586 publicaría el libro *Traité des chiffres où secrètes manières d'escrire[8]*, en el que expone su nuevo método de cifrado, que está basado en la cifra de César y utiliza ideas de Alberti.

La idea principal de este método es cifrar mediante el cifrado de César pero con un desplazamiento arbitrario[8], y no fijo como lo hacía el propio César. Es decir, cifrar la primera letra a desplazamiento 3, la segunda a desplazamiento 7… Este método resiste al análisis de frecuencias, pues cada letra se codifica de muchas formas distintas. Pero claro, si cambiamos arbitrariamente la cifra César, ni nosotros mismos vamos a ser capaces de descifrarla. Para ello, Vigenère utiliza el concepto de palabra clave.

Una palabra clave, por ejemplo, podría ser VIGENERE. Lo que queremos conseguir con esto es tener este desplazamiento arbitrario mediante una clave. Si nos fijamos, podemos cifrar la primera letra de un texto con un alfabeto César que empiece por "V", la segunda por "I", la tercera por "G", y así hasta llegar a la octava en nuestro caso y volver a empezar por la "V". Por lo tanto, cuanto más larga sea la clave, más alfabetos utilizamos y por lo tanto, podemos variar mucho el resultado del criptograma.

Vamos a ver un ejemplo sencillo de cómo sería poner esta explicación en práctica. Para ello, nos ayudaremos de una tabula recta, Figura 12. Es una tabla que se le da origen a Tritemius en su obra *Polygraphia*. Con esta, podemos ver cualquier tipo de alfabeto a cualquier desplazamiento. Podemos leerla tanto horizontal como verticalmente, el resultado no varia.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P
R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q
S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S
U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T
V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U
W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X
Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y

Figura 12. Ejemplo de Tabula Recta[11]

Imaginemos que queremos cifrar el mensaje: lacifradevigenere con la clave que mencionamos anteriormente, VIGENERE y que vamos a leer la tabla horizontalmente. Primero, buscamos en la primera fila la letra que queremos cifrar ("l"). Como es la primera letra, tal como dijimos, empezaremos a cifrar con un alfabeto que empiece por "V", por lo que iremos a la fila de la "V" en la primera columna y veremos que letra se entrecruza con la letra "l" de la primera fila. En este caso, la letra "G".

El siguiente paso sería buscar en la primera fila la siguiente letra: la "a" y buscar donde se cruza con la fila correspondiente al alfabeto que empiece por "l". Por lo tanto, la letra resultante sería: "I". Y así sucesivamente.

Sin embargo, en el siglo XIX, dos personajes lograron "romper" la cifra de Vigenère. Uno de ellos fue Charles Babbage y el otro el militar prusiano, Kasiki. Estos, se dieron cuenta de que realmente sí que se podía aplicar el análisis de frecuencias para desencriptar el mensaje. Bastaba con darse cuenta de que si se buscaban repeticiones de patrones y el cifrado era lo suficientemente largo, se podía intuir la longitud de la clave. Si por ejemplo, la clave, como en el caso anterior que hemos detallado, tiene 8 caracteres, resultará que las letras que ocupan las posiciones 1,9,17… se habrán cifrado con el mismo alfabeto (recordemos que mencionamos que cuando se llegaba a la última letra de la clave, volvíamos a la primera).

2. Máquina Enigma

Tal y como hemos visto y comentado, durante la historia de la humanidad, el ser humano ha necesitado encriptar los mensajes para que fuesen inteligibles para el enemigo. En el año 1918, un ingeniero alemán llamado Arthur Shcerbius[12], junto con su empresa "*The German firm Scherbius & Ritter*" patentó las ideas de una máquina de cifrado. Fue finalmente, en el año 1923 cuando se empezó a comerciar con el diseño final y se le dio el nombre de Enigma. Su uso principal estaba destinado al mercado y poco a poco fue cogiendo fuerza en el ámbito diplomático, hasta que finalmente, dio su gran paso y pasó a ser un elemento imprescindible en las comunicaciones militares, donde la modificaron para que esta alcanzase una gran complejidad.

2.1 Componentes

2.1.1 Teclado

Usado para escribir el texto el cual quería ser encriptado o desencriptado. Contenía 26 letras del alfabeto con una disposición QWERT[13]. Cuando cualquier letra es pulsada, se envía una señal eléctrica al siguiente componente.

2.1.2 Rueda de entrada/salida

La rueda de entrada/salida es el componente intermediario entre lo exterior a la máquina y lo interior. Se encarga de llevar la corriente de una letra que viene del teclado a los rotores y de llevar la respuesta de los rotores al panel de bombillas.

Está constituida por un cable grueso que contiene 26 cables en el interior, cada uno representando una letra. En la cara de la rueda, tenemos 26 pines planos metálicos, que son la terminación de cada cable, Figura 13.

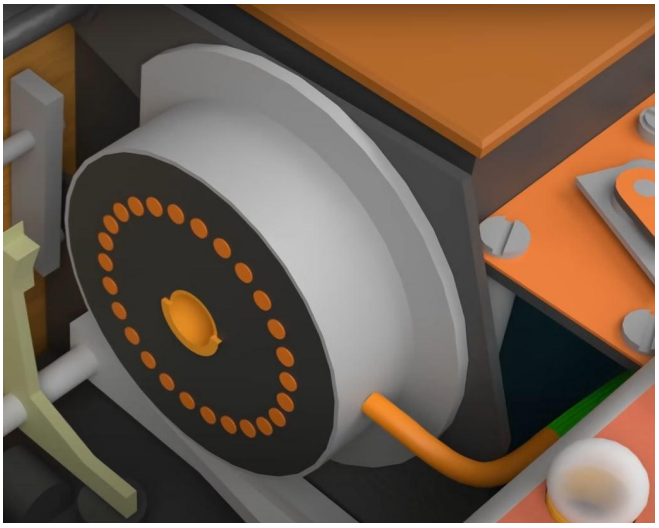


Figura 13. Modelado 3D de una rueda de entrada/salida[14]

2.1.3 Rotores

Los rotores son el corazón de la máquina Enigma. Son los encargados de transformar una letra de entrada en una de distinta como salida.

Para la primera máquina Enigma (1930), se fabricaron 3 rotores[14], con nombres: I, II y III (los mantuvieron a lo largo de la trayectoria de la máquina). En 1938, se fabricaron dos

unidades adicionales: IV y V. Y por último, en el año 1939, se fabricaron exclusivamente los rotores VI, VII y VIII para el ejército naval. Para saber con qué rotor se estaba trabajando, el número de este estaba en la cara de entrada, Figura 14.

Figura 14. Cara de entrada del rotor III[15]

En la Figura 15, se muestra un rotor visto por las dos caras, desmontado y con números para indicar las distintas partes de este. La cara de la izquierda hace referencia a la salida, y la de la izquierda a la de entrada (siempre tenemos la rueda 9 a la derecha de la tira alfabética 3).

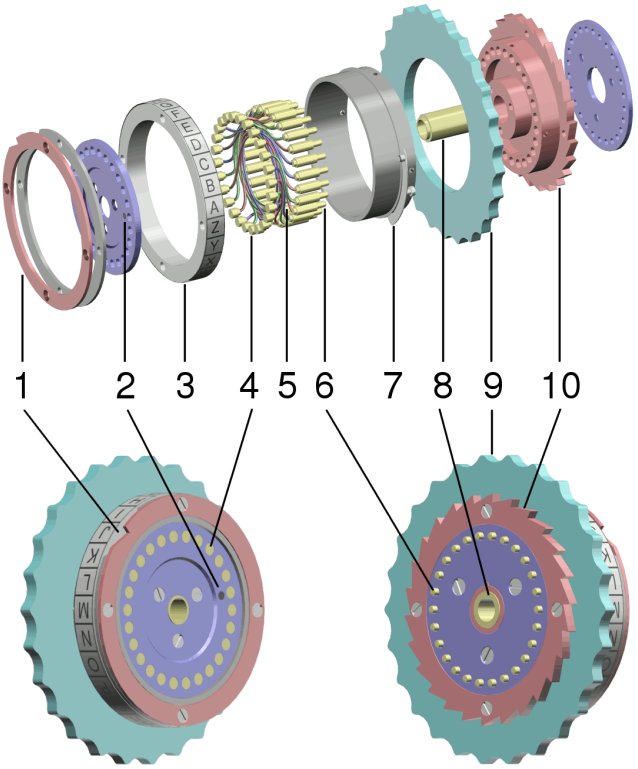


Figura 15. Composición completa de un rotor[14]

1- Anillo rojo que indica qué letra es la muesca, en el caso del rotor de la Figura 15, la letra "J".
2- Marca en forma de círculo rojo que nos indica, para cada rotor, cual es la letra "A" de este.
3- Anillo con el alfabeto de la "A" a la "Z".
4- Pin plano metálico de salida del rotor.
5- Cableado que conecta el pin metálico de entrada de un rotor con el pin plano metálico de entrada de este.
6- Pin metálico de entrada de un rotor.
7- Muesca que sirve para cambiar la relación que hay entre la entrada que le llega a un rotor y la entrada real de este. Se introdujo más adelante, no lo cubrimos en este proyecto. Por este motivo es importante el elemento 2, para ser conocedor de cual es la letra "A" del rotor.
8- Tubo que sujeta los rotores.
9- Rueda utilizada para dar posición inicial a los rotores por parte de los operarios.
10- Rueda en forma de sierra que gracias a una palanca (no perteneciente a los rotores), hace que estos giren.

El conexionado que se hace mediante cable que acabamos de mencionar (elemento 5), es fijo para cada rotor, era conocido por todos en la armada alemana y acabó siendo conocido también por los ingleses. Cómo lo obtuvieron se desconoce. En la Figura 16 se ve la salida que producía cada rotor con la entrada de este.

Rotor #	ABCDEFGHIJKLMNOPQRSTUVWXYZ
I	EKMFLGDQVZNTOWYHXUSPAIBRCJ
II	AJDKSIRUXBLHWTMCQGZNPYFVOE
III	BDFHJLCPRTXVZNYEIWGAKMUSQO
IV	ESOVPZJAYQUIRHXLNFTGKDCMWB
V	VZBRGITYUPSDNHLXAWMJQOFECK
VI	JPGVOUMFYQBENHZRDKASXLICTW
VII	NZJHGRCXMYSWBOUFAIVLPEKQDT
VIII	FKQHTLXOCBJSPDZRAMEWNIUYGV

Figura 16. Interconexionado de cada rotor[14]

Por ejemplo, para el rotor II, si tenemos una "A" de entrada, este seguirá sacando una "A" de salida. Si entra una "B", sacará una "J". Si entra una "C", una "D", y así sucesivamente.

Hemos mencionado el término “muesca” en el elemento 1 de este mismo apartado. La muesca indica que llegado el rotor a esa posición, el siguiente rotor debe de hacer un giro. Estas muescas, también eran fijas para cada rotor y también conocidas, Figura 17.

Rotor	Notch
I	Q
II	E
III	V
IV	J
V	Z
VI, VII, VIII	Z+M

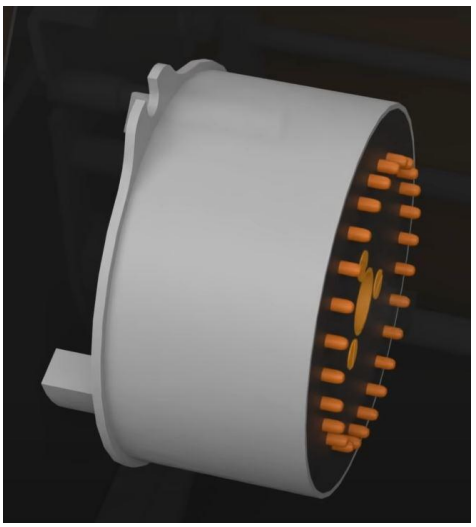
Figura 17. Muescas de cada rotor[14]

Como vemos, los rotores VI, VII y VIII se les introdujo una segunda muesca, aumentando la complejidad considerablemente, ya que los rotores giraban con frecuencia x2, y por lo tanto, era más complicado imaginarse el recorrido de estos.

2.1.4 Reflector

El reflector es el componente encargado de crear un camino de vuelta a la señal eléctrica que le viene del rotor que está conectado a él. Con esto, conseguimos que una letra de entrada nunca de la misma como salida, ya que el recorrido que va a llevar la corriente de vuelta va a ser distinto que el de ida.

Es una rueda constituida por 26 pines metálicos, tal y como se ve en la Figura 18, que toman contacto con los del rotor más cercano. Cada pin del reflector, para crear un camino de vuelta distinto, está interconectado mediante cables con otro pin, Figura 19.



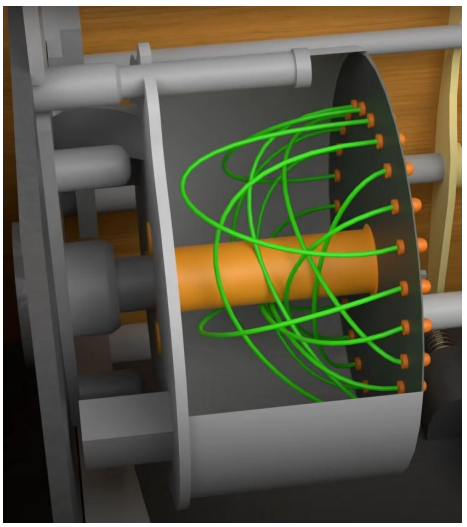


Figura 18. Modelado 3D del reflector[16] Figura 19. Modelado 3D del interconexionado del reflector[16]

Existían dos reflectores distintos, llamados reflector B y reflector C, cada uno con conexiones distintas. Al igual que los rotores, estas conexiones eran conocidas y siempre las mismas[17], Figura 20.

reflector B	(AY) (BR) (CU) (DH) (EQ) (FS) (GL) (IP) (JX) (KN) (MO) (TZ) (VW)
reflector C	(AF) (BV) (CP) (DJ) (EI) (GO) (HY) (KR) (LZ) (MX) (NW) (TQ) (SU)

Figura 20. Interconexionado de los dos reflectores[17]

2.1.5 Panel de bombillas

Consistía en un total de 26 bombillas, con la misma distribución del *layout* del teclado (QWER) y que cada una de ellas tenía impresa una letra. La bombilla iluminada era la letra encriptada/desencriptada que se había pulsado en el teclado.

2.1.6 *Plugboard*

Fue el elemento innovador creado por la milicia nazi para aumentarle el grado de complejidad que tenía la máquina enormemente[12], luego veremos esto.

Consistía en un panel con dos agujeros verticales por carácter del abecedario, por lo tanto, 52 agujeros totales, tal y como se muestra en la Figura 21. Uno de los agujeros servía como entrada a la máquina, y el otro como salida. Cuando la corriente circulaba por el agujero de entrada, significaba que esa letra era la que se había presionado en el teclado.

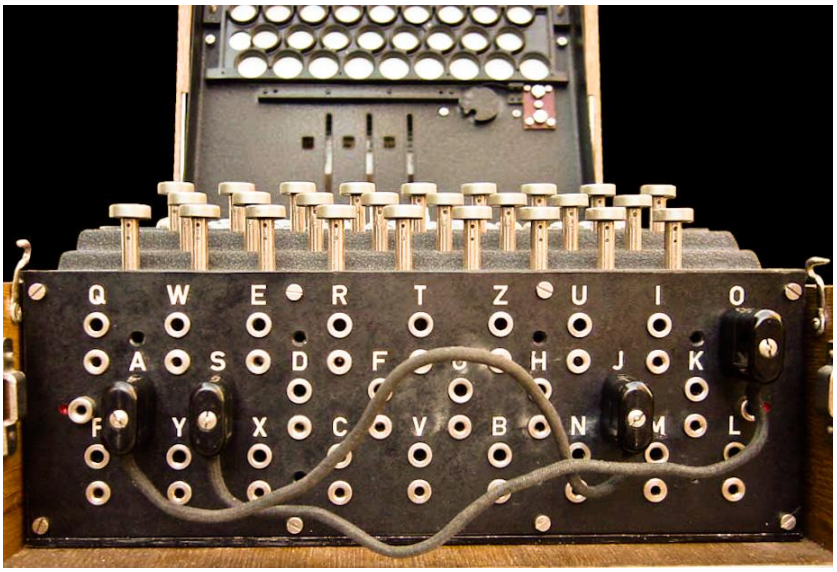


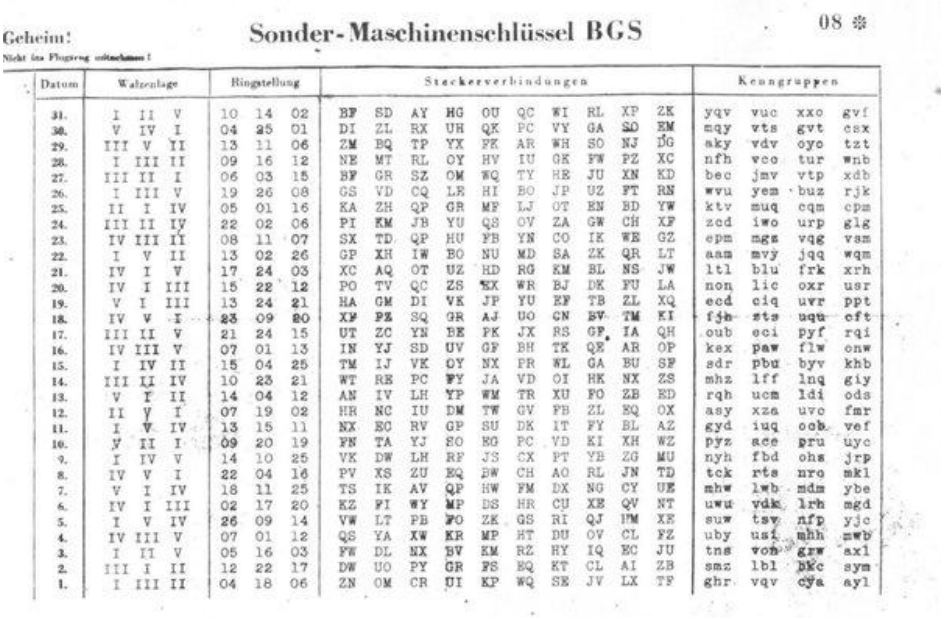
Figura 21. Vista frontal (*plugboard)* de una máquina Enigma[12]

Para ello, se utilizaban unos cables simétricos (ambas terminaciones eran iguales), cada una de ellas con dos pines que servían para conectar dos letras, los dos agujeros de la entrada y los dos agujeros de la salida.

Un ejemplo lo tenemos también en la Figura 13, donde un cable conecta la letra A con la J y viceversa. El pin de entrada de la terminación "A", será conectado con el pin de entrada de la terminación "J", haciendo que si se pulsa la tecla "A", el corriente entrará a la máquina como si se hubiera presionado la tecla "J" y viceversa. Por otro lado, el pin de salida de la terminación "A", será conectado con el pin de salida de la terminación "J", y el comportamiento será el mismo que el descrito con la entrada.

2.2 Funcionamiento

Para explicar un ejemplo de iteración de la Enigma, vamos a proponer una configuración inicial y vamos a ir intercalando la explicación electromecánica con la ayuda de un simulador[13] de la máquina que podemos encontrar por Internet. En aquella época, los operarios tenían una hoja con todas las configuraciones de la Enigma para todo el mes, Figura 22.



Geheim! Sonder-Maschinenschlüssel BGS 08 ❄

Nicht ins Flugzeug mitnehmen!

Datum	Walzenlage	Ringstellung	Steckerverbindungen	Kenngruppen
31.	I II V	10 14 02	BF SD AY HG OU QC WI RL XP ZK	yqv vuc xxo gvf
30.	V IV I	04 25 01	DI ZL RX UH QK PC VY GA SD EM	mqy vts gvt csx
29.	III V II	13 11 06	ZM BQ TP YX FK AR WH SO NJ IG	aky vdv oyo tzt
28.	I III II	09 16 12	NE MT RL OY HV IU GK FW PZ XC	nfh vco tur wnb
27.	III II I	06 03 15	BF GR SZ OM WQ TY HE JU XN KD	bec jmv vtp xdb
26.	I III V	19 26 08	GS VD CQ LR HI BO JP UZ FT RN	wvu yem buz rjk
25.	II I IV	05 01 16	KA ZH QP GR MF LJ OT EN BD YW	ktv muq cqm cpm
24.	III II IV	22 02 06	PI KM JB YU QS OV ZA GW CH XF	zcd iwo urp glg
23.	IV III II	08 11 07	SX TD QP HU FB YN CO IK WE GZ	epm mgs vqg vsm
22.	I V II	13 02 26	GP XH IW BO NU MD SA ZK QR LT	aam mvy jqq wqm
21.	IV I V	17 24 03	XC AQ OT UZ HD RG KM BL NS JW	ltl blu frk xrh
20.	IV I III	15 22 12	PO TV QC ZS SX WR BJ DK FU LA	non lic oxr usr
19.	V I III	13 24 21	HA GM DI VK JP YU EF TB ZL XQ	ecd ciq uvr ppt
18.	IV V I	23 09 20	XF PZ SQ GR AJ UO CN BV TM KI	fjh sts uqu cft
17.	III II V	21 24 15	UT ZC YN BE PK JX RS GF IA QH	oub eci pyf rqi
16.	IV III V	07 01 13	IN YJ SD UV GP BH TK QE AR OP	kex paw flw onw
15.	I IV II	15 04 25	TM IJ VK OY NX PR WL GA BU SF	sdr pbu byv khb
14.	III II IV	10 23 21	WT RE PC FY JA VD OI HK NX ZS	mhz lff lnq giy
13.	V I II	14 04 12	AN IV LH YP WM TR XU FO ZB ED	rqh ucm ldi ods
12.	II V I	07 19 02	HR NC IU DM TW GV FB ZL EQ OX	asy xza uvo fmr
11.	I V IV	13 15 11	NX EO RV GP SU DK IT FY BL AZ	gyd iuq oob vef
10.	V II I	09 20 19	FN TA YJ SO EG PC VD KI XH WZ	pyz ace pru uyc
9.	I IV V	14 10 25	VK DW LH RF JS CX PT YB ZG MU	nyh fbd ohs jrp
8.	IV V I	22 04 16	PV XS ZU EQ BW CH AO RL JN TD	tck rts nro mkl
7.	V I IV	18 11 25	TS IK AV QF HW FM DX NG CY UE	mhw lwb mdm ybe
6.	IV I III	02 17 20	KZ FI WY MP DS HR CU XE QV NT	uwu vdk lrh mgd
5.	I V IV	26 09 14	VW LT PB FO ZK GS RI QJ IM XE	suw tsy nfp yjc
4.	IV III V	07 01 12	QS YA XW KR MP HT DU OV CL FZ	uby usi mhh mwb
3.	I II V	05 16 03	FW DL NX BV KM RZ HY IQ EC JU	tns von grw axl
2.	III I II	12 22 17	DW UO PY GR FS EQ KT CL AI ZB	smz lbl bkc sym
1.	I III II	04 18 06	ZN OM CR UI KP WQ SE JV LX TF	ghr vqv cya ayl

Figura 22. Ejemplo de las configuraciones iniciales de todo un mes[18]

Para nuestro caso, el rotor lento será el IV, el del medio el II y el rápido el I. La posición inicial del rotor lento será la “B”, la del medio la “T” y la del rápido la “H”. El reflector será el B. En la Figura 23 podemos observar esta configuración. No conectamos ninguna pareja del *plugboard*. La líneas rojas hacen referencia a la letra que es la muesca en cada rotor.

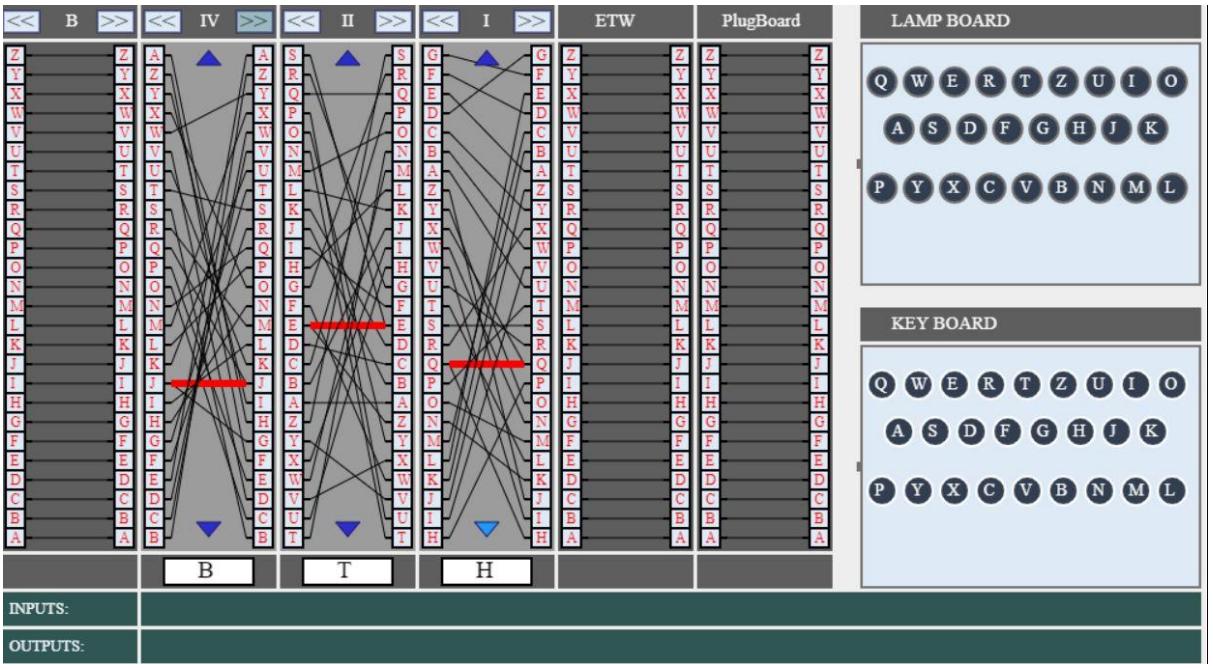


Figura 23. Estado inicial de la Enigma mediante un simulador

Como hemos comentado, la Enigma era un dispositivo electromecánico. Se empezaba pulsando en el teclado la letra que quisiéramos encriptar/desencriptar (la “R” en nuestro ejemplo) y acto seguido, había un giro en los rotores (solo se mueve el rotor rápido, ya que ninguno está en su muesca), Figura 24.

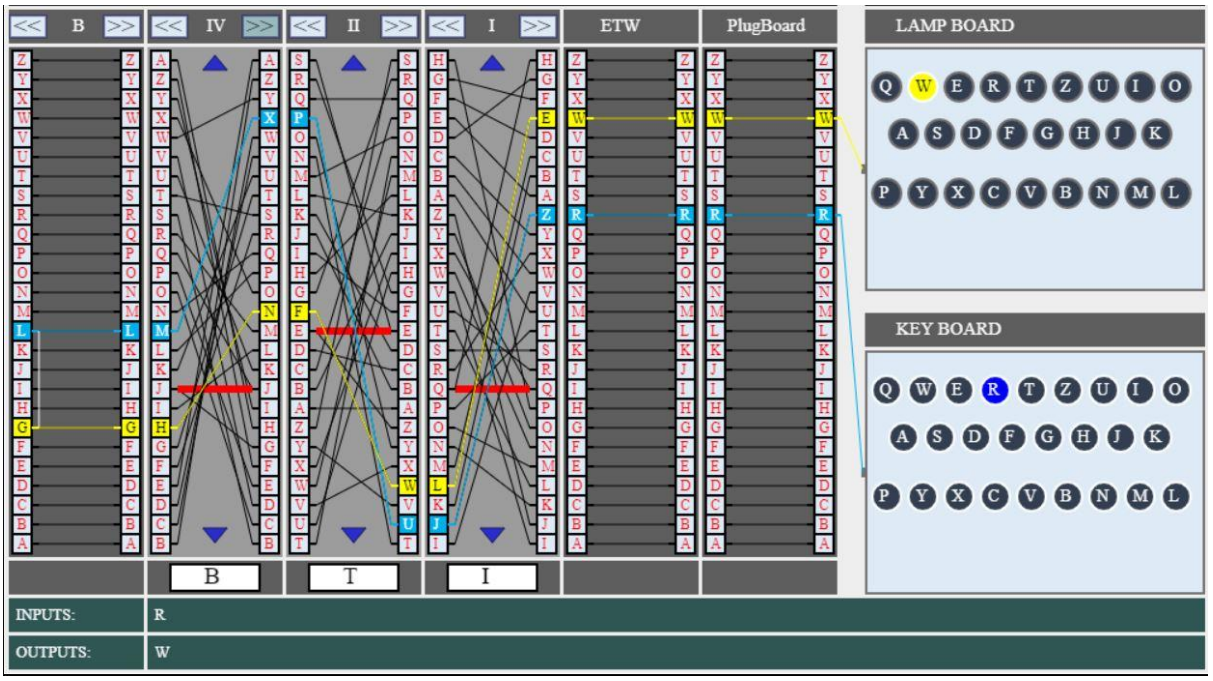


Figura 24. Estado de la primera iteración de la Enigma mediante un simulador

La señal eléctrica que se generaba (línea azul) desde el teclado viajaba al *plugboard.* Si no había cable en el *plugboard* la señal seguía el cable de la letra correspondiente con el teclado (como se ve en el ejemplo, la “R” será “R”). Si por el contrario, si había cable creando una pareja, la señal iba al nuevo clave.

Esta señal llegaba a la rueda de entrada/salida (referenciada como “ETW” en el ejemplo) que hacía contacto con el primer rotor que tenía pegado a ella (el I). De los 26 pines metálicos que tenía esta rueda, solo tenía señal el que venía del cable del *plugboard.* Por lo tanto, este pin (en nuestro ejemplo es nuestra “R”) hacía contacto con el pin del rotor rápido (la “Z”). Si nos fijamos en la Figura 24, vemos que era la letra “Y”, pero hubo un giro en este rotor nada más presionar el teclado.

Mediante el interconexionado del rotor, la señal viajará hasta el otro extremo donde habrá un pin metálico que estará tocando con otro pin del siguiente rotor (mismo procedimiento que acabamos de ver).

Al llegar al último rotor, este tendrá contacto con un pin del reflector, en nuestro ejemplo, la letra “L” de este. Como vimos, el reflector lo que hacía era generar un camino de vuelta

(línea amarilla en nuestro ejemplo) único, por lo que cada letra que le entraba, tenía su única conexión con una de salida. En nuestro caso, al ser la "L", esta letra de salida es la "G".

La señal viajará por los rotores hasta llegar a la rueda de entrada/salida. Esta llevará la señal al *plugboard* y depende de si la letra está conectada o no, se iluminará una bombilla u otra. En nuestro caso, le llega al *plugboard* una "W" y como no tiene pareja, esta es la letra que se iluminará en las bombillas (podemos verlo tanto en la zona llamada "LAMPBOARD" como en "OUTPUTS".

2.2.1 Giro de los rotores

Cada rotor tiene asociada una palanca que a cada pulsación del teclado esta sube. Para que un rotor gire, esta palanca tiene que poder empujar uno de los dientes de sierra de dicho rotor. Si nos fijamos en la Figura 15 del apartado "**2.1.3 Rotores**", esta sierra es el elemento 10. La palanca del rotor rápido siempre estará colocada de tal forma que siempre que suba pueda empujar un diente del rotor, y por lo tanto, hacerlo mover.

Las palancas de los siguientes rotores nunca pueden empezar un diente de sierra de su rotor correspondiente, debido a que se lo impide el elemento 1 del anterior rotor (su muesca). El disco de esta muesca está más elevado que los dientes de sierra. Una vez un rotor llega a su muesca, esta hará que la palanca del siguiente rotor baje de nivel y pueda empujar un diente de sierra de su rotor.

Para una explicación mucho más detallada y visual, recomiendo ver el vídeo[16], de donde ya hemos ido sacando imágenes de algún componente de la Enigma. En concreto, la representación de los giros de los rotores.

2.2.1.1 Doble giro de rotores

Un dato a tener en cuenta, y que si no se entiende el funcionamiento del apartado anterior "**2.2.1 Giro de los rotores**" no es lógico, es de que hay ocasiones en que el rotor lento provoca un giro en el mismo y en el rotor anterior (el del medio). De hecho, eso pasa de la misma forma con el rotor del medio, haciendo que gire el rápido también, pero como el rápido gira en cada pulsación de teclado, este efecto queda oculto.

Cuando el rotor del medio llega a su muesca, tal como explicamos en el apartado anterior, dicha muesca permite a la palanca del rotor lento poder empujar un diente de su sierra para hacerlo girar. Sin embargo, la palanca también queda nivelada con la muesca del rotor del medio, empujándola y como esta, está sujeta al rotor del medio, este también se mueve.

Un ejemplo visual de este doble giro del rotor del medio puede verse en el vídeo[19], el cual es una representación de la parte mecánica de los rotores de la Enigma hecha de madera.

2.3 Código secuencial

Con el código secuencial hemos tratado de simular lo más fielmente el funcionamiento de la máquina Enigma visto previamente.

Como hemos visto anteriormente, el conexionado interno de cada rotor era conocido. Para lograr esto, hemos creado una matriz con 8 vectores de 26 elementos cada uno, Figura 24. Los 8 vectores simulan los 8 rotores que se llegaron a crear, aunque en este proyecto sólo trabajamos con 5, y los 26 carácteres simulan todas las letras del abecedario. Esta matriz solo nos sirve como el camino de ida.

```
char rotor[8][26]={
        /* Input   "ABCDEFGHIJKLMNOPQRSTUVWXYZ" */
        /* I: */    "EKMFLGDQVZNTOWYHXUSPAIBRCJ",
        /* II: */   "AJDKSIRUXBLHWTMCQGZNPYFVOE",
        /* III: */ "BDFHJLCPRTXVZNYEIWGAKMUSQO",
        /* IV: */   "ESOVPZJAYQUIRHXLNFTGKDCMWB",
        /* V: */    "VZBRGITYUPSDNHLXAWMJQOFECK" };
```

Figura 24. Código del interconexionado de ida de los rotores

Lo que se quiere conseguir es que para cada posición de cada vector, simulando una letra de entrada, genere una letra de salida. Por ejemplo, si utilizamos el rotor II y la letra de entrada es la “C”, se está accediendo al elemento rotor[1][2], y por lo tanto, la salida es “D”. Recordemos que “A” es la posición 0 y “Z” la 25, como en código ASCII.

Para el camino de vuelta, hacemos una matriz de las mismas dimensiones pero cada elemento será el inverso que vimos para el camino de ida, Figura 25. Por ejemplo, si utilizamos el rotor II y la letra de entrada es la “D”, se está accediendo al elemento rotorV[1][3], la salida es “C” y por lo tanto, lo inverso del ejemplo anterior.

```
char rotorV[8][26]={
        /* Input   "ABCDEFGHIJKLMNOPQRSTUVWXYZ" */
        /* I: */    "UWYGADFPVZBECKMTHXSLRINQOJ",
        /* II: */   "AJPCZWRLFBDKOTYUQGENHXMIVS",
        /* III: */  "TAGBPCSDQEUFVNZHYIXJWLRKOM",
        /* IV: */   "HZWVARTNLGUPXQCEJMBSKDYOIF",
        /* V: */    "QCYLXWENFTZOSMVJUDKGIARPHB"};
```

Figura 25. Código del interconexionado de vuelta de los rotores

Es importante también tener guardado el reflector, ya que como vimos, también era conocido, Figura 26. Es una matriz de dos vectores, ya que solo existían dos reflectores, con 26 elementos por vector; simulando las 26 letras del abecedario. El comportamiento es idéntico a lo visto con los rotores, le llega una entrada mapeada a una posición del vector correspondiente y el elemento de esa entrada es la letra de salida del reflector.

```
char Reflector[2][26]={
        /* Input   "ABCDEFGHIJKLMNOPQRSTUVWXYZ" */
        /* B: */    "YRUHQSLDPXNGOKMIEBFZCWVJAT",   // (AY) (BR) (CU) (DH) (EQ) (FS) (GL) (IP) (JX) (KN) (MO) (TZ) (VW)
        /* C: */    "FVPJIAOYEDRZXWGCTKUQSBNMHL"};  // (AF) (BV) (CP) (DJ) (EI) (GO) (HY) (KR) (LZ) (MX) (NW) (TQ) (SU)
```

Figura 26. Código del interconexionado de los dos reflectores

Por último, en cuanto a datos conocidos, tenemos las muescas de cada rotor guardadas en un vector, Figura 27. El mapeado en cuanto a rotor y posición es el mismo que acabamos de ver, siendo la posición 0 para el rotor I y la posición 4 para el rotor V.

```
char Muescas[5]="QEVJZ";
```

Figura 27. Código de las muescas de cada rotor

En este punto, necesitamos meternos en el papel del operario que configuraba la máquina para que trabaje con esa configuración todo el día. Para ello, hemos creado una estructura de datos donde guarda toda la información necesaria para poner en marcha la máquina, Figura 28.

```
typedef struct {
   int rF;   // rotor Rapido
   int rM;   // rotor Central
   int rS;   // rotor Lento
   int refl; // reflector
   char pF;  // posicion inicial rotor Rapido
   char pM;  // posicion inicial rotor Central
   char pS;  // posicion inicial rotos Lento
   char Plg[30]; // Plugboard

} EnigmaSet;
```

Figura 28. Datos de la configuración inicial de la Enigma

Una vez tenemos la configuración inicial, esta es pasada a una serie de vectores con los que el código trabajará a lo largo de la ejecución de este. Estos vectores son:

- Para simular los rotores colocados en ese momento, tenemos unos vectores de 26 carácteres: rFast (rotor de ida rápido), rVFast (rotor de vuelta rápido), rMiddle (rotor de ida del medio), rVMiddle (rotor de vuelta del medio), rSlow (rotor de ida lento) y rVSlow (rotor de vuelta lento). Estos son obtenidos mediante las matrices rotor y rotorV.
- Para simular el reflector se utiliza un vector de 26 caracteres llamado Volta. Se obtiene con el vector correspondiente de la matriz Reflector.
- Por último, el *plugboard* es simulado con un vector de 26 carácteres que se obtiene de la estructura de datos de la configuración inicial; de campo Plg.

Ahora ya solo queda que la Enigma itere para cada entrada que le llega. Lo más importante del cuerpo de esta es, y para hacerlo lo más fiel posible, hacer por lo menos el giro del rotor rápido antes de empezar a calcular la salida (ya que este siempre lo hace). Por lo tanto, tenemos que tener las muescas de los rotores siempre presentes antes de iterar, tal y como se muestra en la Figura 29.

```
if (pMiddle == mMiddle) {
  pFast = (pFast + 1) % 26;
  pMiddle = (pMiddle + 1) % 26;
  pSlow = (pSlow + 1) % 26;
} else if (pFast == mFast) {
  pFast = (pFast + 1) % 26;
  pMiddle = (pMiddle + 1) % 26;
} else
  pFast = (pFast + 1) % 26;
```

Figura 29. Comprobación de giro de cada rotor

Como vemos, se le da prioridad a mirar la condición de la muesca del rotor del medio. Esto es, porque si la posición actual del rotor del medio es la misma que su muesca, esto hace que gire este rotor y además, el rotor lento. Si por el contrario, esto no sucede, y la posición actual del rotor rápido es igual a su muesca, hará girar solamente al rotor del medio. Recordemos que pase lo que pase, el rotor rápido siempre gira. Esto lo vimos explicado mecánicamente en el apartado "**2.2.1.1 Doble giro en el rotor del medio**".

A partir de aquí, le entra una letra y se simula el comportamiento de la enigma mediante los vectores ya vistos (por orden de acceso): Plug, rFast, rMiddle, rSlow, Volta, rVSlow, rVMiddle, rVFast y por último, Plug otra vez, dándonos la salida correspondiente.

2.3.1 Resultados de ejecución

2.3.1.1 Fichero a encriptar/desencriptar

Para las ejecuciones de este proyecto vamos a usar ficheros de texto que corresponden con libros completos de distintos autores. Es necesario que estos textos contengan un formato en particular para que lo que se va a ver más adelante en este proyecto funcione y cobre sentido; empezarán todos con la palabra "TITLE" seguido de su título. Después vendrá la palabra "AUTHOR" seguido del nombre y apellido del autor (por este orden).

En particular, vamos a trabajar con un fichero de texto de 235,974 bytes. Este es el libro "Las Indias Negras" de Julio Verne.

2.3.1.3 Resultados de ejecución

En la Tabla 1 podemos ver el resultado de los tiempos de todo el programa y del trozo de código que hace referencia al cálculo de la encriptación/desencriptación del texto. Se han recogido los resultados de 100 ejecuciones del código en el orden de milisegundos.

		Total	Cálculo
Enigma secuencial	Media	927.87 ms	927.35 ms

Tabla 1. Tiempos total y de cálculo de la Enigma secuencial

Como podemos ver, el tiempo del programa que hace referencia al cálculo de la encriptación/desencriptación es del 99.94% del programa, por lo que nos interesa trabajar sobre esta parte y ver si es paralelizable.

2.4 Código optimizado

La idea principal de este código es la de prepararlo para la ejecución en cuda. Podríamos transformar el código secuencial visto en el apartado "**2.3 Código secuencial**" directamente a cuda, pero haciendo una serie de optimizaciones, vamos a mejorar el resultado de este. Estas optimizaciones van a ser las que vamos a ver en este apartado: precálculo de la matriz Encriptador, precálculo de las matrices Camino y Ruta y el acceso a dichas matrices.

2.4.1 Matriz Encriptador

Como sabemos que los tres rotores van a ser los mismos, y en el mismo orden durante toda la ejecución de la enigma, nos vamos a aprovechar de esto haciendo un precálculo. Este precálculo se basa en tener almacenados, para cualquier entrada y cualquier posición actual de la letra de cada rotor, la salida de la enigma.

Para conseguir esto, vamos a tener una matriz de 26x26x26x26, que la vamos a llamar Encriptador. Estas dimensiones son debidas a que tenemos 26 posiciones posibles para el rotor lento, 26 posiciones posibles para el del medio, 26 posiciones posibles para el rápido, y finalmente, 26 letras posibles de entrada para cada posición. Por lo tanto, sean cuales sean las dimensiones del texto que queremos encriptar/desencriptar, la máquina enigma va a tener que iterar 456,976 veces.

Esto último es muy importante, ya que es muy probable de que ejecutemos más veces la máquina enigma de lo necesario solo para tener guardadas todas las combinaciones. Sin embargo, y tal como he mencionado, el código sirve para explotar la paralelización con cuda. Veremos los resultados más adelante.

Como vamos a tener que acceder a la matriz Encriptador de forma no secuencial, vamos a tener también que haber precalculado el recorrido que va a llevar la enigma a partir del orden de los rotores y su posición inicial. Para ello, tenemos que identificar primero los 3 casos iniciales que existen en la Enigma (los llamamos A, B y C, Figura 30). Esto solo se ejecuta una vez, al principio junto a la configuración inicial.

```
if (pFast == ((mFast+1)%26) && pMiddle == ((mMiddle+1)%26))
  Caso = 'B';
else if (pFast == ((mFast+1)%26) && pMiddle == mMiddle)
  Caso = 'A';
else if (mFast == pFast && mMiddle == pMiddle)
  Caso = 'C';
else if (mMiddle == pMiddle)
  Caso = 'B';
else
  Caso = 'A';
```

Figura 30. Casos iniciales de la Enigma

Estos tres casos sirven para identificar cómo se comporta la enigma en los primeros giros de los rotores, por eso se tienen en cuenta las muescas del rotor rápido y del rotor del medio. A continuación, vamos a analizar los tres casos para así saber, el número de posiciones que se recorre en cada uno de ellos y así saber de cuanto debe de ser nuestra estructura de datos que guarde este recorrido.

2.4.1.1 Caso A

Para el caso A, vamos a seleccionar los siguientes rotores: I (rápido), II (medio), III (lento). Además, el rotor I va a tener la letra “Q” como posición inicial (su muesca), el rotor II la letra “F” como posición inicial (siguiente a su muesca) y el rotor III la letra “A” como posición inicial.

La siguiente posición a la posición inicial AFQ es AGR, ya que al ser “Q” la muesca del rotor I, hace que gire el rotor II a la siguiente posición. A partir de aquí, el rotor II va a girar una posición cada 26 posiciones del rotor I hasta llegar a la posición AER, donde “E” es la muesca del rotor II. Hasta aquí, llevamos 1 + 1 + 26 x 24 (626) posiciones recorridas.

La siguiente posición a AER es BFS, ya que al ser “E” la muesca del rotor II, hace que gire este y el rotor III. Deberán pasar 25 posiciones para llegar a la posición BGR. Una vez aquí, 26 posiciones para llegar a la posición BHR y finalmente, se repetirá esto último hasta llegar a la posición BER. Nótese el bucle aquí, ya que ahora estamos en BER y antes en AER, por lo que se repetirá el patrón. Por lo tanto, hasta llegar a ZER recorreremos: (1 + 25 + 24 x 26) x 25 (16,250) posiciones.

La siguiente posición a ZER es AFS. Nosotros ya visitamos al principio la posición AFQ, pero no las demás que hay entre AFS y esta última. Por lo tanto, tendremos que recorrer 24 posiciones más.

En total, habremos recorrido 626 + 16,250 + 24 (16,900) posiciones para llegar a la inicial.

2.4.1.2 Caso B

El caso B va a tener la misma configuración de rotores con la diferencia, de que el rotor I va a tener la letra “R” como posición inicial (siguiente a su muesca), el rotor II va a tener la letra “F” como posición inicial (siguiente a su muesca) y el rotor III va a tener la letra “A” como posición inicial.

La siguiente posición no va a producir ningún giro del rotor II, por lo que no será hasta las 26 posiciones siguientes cuando suceda; AGR. Esto va a repetirse hasta que lleguemos a la posición AER, donde “E” es la muesca del rotor II. Por lo tanto, hasta aquí llevaremos 1 + 25 x 26 (651) posiciones.

La siguiente posición a AER es BFS, ya que al ser “E” la muesca del rotor II, hace que gire este y el rotor III. Podemos notar que a partir de aquí, tenemos la misma situación que en el “Caso A”. Por lo tanto, para llegar a la posición ZER será lo mismo: 16,250 posiciones. Sin embargo, ya no necesitaremos recorrer más posiciones, dado que, a diferencia del “Caso A”, no hemos hecho un giro del rotor II en la primera posición y por lo tanto, no nos quedan posiciones que recorrer.

El total de posiciones recorridas para este caso es de 651 + 16,250 (16,901) hasta llegar a la posición inicial.

2.4.1.3 Caso C

Misma configuración de rotores salvo que el rotor I tendrá la letra “Q” como posición inicial (su muesca), el rotor II tendrá la letra “E” como posición inicial (su muesca) y el rotor II tendrá la letra “A” como posición inicial.

La siguiente posición a la posición inicial AEQ generará un giro de los tres rotores, debido a que la letra “E” es la muesca del rotor II, quedando en la posición BFR. Para el siguiente giro del rotor II, posición BGR, tendrán que pasar 26 posiciones y así sucesivamente hasta que el rotor III gire ya que se ha alcanzado la letra “E” en el rotor II (BER). Por lo tanto, hasta este punto habremos recorrido 1 + 1 + 25 x 26 (652 posiciones).

La siguiente posición a BER es CFS, ya que la letra “E” es la muesca del rotor II, por lo tanto, los tres rotores giran. Para llegar a CGR deberán pasar 25 posiciones. Una vez aquí, pasarán 26 posiciones hasta el próximo giro del rotor II, en la posición CHR y esto se repetirá hasta la

posición anterior al próximo giro del rotor III, CER. Y a partir de aquí, se repetirá el bucle al igual que teníamos con la posición BER. Por lo tanto, recorreremos (1 + 25 + 24 x 26) x 25 (16,250) posiciones.

En total, habremos recorrido 652 + 16,250 (16,902) posiciones hasta llegar a la posición inicial.

2.4.2 Matrices Camino y Ruta

Recapitulando, el Caso A ha recorrido 16,900 posiciones, el Caso B 16,901 y el Caso C 16,902.

Por lo tanto, vamos a crear una matriz Ruta con 16,900 vectores con 3 elementos cada uno. Esto simulará para cada una de las 16,900 posiciones en la ruta actual, qué posiciones tienen los tres rotores.

Crearemos también una matriz Camino, de 2 vectores con 3 elementos cada uno. Esto es, porque los casos B y C se comportan distinto al principio que el Caso A, debido a que empiezan con un giro en el rotor del medio. El Caso B llenará solo un vector, y el Caso C los 2.

2.4.3 Acceso a Encriptador

Si por ejemplo, se quisiera acceder a la iteración 123 de la ejecución de la enigma, bastaría mirar primero de todo el caso en el que estamos y luego acceder a la posición de la matriz Ruta correspondiente. Por ejemplo, si es el Caso C, para tener la posición del rotor I (PF), accederíamos a Ruta[121][0], para la posición del rotor II (PM), a Ruta[121][1] y para la posición del rotor III (PS), a Ruta[121][2]. Recordemos que accedemos al vector 121 de Ruta debido a que tenemos 2 extras almacenados en Camino ya que estamos en el Caso C.

Lo siguiente será obtener la letra de entrada que queremos encriptar/desencriptar y transformarla en un entero (alfb).

Por último, accederemos a la matriz Encriptador[PS][PM][PF][alfb] que nos dará el resultado previamente guardado de la encriptación/desencriptación.

2.4.4 Tiempos y análisis

Para los tiempos que se van a mostrar a continuación, se ha utilizado el mismo texto a encriptar/desencriptar, el mismo hardware y el mismo número de ejecuciones del programa que en el apartado "**2.3.1.3 Tiempos y análisis**". En la Tabla 2 pueden verse los resultados en el orden de milisegundos.

		Total Secuencial	Total Optimizado	Ruta y Camino	Encriptador	Acceso a Encriptador
Enigma	Media	927.87	903.05	0.104	17.083	885.38

Tabla 2. Tiempo total y desglosado de la Enigma paralela

Como vemos, obtenemos una pequeña mejora con este código respecto al código secuencial (un *speedup* del 1.03 de media). Esto es debido a que, a pesar de realizar cálculos extra para guardar resultados en las matrices Ruta, Camino y Encriptador, tanto a la matriz Ruta como a la matriz Camino vamos a accederlas en posiciones contiguas de memoria, por lo que vamos a ganar tiempo. Además, nos vamos a aprovechar de los resultados de la Enigma previamente calculados al acceder a la matriz Encriptador.

A la matriz Encriptador nunca vamos a acceder en posiciones contiguas de memoria ya que accedemos por posiciones de rotores, y para cada posición, tenemos 26 *inputs* únicos distintos de entrada (a-z).

2.5 Código cuda

Como dijimos en el apartado anterior, la idea de crear un código optimizado era para que podamos transformar en un código cuda y que cada thread trabaje la encriptación/desencriptación de una letra del texto. Para ello, era necesario que cada uno de estos threads, pudiera acceder a la posición con la que quería trabajar.

Este código solo puede paralelizar dos subapartados que hemos visto en el apartado "**2.4 Código optimizado**". Se puede paralelizar el subapartado "**2.4.1 Matriz Encriptador**", ya que como se calculan todas las enigmas posibles no hay dependencias, y el subapartado "**2.4.3 Acceso a Encriptador**", ya que solo accedemos a la matriz Encriptador de forma masiva. El subapartado "**2.4.2 Matrices Camino y Ruta**" no se puede paralelizar, ya que la ejecución de una ruta de los rotores es secuencial, y una posición actual depende de la anterior.

2.5.1 Entorno de trabajo

2.5.1.1 Fichero a encriptar/desencriptar

Mismo libro que en las ejecuciones para los apartados "**2.3 Código secuencial**" y **"2.4 Código optimizado**".

2.5.1.2 Hardware para la ejecución

La GPU con la que vamos a ejecutar nuestro programa en cuda es una Tesla K40c, con las características que se pueden observar en la Figura 31. Estas características van a ser importantes para así poder pensar en un mejor aprovechamiento del trabajo de los threads para la ejecución de nuestro código.

```
Device 1: "Tesla K40c"
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 11440 MBytes (11995578368 bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Max Clock rate:                            745 MHz (0.75 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  . . .
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
```

Figura 31. Características de una Tesla K40c

Sin embargo, nos faltan aún unos datos muy importantes que hacen referencia a las características que soporta el hardware de la GPU según de la familia que sea la GPU en cuestión. En nuestro caso, en la Figura 31, podemos ver que es la versión 3.5, y sus características son las mostradas en la Figura 32:

Figura 32. Características del hardware de la versión 3.5

2.5.2 Objetivos de una ejecución en cuda

Los 3 principales objetivos que hay detrás de conseguir unos buenos resultados con una ejecución en cuda es que, mediante el código consigamos:

- Una buena distribución del trabajo para los *threads*. Esto significa, que si el trabajo total a hacer lo permite, estén todos los threads de la GPU trabajando al mismo tiempo, sin dejar *threads* inactivos.
- No haya divergencia en el trabajo de los *threads*. Esto significa, que todos los threads ejecuten las mismas instrucciones (ausencia de sentencias *if*) pero con datos distintos.
- Los accesos a memoria sean a posiciones contiguas y alineadas. Así podemos conseguir que varias peticiones a memoria de distintos *threads* de un mismo *warp* sean realizadas como una única transacción.

2.5.3 Paralelizar matriz Encriptador

2.5.3.1 Envio de información al *device*

La enigma necesita de varios datos para poder iterar, los hemos visto anteriormente. Los vectores: Plug, rFast, rMiddle, RSlow, rVFast, rVMiddle, rVSlow y Volta. Estos vectores, hay que enviarlos al *device* para que pueda calcular, Figura 33. Se deberá enviar también donde almacenar los resultados del kernel, por lo que si queremos que sea la matriz Encriptador, deberemos de enviar 26^4 para almacenar (recordemos Encriptador[26][26][26][26]).

```
cudaMalloc((char **) &d_Plug, 26);
cudaMalloc((char **) &d_rFast, 26);
cudaMalloc((char **) &d_rMiddle, 26);
cudaMalloc((char **) &d_rSlow, 26);
cudaMalloc((char **) &d_rVFast, 26);
cudaMalloc((char **) &d_rVMiddle, 26);
cudaMalloc((char **) &d_rVSlow, 26);
cudaMalloc((char **) &d_Volta, 26);
cudaMalloc((char **) &output, 26*26*26*26);

cudaMemcpy(d_Plug, &Plug, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rFast, &rFast, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rMiddle, &rMiddle, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rSlow, &rSlow, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rVFast, &rVFast, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rVMiddle, &rVMiddle, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rVSlow, &rVSlow, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_Volta, &Volta, 26, cudaMemcpyHostToDevice);
```

Figura 33. Asignación de memoria y transferencia de datos entre GPU-CPU

2.5.3.2 Distribución del trabajo

De todas las combinaciones que hemos probado, las tres siguientes son las que mejores resultados nos han proporcionado. Para la explicación, nos basaremos en que la matriz Encriptador, tal como ya mencionamos, ocupa 26^4, es decir, 456,976 caracteres.

2.5.3.2.1 128 threads por bloque

Para saber el número de bloques que necesitamos que contengan 128 threads, hemos de hacer el siguiente cálculo: (456,976 + 127) / 128. Siempre sumamos un el número de threads menos uno para que no nos quedemos cortos en el número de bloques. El resultado de esta operación son 3,571 bloques de 128 threads.

Si observamos la Figura 32 en el apartado "**2.5.1.2 Hardware para la ejecución**", vemos que podemos utilizar como máximo 16 bloques por SM. Además, si nos fijamos en la Figura 31 de ese mismo apartado, veremos que el número máximo de threads por SM es de 2048. Por lo que, en este caso, podemos utilizar los 16 bloques por SM (16 x 128 = 2048 threads).

Si volvemos a fijarnos en la Figura 31, veremos que tenemos un total de 15 SMs, por lo que vamos a tener trabajando al mismo tiempo 240 bloques (16 x 15). Y por lo tanto, van a estar trabajando 30,720 threads paralelamente (240 x 128). Hemos conseguido utilizar el número

total de threads, ya que la GPU tenía 2048 threads por SM y 15 SMs en total (2048 x 15 = 30,720).

2.5.3.2.2 256 threads por bloque

El procedimiento aquí es idéntico al del apartado anterior. El número de bloques es 1,786 ((456,976 + 255) / 256) y por lo tanto, 8 bloques por SM (8 x 256 = 2048, que es el máximo de threads que se soporta en un SM).

Por lo tanto, si tenemos 15 SMs, tendremos 120 bloques de 256 *threads* trabajando al mismo tiempo, es decir, los 30,720 threads que también conseguimos antes (120 x 256).

2.5.3.2.3 512 threads por bloque

El número de bloques es (456,976 + 511) / 512, por lo que 893 bloques totales de 512 threads. Así que como máximo, vamos a poder utilizar 4 bloques por SM (4 x 512 = 2048 threads). Por lo que 60 bloques totales trabajando a la vez (4 x 15). Es decir, 30,720 threads trabajando paralelamente (60 x 512).

2.5.3.3 Kernel

Una vez en el kernel, cada thread debe saber que posición es la que calcula, Figura 34. Con la variable pos, obtenemos la posición del thread respecto a todo el *device*. Con esta, podemos hacer referencia a todos las posiciones que necesitamos con la misma nomenclatura que vimos en el apartado "**2.4.5 Acceso a Encriptador**" con la excepción de que aquí he llamado "elemento" a la letra que se quiere encriptar/desencriptar en lugar de "alfb", como mencioné previamente en dicho apartado.

```
int pos = blockIdx.x * blockDim.x + threadIdx.x;

int elemento = pos % 26;
int PF = (pos / 26) % 26;
int PM = (pos / n) % 26;
int PS = (pos / N) % 26;
```

Figura 34. Posición Encriptador mediante identificador del *thread* de un bloque

Por ejemplo, siendo "n" 26*26 y "N" 26*26*26, si "pos" es 1730, significa que vamos a estar calculando el elemento Encriptador[0][2][14][14], es decir, Encriptador[PS][PM][PF][elemento]. Cada iteración de la Enigma necesita estas variables para saber qué posición tiene cada rotor.

Antes de que el thread ejecute una iteración de la Enigma, se necesita comprobar que “pos” sea menor de los 26^4 elementos que tiene el vector (que simula ser la matriz Encriptador) que enviamos al *device*.

Por último, una vez iterada la Enigma y adquirido el resultado, se va a guardar en la posición “pos” del vector que se envió al *device*.

2.5.3.3.1 Resultados de ejecución

El tiempo de ejecución para el precálculo de la matriz Encriptador se muestra en la Tabla 3. El tiempo “Total” del precálculo de la matriz Encriptador es el tiempo que se obtiene del *kernel* + las transmisiones de información entre GPU y CPU de forma bidireccional + la puesta en punto de dichas funciones. Los tiempos son en ms y se han cogido la muestra de 100 ejecuciones del código.

			Optimizado	128 th/bl	256 th/bl	512 th/bl
Encriptador	Total	Media	17.08	0.551	0.539	0.561
	Kernel	Media	No aplica	0.126	0.125	0.126

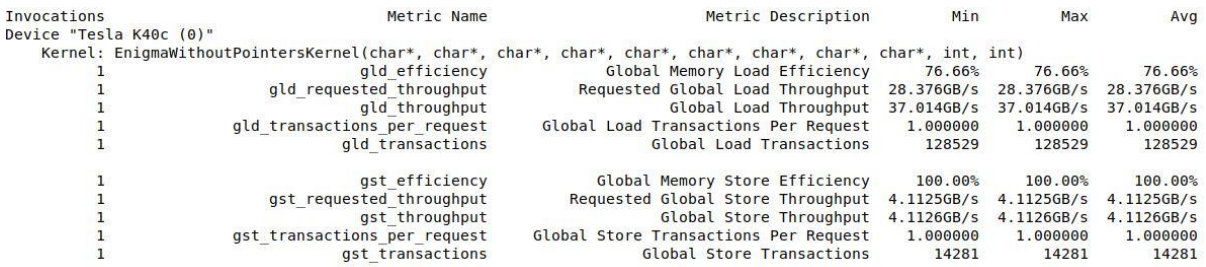
Tabla 3. Tiempos cálculo matriz Encriptador según distribución

Como vemos, conseguimos los siguientes *speedup*, Tabla 4, respecto al código optimizado:

		128 th//bl	256 th/bl	512 th/bl
Encriptador (Total)	*Speedup*	31.01	31.69	30.04

Tabla 4. *Speedup* respecto Enigma paralela según distribución

Si nos basamos en los tres puntos vistos en el apartado “**2.5.2 Objetivos de una ejecución en cuda**”, dos de ellos los cumplimos: hay buena distribución de los *threads* tal y como hemos visto en el apartado “**2.5.3.2 Distribución del trabajo**” y todos ellos ejecutan las mismas instrucciones. Sin embargo, el cómo se utiliza la memoria en este código no es la forma más óptima, y es precisamente esta parte la que penaliza el rendimiento de este programa. Para mostrar esto, vamos a sacar el uso de la memória para la ejecución de 256*threads*/bloque con el profiling de cuda “*nvprof*”, Figura 35. Escojo la distribución de 256 *threads*/bloque ya que es la que menos tiempo de ejecución total y de *kernel* tiene (aunque solo analizemos el *kernel*).



```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: EnigmaWithoutPointersKernel(char*, char*, char*, char*, char*, char*, char*, char*, int, int)
          1                            gld_efficiency             Global Memory Load Efficiency      76.66%      76.66%      76.66%
          1                  gld_requested_throughput          Requested Global Load Throughput  28.376GB/s  28.376GB/s  28.376GB/s
          1                            gld_throughput                    Global Load Throughput  37.014GB/s  37.014GB/s  37.014GB/s
          1              gld_transactions_per_request      Global Load Transactions Per Request    1.000000    1.000000    1.000000
          1                         gld_transactions                  Global Load Transactions      128529      128529      128529

          1                            gst_efficiency            Global Memory Store Efficiency     100.00%     100.00%     100.00%
          1                  gst_requested_throughput         Requested Global Store Throughput  4.1125GB/s  4.1125GB/s  4.1125GB/s
          1                            gst_throughput                    Global Store Throughput  4.1126GB/s  4.1126GB/s  4.1126GB/s
          1              gst_transactions_per_request     Global Store Transactions Per Request    1.000000    1.000000    1.000000
          1                         gst_transactions                  Global Store Transactions       14281       14281       14281
```

Figura 35. Nvprof con métricas de memoria para ejecución 256 th/bl

Si nos fijamos en el bloque de las lecturas, vemos que se nos indica que la eficiencia de las lecturas es de un 76.66%, es decir, se nos entregan más datos de los que necesitamos. Esto es debido a que las peticiones se hacen a nivel de *warp*, es decir, a nivel de 32 *threads*. Sin embargo, al ser vectores de 26 posiciones, hay *threads* del mismo *warp* que quieren acceder a las mismas posiciones.

Si nos fijamos en los 28.376 GB/s (ancho de banda efectivo), vemos que si el *kernel* ha tardado en ejecutarse 0.103 ms (menos de lo indicado en la Tabla 3, ya que este es el tiempo efectivo del *kernel*) tenemos que se han querido leer 3,138,255 bytes en total. Si tenemos 128,529 peticiones (ya que hay 1 transacción por petición), tenemos que se quieren leer 24 bytes por petición (no llega a los 32 bytes que se nos ofrece por petición). Incluso vemos que hay *warps* en que no se necesitan leer como mínimo las 26 posiciones del vector.

Si se nos indica que el ancho de banda global ha sido de 37.014 GB/s, significa que se han entregado 4,086,424 bytes en total. Si tenemos 128,529 peticiones vemos que se entregan los 32 bytes por petición.

Por otro lado, si nos fijamos en el bloque de las escrituras este si tiene una eficiencia del 100%, ya que cada *thread* escribe en una posición distinta del vector resultado. Por lo tanto, se realizarán peticiones de escritura de 32 *bytes* y por ende, se realizarán transacciones de 32 *bytes* de escritura

Visto esto, sabemos que el ancho de banda de las lecturas está “inflado”, ya que hay datos que trae que no se utilizan, y que sumado al ancho de banda de las escrituras tenemos que hay un total de 32.489 GB/s. Sabiendo que la K40c tiene un ancho de banda máximo de 288 GB/s, utilizamos solo el 11.281% de este. Sin embargo, el funcionamiento del código no permite mejorar el acceso a memoria por las dos razones comentadas: se repiten accesos y además, son accesos a *bytes*, por lo que la memoria nunca nos va a entregar transacciones de su máxima capacidad.

2.5.3.4 Kernel con shared memory

Acabamos de ver que el programa no permite acceder a la memoria de la forma más óptima. Sin embargo, también hemos visto que datos leídos son aprovechados por más de un *thread* en el mismo warp. Si esto es así, se pueden aprovechar estos datos aún más para todo el bloque de *threads*. Utilizaremos la *shared memory* para ello.

Para ello, vamos a inicializar nuevos vectores, Figura 36, indicando que van a formar parte de la *shared memory*.

```
__shared__ char sPlug[26];
__shared__ char srFast[26];
__shared__ char srMiddle[26];
__shared__ char srSlow[26];
__shared__ char sVolta[26];
__shared__ char sVSlow[26];
__shared__ char sVMiddle[26];
__shared__ char sVFast[26];
```

Figura 36. Inicialización de la *shared memory*

Para cargarlos con los mismos valores que hay en la memoria global, vamos a utilizar los primeros 26 *threads* de cada bloque, tal y como se puede ver en la Figura 37. Por último, se sincronizan los *threads* del bloque para que ninguno empiece su ejecución sin tener los vectores de la *shared memory* completamente cargados.

```
if (thread < 26) {
  sPlug[thread] = Plug[thread];
  srFast[thread] = rFast[thread];
  srMiddle[thread] = rMiddle[thread];
  srSlow[thread] = rSlow[thread];
  sVolta[thread] = Volta[thread];
  sVSlow[thread] = rVSlow[thread];
  sVMiddle[thread] = rVMiddle[thread];
  sVFast[thread] = rVFast[thread];
}

__syncthreads();
```

Figura 37. Escrituras en la *shared memory* dependiendo del id tel *thread*

2.5.3.4.1 Resultados de ejecución

En la Tabla 5 vamos a ver un resumen de como quedan los nuevos tiempos de ejecución, en ms, para cada distribución de trabajo. Seguimos trabajando con una muestra de 100 ejecuciones.

			Optimizado	256 th/bl (no shared)	128 th/bl (shared)	256 th/bl (shared)	512 th/bl (shared)
Encriptador	Total	Media	17.08	0.539	0.535	0.510	0.458
	Kernel	Media	No aplica	0.125	0.107	0.107	0.106

Tabla 5. Tiempos cálculo matriz Encriptador según distribución con *shared memory*

Por lo que los *speedup* quedarían como en la Tabla 6 escogiendo los nuevos tiempos de ejecución de la distribución de 512 *threads*/bloque con *shared memory* (menor tiempo total).

		Optimizado	256 th/bl (no *shared*)
Encriptador (Total)	*Speedup*	37.3	1.18

Tabla 6. Speedup respecto Enigma paralela y 256 th/bl de la distribución 512 th/bl con shared memory

Si volvemos a analizar el comportamiento del acceso a memoria, vemos que paras las lecturas vamos a aumentar la eficiencia hasta el 81.25% y las escrituras se quedan igual Figura 38.

```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: EnigmaWithoutPointersKernelShared(char*, char*, char*, char*, char*, char*, char*, char*, char*, int, int)
          1                            gld_efficiency             Global Memory Load Efficiency      81.25%      81.25%      81.25%
          1                  gld_requested_throughput          Requested Global Load Throughput  2.0126GB/s  2.0126GB/s  2.0126GB/s
          1                            gld_throughput                    Global Load Throughput  2.4771GB/s  2.4771GB/s  2.4771GB/s
          1              gld_transactions_per_request      Global Load Transactions Per Request    1.000000    1.000000    1.000000
          1                         gld_transactions                  Global Load Transactions        7144        7144        7144

          1                            gst_efficiency            Global Memory Store Efficiency     100.00%     100.00%     100.00%
          1                  gst_requested_throughput         Requested Global Store Throughput  4.9515GB/s  4.9515GB/s  4.9515GB/s
          1                            gst_throughput                   Global Store Throughput  4.9517GB/s  4.9517GB/s  4.9517GB/s
          1              gst_transactions_per_request     Global Store Transactions Per Request    1.000000    1.000000    1.000000
          1                         gst_transactions                 Global Store Transactions       14281       14281       14281
```

Figura 38. Nvprof con métricas de memoria para ejecución 512 th/bl con *shared memory*

Si nos fijamos en el bloque de las lecturas, se realizan muchas menos peticiones, debido a que solo se realizan aquellas en que el dato que se necesita no está en la *shared memory*. Si tenemos que el ancho de banda efectivo es de 2.0126 GB/s, y el *kernel* se ejecuta en 0.087 ms (más rápido que antes), tenemos que se leen 188,926 *bytes* (menos que antes) y si se

realizan 7,144 peticiones tenemos que se piden 26 *bytes* por petición (el número de elementos de los vectores).

Por otro lado, las escrituras se comportan igual en cuanto a memoria global, ya que la *shared memory* no influye en estas.

Veamos en la Figura 39 cómo se comportan los accesos en la *shared memory.*

```
Invocations                               Metric Name                               Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: EnigmaWithoutPointersKernelShared(char*, char*, char*, char*, char*, char*, char*, char*, int, int)
          1                         shared_efficiency                         Shared Memory Efficiency      12.38%      12.38%      12.38%

          1                    shared_load_throughput                    Shared Memory Load Throughput  356.17GB/s  356.17GB/s  356.17GB/s
          1                   shared_load_transactions                   Shared Load Transactions      128529      128529      128529
          1      shared_load_transactions_per_request    Shared Memory Load Transactions Per Request    1.000000    1.000000    1.000000

          1                   shared_store_throughput                   Shared Memory Store Throughput  19.797GB/s  19.797GB/s  19.797GB/s
          1                  shared_store_transactions                  Shared Store Transactions        7144        7144        7144
          1     shared_store_transactions_per_request   Shared Memory Store Transactions Per Request    1.000000    1.000000    1.000000
```

Figura 39. Nvprof con métricas de *shared memory* para ejecución 512 th/bl

Vemos que la eficiencia cae drásticamente con la *shared memory.* Esto es debido a que esta hace transacciones de 256 *bytes*, mientras que las peticiones son de 1 *byte* por *thread* en el *warp,* es decir, 32 *bytes*.

Si quisiéramos comprobar la eficiencia del 12.38% que se nos muestra tenemos que ver lo siguiente:

- Realizamos 128,529 peticiones de lectura (1 transacción por petición) y cada una de ellas de 32 *bytes*, ya que cada *thread* quiere leer 1.
- Si sabemos que el programa tarda 0.087, tenemos un ancho de banda en lecturas de un 43.815 GB/s.
- Realizamos también 7,144 peticiones de escritura (1 transacción por petición) y cada una de ellas de 26 *bytes,* ya que los vectores tienen 26 posiciones.
- Por lo tanto, el ancho de banda para las escrituras es de 1.979 GB/s.
- Si sumamos el ancho de banda de la *shared memory* es de 375.967 GB/s. Si el ancho de banda efectivo es de 45.794 GB/s tenemos que este es un 12.18% del ancho de banda total. Obtenemos una desviación del 0.2% debido a que los anchos de banda se han tomado solo con una ejecución.

Si nos fijamos, el número de peticiones de escritura de la Figura 39 concuerda con el número de peticiones de lectura de la Figura 38, ya que son peticiones únicas de lecturas (sin repeticiones) para escribirlas en la *shared memory* una única vez por bloque.

Otro dato para fijarse es de que según la distribución del trabajo, los anchos de banda en la *shared memory* van a variar. A mayor número de bloques trabajando concurrentemente en

el *device,* vamos a tener un mayor ancho de banda en las escrituras y uno menor en las lecturas. Se obtiene un mayor ancho de banda en escrituras ya que las escrituras son por bloque, y al haber más, hay más peticiones de estas.

Por lo tanto, el código al ser así no aprovecha el ancho de banda que puede ofrecernos la *shared memory* pero, no lo necesita. Por tanto, la limitación está en este.

2.5.3.5 MultiGPU + *shared memory*

En el nodo donde se ejecutan los códigos de este proyecto hay 2 GPUs en total, por lo que podríamos dividir el trabajo total en estas dos, siempre y cuando, cada GPU vaya a trabajar al 100% (o por lo menos una). En este caso esto sucede siempre, ya que como hemos visto en el apartado "**2.5.3.2 Distribución del trabajo**", todas las distribuciones de trabajo tienen a los 30,7720 *threads* de la GPU trabajando concurrentemente y además, nos sobran bloques de trabajo para encargarlos a la otra GPU.

2.5.3.5.1 Envío de información a los *devices*

Nuestra idea detrás de trabajar con dos GPUs es ganar algo de concurrencia, por lo que también podemos ganarla mediante las llamadas asíncronas de transmisión de datos entre GPU y CPU.

Para conseguir esto, lo primero que hemos hecho es que la información del *host* que habrá que copiarla al *device* se genere con *pinned memory,* Figura 40, donde se asigna memoria no paginada y por lo tanto, puede alcanzarse una velocidad x2 respecto a un *malloc* siendo este memoria paginada.

```
cudaMallocHost((char**)&Plug, 26);
cudaMallocHost((char**)&rFast, 26);
cudaMallocHost((char**)&rMiddle, 26);
cudaMallocHost((char**)&rSlow, 26);
cudaMallocHost((char**)&rVFast, 26);
cudaMallocHost((char**)&rVMiddle, 26);
cudaMallocHost((char**)&rVSlow, 26);
cudaMallocHost((char**)&Volta, 26);
cudaMallocHost((char**)&Encriptador, 26*26*26*26);
```

Figura 40. Asignación no paginada de memoria, *pinned memory*

Con las llamadas de transferencia de datos asíncronas de la Figura 41, conseguimos que en cuanto lancemos la llamada, se le devuelve el control a la CPU mientras que la GPU realiza esta transferencia, ganando así concurrencia.

```
cudaMemcpyAsync(&d_Plug[0], &Plug[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rFast[0], &rFast[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rMiddle[0], &rMiddle[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rSlow[0], &rSlow[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rVFast[0], &rVFast[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rVMiddle[0], &rVMiddle[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rVSlow[0], &rVSlow[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_Volta[0], &Volta[0], 26, cudaMemcpyHostToDevice);
```

Figura 41. Transferencias asíncronas de datos entre GPU-CPU

Estas transferencias habrá que hacerlas dos veces, una para cada *device*.

2.5.3.5.2 *Kernel*

Al distribuirse el trabajo entre dos GPUs, vamos a tener que modificar el *kernel* mínimamente. El primer *device* ejecutará todas las posiciones que puede y el segundo las que quede, ya que lo más posible es que para el segundo *device*, le sobren posiciones a ejecutar.

Esto lo vamos a ver mediante un par de parámetros que vamos a pasar a cada *kernel*: "*offset*", que nos indicará que GPU está ejecutando el *kernel* y "numBloques", que nos va a indicar cuántos bloques tiene que ejecutar el *device.*

La idea detrás de esto es saber en que posición empieza a calcular cada *device*, y esto lo obtenemos mediante la sentencia mostrada en la Figura 42.

```
int i = blockIdx.x * blockDim.x + threadIdx.x + offset * (numBloques - 1) * blockDim + blockDim;
```

Figura 42. Cálculo del primer elemento a calcular para cada GPU

El resultado varía respecto al número de bloques que hay. Por ejemplo, si el *device* es el 0 y la distribución es de 128 *threads* por bloque (1786 bloques), sabemos que el último elemento que va a calcular va a ser el 228,607 (*blockIdx.x* será 1785, *blockDim.x* será 128 y *threadIdx.x* será 127). Por lo tanto, el *device* 1 tiene que empezar calculando por la posición 228,608.

2.5.3.5.2.1 Resultados de ejecución

En la Tabla 7 se muestran los nuevos tiempos de ejecución para dos GPUs. Seguimos teniendo una muestra de 50 ejecuciones y en el orden de ms.

		Optimizado	512 th/bl (shared)	128 th/bl (2 GPUs)	256 th/bl (2 GPUs)	512 th/bl (2 GPUs)
Encriptador (Total)	Media	17.08	0.458	0.114	0.118	0.117

Tabla 7. Tiempos cálculo matriz Encriptador según distribución con 2 GPUs

Por lo tanto, el *speedup* final cogiendo la nueva distribución de 128 th/bl (las tres son perfectamente válidas, ya que la desviaciones son del orden 0.001) para el cálculo de la matriz Encriptador de nuestro código de la Enigma es el de la Tabla 8.

		Optimizado	512 th/bl (shared)
Encriptador (Total)	*Speedup*	149.85	4.02

Tabla 8. *Speedup* respecto Enigma paralela y 512 th/bl con *shared memory* de la distribución 128 th/bl con 2 GPUs

Como podemos observar, aún ejecutando con 2 GPUs sobrepasamos el 2 de *speedup* respecto a la anterior versión de *shared memory.* Esto es debido a las transmisiones asíncronas y a la concurrencia que estas permiten. Durante los resultados que hemos ido viendo de la matriz Encriptador, se utilizaba más tiempo en las transmisiones de datos que en el propio *kernel*, y aquí, cuando permitimos concurrencia, se refleja mejor.

2.5.4 Paralelizar el acceso a Encriptador

2.5.4.1 Envio de información al *device*

Para cualquiera de los tres casos vistos anteriormente; Caso A, Caso B y Caso C, el kernel va a necesitar toda la matriz Encriptador, todo el texto que queremos encriptar/desencriptar y toda la matriz Ruta, Figura 43. Como el acceso a la matriz Camino se hace dos veces para el Caso C, y una vez para el Caso A, estos, los hacemos antes de mandar a paralelizar, así no tenemos que enviar esta matriz al *device* y por ende, nos ahorramos un malloc.

```
cudaMalloc((char **) &d_Encriptador, 26*26*26*26);
cudaMalloc((char **) &d_Data, tam);
cudaMalloc((char **) &d_RutaS, 16900);
cudaMalloc((char **) &d_RutaM, 16900);
cudaMalloc((char **) &d_RutaF, 16900);

cudaMemcpy(d_Encriptador, &Encriptador, 26*26*26*26, cudaMemcpyHostToDevice);
cudaMemcpy(d_Data, Data, tam, cudaMemcpyHostToDevice);
cudaMemcpy(d_RutaS, &RutaS, 16900, cudaMemcpyHostToDevice);
cudaMemcpy(d_RutaM, &RutaM, 16900, cudaMemcpyHostToDevice);
cudaMemcpy(d_RutaF, &RutaF, 16900, cudaMemcpyHostToDevice);
```

Figura 43. Asignación de memoria y transferencia de datos entre GPU-CPU

2.5.4.2 Distribución del trabajo

Al igual que en el apartado "**2.5.3.2 Distribución del trabajo**", los tres mejores resultados los hemos obtenido con las siguientes distribuciones: 128 threads por bloque, 256 threads por bloque y 512 threads por bloque.

En este caso, el tamaño con el que trabajaremos va a ser el tamaño del texto a encriptar/desencriptar. Para nosotros, 235,974 caracteres. No se va a hacer una explicación detallada, ya que se realizó en el apartado "**2.5.3.2 Distribución del trabajo**" y los cálculos són idénticos.

2.5.4.2.1 128 threads por bloque

En total hay 1,844 bloques de 128 threads cada uno ((235,974 + 125) / 126), 16 bloques trabajando por SM y por lo tanto, 240 bloques trabajando en el *device*: 30,720 threads trabajando.

2.5.4.2.2 256 threads por bloque

En total hay 922 bloques de 256 threads cada uno ((235,974 + 255) / 256), 8 bloques trabajando por SM y por lo tanto, 120 bloques trabajando en el *device*: 30,720 threads trabajando.

2.5.4.2.2 512 threads por bloque

En total hay 461 bloques de 512 threads cada uno ((235,974 + 511) / 512), 4 bloques trabajando por SM y por lo tanto, 60 bloques trabajando en el *device*: 30,720 threads trabajando.

2.5.4.3 *Kernel*

Creamos tres kernels para cada uno de los casos. Estos kernels se diferencian en el primer elemento que se accede de la matriz Ruta. En el Caso C, se accede al primer elemento de Ruta, ya que la posición inicial de los rotores y la siguiente (donde había cambio en el rotor central) están guardadas en la matriz Camino que ya se ha accedido. En el Caso B se accede también a la primera posición de Ruta, ya que en Camino solo guardamos la posición inicial de los rotores. Por último, con el Caso A tenemos que acceder a la segunda posición de Ruta, ya que en la primera, tenemos la posición inicial de los rotores.

Si por ejemplo cogemos de ejemplo el kernel del Caso B, Figura 44, vemos como vamos a acceder con la variable "j" a la primera posición del vector, ya que el primer thread del bloque 0 nos va a devolver el valor de 0 para la variable "i".

```
int i = blockIdx.x * blockDim.x + threadIdx.x;

if (i < tam) {
  char PS, PM, PF, alfb;
  int j = i%16900;

  PS = RutaS[j];
  PM = RutaM[j];
  PF = RutaF[j];

  alfb = Data[i];
  Data[i] = Encriptador[PS*26*26*26 + PM*26*26 + PF*26 + alfb-'A'];
}
```

Figura 44. Kernel de acceso a Encriptador para el Caso B

Por lo tanto, para el Caso C, tendremos la instrucción "int j = (i-1)%16900" y para el Caso A la instrucción "int j = (i+1)%16900".

Una vez tenemos la posición obtenida de los tres rotores, conseguimos la letra que queremos encriptar/desencriptar del texto y accedemos a la matriz Encriptador, tal y como vimos, previamente calculada.

En definitiva y tal como vemos en la Figura 44, lo que paralelizamos es el acceso a memoria del vector Ruta, del texto que queremos encriptar/desencriptar y de la matriz Encriptador.

2.5.4.3.1 Resultados de ejecución

El resultado de los tiempos de ejecución para el acceso a Encriptador es el que se muestra en la Tabla 9. Muestra de 100 ejecuciones del código y en el orden de ms.

			Optimizado	128 th/bl	256 th/bl	512 th/bl
Acceso	Total	Media	885.38	0.353	0.356	0.358
	Kernel	Media	No aplica	0.047	0.047	0.047

Tabla 9. Tiempos acceso Encriptador según distribución

Como vemos, conseguimos los siguientes *speedup*, Tabla 10, respecto al código optimizado.

		128 th/bl	256 th/bl	512 th/bl
Acceso (Total)	*Speedup*	2508.16	2487.02	2473.13

Tabla 10. *Speedup* respecto Enigma paralela según distribución

Vemos que el *speedup* es muy elevado en este caso. Esta optimización se comporta mejor que la que vimos en el apartado "**2.5.3.3.1"** donde miramos los resultados de los tiempos sin utilizar *shared memory* ni multiGPUs para el precálculo de la matriz Encriptador. Si nos volvemos a basar en lo que se mostró en el apartado "**2.5.2 Objetivos de una ejecución en cuda**", volvemos a cumplir por lo menos, dos de los tres objetivos que propusimos: tenemos una buena distribución del trabajo (todos los *threads* tienen trabajo) y no hay prácticamente divergencia en el camino de estos (solo en los últimos).

Veamos cómo se comportan los accesos a memoria para este *kernel,* Figura 45. Nos basaremos en la distribución de 128 th/bl que es la que mejor *speedup* nos ha dado.

```
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: kernelC(char*, char*, char*, char*, char*, int)
          1                            gld_efficiency             Global Memory Load Efficiency      15.32%      15.32%      15.32%
          1                  gld_requested_throughput          Requested Global Load Throughput  38.700GB/s  38.700GB/s  38.700GB/s
          1                              gld_throughput                Global Load Throughput  252.68GB/s  252.68GB/s  252.68GB/s
          1                gld_transactions_per_request      Global Load Transactions Per Request    2.742915    2.742915    2.742915
          1                           gld_transactions                  Global Load Transactions      101145      101145      101145

          1                            gst_efficiency             Global Memory Store Efficiency      50.00%      50.00%      50.00%
          1                  gst_requested_throughput          Requested Global Store Throughput  7.7399GB/s  7.7399GB/s  7.7399GB/s
          1                              gst_throughput                Global Store Throughput  15.481GB/s  15.481GB/s  15.481GB/s
          1                gst_transactions_per_request     Global Store Transactions Per Request    1.249898    1.249898    1.249898
          1                           gst_transactions                  Global Store Transactions        9218        9218        9218
```

Figura 45. Nvprof con métricas de memoria para ejecución 128 th/bl

Como podemos observar, aunque la mejora haya sido tan grande, la eficiencia en las lecturas es muy baja, 15.32%. Esta eficiencia va a variar conforme que tipo de caso se esté ejecutando (A, B, C), vistos en el apartado "**2.4.1 Matriz Encriptador**". El Caso A, accede a la

vector Ruta con un *offset* de 1, el B con un *offset* de 0 y el C, también con un *offset* de 0. Si el el *offset* es 0 no hay problema, ya que los accesos van a estar alineados a 32, pero si tenemos un *offset* de 1, el *thread* 31 del *warp* va a acceder a la posición 32 del vector, por lo tanto, en ese caso serán 2 transacciones de lectura por petición de ella.

También vamos a tener alineamiento o no para acceder al vector Data dependiendo del caso. En el Caso C, la primera posición que se quiere acceder del vector Data en el *kernel* es el 1, ya que la posición 0 ya se ha calculado fuera de él. Por lo tanto, tenemos un *offset* de 1.

Finalmente, la que más penalización nos da es el acceso a la matriz Encriptador, ya que su posición depende de la letra del texto que queremos encriptar/desencriptar, y esta no se puede saber. Por lo tanto, podríamos decir que los accesos son aleatorios.

Para las lecturas, escribimos en la misma posición del vector Data que la que hemos leído para obtener la letra (transformamos la letra). Por lo tanto, al ser el Caso C el único que tiene *offset* en el acceso a Data, tenemos que la eficiencia de las lecturas es del 50%, ya que por petición, se van a generar 2 transacciones.

Por lo tanto, vemos que los accesos a memoria no son precisamente buenos, sin embargo, el *speedup* que obtenemos es muy bueno. Esto se debe a que ocultamos la latencia con memoria de una forma óptima, ya que las instrucciones de los *threads* son únicamente accesos a memoria.

2.5.4.4 MultiGPU

Igual que vimos antes, antes de todo hemos de mirar si se van a utilizar por lo menos una de las GPUs al 100%. Si revisamos el apartado "**2.5.4.2 Distribución del trabajo**", todas las distribuciones tienen los 30,720 *threads* trabajando concurrentemente. Además, nos sobran bloques de trabajo para que se encargue la otra GPU.

2.5.4.4.1 Envío de información a los *devices*

Como ya vimos, nos vamos a beneficiar de las transmisiones de datos asíncronas. Por lo tanto, lo primero va a ser asignar memoria no paginada en el *host* con *pinned memory*, Figura 46 y luego haremos las transferencias pertinentes dos veces (una por GPU), Figura 47.

```
cudaMallocHost((char**)&RutaS, 16900);
cudaMallocHost((char**)&RutaM, 16900);
cudaMallocHost((char**)&RutaF, 16900);
```

```
cudaMallocHost((char**)&Data, strlen(DataAux));
```

Figura 46. Asignación de memoria no paginada, *pinned memory*

```
cudaMemcpyAsync(&d_Encriptador[0], &Encriptador[0], 26*26*26*26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_Data[0], &Data[0], tam, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_RutaS[0], &RutaS[0], 16900, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_RutaM[0], &RutaM[0], 16900, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_RutaF[0], &RutaF[0], 16900, cudaMemcpyHostToDevice);
```

Figura 47. Transferencia de información entre GPU-CPU

2.5.4.4.2 *Kernel*

Al igual que vimos en el apartado "**2.5.3.5.2 *Kernel*",** tenemos que dividir el trabajo total en dos GPUs. Este número no lo sabemos fijo como pasaba en ese apartado, ya que el tamaño varía según el fichero de texto que queramos encriptar/desencriptar. Seguiremos pasando los parámetros de tipo entero que llamamos "*offset*" Y "numBloques" para saber qué GPU está ejecutando ese *kernel* y por ende, desde donde tiene que calcular, Figura 48 para el Caso A y B y Figura 49 para el Caso C.

```
int i = blockIdx.x * blockDim.x + threadIdx.x + offset * (numBloques - 1) * blockDim + blockDim;
```

Figura 48. Primer elemento a calcular por *device*. Casos A y B

```
int i = blockIdx.x * blockDim.x + threadIdx.x + offset * (numBloques - 1) * blockDim + blockDim - 1;
```

Figura 49. Primer elemento a calcular por device. Caso C

El Caso C es distinto, porque como ya comentamos, el primer *thread* accede a la segunda posición de Data, y por lo tanto, siempre hay un *offset* de 1.

2.5.4.4.2.1 Resultados de ejecución

En la Tabla 11 se muestran los nuevos tiempos de ejecución para dos GPUs. Seguimos teniendo una muestra de 50 ejecuciones y en el orden de ms.

		Optimizado	128 th/bl	128 th/bl (2 GPUs)	256 th/bl (2 GPUs)	512 th/bl (2 GPUs)
Acceso (Total)	Media	885.38	0.353	0.134	0.134	0.132

Tabla 11. Tiempos acceso Encriptador según distribución con 2 GPUs

Por lo tanto, conseguimos los siguientes *speedup*, Tabla 12, respecto al código optimizado y a la ejecución de 128 *threads* por bloque ejecutando el *kernel* en una sola GPU.

		Optimizado	128 th/bl
Acceso (Total)	*Speedup*	6707.43	2.67

Tabla 12. *Speedup* respecto Enigma paralela y 128 th/bl de la distribución 128 th/bl con 2 GPUs

Seguimos sobrepasando el 2 de *speedup* aún ejecutando con el doble de GPUs que la versión de 128 th/bl, por lo que las transferencias de información asíncronas siguen ayudándonos a mejorar.

2.6 Resultado de ejecución final

Si hacemos una tabla resumen con la progresión que ha ido adquiriendo el código optimizado a lo largo de los distintos apartados, obtenemos lo que podemos ver en la Tabla 13. Tiempos en ms.

			Optimizado	2 GPUs
Encriptador	Total	Media	17.08	0.114
	Kernel	Media	No aplica	No considerado
Acceso	Total	Media	885.38	0.132
	Kernel	Media	No aplica	No considerado
Total		Media	903.05	0.730

Tabla 13. Tiempo total de la Enigma paralela y la ejecución con 2 GPUs

Por lo que el *speedup* final del código de la Enigma es de un 1237.06.

3. Máquina Bombe

Cuando los ingleses se dieron cuenta de que el ejército alemán estaba encriptando sus mensajes, empezaron a idear cómo harían para desencriptarlos, y así, ganar tiempo para responder al movimiento de tropas.

Cuando se interceptaba un mensaje encriptado, si este quería ser desencriptado, se necesitaba tener la configuración inicial de la Enigma para ese dia, ya que como hemos mencionado, la Enigma trabaja en las dos direcciones con la misma configuración (encripta-desencripta).

3.1 Complejidad de la Enigma

Si hacemos una serie de cálculos, podemos observar las combinaciones totales que podía tener la Enigma, y por lo tanto, ver el problema al que se enfrentaban los ingleses para conseguir la configuración inicial.

Partiendo de tener 5 rotores totales y pudiendo poner 3 de estos 5 en cualquier orden (sin repetir), tenemos 60 posibles combinaciones para la selección de los rotores (5 x 4 x 3).

Una vez teníamos los rotores colocados, cada uno de ellos podía tener su letra inicial distinta, por lo que teníamos 17 576 posibles combinaciones iniciales (26 x 26 x 26).

Por último, y el mayor grado de complejidad que tenía la Enigma, el *Plugboard*. Sabiendo que el máximo de parejas que se podían conectar en el *Plugboard* era de 10, tenemos un total de 150 738 274 937 250 combinaciones posibles (26!/(6! x 10! x 2^10))[19]. Solo cuenta las combinaciones de 10 parejas porque es la más complicada de *crackear* debido a su complejidad y por lo tanto, eran las que siempre utilizabanm la milicia nazi, tal como vimos en la Figura 22 del apartado "**2.2 Funcionamiento**". Más adelante, nosotros tendremos en cuenta parejas menores de 10.

Si combinamos todo esto, existían 158 962 555 217 826 360 000 combinaciones posibles para la configuración inicial.

La primera forma para resolver esto y que a todos se nos ocurre es, a partir de tener una réplica de una máquina Enigma probar todas las configuraciones iniciales para esta y ver si lo que vamos desencriptando cobra sentido, es decir, fuerza bruta.

Esto es inviable, por dos motivos: el primero es por el número que acabamos de ver el total de combinaciones posibles; estas combinaciones hay que probarlas a mano. El segundo, para cada combinación, ir mirando si lo que vamos desencriptando tiene sentido. Además

de todo esto, no nos debemos de olvidar de que la configuración inicial de la Enigma cambiaba a diario, por lo que el descubrir la configuración inicial de ese día debía de conseguirse como mucho en horas, antes del próximo cambio.

3.2 Solución teórica

En este punto, había que pensar en un método que evitase probar todas las combinaciones posibles. El camino que habría que seguir es el de algún método que nos haga descartar combinaciones hasta acercarnos lo máximo posible a la correcta.

El matemático británico Alan Turing, se dio cuenta, de que era muy posible que los mensajes encriptados de los alemán siguieran un mismo patrón en algunas partes del texto[20]. Esto, era muy posible que ocurriera ya que se habían observado el comportamiento de los nazis y sobre todo, los discursos de los líderes y la respuesta de los seguidores.

Por ejemplo, sabemos que el gesto del saludo fascista iba la mayoría de veces acompañado de la frase *Heil Hitler!*. También, se había observado que se saludaba a los oficiales superiores con la frase *Sieg Heil*. Por lo tanto, la primera frase era muy posible que apareciese en las partes finales de los mensajes o la segunda frase, en la parte inicial de estos.

Sin embargo, las fuerzas de inteligencia de la parte aliada descubrieron que los mensajes militares alemanes seguían el mismo patrón al principio de cada mensaje: daban el parte meteorológico para ese día[20].

A partir de aquí, podíamos realizar los primeros pasos para obtener lo que habíamos comentado al inicio de este apartado: buscar combinaciones que nos dieran contradicciones para poder descartarlas. Vamos a ver un ejemplo de esto.

Imaginemos que tenemos un mensaje que nos llega encriptado con el siguiente aspecto:

QFZWRWIVTYRESXBFOGKUHQBAISEZ…

Y ahora, imaginemos que en ese momento, se sabía que había movimiento de tropas y enfrentamiento en el Golfo de Vizcaya. Podríamos suponer la frase *Wettervorher sage Biskaya*, que significa “Predicción del tiempo Vizcaya”. Si sabemos que todo lo relacionado con el tiempo era al principio del texto, podríamos suponer lo que se observa en la Figura 50. De ahora en adelante, esta hipótesis de texto la llamaremos “*crib*”, tal y como hacían los ingleses[20].

Figura 50. Posicionamiento del *crib* respecto al texto encriptado[20]

Con esto lo que se quiere demostrar es que cuando desencriptamos, la parte de arriba del texto sea la de abajo. Sin embargo, hay un problema. Como podemos observar en la Figura 50, en la posición 13 hay tanto una S arriba como una S abajo, y esto es imposible, ya que la enigma cuando encriptaba una letra jamás devolvía la misma (ya que el reflector generaba un camino distinto al de la ida). Por lo tanto, teníamos que desplazar el texto una posición hasta que esto no ocurriese, Figura 51.

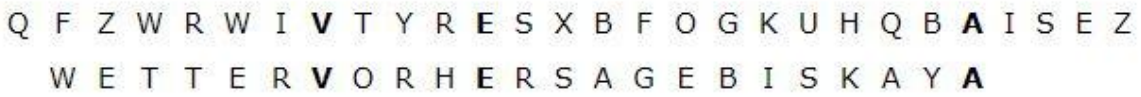


Figura 51. Desplazamiento del *crib* una posición[20]

En la posición 8 hay dos Vs, en la 12 dos Es y en la 24 dos Aes. Por lo que deberíamos seguir desplazando el texto hasta que no hubiera ninguna letra duplicada y conseguir lo que se muestra en la Figura 52.

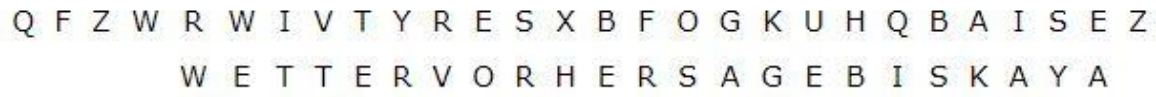


Figura 52. Desplazamiento del *crib* hasta no encontrar letras repetidas[20]

Tal y como he mencionado, queremos que cuando desencriptemos la parte "RWIVTYRESXBFOGKUHQBAISE" el resultado sea "WETTERVORHERSAGEBISKAYA". Para eso, vamos a escoger la letra que más aparezca en ambos textos y "simular" que la conectamos con otra letra en el *Plugboard*. Por ejemplo, vemos que la letra que más se repite en la Figura 52 es la "E". Podemos empezar haciendo que la "E" y la "A" estén conectadas en el *Plugboard*. Esto significa, que cada vez que en el texto veamos una E, a la Enigma le entrará una A y viceversa hasta que se llegue a una contradicción o no.

Un ejemplo de contradicción sería por ejemplo: en la posición 2, nosotros pulsamos la tecla “E” y a la Enigma le entra una “A” debido a la conexión que hemos dicho EA (y viceversa) del *Plugboard*, y saca una “P”. Como para la entrada “E” hemos forzado de que la salida debe de ser una “W” significa que hay la conexión PW (y viceversa) en el *Plugboard*. Si seguimos iterando y para la posición 14, hemos descubierto varias conexiones de *Plugboards* y una de ellas es GT (y viceversa), a la Enigma le va a entrar la letra “T”, debido a la última conexión que acabo de comentar. Si para esta ejecución, la salida es por ejemplo, una V, significa que estamos diciendo que existe la conexión AV (y viceversa), pero nuestra hipótesis inicial era que “A” estaba conectada “E”, por lo que hay una contradicción.

Además de esto último, Alan Turing se dio cuenta de que si llegamos a una contradicción, todas las hipótesis del camino pueden ser descartadas también y no hace falta volver a mirarlas, por lo que las conexiones EA, AE, GT, TG, AV, VA y todas las del camino son erróneas[21].

En este punto, si se han comprobado todas las posibles combinaciones de *Plugboard* para la letra “A” y todas han resultado contradicción, significa que o el texto debe de ser desplazado aún más (realmente no estará en esta porción del texto que hemos supuesto) o que la frase que hemos supuesto no es la correcta o no contiene suficientes letras diferentes como para descubrir contradicciones.

Alan Turing, junto a su equipo se dio cuenta de que aún con este método, era lento sacar la configuración inicial de la Enigma, ya que había igualmente que comprobar muchas operaciones. Por eso, idearon una máquina que hiciera estos descubrimientos de conexiones del *Plugboard* mediante la corriente, y por lo tanto, de una forma mucho más rápida y eficaz. A esta máquina se la conoce como la *Bombe*.

3.3 Componentes de la *Bombe*

La *Bombe* era una máquina que albergaba tres hileras de 12 conjuntos (horizontales) de 3 rotores (verticales), Figura 53. Es decir, cada conjunto de 3 rotores simulaba una máquina Enigma independiente. Por lo tanto, 36 máquinas Enigma. Esto permitía trabajar con *cribs* de una longitud máxima de 36 letras, es decir, una Enigma por cada pareja de letras del *crib* y del texto encriptado.

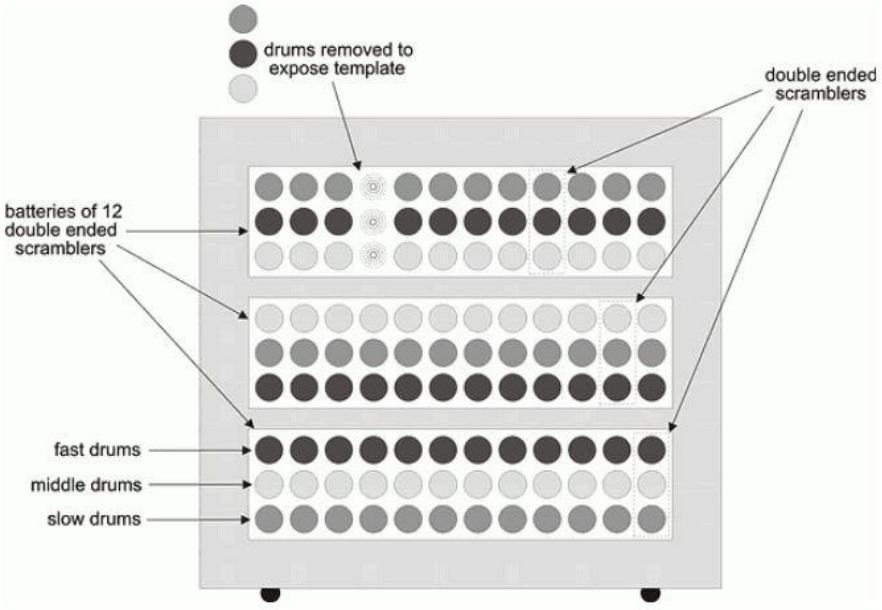


Figura 53. Dibujo del panel frontal de la *Bombe*[22]

3.4 Puesta a punto de la *Bombe*

Una vez visto el panel frontal de la *Bombe*, hay que conocer el panel trasero (el conexionado eléctrico) para ver como realmente funcionaba y además, nos servirá para transformar este funcionamiento en el código que simulará la ejecución de la *Bombe*.

Para la explicación, vamos a suponer que el abecedario va estar reducido a 8 letras (A-H). Esto va a simplificar la explicación y las Figuras. Vamos a suponer entonces, que tenemos el texto encriptado abajo y nuestro *crib* arriba en la Figura 54.

1	2	3	4	5	6	7	8	9
B	E	A	C	H	H	E	A	D
E	D	B	G	E	A	H	D	B

Figura 54. Posicionamiento del *crib* respecto al texto encriptado para un alfabeto reducido[23]

La máquina *Bombe* tenía 8 grupos de 8 cables cada grupo[23]. Cada grupo representaba una letra de las 8 totales, y cada cable una letra de las 8 totales. Por ejemplo, el grupo “A” contenía 8 cables que representaban las letras a-h, el grupo “B” contenía 8 cables que

representaban las letras a-h, y así sucesivamente, Figura 55. Esto servía para simular el *Plugboard* de la máquina Enigma; si queríamos que "A" y "R" estuvieran conectadas en el *Plugboard*, la letra "r" del grupo "A" era en la que se inyectaba la corriente.

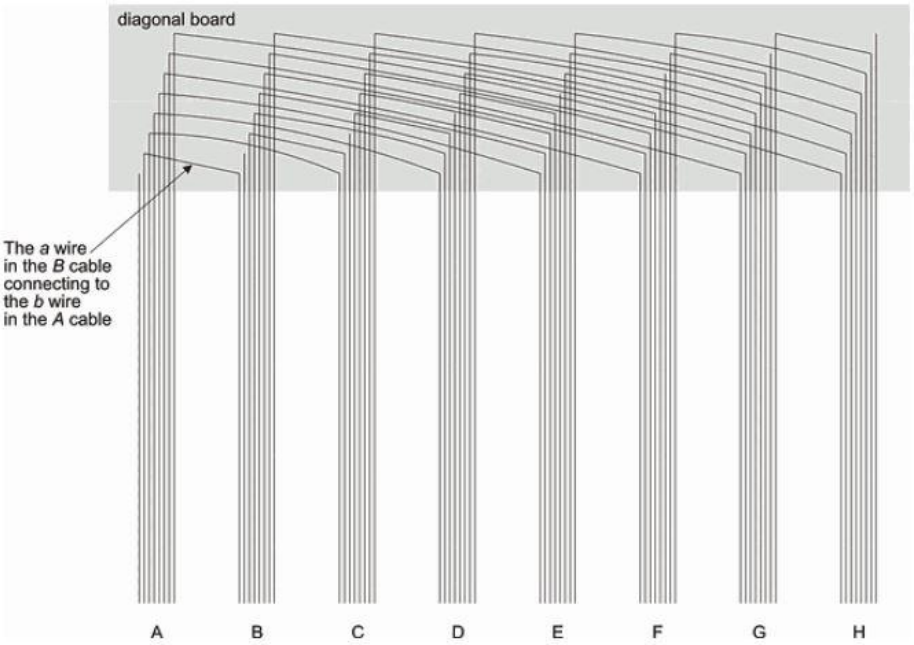


Figura 55. Conexionado en diagonal de la *Bombe*[23]

Además, sabemos que el *Plugboard* funciona a dos bandas; si "A" está conectada con "R", significa que "R" está conectada con "A". Es por esto, que se hace la conexión en diagonal que vemos en la Figura 55. Tal y como está explicado en inglés, si por ejemplo, queremos que "A" esté conectada con "B", se inyectará corriente en el cable "b" del grupo "A". Este cable también tendrá que estar conectado al cable "a" del grupo "B" para que también haya corriente en dicho cable.

Una vez tenemos el cableado colocado, pasamos a posicionar correctamente las Enigmas (conjunto de tres rotores en vertical).

Tal y como mencioné en el apartado anterior, "**3.3 Componentes de la *Bombe***", cada Enigma ejecutaría un par de letras de nuestro conjunto que creíamos que podía ser el correcto. Si nos basamos en este caso actual, el *crib* tiene un total de 9 letras, por lo que solo haría falta colocar 9 Enigmas.

Cada Enigma la tenemos que colocar con una posición inferior a la siguiente, o visto de otra forma, la próxima que coloquemos con una posición siguiente a la actual, ya que es el comportamiento de las iteraciones de una Máquina Enigma. Si seguimos trabajando con la

disposición de la Figura 54, si colocamos la primera Enigma (la que calcula la entrada “E” y la salida “B” o viceversa) en la posición AAA, la segunda Enigma (que calcula la entrada “D” y la salida “E” o viceversa) deberá estar en la posición AAB y así sucesivamente, Figura 56.

drum	cables connected by drum's scrambler	scrambler position in crib-ciphertext pairing	drums letter in 12 o'clock position
top	B and E	1	A
top	D and E	2	B
top	A and B	3	C
top	C and G	4	D
top	E and H	5	E
top	A and H	6	F
top	E and H	7	G
top	A and D	8	H
top	B and D	9	A
middle	B and D	9	B

Figura 55. Posición inicial de cada Enigma respecto a la posición en el texto[23]

Como podemos observar, se incrementa siempre el rotor “*top”,* es decir, el rápido. Y cuando se vuelve a llegar a la “A” en este, significa que ha dado una vuelta, por lo que incrementamos en uno el rotor “*middle*”.

Por lo tanto, el conexionado de las Enigmas (representado en cuadrados y haciendo referencia a la Figura 55) con el cableado sería el de la Figura 56. Como vemos, cada cuadrado tiene indicado la posición del texto con el que estamos trabajando y la posición inicial de los rotores. La Enigma con un “1”, trabajaría el descubrimiento de contradicción para las letras EB y viceversa (por eso está conectada entre los grupos “E” y “B”, ya que forzamos que si pulsamos la tecla “E”, la salida definitiva sea la “B” y viceversa. La Enigma con un “2” para las letras DE y viceversa, la “3” con las letras BA y viceversa y así sucesivamente.

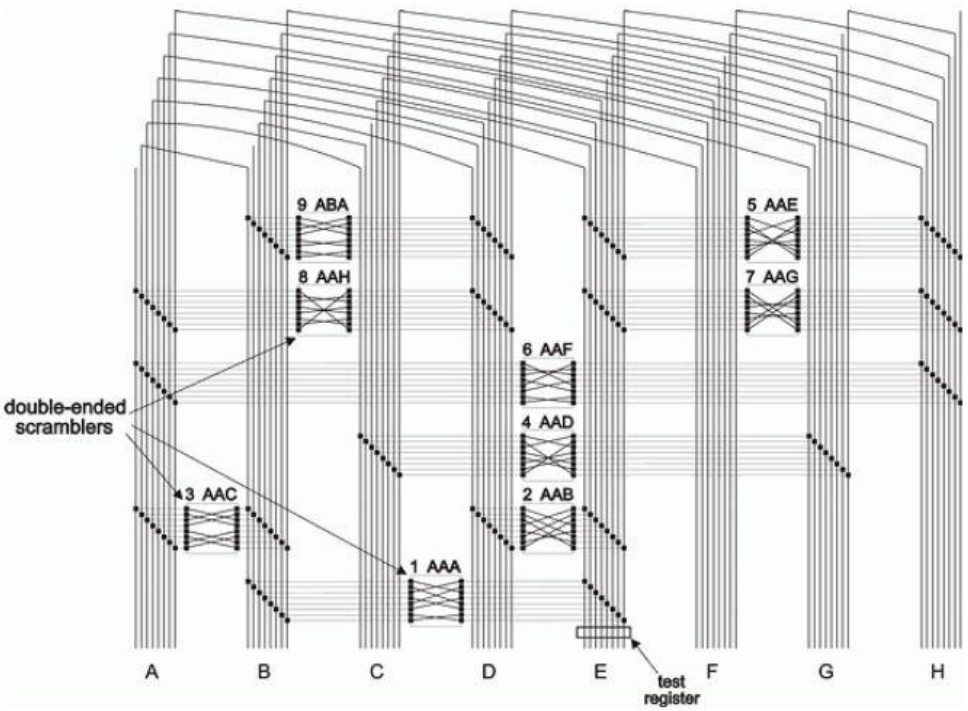


Figura 56. Cableado y colocación de las Enigmas[23]

También podemos observar en la Figura 56, que colocan un registro en un grupo de cables. Este, se coloca en el grupo que representa la letra que más aparece en el texto que estamos trabajando tal y como también hicimos en el apartado "**3.2 Solución teórica**", ya que será este registro el que nos indique si hay contradicciones o no.

3.5 Funcionamiento

Como acabamos de decir, vamos a trabajar con la letra "E", ya que es la que más aparece en nuestro texto, Figura 54. Y vamos a empezar creando la hipótesis de que "E" está conectada con "A" (primera letra del alfabeto que no es la "E") y viceversa.

1	2	3	4	5	6	7	8	9
B	E	A	C	H	H	E	A	D
E	D	B	G	E	A	H	D	B

Figura 54. Posicionamiento del *crib* respecto al texto encriptado para un alfabeto reducido[23]

Lo primero que habremos de hacer una vez hecho el conexionado y la puesta de los rotores, será inyectar voltaje por el cable de nuestra hipótesis, en nuestro caso, el cable “a” del grupo “E”, marcado de color negro en la Figura 57 (no hay que fijarse por ahora en los otros cables en negro). Sí que nos hemos de fijar en que este cable negro, sigue el conexionado en diagonal que vimos en el apartado anterior y por lo tanto, el cable “e” del grupo “A” también está en negro (con voltaje), creando las conexiones del *Plugboard* AE y viceversa.

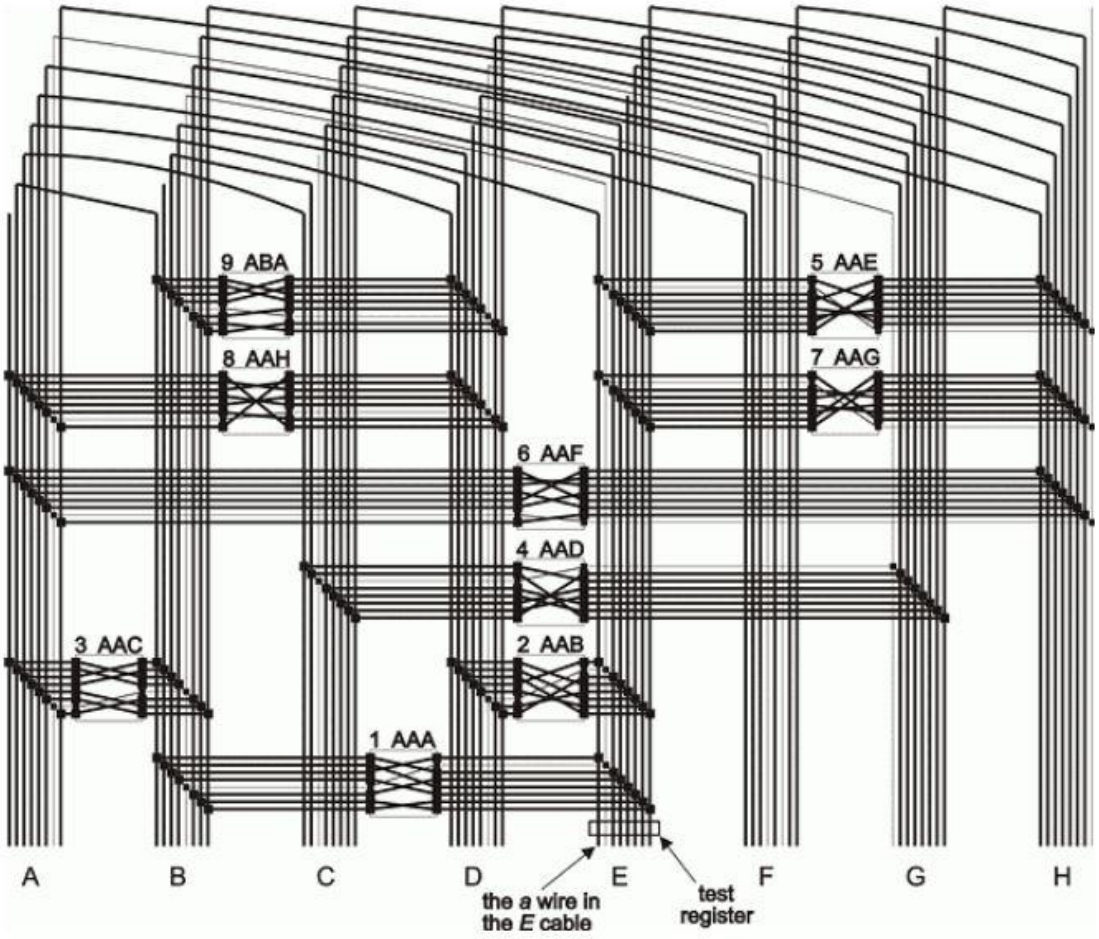


Figura 57. Muestra de contradicciones[24]

Como vimos en el apartado “**3.2 Solución teórica**”, la idea de la *Bombe* es descubrir contradicciones mediante descubrir incongruencias en los *Plugboards,* y esto es precisamente lo que se ve en la Figura 57.

Si volvemos a fijarnos en el cable negro (cable “a” del grupo “E”), veremos que es entrada de la Enigma etiquetada como 1 AAA, por lo que ejecutará el par de letras en la posición 1 de la Figura 54 con los rotores en la posición AAA. Este par de letras son: “B” y “E”. Como solo

tenemos voltaje en uno de los cables del grupo “E” solo será este el que funciona como entrada para la Enigma. Veamos en la Figura 58 una ampliación de dicha Enigma.

Figura 58. Interconexionado de la primera Enigma[24]

La entrada de la Enigma sería el cable “a” del grupo “E”, es decir, la primera entrada de arriba a la derecha. Si seguimos la trayectoria del conexionado interno (está simplificado al de una Enigma real), la salida es la tercera de la izquierda, por lo que sería el cable “c”.

Por ahora tenemos, que se pulsa la tecla “E”, y como esta, está conectada a la letra “a”, es esta última la que le entra a la Enigma. La Enigma itera y saca una “c”. Sin embargo, en nuestras suposiciones de la Figura 54, nosotros forzamos a que cuando presionamos la tecla “E”, la Enigma nos devuelva una “B”, por lo que suponemos, que “B” y “c” están conectadas mediante el *Plugboard*. Por lo tanto, y como se vuelve a ver en la Figura 57, a parte de tener voltaje en el cable “c” del grupo “B”, le llega también al cable “b” del grupo “C” mediante el conexionado en diagonal.

Con esto último, hemos inyectado voltaje en dos grupos más, por lo que las Enigmas conectadas con dichos grupos, van a tener corriente para computar una salida. Por ejemplo, hemos inyectado voltaje en el cable “c” del grupo B, por lo que las Enigmas que estén conectadas a ese grupo computarán, en este caso tanto la 3 AAC como la 9 ABA.

Para que se entienda del todo las contradicciones, veamos lo que sucede con la Enigma 3 AAC, Figura 59.

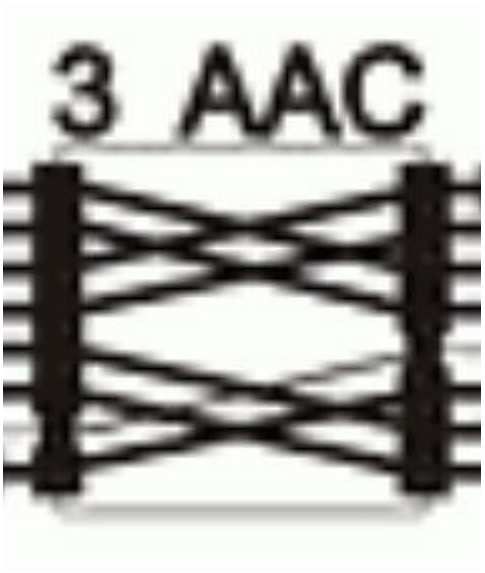


Figura 59. Interconexionado de la tercera Enigma[24]

Como hemos dicho, le va a entrar la letra “c” del grupo “B”, por lo que es la tercera entrada de la derecha. Si seguimos el conexionado, computará la primera salida de la izquierda, es decir, una “a”. Dado que esta Enigma está conectada con el grupo “A”, significa que la letra “A”, estará conectada con “a”. Esto podría ser correcto, ya que si ocurre esto es lo mismo que no conectar la letra “A” con nada en el *Plugboard*. Sin embargo, al principio dijimos que las letras “E” y “A” estaban conectadas, por lo que tenemos una contradicción. Además, dos cables del grupo “A” pasarán a tener voltaje, el “e” que le llegó de la primera iteración y el “a” de esta.

Sabemos que hay una contradicción porque estamos analizando con un dibujo, pero la *Bombe* lo detectaba mediante el registro que pusimos en el grupo “E”. Es decir, se esperará a ver si se llenan más cables con voltajes y se parará (lo veremos en el próximo apartado), como le ha pasado al grupo “A”.

En la Figura 57, vemos que al final se han llenado de voltaje todos los cables del grupo “E” menos uno, lo que significaría que la letra “E”, estaría conectada con “a”, “c”, “d”, “e”, “f”, “g” y “h”, cosa que no es posible.

En resumen, la idea que vemos es que nosotros solo inyectamos voltaje en nuestra hipótesis inicial y la máquina nos descubre si es correcta o no llenando con más voltaje más cables del mismo grupo.

3.5.1 Parada de la *Bombe*

La *Bombe* tenía un mecanismo de parada únicamente cuando en el registro que hemos colocado en el grupo de nuestra hipótesis, en nuestro caso el grupo "E", se detectaban solo 1 cable con voltaje (el de nuestra hipótesis inicial) o 25 cables con voltaje.

Si solo veíamos a nuestro cable con el cual habíamos inyectado voltaje significaba que el grupo de nuestra hipótesis no había recibido ninguna contradicción por ahora. Si por el contrario, veíamos que nuestro grupo de la hipótesis tenía 25 cables con voltaje, como lo visto en la Figura 57, significaba que sí había tenido contradicciones excepto para un cable, el que no tenía voltaje y este podía ser la hipótesis correcta. En la Figura 60 podemos ver si inyectamos solo con voltaje el cable que no había salido con voltaje en la Figura 57, vemos que no se crea ninguna contradicción y la *Bombe* también pararía.

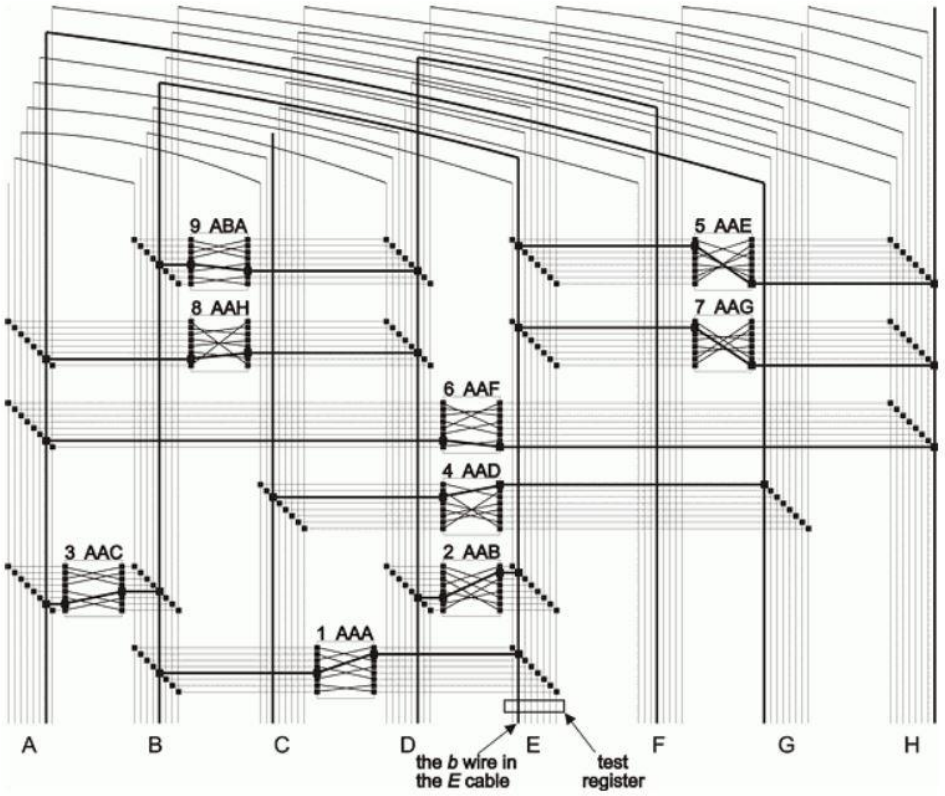


Figura 60. Muestra de no encontrar contradicciones[24]

Con cualquiera de las dos opciones, la *Bombe* entonces miraba si se cumplía lo mismo en los demás grupos (si solo tenían 1 cable con voltaje o 25), y si era así, se paraba y entonces un encargado anotaba la posición de los rotores de cada enigma y el cable con voltaje o sin voltaje de cada grupo de cables.

Si la bombe ya había hecho una iteración de descubrimiento de contradicciones y no se daba ninguna de las dos situaciones anteriores, la *Bombe* incrementaba la posición de cada

Enigma en 1, se volvía a inyectar voltaje en el cable de nuestra hipótesis inicial y volvía a descubrir contradicciones. Esto era, porque la posición de los rotores no era la correcta en la configuración inicial de la Enigma que habían utilizado para encriptar el mensaje.

Si se llegaba a la última posición de los rotores y la *Bombe* aún no había parado podía significar tres cosas: la primera, es que a lo mejor nuestro *crib* no estaba correctamente colocado respecto al texto encriptado y por lo tanto, teníamos que desplazarlo aún más. La segunda, era que a lo mejor este *crib* no era correcto, y la tercera, que los rotores no eran los correctos o su orden no era el correcto.

Aquí nos podemos dar cuenta de la complejidad que tenía esta operación, por eso, utilizaban más de una *Bombe*, para probar distintos rotores, con distinto orden, distinta posición del *crib*…

3.6 Código secuencial

Con el código secuencial hemos tratado de ser lo más fiel posible a la máquina *Bombe*, con la diferencia de que al tener una máquina tan potente como son los ordenadores de hoy en día, simulamos una *Bombe* que pruebe con todas las posibles combinaciones de rotores. Recordemos que en aquella época, el orden de los rotores y qué rotores poner lo hacían a mano.

Además, para el *cribe* que estamos probando, se harán un total de 50 desplazamientos, ya que es dentro de las primeras 50 letras de un texto encriptado donde se encontrará el parte meteorológico, tal y como comentamos anteriormente.

En la Figura 61 puede verse el cuerpo principal de nuestro código secuencial para la máquina *Bombe*. El primer bucle hace referencia a los dos tipos de reflectores que puede tener una máquina Enigma. Los tres siguientes hacen todas las posibles combinaciones de tres rotores con los 5 totales que hay, mirando que no se repitan ninguno. Una vez tenemos estos datos, se inicializa una Enigma.

```
for (int reflector=1; reflector<3; reflector++)
  for (int slow=0; slow<5; slow++)
    for (int middle=0; middle<5; middle++) {
      if (middle != slow) {
        for (int fast=0; fast<5; fast++) {
          if (fast != middle && fast != slow) {
            EnigmaSet ConfigEnigma = {fast, middle, slow, reflector + 'A', 'A', 'A', 'A', ""};
            InitEnigmaMachine(ConfigEnigma) ;
            for (int p=0; p<50; p++) {
              int duplicated = 0;
              for (int i=0; i<strlen(DataIn)-1 && !duplicated; i++)
                if (DataIn[i] == DataIn2[i+p])
                  duplicated = 1;
                if (!duplicated) {
                  for (int s=0; s<26; s++)
                    for (int m=0; m<26; m++)
                      for (int f=0; f<26; f++) {
```

Figura 61. Código del cuerpo de la *Bombe*

Esta Enigma recorrerá las 50 primeras posiciones del texto, tal y como dijimos. Mirará si es una posición posible (es decir, comprobará que no hay duplicación de letras, igual que hicimos en el apartado "**3.2 Solución teórica**" y probará si se genera contradicciones para cualquier posición inicial del primer rotor.

Para que quede más clara la diferencia con que haría una *Bombe* real transformada en código sería la de la Figura 62. Lo único automatizado que sería son los tres bucles de las posiciones de los rotores, ya que como dijimos, si la *Bombe* no detectaba una parada, esta incrementaba el una posición el rotor rápido (y si a los otros dos rotores les tocaba, también). Tanto la colocación de los rotores, como colocar las Enigmas de la *Bombe* simulando la posición adecuada en el texto lo hacían los operarios a mano.

```
EnigmaSet ConfigEnigma = {fast, middle, slow, reflector + 'A', 'A', 'A', 'A', ""};
InitEnigmaMachine(ConfigEnigma) ;
for (int s=0; s<26; s++)
  for (int m=0; m<26; m++)
    for (int f=0; f<26; f++) {
```

Figura 62. Código del cuerpo real de la *Bombe*

Una vez lista la *Bombe* para iterar con una posición inicial para los rotores, inicializamos a 0 una matriz (wires) que simula el cableado explicado en el apartado "**3.4 Puesta a punto de la *Bombe***" Figura 55. Por lo tanto, será una matriz con 26 vectores de 26 posiciones por vector. Cada uno de los vectores hace referencia a un grupo de cables representado una letra (A-Z), y cada una de las 26 posiciones hace referencia a uno de los 26 cables que tiene cada grupo (a-z). El hecho de inicializar a 0 simula que ningún cable contiene voltaje en ese momento.

Después de esto, se pone a 1 el elemento de la matriz correspondiente a nuestra hipótesis inicial, haciendo referencia que en ese cable, de ese grupo hay voltaje. Por ejemplo, si nuestra hipótesis inicial es que "E" está conectada a "A" y viceversa, significa, como vimos, que el cable "a" del grupo "E" tendrá voltaje, y que el cable "e" del grupo "A" también tendrá. Si llamamos a "E" "hip" y a "A" "hip2", quedaría como se ve en la Figura 63.

```
wires[hip - 'A'][hip2 - 'A'] = wires[hip2 - 'A'][hip - 'A'] = '1';
```

Figura 63. Sentencia que simula inyección de voltaje en la hipótesis

A partir de tener un cable ya con voltaje, la Enigma/s que está/n conectada/s a ese cable ya puede producir salidas y por lo tanto, dejar con voltaje esos cables y por lo tanto, posibilitar a otras Enigmas que saquen salidas y así sucesivamente. Por lo tanto, cada vez que a un cable le llegue voltaje por primera vez, lo encolaremos de la misma manera que hemos hecho con la hipótesis inicial. Y cuando todas las Enigma de un cable hayan computado, se desencola este cable (haciendo ver que ya se han descubierto salidas por ellos y por lo tanto, ya no hace falta volver a computar con ellos).

Una vez no queden elementos en la cola, la *Bombe* pasará a ver cuántos cables hay con voltaje para el grupo de nuestra hipótesis inicial (letra "E" o lo que habíamos llamado en el código: "hip"), Figura 64. Esto simula lo que hacía la *Bombe* mirando los cables con voltaje que había en el registro. Recordemos que solo nos interesaba que hubiera 1 o 25.

```
count = 0;

for (int i=0; i<26; i++)
  if (wires[hip - 'A'][i] == '1')
    ++count;
```

Figura 64. Código que simula la detección de voltaje en el registro

Si el número de cables con voltaje es 1, lo único que el código va a hacer es una pasada por todos los grupos de cables restantes viendo si hay solo 1 cable con voltaje por grupo o ninguno, Figura 65. Es decir, tal y como vemos, si hay más de un cable con voltaje significa que hay una contradicción en ese grupo de cables, por lo que no nos sirve esa salida de la *Bombe*.

```
int contra = 0;

for (int i=0; i<26 && !contra; i++) {
  count = 0;
  for (int j=0; j<26; j++)
    if (wires[i][j] == '1')
      ++count;
  if (count > 1)
    contra = 1;
}
```

Figura 65. Código que comprueba que todos los grupos tienen como máximo 1 cable con voltaje

Si por el contrario, el número de cables con voltaje es 25, el código hará una iteración extra de la *Bombe* inyectando voltaje en el cable que no lo tenía y luego, hará lo mismo que en la Figura 60.

3.6.1 Formato de salida

Una vez el programa ha trabajado todas las posiciones posibles, nos va a volcar un *output* como el de la Figura 66. El resultado son todas las posibles combinaciones de rotores, reflector, posición inicial de los rotores y *plugboard* (por este orden) que son candidatos a ser la configuración inicial de la Enigma correcta. Son candidatos porque aún no han generado contradicción (esto dependerá, tal y como vimos en el apartado "**3.2 Solución teórica**", de la cantidad de letras distintas que podamos descubrir mediante nuestro texto.

```
I II IV B VWD AU CD EX FZ HW IS KM LL PY QT RV
I V II B TSA BB CI DE FJ GQ HN LM PU TX VV YZ
I V III B ADO BB CX EE GR HY IM JQ KL NN PZ SV UU
II I III B LKD AU BX CS DT EW FZ GO IP QY RV
II I III B SPG AP BB CI DH EV FU GO KK LM QZ SW
II I IV B GBI AX BG CD EP FZ IY KU LM RS TV
II III I B CSA AM BD CO EL FJ GH QZ RR SX VW YY
II IV I B ZIQ AZ BY EV FU GT HS IR JQ KK LL NN OO
II IV I B ZJQ AZ BY EV FU GT HS IR JQ KK LL NN OO
II V III B HKC AM BY CV DE FS GR HN JK LP TX
III II I B PIG AH BB CD EW FK GI LL NX PS RR TZ VV
III II V B PSV AW BD CR ES FX GH JJ LL MM OV PQ YZ
III V IV B YYS AC BG DD EJ FX HM KT LQ NV PP RS
IV III II B XKF AO BH CJ DL EG FF MS PQ RZ VX WY
IV III II B DZN AX BD EE FH GG IT JN KP LO MW QU YZ
IV V I B PPX AZ BQ CE FJ GU HI KL MO PY TX
V IV III B YLY AM BD EQ FJ GP HK LS NV RX TY
I II V C TML AB DI EL FH GZ JT KQ MN PV UY
I IV V C OHL AA BD EE FZ GS IX KU OO PQ RR TY VW
I V II C EDO AA BM CD ES FV HX JY KT LQ NR PP
II I IV C GVO AB CS DO EH FQ GG IP KU LW MV XZ
III I IV C HND AD BB EP FG HY JK LL MS NN QT RR VX
III I V C UDT AA BB CV DR ES FG HX JO LZ MP QQ YY
III V I C QLU AX BW CR DO EJ FV GS HY LL MP QQ
IV II I C FPA AR BB CS DY EP FX HJ LO MM QW VZ
IV III II C EVR AH BO DS EL FF GM IP KK QU VW XZ
IV III II C FWR AH BO DS EL FF GM IP KK QU VW XZ
IV V II C WQM AM BV DZ EE FF GR HW JX LQ OO PY SS
V III I C SSE AO BC DE FW GK HH IM LQ PS UU VZ
V III IV C GSJ AH BV CJ DD EQ FX GM LP OW RR SY
V IV III C NWK AQ BT CF DM EP GL HH JS KV NX RY
```

Figura 66. Resultado de todas las paradas de la *Bombe* sin contradicciones

3.6.2 Fichero encriptado y *crib*

El fichero encriptado con el que vamos a trabajar va a ser el resultado de haber pasado por una máquina Enigma el fichero de texto que vimos en el apartado "**2.3.1.1 Fichero a encriptar/desencriptar**", es decir, el libro "Las Indias Negras" de Julio Verne.

Nuestro *crib*, va a ser "AUTHORJULIOVERNE", ya que tal como mencionamos en ese mismo apartado, todos los textos iban a tener dentro de sus 50 primeras letras la palabra "AUTHOR" seguida del nombre y apellido del autor. Esto simula cómo empezaban los alemanes los textos encriptados (con el parte meteorológico).

3.6.3 Resultados de ejecución

En la Tabla 14 podemos observar el tiempo total de la ejecución secuencial y el tiempo de la parte que podemos paralelizar. Los tiempos son en el orden de segundos.

	Total	Paralelizable
Bombe secuencial	6262.85	6262.61

Tabla 14. Tiempo de ejecución total y paralelizable de la *Bombe*

Por lo tanto, podemos observar que el 99.996% del tiempo del programa es paralelizable.

3.7 Código cuda

Una vez visto el código secuencial, vemos que el cuerpo principal del código tenía muchos bucles, por lo tanto, podíamos paralelizarlo. En total, tenemos las siguientes iteraciones:

2x bucle reflector
5x bucle slow
5x bucle middle
5x bucle fast
50x bucle p
26x bucle s
26x bucle m
26x bucle f

Por lo tanto, 219,700,000 iteraciones en el peor caso. Recordemos que en el bucle p, se veía si había letras duplicadas, por lo que sí existen estas, nos ahorraremos 26^3 iteraciones de los bucles “s”, “m”, y “f”. Si queremos que cada iteración la ejecute un thread de la GPU vemos que hay iteraciones suficientes para ello, 30720 threads por *device*. Es por eso, que he decidido dejar fuera de la paralelización el bucle del reflector y los tres bucles de los rotores (seguimos teniendo como máximo 878,800 iteraciones por delante para computar.

3.7.1 Envío de información al *device*

En este código, el *device* va a necesitar que se le envíe la misma información que enviamos en el código cuda para la Enigma, puesto que la máquina *Bombe*, no es más que diversas Enigmas, Figura 67. Además, va a necesitar también el trozo de texto (“d_DataIn2”) que suponemos que va a estar dentro del texto encriptado y un espacio para almacenar nuestra hipótesis si la iteración de la *Bombe* ha dado contradicción o no (“d_output”).

```
cudaMalloc((char **) &d_Plug, 26);
cudaMalloc((char **) &d_rFast, 26);
cudaMalloc((char **) &d_rMiddle, 26);
cudaMalloc((char **) &d_rSlow, 26);
cudaMalloc((char **) &d_rVFast, 26);
cudaMalloc((char **) &d_rVMiddle, 26);
cudaMalloc((char **) &d_rVSlow, 26);
cudaMalloc((char **) &d_Volta, 26);
cudaMalloc((char **) &d_output, 26*26*26*50);
cudaMalloc((char **) &d_DataIn, strlen(DataIn));
cudaMalloc((char **) &d_DataIn2, 100);

 cudaMemcpy(d_Plug, &Plug, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rFast, &rFast, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rMiddle, &rMiddle, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rSlow, &rSlow, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rVFast, &rVFast, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rVMiddle, &rVMiddle, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_rVSlow, &rVSlow, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_Volta, &Volta, 26, cudaMemcpyHostToDevice);
cudaMemcpy(d_DataIn, DataIn, strlen(DataIn), cudaMemcpyHostToDevice);
cudaMemcpy(d_DataIn2, DataIn2, 100, cudaMemcpyHostToDevice);
```

Figura 67. Asignación de memoria en el *device* y transferencia de información entre GPU-CPU

3.7.2 Distribución del trabajo

He analizado 4 combinaciones de distribuciones de trabajo hasta toparme con una que ralentizase mucho el tiempo de ejecución respecto a las otras. Estas combinaciones son las que vamos a ver a continuación (no van a tener una explicación detallada, dado que es el mismo procedimiento que hemos visto tanto en el apartado "**2.5.3.2 Distribución del trabajo"** como en "**2.5.4.2 Distribución del trabajo**".

Importante recordar que aquí el tamaño del programa va a ser 26^3*50 (878,800), tal y como mencionamos al final del apartado "**3.7 Código cuda**", ya que dejábamos fuera el bucle del reflector y los 3 bucles de los rotores. Por lo tanto, estas iteraciones deberán ser multiplicadas por 2*26^3.

3.7.2.1 128 *threads* por bloque

Tenemos 6,866 bloques de 128 *threads* cada uno., 16 bloques por SM y por lo tanto, 240 bloques de 256 *threads* trabajando a la vez (30,720 *threads*).

3.7.2.2 256 *threads* por bloque

Tenemos 3,433 bloques de 256 *threads* cada uno, 8 bloques por SM y por lo tanto, 120 bloques de 256 *threads* trabajando a la vez (30,720 *threads*).

3.7.2.3 512 *threads* por bloque

Tenemos 1,717 bloques de 512 *threads* cada uno, 4 bloques por SM y por lo tanto, 60 bloques de 512 *threads* trabajando a la vez (30,720 *threads*).

3.7.2.4 1024 *threads* por bloque

Tenemos 859 bloques de 1024 *threads* cada uno, 2 bloques por SM y por lo tanto, 30 bloques de 1024 *threads* trabajando a la vez (30,720 *threads*).

3.7.3 Kernel

Cada *thread* debe saber cuál es la iteración de la *Bombe* que tiene que ejecutar. Esto lo podemos ver en la Figura 68. Lo primero es ver si nuestro thread está dentro del número de iteraciones que mandamos a paralelizar (como vimos, 878,800).

```
int pos = blockIdx.x * blockDim.x + threadIdx.x;

if (pos < 17576*50) {

f = pos % 26;
m = (pos / 26) % 26;
s = (pos / (26*26)) % 26;
p = (pos / (26*26*26)) % 50;
```

Figura 68. Código de la posición a ejecutar por el *thread*

Si el thread tiene asignada una posición correcta dentro de la ejecución del *device,* hemos de mirar cual es la iteración de la *Bombe* que le corresponde. Por ejemplo, cogemos la distribución de trabajo de 128 threads por bloque y tenemos el *thread* 73 del bloque 20. Pos sería 20 * 128 + 73 (posición 2,633). Por lo tanto, estaríamos mirando la iteración de la *Bombe* con:

- Rotor rápido en la posición 7 (letra “H”)
- Rotor del medio en la posición 23 (letra “X”)
- Rotor lento en la posición 3 (letra “D”)
- Posición 0 del texto encriptado

Una vez el *thread* sabe con qué iteración de la *Bombe* tiene que trabajar, este ejecutará el mismo código que se describió en el apartado “**3.6 Código secuencial**”. Si su iteración no ha llegado a ninguna contradicción, este guardará en el espacio que le mandamos al *device* la hipótesis para la que no se ha cumplido ninguna contradicción. Si por el contrario, si ha ocurrido contradicciones, pondremos en esa posición un “[“ (podría ser cualquier carácter < “A” o > “Z”).

Fuera del kernel, el programa revisará este espacio de direcciones y para cada una de sus posiciones, verá si el elemento pertenece a “A”-”Z” o distinto, Figura 69. Si lo es, lo ignorará. Por el contrario, ejecutará una iteración de la *Bombe* con esa hipótesis e imprimirá el resultado.

```
for (int p=0; p<50; p++)
  for (int s=0; s<26; s++)
    for (int m=0; m<26; m++)
      for (int f=0; f<26; f++)
        if (output[p*26*26*26+s*26*26+m*26+f] >= 'A' && output[p*26*26*26+s*26*26+m*26+f] <= 'Z')
```

Figura 69. Código que valida si ha habido contradicción o no

3.7.3.1 Resultados de ejecución

El tiempo de ejecución para la máquina *Bombe* y para las distintas distribuciones de trabajo, así como para la ejecución secuencial se muestra en la Tabla 15. Los tiempos son en el orden de segundos.

			Secuencial	128 th/bl	256 th/bl	512 th/bl	1024 th/bl
Bombe	Cuda	Total	6 262.61	254.74	255.43	256.61	359.58
		Kernel	No aplica	254.71	255.40	256.58	359.55

Tabla 15. Tiempo de cuda total y del kernel para cada distribución

Como vemos, conseguimos los siguientes *speedup*, Tabla 16, respecto al código secuencial.

			128 th/bl	256 th/bl	512 th/bl	1024 th/bl
Bombe	Cuda (Total)	*Speedup*	24.58	24.52	24.41	17.42

Tabla 16. *Speedup* de cuda total respecto al código secuencial

Como siempre, hemos de analizar el comportamiento del código en cuda para ver si estos resultados son buenos o pueden obtenerse de mejores. Si nos basamos en los tres objetivos que marcamos para toda ejecución en cuda del apartado "**2.5.2 Objetivos de una ejecución en cuda**", solo conseguimos el primero, una buena distribución de los *threads* (excepto para la distribución de 1024 *threads* por bloque), haciendo que todos los del *device* tengan trabajo que hacer.

El problema principal de este código es que como se ha querido hacer lo más real a como funcionaba la máquina *Bombe*, no es un programa pensado para ejecutarse en cuda. Vemos que hay mejora respecto al código secuencial (para nada la que se suelen conseguir con buenos programas en cuda) pero es simplemente porque tienes a miles de *threads* ejecutando un código a diferencia de solo uno en la ejecución en secuencial. Esto se ve en que hay una gran divergencia en el camino de los *threads,* ya que el programador no sabe cuánto trabajo va a hacer cada uno, ya que se trata de conseguir contradicciones que pueden ser en la iteración 1, en la 2, en la 20, …

Por lo tanto, si hay divergencia de *threads*, los accesos a memoria tampoco van a ser óptimos ya que van a ser en desorden. Es decir, no vamos a poder ocultar la latencia a memoria, cosa que vimos anteriormente que nos daba muy buen rendimiento en los programas.

En la Figura 70 se muestra la salida del *profiling* de nvidia para ver como son los accesos a memoria. Se ha hecho la ejecución de solo 1 posición de rotores, ya que es indiferente, puesto que todo el programa se comporta de la misma forma.

```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: kernel(char, char, char, char*, char*, int, char*, char*, char*, char*, char*, char*, char*, char*, char*)
          1                            gld_efficiency             Global Memory Load Efficiency      5.39%       5.39%       5.39%
          1                      gld_requested_throughput          Requested Global Load Throughput  2.0959GB/s  2.0959GB/s  2.0959GB/s
          1                            gld_throughput                    Global Load Throughput  38.858GB/s  38.858GB/s  38.858GB/s
          1                gld_transactions_per_request      Global Load Transactions Per Request    1.000000    1.000000    1.000000
          1                            gld_transactions                  Global Load Transactions  2775939324  2775939324  2775939324

          1                            gst_efficiency            Global Memory Store Efficiency      68.32%      68.32%      68.32%
          1                      gst_requested_throughput         Requested Global Store Throughput  266.04KB/s  266.04KB/s  266.04KB/s
          1                            gst_throughput                    Global Store Throughput  389.42KB/s  389.42KB/s  389.42KB/s
          1                gst_transactions_per_request     Global Store Transactions Per Request    1.000000    1.000000    1.000000
          1                            gst_transactions                  Global Store Transactions       26531       26531       26531
```

Figura 70. Nvprof con métricas de memoria para ejecución 128 th/bl

Como vemos, la eficiencia de las lecturas es muy baja, 5.39%. Sin embargo, vemos que se ejecuta una transacción de lectura por petición y esto es lo que siempre queremos obtener. Si hacemos unos cálculos (sabiendo que el *kernel* para solo esta posición de rotores tarda 2.122 segundos) tenemos que se piden 1,72 bytes por petición. Si rescatamos que se gestiona una transacción por petición, queda demostrado que hay divergencia en los *threads* y que no se accede a memoria paralelamente.

A partir de ahora ignoraremos la ejecución con la distribución de 1024 *threads*/bloque debido a que vemos que es a partir de aquí en que el rendimiento cae.

3.7.4 Kernel con shared memory

Una cosa que tenemos clara y segura es que cada *thread* va a tener que trabajar con los dos textos que le hemos pasado al *device*: nuestro *crib* y la parte del texto encriptado. Por lo tanto, sabemos que por lo menos, en este aspecto nos vamos a beneficiar si utilizamos *shared memory*.

Además, vamos a beneficiarnos de traernos a la *shared memory* los vectores de los rotores y el reflector, tal y como hicimos en el apartado "**2.5.3.4 Kernel con shared memory**" del cálculo de la matriz Encriptador.

Para ello, vamos a inicializar nuevos vectores, Figura 71, indicando que van a formar parte de la *shared memory*.

```
__shared__ char sDataIn[100];
__shared__ char sDataIn2[100];
__shared__ char sPlug[26];
__shared__ char srFast[26];
__shared__ char srMiddle[26];
__shared__ char srSlow[26];
__shared__ char srVFast[26];
__shared__ char srVMiddle[26];
__shared__ char srVSlow[26];
__shared__ char sVolta[26];
```

Figura 71. Inicialización de la *shared memory*

Para cargar los dos textos, vamos a utilizar los primeros 100 *threads* de cada bloque y para la resta de vectores, los 26 primeros, Figura 72. Por último, se sincronizan los *threads* del bloque para que ninguno empiece su ejecución sin tener los vectores de la *shared memory* completamente cargados.

```
if (threadIdx.x < 100) {

    sDataIn[threadIdx.x] = DataIn[threadIdx.x];
    sDataIn2[threadIdx.x] = DataIn2[threadIdx.x];

}

if (threadIdx.x < 26) {

  sPlug[threadIdx.x] = Plug[threadIdx.x];
  srFast[threadIdx.x] = rFast[threadIdx.x];
  srMiddle[threadIdx.x] = rMiddle[threadIdx.x];
  srSlow[threadIdx.x] = rSlow[threadIdx.x];
  srVFast[threadIdx.x] = rVFast[threadIdx.x];
  srVMiddle[threadIdx.x] = rVMiddle[threadIdx.x];
  srVSlow[threadIdx.x] = rVSlow[threadIdx.x];
  sVolta[threadIdx.x] = Volta[threadIdx.x];

}

__syncthreads();
```

Figura 72. Escrituras en la *shared memory* dependiendo del id del *thread*

3.7.4.1 Resultados de ejecución

En la Tabla 17 vamos a ver un resumen de como quedan los nuevos tiempos de ejecución, en segundos, para cada distribución de trabajo. Como estoy mostrando en segundos, la parte restante del código que se ejecuta en la CPU no la muestro, ya que es del orden de muy pocos milisegundos y no muestra relevancia en el resultado de los tiempos.

			Secuencial	128 th/bl (no *shared)*	128 th/bl (*shared)*	256 th/bl *(shared)*	512 th/bl *(shared)*
Bombe	Cuda	Total	6 262.61	254.74	176.89	178.01	177.44
		Kernel	No aplica	254.71	176.86	177.97	177.41

Tabla 17. Tiempo de cuda total y del kernel para cada distribución con *shared memory*

Por lo que los *speedup* de la distribución de 128 *threads*/bloque con *shared memory* respecto a la ejecución secuencial a la mejor distribución sin *shared memory* (128 *threads* por bloque) van a quedar como se muestra en la Tabla 18.

	Secuencial	128 th/bl (no *shared)*

Bombe	Cuda (total)	*Speedup*	35.40	1.44

Tabla 18. *Speedup* de cuda total respecto al código secuencial y a la mejor distribución sin *shared memory*

Si para la ejecución sin *shared memory* vimos que la eficiencia de las lecturas eran muy baja, la eficiencia en la *shared memory* lo va a ser más. Esto es, porque seguimos con la misma divergencia de los *threads*, por lo tanto, seguiremos no accediendo paralelamente a la *shared* (lectura de 1 *byte)* y esta, como ya vimos en el apartado "**2.5.3.4.1 Resultados de ejecución",** nos va a entregar 256 *bytes* por transacción. Veámoslo igualmente en la Figura 73.

```
Invocations                               Metric Name                               Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: kernelShared(char, char, char, char*, char*, int, char*, char*, char*, char*, char*, char*, char*, char*)
          1                         shared_efficiency                         Shared Memory Efficiency      2.29%       2.29%       2.29%
          1                    shared_load_throughput                    Shared Memory Load Throughput  450.43GB/s  450.43GB/s  450.43GB/s
          1                   shared_load_transactions                   Shared Load Transactions  2776197247  2776197247  2776197247
          1      shared_load_transactions_per_request    Shared Memory Load Transactions Per Request    1.000093    1.000093    1.000093

          1                   shared_store_throughput                   Shared Memory Store Throughput  18.252MB/s  18.252MB/s  18.252MB/s
          1                  shared_store_transactions                  Shared Store Transactions      109860      109860      109860
          1     shared_store_transactions_per_request   Shared Memory Store Transactions Per Request    1.000036    1.000036    1.000036
```

Figura 73. Nvprof con métricas de *shared memory* para la ejecución de 128 th/bl

Como dijimos, sí que aprovechamos la *shared memory,* pero no obtenemos tan buenos resultados en comparación a si los accesos a memoria fueran óptimos desde un principio. Estos tipos de código no se pueden analizar previamente al 100%, ya que como dijimos, al haber divergencia en los *threads*, no puedes saber el comportamiento de estos, y esto, en la mayoría de casos es penalizante en cuanto a rendimiento.

3.7.5 MultiGPU + *shared memory*

Como siempre, miremos primero si el trabajo que hay total (*threads* trabajando a la vez en el *device*) nos llega por lo menos para que cargar una GPU al 100%. Si revisamos el apartado "**3.7.2 Distribución del trabajo**", todas las distribuciones tienen los 30,720 *threads* trabajando y les sobran bloques para la otra GPU.

3.7.5.1 Envío de información a los *devices*

Como ya hemos visto anteriormente, lo primero va a ser asignar memoria no paginada en el *host,* Figura 74.

```
cudaMallocHost((char**)&Plug, 26);
cudaMallocHost((char**)&rFast, 26);
cudaMallocHost((char**)&rMiddle, 26);
cudaMallocHost((char**)&rSlow, 26);
cudaMallocHost((char**)&rVFast, 26);
cudaMallocHost((char**)&rVMiddle, 26);
cudaMallocHost((char**)&rVSlow, 26);
cudaMallocHost((char**)&Volta, 26);
cudaMallocHost((char**)&output, 26*26*26*50);
cudaMallocHost((char**)&DataIn, strlen(DataInAux));
cudaMallocHost((char**)&DataIn2, strlen(DataIn2Aux));
```

Figura 74. Asignación no paginada de memoria, *pinned memory*

Y por último, hacer las transferencias entre GPU-CPU asíncronas, Figura 75. Recordemos que estas transferencias se harán dos veces, una por GPU.

```
cudaMemcpyAsync(&d_Plug[0], &Plug[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rFast[0], &rFast[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rMiddle[0], &rMiddle[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rSlow[0], &rSlow[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rVFast[0], &rVFast[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rVMiddle[0], &rVMiddle[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_rVSlow[0], &rVSlow[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_Volta[0], &Volta[0], 26, cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_DataIn[0], &DataIn[0], strlen(DataIn), cudaMemcpyHostToDevice);
cudaMemcpyAsync(&d_DataIn2[0], &DataIn2[0], 100, cudaMemcpyHostToDevice);
```

Figura 75. Transferencia asíncronas de datos entre GPU-CPU

3.7.5.2 Resultados de ejecución

Los resultados de la ejecución de la *Bombe* para las diferentes distribuciones se observa en la Tabla 19. Los tiempos están en el orden de segundos.

		Secuencial	128 th/bl (*shared)*	128 th/bl (2 GPUs)	256 th/bl (2 GPUs)	512 th/bl (2 GPUs)
Bombe	Cuda (total)	6 262.61	176.89	74.56	75.01	74.82

Tabla 19. Tiempo de cuda total para cada distribución con 2 GPUs

Por lo tanto, el *speedup* que obtenemos respecto a la ejecución secuencial a la mejor distribución con *shared memory* para 128 *threads* por bloque con 2 GPUs es la que se muestra en la Tabla 20.

			Secuencial	128 th/bl (*shared)*
Bombe	Cuda (total)	*Speedup*	83.99	2.37

Tabla 20. *Speedup* de cuda total respecto al código secuencial y a la mejor distribución con *shared memory*

Como vemos, superamos el 2x de *speedup* al usar el doble de GPUs. Esto es por lo mismo que ya hemos visto anteriormente, nos beneficiamos tanto de la *pinned memory,* que es una asignación de memoria más rápida y de las transmisiones de información asíncronas.

3.8 Resultado de ejecución final

Si hacemos una tabla resumen con la progresión que ha ido adquiriendo el código secuencial a lo largo de los distintos apartados, obtenemos lo que podemos ver en la Tabla 21. Tiempos en el orden de segundos.

		Secuencial	2 GPUs
Bombe	Cuda (Total)	6 262.61	74.56
	Total	6 262.85	74.80

Tabla 21. Comparación de tiempos de cuda total y total entre código secuencial y 2 GPUs

Por lo que el *speedup* final del código con dos GPUs respecto a la ejecución en secuencial es de 83.72.

4. Máquina check

Que la *Bombe* haya parado y por lo tanto, el operario haya apuntado el resultado no significa que este resultado sea el definitivo. Como dijimos, es un *plugboard* que no ha obtenido contradicciones para las letras descubiertas, pero si nuestro *crib* no es lo suficientemente complejo, en cuanto a número de letras que cubre, puede que este *plugboard* no sea correcto.

4.1 Combinaciones restantes

Recordemos que para crackear la máquina Enigma, antes de ejecutar las iteraciones con la máquina *Bombe*, la Enigma tenía un total de 150,738,274,937,250 combinaciones posibles en su configuración inicial, tal y como vimos en el apartado "**3.1 Complejidad de la Enigma**".

4.1.1 Completar el *plugboard*

En el código de la *Bombe* limitamos las salidas a que sean solo generadas las que generen como mínimo 8 parejas de *plugboard* distintas (con distintas me refiero a una pareja de dos letras distintas). Por lo tanto, las combinaciones restantes para cada salida de la *Bombe* estarán determinadas por este número de parejas distintas. Cuantas más haya descubierto, menos iteraciones habrá que ejecutar.

En la Figura 76 muestro las tres primeras líneas de la Figura 66 del apartado "**3.6.1 Formato de salida**", para que sea más fácil llevar la explicación. Si nos damos cuenta, en el primer *output* se han descubierto 11 parejas de *plugboard* en total, de las cuales, 10 de ellas son parejas distintas. Al ser 10 el máximo de parejas distintas que puede tener un *plugboard*, solo hay una combinación posible, y es precisamente la ya generada. Recordemos que poner una pareja de mismas letras es como no poner nada, ya que no se hace ninguna conexión en el *plugboard*.

```
I II IV B VWD AU CD EX FZ HW IS KM LL PY QT RV
I V II B TSA BB CI DE FJ GQ HN LM PU TX VV YZ
I V III B ADO BB CX EE GR HY IM JQ KL NN PZ SV UU
```

Figura 76. *Output* reducido de la *Bombe*

En el segundo *output* se han generado 11 parejas, de las cuales 9 son distintas. Por lo tanto, quedará como máximo una pareja de letras distintas con 6 letras que aún están libres (A, K, O, R, S y W). Esto significa 6 combinaciones de mismas letras (AA, KK, OO …) y 15 combinaciones de letras distintas (AK, AO, AR …). Este último número sale de que tenemos que combinar 6 letras (6!). Como queremos hacer solo 1 pareja de dos, nos sobran combinaciones de 4 letras (6-2), dividimos por 4! Como el orden de esta pareja nos es indiferente, dividimos por 1! Por último, AB es lo mismo que BA, por lo que dividimos por 2^1 (dos elevado al número de parejas que queremos hacer). Por lo tanto: 6! / 4!2. Si lo sumamos a las 6 combinaciones y a la que ya hay, para este *output* de la *Bombe* hay que comprobar 22 combinaciones totales.

Para el tercer *output*, se han generado 12 parejas de *plugboard*, de las cuales 8 son distintas. Por lo tanto, quedarán dos parejas distintas como máximo con las 6 letras que aún están libres (A, D, O, T, U y W). Por lo tanto, en este caso hay tres opciones de *plugboard*:

- El *plugboard* que sale directamente del *output* de la *Bombe*.
- El *plugboard* de 9 parejas distintas (hemos de generar una pareja más), por lo tanto las mismas combinación que en el caso anterior (segundo *output*).
- El *plugboard de* 10 parejas distintas (hemos de generar dos parejas más), por lo tanto el cálculo será: 6! / 2!2!2^2, es decir, 45 combinaciones

Por lo tanto, para este tercer *output* tenemos 1 + 21 + 45 combinaciones totales (67).

Si por ejemplo, nos saliese un *output* con 11 parejas de letras distintas, esto no es posible, ya que el máximo son 10. Por lo tanto, este ni se comprueba y se descarta.

4.1.2 Posición inicial de los rotores

Recordemos que la posición de los rotores que anotaba el encargado solo era la posible candidata a ser cierta si la primera posición de nuestro *crib* coincidía con la primera posición del texto encriptado. Si no era así, se debía de tirar atrás tantas posiciones como la primera posición de nuestro *crib* respecto a la del texto encriptado.

Si rescatamos el *crib* que teníamos en la Figura 54 del apartado "**3.4 Puesta a punto de la *Bombe***", vemos que el *crib* coincide con la primera posición del texto encriptado, puesto que si por ejemplo, la *Bombe* ha parado en la posición de los rotores "JRS", es con esta posición inicial con la que hay que trabajar.

Sin embargo, si rescatamos el *crib* de la Figura 52 del apartado "**3.2 Solución teórica**", vimos que hicimos desplazamientos, por lo que la posición "QFS" habrá que desplazarla las mismas posiciones hacia atrás. Esto no puede parecer un gran problema a simple vista, pero si recordamos que existen ocasiones en que el rotor de el medio puede hacer doble giro, se nos abren más caminos de posibilidades para llegar a una posición en concreto.

Si analizamos la posición "QFS", podemos suponer que la posición anterior es "QFR", solo ha girado el rotor rápido una posición, como hace siempre. Sin embargo, si estamos en la posición "PER", y el rotor de el medio es el II, la "E" es la muesca de este, por lo tanto, habrá doble giro de rotor: "QFS". Al estar el rotor del medio en su muesca, giran tanto él como el rotor lento.

4.2 Máquina *check* histórica

Se trataba de una máquina parecida a una máquina Enigma con la diferencia de que no tenía el mecanismo de giro de los rotores, sino que era manual. Esto servía para ir cambiando los giros de estos de forma más fácil.

Al operario encargado de esta máquina se le dejaba lo que había generado la *Bombe*. Este, tenía que ir desencriptando el mensaje y cuando llegaba a conjuntos de letras que no tenían sentido, debía hacer un cambio en el *plugboard* (apartado "**4.1.1 Completar el *plugboard***") y/o un cambio en las posiciones iniciales de los rotores (apartado **4.1.2 Posición de los rotores**").

Podría darse la ocasión de que cambiase lo que cambiase no se llegaba a entender nada, por lo tanto ese *output* de la *Bombe* lo descartaban.

4.3 Nuestra máquina *check*

Como nuestra máquina *Bombe* sí tiene en cuenta el recorrido de los rotores hacía atrás, nuestra máquina *check* solo va a tener que comprobar las combinaciones restantes del *plugboard* (la posición inicial siempre será correcta). Para comprobar estas combinaciones, pensamos en dos formas distintas: una que se acercaba a lo que hacían los ingleses, y la segunda una más automática.

Los ingleses, como he comentado, trataban de ir mirando si se iba entendiendo el texto que iban desencriptando o no. Como nuestros textos son en castellano, decidimos simular esto contabilizando el número de "QUE" que se generaban en el texto desencriptado. Si llegaba a un porcentaje mínimo respecto a las letras totales, dábamos por posiblemente buena esa combinación, y pasaría a la siguiente fase. Esta fase sería poner esta combinación en una máquina Enigma y lo generado, hacer un comando "*diff"* con el texto original.

Este porcentaje mínimo que he mencionado es del 0.7%. Se hicieron pruebas con 31 libros de distintas longitudes que siguieran el mismo formato de este proyecto y salió que el mínimo de "QUE" en uno de los libros era del 0.73%.

El hecho de que nos basáramos en la palabra "QUE" viene dada a que mediante un histograma de varios textos largos, vimos que era la combinación de tres letras que más aparecía en castellano, alejándose bastante de la segunda, "ENT".

La segunda forma que se nos ocurrió, era para cada combinación generar un desencriptado y compararlo con un comando "*diff*" con el texto original. Esto nos daría automáticamente si la combinación es válida o no. Sin embargo, dejamos de simular el hecho de que tenían que ir entendiendo el texto que iban desencriptando.

4.4 Código secuencial

Nuestro programa partirá de leer el *output* que la *Bombe* ha generado y ha escrito en un fichero de tipo texto. Por ejemplo, si nos fijamos en el trozo de *output* que hay en la Figura 76 del apartado "**4.1.1 Completar el *plugboard***", veremos que de la primera línea va a aprovechar todo excepto el número que está entre la posición inicial de los rotores y la primera pareja del *plugboard*, ya que este número es indiferente para este código.

La base principal de este código es saber cuánto es el número de parejas de distintas letras que le queda al *plugboard* del *output*, ya que cuantas menos parejas le queden menos iterará. Por lo tanto, cada vez que leamos de la línea del *output* una pareja de distintas letras incrementaremos un contador que al final, lo acabaremos restando de 10 (máximo de parejas).

Otro dato importante que necesitamos es qué letras son las que están libres. Esto lo haremos mediante un vector que hemos llamado "libres" el cual, tiene 26 elementos. Este vector se encarga de dejar a 0 el elemento cuya posición sea igual a la letra que está en una pareja del *plugboard* (esto se hace dos veces por pareja, ya que son letras distintas) y solo una vez para parejas de letras iguales. Un pequeño ejemplo está mostrado en la Figura 77, donde primero vemos que hacemos el conexionado de la pareja con la función "Clavar", marcamos estas dos letras como "ya no son libres" y aumentamos conforme es una pareja cogida.

```
Clavar(str[0], str[1]);
libres[str[0] - 'A'] = libres[str[1] - 'A'] = 0;
if (str[0] != str[1]) {
  ++parejasCogidas;
}
```

Figura 77. Código para calcular el número de parejas del *plugboard*

A partir de aquí, nos falta generar todo el número posible de combinaciones de parejas de letras distintas que nos quedan, de como máximo 10. Como dijimos, en la *Bombe* hicimos que solo mostrara *outputs* que tenían como mínimo 8 parejas de letras distintas. El cuerpo de los bucles para las combinaciones se muestra en la Figura 778.

```
if (parejasRestantes > 0) {
  for (int i=0; i<letrasLibres; i++) {
    for (int j=0; j<letrasLibres; j++) {
      if (i != j && actually_checked[i][j] == 0 && actually_checked[j][i] == 0) {
        actually_checked[i][j] = actually_checked[j][i] = 1;

        //código de una pareja aquí

        if (parejasRestantes > 1) {
          for (int k=0; k<letrasLibres; k++) {
            if (k != i && k != j) {
              for (int l=0; l<letrasLibres; l++) {
                if (l != i && l != j && l != k && actually_checked2[i][j][k][l] == 0 && actually_checked2[i][j][l][k]
== 0 && actually_checked2[j][i][k][l] == 0 && actually_checked2[j][i][l][k] == 0 && actually_checked2[k][l][i][j] == 0
&& actually_checked2[k][l][j][i] == 0 && actually_checked2[l][k][i][j] == 0 && actually_checked2[l][k][j][i] == 0) {
                  actually_checked2[i][j][k][l] = actually_checked2[i][j][l][k] = actually_checked2[j][i][k][l] =
actually_checked2[j][i][l][k] = actually_checked2[k][l][i][j] = actually_checked2[k][l][j][i] = actually_checked2[l][k]-
[i][j] = actually_checked2[l][k][j][i] = 1;

                  //código de dos parejas aquí
```

Figura 78. Bucle del código para generar las combinaciones restantes

La matriz "*actually_checked*" graba las combinaciones de una pareja que ya se han generado para no volver a generarlas, y la matriz "*actually_checked2*" graba las combinaciones de dos parejas. Si nos volvemos a fijar en la Figura X, podemos ver que para cada combinación de una pareja, la probamos y generamos las combinaciones restantes de dos parejas con esta y las probamos.

Fuera de esta sentencia "*if (parejasRestantes > 0)"* tendremos que probar una combinación extra que es la que se genera en el *output*, es decir, sin crear ningún conexionado más en el *plugboard*. Siempre y cuando, "parejasRestantes" sea igual a 0, ya que 10 es el máximo que pueden haber. Recordemos que "parejasRestantes" lo obteníamos de restar 10 a "parejasCogidas".

Para cada combinación generada, se operará como una Enigma, es decir, con las mismas matrices Encriptador, Camino y Ruta que vimos en el apartado "**2.4 Código optimizado**". Se guardará la desencriptación del texto encriptado en un espacio, "*DataOut*" y contabilizaremos el número de "QUE" que se han generado, tal y como comentamos en el apartado anterior "**4.3 Nuestra máquina *check***", Figura 79.

```
for (int i=0; i<strlen(DataOut)-2; i++)
  if (DataOut[i] == 'Q' && DataOut[i+1] == 'U' && DataOut[i+2] == 'E')
    ++countPalabra;
```

Figura 79. Código que contabiliza en número de aparición de "QUE"

Por último, solo quedará ver si el número de veces que aparece la palabra "QUE" pertenece al 0.7% del total de extensión del archivo desencriptado, Figura 80.

```
double histo = countPalabra * 100.0 / strlen(DataIn);
if (histo >= 0.7) {
  //impresión del resultado
```

Figura 80. Código que mira si se llega al límite de “QUE”

4.4.1 Formato de salida

El *output* de nuestra máquina *check* nos mostrará un listado de las combinaciones que han dado un porcentaje mayor o igual al 0.7 contando el número de “QUE” que generan al desencriptar el mensaje. Para el ejemplo de la Figura 66 del apartado “**3.6.1 Formato de salida**”, el resultado lo vemos en la Figura 81.

```
II IV I B ZIQ AZ BY CX EV FU GT HS IR JQ
II IV I B ZIQ AZ BY CX DW EV FU GT HS IR JQ
II IV I B ZIQ AZ BY DW EV FU GT HS IR JQ
II IV I B ZJQ AZ BY CX EV FU GT HS IR JQ
II IV I B ZJQ AZ BY CX DW EV FU GT HS IR JQ
II IV I B ZJQ AZ BY DW EV FU GT HS IR JQ
```

Figura 81. *Output* de la máquina *Check*

4.4.2 Resultados de ejecución

Para los resultados que vamos a ver a continuación, hemos escogido como *output* de la *Bombe* el mismo que se ve en la Figura 66 del apartado “**3.6.1 Formato de salida**”. Como en este código utilizamos la misma funcionalidad de la Enigma que vimos en el apartado “**2.4 Código optimizado**”, sabemos que vamos a poder paralelizar el guardado en la matriz Encriptador y el acceso a Encriptador, Tabla 22. El tiempo es del orden de segundos.

	Total	Encriptador	Acceso a Encriptador
Check	671.60	9.37	662.09

Tabla 22. Tiempo de ejecución del *check* secuencial

Viendo estos resultados, observamos que un 99.98% del tiempo de ejecución pertenece a partes que pueden ser paralelizadas, por lo tanto, este código es altamente paralelizable.

Además, sabemos cómo se van a comportar las dos partes al paralelizarlas, ya que las vimos anteriormente.

Sabiendo que el acceso a Encriptador es altamente paralelizable y que es un 98.60% de la ejecución paralela, podemos predecir que este código va a obtener un alto speedup en cuanto lo paralelizemos.

4.5 Código cuda

Para transformar nuestro código secuencial en paralelizable mediante cuda, es lo mismo que ya vimos en el apartado "**2.5 Código cuda**", por lo que no explicaremos en detalle cómo se ha realizado, así como las diferentes distribuciones de trabajo que pueden verse en ese mismo apartado.

Sin embargo, sí que crearemos dos programas distintos: uno con *shared memory* y otro con 2 GPUs + *shared memory*, tal y como vimos.

4.5.1 *Kernel* con *shared memory*

4.5.1.1 Resultados de ejecución

En la Tabla 22 podemos observar los tiempos de ejecución del programa paralelizado , es decir, matriz Encriptador y el acceso a esta paralelizados en cuda. Para la ejecución de la matriz Encriptador, se utiliza la distribución de 512 *threads* por bloque, ya que fue la mejores resultados nos dio. Para el acceso a dicha matriz, utilizaremos la distribución de 128 *threads* por bloque por la misma razón. Muestra de 50 ejecuciones y tiempos en segundos.

			Secuencial Encriptador	Secuencial Acceso	Paralelizado Encriptador	Paralelizado Acceso
Check	Media	Cuda	9.37	662.09	0.362	0.342

Tabla 23. Tiempo de ejecución del *check* secuencial y del *check shared memory* en cuda

En la Tabla 24 podemos observar el *speedup* que se obtiene respecto a la ejecución en secuencial del cálculo de la matriz Encriptador, del acceso a esta y del total de ambos.

			Encriptador	Acceso	Total
Check	Media	*Speedup* cuda	25.88	1 935.93	953.78

Tabla 24. *Speedup* del *check* con *shared memory* respecto al secuencial

4.5.2 MultiGPU + *shared memory*

Anteriormente vimos que si podíamos utilizar dos GPUs para la ejecución de este código, ya que tanto en el cálculo de la matriz Encriptador como en su acceso, por lo menos una de las GPUs utilizaba todos sus *threads*.

4.5.2.1 Resultados de ejecución

En la Tabla 25, podemos observar los tiempos de ejecución del programa paralelizado ejecutándolo con dos GPUs. Para el cálculo de la matriz Encriptador se utiliza la distribución de 128 th/bl, y para el acceso a ella la misma. Lo comparamos con los tiempos obtenidos anteriormente de la versión solo con *shared memory.* Muestra de 50 ejecuciones y tiempo en segundos.

			Paralelizado Encriptador	Paralelizado Acceso	2 GPUs Encriptador	2 GPUs Acceso
Check	Media	Cuda	0.362	0.342	0.072	0.073

Tabla 25. Tiempo de ejecución del *check shared memory* y del *check* 2 GPUs en cuda

Por lo que obtenemos, Tabla 26, los siguientes *speedup* totales de la parte de cuda para nuestra ejecución de 2 GPUs respecto al secuencial y a la versión solo con *shared memory*.

			Secuencial	Paralelizado
Check	Media	*Speedup* cuda	4 630.76	4.85

Tabla 26. *Speedup* total de cuda por parte de 2 GPUs respecto al código secuencial y paralelizado

En esta tabla podemos ver el alto rendimiento que se consigue al asignar memoria no paginada y realizar transmisiones de datos de forma asíncrona. Visto fríamente, prácticamente es como si estuviéramos ejecutando el código con 5 GPUs.

4.6 Resultado de ejecución final

En el apartado "**4.4.2 Resultados de ejecución**", vimos que la parte que no se podía paralelizar era de un 0.02% del total del tiempo de ejecución del código. Esto son 0.14 segundos, que se compartirán con ambas versiones que hemos creado del código en cuda. Por lo tanto, si sumamos dicho tiempo a cada ejecución obtendremos la tabla resumen siguiente, Tabla 27.

		Secuencial	*Shared Memory*	2 GPUs
Check	Total	671.60	0.844	0.285

Tabla 27. Comparación de tiempos finales de las dos versiones del *check* en cuda respecto a la secuencial

5. *Bombe* + *check* con fuerza bruta

Después de ver lo que tardamos en sacar los posibles *plugboards* válidos con la *Bombe* y luego probarlos en la máquina *check* para ver si llegan a un número determinado de "QUE", nos preguntamos cuánto tardaríamos en sacar el mismo resultado si hiciéramos una ejecución mediante fuerza bruta. Es decir, lo que para en la época de la Segunda Guerra mundial era imposible y por eso, idearon el método de hipótesis y contradicciones, cómo se comportaría hoy en día.

De hecho, ya hemos visto un programa de fuerza bruta, el de la máquina *check*. Pasados unos rotores con una posición determinada, un reflector y un *plugboard* incompleto, probaba todas las combinaciones de *plugboard* estantes y mirábamos si se llegaba al 0.7% de "QUE" respecto al número de letras totales del texto desencriptado.

El código de fuerza bruta haría lo mismo pero con todas las combinaciones de rotores, posiciones de estos, posiciones iniciales, rotores y *plugboard* (desde 0 parejas hasta las 10 máximas).

Este código lo dejé incompleto, de hecho, solo medí tiempos para la generación máxima de dos parejas (de las 10 máximas totales) y una sola posición específica de rotores y de reflector. En este caso, hice la prueba con los rotores II-IV-I, el reflector "B" y la posición inicial "ZJQ". Pese a estar incompleto, se ha decidido mencionarlo y dedicarle un apartado,

ya que nos va a mostrar los beneficios que se pueden llegar a lograr evitando la fuerza bruta mediante otros métodos (si estos, son posibles).

5.1 Resultados de ejecución

Si computáramos solo una pareja del *plugboard* y rescatamos la fórmula que vimos en el apartado "**3.1 Complejidad de la Enigma**", vemos que obtenemos 325 combinaciones distintas de este (26!/(24!1!2^1)). Esto tardó 0.266 segundos.

Sin embargo, si computáramos para dos parejas del *plugboard,* tenemos que añadir a las combinaciones anteriores, 44 850 (26!(22!2!2^2)), por lo que ahora serían 45 175 combinaciones totales. Esto tardó 36.13 segundos.

En este punto vemos varias cosas:

- El tiempo es exponencial
- Para sola una pareja tenemos que se tarda de media 0.000696 segundos por combinación.
- Para dos parejas tenemos que se tarda de de media 0.000799 segundos por combinación.

Es decir, hay un tiempo extra que siempre se va a ir incrementando cada vez que añadamos parejas. Al no poder hacer una ejecución para todas las parejas, voy a coger de tiempo por combinación, 0.000696, para tirar al mínimo.

Si siguiéramos sumando parejas al código, llegaríamos a un total de tenerse que computar: 216 751 068 429 025 combinaciones. Por lo que tardaríamos 150 858 743 626.6 segundos. Esto son 4 783.69 días (como mínimo). Recordemos que esto era para solo una posición, si quisiéramos comprobar todas las posibles (5x4x3x26x26x26, 1 054 560), serían en total 228 577 006 722 512 604 000 combinaciones. Esto en días son 7 248 129 335 442.43.

6. Informe de GEP

6.1 Introducción

6.1.1 Identificación del problema

Para identificar el problema que existe, y que queremos resolver, tenemos que imaginarnos que estamos en la época de la segunda guerra mundial y que, sobre todo, pertenecemos al bando aliado.

Los británicos, como ya he comentado, han creado la máquina Bombe y han resuelto como crackear la máquina Enigma. El problema está, que cada día tienen que hacer un cálculo nuevo para crackearla, ya que la configuración de la máquina Enigma cambia diariamente. Este cálculo, tarda alrededor de 20 minutos en realizarse, por lo que sí tardásemos menos, conseguiríamos ser más proactivos en la guerra y por lo tanto, conseguir ventaja sobre el enemigo cuando son ellos los que se creen que por cifrar los mensajes y que se va a tardar mucho en descifrarlos, creen que la tienen.

Destacar también, que para que el tiempo de cálculo no supere los 20 minutos, hacen falta varias personas trabajando a una velocidad alta y teniendo un conocimiento también alto del tema en cuestión.

6.1.2 Actores implicados

Este proyecto va dirigido a todo aquél interesado en cualquiera de los dos temas que se tratan: todo lo relacionado con la segunda guerra mundial relacionándola con la máquina Enigma, y con todo lo relacionado con el cómputo de tarjetas gráficas mediante CUDA.

Además, puede llegar a ser de gran ayuda para cualquier persona no experimentada que esté trabajando en CUDA. Durante el proyecto, se va a ir mostrando y explicando cómo vamos a ir paralelizando un código mediante esta herramienta de programación, qué otras opciones podemos implementar (mejores o peores) y los resultados que vamos a obtener con todas ellas.

Por último, también puede ser también interesante para alguien que sí que esté experimentada en trabajar en CUDA, ya que puede estudiar la paralelización hecha y ver si puede aún mejorarla más.

6.2 Justificación

Tras hacer una búsqueda de diferentes trabajos de búsqueda relacionados con la máquina Enigma, la mayoría que he encontrado son más históricos que científicos. Es decir, se

centran más en todos los hechos históricos relacionados con el tema en lugar de trabajar y estudiar el caso de la máquina en sí.

Sin embargo, hay un trabajo de una ex-alumna de matemáticas de la universidad de Zaragoza, concretamente, Sandra Paño Badía, que hace un trabajo titulado "El uso de permutaciones para encontrar el cableado de la máquina Enigma"[25].

Este trabajo me viene perfecto de referencia, ya que quiero mostrar, mediante el cómputo por fuerza bruta y luego la paralelización de este, todas las explicaciones que ella da sobre las permutaciones necesarias para llevar a cabo el crackeo de la máquina Enigma. Básicamente, va a servir para dar números en cuanto a tiempos de ejecución sobre lo que ella habla a lo largo de su proyecto.

6.3 Alcance

6.3.1 Objetivo principal

El objetivo principal del proyecto es el de crear un código que simule el funcionamiento de la máquina *Bombe* para crackear la máquina Enigma. Una vez se tenga, se pretende paralelizar este lo máximo posible, mediante CUDA, para conseguir un tiempo de ejecución mucho mejor.

6.3.2 Sub Objetivos

Durante todo el proceso del proyecto, se espera también encontrar y explicar los fallos que se cometieron en aquella época (sobretodo por el bando de la alemania nazi) que llevaron a cabo a que los ingleses dieran con la forma de crackear la máquina.

Una vez hecho el crackeo de la máquina, una idea que tuvo el profesor Agustín fue, de incorporar el proyecto a la asignatura de tarjetas gráficas de forma incremental. Esto es, porque el laboratorio de la asignatura es así, pero con distintos códigos. Si se utilizase este código y desde un principio se sabe a dónde se quiere llegar, se verían los cambios (mejora) en cuanto a los tiempos de ejecución que se van obteniendo con el código a lo largo de la asignatura.

6.3.3 Obstáculos y riesgos

Existen una serie de riesgos en todo trabajo de investigación, los cuales, se convertirán en obstáculos, ya que nos impedirán seguir la metodología del trabajo que queríamos desde

un principio. Sobretodo, la mayoría de estos, nos retrasarán en nuestra planificación temporal. Se pueden dar las siguientes situaciones:

- Quedarse estancado en la implementación de una parte del código, porque no se sabe seguir o porque no da los resultados en cuanto a tiempo de ejecución que uno pensaba de un principio que iba a conseguir.
- Tener una idea mejor que la que ya se ha implementado, y por lo tanto, probarla para ver si realmente da mejores resultados.
- No encontrar la información necesaria (detallada) para realizar la parte de la implementación que esta representa.
- Los resultados en cuanto a tiempo de ejecución no son los esperados y no "existe" o no se encuentra, una forma de mejorarlo.
- Puede que no sea posible conseguir el sub objetivo de utilizar dicho código para la asignatura de tarjetas gráficas (o solo conseguirlo parcialmente), porque los métodos de paralelización no dan para abarcar todos los conceptos que se dan en la asignatura.
- Ser altamente dependiente del funcionamiento de los nodos boada, del ordenador desde el cual se trabaja, de la luz, el internet …

6.4 Metodología

6.4.1 Cómo se desarrollará

Como he comentado anteriormente, el objetivo principal del proyecto es ver las mejoras de tiempo al cual podemos llegar, por lo tanto, cada vez que se vayan incorporando mejoras en el código, estos tiempos se irán comparando con la implementación anterior y con la secuencial del principio.

Se empezará haciendo una optimización al código secuencial mediante la técnica *memoization*, ya que podemos aprovechar el cálculo de una situación (entrada, posición de los 3 rotores principales y posición el rotor reflector) para guardarla y utilizarla en un futuro si se vuelve a dar. Se comparará el resultado de los tiempos de esta versión con la secuencial con distintos textos de distintas longitudes para ver qué relación se obtiene y analizarla.

Una vez en este punto, se pasará a la parte de *crackear* la máquina. Para esto, se buscará todo tipo de información sobre el procedimiento que llevaron a cabo en aquella época Alan Turing y todo su equipo, junto con la máquina Bomb, diseñada para este propósito.

Simularé el comportamiento de dicha máquina pero con un código ejecutado con tarjetas

gráficas para ver el tiempo que se tarda en conseguir la configuración inicial de la máquina enigma a partir de un texto ya cifrado. Con este tiempo, y el que se haya encontrado en relación a lo que tardaban en descubrir dicha configuración inicial, se podrá ver cuánto hubiese ayudado tener este tipo de *hardware* en aquella época.

Por último, se comentará el impacto del proyecto en el que estaba relacionado Alan Turing y la máquina Bomb en relación a la situación mundial que se estaba viviendo en aquél momento.

6.4.2 Con qué medios se desarrollará

Los medios con los que dispongo en un principio serán todos los que se espera en un principio que vaya a necesitar a lo largo del proyecto. Estos son: el código secuencial con el que parte el proyecto, la documentación de la asignatura previamente realizada de tarjetas gráficas, la documentación que se encuentre en internet u otros medios, una cuenta en el servicio boada de la universidad con la/s tarjeta/s gráfica/s que haya en este para los cálculos, ordenador propio y conexión a internet con el que trabaje a lo largo del proyecto y comunicación con el director del proyecto.

El código, realizado por el profesor Agustín Fernández, simula el comportamiento de la máquina enigma con 3 rotores principales funcionando (a elección de 5 rotores totales), un rotor reflector y un *plugboard* que conecta 2 letras (un máximo de 10 parejas, tal y como era la máquina original). El texto a encriptar/desencriptar es el que está escrito en el código o un texto externo a este.

La documentación que se va a utilizar principalmente relacionada con la asignatura de tarjetas gráficas es toda la que explica el funcionamiento de la paralización de un código mediante cuda.

Se va a utilizar el servicio ssh para conectarse remotamente desde el ordenador donde esté trabajando al servicio Boada de la universidad, y así poder ejecutar el código con el *hardware* del que disponen.

Por último, las reuniones con Agustín van a tener una importancia notable, ya que al haber sido él, el precursor del trabajo y por lo tanto, conocedor del tema que se trata, podrá ser de gran ayuda en cuanto a detalles que se me pasen, información que no tenga y guía del trabajo en sí.

6.5 Descripción de las tareas

Se estima para este proyecto que la jornada laboral diaria es de 8 horas, siendo laborables también los fines de semana y los festivos, puesto que es un trabajo académico. También se tiene en cuenta, que si la tarea prevista para ese día se acaba antes de las 8 horas laborables, no se realizará otra tarea ese día, sino que se empezará el día previsto.

Se estimó que la duración del proyecto iba a ser de 42 días (296 horas), iniciándose el 21 de marzo de 2022 y con fecha de finalización prevista el 2 de mayo de 2022. Sin embargo, acabó finalizando el **20 de junio de 2022**. Por lo tanto, la duración total ha sido de **92 días** (**401 horas**).

6.5.1 Primera parte del proyecto (P1)

Búsqueda de información de la máquina Enigma (P1.1) (10 h)

Finalmente se han tardado 20 horas.

Es fundamental entender el funcionamiento de la máquina Enigma, ya que es un componente electromecánico que vamos a tener que simular mediante código.

Se utilizará cualquier medio de información que pueda ser útil: Youtube, distintas páginas webs, búsqueda de algún libro que trate de ello…

Estudiar el código inicial secuencial (P1.2) (10 h)

Partiendo del código secuencial del funcionamiento de la máquina, generado por el profesor Agustín Fernández, hay que entenderlo y relacionarlo con lo descubierto del funcionamiento de la máquina. Por lo tanto, la tarea (P1.1) es necesaria haberla realizado para empezar con esta.

Tras haber estudiado el código, y con la ayuda de diferentes simuladores de la máquina Enigma que hay por internet, verificaremos que las salidas son correctas para un número determinado de caracteres como entrada.

Una vez verificadas las salidas y ver que son correctas, generaremos los tiempos de ejecución para distintas entradas con las que trabajaremos a lo largo del proyecto. Estas entradas se tratan de libros en castellano que ya dispongo en formato txt, por lo que hay centenares de miles de letras (caracteres).

Optimizar el código secuencial (P1.3) (10 h)

Una vez estudiado el código, podemos deducir que partes de este son optimizables. Por lo tanto, la tarea (P1.2) es necesaria haberla realizado para empezar con esta.

Se utilizarán todos los métodos de optimización de código que se han aprendido en la asignatura de PCA que se puedan aplicar al código. De antemano, se sabe que uno se puede aplicar sí o sí, el de *memoization*, ya que podemos calcular previamente todos los caminos posibles para utilizarlos cuando se de dicha combinación.

Después, verificaremos si generamos la misma salida que las ejecuciones con el código sin optimizar. Se utilizarán los mismos textos que se utilizaron con la tarea 2.

Una vez hemos visto que los outputs coinciden, hay que sacar los tiempos de ejecución de estos para poder compararlos con los generados con el código sin optimizar.

Existe la posibilidad, y bastante alta, de que la optimización no haya sido suficiente y el código sin optimizar nos genere mejores tiempos de ejecución. Esto es debido, a que el texto que introducimos para encriptar no es lo suficientemente extenso como para que por lo menos, la técnica de *memoization* tenga efecto. No hay que preocuparse de ello, puesto que dicha técnica permite ser paralelizada.

Revisar programación en cuda (P1.4) (6 h)

Como hemos comentado, la técnica de *memoization* puede ser paralelizada, por lo tanto, vamos a recapitular toda la información sobre la programación en cuda que obtuve en la asignatura de tarjetas gráficas: la idea general de la programación en cuda, la programación de varias tarjetas gráficas y la ejecución con streams. Se va a utilizar la documentación de dicha asignatura.

Esta tarea puede ir haciéndose en cualquier momento, ya que no necesita de una tarea previa para realizarse.

Revisar el *hardware* en boada (P1.5) (minutos)

Para la programación en cuda que vamos a realizar, nos es importante saber con qué tarjetas gráficas vamos a estar trabajando, sobre todo a nivel de bloques y de *threads* de los que disponen.

Esta tarea, al igual que la tarea 4, puede ir realizándose en cualquier momento, ya que no es

dependiente de otra.

Programar en cuda el código optimizado (P1.6) (20 h)

Finalmente se han tardado 30 horas.

Vamos a implementar en el código optimizado, por lo menos en la parte de la técnica de *memoization,* la idea general de la programación en cuda (ejecución en bloques con un número específico de *threads* por bloque, …). Por lo tanto, la tarea (P1.3) debe de haberse realizado para empezar con esta.

Una vez implementado, hemos de verificar si la salida de este sigue siendo la misma que la del código general. Si es así, pasaremos a sacar los tiempos de ejecución para las mismas entradas y compararlo con los obtenidos anteriormente, para así, ver si cuánto hemos mejorado con la paralelización.

Mejoras en el código paralelizado (P1.7) (10 h)

Tarea obsoleta, se hace junto a la tarea P1.6.

Una vez hemos paralelizado el código y visto los tiempos de ejecución, podemos tratar de mejorarlo haciendo modificaciones al código, como por ejemplo, utilizar una distribución distinta de los bloques y los *threads* por bloque de la que hemos hecho inicialmente. Por lo tanto, la tarea (P1.6) debe haberse realizado para empezar con esta.

Luego, haríamos lo mismo, generaríamos los nuevos tiempos de ejecución de esta versión y los analizaremos con los generados por la versión inicial, la optimizada y la versión anterior paralelizada.

Programación multiGPU (20 h)

Nueva tarea, no se tuvo en cuenta.

Una vez tenemos nuestra versión de la máquina Enigma en cuda, paralelizaremos la ejecución con las 2 GPUs que hay en el nodo 9 de boada. Por lo tanto, se necesita la tarea P1.6 previamente realizada para empezar con esta.

6.5.2 Segunda parte del proyecto (P2)

Búsqueda de información de como crackearon los ingleses la máquina Enigma

(P2.1) (30 h)

Es muy importante saber el procedimiento que utilizaron para crackear la máquina, sobre todo en que se basaron para ello. Esto servirá para luego plasmarlo en el código que generamos que simulará lo mismo. Por lo tanto, la tarea (P1.1) debe haber sido realizada para empezar con esta.

Se utilizará cualquier medio de información que pueda ser útil: Youtube, distintas páginas webs, búsqueda de algún libro que trate de ello…

Generar código general del crackeo (P2.2) (50 h)

Finalmente se han tardado 70 horas.

Se implementará una primera versión del código para crackear la máquina Enigma. Esta versión va a ser lo más fiel posible a la *Bombe* original. Por lo tanto, la tarea (P2.1) debe haberse realizado para empezar con esta.

Una vez implementado, se verificará si la salida es la correcta, es decir, a través de una entrada, en este caso un texto encriptado (salida del código inicial del proyecto), encontrará una configuración inicial de la máquina. Si utilizamos nuestro código anterior de la máquina con esta configuración inicial, y se desencripta correctamente, habremos logrado el objetivo. Por lo tanto, la tarea 2 debe de haberse realizado para empezar con esta.

Tras verificar la salida y ver que es correcta, pasaremos a obtener el tiempo de ejecución.

Optimizar código general del crackeo (P2.3) (10 h)

Tarea obsoleta, no era necesaria.

Una vez verificamos que el código general del crackeo funciona, vamos a implementar optimizaciones en el código para mejorar el tiempo de ejecución de este, ya que como vimos, funciona por fuerza bruta. Por lo tanto, la tarea (P2.2) debe haberse realizado para empezar con esta.

Como en este punto en el que estoy escribiendo la documentación, no se como funciona al cien por cien el crackeo de la máquina porque aun no he buscado suficiente información, no estoy seguro de que como mínimo, se pueda utilizar la técnica de *memoization*, por lo que es muy posible de que el código no pueda ser optimizado.

Si hemos podido optimizar el código, pasamos a verificar si la salida que genera es la misma que el código general del crackeo para la misma entrada.

Tras verificar la salida y ver que es correcta, pasaremos a obtener el tiempo de ejecución y a compararlo con el que obtuvimos con el código general del crackeo.

Programar en cuda el código general del crackeo (P2.4) (40 h)

Una vez funciona nuestro código que simula el funcionamiento de la *Bombe*, pasamos a paralelizarlo con cuda.

Tras la implementación, verificaremos que la salida que nos genera es la misma que el código general del crackeo, y si vemos que es correcto, sacaremos los tiempos de ejecución de nuestra versión paralelizada y los compararemos con los tiempos sacados de la versión general y de la optimizada.

Programación multiGPU (P2.5) (20 h)

Una vez hemos verificado que nuestra versión paralelizada con tarjeta gráfica es la mejor y más rápida, probaremos a generar un código para realizar el mismo cálculo pero con más de una tarjeta gráfica. Por lo tanto, la tarea (P2.4) debe haberse realizado previamente para empezar esta.

Tras esto, verificaremos si la salida sigue siendo la correcta, sacaremos el tiempo de ejecución para distintos números de tarjetas gráficas y los compararemos con las versiones anteriores.

Programación con streams (P2.6) (20 h)

Tarea obsoleta, no era necesaria.

Una posible mejora a la ejecución de un código ejecutado por varias tarjetas gráficas es mejorarlo con la ejecución por streams. Por lo tanto, implementaremos una versión con streams de nuestro código multiGPU, por lo que la tarea (P2.5) debe haberse realizado para empezar con esta.

Como siempre, tras verificar que la salida es correcta, generamos los tiempos de ejecución y los compararemos con la versión multiGPU, para ver si la hemos podido mejorar.

Máquina *check* secuencial (50 h)

Nueva tarea, no se tuvo en cuenta por desconocimiento.

Una vez visto el resultado de la *Bombe*, seguramente este sea incompleto, por que deberemos de crear otro programa que simule lo que hacían los ingleses para probar configuraciones hasta que diesen con la correcta. Para que esta tarea pueda desarrollarse, hace falta que la tarea (P2.2) se haya finalizado y funcione correctamente.

Máquina *check* con cuda (30 h)

Nueva tarea, no se tuvo en cuenta.

Una vez visto que llegamos a una configuración de la Enigma correcta, hemos de mejorar nuestro código de la máquina *check* para que mediante programación en cuda, se ejecute más rápido. Por lo tanto, necesitamos que la tarea anterior se haya realizado y funcione correctamente.

Crackeo con fuerza bruta (10 h)

Nueva tarea, no se tuvo en cuenta.

Para mostrar lo importante que fue el método con el que crackearon la máquina Enigma, evitando a toda costa la fuerza bruta, se hará un programa que probará todas las configuraciones de Enigma posibles para el texto que queremos desencriptar. Para que esta tarea pueda realizarse, debe de haberse terminado la tarea anterior.

6.5.3 Tercera parte del proyecto (P3)

Documentación (P3.1) (30 h)

Finalmente se han tardado 60 horas.

Tras haber ido apuntando y analizando todos los resultados obtenidos hasta llegar aquí, tendremos que documentar todo el proceso del proyecto y sacar conclusiones de dichos resultados. **Finalmente se han anotado los resultados una vez estábamos ya documentando, por lo que esta tarea ha sufrido un gran retraso frente a lo previsto.**

La tarea (P2.6) debe haberse realizado para empezar con esta, ya que es la última del proyecto.

6.5.4 Resumen de las tareas

Tabla obsoleta, el cómputo de horas totales es de 401, debido al aumento del tiempo requerido para realizar algunas de las ya previstas y de la introducción de nuevas tareas que no se habían tenido en cuenta por falta de información sobre el tema en cuestión.

Para la lectura de la Tabla X, vamos a utilizar sigla “PR” para hacer referencia al programador, y “RB” para el responsable de boada.

Siglas	Descripción	Horas trabajo	Horas reunión	Requisitos	Recursos	
					Humanos	Materiales
Primera parte del proyecto						
P1.1	Búsqueda de información de la máquina Enigma	10			PR	PC, internet
P1.2	Estudiar el código secuencial	10	1	P1.1	PR, RB	PC, boada
P1.3	Optimizar el código secuencial	20	1	P1.2	PR, RB	PC, boada
P1.4	Revisar programación en cuda	6			PR	PC
P1.5	Revisar el hardware de boada	1			PR	PC, boada
P1.6	Programar en cuda el código optimizado	20	1	P1.3, P1.4, P1.5	PR, RB	PC, boada
P1.7	Mejorar el código paralelizado	10	1	P1.6	PR, RB	PC, boada
Segunda parte del proyecto						
P2.1	Búsqueda de información del crackeo de la máquina Enigma	30		P1.1	PR	PC, internet
P2.2	Implementar código general del crackeo	50	1	P2.1	PR, RB	PC, boada
P2.3	Optimizar código general del crackeo	20	1	P2.2	PR, RB	PC, boada
P2.4	Implementar el código con cuda	40	1	P2.3	PR, RB	PC, boada
P2.5	Programar con multiGPU	20	1	P2.4	PR, RB	PC, boada
P2.6	Programar con streams	20	1	P2.5	PR, RB	PC, boada
Tercera parte del proyecto						
P3.1	Documentación	30		P2.6	PR	PC
Horas parciales		257	9			
Horas totales		296				

Tabla 28. Resumen de las tareas

6.6 Estimaciones y Gantt

Diagrama de Gantt obsoleto, debido al número de horas totales.

Para basarme en el diagrama de Gantt de la Figura X, he supuesto que la jornada laboral diaria es de 8 horas. Como es un trabajo académico, fines de semanas y festivos también son laborales.

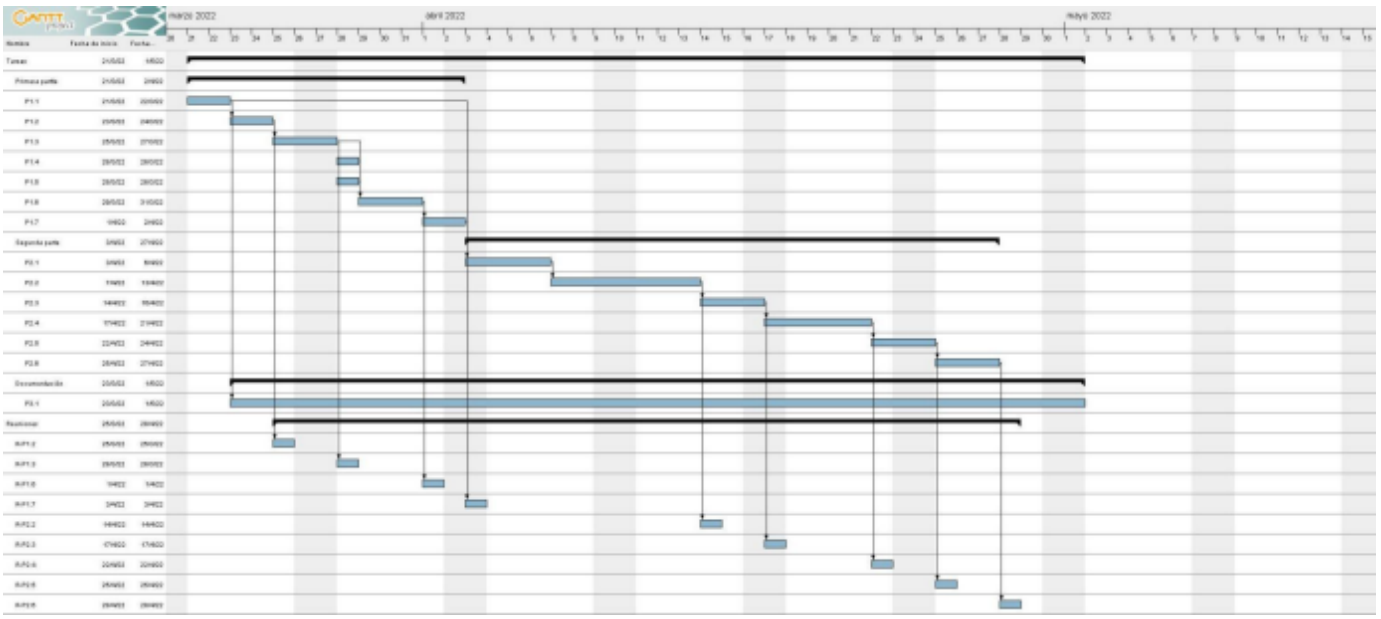


Figura 82. Diagrama de Gantt

6.7 Gestión del riesgo: planes alternativos y obstáculos

Tal y como hemos comentado anteriormente, pueden originarse a lo largo del proyecto una serie de obstáculos que algunos de ellos tienen fácil solución (independientemente del tiempo que se emplee en ella) y otros que no o es imposible.

Al ser un trabajo incremental (se quiere mejorar siempre lo que ya se tiene), si nos estancamos en una parte de la implementación del código y no hay forma de proseguir, se tratará de realizarlo de una forma más sencilla o, utilizar la que ya está implementada.

Siempre que se piense en una idea mejor de la que se ha implementado, se probará para ver si realmente mejora lo ya realizado. Si es así, está será la versión con la que seguiremos trabajando. Si por el contrario, no lo es, esta se descartará.

Si por alguna razón, no encontramos en internet la información necesaria para implementar una tarea, se recurriría a buscar libros. Esto hay que tenerlo en cuenta, ya que se invertiría tiempo de espera si se hace una compra *online* y un gasto que no se contempla desde un principio.

Al estar trabajando con material informático, cualquier problema en estos dispositivos nos penalizará. Si el problema está en el ordenador desde donde se trabaja, se utilizará otro ordenador (situado en un ámbito de trabajo que no es el mio) y se pedirán las piezas que haya que pedir para reparar el ordenador. Se optará por la misma solución si hay un

problema en la red eléctrica o en internet, ya que el ordenador desde el cual se trabaja, es de sobremesa. Si por el contrario, el problema viene de boada, la única solución que existe es comprar los componentes que se hayan estropeado e instalarlos.

Por último, si vemos que lo que se ha empezado de CUDA no cubre todo lo que la asignatura de tarjetas gráficas abarca, no hay problema con ello, ya que no es el objetivo principal del proyecto, si no una ampliación con la que el profesor Agustín Fernández se podría ver beneficiado.

6.8 Gestión Económica

6.8.1 Costes de personal (CPA)

Para este proyecto, nos hacen falta dos perfiles de trabajador: el programador, y el encargado de gestionar los nodos boada que están situados en la universidad.

He realizado una búsqueda sobre lo que cobra de media un programador en barcelona, mediante la web glassdoor [26], y el resultado ha sido de 30 000€ al año. Suponiendo que una jornada laboral son de 8 horas diarias, de lunes a viernes y un mes al año de vacaciones:

30 000€/año / 1.76horas/año = 17.04€/hora

Teniendo en cuenta el precio que hay que pagar para la seguridad social, el coste de la empresa para este trabajador sería de:

30 000€/año x 1.3 = 39 000€/año, y por lo tanto:
39 000€/año / 1.76horas/año = 22.16€/hora

Por otra parte, lo que cobra el encargado de gestionar y mantener activos los nodos boada es alrededor de 40 000€ al año, según el profesor Agustín Fernández, que trabaja en la universidad. Suponiendo las mismas condiciones de trabajo:

40 000€/año / 1.76horas/año = 22.73€/hora

Teniendo en cuenta el precio que hay que pagar para la seguridad social, el coste de la empresa total por este trabajador sería de:

40 000€/año x 1.3 = 52 000€/año, y por lo tanto:
52 000€/año / 1.76horas/año = 29.54€/hora

En la Tabla 1.1, se muestra una tabla resumen de los perfiles que van a ejecutar cada tarea, el tiempo que lo van a estar haciendo y el coste para la empresa de cada una de ellas.

La siguiente tabla está obsoleta debido al número de horas totales necesarias para la realización del proyecto. El número de horas trabajadas por parte del programador es de 401 horas, y el del responsable de los nodos boada es de 180 horas (se calculó mal en la Tabla 1.1 las horas de este trabajador, ya que solo es necesario que realice su trabajo cuando el programador necesita ejecutar en boada). Teniendo en cuenta los costes por hora acabos de ver, tenemos que los costes directos por actividad son de:

- **Programador: 8 886.16€**
- **Responsable boada: 5 317.2€**
- **Total: 14 203.36€**

Siglas	Descripción	Horas		Coste
	Primera parte del proyecto	Programador	Responsable boada	
P1.1	Búsqueda de información de la máquina Enigma	10		221,60 €
P1.2	Estudiar el código secuencial	11		243,76 €
P1.3	Optimizar el código secuencial	21	20	1.056,16 €
P1.4	Revisar programación en cuda	6		132,96 €
P1.5	Revisar el hardware de boada	1		22,16 €
P1.6	Programar en cuda el código optimizado	21	20	1.056,16 €
P1.7	Mejorar el código paralelizado	11	10	539,16 €
	Segunda parte del proyecto			
P2.1	Búsqueda de información del crackeo de la máquina Enigma	30		664,80 €
P2.2	Implementar código general del crackeo	51	50	2.636,23 €
P2.3	Optimizar código general del crackeo	21	20	1.056,16 €
P2.4	Implementar el código con cuda	41	40	2.113,53 €
P2.5	Programar con multiGPU	21	20	1.056,16 €
P2.6	Programar con streams	21	20	1.056,16 €
	Tercera parte del proyecto			
P3.1	Documentación	30		664,80 €
Total		296	200	12.519,80 €

Tabla 29. Costes directos por actividad

6.8.2 Costes genéricos (CG)

6.8.2.1 Amortizaciones

Hardware

Para el programador, se va a necesitar un ordenador de sobremesa con el que trabajar. Actualmente, se dispone de uno valorado en 900€ y una pantalla de 150€. Si suponemos, que el tiempo medio de su vida útil es de unos 4 años (72 meses), que el proyecto va a durar 296 horas, que al año hay 220 días laborables y que estos días tienen 8 horas de trabajo:

La siguiente amortización está obsoleta, el número de horas totales es de 401, por lo que la amortización del ordenador será de 59.81€.

(900€ + 150€) x 296h/4años x 1año/220dias x 1dia/8horas = 44€

Por la parte de boada, tenemos un intel Xeon E5-26020v2 que cuesta alrededor de 450 euros [27]. Además, disponemos de 4 tarjetas gráficas TESLA k40c de NVIDIA, que cuestan alrededor de 800 euros cada una [28]. Por lo tanto, si seguimos suponiendo la misma vida útil, la amortización será de:

La siguiente amortización está obsoleta, se ha acabado trabajando en un nodo de boada que solo tiene 2 gráficas en lugar de 4 y las horas totales del nodo activo no han sido las planificadas, sino 180, por lo que el coste de la amortización será de 44.74€.

(450€ + 4 x 800€) x 296h/4años x 1año/220dias x 1dia/8horas = 153,46€

Software

Todo el software que se va a utilizar es una distribución de linux, por lo tanto, un sistema operativo gratuito.

La siguiente amortización total está obsoleta: 104.55€.

En total, las amortizaciones son de: 44€ + 153.46€ = 197.46€

6.8.2.2 Consumo eléctrico

Para el ordenador de sobremesa que hemos comentado anteriormente, observamos que el TDP de su procesador (i5-9600k)) es de 95W [29], y el de su tarjeta gráfica (rtx 2060) es de 160W [30]. Sin embargo, este ordenador no tendrá una carga de trabajo para nada alta, por lo que el consumo será mucho menor. Además, tiene un monitor cuyo TDP es de 45W [31]. Por lo tanto, teniendo en cuenta que va a estar trabajando las 296 horas del proyecto:

El siguiente consumo está obsoleto debido al número de horas totales: 50 125 = 50.12 kWh.

(~50W + ~30W + ~45W) x 296h = 37 000Wh = 37kWh

A partir de aquí, a fecha de consulta 10 de marzo de 2022, el precio del kWh es de

0.505998€ [32], podemos saber que el coste eléctrico de dicho ordenador va a ser de:

El siguiente coste está obsoleto debido a consumo eléctrico: 25.36€/proyecto.

0.505998€/kWh x 37kWh = 18.72€/proyecto

Por otra parte, los núcleos boada vienen instalados con un Intel Xeon E5-26020v2, cuyo TDP es de 80W [2] (se consumirá menos). Además, dispone de 4 tarjetas gráficas TESLA K40c, con un TDP de 245W [11]. Teniendo en cuenta, que el consumo en las tarjetas gráficas, será solo alto cuando mandemos el código a ejecutar, vamos a suponer que de las 200 horas que boada tiene que estar activo, va a estar trabajando al 100% media hora:

El siguiente consumo está obsoleto debido al número de horas totales, número de horas donde boada trabaja al 100% (50%, debido a que mucha parte del proceso de documentación necesita de resultados de tiempos de ejecución) y al número de tarjetas gráficas del nodo boada: 11.91 kWh 50% de carga y 51.3 kWh al 100%. Por lo que 63.21 kWh totales.

(~65W + 30Wx4gpus) x 199.5h = 36 907.5Wh = 36.91kWh

(80W + 245Wx4gpus) x 0.5h = 530Wh = 0.53kWh

Por lo tanto, el coste eléctrico de los nodos boada va a ser de:

El siguiente coste está obsoleto debido a consumo eléctrico: 31.98€/proyecto.

0.505998€/kWh x (36.91kWh + 0.53kWh) = 18.94€/proyecto

El siguiente coste está obsoleto: 25.36€/proyecto + 31.98€/proyecto = 57.43€/proyecto.

En total, el coste eléctrico del proyecto va a ser de: 18.72€ + 18.94€ = 37.66€

6.8.2.3 Factura de internet

El coste mensual de la tarifa de internet contratada con el ordenador de sobremesa es de 23.66€. Suponiendo que el nodo de boada va funcionar con la misma compañía, el coste de internet en lo que dura el proyecto es de:

3meses x 2tarifas x 23.66€/mes = 142 euros/proyecto

6.8.2.4 Total costes genéricos

En la Tabla 1.2, se muestra un coste total de todos los costes genéricos que hemos comentado a lo largo del documento hasta ahora.

La siguiente tabla está obsoleta, debido a todos los cambios que hemos visto a lo largo del apartado "8.2 Costes genéricos (CG)". El total es de 104.55€ + 57.43€ + 142€ = 303.98€.

Concepto	Coste
Amortizaciones	197,46 €
Consumo eléctrico	37,66 €
Factura internet	142,00 €
Total	377,12 €

Tabla 30. Costes genéricos

6.8.3 Contingencias

Ya que siempre es importante contemplar la posibilidad de que hayan complicaciones y/o contratiempos durante la realización del proyecto, estos, pueden hacer que el coste final sea mayor. Por lo tanto, podemos dar un margen del 15% del precio:

Las siguientes contingencias están obsoletas, debido a los cambios vistos anteriormente: (14 203.36€ + 303.98€) x 0.12 = 1 740.88€.

(12 519.80€ + 377.12€) x 0.12 = 1 547.63€

6.8.4 Imprevistos

Por último, hace falta valorar cualquier tipo de imprevisto que pueda suceder a lo largo del trabajo. En el caso de este proyecto, cualquier reemplazo de hardware tiene que tenerse en cuenta, ya que será un extra al coste total del proyecto. También, y algo muy probable, es que el tiempo de implementación de algunas tareas se alargue, ya que puede que muchas funcionalidades cuesten más de lo que se cree en ser ideadas. La Tabla 1.3 muestra un resumen de los imprevistos que pueden darse y de su coste, así como la probabilidad de que estos ocurran.

Imprevisto	Coste parcial	Probabilidad	Coste
PC Sobremesa			
Componente	150,00 €	5,00%	7,50 €
Nodo boada			
Nueva CPU	450,00 €	5,00%	22,50 €
Nueva GPU	800,00 €	20,00%	160,00 €
Humano			
Incremento tiempo reemplazo hardware (1h)	29,54 €	30,00%	8,86 €
Incremento tiempo implementación (10h)	227,30 €	80,00%	181,84 €
Total			380,70 €

Tabla 31. Costes de imprevistos

Tal y como se puede observar, la probabilidad de reemplazo de hardware es la suma de probabilidades de reemplazo de componente del PC de sobremesa, de la CPU y/o GPU del nodo boada (5% + 5% + 20%).

6.8.5 Coste total del proyecto

En la Tabla 1.4, se muestra el coste total del proyecto, teniendo en cuenta los apartados anteriores.

La siguiente tabla está obsoleta debido a todos los cambios que hemos ido viendo: (14 203.36€ + 303.98€ + 1 740.88€ + 380.7€ = 16 628.92€.

Concepto	Coste
CPA	12.519,80 €
CG	377,12 €
Contingencias	1.547,63 €
Imprevistos	380,70 €
Total	14.825,25 €

Tabla 32. Coste total del proyecto

6.8.6 Control de gestión

Debido a los problemas externos a este proyecto, que están afectando a la subida de precios de muchos productos, y en el que en este caso nos concierne, la luz, se hará a la semana un cálculo de lo que se haya gastado de electricidad y se pagará la diferencia con parte del coste de contingencias que habíamos calculado.

Por ahora, sabemos que el ordenador va a tener un coste de luz de 18.72€ en todo el proyecto. Si va a estar trabajando 296 horas en total, y cada semana tiene 40 horas

laborables. **Obsoleto, sabemos que el ordenador va a tener un coste de luz de 25.36€, y que va a estar trabajando 401 horas. El siguiente coste por semana es correcto pero porque coincide el resultado con los nuevos datos.**

296horas x 1semana/40horas = 7.4 semanas, por lo que:
18.72€/7.4semanas = 2.53€/semana

Por otro lado, sabemos que el nodo boada va a tener un coste de luz de 18,94€ en todo el proyecto. Si va a estar trabajando 200 horas en total. **Obsoleto, sabemos que el coste de luz del nodo boada va a ser de 31.98€ que el total de horas van a ser de 180, por lo que el coste por semana es de: 7.11€.**

200horas x 1semana/40horas = 5 semanas, por lo que:
18.94€/5semanas = 3.79€/semana

6.9 Informe de Sostenibilidad

6.9.1 Fita inicial

6.9.1.1 Autoevaluación

Siempre he sido partidario de reutilizar lo máximo posible los recursos. En mi caso, todo el hardware de los ordenadores viejos siempre he aprovechado al máximo posible todo lo que funcionaba para comprar los mínimos componentes posibles para así, darle un ordenador en condiciones a un familiar o a amigos/conocidos. Todo esto es debido, a que en varias ocasiones he visto documentales sobre el tema de verter componentes, y tengo muy integrado las consecuencias que esto conlleva (tanto ambientales como sociales).

Sin embargo, la encuesta me ha dado a conocer muchos otros aspectos que desconocía y que por ende, nunca los tenía en cuenta, como por ejemplo todo lo relacionado con el código deontológico, consecuencias en la justicia social, … Es decir, los aspectos generales si los tenía presente, pero cuando vas a detalles más específicos de dimensión social, nunca me han hablado de ellos.

6.9.1.2 Dimensión Económica

Tal y como comentamos en el apartado de “Dimensión Ambiental”, el hardware utilizado en el nodo boada tiene un precio que siempre va a ir al alza, ya que cuanto más potente sea más nos beneficia, y por ende, más caro. Sin embargo, el ordenador de sobremesa que sirve

para programar, sí que puede ser uno de menos sofisticado, y por ende, más barato.

Es decir, si comparamos los costes de este proyecto con similares, una parte del proyecto sí que podría estar reducido, es decir, el coste del ordenador de sobremesa, ya que no se necesita de mucha potencia para la realización de este. Sin embargo, la parte de boada, aumentaría cada vez que se quiere revisar este proyecto o mejorarlo, ya que las tarjetas gráficas cada vez són más sofisticadas y por lo tanto, más caras.

Una vez comentado esto, y viendo los costes que hemos ido anotando a lo largo de este documento, visto que no hay necesidad del proyecto (apartado “Dimensión Social”), no es rentable hacer el proyecto, por su elevado coste.

6.9.1.3 Dimensión Ambiental

Por parte del nodo de boada, se necesita de un hardware potente, ya que cuanto más lo sea, más potencia de cálculo nos va a ofrecer, y eso es precisamente lo que necesitamos. Por lo tanto, no podemos reutilizar cualquier tarjeta gráfica e instalarla en el nodo, sinó que tenemos que buscar un conjunto de 4, cuanto más potentes mejor.

Sin embargo, por parte del ordenador de sobremesa que va a ser utilizado para programar, sí que podemos buscar uno que sea mucho menos sofisticado que el que se ha propuesto en el proyecto (ya que me he basado en que es el que dispongo actualmente). Con esto, lo que haremos será disminuir, obviamente, el stock de esta gama de ordenadores y mandarán a fabricar otros de la misma gama, haciendo que estas fábricas gasten menos recursos de que si hubieran mandado a fabricar ordenadores más potentes.

6.9.1.4 Dimensión Social

A nivel personal, este proyecto me va a aportar mucho, ya que primero de todo, me adentro en el mundo de la gestión de un proyecto, cosa que en un futuro es muy posible que esté involucrado en uno, tanto como el que lo gestiona, como un trabajador. Además, voy a trabajar un ámbito de la informática (hardware) que es a la que quiero dedicarme, por eso he elegido la especialización de “Ingeniería de computadores” y más específico, en el apartado de investigación, ya que el proyecto va de eso.

Como dijimos al principio del primer documento de GEP, el trabajo no pretende ayudar socialmente con su resultado, ya que es un trabajo de investigación basado en un momento concreto de la historia. Lo que se quiere dar, es una idea de la evolución que ha tenido el hardware hasta ahora, comparando la máquina Bombe, con la que crackearon la codificación de la máquina Enigma, con las tarjetas gráficas actuales. Por lo tanto, no existe

una necesidad del proyecto.

6.9.2 Fita final

6.9.2.1 Desarrollo del proyecto

6.9.2.1.1 Impacto Económico

Si se comparan los costes que hubo inicialmente con los finales se observan desviaciones por dos motivos:

- El proyecto ha tardado más en realizarse (132 horas adicionales).
- Se ha reducido las horas de trabajo del nodo de boada respecto a las propuestas inicialmente, debido a que todo código secuencial puede trabajarse en local y solo mandarlo a boada cuando necesitemos trabajar en cuda.

Para el coste del personal, se ha pasado de tener un coste de 12 519.89€ a un coste de 14 203.36€.

Para los costes genéricos:

- Las amortizaciones han pasado de tener un coste de 197.46€ a un coste de 104.55€ debido a que no se ha utilizado tanto *hardware* como el que se creyó necesario.
- El consumo eléctrico ha pasado de tener un coste de 37.66€ a un coste de 57.45€.
- La factura eléctrica ha quedado igual.
- El total de costes genéricos ha pasado de 377.12€ a 303.98€.

Las contingencias han pasado de tener un coste de 1 547.63€ a un coste de 1 740.88€.

Por lo tanto, el coste total del proyecto ha pasado de ser 14 825.25€ a un coste de 16 628.92€, todo y habiendo reducido en gran parte las horas activas del nodo boada. Sin embargo, las horas totales del proyecto se han elevado bastante.

6.9.2.1.2 Impacto ambiental

Partiendo de los dos puntos vistos en el apartado anterior, que diferencian las predicciones del documento inicial con lo necesito para el l final, tenemos que:

- Para el documento inicial, un consumo eléctrico del *hardware* de 74.44 kWh. Además, partiendo de que un trabajador consume 0.1 kWh en su rutina habitual, tenemos que el programador consume 29.6 kWh y que el encargado de boada consume 20 kWh, por lo que un total de 124.04 kWh.
- Para los resultados finales, tenemos un total de 113.33 kWh para el consumo eléctrico del *hardware*, un consumo de 40 kWh por el programador y uno de 18 kWh para el encargado del nodo de boada, por lo que un total de 171.33 kWh.

Al igual que en el apartado anterior, la idea es solo estar conectado a boada cuando sea necesario, es decir, cuando tenemos que tomar tiempos o programar en paralelo. Para probar ejecuciones secuenciales, las podemos realizar desde casa, ya que el *hardware* es menos potente y por lo tanto, consumiremos menos.

6.9.2.1.3 Impacto social

La realización de este proyecto nos ha hecho pensar mucho tanto a mi director de proyecto como a mi sobre los resultados que se pueden obtener mediante el ingenio humano en contra de puros algoritmos de fuerza bruta. Por lo tanto, ser conscientes de estas ideas para luego ponerlas en práctica en nuestro ámbito de trabajo tan pronto como sea posible.

6.9.2.2 Vida útil

6.9.2.2.1 Impacto social

Todo aquél que quiera aprender o leer sobre ejecuciones y decisiones tomadas sobre programas paralelizados en cuda se va a ver beneficiado de este proyecto, ya que en este, no solo se muestran ideas de cómo ejecutar algoritmos, sino de análisis de por qué una ejecución es buena, por que no lo es, como se podría mejorar…

Además, existe la posibilidad de que el profesor Agustín Fernández, utilice algún código de este proyecto para llevarlo al campo de la asignatura que ejerce, Targetas Gráficas y Aceleradores, para así, organizar las clases de laboratorio de una forma progresiva, trabajando con el código de la Enigma.

7. Conclusiones y líneas abiertas

A lo largo del proyecto, nos hemos ido dando cuenta de un problema que ha pasado desde siempre en el ámbito de la criptografía, que sigue ocurriendo y que lo seguirá haciendo, es el

tema del fallo humano. Una brecha de seguridad que si otro individuo la detecta, la va a explotar y podrá realizar un ataque. En el caso de la Enigma, dos fallos humanos que fueron los causantes de que se pudiera *crackear*: el primero, que una letra que se introducía en la Enigma no podía generar la misma, y la segunda, que la milicia nazi empezaban los mensajes siempre con el mismo patrón (el parte meteorlógico).

Nos hemos dado cuenta también de que cuando existen alternativas a la fuerza bruta hay que aplicarlas inmediatamente, ya que la reducción en el tiempo de cómputo se ve favorecida enormemente. Habrá veces que para llegar a tal reducción como se llegó con la *Bombe*, se requiera de mucho ingenio como el de Alan Turing, pero vale la pena pensar estas alternativas a la fuerza bruta.

En cuanto al ámbito de cuda, hemos visto que no todos los programas están pensados para que tengan un buen rendimiento en cuda; a unos se les podrá hacer cambios menores (como es el caso de la *Bombe*, ya que se quería llegar a una ejecución que simulase lo más fielmente a lo que se realizó), a otros, habrá que pensar soluciones alternativas para acomodarlos mejor a una ejecución en cuda. Como hemos visto y comentado, este acomodamiento se basa, principalmente en 3 factores: que no haya divergencia en el camino de ejecución de los *threads,* que se hagan accesos a memoria contiguos y alineados y que se oculte la latencia de memoria, explotando los accesos a esta.

Una vez finalizado el proyecto, toda la idea que hay detrás de este y los códigos pueden ser empleados y expandidos de muchas formas:

- Se comentó con el profesor Agustín Fernández de utilizar algún código en los laboratorios de la asignatura de TGA para ver qué mejoras recibe un código al ser paralelizado. También pueden llegar a ser útiles para entender y analizar conceptos de cuda, sobre todo conceptos como los accesos a memoria (analizarlos con el *profiler* de nvidia, por ejemplo), tal y como se ha hecho en varios apartados.
- Pueden expandirse los códigos para que hagan uso de lo que llamaron "*ring settings*", el cual, permite seleccionar los primeros contactos entre dos rotores, por lo que se elevaría cada posible posición de un rotor de 26 posibles posiciones iniciales a 26x26.
- Pueden expandirse los códigos para que hagan uso de más parejas del *plugboard*, hasta un máximo de 13. Esto aumentaría la complejidad de la Enigma enormemente, y por lo tanto, de los cálculos.
- Puede expandirse los códigos para que ejecuten una máquina Enigma con 5 rotores, y dos de ellos con 2 muescas, tal y como hicieron la milicia nazi más tarde con las comunicación del ejército naval. Por lo tanto, la máquina *Bombe* debería ser trabajada de nuevo.

- Puede estudiarse la mejora del código de fuerza bruta, aunque esté incompleto. La mejora sustancial se vería afectada por una generación de las combinaciones más óptima de la que he realizado, ya que esta, se basa en acceder a matrices cada vez más grandes.

8. Referencias

[1] Jesús Velasco, J. Breve historia de la criptografía. El Diario [en línea]. 20 Mayo 2014. Disponible en: <https://www.eldiario.es/turing/criptografia/breve-historia-criptografia_1_4878763.html>

[2] Atbash. En: Wikipedia [en línea]. [Consulta: 20 Mayo 2022]. Disponible en: <https://es.wikipedia.org/wiki/Atbash>

[3] Cifrado por sustitución. En: Wikipedia [en línea]. [Consulta: 20 Mayo 2022]. Disponible en: <https://es.wikipedia.org/wiki/Cifrado_por_sustituci%C3%B3n>

[4] Cifrado por transposición. En: Wikipedia [en línea]. [Consulta: 20 Mayo 2022]. Disponible en: <https://es.wikipedia.org/wiki/Cifrado_por_transposici%C3%B3n>

[5] Escítala. En: Wikipedia [en línea]. [Consulta: 20 Mayo 2022]. Disponible en: <https://es.wikipedia.org/wiki/Esc%C3%ADtala>

[6] Cifrado César. En: Wikipedia [en línea]. [Consulta: 20 Mayo 2022]. Disponible en: <https://es.wikipedia.org/wiki/Cifrado_C%C3%A9sar>

[7] Al-Kindi. En: Wikipedia [en línea]. [Consulta: 20 Mayo 2022]. Disponible en: <https://es.wikipedia.org/wiki/Al-Kindi>

[8] Tabara Carbajo, JL. Criptografía clásica [en línea]. [Consulta: 20 Mayo 2022]. Disponible en: <https://joseluistabaracarbajo.gitbooks.io/criptografia-clasica/content/Cripto11.html>

[9] Cifrado de Alberti. En: Wikipedia [en línea]. [Consulta: 20 Mayo 2022]. Disponible en: <https://es.wikipedia.org/wiki/Cifrado_de_Alberti>

[10] Alberti cipher. En: Wikipedia [en línea]. [Consulta: 20 Mayo 2022]. Disponible en: <https://en.wikipedia.org/wiki/Alberti_cipher>

[11] Cifrado de Trithemius. En: Wikipedia [en línea]. [Consulta: 20 Mayo 2022]. Disponible en: <https://es.wikipedia.org/wiki/Cifrado_de_Trithemius>

[12] Enigma machine. En: Wikipedia [en línea]. [Consulta: 20 Mayo 2022]. Disponible en: <https://en.wikipedia.org/wiki/Enigma_machine>

[13] Enigma Simulator (2016). Piotte13 github [simulador web]. [Consulta: 11 Marzo 2022]. DIsponible en: <https://piotte13.github.io/enigma-cipher/>

[14] Enigma rotor details. En: Wikipedia [en línea]. [Consulta: 2 Junio 2022]. Disponible en: <https://en.wikipedia.org/wiki/Enigma_rotor_details>

[15] Enigma wiring. En: Cryptomuseum [en línea]. [Consulta: 2 Junio 2022]. Disponible en: <https://www.cryptomuseum.com/crypto/enigma/wiring.htm>

[16] Owen, J. How did the Enigma machine work? [YouTube]. [Consulta: 2 Junio 2022]. Disponible en: <https://www.youtube.com/watch?v=ybkkiGtJmkM&ab_channel=JaredOwen>

[17] Technical Specification of the Enigma. En: codesandciphers [en línea]. [Consulta: 2 Junio 2022]. Disponible en: <https://www.codesandciphers.org.uk/enigma/rotorspec.htm>

[18] https://qph.cf2.quoracdn.net/main-qimg-1fba8c31b98673a2c02ac8f9ca50c79e-pjlq

[19] Iedermueller. Enigma Machine Mechanism (feat. a 'Double Step') [YoutTube]. [Consulta: 11 Junio 2022]. Disponible en: <https://www.youtube.com/watch?v=hcVhQeZ5gI4&ab_channel=Iedermueller>

[20] Ellsbury, G. The Turing Bombe: Cribs and Menus [en línea]. [Consulta: 26 Marzo 2022]. Disponible en: <http://www.ellsbury.com/bombe1.htm>

[21] Numberphile. Flaw in the Enigma Code [YoutTube]. [Consulta: 3 Febrero 2022]. Disponible en <https://www.youtube.com/watch?v=V4V2bpZlqx8&ab_channel=Numberphile>

[22] Ellsbury, G. The Turing Bombe: Description of the Bombe [en línea]. [Consulta: 26 Marzo 2022]. Disponible en: <http://www.ellsbury.com/bombe2.htm>

[23] Ellsbury, G. The Turing Bombe: How the Bombe was Plugged-Up [en línea]. [Consulta: 26 Marzo 2022]. Disponible en: <http://www.ellsbury.com/bombe3.htm>

[24] Ellsbury, G. The Turing Bombe: How the Bombe Worked [en línea]. [Consulta: 26 Marzo 2022]. Disponible en: <http://www.ellsbury.com/bombe4.htm>

[25] Paño Badía, S. El uso de permutaciones para encontrar el cableado de la máquina Enigma [en línea]. Trabajo final de grado, Universidad de Zaragoza. Departamento de Matemáticas, 2017 [Consulta: 18 marzo 2022]. Disponible en: <https://zaguan.unizar.es/record/64250/files/TAZ-TFG-2017-2158.pdf>

[26] Sueldos para Programador. En: glassdoor [en línea]. [Consulta: 9 marzo 2022]. Disponible en: <https://www.glassdoor.es/Sueldos/barcelona-programador-sueldo-SRCH_IL.0,9_IC2547194 _KO10,21.htm>

[27] Procesador Intel Xeon E5-2620 v2. En: Intel [en línea]. [Consulta: 9 marzo 2022]. Disponible en <https://ark.intel.com/content/www/es/es/ark/products/75789/intel-xeon-processor-e5262 0-v2-15m-cache-2-10-ghz.html>

[28] NVIDIA Tesla K40c: especificaciones técnicas y pruebas. En: technical city [en línea]. [Consulta: 9 marzo 2022]. Disponible en: <https://technical.city/es/video/Tesla-K40c>

[29] Procesador Intel Core i5-9600k. En: Intel [en línea]. [Consulta: 99 marzo 2022]. Disponible en: <https://ark.intel.com/content/www/es/es/ark/products/134896/intel-core-i59600k-processor-9m-cache-up-to-4-60-ghz.html>

[30] GeForce RTX 2060 Ventus 6G OC. En: MSI [en línea]. [Consulta: 9 marzo 2022]. Disponible en: <https://es.msi.com/Graphics-Card/GeForce-RTX-2060-VENTUS-6G-OC/Specification>

[31] BenQ ZOWIE XL2411P 24" LED 144Hz e-Sports. En: PCComponentes [en línea]. [Consulta: 9 marzo 2022]. Disponible en: <https://www.pccomponentes.com/benq-zowie-xl2411p-24-led-144hz-e-sports>

[32] Precio de la luz por horas. En: Tarifaluzhora [en línea]. [Consulta: 10 marzo 2022]. Disponible en: <https://tarifaluzhora.es/>