Emulador Software para el cálculo de primos de Mersenne

AUTORES:

Pablo Cerro Cañizares

Cristina María Esteban Luis

Grado en Ingeniería de Computadores

Facultad de Informática

Departamento de Arquitectura de

Computadores y Automática

UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO DE FIN DE GRADO EN INGENIERÍA DE COMPUTADORES

Director: Dr. Alberto A. Del Barrio

Autorización de difusión y utilización

Los abajo firmantes, matriculados en el Grado en Ingeniería de Computadores de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores el presente Trabajo de Fin de Grado: "**Emulador Software para el cálculo de primos de Mersenne**", realizado durante el curso académico 2012-2013 bajo la dirección del Dr. Alberto Del Barrio en el Departamento de Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

De Cristina:

A mi padre y a mi hermano, porque sé que seguís conmigo.

A mi madre y a mis hermanas, porque habéis estado aquí.

De Pablo:

A vosotros, por darme todo lo que tengo e inculcarme todo lo que soy, la familia es lo primero.

"El trabajo bien hecho siempre será mi mejor almohada"

Índice

Índice de figuras

Índice de Tablas

Resumen

El objetivo de este proyecto consiste en el desarrollo de un emulador software para el cálculo de primos de Mersenne, en concreto para la resolución de los problemas que conlleva el tratamiento de números de una magnitud tan grande como los mencionados primos.

El estudio de los números primos de Mersenne es importante dado que por sus propiedades características son especialmente útiles en distintas aplicaciones, especialmente criptográficas, como el protocolo RSA o la criptografía basada en curvas elípticas.

La presente memoria se halla dividida en siete capítulos, cada uno de los cuales describiremos brevemente a continuación.

El primer capítulo incluye la introducción, que contiene información general acerca del tema que nos ocupa, datos históricos sobre los descubrimientos de los números primos de Mersenne, así como su definición y algunas de sus propiedades características. Además, se habla acerca de los problemas que se pretenden resolver con el trabajo elaborado y las motivaciones del mismo.

En el segundo capítulo se puede encontrar información acerca de los antecedentes que han servido como base a este proyecto. Datos acerca del proyecto GIMPS, y de programas ya desarrollados que guardan relación con nuestro problema, como Prime95 o Mprime.

En el tercer capítulo se explica en detalle la arquitectura del emulador desarrollado, el sistema de representación empleado, una descripción detallada de la biblioteca de funciones desarrollada para construir el emulador y las mejoras propuestas al diseño inicial del mismo.

El cuarto capítulo versa sobre CUDA y su uso en este proyecto, este capítulo incluye datos tanto de la arquitectura CUDA, su funcionamiento y algunas nociones básicas de su lenguaje de programación.

El quinto capítulo muestra y discute los resultados obtenidos con el emulador y sus diversas optimizaciones.

El sexto capítulo hace referencia a las conclusiones obtenidas de la investigación y a las vías de investigación de trabajo futuro que proponemos en caso de continuar con este proyecto, como son el uso de una aritmética RNS, el uso del estándar MPI o el uso de arquitecturas más modernas que soporten CUDA.

Por último el séptimo capítulo contiene la bibliografía consultada para la elaboración de este proyecto.

Palabras clave: *Primos de Mersenne, primalidad, test de Lucas-Lehmer, CUDA, Sumador Kogge-Stone, Aritmética modular.*

Abstract

The objective of this project consists of developing a Software Emulator focused on searching Mersenne Primes, particularly addressing the tremendous problems associated with such huge numbers, namely: precision and performance.

The study of Mersenne primes is important due to their peculiar properties, that are specially useful in some cryptographic applications, such as the RSA protocol or the ECC (Elliptic Curve Criptography).

This report is divided in seven chapters, which will be described in the following paragraphs.

The first chapter includes the introduction, which contains generic information about the subject that will be studied, history about Mersenne primes discoveries, their definition and some of their special properties. Besides, this chapter deals with the problems that we expect to solve and the motivations that caused this work to be done.

The second chapter contains information about previous work that support this project. Information concerning project GIMPS, and other programmes belonging to the state of the art and which are related to our work, such as Prime95 or Mprime, is described in this chapter.

The third chapter will explain the emulator's architecture, data representation, and a description about the library implemented as well as the improvements proposed over the baseline design.

The fourth chapter is about CUDA and its use in this project. Besides, it includes information about CUDA architecture, its way of working and some basic ideas about its programming language.

The fifth chapter shows and discusses the results obtained with the baseline version of the emulator, as well as with its different optimizations.

The sixth chapter gives some remarks about the project. Moreover, it describes our suggestions for continuing this work in the future, as for example: the utilization of the Residue Number System (RNS), the MPI standard or the use of the most up-to-date CUDA architectures.

Finally, the seventh chapter contains the bibliography consulted to develop this project.

Keywords: *Mersenne Primes, primality, Lucas-Lehmer Test, CUDA, Kogge-Stone Adder, modular arithmetic.*

1 Introducción

1 Un breve recorrido por la historia de los Números Primos de Mersenne.

Ya desde el Génesis de la Aritmética [1], se mencionan los números primos. Desde entonces la búsqueda de estos particulares números ha fascinado a grandes matemáticos a lo largo de toda la historia.

En este proyecto nos centraremos en la búsqueda de unos números primos más concretos, los números primos de Mersenne. Se denominan así en honor al monje y filósofo Marin Mersenne.

Aunque reciben su denominación en honor a Marin Mersenne, a lo largo de la Historia estos números han sido analizados en varias ocasiones por matemáticos de renombre, incluso anteriores a Mersenne, y han sido protagonistas de varias anécdotas curiosas.

Inicialmente se pensaba que cualquier número de la forma 2^p -1 (con p primo) sería primo en todos los casos. No fue hasta 1536 que Hudalricus Regius fue capaz de demostrar que esto no era cierto para el caso $2^{11} - 1$, dado que es el resultado de la multiplicación 23*89.

Años después, Pietro Cataldi confirmó que 2^{17}-1 y 2^{19}-1 son primos, sin embargo afirmó lo mismo para los exponentes 23, 29, 31 y 37, afirmación que acabó mostrándose incorrecta, para los números 29 y 31 en 1640, por intervención de Fermat y mucho más tarde, en 1738, por Euler.

En 1644, fue, por fin, Mersenne, en su *Cognitata Physica-Mathematica,* quien postuló que los números de la forma 2^n-1 eran números primos para :

$$n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127 \text{ y } 257$$

Y afirmando para todos los exponentes menores que 257 el hecho de que 2^n -1 no era primo.

Como podemos observar Mersenne incluyó en su lista erróneamente los exponentes 67 (la factorización de $2^{67} - 1$ corresponde a 193707721*761838257287) y 257 (cuya factorización haría pública F.N. Cole mucho más adelante).

El propio Mersenne reconoció que no había podido comprobar la primalidad de todos aquellos números, pero tampoco lo logró ninguno de sus coetáneos. Fue necesario esperar alrededor de cien años, para ver confirmada la primalidad de 2^{31}-1, gracias a Euler.

De nuevo hay que esperar alrededor de un siglo para hallar novedades entre los números primos de Mersenne.

Fue entonces cuando aparecieron las primeras calculadoras mecánicas; los siguientes primos de Mersenne fueron encontrados haciendo uso de ellas: en 1876 Édouard Lucas verifica la primalidad de $2^{127} - 1$ y siete años más tarde Ivan Mikheevich Pervushin demuestra que 2^{61}-1 es primo. Por tanto, es así como se encuentra la primera omisión en la lista de Mersenne.

Más tarde, en 1903 F.N.Cole en una reunión de la American Mathematical Society demostró que $2^{257} - 1$ correspondía al resultado de la multiplicación de 193707721 y 761838257287.

En 1911 y 1914 Ralph Ernest Powers demuestra dos omisiones más en la lista de números primos de Mersenne, 2^{89} -1 y 2^{107}-1.

Sin embargo, no es hasta 1947 que el rango $n < 258$ se ve totalmente comprobado y se determina que la lista correcta es:

$$n = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107 \text{ y } 127.$$

Gracias a la suma del Test de Lucas-Lehmer, del que hablaremos con más extensión más adelante, y a los inicios del desarrollo de la computación, la verificación de primos de Mersenne avanzó significativamente.

En esta nueva etapa de la búsqueda de primos de Mersenne podemos resaltar cómo se encontró M_{521}, en enero de 1952, gracias al programa escrito y ejecutado por el profesor R.M. Robinson bajo la dirección de Lehmer, y cómo sólo unas horas más tarde se encontró M_{607}. Durante los meses siguientes fueron encontrados M_{1279}, M_{2203}, y M_{2281}.

Posteriormente, los siguientes primos de Mersenne fueron descubiertos incrementando la potencia de cálculo con computadores como el Cray I o el IBM 7090.

En la actualidad la búsqueda se basa en la computación distribuida. En 1996, surge el proyecto GIMPS, el cual se dedica a la búsqueda de números primos de Mersenne mediante el uso colaborativo de recursos a través de la red y la computación distribuida. En la actualidad han llegado a acumular una capacidad de cómputo que les habría otorgado el puesto 330 en el ranking del top500.

Desde el inicio del proyecto, se han encontrado 13 nuevos primos de Mersenne, el último de ellos este mismo año, el 25 de Enero. $M_{57.885.161}$ con la nada despreciable longitud de 17.425.170 cifras.

En total, a lo largo de la Historia, tan solo se conocen 48 primos de Mersenne.

2 Definición, propiedades y aplicaciones de los números primos de Mersene.

Se dice que un número es de Mersenne si es un número de la forma:

$$M_n = 2^n - 1$$

Adicionalmente se dice que un número de Mersenne es primo de Mersenne si es un número primo de la forma:

$$M_p = 2^p - 1 \text{ con } p \text{ primo.}$$

El estudio de este tipo de números supone un área de especial interés por varios motivos, entre ellos las relaciones que se producen entre los primos de Mersenne y otras categorías de números útiles en distintos contextos.

Los números de Mersenne son los *repunit* (del inglés repeated unit) de la base binaria, esta propiedad resulta interesante, sobre todo al realizar operaciones con computadores, ya que éstos trabajan en base binaria, puesto que ciertos cálculos se simplifican en extremo.

Estos números además guardan una estrecha relación con los *números perfectos*, dado el hecho de que si M es un número primo de Mersenne entonces $M \cdot (M+1)/2$ es un número perfecto.

Un número perfecto es un número natural que es igual a la suma de sus divisores propios positivos sin incluirse él mismo. Dichos números también son conocidos desde los tiempos de Euclides.

Mencionaremos también los números dobles de Mersenne, que son primos de la forma:

$$M_{Mp} = 2^{(2^p)-1} - 1 \text{ con } p \text{ primo.}$$

Todos estos números primos son de gran importancia, especialmente en aplicaciones criptográficas como por ejemplo el protocolo RSA, ya que los números primos más grandes que se conocen son números primos de Mersenne. Debido a sus propiedades características, algunas de

las cuales ya hemos mencionado, es posible acelerar procesos asociados a la criptografía de clave pública y a los procesos de criptografía basados en curvas elípticas[2] [3] .

También es reseñable el uso de los números de Mersenne en la generación de números pseudoaleatorios [4] [5] .

3 La búsqueda de la primalidad: el test de Lucas-Lehmer

```
Lucas_Lehmer_Test(p):
 s := 4;
 for i from 3 to p do s := s^2-2 mod 2^p-1;
 if s == 0 then
    2^p-1 is prime
 else
    2^p-1 is composite;
```

La primera versión de este test fue descubierta por Édouard Lucas, y ella fue capaz de demostrar la primalidad de $2^{127} - 1$ en 1876, el mayor primo calculado sin asistencia computacional [6].

Años más tarde, en 1930 el test de Lucas fue mejorado por Derrick Henry Lehmer, dando lugar al test que se utiliza actualmente para comprobar la primalidad de los números de Mersenne y que también se utilizará en este proyecto.

El test de Lucas Lehmer [7] se basa en la comprobación siguiente:

Sea S_n definido con la fórmula recursiva $S_n = S^2_{n-1} - 2$, dado un número de Mersenne

$M_p = 2^p - 1$:

$2^p - 1$ es primo ↔ El módulo de $S_{p-1} / 2^{p-1}$ es 0.

4 Problemas de la búsqueda de primos de Mersenne

Una característica perfectamente reconocible de los números primos de Mersenne es su rápido crecimiento. Se trata de números que crecen de manera exponencial. En otras palabras, poseen una gran magnitud, por lo que son difíciles de representar y almacenar. Este problema, al que denominaremos a partir de ahora de *precisión*, conlleva otra dificultad: a mayor tamaño, el número tardará más en ser analizado. En otras palabras, también tendremos un problema de *rendimiento*.

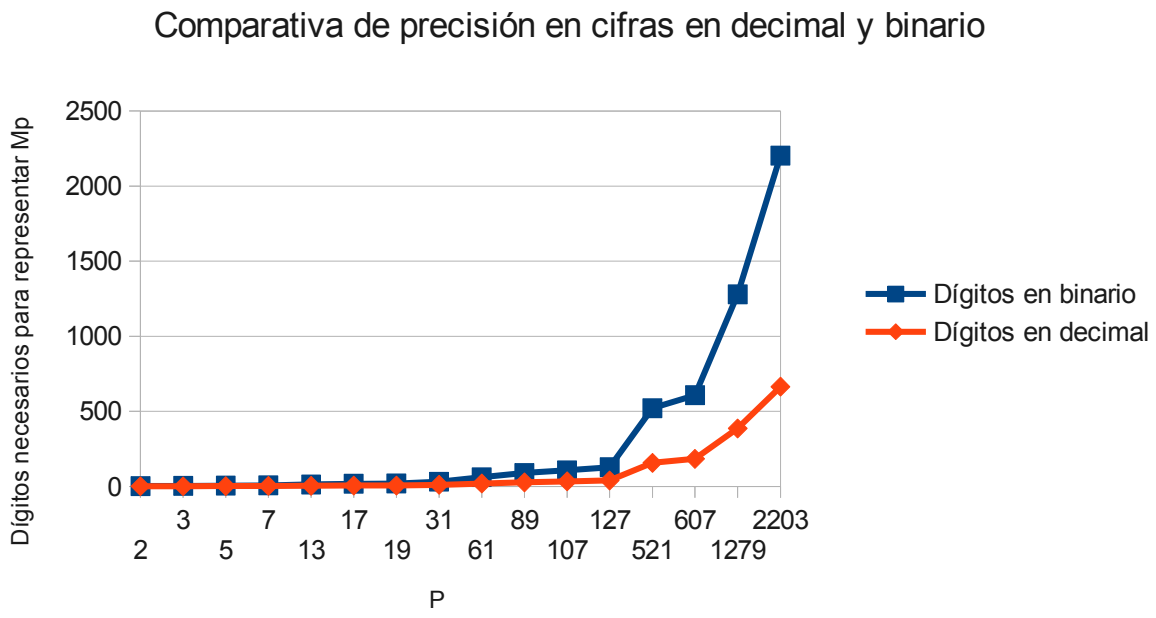

Fig 1.1: Comparativa de precisión en cifras en decimal y binario.

Este proyecto se centrará en el primero de ellos: la precisión, dado que, en cuanto a rendimiento, las posibilidades de superar la capacidad de cálculo obtenida por el GIMPS, con el uso del clúster, carece de sentido, por una simple cuestión de recursos. Sin embargo, creemos que es posible dar un nuevo enfoque a la búsqueda de números primos mediante el análisis y resolución del problema de la precisión. Dicho problema es claramente identificable si observamos la gráfica siguiente, en la que se muestra el crecimiento producido en cuanto a precisión, tanto en cifras decimales como en cifras binarias.

5 *Objetivos y motivaciones de este proyecto.*

El objetivo principal, como hemos mencionado, es la resolución del problema de la precisión en la búsqueda de números primos de Mersenne de este trabajo con los recursos que están a nuestra disposición.

Del objetivo principal derivamos un objetivo secundario, consistente en extender las capacidades de un computador de propósito general mediante el uso, en primer lugar, de un emulador software de la arquitectura hardware que se podría construir para la resolución del problema. Con este enfoque pretendemos darle al computador la posibilidad de efectuar cálculos de mayor complejidad que aquellos que podrían efectuarse utilizando los recursos de dicho computador de forma convencional. En segundo lugar, proponemos el uso de CUDA para acelerar el cálculo de ciertas funciones. De este modo, aprovecharemos los recursos hardware presentes en gran parte de los computadores actuales: las GPUs.

Consideramos importante destacar el motivo por el que hemos elegido la elaboración de un Emulador Software y aceleración mediante CUDA para la consecución de nuestros objetivos, en lugar de la utilización de lenguajes de descripción hardware como VHDL o la implementación física de esta arquitectura mediante un dispositivo de hardware reconfigurable, como bien podría ser el uso de una FPGA.

En resumen, hemos fundamentado nuestra elección en tres razones, principalmente:

- La escalabilidad. Pensamos que un emulador software capaz de resolver el problema de la precisión es más fácilmente escalable que una FPGA, por ejemplo, en la que hay que definir un tamaño máximo para los operandos.

- La disponibilidad de recursos. El acceso a un computador de propósito general que posea una GPU está mucho menos restringido que el acceso a un dispositivo hardware reconfigurable, de modo que, en caso de que otro investigador continúe con este trabajo será más sencillo conseguir los elementos que lo componen.
- El conocimiento a priori de las tecnologías en las que fundamentar el proyecto.

Unido con el anterior punto de la lista, dadas las circunstancias especiales acontecidas durante este curso en relación a los proyectos de fin de grado, este proyecto no pudo comenzarse hasta noviembre. Por tanto fue necesario tomar una decisión en cuanto a la vía que tomaría el proyecto. Las opciones disponibles comprendían la aceleración mediante CUDA o la implementación del dispositivo físicamente. La elección final fue CUDA, principalmente porque es una tecnología más novedosa y presente en casi todos los computadores actuales. Además, la programación en CUDA requiere conocimientos de programación en C++, y los integrantes del equipo poseían conocimientos avanzados de programación en C/C++, de modo que el avance del proyecto, presumiblemente, sería más rápido y efectivo con la utilización de esta tecnología.

2 Introduction

1 A brief tour through the Mersenne primes history.

Prime numbers were very well known since the beginning of Arithmetic. Their searching has fascinated the mosth important mathmaticians through the whole History.

In this project, we will be focused on a concrete type of primes: the Mersenne ones. They receive this name because of the monk and philosohper Marin Mersenne.

Nevertheless, although they where named in honor of Marin Mersenne, throughout History, these numbers have been analyzed repeatedly by renowned mathematicians, even prior to Mersenne, and have been involved in several curious anecdotes.

Initially it was thought that any number of the form 2^p -1 (being p prime) would be prime in all cases. It was not until 1536 that Hudalricus Regius was able to show that this was not true in the case $2^{11} - 1$, as it is the result of the multiplication 23 * 89.

Some years later, Pietro Cataldi confirmed that 2^{17}-1 and 2^{19}-1 are primes. However, he said the same for the exponents 23, 29, 31, 37, but this was proved to be wrong for 23 and 37 by Fermat in 1640, and for 29 and 31 by Euler in 1738..

In 1644, it was finally Mersenne, in his *Cognitata Physica-Mathematica*, who postulated that the numbers with the form 2^n-1 were prime numbers for:

n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127 and 257

And finally, he also claimed that for the rest of exponents lower than 257, 2^n -1 was not prime.

As we can see, Mersenne mistakenly included in his list of exponents 67 (while 2^{67}-1 corresponds to 193707721 * 761838257287) as well as 257 (the F.N. Cole factorization public would much later).

Mersenne himself acknowledged that he could not verify the primality of all those numbers, but neither did any of his contemporaries. It was about a hundred years later, when the primality of 2^{31}-1 is confirmed by Euler.

Again it was necessary to wait about a century to find new Mersenne primes.

That was when the first mechanical calculators appeared; the following Mersenne primes were found by making use of them: in 1876 Édouard Lucas checks the primality of 2^{127}-1 and seven years later Ivan Mikheevich Pervushin shows that 2^{61}-1 is prime. Thus, in this way the first omission in the list of Mersenne is found .

Later in 1903 F.N.Cole, during an American Mathematical Society meeting showed that 2^{257}-1 corresponded to the result of multipliying 193707721 and 761838257287.

In 1911 and 1914 Ralph Ernest Powers demonstrates two further omissions in the list of Mersenne primes, 2^{89} - 1 and 2^{107}-1.

In 1947 the rank n <258 is fully tested and the correct list is determined to be:

n = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107 and 127.

Thanks to the conjunction of the Lucas-Lehmer test, discussed in depth below, and the early development of computing, the verification of Mersenne primes advanced significantly.

In this new era for the Mersenne primes' search, it can be highlighted how M_{521} was found in January 1952, thanks to the program written by Professor R.M. Robinson who was led by Lehmer, and how a few hours later were found M_{607}. During the following months M_{1279}, M_{2203}, and M_{2281}.

Subsequently, the following Mersenne primes were discovered by increasing computing power with computers such as the Cray I or the IBM 7090.

Nowadays the search is based on distributed computing. In 1996, the GIMPS project appears. This project is devoted to find Mersenne primes using collaborative resources through the network and distributed computing. Currently have come to accumulate computing power that would have granted GIMPS with the rank 330 in the ranking of the top500[1][2].

Since the beginning of the project, 13 new Mersenne primes have been found, the last one this year, on January 25th. $M_{57.885.161}$ which possesses the impressive length of 7,425,170 decimal digits.

Overall, throughout History, only 48 Mersenne primes are known.

2 ***Definition, properties and applications of Mersenne primes.***

A Mersenne number is defined as:

$$M_n = 2^n - 1$$

Additionally it is said that a Mersenne number is a Mersenne prime if :

$$M_p = 2^p - 1, p \text{ prime.}$$

The study of such numbers is an area of special interest for several reasons, including the relationships that exist between Mersenne primes and other categories of numbers useful in different contexts.

Mersenne numbers are the repunit (Repeated unit) binary base. This property is interesting especially when working with computers, as they work in binary base, since certain calculations are very simplified.

These numbers are also closely related to the perfect numbers. Given the fact that if M is a Mersenne prime then $M \cdot (M+1) / 2$ is a perfect number.

A perfect number is a natural number which is equal to the sum of its positive divisors without including himself. These numbers are also known since the time of Euclid.

Finally, we will also mention the double Mersenne primes, that are primes with the form:

$$M_{Mp} = 2^{(2^p)-1} - 1 \text{ with } p \text{ prime.}$$

All these primes really relevant, especially in cryptographic applications such as the RSA protocol, since the largest known prime numbers are the Mersenne ones. Due to its features, some of which we have already mentioned, it is possible to speed up processes associated with public key cryptography and processes based on elliptic curve cryptography [3] [4].

Furthermore, it should be noted the importance of Mersenne primes in pseudorandom number generation [5] [6] .

3 *The pursuit of primality.*

```
Lucas_Lehmer_Test(p):
 s := 4;
 for i from 3 to p do s := s^2-2 mod 2^p-1;
 if s == 0 then
    2^p-1 is prime
 else
    2^p-1 is composite;
```

The first version of this test was developed by Édouard Lucas, thanks to it in 1876 he was able to prove the primality of 2^{127}-1, the largest prime calculated without computer assistance [7].

Years later, in 1930 the Lucas test was improved by Derrick Henry Lehmer, leading to the test that is currently used to check the primality of Mersenne numbers which will also be used in this project.

The Lucas Lehmer test is based on the following assumption:

Let S_n to be defined with the recursive formula $S_n = S^2_{n-1} - 2$, hence given a Mersenne number[8]:

$$M_p = 2^p - 1$$

$2^p - 1$ is prime $\leftrightarrow$ module $S_{p-1} / 2^{p-1}$ is 0.

4 *The problems of Mersenne primes search*

An easily recognizable feature of Mersenne primes is their rapid growth. These numbers grow exponentially. In other words, they have a large magnitude, so they are are difficult to represent and store. This problem, which will be named precision from now on, involves another difficulty: the larger the number, the longer it will take to be analyzed. In other words, there will also be a performance problem.

This project will be focused on the first one: the precision, since in terms of performance, the chances of overcoming the computing power obtained by the GIMPS, which uses a huge grid, are unrealistic, mainly because of the lack of resources. However, we believe it is possible to develop a new approach for finding Mersenne primes by analyzing and solving the precision problem. This is illustrated by figure 2.1, which shows the number of digits, both in decimal and binary format, to represent Mersenne primes with regard to the exponent..

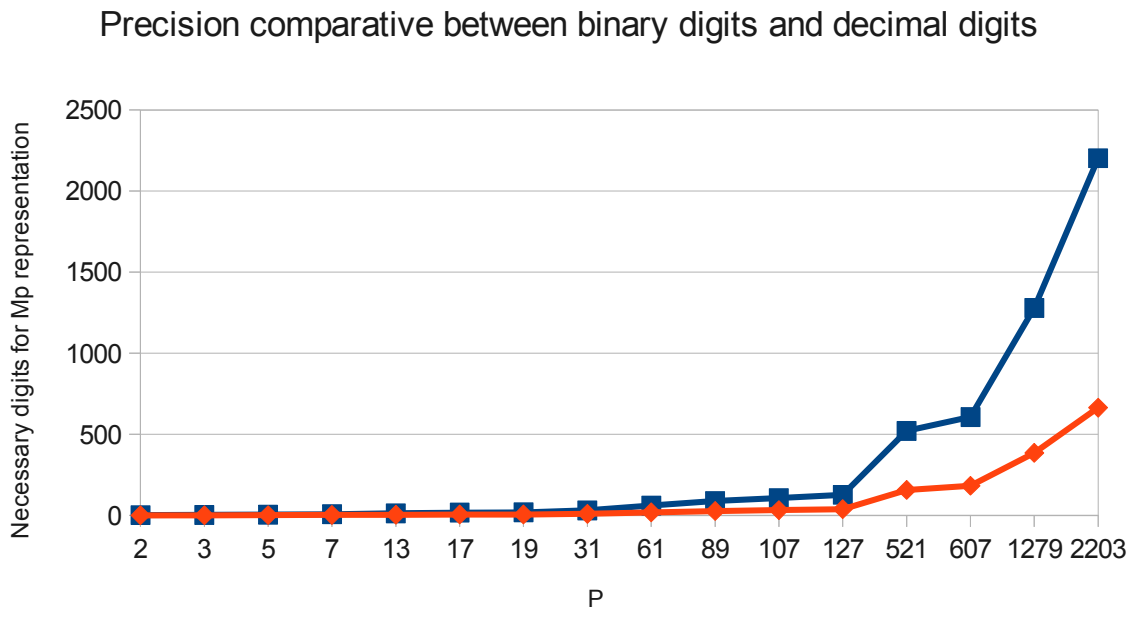

Fig 2.1: Precision comparative between decimal digits and binary digits.

5 *Objectives and motivations of this project.*

The main objective of this work, as mentioned, is the resolution of the precision problem for finding Mersenne primes, given a relatively reduced set of resources .

A secondary target derives from this main objective, which consists of extending the capabilites of a general purpose computing by using a software emulator. This emulator mimics the hardware architecture which could be built to solve the Mersenne primes' search. With this approach, our purpose is to provide the computer with the ability to perform more complex calculations than those that could be realized with conventional resources. Thus, we also propose the use of CUDA to accelerate the computation of certain functions. In this way, we will utilize the hardware resources that are available in the majority of the computers nowadays: the GPUs.

Now we will explain why we have chosen the development of a software emulator and its later acceleration using CUDA to achieve our goals, instead of considering hardware description languages such as VHDL or the physical implementation of this architecture using a reconfigurable hardware device, as an FPGA.

In summary, we have based our choice on three reasons, namely:

- The scalability. We think that a software emulator capable of solving the precision problem is more easily scalable than an FPGA, for example, where it is required to define a maximum size for the operands.

- The availability of resources. The access to a general purpose computer with a GPU is less restricted than a reconfigurable device. Hence, in this way we believe that it will be easier for any researcher, including ourselves, to continue improving the emulator.

- A priori knowledgte about the technologies that support the project.

Moreover, and related to the last item of the aforementioned list, provided that the project could not start until November, it was necessary to accelerate the deployment of an initial prototype for meeting the deadlines. Hence, considering our greater C/C++ programming skills and our interest in learning about a hot-topic as CUDA, we made this choice.

3 Trabajo relacionado.

1 El proyecto GIMPS

En esta sección haremos un repaso algunos estudios previos a nuestro proyecto y que han servido de punto de partida del mismo.

Comenzaremos hablando del proyecto GIMPS, el cual ya ha sido mencionado con anterioridad, pero en esta ocasión lo abordaremos con más detalle.

El proyecto GIMPS es una organización dedicada a la búsqueda de los números primos de Mersenne. Fue fundado en enero de 1996, y ha tenido éxito en su empresa, ya que en en el GIMPS han sido descubiertos trece números de los cuarenta y ocho números primos de Mersenne conocidos. De esos trece números, once han sido el "número primo más grande conocido" en el momento de su descubrimiento[8].

1 Prime95[9, p. 95]

En el proyecto GIMPS se utiliza el programa Prime95, escrito por George Woltman, para la búsqueda de números primos de Mersenne.

La mayor parte del código de Prime95 está disponible públicamente, con la excepción del fragmento utilizado para el cálculo de las sumas de comprobación (checksums) por motivos de seguridad. Aunque su código se encuentre disponible públicamente, Prime95 no se considera como un programa de código libre puesto que con su uso han de aceptarse ciertas condiciones de distribución.

Actualmente el programa Prime95 utiliza el conjunto extendido de instrucciones x86 AVX (Advanced Vector Extension) presente en las arquitecturas Sandy Bridge e Ivy Brigde de Intel para mejorar su rendimiento.

1 Soporte para aceleración GPU

Actualmente el programa Prime95 carece de soporte para aceleración mediante GPUs, aunque Woltman ha comunicado que dicho soporte está en desarrollo.

2 Límites a la precisión

Hasta la versión 24, el programa prime95 no podía realizar el test sobre números de Mersenne superiores a $2^{79300000} - 1$. Actualmente su limitación de precisión se encuentra en $2^{256000000} – 1$.

Cuando se habla de Prime95 se hace referencia a las versiones para MacOSX y Windows. Para sistemas Linux existe una versión equivalente denominada Mprime.

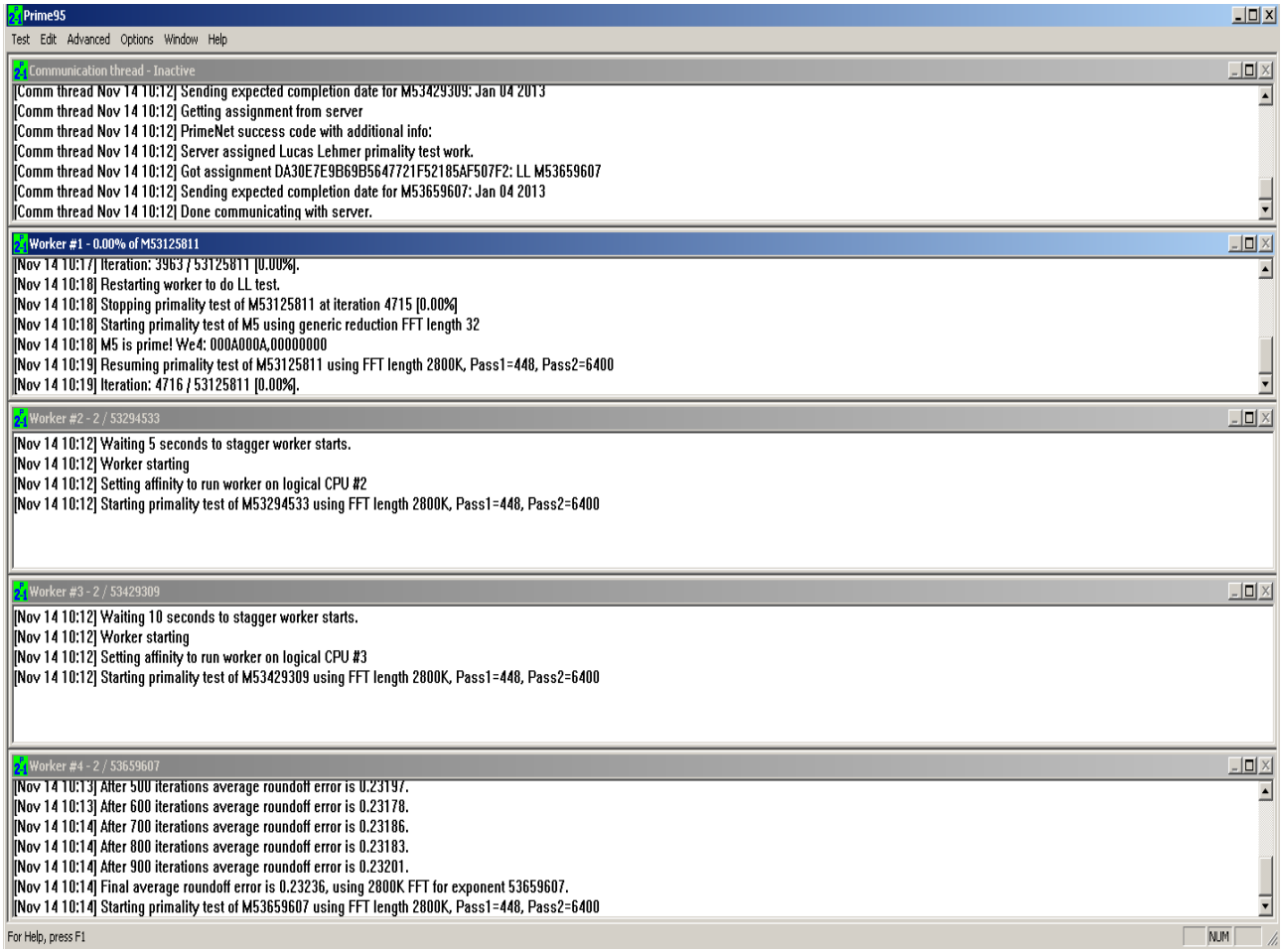

Fig 3.1: El programa prime95

2 Mprime

Se trata de la versión de Prime95 para los sistemas operativos Linux. Dicha versión posee todas las funcionalidades de Prime95 a excepción de la interfaz gráfica de usuario.

2 Glucas, Mlucas

Estos programas son implementaciones completamente libres del Test de Lucas-Lehmer, carecen de las restricciones impuestas por Prime95.

3 *Aceleración CUDA*

En el plano de aceleración nos ha inspirado especialmente el proyecto de código abierto GPULucas. Desarrollado por Andrew Thall, supone una revolución en las metodologías de resolución del test de Lucas-Lehmer incluyendo la implementación del método de Crandall & Fagin IBDWT (Irrational Base Discrete Weighted Transform) [10]. Con esto, se convierte en el referente actual obteniendo los tiempos más bajos de resolución en esta arquitectura.

4 Arquitectura del emulador.

En este capítulo trataremos en profundidad los medios utilizados para la consecución del primero de los objetivos propuestos: la precisión. Superar la barrera impuesta por los tipos del computador (int 32-bits, long 64-bits) es fundamental para ser capaces de detectar primos de Mersenne que requieren miles de bits para ser representados.

Para ello, emularemos el funcionamiento de un dispositivo hardware especialmente orientado a la búsqueda de números primos de Mersenne. Este emulador está programado en lenguaje C, utilizando el compilador libre gcc sobre la plataforma Linux Ubuntu.

Se decidió utilizar el lenguaje C frente a otros lenguajes debido al mayor rendimiento ofrecido por C frente a otros lenguajes como Java o C#.

Este emulador reproducirá con exactitud la hipotética arquitectura hardware que implementase el test de Lucas-Lehmer, pero además aprovechará algunas de las características del lenguaje en el que está implementado (C) para la resolución de la precisión, como por ejemplo el uso de la memoria dinámica. Haciendo uso de estas ventajas podremos acometer el análisis de números de mayor extensión sin la necesidad de modificar el diseño, al contrario de lo que ocurriría en plataformas puramente hardware como FPGAs o ASICs.

1 La biblioteca de operaciones.

En esta sección se describirá con detalle la biblioteca de funciones utilizada para implementar el emulador.

1 Representación de datos

Para resolver el problema de la precisión decidimos emplear una representación basada en vectores de enteros, sacrificando consumo de memoria a cambio de eliminar la limitación derivada del ancho permitido por el lenguaje C. Este tipo de datos emulará el vector de bits que se utilizaría en una plataforma hardware.

En este emulador utilizaremos un sistema de representación basado en pares vector-entero, en los que el vector contiene la representación binaria del número, y el entero indica la longitud de éste. De este modo, además, lograremos trabajar con números, en complemento a 2, de longitud variable y eliminaremos la necesidad de replicar módulos o concatenar resultados al finalizar ciertas operaciones.

Por ejemplo, en una operación de suma según esta representación tendríamos:

pVector1: 1101

nSize1: 4

pVector2: 1100

nSize2: 4

El resultado de nuestra operación generaría el siguiente vector:

pResult: 11001

nSizeRes: 5

En lugar de 1001 con carry = 1.

2 *Operaciones de la Biblioteca*

Como primer paso en el desarrollo del emulador se procedió a la implementación en C de las operaciones básicas de un conjunto universal de puertas el cual permitiera la implementación posterior de operaciones más complejas:

1 *Operación AND*

int andBinary (int a, int b)

Esta función devolverá el resultado de efectuar la operación lógica AND sobre los bits de entrada *a* y *b*.

2 *Operación OR*

int orBinary (int a, int b)

Esta función devolverá el resultado de efectuar la operación lógica OR sobre los bits de entrada a y b.

3 *Operación NOT*

int notBinary (int a)

Esta función devolverá el resultado de efectuar la operación lógica NOT sobre el bit de entrada a.

4 *Operación XOR*

int xorBinary (int a, int b)

Esta función devolverá el resultado de efectuar la operación lógica XOR sobre los bits de entrada a y b.

A partir de las operaciones anteriores implementamos las operaciones enumeradas a continuación.

5 *Suma de 2 bits*

int sum2bits (int a, int b, int cin, int cout)*

Esta función representa el comportamiento de un sumador completo o Full Adder. [11]

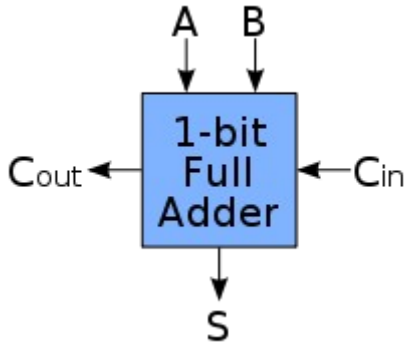

Fig 4.1: Esquema de un sumador completo de 1 bit.

Un sumador completo suma tiene como entrada tres bits, A y B, que son los operandos y Cin que es el acarreo de entrada producido por la etapa anterior de cálculo. Produce dos bits de salida, el primero, S es el resultado de la suma y Cout que es el acarreo de salida.

La tabla de verdad de un sumador completo es como sigue:

Entradas			**Salidas**	
A	***B***	C_{in}	C_{out}	***S***
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

Tabla 4.1: Tabla de verdad de un sumador completo

Simplificando la tabla anterior, podemos obtener las siguientes funciones de conmutación para S y Cout.

$$S = ((A \oplus B) \oplus Cin)$$

$$Cout = (A \cdot B) + (Cin \cdot (A \oplus B))$$

De esta forma, el diseño de cada Full Adder sería el mostrado en la figura Fig 4.2.

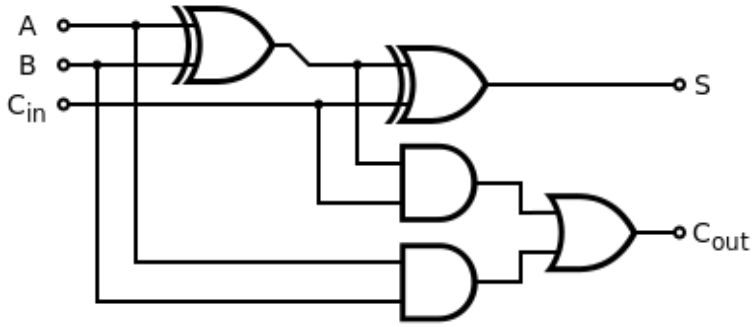

Fig 4.2: Diagrama lógico de un sumador completo.

6 Suma de N bits

void AddOperation(int pVector1[], int nSize1, int pVector2[], int nSize2, int pResult[], int nSizeRes)*

Esta operación recibe por parámeto la representación correspondiente a dos números binarios en los arrays de enteros pVector1 y pVector2, y devuelve el resultado de la suma efectuada en el array pResult, indicando su tamaño en la variable de salida nSizeRes.

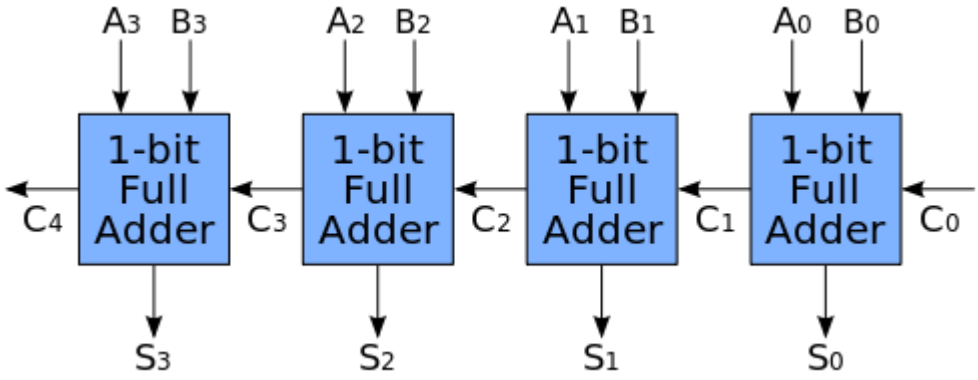

Fig 4.3: Ripple Carry Adder de 4 bits.

La suma de la versión básica de este emulador se implementa con un Ripple Carry Adder (RCA). Se trata de un sumador de N bits creado a partir de N sumadores completos de dos bits, como puede observarse en la figura 4.3 para un N = 4, en el que la entrada Cin de cada sumador completo es la salida Cout del anterior. Recibe este nombre (Ripple Carry Adder) del inglés "ripple", propagar, ya que cada acarreo se propaga al siguiente sumador.

La estructura de este tipo de sumadores es simple y permite que sean diseñados con cierta facilidad. Sin embargo, son lentos, ya que cada sumador debe esperar el retardo de propagación del bit de acarreo anterior para poder realizar su operación. Este proceso se da para todos los sumadores que componen el sumador de N bits. En resumen, para calcular la suma de un número de N bits será necesario esperar el retardo de propagación de los N sumadores que lo componen.

7 *Resta de N bits*

void SubtractOperation(int pVector1[], int nSize1, int pVector2[], int nSize2, int pResult[], int nSizeRes)*

Esta operación recibe el minuendo de la resta en el array pVector1 y el sustraendo de la misma en pVector2, devolviendo la diferencia entre ambos en el array de enteros pResult. Para calcular dicha diferencia esta operación se sirve del cálculo del complemento a dos del sustraendo y acto seguido se lo suma al minuendo. Para conseguir este objetivo la función SubstractOperation se sirve de otra función de la biblioteca:

8 *Complemento a 2*

*void Complement2 (int pVector[], int nSize, int pResult [], int * nSizeRes)*

Esta función devuelve en pResult el complemento a 2 del número indicado por pVector.

El complemento a dos de un número N cuya representación en base binaria consta de n bits está definido como:

$$\text{Complemento a 2 de } N = 2^n - N$$

Para efectuar el cálculo del complemento a dos de un número, si éste es positivo no se efectúan cambios en él. Si el número es negativo será necesario invertir el valor de cada una de sus cifras, o lo que es lo mismo, calcular el complemento a uno, o NOT, y posteriormente sumarle 1.

Otra opción para el cálculo del complemento a dos consiste en la utilización de la conversión rápida, consistente en copiar el número binario comenzando por la cifra menos significativa hasta dar con el primer '1' de la secuencia. Una vez copiado este uno, el resto de dígitos se invierten.

9 Multiplicación

void MultOperation(int pVector1[], int nSize1,const int pVector2[], int nSize2, int pResult[], int nSizeRes)*

Con esta función se emula el funcionamiento de un multiplicador que utiliza el algoritmo de sumas y desplazamientos. El resultado se inicializa a cero. En cada iteración se evalúa el bit menos significativo del multiplicador, y si éste es '1', se suma el multiplicando desplazado a la izquierda tantas posiciones como bits menos significativos haya anteriores al bit del multiplicador examinado.

10 División

void DivOperation(int pVector1[], int nSize1,const int pVector2[], int nSize2, int pResult[], int nSizeRes)*

Esta función emula el funcionamiento de un divisor que utiliza el algoritmo de de restas sucesivas. A continuación mostraremos el esquema que hemos seguido para la realización de la misma.

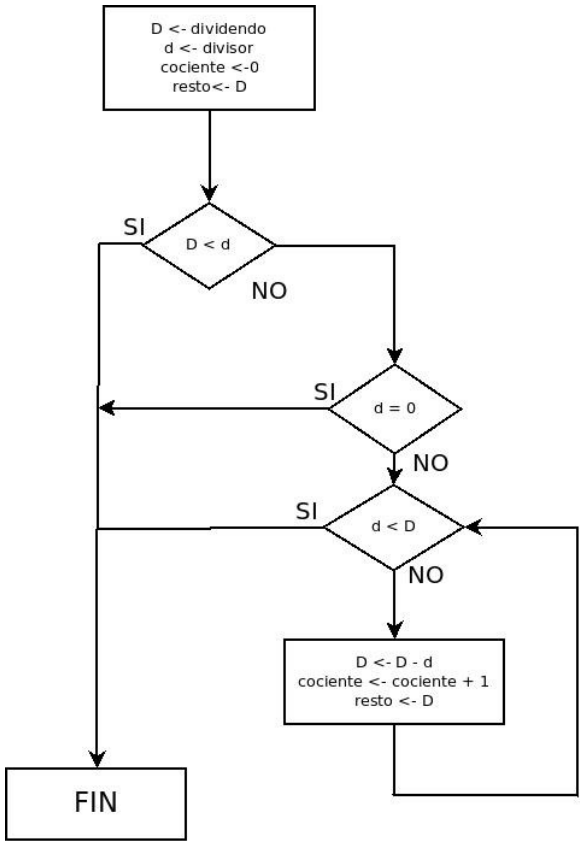

Fig 4.4: Diagrama de flujo de la división por restas sucesivas.

11 Módulo

void ModOperation(int p, int xVector[], int nSizeX, int mVector[], int nSizeM, int pResult[], int nSizeRes)*

Esta función devuelve en pResult el resultado del número indicado en el array xVector módulo mVector. Ha sido implementada haciendo uso de la reducción de Barrett [12].

La reducción de Barret es un algoritmo de reducción introducido en 1986 por P.D. Barrett. Se trata de un algoritmo diseñado para realizar la operación del módulo utilizando multiplicaciones en lugar de divisiones. Para ello, suponiendo que queremos calcular un valor tal que:

$$C = a \pmod{n}$$

Asumiendo que n es constante y que $a < n^2$ se puede realizar el módulo mediante el siguiente algoritmo.

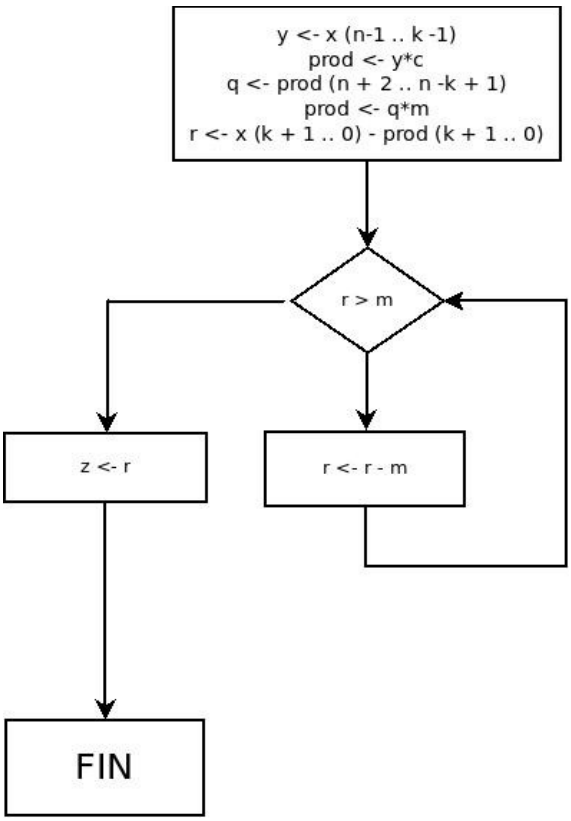

Fig 4.5: Diagrama de flujo del algoritmo de Barrett.

2 Mejoras propuestas.

1 Sustitución del sumador RCA por un sumador Kogge-Stone

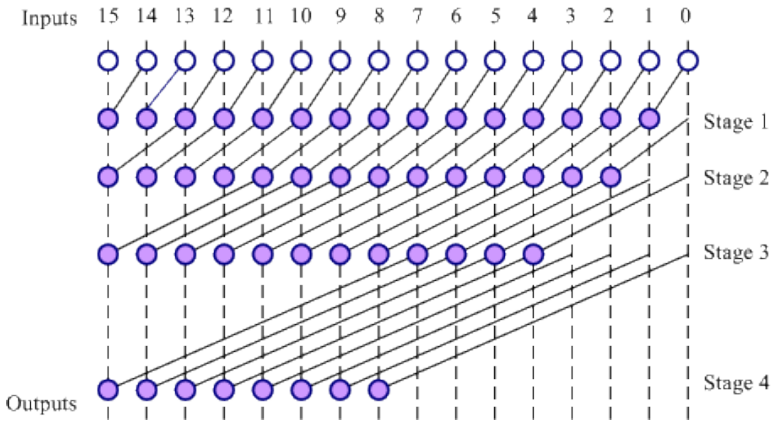

Fig 4.6: Diagrama de un sumador Kogge-Stone de 16 bits.

El sumador implementado en la versión básica es un sumador de propagación. Como hemos comentado con anterioridad, el sumador RCA tiene un retardo de propagación proporcional al número de bits de los sumandos. Para reducir el retardo producido por la suma se ha implementado un sumador Kogge-Stone, perteneciente a la categoría de los Carry-Lookahead adders (sumadores con aceleración del acarreo), que reduce el tiempo requerido para calcular los bits de carry.

En la primera etapa, que se corresponde con los nodos blancos, siendo *a* y *b* dos entradas de n-bits se calculan las siguientes señales:

$$P_i = A_i \oplus B_i$$

$$G_i = A_i \cdot B_i$$

Para las siguientes etapas, correspondientes con los nodos morados, siendo (P', G') y (P'', G'') dos pares de señales de (propagación,generación), con X' más significativa que X'', se calcula la siguiente operación:

$$(P, G) = (P' \cdot P'', G' + G'' \cdot P')$$

2 *Sustitución del módulo de Barrett por la reducción modular 2^p-1*

Si se desea realizar el módulo 2^p - 1 de un número N de 2*p* bits, N módulo 2^p - 1 puede implementarse dividiendo el número N en dos fragmentos de p bits, la parte más significativa y la parte menos significativa, y efectuando la suma de ambas. Si se produce carry de salida en esta operación debe sumarse de nuevo al resultado como carry de entrada [10].

5 CUDA

CUDA son las siglas de ***C**ompute **U**nified **D**evice **A**rchitecture* (Arquitectura Unificada de Dispositivos de Cómputo) que hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por NVIDIA que permiten a los programadores usar una variación del lenguaje de programación C. Dichas herramientas permiten codificar algoritmos que se ejecutarán en GPUs (Graphics Processing Unit) de NVIDIA.

CUDA intenta explotar las ventajas de las GPU frente a las CPU de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un gran número de hilos simultáneos. Este nuevo término se conoce como GPU computing.

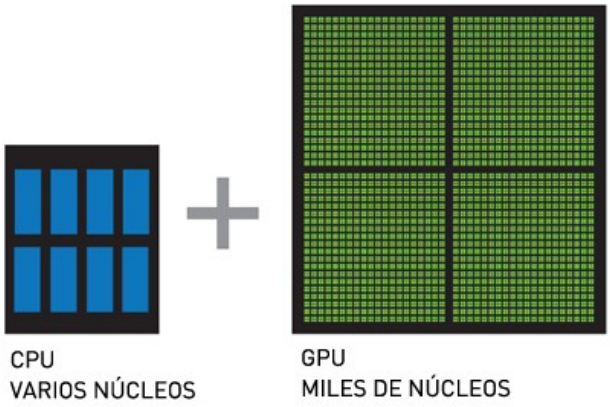

Fig 5.1: CPU vs. GPU

1 ¿Qué es el GPU computing?

GPU Computing consiste en la utilización de GPUs para acelerar ciertos fragmentos de las aplicaciones, en lugar de ejecutarlas sólo en la CPU. Después de que NVIDIA sentara las bases, el cálculo en la GPU se ha convertido rápidamente en un estándar del que disfrutan millones de usuarios de todo el mundo y que emplean prácticamente todos los computadores.

El cálculo en la GPU ofrece un rendimiento de aplicaciones sin igual al descargar en la GPU las partes de la aplicación que requieren gran capacidad computacional, mientras que el resto del código sigue ejecutándose en la CPU. Desde la perspectiva del usuario, las aplicaciones simplemente se ejecutan más rápido.

Una CPU + GPU constituye una potente combinación porque la CPU está formada por varios núcleos optimizados para el procesamiento en serie, mientas que la GPU consta de millares de núcleos más pequeños y eficientes diseñados para el cómputo en paralelo. Por tanto, la idea consiste en ejecutar el código que no puede ser paralelizado en la CPU, y el resto en la GPU.

Hoy en día, la importancia de las GPUs es tal, que podemos encontrar opiniones de expertos como la siguiente:

"Las GPUs han evolucionado hasta un punto en que muchas de las aplicaciones industriales actuales se ejecutan en ellas con niveles de rendimiento muy superiores a los que ofrecerían si se ejecutasen en sistemas multinúcleo. Las arquitecturas informáticas del futuro serán sistemas híbridos con GPUs compuestas por núcleos de procesamiento paralelo que trabajarán en colaboración con las CPUs multinúcleo."

Jack Dongarra

Catedrático del Innovative Computing Laboratory

Universidad de Tennessee

2 El modelo CUDA

CUDA intenta aprovechar el gran paralelismo y el alto ancho de banda de la memoria en las GPU en aplicaciones con un gran coste aritmético frente a realizar numerosos accesos a memoria principal, lo que podría actuar de cuello de botella.

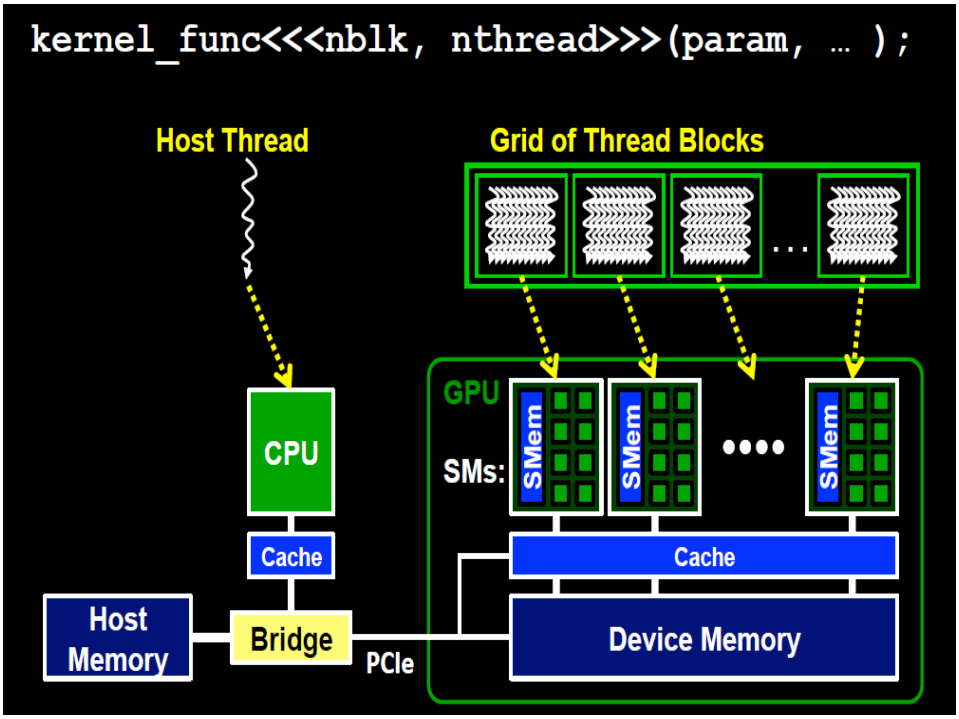

Fig 5.2: modelo de programación CUDA

El modelo de programación de CUDA está diseñado para que las aplicaciones escalen su paralelismo de forma transparente (está basado en el paradigma SIMT - Single Instruction Multiple Thread -) incrementando el número de núcleos computacionales. Este diseño contiene cuatro conceptos clave: **la jerarquía de grupos de hilos**, el **kernel,** las **memorias compartidas** y las **barreras de sincronización**.

3 *Jerarquía de grupos de hilos*

La estructura que se utiliza en este modelo está definido por un *grid*, dentro del cual hay *bloques de hilos* que están formados por como máximo 512 *hilos* distintos.

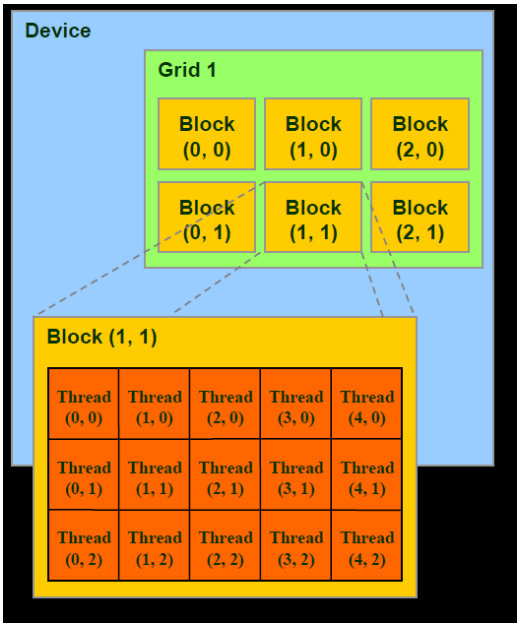

Fig 5.3: Jerarquía de hilos

A continuación daremos una serie de definiciones que serán de utilidad en el resto del capítulo:

Grid: Conjunto de bloques de hilos utilizados en una ejecución.

Bloque de hilos: conjunto de hilos que pueden cooperar entre sí a través de memoria compartida, utilizando mecanismos de sincronización.

Thread: Unidad de abstracción mínima con la que se representa un hilo de ejecución.

Cada thread queda determinado por un identificador único dentro de un bloque, que se accede con la variable threadIdx. Esta variable es muy útil para repartir el trabajo entre distintos hilos. La variable threadIdx tiene 3 componentes (x, y, z), coincidiendo con las dimensiones de bloques de hilos. Así, cada elemento de una matriz, por ejemplo, lo podría tratar su homólogo en un bloque de hilos de dos dimensiones.

Ejemplo: Si threadIdx = {x, 0, 0} el bloque de hilos podrá ser representado por un array. Si por el contrario threadIdx = {x, y, 0} representaría una matriz, y en el caso de que todas sus

componentes tengan valor threadIdx = {x, y, z} se representaría con una matriz de 3 dimensiones.

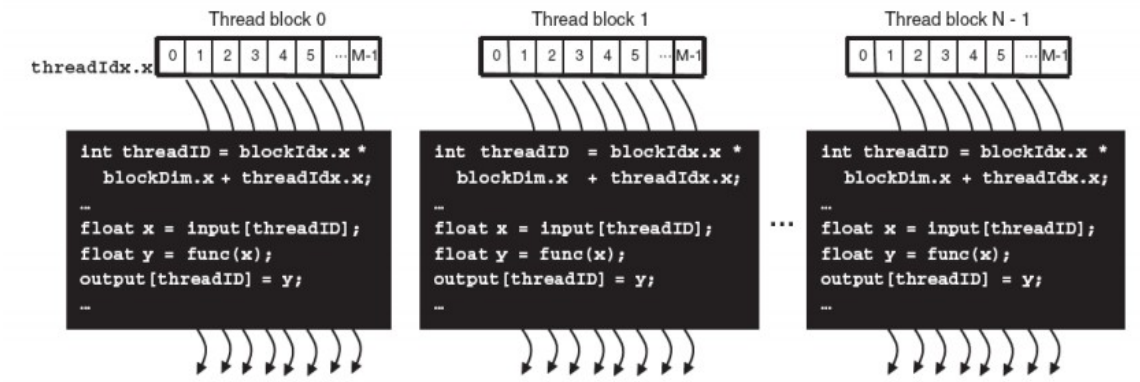

Fig 5.4: Bloques CUDA

Al igual que los hilos, los bloques se identifican mediante blockIdx (en este caso con dos componentes x e y). Otro parámetro útil es blockDim, para acceder al tamaño de bloque.

4 Kernel

El *kernel* está constituido por código paralelo el cual es lanzado y ejecutado como **grid** de hilos. Se define incluyendo __global__ en la declaración. Por ejemplo:

```
//Definición del kernel
__global__ void f(int a, int b, int c)
{
}
```

Por ejemplo, si queremos que una función f calcule la diferencia entre dos vectores A y B y lo almacene en un tercero C:

```
__global__ void f(int* A, int* B, int* C)
{
        int i = threadIdx.x;
        C[i] = A[i] - B[i];
}
```

Esta función se ejecutaría una vez en cada hilo, reduciendo el tiempo total de ejecución en gran medida, y dividiendo su complejidad, O(*n*), por una constante directamente relacionada con el número de procesadores disponibles.

El mismo ejemplo con matrices sería:

```
__global__ void f(int** A, int** B, int** C)
{
        int i = threadIdx.x; //Columna del bloque que ocupa este
determinado hilo
        int j= threadIdx.y; //Fila
        C[i][j] = A[i][j] - B[i][j];
}
```

1 *Invocaciones a un kernel*

En una llamada a un kernel, el tamaño del grid y de bloque deben pasarse por parámetro. Por ejemplo, en el main del ejemplo anterior podríamos añadir:

```
dim3 bloque(N,N); //Definimos un bloque de hilos de N*N
dim3 grid(M,M) //Grid de tamaño M*M

f<<<grid, bloque>>>(A, B, C);
```

En el momento que se invoque esta función, los bloques de un grid se enumerarán y distribuirán por los distintos procesadores libres.

A continuación se muestra un ejemplo de flujo completo de ejecución en cuda, incluyendo las invocaciones pertinentes al kernel.

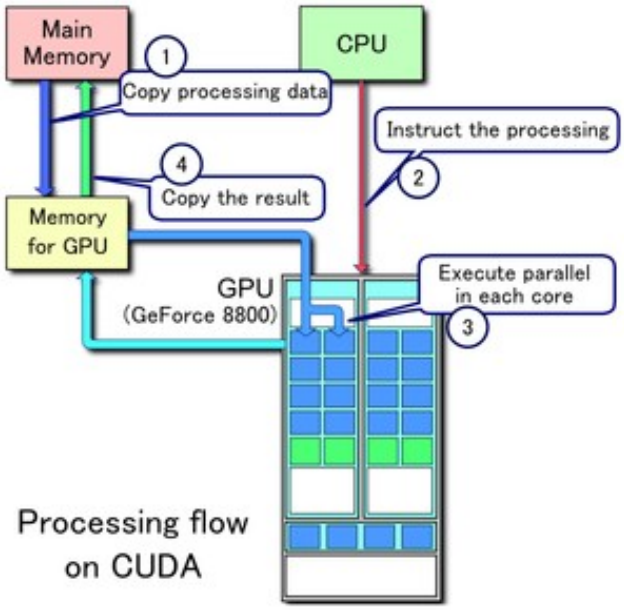

Fig 5.5: Flujo de procesamiento CUDA

2 *Flujo de procesamiento CUDA*

1. Se copian los datos de la memoria principal a la memoria de la GPU.
2. La CPU encarga el proceso a la GPU.
3. La GPU lo ejecuta en paralelo en cada núcleo.
4. Se copia el resultado de la memoria de la GPU a la memoria principal.

5 *Sincronización*

Debido a que los distintos hilos colaboran entre ellos y pueden compartir datos, se requieren unas directivas de sincronización. En un kernel, se puede hacer explícita una barrera incluyendo una llamada a __syncthreads(), gracias a la cual todos los hilos esperarán a que los demás lleguen a ese mismo punto.

6 *Jerarquía de memoria*

Los hilos en CUDA pueden acceder a distintas memorias, unas compartidas y otras no.

- En primer lugar, se encuentra la memoria privada de cada hilo, solamente accesible desde él mismo.
- Cada bloque de hilos posee también un espacio de memoria, compartida en este caso por los hilos del bloque y con un ámbito de vida igual que el del propio bloque.
- Todos los hilos pueden acceder a una memoria global.

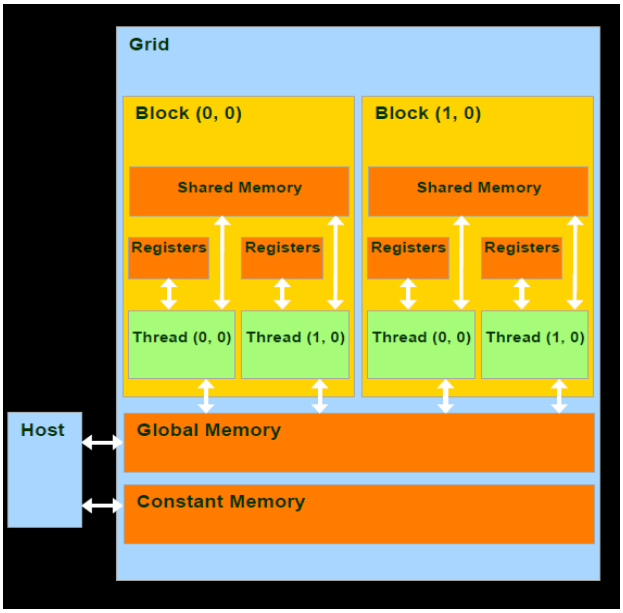

Fig 5.6: Jerarquía de memoria en CUDA

Además, existen otros dos espacios de memoria más, que son de solo lectura y accesibles por todos los hilos. Son la memoria constante y la de texturas. Todas las memorias de acceso global persisten mientras esté el kernel en ejecución.

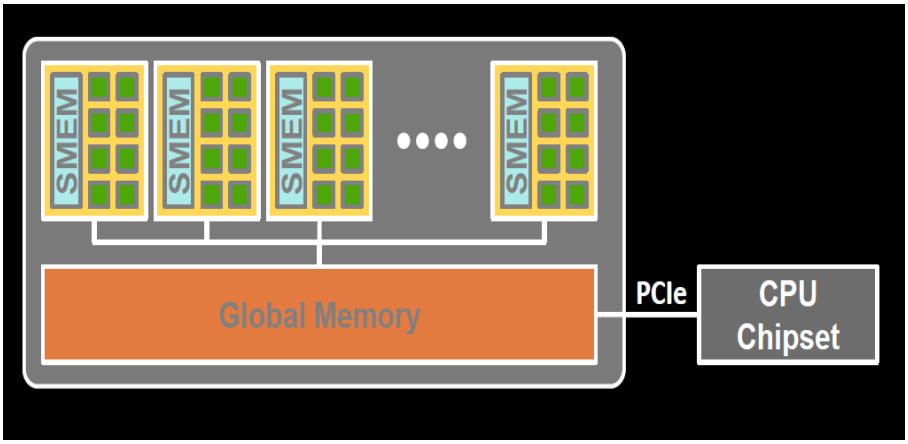

Fig 5.7: Esquema de memoria de acceso global en CUDA

Por último, hay que tener en cuenta el tipo de memoria utilizada en los procedimientos desarrollados, debido a que pueden influir muy negativamente al tiempo en la resolución del problema, tal como se aprecia en la tabla siguiente.

1 Rendimiento de tipos de variables en CUDA

Declaración de variable	Memoria	Penalización
int var;	Registro	1x
int array_var[10];	Local	100x
__shared__ int shared_var;	Compartida	1x
__device__ int global_var;	Global	100x
__constant__ int constant_var;	Constante	1x

7 Arquitectura CUDA

El arquetipo de dispositivo CUDA está basado en multiprocesador(es) que contiene(n) varios procesadores escalares, unidades especiales para funciones trascendentales, una unidad multihilo de instrucciones y una memoria compartida. El multiprocesador crea y maneja los hilos sin ningún tipo de overhead por la planificación, lo cual unido a una rápida sincronización por barreras y una creación de hilos muy ligera, consigue que se pueda utilizar CUDA en problemas de muy baja granularidad, incluso asignando un hilo a un elemento por ejemplo de una imagen (un píxel).

A continuación, evaluaremos las ventajas e inconvenientes de utilizar esta arquitectura.

1 *Ventajas*

CUDA presenta ciertas ventajas sobre otros tipos de computación sobre GPU utilizando APIs gráficas.

- Lecturas dispersas: se puede consultar cualquier posición de memoria.
- Memoria compartida: CUDA pone a disposición del programador un área de memoria de 16KB (o 48KB en la serie Fermi) que se compartirá entre threads. Dado su tamaño y rapidez puede ser utilizada como caché.
- Lecturas más rápidas de y hacia la GPU.
- Soporte para enteros y operadores a nivel de bit.

2 *Limitaciones*

Pese a contribuir a la aceleración de la mayoría de los programas, CUDA también presenta ciertos inconvenientes:

- No se puede utilizar recursividad, punteros a funciones, variables estáticas dentro de funciones o funciones con número de parámetros variable
- No está soportado el renderizado de texturas
- En precisión simple no soporta números desnormalizados o NaNs
- Puede existir un cuello de botella entre la CPU y la GPU por los anchos de banda de los buses y sus latencias.
- Los threads, por razones de eficiencia, deben lanzarse en grupos de al menos 32, con miles de hilos en total.

8 *Dispositivo*

El dispositivo utilizado para el desarrollo de los experimentos ha sido el modelo GTS 250 de NVIDIA, décima generación de GPUs de la marca, la cual dio lugar a la creación de la arquitectura Tesla.

Fig 5.8: GPU NVIDIA GTS 250

Sus características son las siguientes:

GPU Engine Specs	
CUDA Cores	128
Multiprocessors	16
Warp size	32
Grid size	65535 x 65535 x 1
Block dim	512 x 512 x 64
Threads/block	512
Registers/block	8192
Graphic clock	738MHz
Processor clock	1836 MHz
Memory Specs	
Total memory	1024 MB
Shared memory / block	16KB
Total constant	64KB
Memory bandwidth	70.4 GB/sec

Tabla 5.1: Características GPU NVIDIA GTS 450

9 Algoritmos

Debido al overhead provocado por la transmisión de datos entre la CPU y la GPU, hemos considerado que la multiplicación es el módulo más apto para ser paralelizado, ya que supone ejecutar una gran cantidad de sumas que pueden realizarse simultáneamente. Por el contrario, ejecutar una suma o una resta en la GPU de forma individual causaría más retardo que ejecutarlas directamente en la CPU.

Creemos que el uso masivo de threads de ejecución ligera se adapta más a este subproblema y tiene mayor potencial de reducción de tiempo de cálculo. Por el contrario, en la CPU se ejecutará el módulo resta basándose en la implementación de Kogge-Stone.

Respecto a la representación de los datos, los cálculos serán realizados con vectores de enteros, representando los números utilizados en las operaciones en notación binaria. De esta forma dado un número X será representado con un vector $V=\{X_N, \ldots, X_1, X_0\}$ tal que X en complento a $2 = X_N*2^N + ... + X_1^1 + X_0^0$. Por esto y basándonos en los puntos fuertes de la tecnología, se asignarán a la ejecución del kernel correspondiente al cálculo del cuadrado tantos threads como elementos tenga V multiplicados por dos. Puede verse reflejado en la siguiente ilustración siendo *n* el número de elementos del vector y *m* el doble del número de elementos:

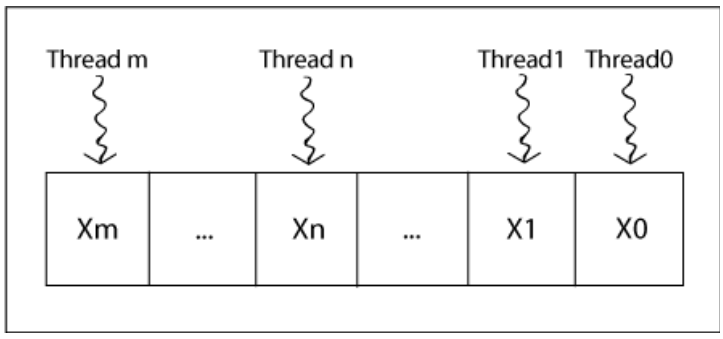

Fig 5.9: Representación de datos elegida para CUDA

La signatura del kernel ejecutado es la siguiente:

```
<<Square_Kernel( ARRAY_INT number, INT size_number, ARRAY_INT )>>
```

- ARRAY_INT **number** : Vector de enteros que representa el número sobre el que hay que calcular su cuadrado representado en binario.
- INT **size_number**: Entero que representa la longitud del parámetro de entrada.
- ARRAY_INT **result**: Vector de enteros que representa el resultado de la operación.
- INT **res_number**: Entero que representa la longitud del resultado.

Se han desarrollado distintas versiones del algoritmo, incluyendo mejoras incrementales, teniendo como objetivo mejorar los tiempos de ejecución, manteniendo el equilibrio con las limitaciones de la arquitectura.

1 Versión 1.

En esta primera versión se calcula el cuadrado del número elegido **number** expresado en notación binaria y representado con un array de enteros. Como particularidad, se construye una matriz NxM (siendo N el valor correspondiente a **size_number** y M size_number*2) intermedia que representa el cuadrado del número utilizando memoria compartida (__shared__) para agilizar el proceso.

La construcción de dicha matriz se basa en el principio de la multiplicación binaria. La primera fila se corresponde el valor con **number** y en el resto de filas de la matriz se introduce el vector **number** desplazado a la izquierda incrementalmente, tal como indica la siguiente ilustración:

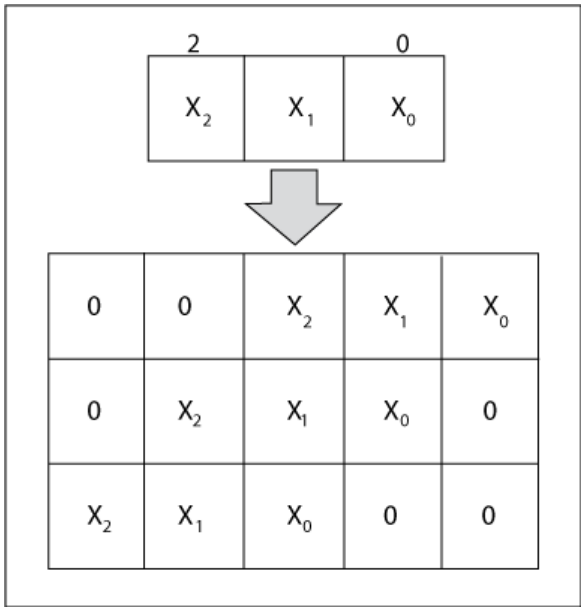

Fig 5.10: Manejo de datos en la versión 1 del algoritmo

Una vez construida se procede a sumar incrementalmente todas las filas de la matriz, utilizando un vector auxiliar para almacenar los desbordamientos, emulando una implementación Carry-Save [13], y almacenando el resultado en el vector de salida. El flujo de ejecución es el siguiente:

```
i = Identificador_thread;
for Iteracion = 1 to size_number-1 do
A[i] = C[i]
B[i] = Matriz[Iteracion]
    Resultado = A[i] + B[i];
    if Resultado > 1
          Carry_save[i] = 1
    if not
          Carry_save[i] = 0
    C[i] = Resultado modulo 2;
Sumar(C, Carry_save)
sincronizar_threads()
end for
```

Cabe destacar que la operación módulo 2 no supone un gran overhead, y tan solo consiste en mirar el bit menos significativo, no hace falta calcular el resto de la división entera.

2 *Versión 2.*

El problema de la versión anterior viene marcado por la cantidad de memoria utilizada por la matriz intermedia. Para conseguir paliar los efectos de la arquitectura respecto al tamaño de la memoria compartida, se decide sustituir la instanciación de la matriz. Para esto ha sido necesario modificar el algoritmo, generando los datos necesarios dinámicamente. Teniendo en cuenta que la operación a realizar es el cuadrado, podemos construir en tiempo de ejecución la fila de la matriz (tal como indica la figura siguiente) que necesitemos sabiendo los desplazamientos a la izquierda que debemos realizar:

```
i = Identificador_thread;
for Iteracion = 1 to size_number-1 do
A[i] = C[i]
B[i] = Desplaza_Izquierda(number, Iteracion)
    Resultado = A[i] + B[i];
    if Resultado > 1
          Carry_save[i] = 1
    if not
          Carry_save[i] = 0
    C[i] = Resultado modulo 2;
Sumar(C, Carry_save)
sincronizar_threads
end for
```

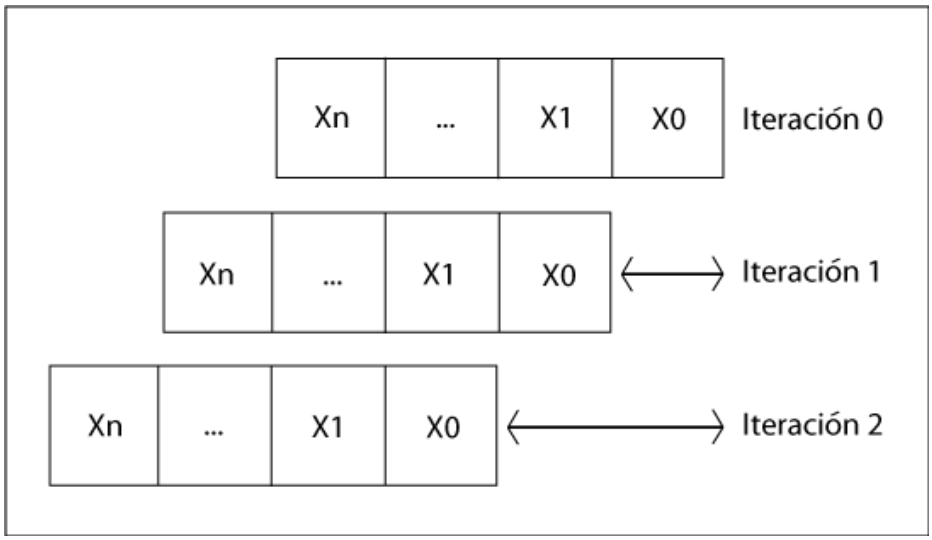

Fig 5.11: Creación dinámica de una matriz

3 *Versión 3.*

La siguiente mejora, está relacionada con el rendimiento. Si examinamos en detalle la suma carry save de las anteriores versiones veremos lo siguiente:

```
for Iteracion = 1 to size_number-1 do
    ...
while carry_save[0..n] != 0 do
    Resultado = C[i] + Carry_save[i-1]
    if Resultado >1
          Carry_save[i]=1
    else
          Carry_save[i]=0
    C[i]= Resultado mod 2;
    sincronizar_threads()
end while
...
end for
```

Como podemos observar en el flujo de ejecución anterior, se realizan muchas sincronizaciones entre threads, lo que provoca una gran pérdida de tiempo. Por tanto la finalidad de esta versión es reducir los tiempos de sincronización. Esto se consigue disminuyendo el número total de iteraciones.

Partiendo de la versión 2, en lugar de realizar N-1 iteraciones sumando incrementalmente las filas y controlando los desbordamientos con carry save, se va a realizar una suma transversal de todas las filas. En otras palabras, efectuaremos una suma por columnas sobre la que aplicaremos un esquema carry save para los números mayores que 1. Tal como indica la siguiente figura:

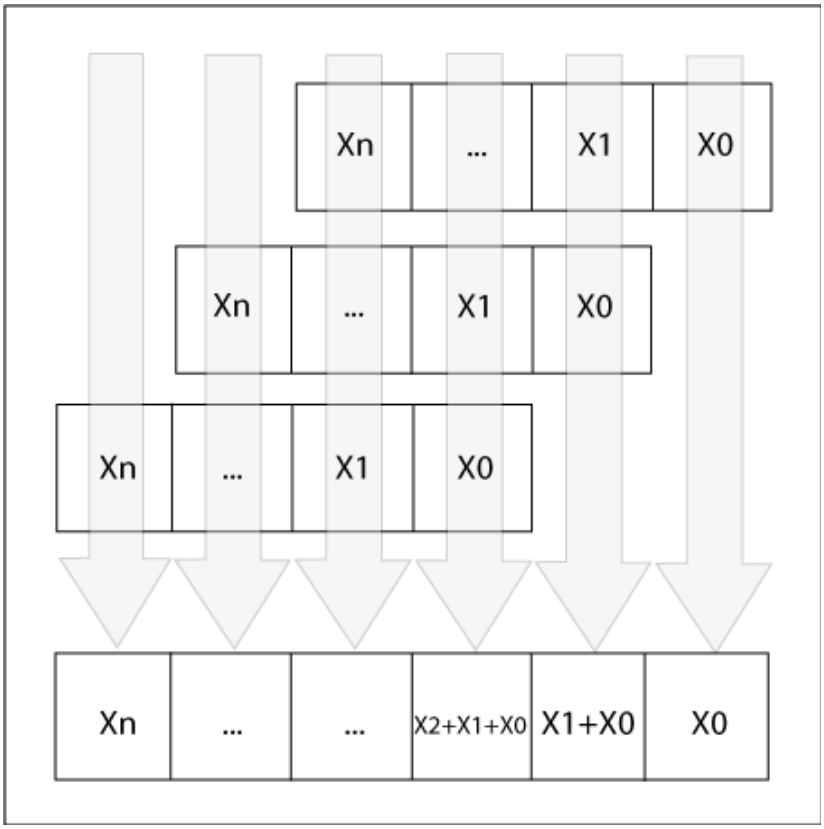

Fig 5.12: Suma transversal

Ejemplo: Siendo N el numero 7, V=[1,1,1] -> V[1,1,1] +V[1,1,1,0] + V[1,1,1,0,0] = V[1,2,3,2,1]

```
for Iteracion = 1 to size_number-1 do
A[i] = A[i] + Elemento(Desplaza_Izquierda(number, Iteracion), i)
end for
   Resultado = A[i];
   if Resultado > 1
         Carry_save[i] = desplazar_derecha(Resultado);
   if not
         Carry_save[i] = 0

   C[i] = Resultado modulo 2;
Sumar(C, Carry_save)
sincronizar_threads()
end for
```

Este algoritmo evita la sincronización de threads en el número de iteraciones del algoritmo respecto a la versión anterior, reduciendo los tiempos de ejecución eficientemente.

6 Experimentos

En este capítulo se mostrarán los resultados obtenidos tras la ejecución del emulador en sus distintas versiones, así como una comparativa con el test de Lucas-Lehmer implementado en C de forma convencional.

Comenzaremos hablando una vez más del problema de la precisión al tratar con los números primos de Mersenne, el crecimiento que se produce en cuanto al tamaño de estos números es exponencial, de modo que trabajar con ellos es complejo.

Realizaremos un análisis de los siguientes casos:

- Comparativa lenguaje C y emulador básico.
- Comparativa emulador básico y emulador mejorado.
- Comparativa emulador mejorado y aceleración CUDA.

1 Primer experimento: lenguaje C y emulador básico.

En nuestro primer experimento efectuaremos una comparativa a nivel de precisión y rendimiento del lenguaje C con nuestro emulador.

Partimos de la implementación en C del test de Lucas-Lehmer [14] y efectuamos una modificación poder observar en qué momento se producía desbordamiento de la variable en C.

Tras la ejecución del test en C fueron obtenidos los tiempos de ejecución medidos en milisegundos:

P	Tiempo Test Mp C (ms)
2	0
3	0
5	0
7	0
13	0
17	0
19	0
31	1
61	19958

Tabla 6.1: Resultados de ejecución en milisegundos del test en lenguaje C

Como podemos observar, el tiempo de ejecución se dispara en el momento en el que se llega al noveno primo de Mersenne. En las iteraciones siguientes, la variable desborda ya que el siguiente primo de Mersenne tiene p=89 bits, mientras que el tipo entero en la versión long no admite una precisión mayor de 64 bits.

En cuanto al resultado obtenido por el emulador en su versión básica, se obtuvieron los siguientes tiempos de ejecución, también en milisegundos:

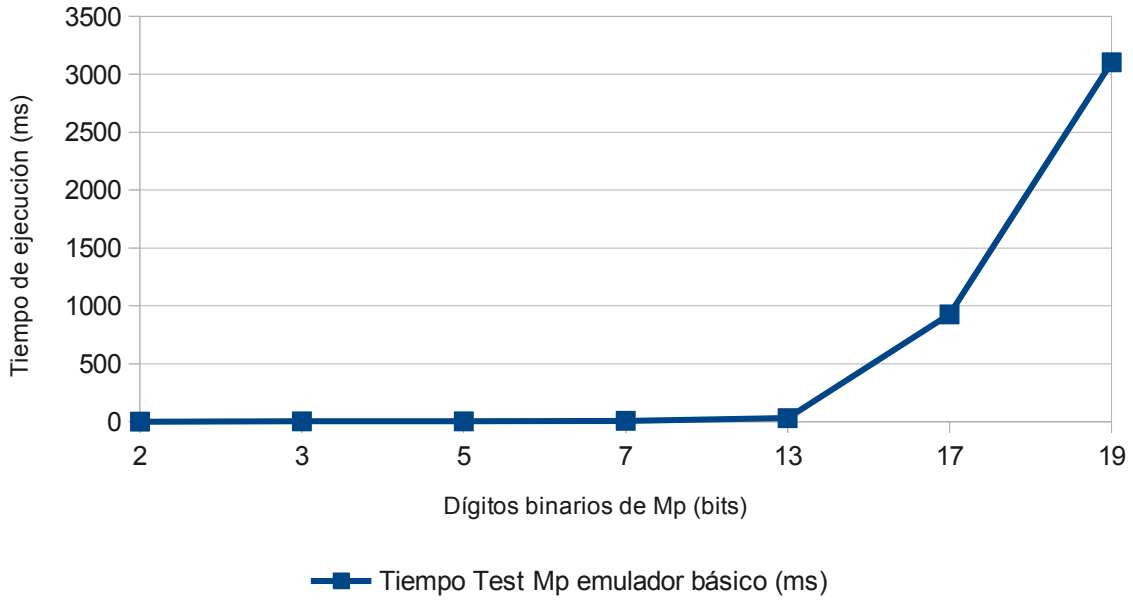

Fig 6.1: Gráfica precisión en cifras binarias - tiempo de ejecución en milisegundos del emulador básico

P	Tiempo Test Mp emulador básico (ms)
2	0
3	3
5	3
7	7
13	31
17	927
19	3103

Tabla 6.2: Tiempo de ejecución en milisegundos del emulador básico

Ante estos resultados, pudimos observar que aunque el ancho del número a estudiar no constituía un problema para el emulador, el tiempo de ejecución requerido por el emulador era muy superior al obtenido por el lenjuaje C incluso antes de llegar al décimo número primo de Mersenne.

2 Segundo experimento: emulador básico y emulador mejorado.

En este segundo experimento probamos las mejoras propuestas al final del capítulo 3 para mejorar el emulador:

- Sustitución del sumador RCA por un sumador Kogge-Stone.
- Sustitución del módulo de Barrett por la operación en aritmética módulo $2^p - 1$.

Al observar una diferencia tan grande entre los resultados obtenidos y los esperados se comenzó a buscar el motivo que podría estar causando el retraso en las operaciones del emulador. Dicho retraso resultó estar motivado por el cálculo de la constante *c* necesaria para la operación del módulo de Barrett, dado que para su cálculo era necesario emplear la división.

Dado que la división está implementada con el algoritmo de restas sucesivas, producía una gran penalización. Además esta división es preciso efectuarla para cada operación módulo.

Tras la ejecución del emulador mejorado, se obtuvieron los siguientes tiempos de ejecución:

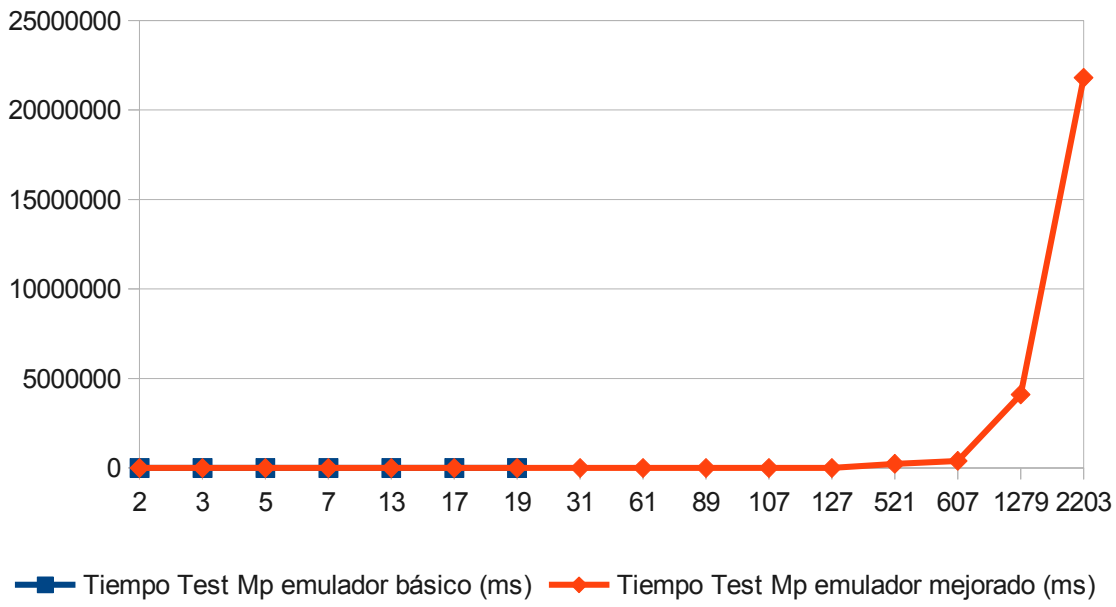

Fig 6.2: Gráfica comparativa entre el tiempo de ejecución en milisegundos del emulador básico y el emulador mejorado

Precisión	**Tiempo Test Mp emulador básico (ms)**	**Tiempo Test Mp emulador mejorado (ms)**
2	0	1
3	3	1
5	3	3
7	7	7
13	31	22
17	927	25
19	3103	34
31		64
61		357
89		1092
107		1874
127		3005
521		234764
607		396427
1279		4107403
2203		21794612

Tabla 6.3: Tabla de resultados de tiempo de ejecución del emulador básico y el emulador mejorado

En la tabla 6.3 podemos observar cómo el cálculo del primo de Mersenne M_{61} con el emulador mejorado requiere 357 ms, mientras que el cálculo del mismo primo con la versión implementada en C requiere 19958 ms. Por tanto, nuestro emulador, además de lograr una mejora considerable en la precisión, consigue alcanzar un tiempo de ejecución muy inferior al obtenido por el test implementado directamente en C.

Nótese que el primo más grande que hemos conseguido verificar es para p=2203, lo cual tarda 21794612 ms (aprox. 6h).

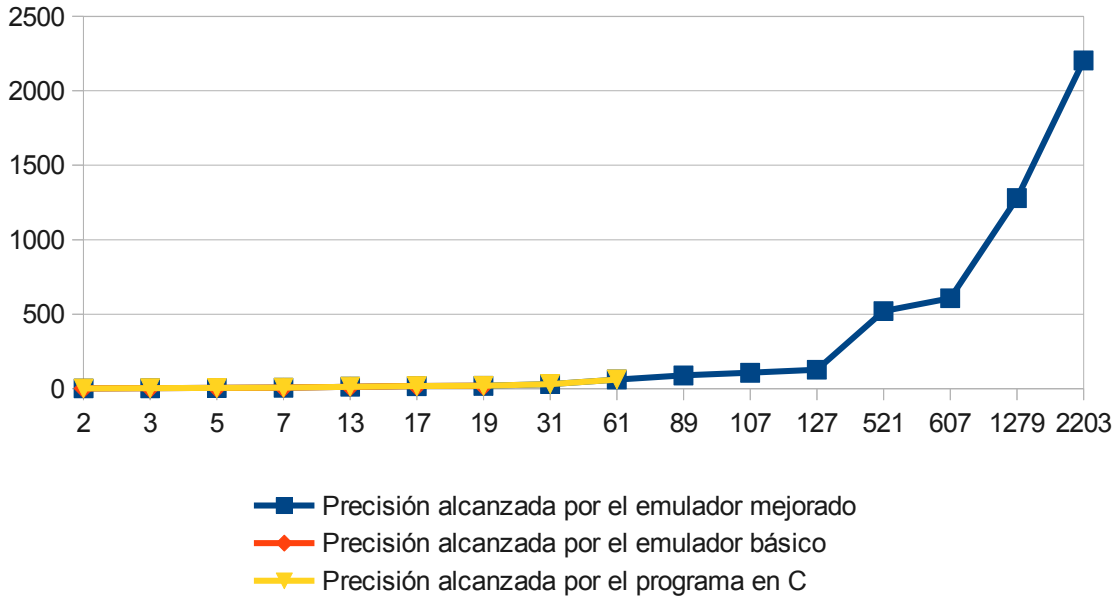

Fig 6.3: Gráfica comparativa entre la precisión alcanzada por el emulador mejorado, el emulador básico y el programa C

Exponentes	**Precisión alcanzada por el emulador mejorado**	**Precisión alcanzada por el emulador básico**	**Precisión alcanzada por el programa en C**
2	2	2	2
3	3	3	3
5	5	5	5
7	7	7	7
13	13	13	13
17	17	17	17
19	19	19	19
31	31		31
61	61		61
89	89		
107	107		
127	127		
521	521		
607	607		
1279	1279		
2203	2203		

3 *Tercer experimento: emulador mejorado y aceleración CUDA*

En nuestro tercer experimento efectuaremos una comparativa a nivel de precisión y rendimiento entre el emulador mejorado y las distintas versiones que utilizan el algoritmo de la multiplicación desarrollado en CUDA. Los resultados se muestran en las siguientes figuras y tablas:

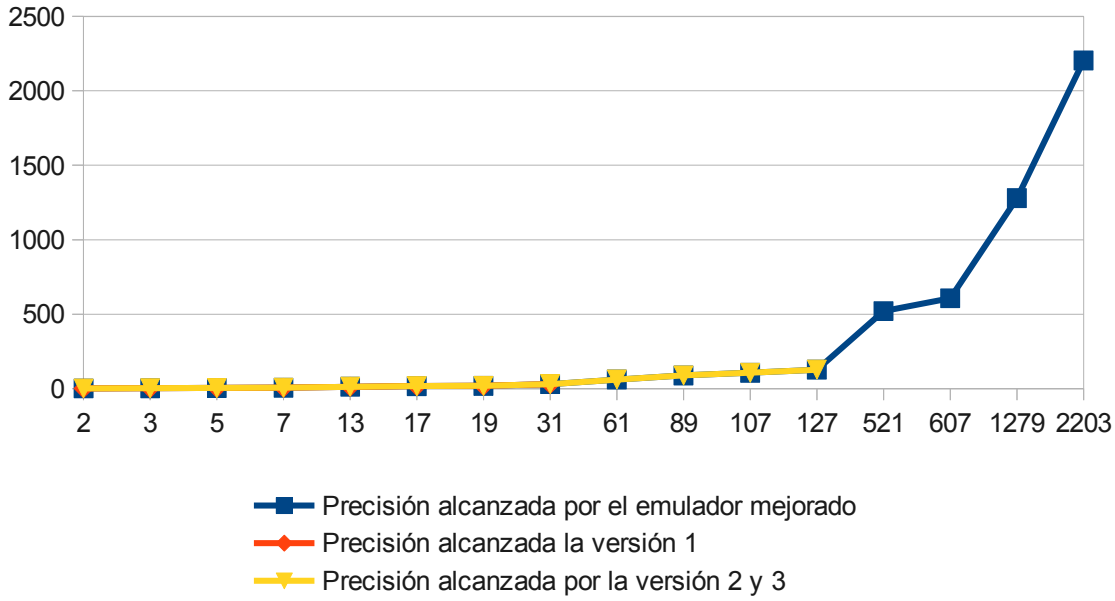

Fig 6.4: Gráfica de precisiones alcanzadas por versiones CUDA y emulador mejorado

Exponentes	Precisión alcanzada por el emulador mejorado	Precisión alcanzada la versión 1	Precisión alcanzada por la versión 2 y 3
2	2	2	2
3	3	3	3
5	5	5	5
7	7	7	7
13	13	13	13
17	17	17	17
19	19	19	19
31	31	31	31
61	61		61
89	89		89
107	107		107
127	127		127
521	521		
607	607		
1279	1279		
2203	2203		

Tabla 6.4: Tabla de precisiones entre el emulador mejorado, versión 1 de algoritmo CUDA y versiones 2 y 3 de altoritmo CUDA

Como comprobamos en la tabla 6.4, la primera versión de CUDA tiene muy poca precisión. Esta carencia fue suplida con el desarrollo de la segunda versión, consiguiendo pasar la barrera del número 32. De esta forma fue posible aumentar la precisión hasta p=127 bits.

En este caso, la precisión viene limitada por el número máximo de threads por bloque: al tener el dispositivo 512 threads/bloque la precisión llega hasta 2^{256}.

Llegado a este punto, el siguiente paso para ampliar la precisión es utilizar aritmética RNS [13], mapearlas sobre un kernel con varios bloques de threads, e incluso utilizar MPI para distribuir la carga proporcionalmente en GPUs distribuidas.

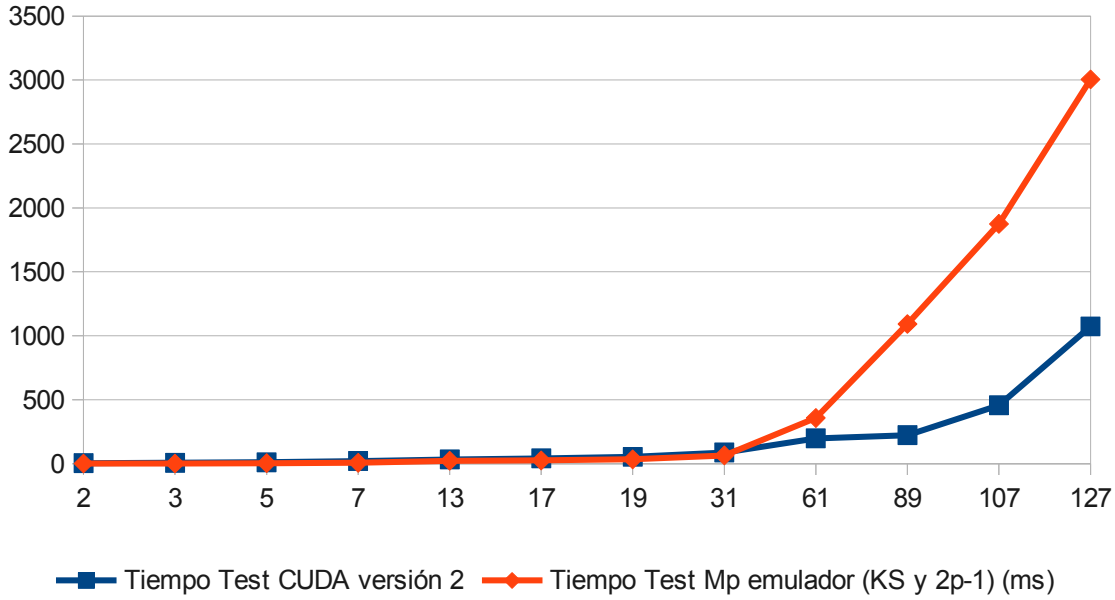

Fig 6.5: Gráfica comparativa tiempo CUDA Algoritmo versión 2 y emulador mejorado

Precisión	**Tiempo Test CUDA versión 2**	**Tiempo Test Mp emulador (KS y 2^p-1) (ms)**
2	4	1
3	7	1
5	11	3
7	20	7
13	32	22
17	41	25
19	53	34
31	87	64
61	198	357
89	222	1092
107	456	1874
127	1073	3005

Tabla 6.5: Tabla comparativa de tiempo de ejecución en CUDA algoritmo versión 2 y tiempo de ejecución del emulador mejorado

Con la segunda versión, además de la mejora de la precisión, se pueden observar resultados positivos en los tiempos de resolución, como se observa en la siguiente tabla.

Se observa que en los primeros números hasta el M_{31} la versión del emulador mejorado es más rápida que la segunda versión de CUDA. Esto se debe a las latencias producidas por los mapeos de memoria desde la CPU a la GPU. A partir de M_{31} se observan una disminución destacable en los tiempos de resolución la versión de CUDA, consiguiendo en el mejor de los casos un speedup de 5.

Con la tercera versión se han obtenido los siguientes resultados de tiempo de ejecución, medidos en milisegundos.

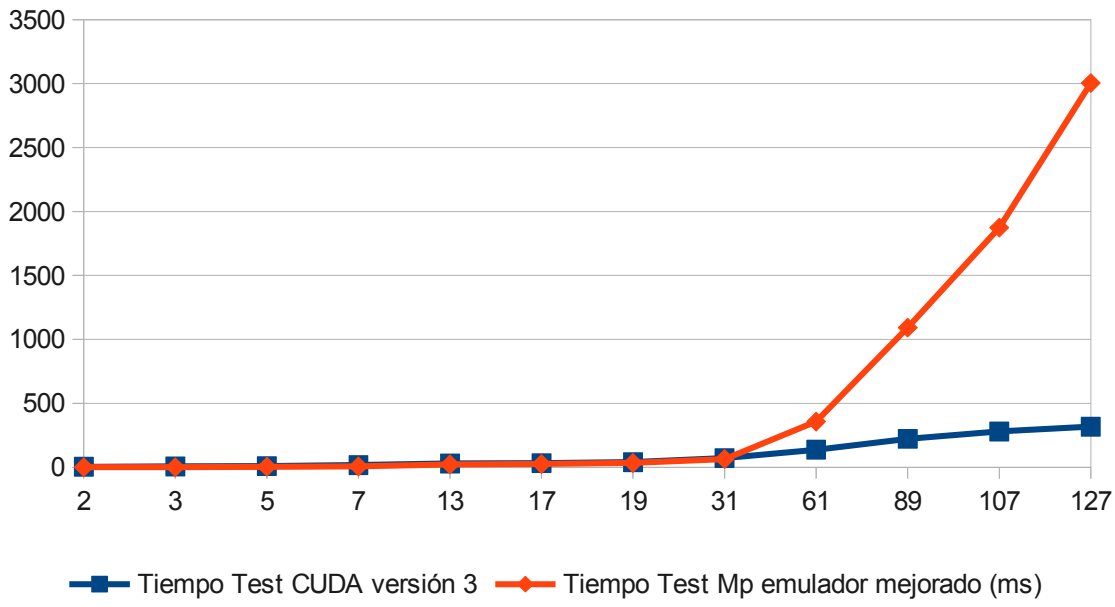

Fig 6.6: Tiempo de test CUDA v3 y tiempo de test emulador mejorado

Precisión	Tiempo Test CUDA versión 3	Tiempo Test Mp emulador mejorado (ms)
2	4	1
3	6	1
5	9	3
7	17	7
13	29	22
17	32	25
19	40	34
31	72	64
61	137	357
89	222	1092
107	280	1874
127	317	3005

Tabla 6.6: Comparativa entre tiempos ejecución CUDA algoritmo versión 3 y emulador mejorado.

Podemos observar que el algoritmo de la multiplicación transversal implementado para este caso, ha obtenido una reducción de tiempos bastante destacable.

Este hecho puede comprobarse con los tiempos de resolución de los números presentes en la tabla, por ejemplo, M_{127}, en el que la versión CUDA obtiene un speedup de 10. Además, es preciso mencionar que los tiempos de ejecución de números más altos, pero menores que 256 y que no resultaron ser primos de Mersenne, el speedup se disparó a más de 20.

7 Conclusiones y trabajo futuro.

En el primer capítulo de esta memoria se definió como objetivo principal de este proyecto la resolución del problema de la precisión en la búsqueda de números primos de Mersenne.

De este objetivo principal derivamos un objetivo secundario que consistía en la extensión de la capacidad de cómputo de un equipo convencional más alla de las limitaciones impuestas por la arquitectura del procesador. Es decir, mejorar la precisión manteniendo un buen rendimiento.

Para la consecución de estos objetivos optamos por las siguientes resoluciones, descritas con anterioridad:

- Implementación de un emulador software de una arquitectura hardware que implementa el test de Lucas-Lehmer.
- Aceleración de las operaciones de dicho emulador mediante el uso de GPUs y CUDA.

El objetivo principal de este proyecto se ha cumplido satisfactoriamente, como puede observarse en el capítulo de experimentos, sin embargo, a parte del trabajo concluido hubiera sido interesante realizar una serie de mejoras que, por falta de tiempo, no han podido introducirse y que quedan como trabajo futuro.

A continuación se expondrán las conclusiones del trabajo realizado y las mejoras futuras.

1 Conclusiones

1 Mejora de la precisión

Ha quedado demostrado que la utilización del emulador software, combinado con el sistema de representación con vectores de tamaño variable utilizado permiten que un sistema

computacional convencional, represente y evalúe números de una envergadura mucho mayor de los que se podrían analizar en circunstancias normales con los tipos de datos convencionales.

2 Mejora del rendimiento

Aunque la resolución del problema de la precisión se consigue desde la versión básica del emulador, esta versión se movía en unos límites de rendimiento poco razonables, de ahí las mejoras de diseño realizadas. Tanto la sustitución del sumador RCA por un sumador Kogge-Stone, acelerando el cálculo de los acarreos en las operaciones de suma, como la sustitución del módulo de Barrett por la operación en aritmética módulo $2^p - 1$, reducen considerablemente el número de operaciones a realizar por el emulador.

3 Aceleración con CUDA

La inclusión de aceleración hardware al emulador ha sido tal como esperábamos, crucial. El encaje de la arquitectura con las características del problema a resolver es tal, que se ha llegado a observar un speedup de 20 respecto a la versión mejorada del prototipo ejecutada en la CPU. Esto maneja márgenes realmente interesantes para mejoras futuras, las cuales se citan en el siguiente capítulo.

2 Trabajo futuro

En esta sección se mostrarán las líneas de investigación que componen el trabajo futuro.

1 Aritmética RNS

Un sistema RNS se define con un conjunto de N constantes enteras

$$C = \{m_1, m_2, m_3, \ldots, m_N\}$$

Siendo M el mínimo común múltiplo de todos los componentes del conjunto, cualquier número entero X menor que M puede ser respresentado en RNS como un conjunto de N enteros más pequeños.

$$\{x_1, x_2, x_3, \ldots, x_N\}$$

Definiendo x_i como X módulo m_i.

Para que este sistema de representación sea eficaz es necesario que todos los miembros del conjunto C sean coprimos, es decir, que su máximo común divisor sea 1, para evitar que a dos enteros distintos les corresponda la misma representación RNS.

Como puede observarse, la aplicación de la aritmética RNS conlleva una paralelización de operaciones. Dado que la aceleración del algoritmo consiste en paralelizar el mayor número de operaciones, la aplicación de la aritmética RNS parece una una extensión natural al trabajo realizado.

2 MPI

En nuestro emulador la representación de los números se basa en vectores y, en contadas ocasiones, en el uso de matrices. Gran parte de nuestras operaciones tienen un paralelismo implícito muy grande, para hacer más eficiente el procesamiento de la información contenida en estos vectores y las operaciones que tienen que ver con ellos.

MPI es un estándar para una biblioteca de paso de mensajes, cuyo objetivo es comunicar procesos, tanto en el interior de un nodo, como en varios nodos.

El objetivo de combinar MPI con este emulador sería dividir los datos necesarios entre varios nodos y así aprovechar el paralelismo de las operaciones realizadas para hacer más eficiente el sistema en general.

3 *CUDA*

Uno de los cambios futuros y principales para conseguir mejoras de rendimiento en CUDA sería la migración del dispositivo sobre el cual se ejecuten los experimentos. Como citamos en el apartado 5.8 el dispositivo utilizado esta basado en una tecnología precursora de la arquitectura Tesla, se trata de un dispositivo fuera de rango con vistas a obtener rendimientos óptimos.

En la actualidad, existen arquitecturas alternativas con mayores prestaciones. Es el caso de FERMI y KEPLER, no sólo por sus prestaciones de computación más elevadas, sino tambien por las mejoras en las técnicas arquitectónicas utilizadas en su diseño. A continuación citaremos algunas de las más relevantes:

- *Dynamic parallelism* Paralelismo dinámico. Esta característica añade a la GPU la capacidad de crear flujos de trabajo, sincronización de resultados y planificación dinámica sin necesidad de involucrar a la CPU. Tal como indica la ilustración:

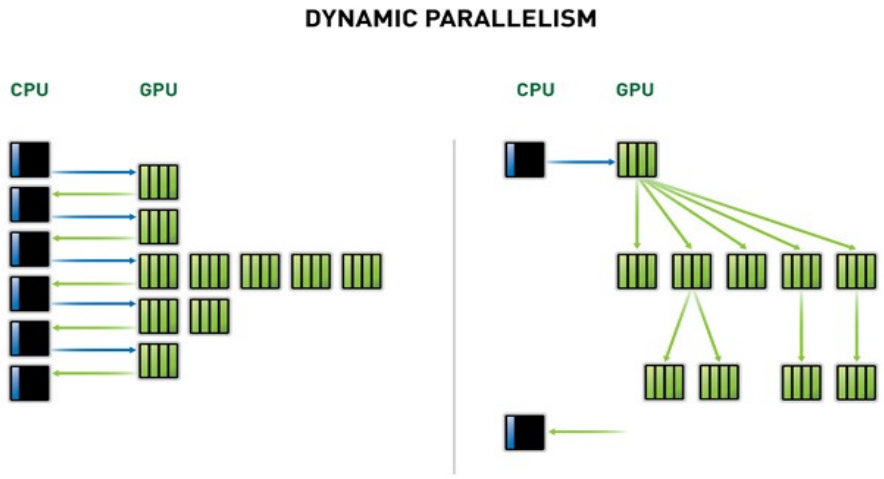

Fig 7.1: Dynamic Parallelism

- *Hyper-Q*. Esta característica reduce el tiempo de inactividad de la CPU al permitir que múltiples núcleos de ésta utilicen una misma GPU, lo que mejora drásticamente la programabilidad y la eficiencia. Característica que potencia el uso de MPI, como vemos en la siguiente gráfica:

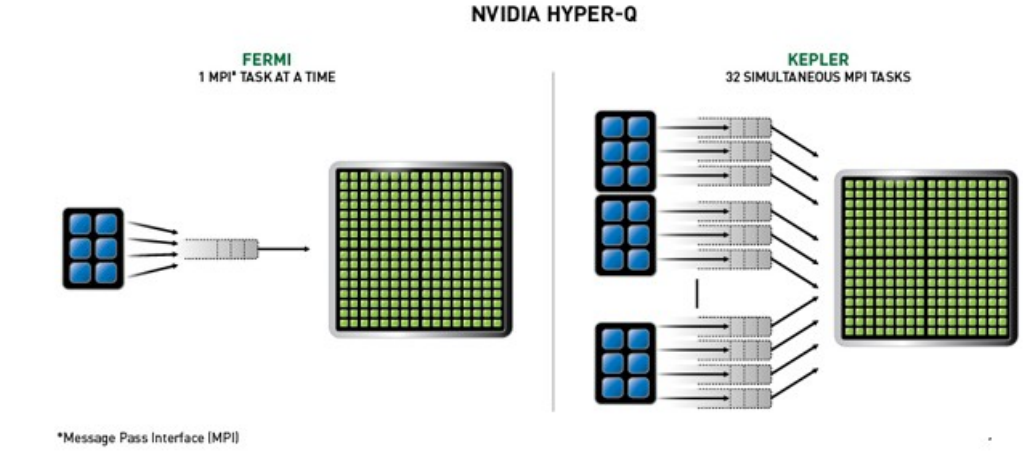

Fig 7.2: Nvidia Hyper-Q

- *SMX.* Esta característica proporciona mayor velocidad de procesamiento y eficiencia gracias a su innovador multiprocesador de streaming, que permite dedicar más espacio a los núcleos de procesamiento que a la lógica de control. Como se observa en la siguiente ilustración, esta técnica reduce el consumo de energía a la vez que mantiene la potencia:

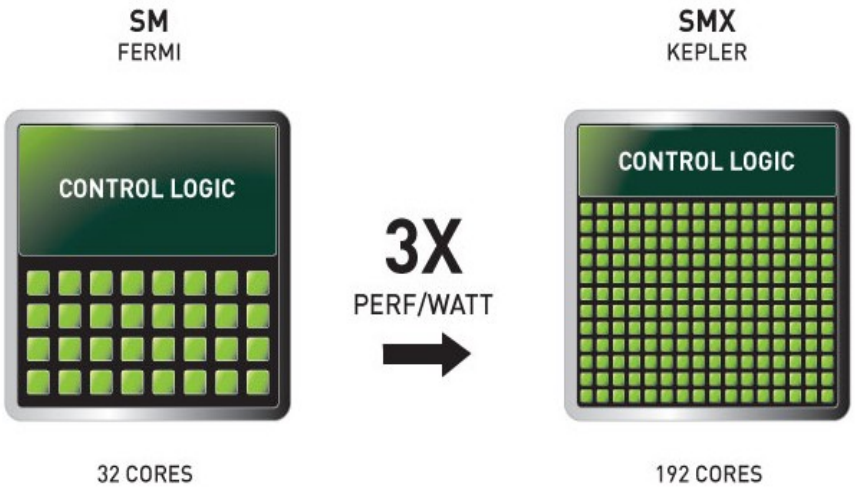

Fig 7.3: SMFermi - SMXKepler

	GeForce GTS 250	FERMI GF104	KEPLER GK110
Compute Capability	1.1	2.1	3.5
CUDA Cores	128	460	2880
Max Registers / Thread	16	63	255
Max Threads / Thread Block	512	1024	1024
Shared Memory Size Configurations (bytes)	16K	48K	48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1

Tabla 7.1: Comparativa de GPUs

Partiendo de la comparativa mostrada de la tabla anterior, unida a las gananacias procuradas por las técnicas arquitectónicas, creemos que migrando el algoritmo de multiplicación a una GPU más actual conseguiremos una gran reducción del tiempo de ejecución del emulador, así como una mayor precisión.

8 Conclussions and future work.

In the first chapter of this report, the main objective of our work was defined, namely: facing the precision problem for dealing with Mersenne primes.

A secondary purpose is derived from the aformentioned main objective. This secondary target consists of extending the computing capability of conventional CPUs. In other words, our target is to increase the precision while keeping a good, or even better, performance.

In order to comply with these two requirements, we chose implementing a software emulator of a hardware architecture that implements the Lucas-Lehmer test.

In addition to this, we have utilized GPUs and CUDA for accelerating some critical operations inside our emulator.

The main objective of this project has been successfully completed, as can be observed in experiments, . However, we consider that it would be interesting to extend our ideas with complementary improvements in the future.

The conclussions of the work and future improvements are shown.

1 Conclusions

1 Improved precision

It has been shown that the use of software emulator, combined with the representation system with variable size vectors, allows a computer to represent and evaluate numbers much larger than the ones which can be analyzed with conventional types.

2 *Improved performance*

Although the resolution of the precision problem is achieved with the basic emulator version, we noticed that our performance was below expected. Therefore, we tried some improvements as the replacement of the RCA by the Kogge-Stone adder, for accelerating the calculation of the addition carries, and as the utilization of $2^p - 1$ arithmetic for computing the modulo operation, instead of the Barrett algorithm. In this way, we have reduced the execution time considerably.

3 *CUDA Acceleration*

The inclusion of the hardware acceleration has been crucial, as expected. The utilization of CUDA has provided us a 20X speedup, as it can be observed when comparing the execution times between the improved emulator with and without CUDA. This fact is really encouraging for the future, as nowadays there are many more powerful GPUs than the one utilized in this project.

2 *Future work*

This section will discuss some future lines of work.

1 *RNS Arithmetic*

An RNS system is defined with a set of N integer constants

$$C = \{m_1, m_2, m_3, \ldots, m_N\}$$

If M is the least common multiple of all components of the set, any X integer less than M can be represented in RNS as a set of N smaller integers

$$\{x_1, x_2, x_3, \ldots, x_N\}$$

Defining x_i as X module m_i

There is only a requirement for utilizing this system: all the set members must be relatively prime, that is, their greatest common divisor must be 1, to prevent two different integers from possesing the same RNS representation.

As it is shown, the application of RNS arithmetic entails parallelizing operations. Since the acceleration consists of parallelizing as many operations as possible, the application of the RNS arithmetic seems to be a natural extension of this work.

2 *MPI*

In our emulator, the representation of numbers is based on vectors and, in rare times the use of matrix. Much of our operations have a large implicit parallel, so in order to accelerate the processing of information contained in these vectors and the operations that have to do with them we propose MPI.

MPI is a standard for message-passing library, which aims to communicate processes, either within a node or in multiple nodes.

The goal of combining MPI with this emulator would be to divide the data needed between multiple nodes and thus take advantage of the parallelism of the performed operations. Hence, we think our system would be more efficient.

3 *CUDA*

One of the major challenges for future performance improvements is to migrate to a more powerful GPU. As quoted in section 5.8, the considered device is based on a previous technology to Tesla architecture, which is out of range if we are looking for the optimal performance.

Nowadays, there are better alternative architectures. This is the case of the FERMI and KEPLER series, not only because of its higher computing performance, but for the improvements performed in their architecture. We will quote some of the most relevant:

- *Dynamic parallelism* this adds the ability to create workflows, synchronization and dynamic scheduling results without involving the CPUsuch as illustrated in figure 8.1:

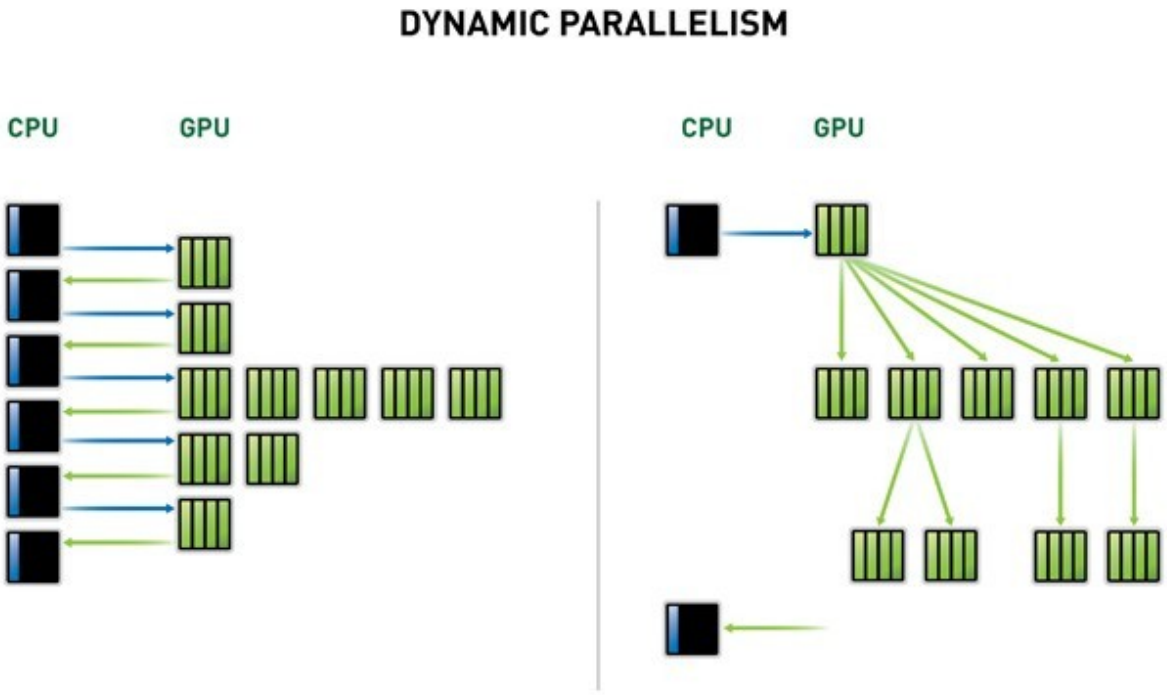

Fig 8.1: Dynamic Parallelism

- *Hyper-q*: This feature reduces the CPU downtime to allow multiple cores using a single GPU, which dramatically improves the efficiency and programmability. This feature also enhances the use of MPI, as shown in the following graph:

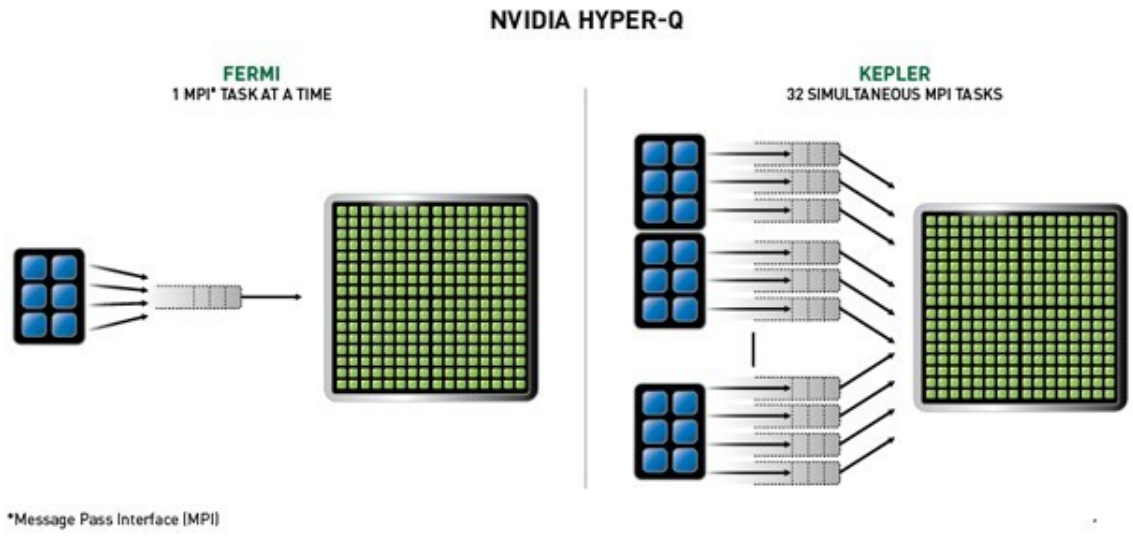

Fig 8.2: Nvidia Hyper-Q

- *SMX*: This feature provides increased processing speed and efficiency with its innovative streaming multiprocessor, which allows more space to processing cores than the control logic. As shown in the following illustration, this technique reduces the energy consumption while maintaining the power.

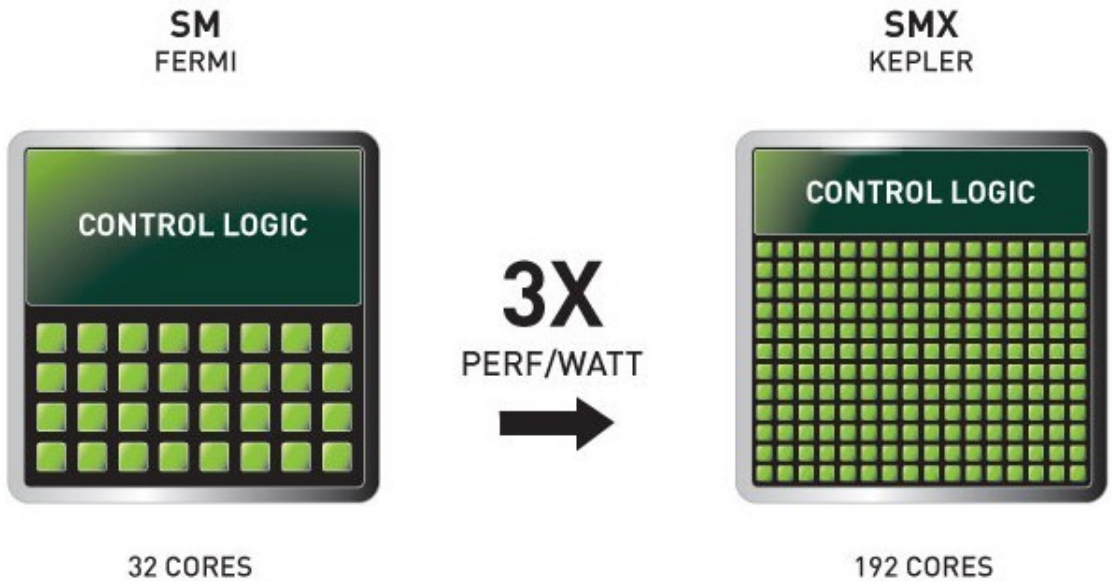

Fig 8.3: SMFermi - SMXKepler

	GeForce GTS 250	FERMI GF104	KEPLER GK110
Compute Capability	1.1	2.1	3.5
CUDA Cores	128	460	2880
Max Registers / Thread	16	63	255
Max Threads / Thread Block	512	1024	1024
Shared Memory Size Configurations (bytes)	16K	48K	48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1

Tabla 8.1: GPUs comparative

Based on the comparison shown in the table above, coupled with performance improvements attempted by architectural techniques, make a firm way to keep reducing the time of problems resolution.

9 Bibliografía.

[1] «Número primo - Wikipedia, la enciclopedia libre». [En línea]. Disponible en: http://es.wikipedia.org/wiki/N%C3%BAmero_primo#Historia_de_los_n.C3.BAmeros_primos. [Accedido: 20-jun-2013].

[2] M. Hamilton, W. P. Marnane, y A. Tisserand, «A Comparison on FPGA of Modular Multipliers Suitable for Elliptic Curve Cryptography over GF(p) for Specific p Values», en *2011 International Conference on Field Programmable Logic and Applications (FPL)*, Sept., pp. 273-276.

[3] S. Baktir, S. Kumar, C. Paar, y B. Sunar, «A state-of-the-art elliptic curve cryptographic processor operating in the frequency domain», *Mob. Networks Appl.*, vol. 12, n.º 4, pp. 259-270, ago. 2007.

[4] D. A. Bader, A. Chandramowlishwaran, y V. Agarwal, «On the Design of Fast Pseudo-Random Number Generators for the Cell Broadband Engine and an Application to Risk Analysis», en *37th International Conference on Parallel Processing, 2008. ICPP '08*, Sept., pp. 520-527.

[5] M. Matsumoto y T. Nishimura, «Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator», *Acm Trans Model Comput Simul*, vol. 8, n.º 1, pp. 3–30, ene. 1998.

[6] «Édouard Lucas», *Wikipedia, the free encyclopedia*. 12-jun-2013.

[7] «Lucas-Lehmer test - Mersennewiki». [En línea]. Disponible en: http://mersennewiki.org/index.php/Lucas-Lehmer_Test. [Accedido: 12-jun-2013].

[8] «GIMPS Home». [En línea]. Disponible en: http://www.mersenne.org/. [Accedido: 16-jun-2013].

[9] «Prime95 - Mersennewiki». [En línea]. Disponible en: http://mersennewiki.org/index.php/Prime95. [Accedido: 16-jun-2013].

[10] Andrew Thall, «Implementing a Fast Lucas-Lehmer Test on Programmable Graphics Hardware», ago. 2007.

[11] «Adder (electronics)», *Wikipedia, the free encyclopedia*. 01-jun-2013.

[12] Deschamps, Jean-Pierre, *Hardware implementation of finite-field arithmetic / Jean-Pierre Deschamps, José Luis Imaña, Gustavo D. Sutter*, New York : McGraw-Hill, 2009. .

[13] «Computer Arithmetic Algorithms - Israel Koren - Google Libros». [En línea]. Disponible en: http://books.google.es/books/about/Computer_Arithmetic_Algorithms.html?id=Dq-L5sHg6vkC&redir_esc=y. [Accedido: 20-jun-2013].

[14] «Lucas-Lehmer test - Rosetta Code». [En línea]. Disponible en: http://rosettacode.org/wiki/Lucas-Lehmer_test#C. [Accedido: 16-jun-2013].

10 Participación en el proyecto

1 Participación en el proyecto de Pablo Cerro Cañizares

Principalmente, y antes de comenzar a hablar de aportes individuales, he de decir que nuestro grupo consta únicamente de dos personas, y tanto Cristina como yo hemos trabajado conjuntamente durante todo el proceso sin ningún tipo de distinción.

Durante el comienzo del proyecto, nos extendimos durante algunas semanas buscando información relacionada. Tanto la definición propia del proyecto como todos los trabajos relacionados.

Se empezó por hacer las primeras pruebas con prime95, tomar los primeros apuntes y sacar las primeras conclusiones.

Seguidamente, construimos el primer prototipo basado en C con Visual Studio en Windows. Sin embargo, finalmente nos decantamos por Linux como sistema operativo. El proyecto trata un tema bastante actual y una de las primeras cosas que intentamos determinar fue el alcance del mismo, ya que es posible encontrar herramientas que trabajan con los primos de Mersenne, e incluso financiación para quien descubra un nuevo primo de Mersenne. No obstante, la idea de elaborar un nuevo enfoque nos pareció estimulante.

A estas alturas, comenzamos a tener claros ciertos aspectos. Las primalidad de un número tan grande, podría utilizarse en campos tan vitales como la criptografía dentro de las comunicaciones. El cifrado de dichas comunicaciones se ha convertido en algo vital en estos días, aportando una capa de blindaje a la emergente salida de nuevos servicios telemáticos como por ejemplo la declaración de la renta o los muchos usos del DNIe.

El primer paso en la elaboración del emulador, fue la construcción en C de la librería básica. Como en casi toda obra, la primera piedra no es la más emocionante, tampoco lo fue en este caso. Particularmente, estoy más acostumbrado a C++ que a C, que es fuertemente tipado y

el compilador permite hacer algunas operaciones que pueden dar problemas. Así que el principio fue un poco lento.

Una vez construida por completo toda la librería y pudiendo hacer las primeras pruebas, empezamos a animarnos. La posibilidad de reducir los tiempos de ejecución empezó a darnos un grado de motivación que hizo que a partir de ese momento el proyecto se nos hiciera corto y bastante bonito.

Construimos las primeras versiones mejoradas de algunas operaciones, consultamos bibliografía de referentes en la materia como José Luis Imaña (reducción de Barrett) o Román Hermida (él nos inspiró la utilización de la aceleración CUDA). Nuestro director de proyecto Alberto Del Barrio nos puso las cosas muy sencillas desde el principio permitiendo escoger la metodología a utilizar para afrontar el problema, de ahí que eligiésemos la opción de CUDA.

¿Por qué CUDA y no VHDL? Muy sencillo, personalmente yo no tengo una FPGA en mi casa y sí una GPU NVIDIA. CUDA es un fenómeno que se está expandiendo con una ferocidad alarmante, pasando desde las GPGPU (General Propose GPU) siguiendo por su ejecución en los procesadores TEGRA 4 y terminando en su utilización en los clusters de cómputo masivo (¡algunos de ellos en el top10 del top500!). Siguiendo algunas reseñas y viendo que en el futuro todos los sistemas many core estarán fundamentados en GPUs, elegimos CUDA sin ningún tipo de duda.

Una vez elegida la tecnología, nos dimos cuenta de su importancia. Tener al alcance de la mano la posibilidad de manejar un "mini-cluster" es un punto de motivación bastante grande.

Sin embargo, manejar CUDA fue bastante difícil al principio. fue bastante costoso. Decidimos utilizar la gran cantidad de documentación de NVIDIA, incluyendo conferencias grabadas e impartidas por componentes de la corporación. A base de resúmenes y ejemplos de prueba sacamos el primer prototipo, todo un éxito. La alegría nos duró poco porque desbordaba la memoria compartida, que es muy pequeña.

Boli y papel en mano, comenzamos a diseñar nuevos prototipos que resolvieran las carencias del primero. Con ayuda de Alberto finalmente todo ha salido muy bien. Con grandes resultados y muy motivante.

En el futuro vemos clave la compatibilidad con MPI (inquietud que nos sembró Alberto Núñez, del departamento de SIC de la facultad) además de todas las apuntadas en la sección de futuros trabajos.

Finalmente he de puntualizar que ha sido CLAVE el trato tanto entre componentes de grupo como con nuestro director, con los que he compartido esta experiencia y estoy muy contento con su rendimiento.

2 *Participación en el proyecto de Cristina Esteban Luis*

Antes de comenzar a describir mi participación en este proyecto, quiero hacer constar el hecho de que es extremadamente complicado describir algo como "participación" en este trabajo, pues este proyecto ha sido ante todo un trabajo de equipo. Tanto mi compañero Pablo como yo hemos invertido una gran cantidad de esfuerzo y trabajo en el desarrollo de este proyecto, y dado que el proyecto es único, el equipo también, de modo que reitero, establecer una separación carecería de sentido.

Una vez tomada la decisión de acometer este proyecto se produjeron reuniones con nuestro director en las que decidimos encaminar el desarrollo de este proyecto hacia el emulador software por los motivos que han sido expuestos con anterioridad en esta memoria.

El proyecto comenzó en noviembre, con la búsqueda de distinta documentación que ha terminado siendo empleada en el desarrollo del mismo, comenzamos con la búsqueda de información acerca del proyecto GIMPS y artículos relacionados con la búsqueda de números primos de Mersenne.

Una vez adquiridos los conocimientos básicos acerca del tema que íbamos a tratar, comenzamos con el desarrollo del emulador. Contemplamos los requisitos necesarios para conseguir replicar la arquitectura hardware.

Debido a nuestra necesidad de resolver problemas de rendimiento y precisión, además de poseer conocimientos avanzados sobre la plataforma, decidimos implementar el emulador en lenguaje C.

También fue entonces cuando se tomó la decisión de comenzar por implementar la lógica básica de un computador para ir ascenciendo en la escala de complejidad de la estructura.

Se planteó la primera fase del proyecto como la creación de una biblioteca binaria para generar una base sobre la que construir el resto de operaciones que serían efectuadas por el emulador.

Se implementaron las operaciones básicas (AND, OR, XOR), y éstas fueron utilizadas para construir operaciones aritméticas más complejas.

Durante los meses siguientes se estuvieron efectuando pruebas para asegurar que cada nivel funcionaba correctamente, una vez se consiguió que todas las operaciones aritméticas funcionasen independientemente de las precisiones utilizadas se consideró terminada la biblioteca binaria y se comezó con la implementación del test de Lucas-Lehmer.

El test de Lucas-Lehmer nos llevó gran cantidad de tiempo, puesto que supuso una especial dificultad la implementación del módulo mediante el método de Barrett. Además de ciertas dificultades derivadadas de fallos que se produjeron al encadenar varias multiplicaciones seguidas, todo debido a defectos que no habían sido hallados en las pruebas unitarias realizadas con la biblioteca. Una vez se consiguió hacer funcionar esta parte del programa, pudo ser completado el test de Lucas-Lehmer.

En la etapa de optimizaciones procedimos a buscar una manera de mejorar el emulador conseguido.

Implementamos las mejoras de las que se ha discutido con anterioridad. Tanto en CUDA como en la utilización de la aritmética modular y el sumador Kogge-Stone.

Por último procedimos a elaborar la presente memoria, haciendo uso de los datos obtenidos mediante las ejecuciones del emulador.