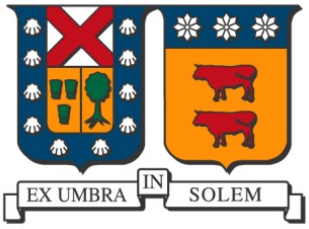
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE




# “VALORIZACIÓN DE DERIVADOS FINANCIEROS MEDIANTE PROCESAMIENTO EN PARALELO HÍBRIDO”

DIEGO VILLEGAS ARENAS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Xavier Bonnaire
Profesor Correferente: Horst von Brand

Agosto - 2019

## DEDICATORIA

A mis padres, quienes me han apoyado incondicionalmente todo este tiempo, mis amigos, con quienes compartí risas y penas, mis profesores, quienes pulieron el potencial que había en mi, y todos aquellos cuyos caminos se cruzaron con el mio a lo largo de esta travesía, a todos ustedes, gracias.

# AGRADECIMIENTOS

Quisiera agradecer a todos aquellos que me brindaron ayuda para la realización de este, mi proyecto de memoria. Estos grupos y/o personas son:

- Alvaro Díaz Valenzuela, por su apoyo constante en el área financiera de este proyecto.
- Computer Systems Research Group (CSRG), por su ayuda con conseguir un servidor donde ejecutar el proyecto.
- Laboratorio INCA de la UTFSM, particularmente Camilo Valenzuela por ayudarme con el contacto.
- Yuri Ivanov, por su apoyo en el trabajo realizado sobre el cluster HPC.
- Creasys S.A, por facilitarme una computadora en la cual realizar pruebas de formas directa.
- Luis Salinas, por ayudarme a obtener acceso al cluster HPC.
- Jacqueline Pinto Fuentes, por su ayuda con la edición de este proyecto

# RESUMEN

**Resumen—** La valorización de derivados financieros es un proceso llevado a cabo a diario por diferentes inversionistas, y cuyo resultado impacta directamente a las decisiones que toman a lo largo del día. Esto obliga a que el cálculo deba ser tanto rápido como preciso. Sin embargo, la velocidad del mismo no resulta fácil de lograr y muchas veces el proceso termina tardando más de lo deseado. A modo de resolver esto, se decidió paralelizar, tanto en CPU, GPU y de forma híbrida (esto es simultáneamente en CPU y en GPU) ciertos cálculos relacionados con la valorización de derivados. Una vez asegurada la precisión del cálculo, las diferentes versiones del sistema se probaron entre sí bajo las mismas condiciones, a modo de generar una comparación a nivel de tiempo de cada una de éstas, con el fin de determinar cuál de las versiones entrega los resultados en el menor tiempo. Por último, se realiza una extrapolación en base a las características del hardware, de cómo el mismo programa se desempeñaría en distintas máquinas. Con estos resultados se espera poder ayudar a diferentes inversionistas, no sólo para tener acceso a un sistema capaz de realizar sus procesos de valorización de la forma más ágil posible, sino que además darles una idea del nivel de inversión monetaria que deberían realizar para lograr resultados óptimos en su contexto.

**Palabras Clave—** Derivados Financieros; Value-at-Risk; Computación en Paralelo; CUDA; Paralelización híbrida.

# ABSTRACT

***Abstract*—** The pricing of Financial Derivatives is a process that is carried out daily by different investors and/or banks from all over the world, and it's results directly influences their financial decisions. This means that the process not only needs to be precise, but also fast. This last characteristic is quite difficult to obtain, and many times the process ends up taking more time than desired. In order to solve this issue, a parallel approach, at the level of CPU, GPU, and hybrid (this means, parallelization on CPU and GPU at the same time), has been implemented for a set of operations involved in the pricing of derivatives. The different versions were tested, under the same conditions, in order to compare their performance to determine which one delivers its results in the least amount of time. Finally, based on the characteristics of the hardware used, we tried to predict how the same system would perform on different machines. With this results we're trying to help investors not only to gain access to a more speed efficient version of this system, but also by giving them an idea of the level of hardware investment they would need to do in order to obtain optimum results in their context.

***Keywords*—** Financial Derivatives; Value-at-Risk; Parallel computing; CUDA; Hybrid Parallelization.

# GLOSARIO

A continuación se lista, en orden alfabético, las diferentes abreviaturas utilizadas a lo largo de este informe:

**UTFSM**: Universidad Técnica Federico Santa María.
**VaR**: *Value-at-Risk*.
**MTM**: *Mark-to-Market*.

# ÍNDICE DE CONTENIDOS

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

# INTRODUCCIÓN

Los derivados financieros dentro del mundo de los negocios son una de las formas más comunes de manipular activos hoy en día, debido a lo fáciles que son de comprender, adquirir y transar, y lo flexibles que pueden ser al momento de querer generar un acuerdo en base a ellos, llegando a ser utilizados no sólo como una moneda de cambio dentro de un negocio, sino que como medida para amortizar pérdidas, para ajustar las tasas de interés de préstamos; para anteponerse a pagos futuros en una moneda extranjera. Sin embargo, el poder transar con ellos de forma inteligente, requiere de una serie de procesos matemáticos que permitan traerlos a lo que se conoce como Valor Presente, esto es, averiguar exactamente qué valor tienen estos derivados hoy en día en la moneda nativa del que los trance, lo cual se conoce como **Valorizar** dicho derivado. Este procedimiento, a pesar de no ser matemáticamente muy complejo, resulta muy largo para ser procesado por la computadora, lo cual se traduce en largos tiempos de espera, cosa que, al tratarse de un mundo tan volátil como son las finanzas, entorpece el trabajo de los tomadores de decisiones, quienes usualmente no disponen de mucho tiempo para concretar una oportunidad de negocios, y al no poseer la información necesaria para conocer si dicha oportunidad le será fructífera o no, corren en riesgo de perder cifras importantes de dinero en el proceso.

Es de ahí donde nace la motivación para este proyecto, el cual busca agilizar el proceso de valorización de derivados financieros y calcular el *Value-at-Risk* de los mismos, esto es, cuanto es lo máximo que se puede llegar a perder en un día, con un cierto grado de confianza. A modo de lograr esto se paraleliza el proceso de valorización, esto es: se correrá de forma simultánea en diferentes unidades de una misma máquina, a modo de ocupar de mejor manera los recursos disponibles de la misma. Este tipo de proceso tradicionalmente se hace a nivel de CPU a través de *threads*, los cuales corren de forma concurrente, pero en los últimos años se ha popularizado el lograr la misma tarea, pero a nivel de GPU, aprovechando la gran cantidad de núcleos que esta posee. Lo que se propone con este proyecto es unificar estos dos métodos, generando un algoritmo de paralelización Híbrido, esto es "*ser capaz de ejecutar el proceso de valorización, de forma simultánea, tanto en CPU como en GPU*".

El objetivo de este proyecto, además de diseñar un modelo de paralelización híbrido y de comparar el desempeño de las diferentes técnicas de paralelización entre sí, es determinar qué tan útil y efectivo es este modelo híbrido al momento de querer resolver un problema como es la valorización de derivados financieros, lo cual se medirá no sólo en términos de velocidad de procesamiento, sino que también en base a la complejidad que conlleva implementar este modelo de procesamiento híbrido, con el fin de descubrir qué tan práctico puede ser implementar este tipo de procesamiento en un ambiente de trabajo, en contraste con realizar esta labor de forma secuencial, o mediante un modelo de paralelización más clásico.

Además, con este proyecto se espera demostrar que el uso de la computación en paralelo para resolver problemas matemáticos densos, a pesar de complejizar la implementación del

mismo, genera de por sí mejoras significativas que justifiquen su uso, cosa que, a pesar de no ser herramientas de vanguardia, son muy poco utilizadas a nivel nacional.

Este informe comenzará definiendo de forma más formal el problema a resolver, continuando con un marco conceptual, el cual busca dejar en claro algunos conceptos más especializados que no todos los lectores puedan conocer, para luego pasar de lleno a explicar las diferentes formas en las cuales se implementan los distintos modelos paralelos, y finalmente comparar el desempeño de los mismos, mediante análisis de tiempo de procesamiento, a modo de descubrir que tan útil realmente puede llegar a ser un modelo de paralelización híbrido para resolver este tipo de problemáticas.

# CAPÍTULO 1
# DEFINICIÓN DEL PROBLEMA

Los derivados financieros son uno de los principales instrumentos financieros que se transan en este país, debido a que, entre otras cosas, permiten a los inversionistas anticiparse y cubrirse de los riesgos y/o cambios que puedan ocurrir en el futuro, de tal manera de evitar ser afectados por situaciones adversas [CMF, 2018]. Sin embargo, el trabajar con estos instrumentos también conlleva una serie de desafíos, tales como la valorización de los diversos datos de una cartera, o la predicción del *Value-at-Risk* de la misma, ambas operaciones son cruciales al momento de tomar una decisión financiera, pero al tratarse de activos con valores cambiantes en el tiempo, estas operaciones se deben realizar a diario, y si tomamos en cuenta la cantidad de datos que pueden existir en una cartera (considerando que es normal que los contratos que conformen dicha cartera se puedan demorar meses, incluso años, en llegar a su fecha de vencimiento) y más aún, si pretendemos simular más datos en pos de una estimación del *Value-at-Risk* mas precisa, estamos hablando de procesar millones de datos día a día, para cada una de las diferentes carteras de negocios de un inversionista o empresa. Por lo tanto, el tiempo de procesamiento tiende a alargarse de forma significativa al trabajar con derivados financieros.

Estos tiempos de procesamiento largos, se convierten en un problema cuando se analiza el coste de oportunidad de las decisiones financieras tomadas, y como este se ve directamente afectado por los datos de los que se dispone al momento de tomar la decisión. Más aún, en el mundo de los derivados financieros, las oportunidades de negocios son pasajeras, y las ofertas de compra y/o venta de un cierto activo tienden a estar disponibles por un tiempo limitado. Aquel inversionista que se decida a invertir en un negocio lo más rápido posible y con el respaldo matemático de que invertir en tal negocio es una buena idea, termina siendo el que obtiene mayores ganancias en este mercado. El tiempo se convierte en un factor clave, y estos inversionistas (los cuales pueden ir desde particulares o empresas, hasta entidades bancarias) siempre están dispuestos a invertir en pos de obtener información que no sólo les entregue una ventaja al momento de tomar decisiones, sino que también sea capaz de entregar dicha ventaja en el menor tiempo posible.

Actualmente, cada entidad que decida invertir en este tipo de instrumento, debe disponer de alguna herramienta para poder obtener información respecto a sus datos, siendo los inversionistas más grandes (generalmente los bancos) los que disponen de las herramientas más poderosas para realizar este tipo de cálculos, pero al mismo tiempo, son estos los que poseen la mayor cantidad de datos para analizar. Por lo tanto, todos los inversionistas que trabajan en este mercado, poseen algún tipo de defecto que buscan mejorar en lo que respecta a la valorización y procesamiento de sus datos. Luego, los inversionistas más pequeños no suelen tener acceso a un sistema propio para obtener esta información, y de tenerlo suele ser bastante rudimentario, y por el poco capital que poseen (en comparación a inversionistas más grandes), buscan sistemas razonablemente rápidos a bajo costo. Por otra parte, los

inversionistas más grandes ya tienen sus propios sistemas funcionando, y son capaces de realizar inversiones más significativas en pos de conseguir mejoras sobre los mismos. A pesar de que requieren resultados en el menor tiempo posible, al poseer sus propios sistemas, van a preferir tecnologías que se acoplen a lo que ya tienen por sobre sistemas completamente nuevos. También se da que existen empresas que se dedican a diseñar este tipo de *software* y venderlo como solución a estas entidades. Pero, a nivel nacional el máximo grado de optimización en términos temporales que se realiza es paralelización en CPU, lo cual, hoy en día, se sabe que entrega peores resultados al trabajar sobre grandes conjuntos de datos que la paralelización en GPU.

A modo de intentar resolver esta problemática, se propone un modelo híbrido, el cual realiza procesamiento en paralelo tanto a nivel de CPU como GPU simultáneamente, con el objetivo de probar si este enfoque logra reducir los tiempos de procesamiento de cálculos sobre una cartera de derivados, en comparación a otros métodos. Por lo tanto, además de diseñar una versión híbrida de los cálculos, se diseñaron versiones secuenciales y paralelas sólo en CPU y en GPU, a modo de poder determinar como la velocidad del cálculo varía dependiendo de la versión que se utilice y del *hardware* donde dicha versión del proceso se ejecute.

## 1.1. OBJETIVOS

- Diseñar una versión paralela híbrida (procesando simultáneamente datos en CPU y en GPU) de los cálculos que usualmente se aplican sobre derivados financieros (esto es, valorización y simulación de escenarios futuros).
- Probar esta versión paralela híbrida contra versiones clásicas del cálculo (esto es, una versión secuencial, una versión paralela en CPU y una versión paralela en GPU), comparando las diferencias en cuanto a tiempo de procesamiento, a modo de comprobar si esta versión paralela híbrida resulta más rápida que métodos más tradicionales de programación.
- Medir el comportamiento de la versión paralela híbrida en diferentes piezas de *hardware*, a modo de extrapolar el comportamiento del código dependiendo de la máquina en la cual se ejecute.

A modo de acotar el alcance del proyecto en pos de cumplir con estos objetivos, las técnicas matemáticas de valorización y simulación no serán excesivamente complejas y no se pondrá mayor énfasis en la precisión del cálculo, por lo que se permitirá un cierto margen de error con respecto al valor real. Todo esto se discutirá con mayor detalle más adelante en el informe.

# CAPÍTULO 2
# MARCO CONCEPTUAL

## 2.1. DERIVADOS FINANCIEROS

En el contexto de las finanzas, un derivado puede definirse como un instrumento financiero cuyo valor depende de (o deriva de) el valor de otra variable más básica, comúnmente referida como el activo subyacente [Hull, 2003]. Estos activos subyacentes pueden ser virtualmente cualquier cosa, desde una materia prima (petróleo, oro, carbón, etc), hasta las acciones de alguna empresa, pero todos tienen una característica en común: su valor fluctúa constantemente a lo largo del tiempo, haciendo que sea muy difícil el comercializar directamente sobre estos activos subyacentes, haciendo que se vuelva cada vez más necesario trabajar sobre un instrumento que represente a estos activos, y cuyo valor sea mucho más fácil de controlar y, por lo tanto, de invertir en el mercado, he ahí la razón por la cual se utilizan los derivados financieros.

Precisar exactamente cuándo se originan estos instrumentos no es tarea sencilla, puesto que la idea de representar bienes complejos de manipular a través de un bien derivado del mismo es tan antigua como la propia escritura, puesto que este tipo de prácticas ya eran habituales durante el 8.000 A.C, en donde se utilizaban vasijas horneadas para representar diferentes activos, a modo de facilitar el trueque, y ya en el 3.500 A.C los sumerios empezaron a utilizar tablillas de piedra para transar sus bienes, en lo que se asemeja bastante a los contratos *Forward* que se utilizan hoy en día. [Mathers, 2013]

Más concretamente, los derivados modernos y las regulaciones para trazarlos, empezaron a surgir en el año 1848 en Chicago, EEUU, con la fundación del *Chicago Board of Trade* (CBOT), el primer *exchange* moderno, un lugar en donde la gente podía vender y comprar derivados fácilmente, sin necesidad de tener que encontrarse de forma privada entre las partes involucradas en el contrato, y con mayor seguridad al tener el contrato respaldado por una institución que velará porque éste se cumpla. Por último, desde 1970, con la masificación de las computadoras, los derivados financieros tomaron un fuerte impulso, ya que gracias a estas el transar los derivados entre las partes involucradas se volvió mucho más rápido, y la capacidad de cálculo de las mismas abrió la puerta a herramientas y técnicas de análisis más complejos, los cuales permitieron tomar más y mejores decisiones en relación a los negocios en donde los derivados se veían involucrados.

Más concretamente, los derivados financieros son, en su forma más básica, **contratos futuros** (los cuales pueden ser regulados por un tercero), en los cuales dos partes acuerdan transar un activo subyacente determinado bajo ciertas condiciones previamente estipuladas. Las condiciones y el rol de cada una de estas partes dependerá exclusivamente del tipo de contrato que se acuerde, pero todos tienen una característica en común, en el hecho de que el contrato se hará efecto en una cierta fecha a futuro. Como el valor del activo subya-

cente fluctúa a lo largo del tiempo, es imposible saber con certeza el valor que el contrato tendrá al momento de cobrarse, por lo cual los inversionistas deben ir analizando los posibles escenarios futuros, mediante simulaciones o proyecciones matemáticas, a modo de prepararse para estos y sacar el mayor provecho posible a su inversión, de ahí es donde nace el aspecto financiero relacionado con la tranza de derivados.

Antes de indagar en los diferentes tipos de derivados financieros, es necesario mencionar los diferentes mercados en los cuales se transan estos instrumentos. Existen dos tipos de mercados cuando se habla de derivados financieros: ***Exchange-Traded Markets*** y ***Over-the-Counter Markets***.

### 2.1.1. *EXCHANGE-TRADED MARKETS*

Este tipo de mercado, conocido comúnmente como *Exchange*, tiene la particularidad de ofrecer contratos estandarizados, por lo cual las partes interesadas deciden adquirir uno de los contratos ofrecidos por esta entidad la cual, por su parte, vela porque el contrato se cumpla dentro de los plazos establecidos. Como ya se mencionó, el *exchange* nació en el año 1848 con la CBOT la cual, en el año 1973, se convirtió en la *Chicago Board Options Exchange* (CBOE), la cual es hoy en día una de las compañías de *exchange* más grandes del mundo, ofreciendo contratos a inversionistas de todo el planeta. [CBOE, 2019]

### 2.1.2. *OVER-THE-COUNTER MARKETS*

No existe un intermediario en este tipo de mercado, los interesados en negociar un contrato se contactan de forma privada y coordinan las especificaciones entre ellos. Al utilizar este tipo de mercado, se hace un poco más difícil contactar a los interesados en comprar o vender un bien, y son mucho más riesgosos; al no existir una entidad reguladora se puede dar que una de las partes del contrato no lo honre en la fecha estipulada. Sin embargo, hoy en día gracias al uso masivo de los computadores, este tipo de contrato se hace cada vez más fácil de ejecutar y es bastante popular, gracias a la flexibilidad que entrega al momento de redactar el contrato, permitiendo la realización de tratos más personalizados que se ajusten mejor a las condiciones de los participantes. Por otra parte, existen ciertos tipos de derivados que sólo pueden trazarse a través de este tipo de mercado, por lo que en ciertas ocasiones se vuelve necesario tener que hacer uso de él.

### 2.1.3. CONTRATOS FORWARD

Existe una cantidad muy diversa de derivados financieros, cada una con contratos, los cuales van desde muy sencillos a sumamente complejo, sin embargo, para términos de la memoria, uno de los más importantes de estudiar es el **Contrato *Forward***.

Este tipo de contrato es el más antiguo, pero a la vez el más simple, tanto en términos de entenderlo como en términos de aplicarlo. Un contrato *Forward* es **un acuerdo en el cual una parte se compromete a vender o comprar un bien a un cierto precio en una cierta fecha futura**. [Hull, 2003]

En este tipo de contrato se pueden identificar dos tipos de participantes: el que toma la **Posición (o Pata) Larga** y el que toma la **Posición (o Pata) Corta**. El primero es el que se compromete a comprar el activo a un cierto precio en la fecha estipulada, mientras que el segundo es quien acuerda vender dicho activo en tal fecha y a tal precio. El tipo de contrato *Forward* más popular, es sobre monedas extranjeras, en el cual uno de los participantes le vende una cierta cantidad de divisa al otro participante en una fecha a futuro y a un cierto valor por divisa.

A continuación se presenta un ejemplo de lo que sería un contrato *Forward*: Supongamos que tenemos a dos inversionistas, A y B, en donde el inversionista A desea comprar 1.000.000 USD, pero los necesita dentro de 3 meses, a modo de pagar una deuda. Luego, el inversionista A desea poder prepararse para pagar esa deuda, pero por cómo fluctúa el precio en pesos chilenos del dólar, es imposible saber cuánto deberá pagar en pesos dentro de tres meses, por lo tanto, pacta un contrato con el inversionista B, en el cual se compromete a comprarle 1.000.000 USD dentro de 3 meses, a 650 CLP cada USD, por lo tanto, el inversionista A asume la posición larga, mientras que B asume la posición corta. De esta forma, el inversionista A es capaz de saber que, independiente de la fluctuación en el valor del dólar, su deuda actual ahora es de 650.000.000 CLP, por lo que le es más fácil separar esa suma de sus ganancias. Por otra parte, si al cumplirse la **fecha de vencimiento** del contrato (esto es la fecha pactada en la que se realiza la venta) el valor del dólar es menor a 650 CLP por USD, el inversionista B estará ganando dinero con este intercambio, al vender el dólar más caro de lo que realmente es, sin embargo, de valer más de 650 CLP por USD a la fecha de vencimiento, B terminara perdiendo dinero con este negocio, y he ahí la apuesta que B realiza al tomar parte en este tipo de contrato.

### 2.1.4. CONTRATOS *SWAP*

Entre las diferentes formas de transar con derivados financieros, probablemente la que hoy en día más se ocupa entre los grandes inversionistas es el **Contrato *SWAP***, tipo de contrato que nació durante la década de 1980, y que resulta sumamente atractivo debido a su flexibilidad y a lo cómodo que resulta el tener que pagarlo.

Un contrato *SWAP* es un contrato del tipo *Over-the-Counter*, en el cual dos entidades (usualmente compañías) acuerdan *transar el flujo de dinero en el tiempo de un cierto derivado financiero*. Al momento de acordar el contrato se deja estipulado no sólo el derivado sobre el cual se va a transar el flujo de dinero, sino que también la fecha de vencimiento del contrato, las tasas de interés a ser utilizadas durante el mismo, y la modalidad de pago de los flujos monetarios (si es que estos se pagarán sólo durante el vencimiento, o si se irán pagando

durante fechas estipuladas anteriormente).

De inmediato, se puede observar la mayor diferencia que existe entre este tipo de contratos y otros más clásicos, como podría ser un *Forward*: el derivado en cuestión no se tranza, sino que se utiliza como una referencia, y lo que se transa es el flujo de dinero que ocurre sobre este derivado, usualmente mediante una cierta tasa de interés que se le aplica al mismo. En este sentido, es similar a un *Forward* en el que, después de haber adquirido el derivado en cuestión, éste se vendiera de inmediato. Por ejemplo, se tiene un contrato *Forward* en el cual se busca comprar 100 onzas de oro a un valor fijo dentro de un año, pero una vez adquirido el oro éste inmediatamente se vende por el valor de mercado; esto sería equivalente a un contrato *SWAP* en el cual una de las partes accede a pagar un cierto valor fijo, y la otra accede a pagar un valor equivalente al precio de 100 onzas de oro durante la fecha de vencimiento, de esta forma no se tranza el oro como tal, sino que únicamente se tranza un flujo de dinero el cual utiliza 100 onzas de oro como su valor de referencia.

Existen diferentes tipos de contratos *SWAP*, pero lejos el que más se tranza es el de Tasa de Interés, también conocido como *Plain Vanilla*. En este, dos compañías intercambian las tasas de interés a ser pagadas para un cierto valor nominal, y se suele utilizar mucho para pagar préstamos, pero no se está contento con el tipo de tasa de interés que se obtiene para el mismo. En un *SWAP Plain Vanilla* tenemos una compañía que paga a otra un flujo monetario igual al interés en tasa fija de un cierto nominal, mientras que la otra compañía, a su vez, le paga el flujo monetario sobre la misma cifra nominal, pero el interés se paga sobre una tasa flotante, de esta forma ambas compañías efectivamente intercambian las tasas de interés que debe pagar para un mismo nominal, teniendo más control sobre la forma en que pagan por sus préstamos.

En este tipo de contrato la tasa flotante con la que más se suele transar es la **L.I.B.O.R** (*London Interbank Offered Rate*), la cual se da a conocer a diario, y especifica valores de tasa de interés para las principales monedas con las que se transa en el mundo, en plazos de 1, 3, 6 y 12 meses, lo cual la convierte en una tasa de interés sumamente completa.

Otro tipo de contrato *SWAP* que se suele utilizar bastante es el de Divisas (conocido como ***Cross Currency SWAP***), en el cual dos entidades se tranzan el flujo de dinero de una cierta divisa, y éste se paga en un moneda diferente a la divisa transada. Por ejemplo, podemos tener una entidad que, dentro de un año, necesita tener la suficiente cantidad de dinero en pesos chilenos como para comprar 100.000.000 USD, pero como el dólar fluctúa tanto no se quiere arriesgar a comprarlos en la fecha que necesita, por lo que realiza un contrato *SWAP* con otra entidad, en la cual él asume una tasa fija, diciendo que para la fecha de vencimiento del contrato, pagará 650 CLP por dólar, mientras que la otra entidad, la cual asume una tasa flotante, está dispuesto a pagar una cifra en CLP equivalente de 100.000.000 USD, con el valor del dólar a la fecha de vencimiento, de esta forma la primera entidad sabe exactamente cuánto dinero tiene que pagar en CLP, en lugar de arriesgarse con las fluctuaciones del dólar, y la segunda entidad hace una apuesta, en la cual, dependiendo del valor del dólar a la fecha, puede ganar más dinero del que invierte.

## 2.2. *MARK-TO-MARKET*

El concepto de ***Mark-to-Market*** (**MTM**) se refiere a un método de contabilidad, el cual estima el valor real que un activo posee actualmente en el mercado, y básicamente muestra cuando se ganaría si dicho activo se fuera a vender hoy [Amadeo, 2019]. De este modo, los inversionistas pueden tener una mejor idea de cómo el precio de sus activos fluctúa a lo largo del tiempo, y cuanto dinero van ganando (o perdiendo) al mantener la posesión de dicho activo.

Este concepto se vuelve especialmente importante al trabajar con derivados financieros, debido a que el valor del activo subyacente varía día a día, el MTM asociado a tal activo también cambiará drásticamente de precio día a día, por lo tanto se vuelve necesario conocer el valor real de los derivados a diario, a modo de entender de mejor manera cómo va evolucionando la inversión, y prepararse para el resultado final de la misma una vez llegada la fecha de maduración del contrato. El proceso de calcular a diario el valor real de los derivados que posee un inversionista es lo que se conoce como **Valorización**, y dependiendo del inversionista y de la cantidad de derivados que actualmente posee, puede consumir bastante tiempo día a día sólo para conseguir esta información.

### 2.2.1. VALORIZACIÓN DE DERIVADOS FINANCIEROS

Como ya mencionamos, al hablar de Valorización nos referimos a calcular día a día, el MTM de todos los derivados que posee un inversionista, a modo de extraer información de los mismos, la cual utiliza para apoyar las decisiones financieras que toma durante dicho día.

Debido a la naturaleza cambiante de los derivados financieros, es de suma importancia realizar este proceso todos los días, ya que cada día el valor de los derivados financieros puede cambiar, y se nos es imposible conocer con certeza el grado de dicho cambio. Es por esto que, además de Valorizar sus derivados, los inversionistas aplican herramientas matemáticas sobre los MTM históricos de un cierto derivado, a modo de identificar patrones en su comportamiento, y utilizar dichos patrones para tratar de predecir su comportamiento futuro, con el fin de prepararse de mejor manera para los cambios que sus inversiones puedan tener.

## 2.3. CARTERA DE DERIVADOS

Trabajar con Derivados financieros es cada vez más sencillo, hoy en día existen múltiples formas de adquirir estos contratos, por lo cual el que un inversionista trabaje con cientos de contratos distintos día a día se está volviendo cada vez más normal, luego, el considerar cada uno de estos derivados como un negocio separado resulta bastante tedioso y termina generando mucha desorganización, dificultando el trabajo de los inversionistas.

Por esta razón, en lugar de ver cada derivado como un negocio separado, lo que se está haciendo es agrupar conjuntos de derivados en lo que se conoce como **Carteras**, y cada cartera se considera un negocio diferente. Esto trae consigo una serie de beneficios: primero, pasamos de tener cientos de derivados que analizar a algunas decenas de carteras, lo cual es una cifra mucho más manejable, pero lo más importante que se obtiene al aplicar este concepto de cartera es que permite ir sopesando las pérdidas de una inversión con las ganancias de otra, de esta forma, en lugar de tener dos derivados, uno que nos está haciendo perder dinero y otro que nos está generando dinero, se agrupan en una sola cartera, y si las ganancias de un derivado son mayores que las pérdidas del otro, dicha cartera nos está generando ganancia, luego, las diferentes decisiones financieras que se toman se hacen en pos de obtener ganancias en cada una de las Carteras de derivados que se poseen actualmente.

## 2.4. *VALUE-AT-RISK*

El *Value-at-Risk* (VaR) se define como la *Predicción de la peor pérdida posible (esperada) para un nivel de confianza dado para un periodo de tiempo determinado*. En otras palabras, dado un cierto rango de tiempo y un cierto grado de precisión esperado, se predice el peor escenario posible para una cierta inversión.

Este concepto nació en la década de 1980, producto de la popularización y crecimiento de los derivados financieros, y del hecho de que las medidas tradicionales para el manejo de riesgos eran inadecuadas para analizar estos tipos de bienes. Sin embargo, no fue hasta la década de 1990 que el uso de VaR se masificó, después de que diferentes comités y grupos de inversionistas de alto prestigio probaron y demostraron la efectividad de este método.

El principal atractivo que posee el VaR es su simplicidad al momento de difundirlo entre los tomadores de decisiones. A modo de entender esto, se presenta el siguiente ejemplo de cómo se explicaría el VaR de una cartera de negocios en un determinado día:

- *El VaR de 1 día de la cartera de negocios de mi empresa es de 100.000.000 CLP con un 95 % de confianza*. [Brand, 2003]

Esto significa que, al finalizar el día de mañana, la probabilidad de que la cartera de negocios de mi empresa valga 100.000.000 CLP menos que lo que vale hoy es del 5 %.

Como se puede ver, para un tomador de decisiones, quien probablemente no tiene la *expertise* matemática para comprender las diferentes operaciones y simulaciones que validan esta declaración, le resulta muy sencillo comprender esto, y con una sola oración es capaz de comprender la situación actual de su cartera de negocios, y se puede aprovechar de esta información para tomar las decisiones financieras correctas a lo largo del día.

## 2.5. SIMULACIÓN DE MONTE CARLO

El método de Monte Carlo fue inventado durante la década de los 40 para ser utilizado durante el proyecto Manhattan [Metropolis y Ulam, 1949], y se refiere a cualquier algoritmo computacional que se base en simular múltiples valores aleatorios a modo de obtener un resultado numérico, con el fin de utilizar la aleatoriedad de estos valores para poder resolver un problema determinista.

Se basa en **estadísticas inferenciales**, y en la idea de que dado un cierto conjunto de datos, cuyo tamaño es demasiado amplio como para ser analizado en su totalidad, el cual se conoce como una **Población**, y entiéndase como **Muestra** a un subconjunto de dicha población. Si consideramos una muestra seleccionada de forma completamente aleatoria, siempre que sea lo suficientemente grande, esta presenta las mismas propiedades que las de la población de la cual se extrae, lo cual mediante inferencia, nos permite obtener información sin tener que analizar por completo a este último.

Además, este método resulta muy útil en campos como la física o la matemática pura, permitiendo resolver problemas complejos de forma empírica, o simulando sistemas con comportamientos difíciles de analizar individualmente. Pero a lo que respecta a la computación, se suele utilizar principalmente en simulaciones, ya que entrega resultados muy similares a la realidad utilizando relativamente pocos datos, razón por la cual juega un rol central en el desarrollo de este proyecto.

## 2.6. COMPUTACIÓN PARALELA

Tradicionalmente, al desarrollar un *software* este se piensa, y, por lo tanto, se escribe de forma secuencial, esto es tomar un problema a resolver, dividir dicho problema en una serie de instrucciones, y ejecutarlas de forma secuencial, una después de la otra, hasta que la última instrucciones se termine de ejecutar y el problema se de por resuelto. Inconscientemente, al diseñar *software* de esta forma, este está pensado para ser ejecutado en una sola computadora, ocupando una sola CPU, de la misma forma en que las soluciones se diseñan durante los primeros años de la computación moderna. Sin embargo, hoy en día no sólo se tiene acceso a computadores mucho más poderosos que los que existían en dicha época, sino que la capacidad de poder trabajar sobre redes de computadores a través de *internet*, hace que este tipo de técnica de programación entregue resultados sub-óptimos y no utilice de forma apropiada todos los recursos disponibles.

Aquí es donde nace la técnica de **Computación en Paralelo**, la cual se puede definir simplemente como el "*uso simultáneo de múltiples recursos computacionales para resolver un problema*" [Barney, 2007], mediante el uso de múltiples CPUs. Al paralelizar una solución lo que se está haciendo, es dividir el problema en secciones discretas, las cuales pueden ser resueltas de forma concurrente. Esto implica que las operaciones que cada una realiza no

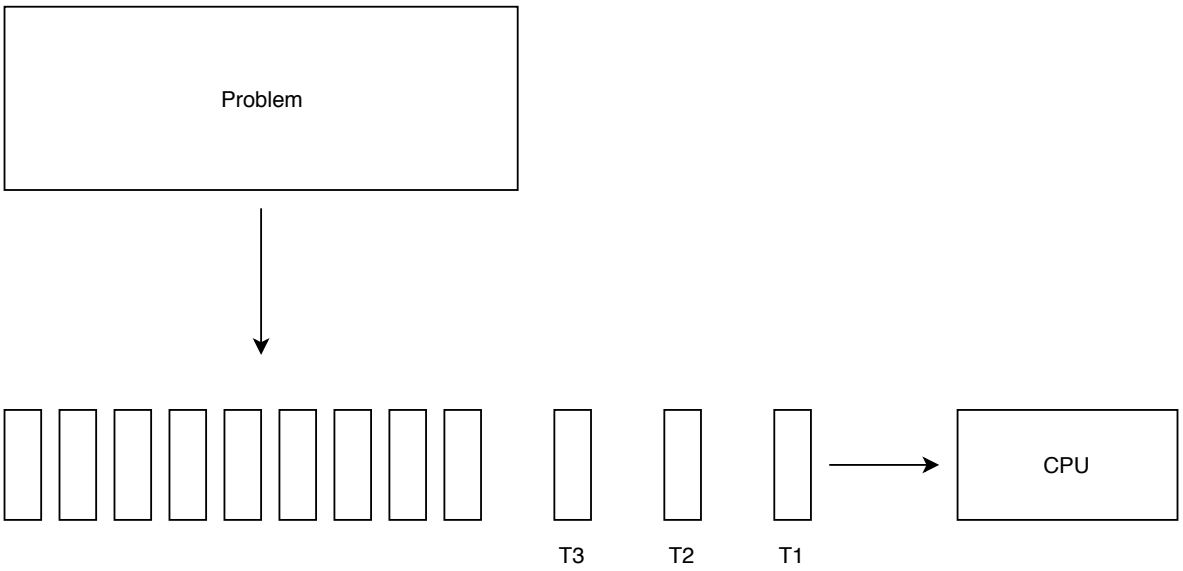



Figura 1: Ejemplo de computación secuencial.
Fuente: Introduction to Parallel Computing. [Barney, 2007]

dependan directamente del resultado de las operaciones de otra secciones, lo cual permite que ambas secciones puedan ser ejecutadas simultáneamente. Cada una de estas secciones se convierte en una serie de instrucciones, las cuales se ejecutarán en CPUs distintas, lo cual permite que todas se completen simultáneamente.

Originalmente, este tipo de técnica se consideraba como computación de alto nivel, y su uso era casi exclusivo para problemas de simulación complejos, o para resolver desafíos computacionales, pero hoy en día, en donde resulta cada vez más sencillo obtener acceso a equipos *multi core*, y debido a la necesidad cada vez más común de tener que procesar grandes volúmenes de data, es que las aplicaciones comerciales de la computación paralela se han vuelto sumamente atractivas e incluso necesarias, generando un mayor en el estudio de la misma.

La principal motivación de implementar este tipo de computación es el tiempo de procesamiento mediante procesamiento en paralelo, se puede llegar a reducir de forma significativa el tiempo que le toma a una solución completarse, lo cual es sumamente deseado por múltiples entidades, desde usuarios promedio hasta grandes empresas, mientras menos tiempo tengan que esperar por una respuesta, más útil les resulta la solución. Por otro lado, la computación en paralelo también permite abordar problemas más grandes y complejos, repartiendo la carga de las operaciones entre múltiples procesadores, con lo cual se logra obtener resultados que, bajo circunstancias normales, le serían imposible de obtener a un solo procesador, además de aportar concurrencia a la solución, permitiendo que esta utilice de mejor manera los recursos de *hardware* de los cuales se dispone.

A modo de entender de mejor manera la idea detrás de la computación en paralelo, se presenta el siguiente ejemplo de la vida cotidiana: imagínese a un individuo que todas las ma-

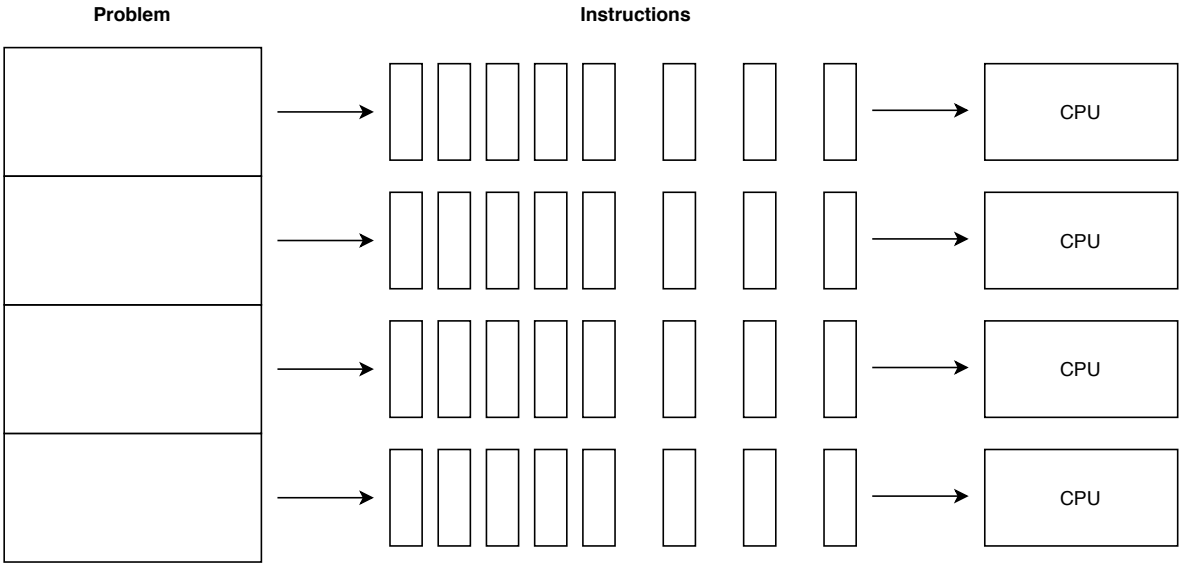



Figura 2: Ejemplo de computación paralela.
Fuente: Introduction to Parallel Computing. [Barney, 2007]

ñanas se prepara su desayuno, el cual consiste en un par de tostadas con huevo revuelto, una taza de café y un vaso de jugo, además dicho individuo tiene la costumbre de preparar el huevo antes de tostar el pan, para que este último no se enfríe, y antes de poder preparar sus café debe hervir el agua.

Según lo que se a visto, existen dos formas en que esta persona puede preparar sus desayuno en la mañana: la primera sería prepararlo de forma secuencial, esto significa completar cada una de las tareas que componen su desayuno una a la vez, luego para preparar su desayuno tendría que, primero, poner el agua en la tetera y esperar a que hierva, tras lo cual se puede preparar su café, acto seguido prepara los huevos en la sartén, tras lo cual pone el pan en la tostadora, esperar a que se tueste, y finalmente se sirve su vaso de jugo y empieza a comer su desayuno. Como se puede apreciar, este método de preparar desayuno ocuparía mucho tiempo, y mucho de este tiempo está desperdiciado, ya que mientras el agua se hierve o el pan se tuesta el individuo se queda haciendo nada, lo que se conoce como estado *Idle*.

Por otro lado, existe el método paralelo para que este individuo prepare su desayuno, el cual consiste en: poner a hervir el agua, y simultáneamente preparar los huevos en la sartén, ya que ambas operaciones son independientes la una de la otra, y no requieren de que algo más se haya completado anteriormente. Una vez cocinados los huevos, la persona puede poner a tostar el pan, y mientras el agua termina de hervir se sirve su vaso de jugo, luego prepara su taza de café, justo cuando los panes salen de la tostadora. Como se puede apreciar, este individuo realizó exactamente la misma acción, pero aprovecho de mejor manera el tiempo disponible, evitando lo más posible quedarse en un estado *idle*, lo cual le permite disfrutar antes de su desayuno y comenzar más temprano su día.

## 2.7. ALGORITMO DE ORDENAMIENTO *QUICKSORT*

Durante el desarrollo de este proyecto, un elemento crítico que se desarrolló fue el algoritmo de ordenamiento *Quicksort*, el cual se utiliza fuertemente al momento de determinar el valor del *Value-at-Risk*. Por esta razón se explica, a grandes rasgos, en qué consiste este algoritmo de ordenamiento.

El algoritmo *Quicksort* se basa en la idea de dividir un problema complejo en dos problemas más sencillos de resolver, los que a su vez pueden ser reducidos a problemas aún más sencillos en cada iteración. [Hoare, 1962]

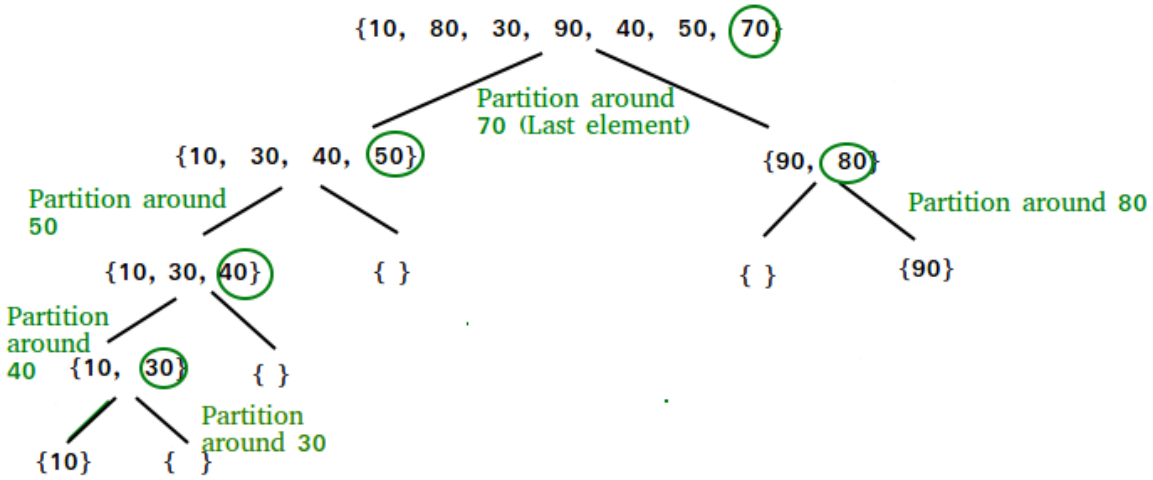



Figura 3: Ejemplo de ordenamiento mediante Quicksort.
Fuente: Quicksort, GeeksforGeeks, 2016 [1].

En términos prácticos, el algoritmo se utiliza para ordenar un grupo de valores, y lo hace mediante la determinación de un pivote, el cual es un cierto valor del conjunto, el cual se utiliza como punto de referencia. Luego, cada otro valor del conjunto se compara con el pivote y se ordenan a izquierda o derecha del mismo, dependiendo si son menores o mayores que el pivote, respectivamente. Una vez hecho eso, se aplica la misma operación sobre los sub-arreglos conformados por los valores a la izquierda y derecha del pivote. Cuando el tamaño de los sub-arreglos sea demasiado pequeño como para ser ordenado, se detiene la operación en dicha rama, y recursivamente se va operando en cada uno de los diferentes sub-arreglos que conforman el arreglo.

[1] QuickSort - GeeksforGeeks. Se recuperó 16/07/2019 desde https://www.geeksforgeeks.org/quick-sort/

# CAPÍTULO 3
# PROPUESTA DE SOLUCIÓN

Como ya se mencionó, a modo de resolver la problemática expuesta, se propuso paralelizar de forma híbrida los cálculos relacionados con la valorización de derivados financieros, con el fin de descubrir qué tan rápido puede llegar a ser este modelo en comparación con las técnicas de procesamiento y aceleración clásicas, como son los modelos secuenciales y los modelos paralelos, tanto en CPU como en GPU.

Antes de explicar la solución propuesta, es necesario mencionar que, a modo de poder asemejarse lo más posible a una situación real, durante el transcurso del proyecto se trabajó sobre datos reales, entregados por una entidad inversionista que prefiere mantenerse anónima. Este conjunto de datos consta de una cartera de operaciones, con cuatro operaciones de derivados financieros dentro de ella, cada una de las cuales tiene alrededor de cien días de pérdidas/ganancias registradas. Este volumen de datos tiene el problema de no ser particularmente grande, por lo tanto, el proceso de determinar el *Mark-to-Market* no se demora lo suficiente como para ser atractivo paralelizar. Por esta razón se decidió agregar el cálculo de VaR mediante simulación de escenarios futuros, a modo de incrementar el tiempo de procesamiento de la versión secuencial con el fin de generar un escenario más atractivo de paralelizar.

## 3.1. DESCRIPCIÓN GENERAL

Con el objetivo de determinar qué tan factible es utilizar paralelismo híbrido para reducir los tiempos de procesamiento, se genero una solución, la cual consiste en la valorización y análisis mediante cálculo de *Value-at-Risk*, de una cartera de derivados financieros reales. Dicha cartera consiste de cuatro operaciones *SWAP*, cada una con su respectiva posición larga y posición corta, y cada una con aproximadamente 100 días de antigüedad.

A modo de poder valorizar esta cartera y poder extraer información de la misma, la solución consiste en lo siguiente:

1. **Valorizar la cartera de derivados**, esto implica valorizar cada una de las posiciones para cada una de las operaciones.
2. Mediante simulación de Monte Carlo, **simular escenarios de valorización futuros**.
3. Aplicar un **algoritmo de ordenamiento sobre los datos simulados**.
4. **Determinar el *Value-at-Risk*** de cada una de las posiciones para cada operación de la cartera.

De manera de poder determinar la efectividad del paralelismo híbrido para solucionar esta problemática, es necesario generar diferentes versiones del mismo programa, a modo de poder ir determinando que se puede paralelizar, de qué forma puede paralelizar, y para poder usar como puntos de comparación una vez se empiecen a realizar pruebas sobre la versión final del código. Para esto se diseñó cuatro versiones de la solución, las cuales son:

- **Versión Secuencial**, la cual actúa como la base de las demás versiones, y en las pruebas funcionará como grupo de control (ninguna de las otras versiones se puede demorar más que ésta en promedio).
- **Versión Paralela en CPU**.
- **Versión Paralela en GPU**.
- **Versión Paralela Híbrida**, esto es, procesando de forma paralela tanto en CPU como en GPU al mismo tiempo.

A continuación se describe cada una de estas versiones y la lógica que existe detrás de ellas.

## 3.2. VERSIÓN SECUENCIAL

Como ya se mencionó, la versión secuencial de la solución propuesta busca funcionar como base, ya que a partir de esta no sólo se diseñan las diferentes versiones de la solución, sino que también actúa como guía al momento de identificar que secciones del código son paralelizables y que tanto se podría ganar al paralelizar dichas secciones. Por último, como ya se mencionó durante la validación de la solución, esta versión actúa como grupo de control para las pruebas, bajo la restricciones de que ninguna de las otras versiones puede ser más lenta que está para que se considere como alternativa válida.

El algoritmo secuencial está compuesto de cuatro pasos que deben realizarse a modo de poder obtener el VaR de cada uno de los contratos analizados. Estas operaciones, tal y como se mencionan en la sección anterior, son:

Valorizar la cartera de derivados. Simulación de escenarios futuros. Ordenar los datos mediante Algoritmo de Ordenamiento. Determinar el VaR.

Cada uno de estos pasos deben realizarse según el orden dado para una misma Operación financiera, ya que cada uno depende de los resultados del paso anterior. A continuación se describe en detalle cada uno de estos pasos.

### 3.2.1. VALORIZACIÓN DE LA CARTERA

El primer paso a completarse durante cada ejecución de la solución es la valorización de cada uno de los contratos que componen la cartera de negocios, lo cual nos dice el valor actual que posee cada uno de estos contratos en cada una de sus posiciones, y sobre el cual se deben realizar todas las simulaciones futuras para dicho día. El Algoritmo 1 muestra las acciones a seguir para completar este paso del algoritmo, y a continuación se explica con mayor detalle.

El proceso comienza cargando en memoria los datos tanto de la cartera de negocios (datos reales) como de la información de mercado, esta última se utiliza al momento de valorizar los derivados, y nos dice como instrumentos similares a éste, con una antigüedad similar a la antigüedad analizada, se comportan hoy en día en el mercado.

Una vez que estos datos se encuentran almacenados en memoria deben procesarse con tal de poder llegar a valorizar la cartera. El primer paso de la valoración es determinar las Pérdidas y Ganancias de un mismo día, lo que también se conoce como el Flujo de Efectivo de la operación. Es importante señalar que, este cálculo se realiza sobre cada una de las posiciones de cada operación, por lo que efectivamente se analizan una cantidad de escenarios igual al doble de la cantidad de operacion involucradas en la cartera (en este caso con una cartera de cuatro operaciones se están analizando ocho escenarios distintos).

Una vez determinado el flujo para cada posición de la operación, estas se almacenan en memoria y se comienza el siguiente paso en la valorización, en el cual se considera la información de mercado como una tasa de descuento que se le aplica al derivado, la cual depende de la antigüedad del mismo y en base a esta se genera un diferencial, el que se aplica sobre las pérdidas y/o ganancias del instrumento sobre cada una de las fechas analizadas a modo de introducir una información externa, la que nos dice como dicho derivado a ido evolucionando con el tiempo. Luego, la suma de productos entre este diferencial y el flujo calculado anteriormente nos entrega la valorización de la operación, en la posición correspondiente, el cual a su vez, como ya se explicó, nos dice cual es el valor efectivo que dicha inversión tiene el día de hoy.

### 3.2.2. SIMULACIÓN DE ESCENARIOS FUTUROS

Una vez valorados los derivados de la cartera y determinado el valor presente de cada uno de los contratos, se pasa a simular escenarios futuros a modo de obtener mayor información, la cual se utilizará posteriormente al momento de predecir el Value-at-Risk del día siguiente.

Para poder realizar la simulación, se toma la información del mercado, y se asume que esta se comporta según una distribución normal, lo cual permite aplicar una simulación de Monte Carlo sobre estos datos, determinando la media y desviación estándar de dicho conjunto y generando datos pseudo aleatorios sobre los cuales realizar las simulaciones.

**Algoritmo 1** Pseudocódigo algoritmo secuencial, valorización de derivados.

Leer datos de las Transacciones
Inicializar Valores_Iniciales
Guardar la inversión inicial en Valores_Iniciales
Inicializar Deals
Guardar los datos de las transacciones en Deals
Leer datos de Mercado
Inicializar Market_Data
Guardar los datos de mercado en Market_Data
Inicializar Vector_Paga
Inicializar Vector_Recibe
**for** Cada Deal en Deals **do**
    Procesar Perdidas y Ganancias del Deal
    **if** dirección de Deal == ”Paga” **then**
        Guardar el resultado en Vector_Paga
    **else**
        Guardar el resultado en Vector_Recibe
    **end if**
**end for**
Inicializar Suma_Paga
Inicializar Suma_Recibe
**for** Cada valor en Vector_Paga y Vector_Recibe **do**
    dif_dias = Fecha Inicio Operación - Fecha Valor
    dscto = lineal_interpol(dif_dias, Market_Data)
    $Df = \frac{1}{(1{,}0+dscto)^{\frac{dif\_dias}{360}}}$
    Agregar $Df * Valor$ a Suma_Paga o Suma_Recibe según corresponda
**end for**
Guardar la diferencia entre los valores iniciales y Suma_Paga o Suma_Recibe según corresponda.

**Algoritmo 2** Pseudocódigo algoritmo secuencial, simulación de escenarios futuros.

```
Mean = media de Market_Data
Std_Deviation = desviación estándar de Market_Data
Inicializar Random
for i < n do
    Random_i = RandomNormals(Mean, Std_Deviation)
    Guardar Random_i en Random
end for
Inicializar Results
for Cada conjunto en Random do
    Valorizar los datos en Vector_Paga y Vector_Recibe, usando el conjunto como la informacion de mercado
    Guardar la diferencia entre el valor actual y el valor simulado en Results
end for
```

A modo de poder generar casos de prueba, lo que se hace es crear una cierta cantidad de conjuntos de información de mercado, en base a un valor entregado por el usuario, donde cada uno de estos conjuntos posee una cantidad de datos igual a la cantidad máxima de fechas registradas entre las operaciones de la cartera financiera. Luego, la simulación involucra valorizar cada una de las operaciones de la cartera, en cada una de sus direcciones, y guardar el valor de la diferencia entre el escenario simulado para el día siguiente y el valor de la operación el día de hoy.

De esta forma, para cada una de las operaciones en cada dirección que posee la cartera, terminaremos con un conjunto de variaciones que tendrá el valor de la cartera para el día siguiente.

Con esto queda claro que la predicción de la estimación depende directamente del valor entregado por el usuario: mientras mayor sea más diverso será el conjunto de simulaciones, por lo que el *Value-at-Risk* entregará resultados más precisos, sin embargo, esto también incrementa la complejidad del cálculo, lo que a su vez ralentiza la velocidad de ejecución del sistema.

Luego, y considerando que los cálculos necesarios para la simulación son independientes entre sí (cada simulación se podría procesar por separado sin afectar al resto), es que el primer paso a paralelizar se descubre, y en las versiones posteriores del sistema se busca mejorar el desempeño del código mediante paralelizar los cálculos involucrados en la simulación de Monte Carlo. Se analizará este punto con mayor detalle en secciones posteriores.

### 3.2.3. ORDENAMIENTO DE LOS DATOS

Como ya se mencionó, el ordenamiento de los datos se realiza mediante un algoritmo *Quicksort*, el cual se describe en el Algoritmo 3.

**Algoritmo 3** Pseudocódigo algoritmo secuencial, algoritmo de ordenamiento Quicksort.

```
Se inicializa Pivote
Se asigna un Elemento de Results al Pivote
for Cada otro Elemento de Results do
    if Elemento < Pivote then
        Colocar Elemento a la izquierda del Pivote en Results
    else
        Colocar Elemento a la derecha del Pivote en Results
    end if
end for
Separar arreglo Results en dos sub arreglos, a la izquierda y derecha del Pivote
if Tamaño de cada sub arreglos es mayor a dos Elementos then
    Repetir Algoritmo sobre cada sub arreglo
end if
```

Se explicará con mayor detalle en secciones subsecuentes, pero se decidió utilizar un algoritmo de ordenamiento *Quicksort* debido en parte a lo eficiente que es al momento de ordenar grandes conjuntos de datos, pero también por lo sencillo que resulta el paralelizar dicho algoritmo, al menos desde el punto de vista conceptual.

Además, a pesar de que *Quicksort* no es el único algoritmo paralelizable, otras opciones como *Mergesort* y *Radixsort* presentaron múltiples problemas al momento de trabajar sobre valores con tantos dígitos (estamos hablando de valores que tienen entre 6 a 10 dígitos cada uno) razón por la cual se decidió no utilizarlos y, en cambio, optar por el algoritmo más sencillo de implementar.

### 3.2.4. CÁLCULO DEL *VALUE-AT-RISK*

En el Algoritmo 4 se presentan los pasos a seguir a modo de poder determinar el *Value-at-Risk* de una cartera de derivados, los cuales simplemente determinan el percentil del conjunto de datos, una operación matemática con un grado de complejidad relativamente bajo y cuya ejecución no debería tener mayor peso en el tiempo de procesamiento total de la solución, razón por la cual se decidió no paralelizar esta sección del código, y simplemente ejecutarla de forma secuencia.

**Algoritmo 4** Pseudocódigo algoritmo secuencial, cálculo *Value-at-Risk*.

```
Determinar grado de precisión (por defecto, Precisión = 5 %)
Inicializar Índice
Índice = (Tamaño de Results)*(Precisión)
Inicializar VaR
if Índice es entero then
    VaR = (Results[Índice-1] + Results[Índice])/2
else
    Se redondea Índice
    VaR = Results[Índice]
end if
Retornar VaR
```

### 3.2.5. EVALUACIÓN VERSIÓN SECUENCIAL

Una vez que se terminó de implementar este algoritmo, y a modo de tomar una mejor decisión al momento de paralelizar, se realizó un *profiling* del mismo, el cual se incluye en el Anexo A de este documento, de tal manera de descubrir qué secciones del código poseen el mayor impacto en términos de tiempo de procesamiento.

Con esta información se puede concluir que, como ya hemos mencionado, las dos secciones que más conviene paralelizar son: la simulación de los escenarios futuros y el ordenamiento de los conjunto de datos creados.

## 3.3. VERSIÓN PARALELA EN CPU

Para la versión paralela del código a nivel de CPU, como ya se determinó anteriormente, se busca paralelizar tanto la simulación de escenarios futuros como el ordenamiento de dichos datos simulados, puesto que estos puntos son los más críticos en términos de tiempo de procesamiento.

Cabe destacar que el ordenamiento depende de la simulación, no se puede empezar a ordenar el conjunto de datos antes de que dicho conjunto exista. Para lidiar con esto existen dos enfoques que se consideraron durante el desarrollo del proyecto: completar todas las simulaciones y después ordenar los conjuntos, o simular un conjunto y mientras este se ordena, simular paralelamente el siguiente,lo que se conoce como *Pipeline* [Ramamoorthy y Li, 1977].

El problema que presentó el enfoque *Pipeline* en este caso está en que, asumiendo que cada una de las operaciones se paraleliza de forma independiente, al ejecutarlas de forma paralela simultáneamente los *threads* disponibles se distribuirán entre ellas, efectivamente reduciendo el grado de paralelización que cada una de esta podría llegar a obtener de forma independiente, y al trabajar sobre máquinas con CPUs que no dispongan de una gran can-

tidad de núcleos, esta reducción se traduciría en una mayor pérdida de tiempo que lo que podría ser la ganancia potencial. Sin embargo, este método sigue siendo bastante viable, y si se tuviera una CPU más poderosa sería sumamente interesante de implementar, y probablemente, con este tipo de *hardware*, su desempeño sería mucho mejor que el otro método, pero para lo que respecta a este proyecto, se optará por generar las simulaciones y luego ordenar los conjuntos, ambos procesos ejecutados de forma paralela.

### 3.3.1. SIMULACIÓN DE MONTE CARLO EN CPU

Como ya se mencionó en la sección anterior, la simulación de Monte Carlo que se realiza busca valorizar la cartera de negocios, utilizando como información de mercado valores pseudoaleatorios, generados a partir de las características propias de dicha información de mercado ya que, si se asume que los datos que la componen se comportan según una distribución normal, esto se puede lograr conociendo únicamente la media y la desviación estándar de dichos datos. Luego, una vez generados estos sets de datos pseudoaleatorios, se realizan una serie de procesos de valorización distintos, cuyos resultados se almacenan para su posterior análisis.

Para poder paralelizar este tipo de proceso, se debe entender donde está la independencia de estos datos, ya que es imposible paralelizar procesos que dependan de otros. Como se pudo apreciar en la sección anterior (Algoritmo 1), el proceso de valorización se puede ver desde tres ángulos distintos, cada uno con un cierto grado de independencia: como **Operación Financiera**, como **Cantidad de Simulaciones** o como **Simulación Individual**.

Si se mira el proceso como el conjunto de las diferentes simulaciones individuales que se tienen que llevar a cabo, se puede apreciar que cada una de estas es independiente, por lo que podrían asignarse como tareas individuales a ser procesadas por los diferentes *threads* del procesador. Sin embargo, como los resultados de las mismas requieren guardarse en arreglos en memoria compartida para poder seguir siendo procesador por el sistema, el generarlos de forma individual conllevara una gran cantidad de conmutación de tareas al momento de guardar los resultados, y como los cálculos individuales no poseen una complejidad matemática tan extensa, se desaprovecha la capacidad de procesamiento de la CPU al forzarla a ejecutar un cálculo, para luego esperar su turno para guardarlo en memoria, por lo tanto se descartó este modelo.

Por lo tanto, concretamente sólo hay dos enfoques sobre los cuales se puede realizar la paralelización del cálculo: por **Operaciones Financieras** o por **Cantidad de Simulaciones**. El primer nivel implica entregar las simulaciones de cada uno de los derivados financieros, en cada una de sus direcciones a una CPU distinta, con lo cual cada una de estas calcularía su propio conjunto de simulaciones correspondientes al derivado que se le asigna. El segundo nivel implica que, para cada derivado financiero, en cada una de sus direcciones, las diferentes simulaciones que se deben realizar se reparten de forma equitativa entre las distintas CPUs que posee el equipo donde se esté ejecutando el proceso, de esta forma la carga de

las simulaciones se reparte de mejor manera entre los procesadores disponibles.

Ambas opciones deberían entregar resultados similares, pero para el desarrollo del proyecto de memoria se optó por la segunda, esto es, para cada una de las operaciones financieras a analizar, se divide el total de simulaciones a realizarse por la cantidad de CPUs disponibles en el equipo en el cual se realiza el cálculo, a modo de que cada una de estas se encargue de simular una fracción del conjunto total para cada una de las operaciones financieras que se analicen. La razón de esta elección se debe principalmente al *hardware* que se tiene disponible para realizar las pruebas, el cual no posee una CPU particularmente poderosa o con múltiples núcleos, haciendo que cálculos más acotados se comporten de mejor manera que cálculos más extensos. Pero cabe destacar que, si se utilizara un procesador con mayor capacidad, la primera opción puede que entregue resultados en un tiempo mucho más acotado que la opción que se optó por implementar en este proyecto, cosa que sería interesante probar a futuro.

### 3.3.2. ALGORITMO DE ORDENAMIENTO EN CPU

Uno de los principales desafíos de programación con los que se enfrentó durante el transcurso de esta memoria, a sido la paralelización del algoritmo *Quicksort*, particularmente en lo que respecta a la paralelización a nivel de CPU.

Las razones por la que se decidió hacer uso de este algoritmo para la realización de este proyecto son: primero, el hecho de que *Quicksort* es completamente paralelizable, ya que cada uno de los sub-arreglos que existe en cualquier momento es completamente independiente de los demás, por lo que no existe problema en operar sobre ellos de forma paralela. La segunda razón por la que se escogió *Quicksort* en lugar de algún otro algoritmo paralelizable, es por la velocidad de éste, ya que el principal punto de este proyecto es disminuir los más posible el tiempo de procesamiento de la valorización de derivados financieros. A pesar de que los resultados no son tan precisos como podrían llegar a ser con otros algoritmos, estos se obtienen de forma más rápida que con otros métodos de ordenamiento capaz de ser paralelizados.

Como ya se mencionó, *Quicksort* es (al menos teóricamente) paralelizable, y existen diversos ejemplos de cómo lograr esta paralelización en la literatura, en los cuales, al generarse un nuevo sub-arreglo, se creaba también un nuevo *thread* en el sistema y se le ordenaba a ese *thread* procesar dicho sub-arreglo, con lo cual se van creando de forma dinámica *threads* que completen la labor de ordenar estos datos, y de forma recursiva van dando con un arreglo final ordenado. El problema es, que todos estos ejemplos e implementaciones operan sobre conjuntos de datos relativamente pequeño en comparación al nuestro, el cual trabaja sobre el orden de millones de valores simulados, donde cada uno de estos valores se mueve entre los 7 y 9 dígitos, por lo que rápidamente se concluyó que las técnicas normales de paralelización de este algoritmo simplemente no serían aplicables en este caso, debido a que el procesador es incapaz de crear una cantidad tan alta de *threads* como para llevar a

acabo esta operación.

Por lo tanto, se generó un modelo capaz de procesar de forma paralela el algoritmo *Quicksort* a nivel de CPU, el cual se detalla en la Figura 4. Este modelo comienza con una iteración normal del algoritmo, dando con los dos sub-arreglos iniciales, tras lo cual se crean dos *threads*, en donde cada uno de estos realiza un *Quicksort* de forma secuencial, hasta generar sub-arreglos cuyo tamaño sea menor a un cierto límite predefinido. Una vez alcanzado esto, los índices de dicho sub-arreglo se almacenan dentro de una pila de tareas, en donde esperan a que ambos *threads* terminen de analizar sus correspondientes secciones del arreglo.

En cuanto la lista de tareas termine de poblarse con los índices de los diferentes sub-arreglos que quedan por procesar, se genera lo que se conoce como un *pool* de *threads*, conformado por una cantidad de *threads* igual al número de procesadores disponibles en el equipo. Al trabajar con este *pool* lo que se hará es que, cada vez que un *thread* acceso al espacio de memoria en donde se almacena la pila de tareas, revisará si en dicha pila existe una tarea por realizar, extraerá dicha tarea, la realizará y volverá a revisar la pila, luego el *thread* hará esto hasta que ya no vea tareas pendientes, tras lo cual se eliminará de la memoria. La idea de trabajar con un *pool* de *threads* en lugar de crearlos de forma independiente es que, agiliza el procesamiento cuando se trabaja sobre una cantidad desconocida de tareas para el usuario, y como no se sabe exactamente cuántos subconjuntos se crearán al analizar el arreglo de datos, el uso de esta metodología resulta ideal para este caso.

Luego, cada uno de los *threads* extrae una tarea de nuestra pila, la cual corresponde a un par de índices únicos que especifican en donde comienza y termina el sub-arreglo por ordenar, tras lo cual procede a realizar su propia iteración del algoritmo *Quicksort* sobre dicho sub-arreglo. Una vez concluida su tarea, cada *thread* revisa si quedan tareas pendiente en la lista, de ser así extrae una nueva tarea y la procesa, caso contrario el *thread* se elimina.

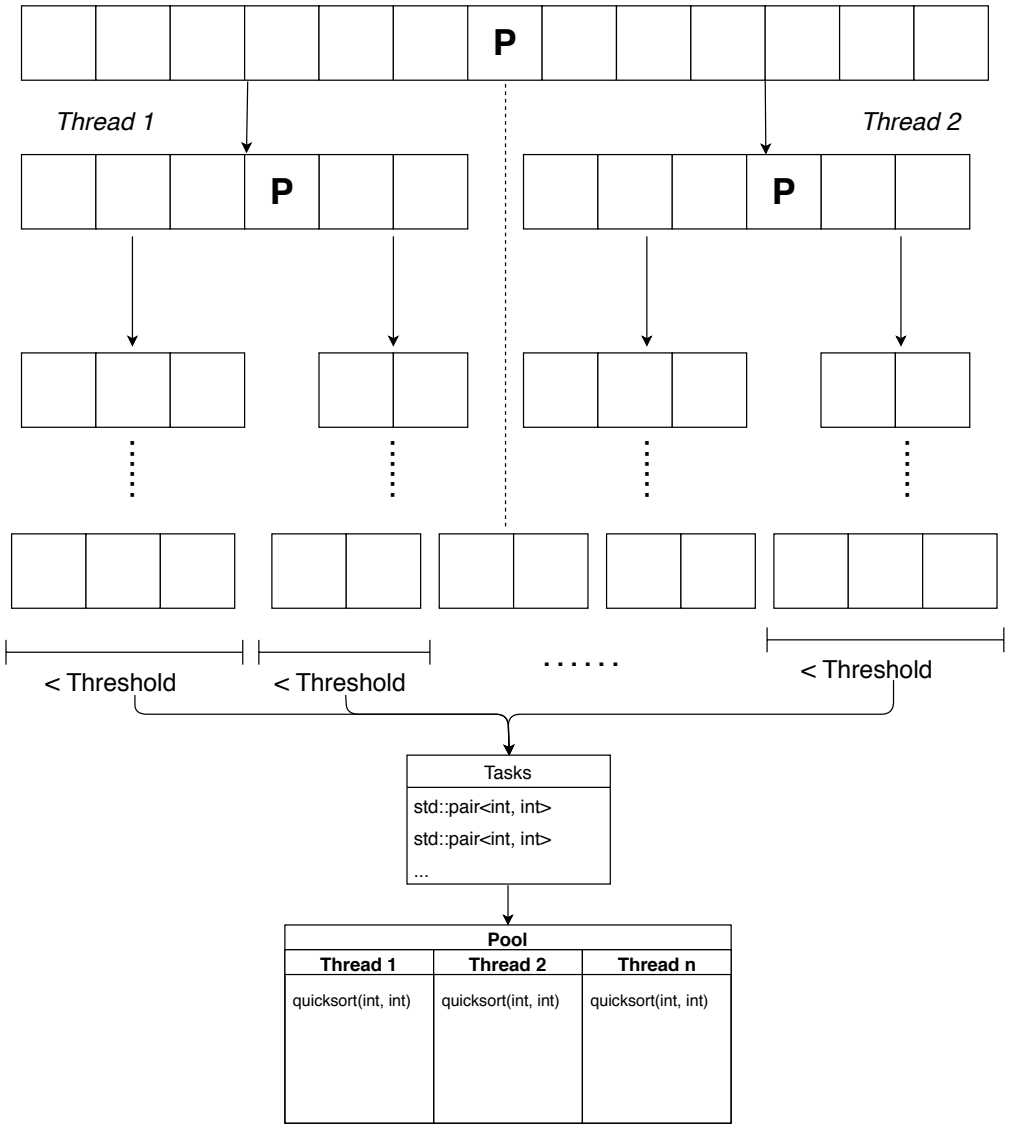



Figura 4: Diagrama de Funcionamiento Algoritmo *Quicksort* Paralelo.
Fuente: Elaboración Propia.

Cabe destacar que las limitaciones de *hardware* entorpecieron la lógica de este proceso hasta cierto punto, la idea inicial era que los *threads* del *pool* fueran llenando de forma dinámica la pila de tareas, evitando el tener que hacer el ordenamiento inicial que se terminó implementando, pero ciertos problemas de memoria impidieron que esto se realizará correctamente, y se optó por no considerarlos en el desarrollo del proyecto. Sin embargo sería interesante poder realizar esta asignación dinámica de las tareas, ya que sería un aporte no menor al estudio de la implementación en paralelo de algoritmos de ordenamiento.

## 3.4. VERSIÓN PARALELA EN GPU

La idea detrás de la paralelización a nivel de GPU, como ya se a ido discutiendo, es aprovechar el hecho de que las GPUs modernas están compuestas de por cientos (en algunos casos, miles) de procesadores, los cuales, a pesar de no ser tan poderosos como los procesadores de una CPU, son capaces de completar operaciones matemáticas relativamente simples, haciendo uso de código *Kernel*. Pese a que sólo son capaces de procesar las operaciones matemáticas más básicas, la gran densidad de procesadores permiten obtener un nivel de paralelismo mucho más alto que con la CPU, distribuyendo de mejor manera las tareas, reduciendo significativamente la carga de cada uno de estos procesadores, lo cual a su vez implica un menor tiempo de cálculo en general.

La versión paralela del sistema, se diseñó utilizando la plataforma CUDA de Nvidia, debido principalmente a que, al tener acceso sólo a una tarjeta gráfica de este fabricante, se optó por aplicar el sistema que pueda utilizar de mejor manera dicha tarjeta, a modo de adquirir el mejor desempeño posible. También se evaluó utilizar *OpenCL*, debido a que es una plataforma independiente de la tarjeta, lo cual le otorga mayor flexibilidad al sistema si es que se desea implementar en equipos que no posean una tarjeta gráfica Nvidia. Sin embargo, durante la investigación se determinó que el desempeño de *OpenCL* tiende a ser menor cuando se trabaja sobre una tarjeta Nvidia que si se utilizara CUDA, sin embargo ambas plataformas son bastante similares en términos de sintaxis y ambas son capaces de realizar las misma operaciones, por lo que si a futuro se desea implementar este sistema para funcionar con *OpenCL*, las modificaciones necesarias para este cambio serían mínimas, y teóricamente no debiera haber mayor problema al intentar migrar la plataforma.

### 3.4.1. SIMULACIÓN DE MONTE CARLO EN GPU

Al igual que con la versión paralela en CPU, lo primero a paralelizar es la simulación de Monte Carlo, debido a que el ordenamiento depende directamente del resultado de ésta, haciendo que sea imposible realizarlo sin antes haber generado los datos simulados a analizar.

Utilizando como base el trabajo que se realizó al paralelizar este calculo en CPU, se decidió tomar un enfoque similar: para cada una de las direcciones, en cada una de las operaciones analizadas, una vez que se generan los conjuntos de valores pseudoaleatorios, (en donde cada uno simula una serie de datos de mercado) estos conjuntos se dividen equitativamente entre la cantidad de procesadores disponibles en la GPU, de esta forma distribuimos la carga de simular las valorizaciones en lugar de dejar todo el trabajo a un solo procesador.

La mayor diferencia que existe entre la versión paralela en CPU y la paralela en GPU está en la lógica detrás de la división de los múltiples conjuntos, ya que en la primera, el conjunto simplemente se divide por la cantidad de procesadores disponibles, y cada procesador se encargaba de trabajar sobre cada uno de los subconjuntos. En contraste, al paralelizar procesos en GPU los datos no se distribuyen de la misma manera, de hecho estos se orga-

nizan en función de bloques llamados ***Blocks***, en donde cada uno de estos posee una cierta cantidad de *threads* ejecutándose simultáneamente, cantidad que dependerá de la arquitectura del *hardware* que se utilice. Estos *blocks* tienen una serie de propiedades que resultan sumamente útiles al momento de procesar data sobre ellos, como la capacidad de poder interpretarse en hasta 3 dimensiones, lo cual se utiliza bastante para el procesamiento de imágenes, ya que si por ejemplo, se busca procesar una imagen 2D, se construyen los bloques en dos dimensiones, así cada *thread* (i,j) se encargada de procesar un *pixel* distinto de dicha imagen. Otra característica interesante, es el hecho de que todos los *threads* dentro de un mismo *block* tienen acceso a un espacio de memoria compartida, lo cual reduce significativamente el tiempo de conmutación de tareas y permite ejecutar operaciones mucho más complejas que las permitidas al trabajar con paralelización a nivel de CPU. Cabe mencionar que cuando se trabaja con un conjunto de *blocks*, esto se conoce como un ***Grid***.

Al trabajar con *Grid*, lo más intuitivo es distribuir los datos equitativamente, y la forma más sencilla de hacer esto es entregarle un dato a cada uno de los *threads*, lo cual resulta mucho más fácil de hacer que en la paralelización con CPU debido a la gran cantidad de *threads* que pueden existir en memoria en cualquier momento. Para hacer esto, es necesario que dichos datos puedan ser identificados por su *thread* correspondiente en cada uno de los *blocks* creados, y la forma en que CUDA logra esto es mediante el uso de índices únicos para cada uno de los *threads* y *blocks* que crea, en cada una de las dimensiones que se utilizan. En la imagen a continuación se puede apreciar como los índices se distribuyen en este caso, trabajando sobre un arreglo en 1D, con 8 *threads*/*block*, sobre un total de 4 *blocks*. Como se puede ver, CUDA utiliza las variables internas **threadIdx.x** y **blockIdx.x** para realizar la indexación del arreglo, en las cuales el uso de la letra x indica que se está observando desde dicha dimensión.

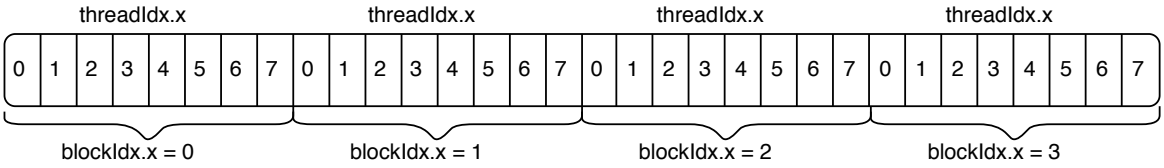



Figura 5: Ejemplo de indexación en arreglo 1D con CUDA.
Fuente: CUDA C/C++ Basics, NVIDIA Corporation, 2011 [Zeller, 2011].

En el caso presentado en la Figura 5, si se desea acceder al elemento 21 del arreglo, este se encontraría en el *block* 2, *thread* 5, por lo tanto, al utilizar estos dos valores, y conociendo la cantidad de *threads* que existen dentro de cada uno de los *blocks*, podemos fácilmente determinar que estos tres valores entregan la posición 21 del arreglo. Con esta información es que se puede recorrer un arreglo en CUDA asumiendo, como ya se dijo, que cada elemento se le asigna a un *thread* diferente.

Para el caso de este proyecto, ocurrió la problemática de que la densidad de los datos a analizar era demasiada como para poder asignar un dato distinto a cada *thread*, por lo tan-

to se tuvo que ampliar la lógica anterior mediante lo que se conoce como un *Grid-Stride Loop* [Harris, 2013]. Este tipo de *loop* lo que hace es asignar la mayor cantidad posible de *threads*, de tal forma que sean divisible por el tamaño del conjunto de datos. De esta forma a cada *thread* se le hace procesar múltiples datos al saltar, en cada iteración, una cantidad de espacios igual al tamaño del *Grid* más el índice que tiene asignado dentro del mismo. Por ejemplo, si se tiene un total de 1280 *threads* en el *grid*, el *thread* cero se encarga de procesar los elementos 0, 1280, 2560, etc. del arreglo, de esta forma se mantiene la lógica de indexación que CUDA emplea por defecto.

### 3.4.2. ALGORITMO DE ORDENAMIENTO EN GPU

Traducir el algoritmo de ordenamiento *Quicksort* a GPU resultó una tarea mucho más compleja de lo que originalmente se había presupuestado, debido principalmente a que los procesadores que conforman las GPUs están diseñados principalmente para completar operaciones matemáticas relativamente simples, por lo cual, ordenarlos a realizar operaciones como comparación entre valores para determinar cuál es el más grande, y sobre todo cuando se le intenta ordenar el manipular un arreglo de datos con operaciones que van más allá de sólo ingresar un valor en una cierta posición dada, el comportamiento de la tarjeta se vuelve más errático, y los resultados terminan siendo sub-óptimos en la mayoría de los casos.

Sin embargo, se realizó el intento de implementar esta lógica en CUDA, ya que hoy en día existen ejemplos de implementaciones de algoritmos de ordenamiento sobre GPU, y de hecho, entre los múltiples ejemplos que provee NVidia se encontraron diferentes tipos de algoritmos de ordenamiento, no sólo *Quicksort*, por lo que se decidió utilizar estas implementaciones como base, e intentar adaptarlas al sistema.

El primer intento que se realizó, fue tomar la implementación del algoritmo *Quicksort* diseñada por NVidia como parte de los ejemplos de CUDA y adaptarla directamente a la versión paralela en GPU que se estaba desarrollando. Sin embargo, este código sufría del mismo problema que las implementaciones del algoritmo paralelizado para CPU que se encontraron cuando se diseñó esa versión del sistema, en las cuales el algoritmo está diseñado para procesar conjuntos de datos con tamaños relativamente reducidos, por lo que ir asignando dinámicamente *threads* en cada división del arreglo no resulta una tarea tan demandante, pero como en el caso a analizar se está trabajando con millones de datos, le es imposible al *hardware* crear tantos *threads* simultáneamente, resultando en errores de memoria que impiden que el proceso se completara satisfactoriamente, el cual es exactamente el mismo problema que surgió durante la versión paralela en CPU del sistema.

Tras esto se intentó adaptar las demás versiones de ejemplo que suministró NVidia dentro de su paquete de desarrollo con CUDA, los cuales implementan los algoritmos *Mergesort* y *Radixsort*. Sin embargo, estos también presentaron una serie de problemas al intentar ser paralelizados en GPU. El primero de estos presentó los mismos problemas que *Quicksort*, ya que dicho algoritmo se basa en dividir el conjunto espacio por espacio para luego ir generan-

do subconjuntos, en los cuales el primer valor es menor a todos los que están a su derecha, de esta forma ir ordenando el arreglo de menor a mayor [Cormen *et al.*, 2009]. Sin embargo, esto significa tener que crear dinámicamente, una cantidad no menor de *threads* que se encarguen de ir comparando y armando estos subconjuntos, lo cual, al trabajar sobre millones de datos como en este caso, el *hardware* es incapaz de crear y gestionar tantos *threads* simultáneamente generando una serie de errores en memoria.

Por otro lado, *Radixsort* es mucho más peculiar, y se basa en ir descomponiendo cada valor numérico en sus diferentes dígitos, y analizar dígito a dígito los diferentes valores a modo de ir ordenando el arreglo sin tener que analizar el número por completo [Cormen *et al.*, 2009]. Sin embargo, esta lógica es incapaz de acomodarse a los datos sobre los que se trabaja por dos motivos: primero, *Radixsort* exige que todos los valores tengan la misma cantidad de dígitos para poder funcionar, y a pesar de que existen formas de lidiar con esto (agregar 0's al principio de cada número si es que este es más pequeño que el más grande del conjunto), al trabajar sobre tantos valores al mismo tiempo y sin saber cual de estos tendrá la mayor cantidad de dígitos de antemano (debido a la aleatoriedad de los mismos), este paso introduce una mayor complejidad al sistema e inevitablemente terminará ralentizando cada paso de ordenamiento de forma innecesario. Como si esto no fuera poco, *Radixsort* está diseñado para trabajar sobre números con una cantidad de dígitos relativamente baja, pero en este caso se está trabajando sobre valores que pueden ir desde 6 a 9 dígitos en algunos casos, por lo que este algoritmo sencillamente no es recomendado de utilizar en esta ocasión.

Por último, similar a lo que se realizó en CPU, se intentó implementar un sistema de *threadpool* a modo de evitar tener que crear tantos *threads* al momento de resolver el *Quicksort*, con el fin de poder procesar correctamente dicho algoritmo de ordenamiento. Sin embargo, la forma en que CUDA distribuye sus tareas y más precisamente, el cómo asigna una porción de memoria por *block*, haciendo que sólo los *threads* dentro del mismo sean capaces de compartir memoria entre sí, impidió que esta lógica se tradujera tan fácilmente a GPU, por lo que se optó por abandonar este enfoque y seguir avanzando con el proyecto. No obstante, es muy posible que esta lógica de *threadpool* sea capaz de ser utilizada para implementar este tipo de algoritmo de ordenamiento sobre una densidad tan grande de datos y se cree que es una tarea futura muy interesante a ser resuelta.

## 3.5. VERSIÓN PARALELA HÍBRIDA

Hasta el momento los cálculos se realizan de forma aislada, en CPU o en GPU, buscando en cada una de estas versiones reducir el tiempo de procesamiento en comparación con la versión secuencial del mismo sistema. Sin embargo, a modo de intentar alcanzar un mejor rendimiento y utilizar de mejor manera el dispositivo de *hardware* del cual se dispone, es que se busca romper este aislamiento y hacer que ambas lógicas de cómputo se ejecuten al mismo tiempo sobre una misma máquina.

La última versión que compone al sistema busca hacer uso de los puntos más fuertes de las

versiones anteriores simultáneamente, mediante una paralelización híbrida del proceso, esto es, procesar al mismo tiempo, los cálculos involucrados en el proceso de valorización de derivados financieros, tanto en CPU como en GPU, a modo de aprovechar de mejor manera todos los recursos de *hardware* disponibles en la máquina en la cual se realice el cálculo, además de intentar agilizar aún más el mismo, bajo la hipótesis de que el utilizar simultáneamente todos los dispositivos disponibles mediante la mejor lógica disponible (paralelización del cálculo), éste se completará en menor tiempo que si estas lógicas se aplicaran de forma aislada.

A modo de poder ejecutar el código de forma simultánea en CPU y en GPU, fue necesario identificar qué partes del mismo se deberían ejecutar en los diferentes dispositivos, similar a la decisión que se tomó en la versión secuencial del código, para decidir qué secciones paralelizar. Debido al hecho de que mediante CUDA sólo se logró paralelizar la simulación de Monte Carlo, la elección se ve limitada en este sentido, obligando a que para la versión híbrida de este proyecto, la GPU se tenga que encargar de simular los escenarios futuros de la cartera de negocios, mientras que la CPU se encargue de ordenar estos resultados a modo de poder determinar el *Value-at-Risk* de la cartera estudiada.

Es importante mencionar que, hasta este punto, ambas secciones del sistema se han ejecutado de forma "secuencial", en otras palabras, a pesar de que cada una de ellas se ha paralelizado, se ha mantenido la estructura de tener que completar todas las simulaciones de la cartera antes de pasar a ordenarlas, cosa que se comentó en las secciones anteriores, pero que para facilitar la lógica de programación se decidió mantener. Sin embargo, para esta última versión, a modo de alcanzar este paralelismo híbrido que tanto se busca, se decidió entrelazar estos pasos mediante lo que se conoce como **Arquitectura *Pipeline***, la cual se explica a continuación:

### 3.5.1. ARQUITECTURA *PIPELINE*

El principal desafío del paralelismo híbrido, es encontrar una forma orgánica en que las diferentes unidades de *hardware* logren trabajar simultáneamente sobre diferentes secciones del proyecto, sin interrumpirse entre ellas y distribuyendo de forma apropiada la carga de trabajo. A modo de lograr esto, se estudió la arquitectura *Pipeline* [Ramamoorthy y Li, 1977], modelo de paralelismo antiguo, bastante popular en el década de los 60 y 70, el cual busca paralelizar instrucciones que se ejecutan sobre un solo procesador, manteniendo ocupada a cada parte de dicho procesador. Esto lo logra dividiendo cada proceso en una serie de instrucciones que se deben completar, en donde cada instrucción requiere los datos procesados por la anterior antes de poder comenzar. Luego, se separa el procesador en múltiples módulos, cada uno encargado de procesar únicamente un tipo de instrucción y con las órdenes de entregar los resultados de su instrucción al siguiente módulo una vez completado, antes de continuar con la ejecución de su siguiente instrucción.

La forma más sencilla de verlo es mediante el diagrama creado por Ramamoorthy y Li, el

cual se encuentra en su escrito acerca sobre *pipeline* del año 1977. Imagine un proceso que puede ser dividir en cuatro instrucciones independientes, las cuales son: extracción de la instrucción (IF), decodificación de la instrucción (ID), extracción de los operadores relevantes a la instrucción (OF) y ejecución de la misma (EXEC); y que se desea construir un proceso que esté constantemente ejecutando múltiples de estas ejecuciones. La forma en que se puede aplicar la arquitectura *pipeline* en el mismo, es mediante la división de dichas instrucciones, la creación de módulos independientes que se encarguen de cada una de estas y la ejecución de las mismas. Como se puede apreciar en el tercer diagrama, al momento en que el módulo EXEC esté ejecutando su primera instrucción, el módulo OF está extrayendo los operadores necesarios para la segunda instrucción, el módulo ID decodifica la tercera instrucción y el módulo IF extrae la cuarta, de esta forma cada módulo se encuentra ocupado en algo distinto y se logra mayor agilidad al momento de ejecutar dicho programa, en comparación a ejecutar cada una de estas instrucciones de forma secuencial.

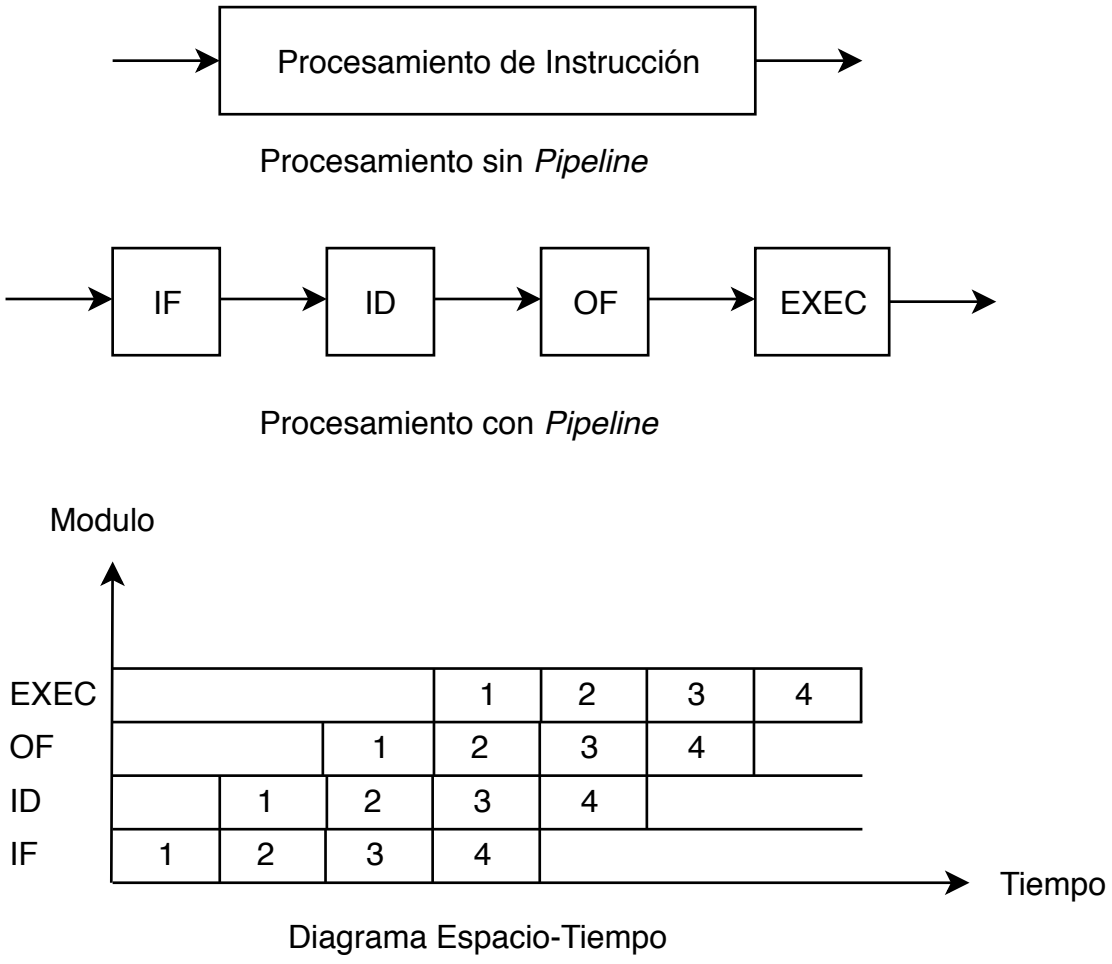



Figura 6: Diagrama de arquitectura *Pipeline*.
Fuente: Pipeline Architecture, C. V. Ramamoorty, 1977 [Ramamoorthy y Li, 1977].

La popularización de los procesadores *multi core* ha hecho que este tipo de arquitectura se

deje de utilizar. Sin embargo, calza perfectamente con lo que se busca lograr en este proyecto, ya que, utilizando el mismo lenguaje, si consideramos las diferentes direcciones de cada una de las operaciones financieras como una instrucción diferente, los pasos de simulación y ordenamiento como dos operaciones independientes, en donde el ordenamiento necesita de los resultados de la simulación para poder procesarse, y si consideramos a la CPU y a la GPU como dos módulos independientes, en donde el primero se encarga de ordenar los datos simulados por el segundo, se llega a la conclusión de que al emplear *pipeline* sobre este proceso en conjunto con la paralelización de los cálculos que ocurren en las diferentes unidades de *hardware*, este último se logra ocupar de mejor manera, entregándonos un grado de paralelización mucho mayor que el que se logra en las versiones anteriores del sistema.

Luego, lo que se busca al diseñar este sistema es que, mientras la CPU se preocupa de realizar el ordenamiento de los datos de forma paralela, la GPU estará simulando los datos de la siguiente operación en la cartera de negocios, a modo de tenerlos listos en cuanto se complete el ordenamiento lo cual, en teoría, debería agilizar aún más el proceso de valorización y cálculo de *Value-at-Risk*, y se deberían de poder observar mayores reducciones en el tiempo de procesamiento que las obtenidas mediante la paralelización del mismo cálculo utilizando solo CPU o GPU.

### 3.5.2. DISTRIBUCIÓN HÍBRIDA DE TAREAS

Una vez identificada la arquitectura a utilizar, es necesario traducir esta lógica al sistema existente, y hacer que se ejecute dentro de las limitaciones que existen en el lenguaje sobre el cual se implementa. A modo de lograr esto, y aprovechando los conocimientos adquiridos en el desarrollo de este proyecto, se aplico lo aprendido con la paralelización del algoritmo *Quicksort* en CPU, y siguiendo una lógica similar, se decidió crear un pseudo *pool* de *threads*, con el cual se puede gestionar a cada una de las unidades de *hardware* que se encargará de procesar los datos.

La forma en que funciona es mediante dos *threads*: uno encargado de la simulación (el cual se identifica como T1), el cual, como se sabe, se ejecuta en paralelo sobre la GPU, y el otro encargado del ordenamiento de los datos simulados (identificado como T2), el cual a su vez también se realiza de forma paralela y cuyo responsable es la CPU. Además, ambos *threads* tienen asignados una pila de tareas diferente, en la cual se especifica las diferentes operaciones que deben ser cumplidas por cada uno de ellos. Para el caso de T1, la pila de tareas se carga con las diferentes operaciones financieras que se analizan, ya que en base a las propiedades de cada una de ellas es que se simulan los escenarios de valorización futuros. A continuación, T1 envía los datos de la simulación a la GPU, a modo de que esta se encargue de procesarlos y, últimamente, devolver los valores simulados para dicha operación. Una vez con los valores, T1 se encarga de almacenar los mismos dentro de una segunda pila de tareas, la cual T2 se encarga de monitorear y mientras esta no se encuentre vacía, extrae los datos simulados de esta y se encarga de crear nuevos *threads* en la CPU, los cuales aplican el

algoritmo *Quicksort* paralelo que se explicó en la sección 3.3.2. para poder ordenar los datos y determinar el *Value-at-Risk* de la operación estudiada.

Esta lógica, a pesar de ser efectiva, presenta un desperfecto clave que se debe tener en cuenta al momento de implementarla, y es el hecho de que, a modo de poder coordinar los diferentes procesos paralelos en las diferentes piezas de *hardware* disponibles, es necesario utilizar *threads* de la CPU para completar esta tarea, lo cual implica que, para lo que respecta al algoritmo de ordenamiento, la CPU dispondrá de dos *threads* menos para realizar este proceso, efectivamente reduciendo su velocidad de procesamiento en comparación con su contraparte paralela exclusivamente en GPU. Sin embargo, debido a que la única tarea que estos *threads* deben llevar a cabo es extraer información desde una pila de tareas y asignarla, ya sea a CPU o a GPU, estos se utilizaran relativamente poco, por lo que, durante su tiempo muerto, se pueden fácilmente asignar a los procesamientos que se lleven a cabo de CPU, con lo cual se espera reducir el impacto en términos de desempeño de esta lógica.

Por último, no se puede pasar por alto las diferencias en termino de capacidad de ordenamiento individual de cada una de estas piezas de *hardware*, punto que resulta mucho más crítico al hacer que trabajen en paralelo. En otras palabras, se sabe de ante mano que el tiempo total de procesamiento va a estar determinado por el procesador más lento, cosa que se explorara más a fondo durante las secciones posteriores, pero vale la pena destacar que, entre más similares sean las piezas de *hardware* en términos de capacidad de procesamiento, se obtendrán en promedio resultados mucho mejores y consistentes que con procesadores con capacidades disparejas entre sí.

# CAPÍTULO 4
# VALIDACIÓN DE LA SOLUCIÓN

A modo de validar la solución propuesta, junto con implementar las diferentes versiones del sistema que se diseñaron y las cuales se explicaron en la sección anterior de este documento, se decidió llevar a cabo una serie de experimentos sobre cada una de éstas, haciendo uso de diversos equipos, cada uno con piezas de *hardware* muy distintas, poniendo especial énfasis en medir el tiempo de procesamiento de cada una de estas versiones, con la finalidad de comparar su desempeño, no sólo frente a una variedad de entradas por parte del usuario, sino que además frente a diferentes elementos de *hardware*, con el fin de poder tener una mejor idea de cómo estos últimos influyen en el desempeño de las soluciones que se proponen.

En este capítulo, y antes de hablar de los experimentos como tales, se comenzará revisando que tan precisos fueron los cálculos obtenidos del *Value-at-Risk*, lo cual, a pesar de no ser de gran impacto para lo que busca el proyecto, sigue siendo necesario de tener en cuenta. Posteriormente se realizara la presentación de las especificaciones técnicas de cada una de las máquinas que se utilizó durante el desarrollo de los experimentos, se definirá precisamente en que consta cada uno de ellos, se presentarán los diferentes resultados obtenidos, y se extrapola en base a la información recabada, el desempeño promedio del sistema frente a diferentes componentes de *hardware*, a modo de predecir empíricamente como este software se desempeñaría de ejecutarse en diversas computadoras.

## 4.1. ANÁLISIS DE RESULTADOS *VALUE-AT-RISK*

Para llevar a cabo este análisis se ejecuto la versión secuencial del sistema sobre 1.000.000 de simulaciones un total de 10 veces. Antes de comenzar este conjunto de simulaciones se eliminaron los últimos 10 registros de datos de las carteras de derivados, y estos se fueron re integrando al final de cada iteración, a modo de poder comparar el valor predecido con el valor real de la cartera.

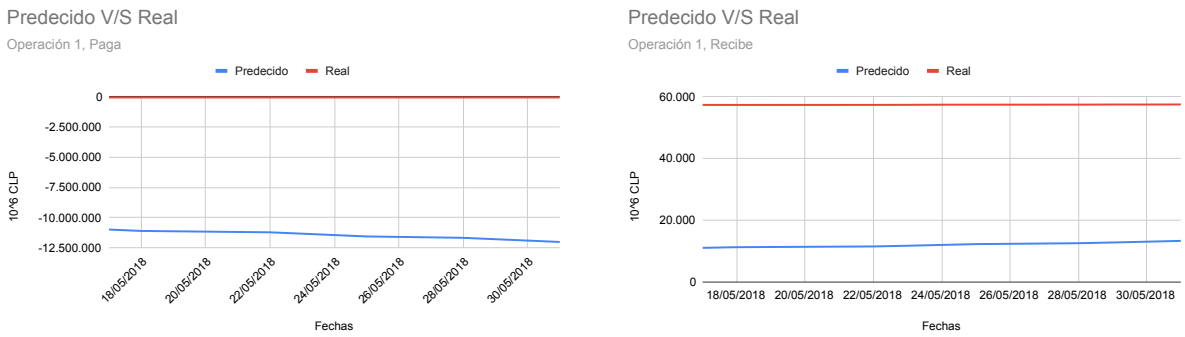



Figura 7: Resultados Operación 1, Predecido v/s Real.
Fuente: Elaboración Propia.

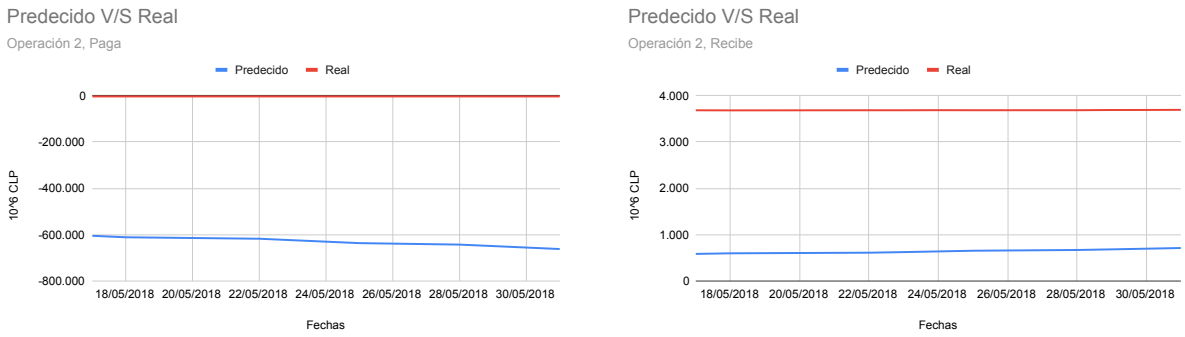



Figura 8: Resultados Operación 2, Predecido v/s Real.
Fuente: Elaboración Propia.

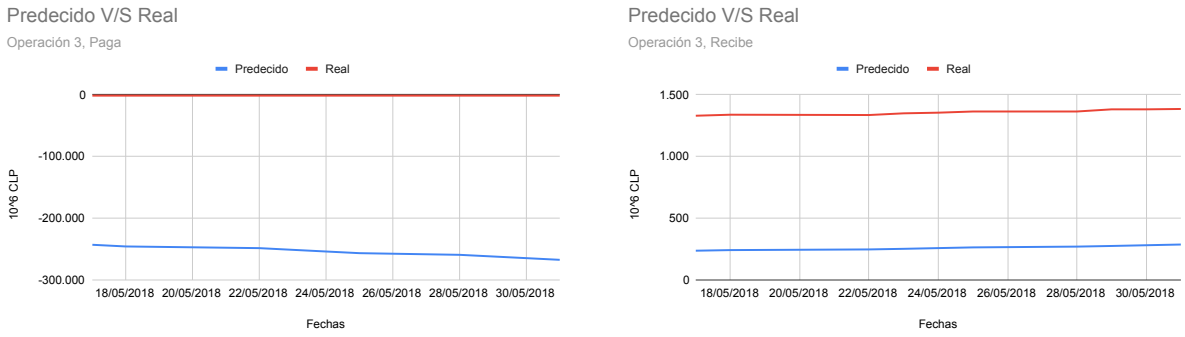



Figura 9: Resultados Operación 3, Predecido v/s Real.
Fuente: Elaboración Propia.

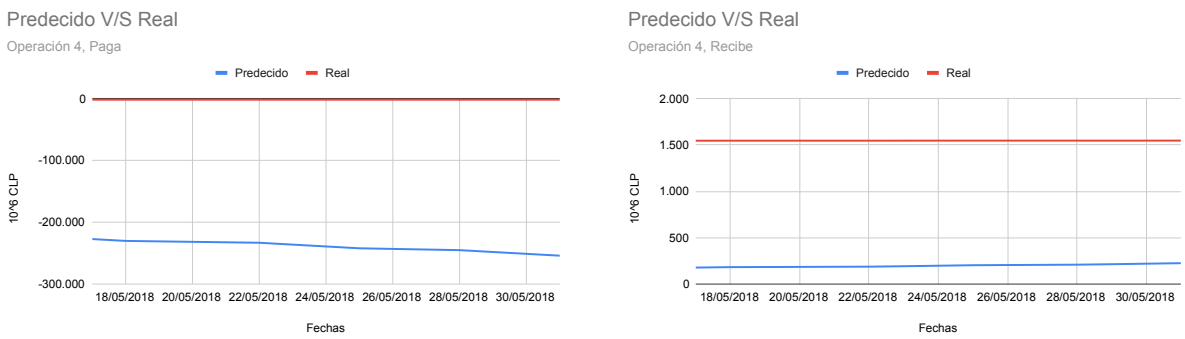



Figura 10: Resultados Operación 4, Predecido v/s Real.
Fuente: Elaboración Propia.

Como se puede observar, los resultados predecidos están siempre por debajo de los resultados reales, lo cual es exactamente lo que se busca lograr al implementar *Value-at-Risk*, prediciendo el peor resultado posible. Sin embargo, los escenarios simulados tienden a ser demasiado conservadores, sobre todo en las direcciones de Paga, lo cual era de esperarse considerando que el método utilizado para el este calculo es bastante simplista, y cuando se aplica en carteras con relativamente poca antigüedad no suele entregar resultados con alta precisión, pero para lo que respecta al objetivo del proyecto, el cual busca asemejarse a una condición real y comparar desempeños en términos de capacidad de procesamiento, este tipo de resultado es mas que aceptable, ya que lo único que se esperaba del mismo es que no cometiera errores de predecir resultados erróneos, esto es, por encima de la curva de valores reales.

## 4.2. DEFINICIÓN DE LOS EXPERIMENTOS

A manera de probar no sólo las capacidad de los diferentes sistemas diseñados, sino que también como su desempeño evoluciona frente a diversos valores de entrada, es que se decidió realizar un conjunto de cinco experimentos, los cuales constaran de ejecutar cinco veces seguidas cada una de las versiones del sistema, a modo de eliminar posibles *outliers* al considerar un comportamiento promedio, y en donde cada uno de los experimentos será ejecutado con un valor de entrada diferente para cada uno.

Como ya se explicó en la sección anterior, cada una de las versiones del sistema recibe como valor de entrada la cantidad de simulaciones que se deberán completar durante su ejecución, lo cual, efectivamente significa la cantidad de conjuntos de información de mercado generada de forma pseudo aleatoria que se simulan. Este valor incide directamente en ambas secciones que se decidieron paralelizar, por lo tanto, a medida que este aumente se van a poder observar de mejor manera como la complejidad de las diferentes versiones del sistema van influyendo en el tiempo de procesamiento.

Concretamente, la cantidad de simulaciones que se realizaron en cada uno de los experimentos son las siguientes:

- Experimento 1: 100.000 simulaciones
- Experimento 2: 500.000 simulaciones
- Experimento 3: 1.000.000 de simulaciones
- Experimento 4: 2.000.000 de simulaciones
- Experimento 5: 5.000.000 de simulaciones

Cabe destacar que el experimento 5, debido a limitaciones de memoria, no se pudo ejecutar en ambas máquinas, lo cual se explicará con mayor detalle en la subsección que viene a continuación.

## 4.3. EQUIPOS DE PRUEBA

Con el fin de diversificar los resultados obtenidos se decidió hacer uso de dos equipos de prueba diferentes, cada uno con sus propias especificaciones técnicas, y en los cuales se intentó que se llevarán a cabo exactamente los mismos experimentos. Los equipos que se utilizaron fueron: un *laptop* **Lenovo Thinkpad E570**, y un Servidor, el cual le pertenece al **Laboratorio INCA de la UTFSM**. A continuación se muestran las especificaciones de cada uno de estos equipos:

Laptop:

- **CPU**: Intel(R) Core(TM) i7-7500U @2.70GHz, 4 Cores (2 físico, 2 lógicos)
- **GPU**: Nvidia GeForce GTX 950M
- **RAM**: 7.91 GB

Servidor:

- **CPU**: Intel(R) Core(TM) i7-7740X @4.30GHz, 8 Cores (4 físicos, 4 lógicos)
- **GPU**: 2x Nvidia GeForce GTX 1080 Ti
- **RAM**: 64 GB

Como se mencionó en la sección anterior, se intentó ejecutar los 5 experimentos en ambas máquinas, pero por temas de utilización de memoria RAM el último experimento fue imposible de completarse en el *laptop*, por lo cual los resultados del mismo solo se registraron para el servidor. Además, para que existiera mínima interferencia al momento de ejecutar las pruebas, se tomaron las debidas precauciones para asegurar que el sistema era el único proceso exigente ejecutándose en las máquinas en cada momento. Para el *Laptop* esto significó cerrar todo otro proceso que activamente hiciera uso de los recursos del sistema, tales como: navegadores, clientes de correo, entre otros, mientras que para el caso del servidor, como múltiples personas pueden hacer uso del mismo, se ejecutaron las pruebas en un momento de bajo volumen de usuarios, más concretamente el día domingo 10/03/19 desde las 09:00 AM hasta las 11:00 AM.

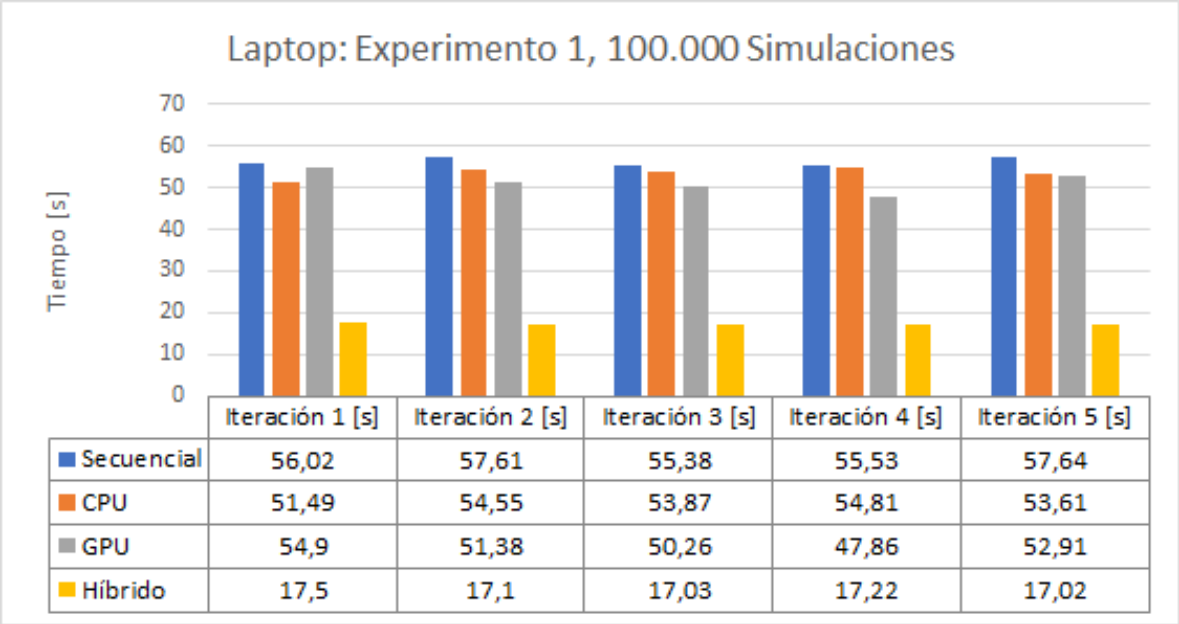
Por último, a pesar de que el servidor cuenta con 2 tarjetas gráficas Nvidia, el sistema no fue capaz de detectar y utilizar el potencial de cálculo de ambas tarjetas simultáneamente, por lo tanto, para lo que respecta al análisis de las pruebas en el servidor, se asumirá que estas se están ejecutando en una sola de las tarjetas gráficas disponibles.

## 4.4. RESULTADOS DE LOS EXPERIMENTOS

A continuación se presenta una tabla con los resultados obtenidos en cada uno de los experimentos. Hay que dejar en claro, que los resultados se presentan en términos de segundos de ejecución, los cuales fueron medidos utilizando el comando time de Linux.

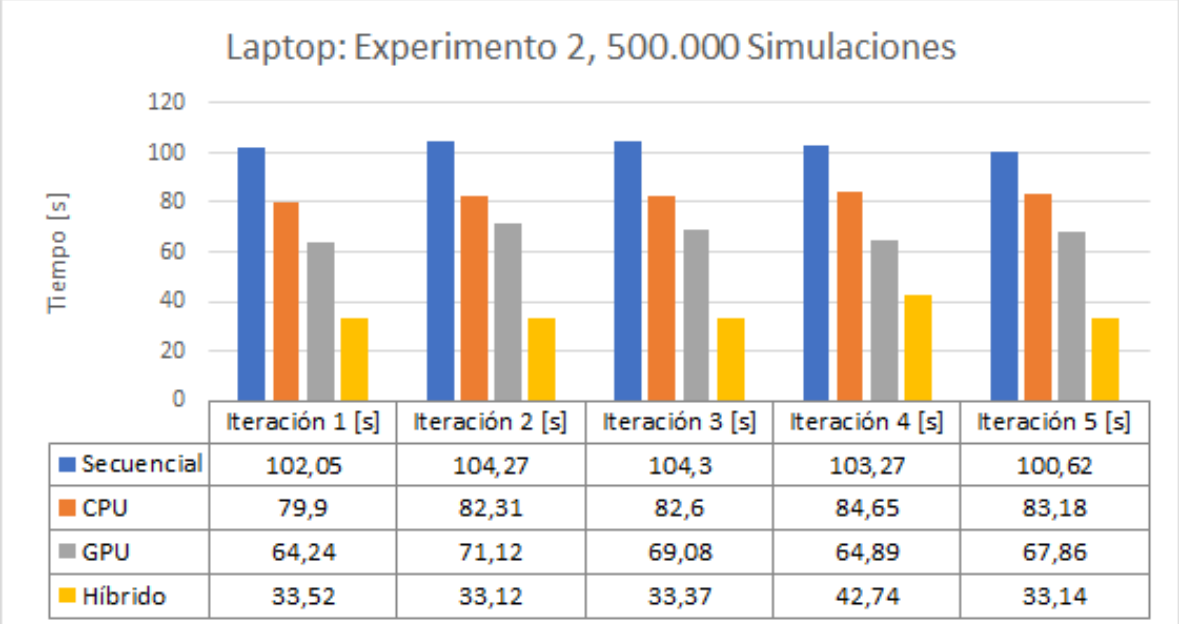
Además, cabe destacar que las tablas mostraran los resultados de tiempo obtenidos en cada una de las iteraciones realizadas, esto es, en cada una de las pruebas que se realizó sobre cada una de las versiones del sistema. Por último, a modo de facilitar la comprensión y análisis de los datos, se decidió presentarlo directamente como gráficos, uno para cada uno de los experimentos realizados, agrupándolos por iteración, e incorporar en cada uno de estos gráficos la tabla con los datos de tiempo exactos que se obtuvieron en cada una de estas pruebas.

### 4.4.1. RESULTADOS LAPTOP



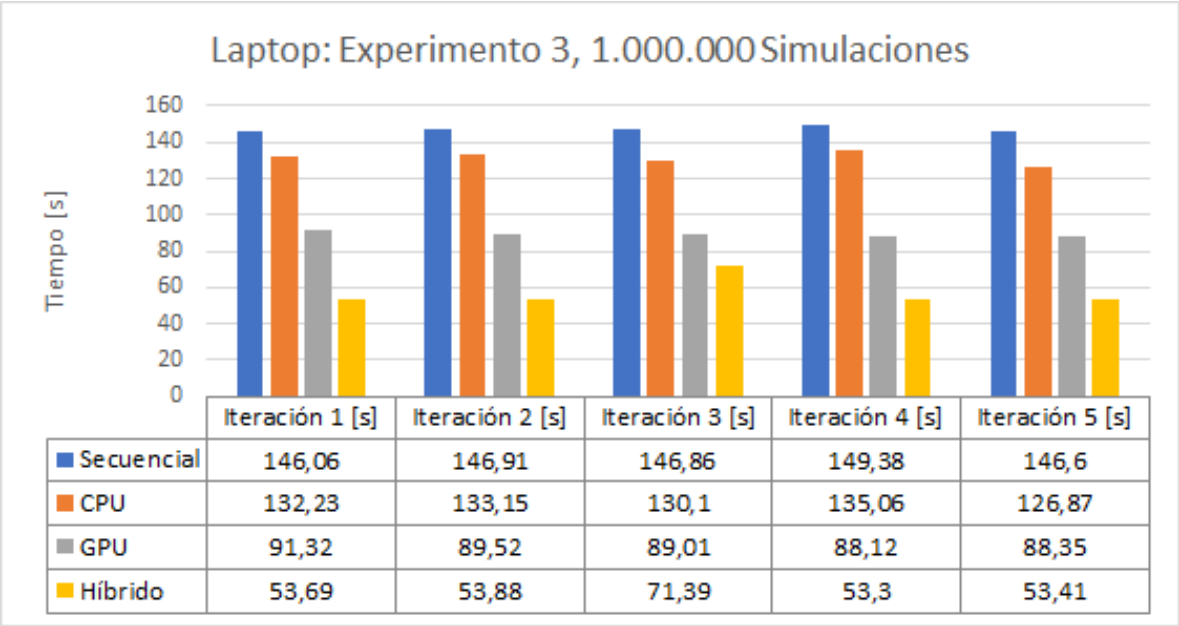

| | Iteración 1 [s] | Iteración 2 [s] | Iteración 3 [s] | Iteración 4 [s] | Iteración 5 [s] |
|---|---|---|---|---|---|
| Secuencial | 56,02 | 57,61 | 55,38 | 55,53 | 57,64 |
| CPU | 51,49 | 54,55 | 53,87 | 54,81 | 53,61 |
| GPU | 54,9 | 51,38 | 50,26 | 47,86 | 52,91 |
| Híbrido | 17,5 | 17,1 | 17,03 | 17,22 | 17,02 |

Figura 11: Resultados Experimento 1 en *Laptop*.
Fuente: Elaboración Propia.



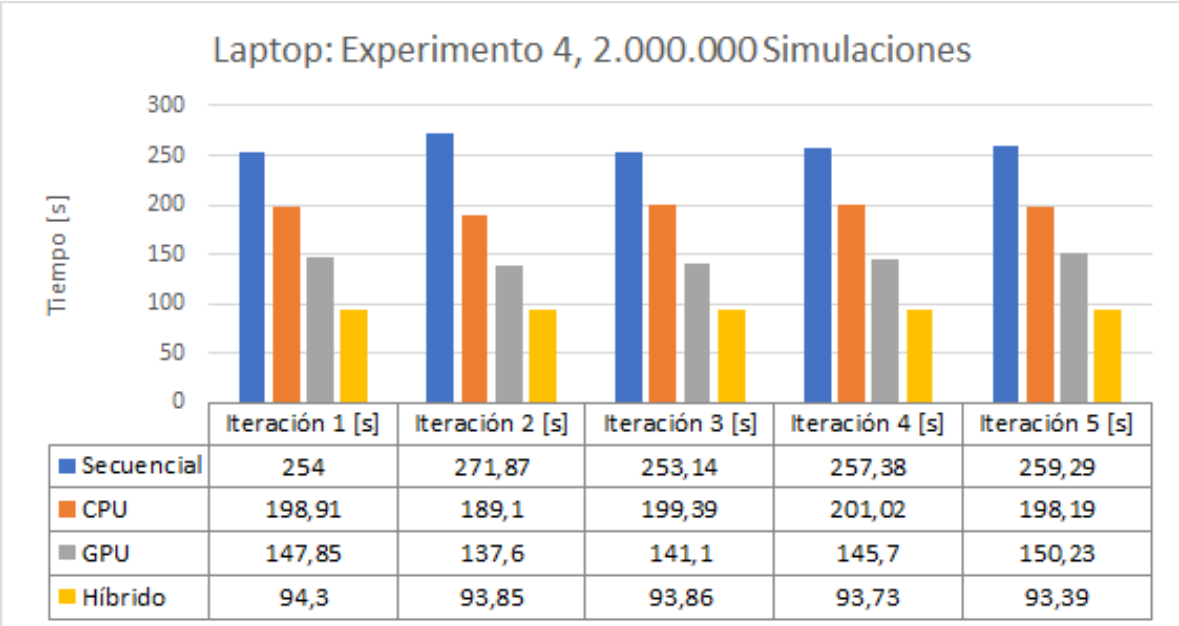

| | Iteración 1 [s] | Iteración 2 [s] | Iteración 3 [s] | Iteración 4 [s] | Iteración 5 [s] |
|---|---|---|---|---|---|
| Secuencial | 102,05 | 104,27 | 104,3 | 103,27 | 100,62 |
| CPU | 79,9 | 82,31 | 82,6 | 84,65 | 83,18 |
| GPU | 64,24 | 71,12 | 69,08 | 64,89 | 67,86 |
| Híbrido | 33,52 | 33,12 | 33,37 | 42,74 | 33,14 |

Figura 12: Resultados Experimento 2 en *Laptop*.
Fuente: Elaboración Propia.




| | Iteración 1 [s] | Iteración 2 [s] | Iteración 3 [s] | Iteración 4 [s] | Iteración 5 [s] |
|---|---|---|---|---|---|
| Secuencial | 146,06 | 146,91 | 146,86 | 149,38 | 146,6 |
| CPU | 132,23 | 133,15 | 130,1 | 135,06 | 126,87 |
| GPU | 91,32 | 89,52 | 89,01 | 88,12 | 88,35 |
| Híbrido | 53,69 | 53,88 | 71,39 | 53,3 | 53,41 |

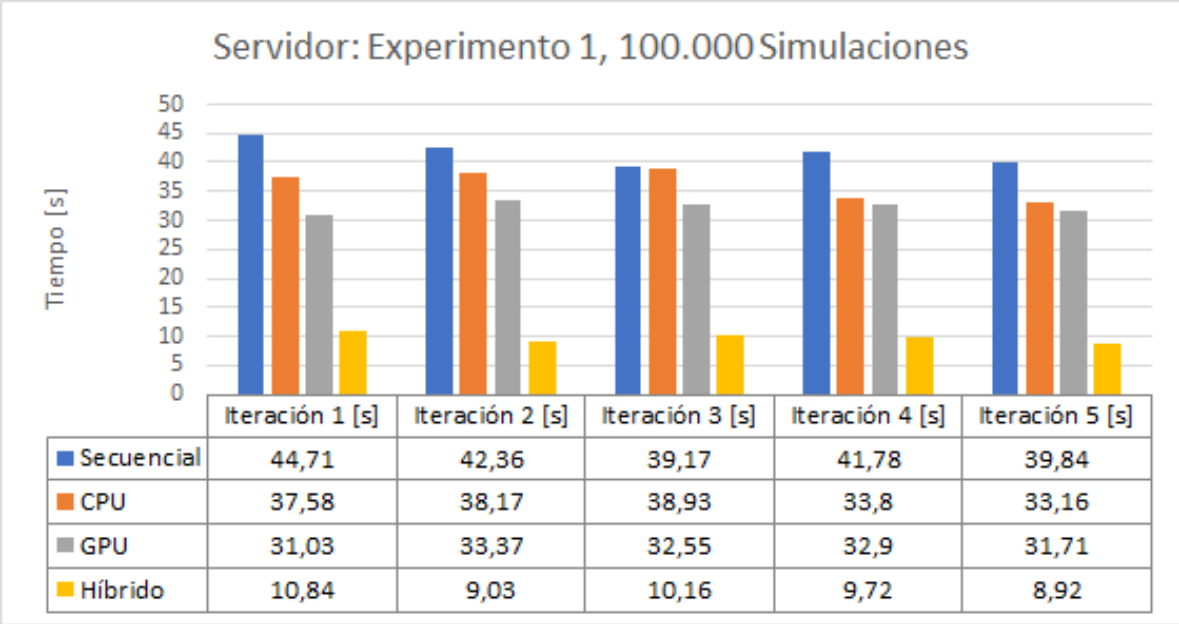
Figura 13: Resultados Experimento 3 en *Laptop*.
Fuente: Elaboración Propia.




| | Iteración 1 [s] | Iteración 2 [s] | Iteración 3 [s] | Iteración 4 [s] | Iteración 5 [s] |
|---|---|---|---|---|---|
| Secuencial | 254 | 271,87 | 253,14 | 257,38 | 259,29 |
| CPU | 198,91 | 189,1 | 199,39 | 201,02 | 198,19 |
| GPU | 147,85 | 137,6 | 141,1 | 145,7 | 150,23 |
| Híbrido | 94,3 | 93,85 | 93,86 | 93,73 | 93,39 |

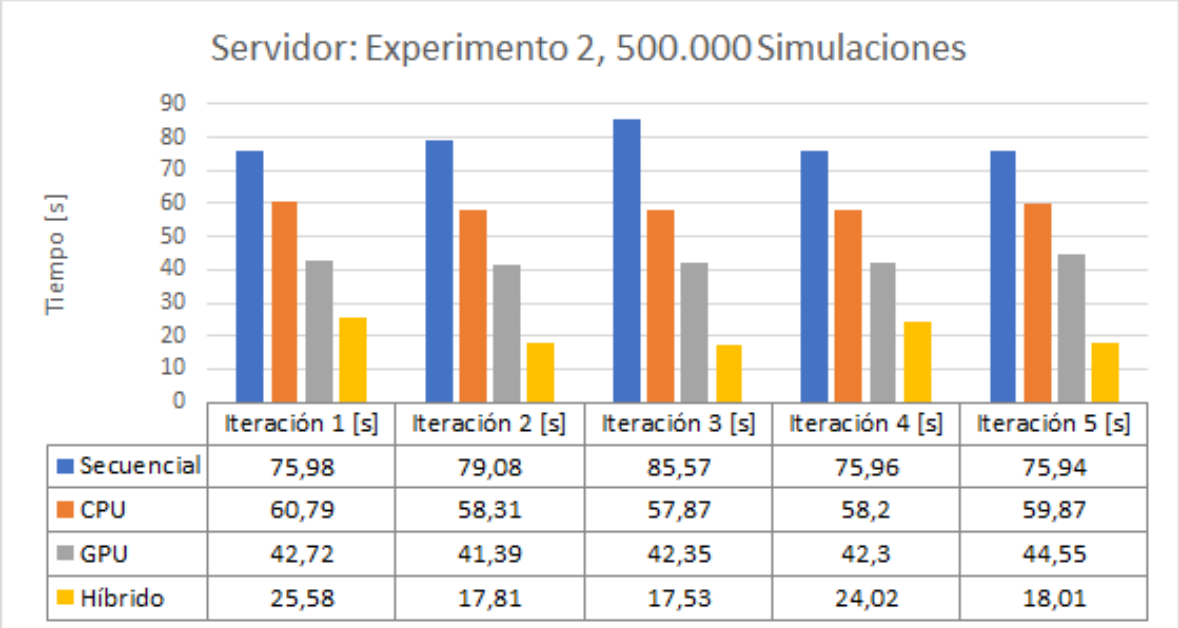
Figura 14: Resultados Experimento 4 en *Laptop*.
Fuente: Elaboración Propia.

### 4.4.2. RESULTADOS SERVIDOR



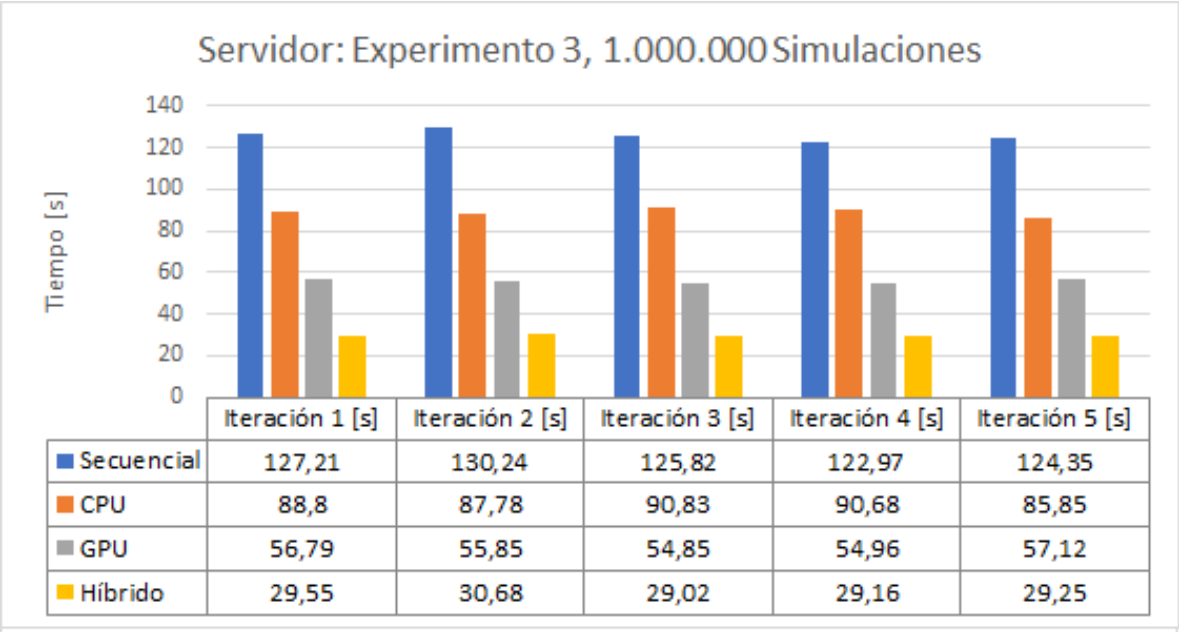

| | Iteración 1 [s] | Iteración 2 [s] | Iteración 3 [s] | Iteración 4 [s] | Iteración 5 [s] |
|---|---|---|---|---|---|
| Secuencial | 44,71 | 42,36 | 39,17 | 41,78 | 39,84 |
| CPU | 37,58 | 38,17 | 38,93 | 33,8 | 33,16 |
| GPU | 31,03 | 33,37 | 32,55 | 32,9 | 31,71 |
| Híbrido | 10,84 | 9,03 | 10,16 | 9,72 | 8,92 |

Figura 15: Resultados Experimento 1 en Servidor.
Fuente: Elaboración Propia.



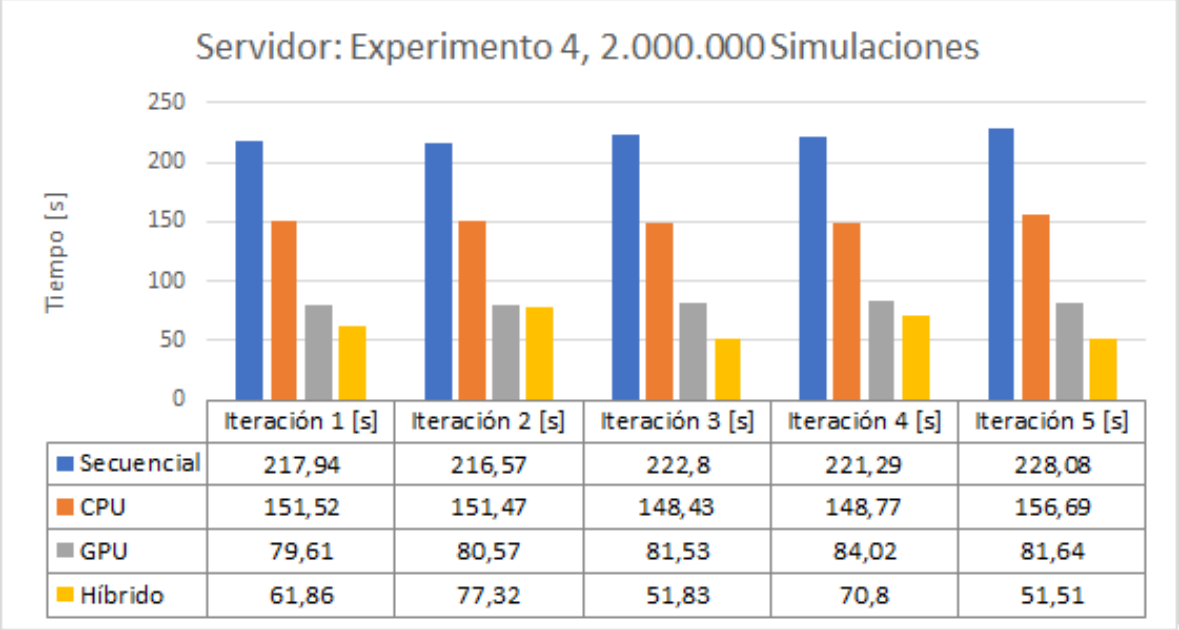

| | Iteración 1 [s] | Iteración 2 [s] | Iteración 3 [s] | Iteración 4 [s] | Iteración 5 [s] |
|---|---|---|---|---|---|
| Secuencial | 75,98 | 79,08 | 85,57 | 75,96 | 75,94 |
| CPU | 60,79 | 58,31 | 57,87 | 58,2 | 59,87 |
| GPU | 42,72 | 41,39 | 42,35 | 42,3 | 44,55 |
| Híbrido | 25,58 | 17,81 | 17,53 | 24,02 | 18,01 |

Figura 16: Resultados Experimento 2 en Servidor.
Fuente: Elaboración Propia.



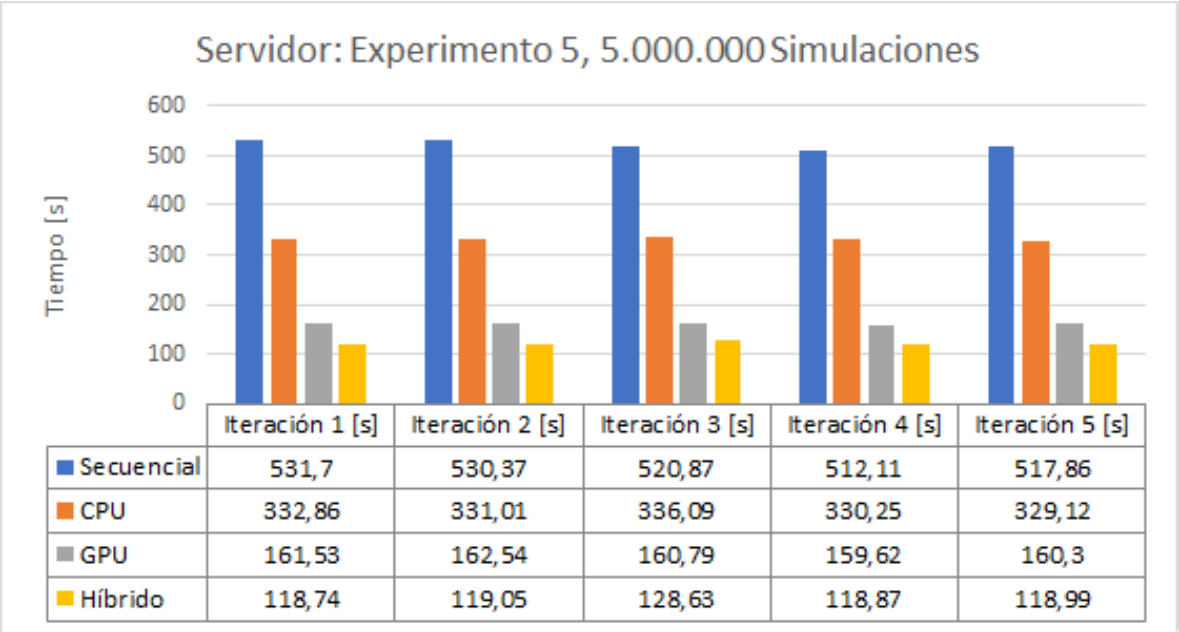

| | Iteración 1 [s] | Iteración 2 [s] | Iteración 3 [s] | Iteración 4 [s] | Iteración 5 [s] |
|---|---|---|---|---|---|
| Secuencial | 127,21 | 130,24 | 125,82 | 122,97 | 124,35 |
| CPU | 88,8 | 87,78 | 90,83 | 90,68 | 85,85 |
| GPU | 56,79 | 55,85 | 54,85 | 54,96 | 57,12 |
| Híbrido | 29,55 | 30,68 | 29,02 | 29,16 | 29,25 |

Figura 17: Resultados Experimento 3 en Servidor.
Fuente: Elaboración Propia.




| | Iteración 1 [s] | Iteración 2 [s] | Iteración 3 [s] | Iteración 4 [s] | Iteración 5 [s] |
|---|---|---|---|---|---|
| Secuencial | 217,94 | 216,57 | 222,8 | 221,29 | 228,08 |
| CPU | 151,52 | 151,47 | 148,43 | 148,77 | 156,69 |
| GPU | 79,61 | 80,57 | 81,53 | 84,02 | 81,64 |
| Híbrido | 61,86 | 77,32 | 51,83 | 70,8 | 51,51 |

Figura 18: Resultados Experimento 4 en Servidor.
Fuente: Elaboración Propia.

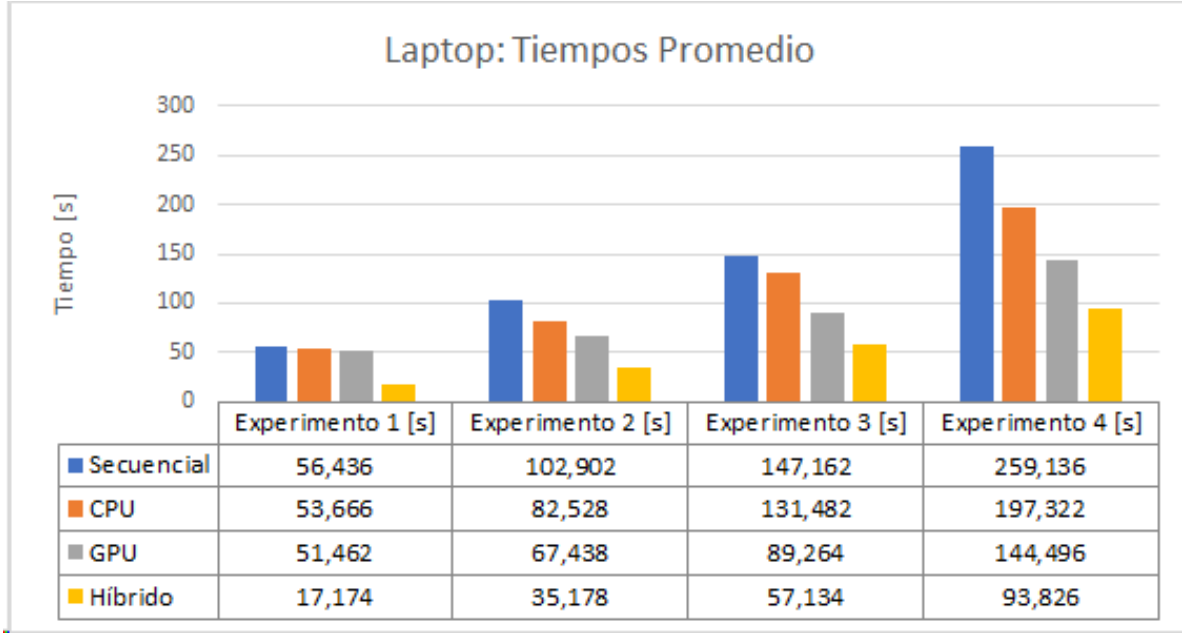



| | Iteración 1 [s] | Iteración 2 [s] | Iteración 3 [s] | Iteración 4 [s] | Iteración 5 [s] |
|---|---|---|---|---|---|
| Secuencial | 531,7 | 530,37 | 520,87 | 512,11 | 517,86 |
| CPU | 332,86 | 331,01 | 336,09 | 330,25 | 329,12 |
| GPU | 161,53 | 162,54 | 160,79 | 159,62 | 160,3 |
| Híbrido | 118,74 | 119,05 | 128,63 | 118,87 | 118,99 |

Figura 19: Resultados Experimento 5 en Servidor.
Fuente: Elaboración Propia.

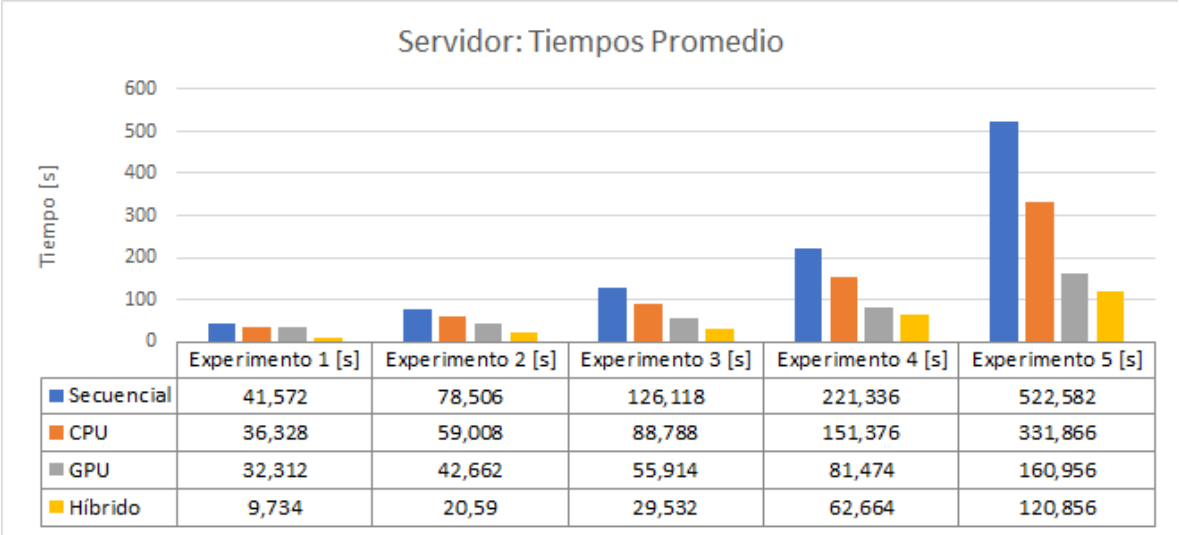
### 4.4.3. RESUMEN DE RESULTADOS

Con el fin, no solamente de facilitar aún más el análisis de los resultados obtenidos en cada una de las pruebas, sino que también para analizar de forma más objetiva el desempeño general de cada uno de los sistemas frente a una determinada cantidad de simulaciones, es que se optó por presentar los resultados en función del promedio general, agrupado por experimento y separado según la máquina en la cual dicho experimento se realiza. Es importante dejar en claro que, en las sub secciones siguientes, todo análisis que se realice será en base a los datos presentados en estos gráficos, a modo de evitar que resultados *outliers* influyen de sobremanera en las conclusiones que se puedan extraer de estos experimentos.




| | Experimento 1 [s] | Experimento 2 [s] | Experimento 3 [s] | Experimento 4 [s] |
|---|---|---|---|---|
| Secuencial | 56,436 | 102,902 | 147,162 | 259,136 |
| CPU | 53,666 | 82,528 | 131,482 | 197,322 |
| GPU | 51,462 | 67,438 | 89,264 | 144,496 |
| Híbrido | 17,174 | 35,178 | 57,134 | 93,826 |

Figura 20: Resultados Promedio de los experimento en el Laptop.
Fuente: Elaboración Propia.




| | Experimento 1 [s] | Experimento 2 [s] | Experimento 3 [s] | Experimento 4 [s] | Experimento 5 [s] |
|---|---|---|---|---|---|
| Secuencial | 41,572 | 78,506 | 126,118 | 221,336 | 522,582 |
| CPU | 36,328 | 59,008 | 88,788 | 151,376 | 331,866 |
| GPU | 32,312 | 42,662 | 55,914 | 81,474 | 160,956 |
| Híbrido | 9,734 | 20,59 | 29,532 | 62,664 | 120,856 |

Figura 21: Resultados Promedio de los experimento en el Servidor.
Fuente: Elaboración Propia.

## 4.5. ANÁLISIS DE LOS RESULTADOS

A continuación, a modo de facilitar la comparación con la solución que existe hoy en día para estos problemas se comenzará presentando un análisis del tiempo de cómputo de la misma.

Luego se presenta el análisis realizado sobre el conjunto de datos presentado en la sección anterior, el cual se divide según la procedencia de dichos datos por lo tanto, se comenzará resumiendo los descubrimientos realizados sobre los datos recabados desde el *laptop*, luego los del servidor, y finalmente se seguirá con un análisis más general de los experimentos.

### 4.5.1. ANÁLISIS DE RESULTADOS VERSIÓN *EXCEL*

Antes de comenzar el análisis como tal de los resultados obtenidos por la solución que se implementó para resolver el problema de la valorización de derivados financieros, resulta interesante para temas de comparación analizar uno de los mecanismos que hoy en día se utiliza para llevar a cabo este mismo procedimiento, el cual se trata de un análisis mediante la herramienta *Microsoft Excel*, la cual, mediante el uso de funciones de cálculo, es capaz de llevar a cabo este mismo procedimiento de valorizar, simular, y ordenar datos a modo de obtener el VaR de una cierta cartera de negocios.

Con tal de poder tener una comparación inicial en términos de tiempo de procesamiento se logró dar con un programa creado en esta herramienta, el cual, a modo de ejemplo, valoriza una cartera ficticia la cual está compuesta de 20 fechas de flujos monetarios para un mismo derivado, y genera un total de 480 simulaciones a modo de dar con el VaR de la misma. Cabe destacar que, a pesar de que este programa funciona mas que nada como una ejemplificación de cómo se lleva a cabo el proceso de determinación del VaR, y que no es la versión más optimizada que poseen hoy en día para llevar a cabo este tipo de procedimiento, se acerca lo suficiente a una situación como para poder ser utilizado como punto de comparación.

Una vez ejecutado el programa, este se tarda en promedio 8,5 [s] en completar su ejecución, esto significa, valorizar, simular y generar un VaR para la cartera analizada. Tomando este tiempo en consideración y sabiendo que solo realiza 480 simulaciones, es posible extrapolar el desempeño de este programa, con lo cual, asumiendo que se utiliza para resolver el experimento 1 propuesto (el cual consta de 100.000 simulaciones), se puede concluir que la versión en *Excel* se demora aproximadamente 1770 [s] en completar la valoración, simulación y determinación del *Value-at-Risk* para una sola cartera, lo cual se traduce a aproximadamente 30 minutos para lograr analizar una sola cartera, o visto desde el punto de vista de los experimentos realizados durante el desarrollo de este proyecto, cerca de dos horas en poder completar la valorización de las cuatro carteras estudiadas.

### 4.5.2. ANÁLISIS DE RESULTADOS OBTENIDOS EN *LAPTOP*

Lo primero que se puede observar es que, tal y como se esperaba, la versión secuencial es consistentemente más lenta en todos los experimentos realizados con respecto a las demás versiones del sistema, sin embargo, lo que no se esperaba ver es el hecho de que la versión paralela en GPU tiene un tiempo de procesamiento consistentemente mejor que la para-

lela en CPU, lo cual, considerando que la primera de estas versiones únicamente paraleliza la simulación de Monte Carlo y no el ordenamiento de los datos simulados, nos dice que este primer paso tiene mucha mayor incidencia en el tiempo total de procesamiento, y el paralelizar mediante GPU ayuda inmensamente al desempeño del sistema.

A continuación, se presentan los *speed-up* obtenidos por cada una de las versiones paralelas del sistema con respecto a la versión secuencial del mismo:

Tabla 1: *Speed-Up* promedio de los experimento en el *Laptop*.
Fuente: Elaboración Propia.

| | **Experimento 1** | **Experimento 2** | **Experimento 3** | **Experimento 4** |
|---|---|---|---|---|
| **CPU** | 1,05 | 1,25 | 1,12 | 1,31 |
| **GPU** | 1,10 | 1,53 | 1,65 | 1,79 |
| **Híbrido** | 3,29 | 2,93 | 2,58 | 2,76 |

Como se puede apreciar las diferencias de tiempo con respecto a la versión secuencial son siempre mayores a 1, lo que indica que cada una de las versiones paralelas tiene un mejor desempeño en términos de tiempo de procesamiento que su contraparte secuencial. Sin embargo, a pesar de que los desempeños de las versiones paralelas en CPU y GPU son bastante buenos, solo la versión híbrida consigue alcanzar un *speed-up* superior a dos de forma consistente, llegando incluso a ser 3 veces más rápido que su contraparte secuencial para el experimento 1 y acercándose bastante a este valor en el experimento 2.

En cuanto a la versión híbrida del sistema, se puede observar que su desempeño es significativamente mejor que el de sus contrapartes paralelas en CPU y GPU, y a pesar de que este parece tender a disminuir en eficiencia a medida que se incrementa la cantidad de simulaciones realizadas, contrario a sus contrapartes que se hacen consistentemente más eficientes, la ganancia de tiempo con respecto a la versión secuencial que este posee sigue siendo significativamente más alta que las otras versiones en todas las pruebas realizadas. Esta tendencia a perder eficiencia con respecto a la cantidad de iteraciones realizadas puede llevar a escenarios donde esta solución híbrida presente un peor desempeño que sus contrapartes, sin embargo, el margen de ganancia en términos de tiempos de procesamiento que se observa es tan alto que este escenario parece poco probable, más aún si se toma en cuenta que analizar casos con una densidad de datos mayor que en el experimento 4 resulta poco viable de llevar a cabo en un computador con características similares a las del *laptop* utilizado, debido a la cantidad de memoria que este tipo de casos requieren.

### 4.5.3. ANÁLISIS DE DATOS OBTENIDOS EN EL SERVIDOR

Continuando con el análisis, probablemente lo más interesante que se pudo apreciar del procesamiento en el servidor es el hecho de que los *speed-up* logrados son dramáticamente más grandes que los conseguidos en el *laptop*, los cuales se pueden apreciar de mejor manera en la siguiente tabla:

Tabla 2: *Speed-Up* promedio de los experimento en el Servidor.
Fuente: Elaboración Propia.

| | **Experimento 1** | **Experimento 2** | **Experimento 3** | **Experimento 4** | **Experimento 5** |
|---|---|---|---|---|---|
| **CPU** | 1,14 | 1,33 | 1,42 | 1,46 | 1,57 |
| **GPU** | 1,29 | 1,84 | 2,26 | 2,72 | 3,25 |
| **Híbrido** | 4,27 | 3,81 | 4,27 | 3,53 | 4,32 |

Como se puede ver en la tabla 2, a partir del experimento 3, las versión paralelas en GPU logra un *speed-up* con valores consistentemente mayores a dos, llegando incluso a ser tres veces más rápido que la versión secuencial del sistema en el experimento 5, lo cual demuestra que el buen desempeño que esta versión paralela posee en comparación con una versión secuencial del mismo sistema, e incluso en comparación con a una paralelización únicamente a nivel de CPU, dejando en claro lo poderoso que puede llegar a ser la paralelización en GPU, y lo bien que la misma es capaz de escalar a medida que los problemas que debe resolver se complejizan. Por otro lado, incluso este desempeño se ve opacado por la gran ganancia que se obtiene al combinar ambas lógicas en la versión híbrida del sistema, el cual fue entre tres a cuatro veces más rápido que la versión secuencial de la solución, obteniendo un desempeño general mucho mejor que cualquiera de las otras versiones implementadas.

Otra observación interesante que se puede extraer del *speed-up* logrado, es la gran diferencia que existe en términos de ganancia en tiempos de procesamiento, entre la paralelización exclusiva en CPU y la paralelización en GPU, ya que con esta última, los resultados fueron consistentemente mejores y por un margen bastante considerable. Esto también se pudo observar en las pruebas realizadas en el *laptop*, sin embargo, donde en esas pruebas las versiones paralelas en GPU fueron, a lo más, un 50 % más rápidas que su contraparte en CPU, en este caso estas versiones llegan a ser aproximadamente dos veces más rápidas que las paralelizadas a nivel de CPU, demostrando no sólo el potencial que tienen con respecto a este tipo de paralelización, sino que además la gran relevancia que tiene el *hardware* disponible al momento de utilizar este tipo de computación, y como frente a operaciones matemáticas relativamente simples pero en un alto volumen, la paralelización con GPU resulta una estrategia sumamente poderosa si se desea optimizar el tiempo de procesamiento.

Como ya se comentó, la versión híbrida del sistema es consistentemente más rápida que cualquiera de las otras soluciones implementadas para la resolución del mismo, logrando *speed-ups* de hasta cuatro veces más rápida con respecto a su contraparte secuencial, sin embargo, como se puede apreciar en la tabla 2, este tiende a ser un tanto inconsistente, y su *speed-up* fluctúa entre 3 y 4 dependiendo de la cantidad de iteraciones que se realizan, pero no escala directamente con respecto a las mismas como ocurre con las otras implementaciones del sistema, haciendo más complejo el poder predecir cómo este comportamiento se mantendrá de ir aumentando la cantidad de simulaciones y/o la densidad de los datos a ser analizados, y dejando la posibilidad de que solo sea útil solo para casos específicos, escalando de mala manera a medida que se complejizan los problemas. Aun así, y a pesar de que se

intentará responder de forma más precisa a estas interrogantes en secciones posteriores de este escrito, la ganancia de tiempo que se obtuvo en los casos probados más que compensa la dificultad en términos de programación para dar con esta versión, y parece ser lo suficientemente eficiente como para que valga la pena implementarla siempre que las propiedades del problema lo permitan.

Sin embargo, se puede apreciar que en términos de consistencia y capacidad de escalamiento, la versión paralela en GPU pareciera ser una opción mucho más segura, al entregar resultados que, a pesar de no ser tan rápidos como su contraparte híbrida, son mucho más rápidos que una implementación secuencial, además de poseer un menor grado de complejidad en términos de programación y escalar de una forma mucho más consistente a medida que los problemas a analizar se vuelven más complejos.

## 4.6. PREDICCIÓN DE COMPORTAMIENTO

Como el comportamiento de este sistema depende tanto de las capacidades de cómputo del *hardware* en el cual se ejecute, y con el fin de poder comprobar algunas de las suposiciones que se han declarado hasta el momento, es que, utilizando como base los datos recopilados en los diferentes experimentos, se determinó un modelo matemático capaz de predecir el comportamiento de las diferentes versiones del sistema al ejecutarse sobre equipos con diferentes capacidades de *hardware*. Para poder lograr esto, se aplica una regresión lineal múltiple sobre los tiempos obtenidos en las diferentes máquinas para cada una de las diferentes versiones del sistema, a modo de poder dar con una función matemática lo suficientemente precisa como para predecir el tiempo de procesamiento que le tomaría a cada versión del sistema completar una cantidad de simulaciones sobre un computador con ciertas capacidades de cálculo.

La principal dificultad al tratar de determinar una función que predice este comportamiento, está en la interpretación de los diferentes factores que inciden en el tiempo de procesamiento de cada versión del sistema: sabemos que la **Cantidad de Simulaciones** que se deben completar es uno de estos factores y que además es bastante fácil de interpretar al tratarse simplemente de un número entero. El problema está en el segundo factor de incidencia, el cual es las **Capacidades de Cálculo** del *hardware* en el cual se esté ejecutando. Este factor resulta mucho más abstracto y de por si no tiene una interpretación numérica sencilla, por lo tanto, a modo de poder dar con dicho factor, se realizaron dos supuestos importantes con respecto a los dispositivos en donde se ejecuten estos sistemas:

1. Las únicas piezas de *hardware* que afectan de forma directa la velocidad de procesamiento del sistema son la CPU y la GPU.

2. A modo de asociar un factor numérico a cada una de estas piezas, se utilizará el valor promedio de las pruebas de rendimiento que cada una posee, lo cual se conoce como

*benchmarking*, los cuales siempre se obtendrán del sitio *web PassMark Software*[2], sitio que provee datos de *benchmark* tanto para CPU como para GPU, para mantener la uniformidad con respecto a la procedencia de los datos utilizados.

Mediante estos supuesto, se determina que **el valor numérico de la capacidad de cálculo de un equipo estará dado por el promedio entre los *rating* de *benchmark* de su CPU y su GPU**. Además, recordando que el sistema se diseño utilizando CUDA, toda predicción que se haga tendrá como restricción que el equipo simulado debe tener una tarjeta gráfica de Nvidia.

Con esta definición, se procede a obtener los valores correspondientes a las piezas de *hardware* utilizadas para las pruebas que se realizaron, tanto en el *laptop* Lenovo como en el Servidor INCA, los cuales son:

Tabla 3: *Rating* de *Benchmark* de los Equipos utilizados.
Fuente: Elaboración Propia.

| **Equipo** | ***Rating* CPU** | ***Rating* GPU** | ***Rating* Promedio** |
|---|---|---|---|
| *Laptop* | 5.159 | 1.884 | 3.521,5 |
| Servidor | 12.273 | 14.243 | 13.258 |

Además, como ya se mencionó, el otro factor de incidencia es la cantidad de simulaciones que se realiza en cada experimento, con lo cual podemos definir cada experimento realizado en función de la cantidad de simulaciones que se le indica al sistema completar, de tal forma que, para efectos prácticos, se tiene que:

- Experimento 1 = 100.000 Simulaciones
- Experimento 2 = 500.000 Simulaciones
- Experimento 3 = 1.000.000 Simulaciones
- Experimento 4 = 2.000.000 Simulaciones
- Experimento 5 = 5.000.000 Simulaciones

Así mismo, y en concordancia con lo que se puede apreciar en la tabla anterior, definimos para efectos prácticos que:

- *Laptop* = 3.521,5
- Servidor = 13.258

[2]PassMark Software. Se visitó el 20 de julio del 2019, https://www.passmark.com/

A partir de estos datos, y conociendo de antemano el tiempo promedio que tardaron en completarse estas simulaciones para cada una de las versiones del sistema (representado en segundos), se puede construir una serie de matrices, una para cada una de estas versiones, que describen el tiempo promedio de ejecución en función de estos dos parámetros numéricos independientes. Luego, se tiene que:

### 4.6.1. ANÁLISIS ESTADÍSTICA DE LA VERSIÓN SECUENCIAL

Tabla 4: Matriz de tiempo de ejecución, Versión Secuencial.
Fuente: Elaboración Propia.

| | **Experimento 1** | **Experimento 2** | **Experimento 3** | **Experimento 4** | **Experimento 5** |
|---|---|---|---|---|---|
| ***Laptop*** | 57,64 | 100,62 | 146,60 | 259,29 | - |
| **Servidor** | 41,57 | 78,51 | 126,12 | 221,34 | 522,58 |

Tabla 5: Regresión Lineal Múltiple, Matriz de Tiempo de la Versión Secuencial.
Fuente: Elaboración Propia.

| | Coeficientes | Error típico | Estadístico t | Probabilidad | Inferior 95 % | Superior 95 % |
|---|---|---|---|---|---|---|
| Intercepción | 60,5391458 | 3,694022 | 16,388407 | 3,2876E-06 | 51,500198 | 69,578093 |
| Variable X 1 | -0,002505 | 0,000371 | -6,752149 | 0,000514 | -0,003413 | -0,001597 |
| Variable X 2 | 9,9247E-05 | 1,2362E-06 | 80,278199 | 2,5156E-10 | 9,6221E-05 | 0,000102 |

Tabla 6: Análisis Estadístico de la regresión, Versión Secuencial.
Fuente: Elaboración Propia.

| Estadísticas de la regresión | |
|---|---|
| Coeficiente de correlación múltiple | 0,99955350 |
| Coeficiente de determinación $R^2$ | 0,99910721 |

### 4.6.2. ANÁLISIS ESTADÍSTICA DE LA VERSIÓN PARALELA EN CPU

Tabla 7: Matriz de tiempo de ejecución, Versión CPU.
Fuente: Elaboración Propia.

| | **Experimento 1** | **Experimento 2** | **Experimento 3** | **Experimento 4** | **Experimento 5** |
|---|---|---|---|---|---|
| ***Laptop*** | 53,61 | 83,18 | 126,87 | 198,19 | - |
| **Servidor** | 36,33 | 59,01 | 88,79 | 151,38 | 331,87 |

Tabla 8: Regresión Lineal Múltiple, Matriz de Tiempo de la Versión CPU.
Fuente: Elaboración Propia.

| | Coeficientes | Error típico | Estadístico t | Probabilidad | Inferior 95 % | Superior 95 % |
|---|---|---|---|---|---|---|
| Intercepción | 71,314782 | 6,3849829 | 11,169142 | 3,073E-05 | 55,691292 | 86,938272 |
| Variable X 1 | -0,003403 | 0,0006414 | -5,304738 | 0,0018215 | -0,004972 | -0,001833 |
| Variable X 2 | 6,237E-05 | 2,137E-06 | 29,185972 | 1,072E-07 | 5,714E-05 | 6,759E-05 |

Tabla 9: Análisis Estadístico de la regresión, Versión CPU.
Fuente: Elaboración Propia.

| Estadísticas de la regresión | |
|---|---|
| Coeficiente de correlación múltiple | 0,9965332 |
| Coeficiente de determinación $R^2$ | 0,9930784 |

### 4.6.3. ANÁLISIS ESTADÍSTICA DE LA VERSIÓN PARALELA EN GPU

Tabla 10: Matriz de tiempo de ejecución, Versión GPU.
Fuente: Elaboración Propia.

| | **Experimento 1** | **Experimento 2** | **Experimento 3** | **Experimento 4** | **Experimento 5** |
|---|---|---|---|---|---|
| ***Laptop*** | 52,91 | 67,86 | 88,35 | 150,23 | - |
| **Servidor** | 32,31 | 42,66 | 55,91 | 81,47 | 160,96 |

Tabla 11: Regresión Lineal Múltiple, Matriz de Tiempo de la Versión GPU.
Fuente: Elaboración Propia.

| | Coeficientes | Error típico | Estadístico t | Probabilidad | Inferior 95 % | Superior 95 % |
|---|---|---|---|---|---|---|
| Intercepción | 77,703477 | 10,286238 | 7,5541198 | 0,0002793 | 52,533958 | 102,87299 |
| Variable X 1 | -0,004018 | 0,0010334 | -3,888227 | 0,0080933 | -0,006547 | -0,001489 |
| Variable X 2 | 2,920E-05 | 3,443E-06 | 8,4832009 | 0,0001467 | 2,078E-05 | 3,763E-05 |

Tabla 12: Análisis Estadístico de la regresión, Versión GPU.
Fuente: Elaboración Propia.

| Estadísticas de la regresión | |
|---|---|
| Coeficiente de correlación múltiple | 0,9619793 |
| Coeficiente de determinación $R^2$ | 0,9254042 |

### 4.6.4. ANÁLISIS ESTADÍSTICA DE LA VERSIÓN PARALELA HÍBRIDA

Tabla 13: Matriz de tiempo de ejecución, Versión Híbrida.
Fuente: Elaboración Propia.

| | **Experimento 1** | **Experimento 2** | **Experimento 3** | **Experimento 4** | **Experimento 5** |
|---|---|---|---|---|---|
| ***Laptop*** | 17,17 | 35,18 | 57,13 | 93,83 | - |
| **Servidor** | 9,73 | 20,59 | 29,53 | 62,66 | 120,86 |

Tabla 14: Regresión Lineal Múltiple, Matriz de Tiempo de la Versión Híbrida.
Fuente: Elaboración Propia.

| | Coeficientes | Error típico | Estadístico t | Probabilidad | Inferior 95 % | Superior 95 % |
|---|---|---|---|---|---|---|
| Intercepción | 34,503654 | 7,2653863 | 4,7490461 | 0,0031618 | 16,725895 | 52,281414 |
| Variable X 1 | -0,002143 | 0,0007299 | -2,936423 | 0,0260685 | -0,003929 | -0,000357 |
| Variable X 2 | 2,476E-05 | 2,432E-06 | 10,182820 | 5,223E-05 | 1,881E-05 | 3,071E-05 |

Tabla 15: Análisis Estadístico de la regresión, Versión Híbrida.
Fuente: Elaboración Propia.

| Estadísticas de la regresión | |
|---|---|
| Coeficiente de correlación múltiple | 0,9722673 |
| Coeficiente de determinación $R^2$ | 0,9453037 |

### 4.6.5. EXPERIMENTACIÓN EN BASE AL ANÁLISIS ESTADÍSTICO

Como se puede apreciar, si se considera el tiempo promedio de ejecución como la variable dependiente y, tomando tanto la cantidad de simulaciones como el *rating* promedio de los equipos, como las variables independientes, se puede generar para cada una de estas matrices una regresión lineal múltiple, lo cual, gracias a las herramientas de análisis estadístico de *Microsoft Excel*, resulta una tarea relativamente sencilla una vez ordenados los datos. De esta forma, se puede observar que el coeficiente de correlación de Pearson [Kent State University, 2019] para cada una de estas regresiones es bastante cercano a 1, lo cual dice que el modelo se ajusta bastante a los valores obtenidos, y mediante el uso de los coeficientes obtenidos para el intercepto y para cada una de las dos variables independientes, se puede generar una función matemática capaz de estimar el tiempo promedio de ejecución en relación a la cantidad de simulación y al *rating* del equipo utilizado.

Un resultado interesante que vale la pena tomar en cuenta, es que el intercepto para cada uno de estos casos se encuentra alrededor de los 60 segundos, con la única excepción siendo la versión híbrida, lo cual nos dice que, en caso de que cualquiera de las variables sea cero, o siendo más realistas, en caso de que la cantidad de simulaciones sea cero, el sistema seguirá tardando por lo menos 60 segundos en completar su ejecución. A pesar de que puede parecer un resultado extraño, este resultado se apega a la realidad, ya que, como se mencionó en la sección anterior, lo primero que hace cada versión del sistema es extraer los datos relevantes de la cartera de derivados y almacenarlos en memoria, proceso que, producto de que ciertos datos deben extraerse directamente desde *internet* y procesarse hasta cierto punto, se tarda entre 30 a 60 segundos dependiendo de la red donde se ejecute, lo cual está en concordancia con este intercepto.

Luego, en base a estos parámetros y utilizando nuevamente los datos de *rating* de *benchmark* obtenidos desde *PassMark Software*, es que se diseñaron una serie de máquinas de prueba, cada una con un *rating* promedio diferente, el cual nace de diferentes piezas de *hardware*. A continuación se listan todos los componentes de las máquinas de prueba simuladas:

- **Máquina 1**:
  - **CPU**: Intel Core i5-3570K @ 3.40GHz, *Rating* 7.174

- **GPU**: GeForce GTX 970, *Rating* 8.623
- ***Rating* Promedio**: 7.898,5

- **Máquina 2**:
  - **CPU**: Intel Core i5-3570K @ 3.40GHz, *Rating* 7.174
  - **GPU**: GeForce GTX 1060, *Rating* 9.090
  - ***Rating* Promedio**: 8.132
- **Máquina 3**:
  - **CPU**: Intel Core i7-3770 @ 3.40GHz, *Rating* 9.282
  - **GPU**: GeForce GTX 970, *Rating* 8.623
  - ***Rating* Promedio**: 8.952,5
- **Máquina 4**:
  - **CPU**: Intel Core i7-3770 @ 3.40GHz, *Rating* 9.282
  - **GPU**: GeForce GTX 1060, *Rating* 9.090
  - ***Rating* Promedio**: 9.186
- **Máquina 5**:
  - **CPU**: Intel Core i5-3570K @ 3.40GHz, *Rating* 7.174
  - **GPU**: GeForce GTX 1080, *Rating* 12.423
  - ***Rating* Promedio**: 9.250,5
- **Máquina 6**:
  - **CPU**: Intel Core i5-3570K @ 3.40GHz, *Rating* 7.174
  - **GPU**: GeForce GTX 1080, *Rating* 12.423
  - ***Rating* Promedio**: 9.798,5
- **Máquina 7**:
  - **CPU**: Intel Core i7-3770 @ 3.40GHz, *Rating* 9.282
  - **GPU**: GeForce GTX 1070, *Rating* 11.327
  - ***Rating* Promedio**: 10.304,5
- **Máquina 8**:
  - **CPU**: Intel Core i7-3770 @ 3.40GHz, *Rating* 9.282
  - **GPU**: GeForce GTX 1080, *Rating* 12.423
  - ***Rating* Promedio**: 10.852,5

- **Máquina 9**:
  - **CPU**: Intel Core i9-9960X @ 3.10GHz, *Rating* 29.959
  - **GPU**: NVIDIA TITAN RTX, *Rating* 18.916
  - ***Rating* Promedio**: 24.437,5

Además de predecir el tiempo promedio que se tardan en completar los diferentes experimentos sobre cada una de estas máquinas simuladas, también se simuló cuánto se tardará en completarse el experimento 5 sobre la máquina *Laptop*, la cual, como ya mencionamos, fue incapaz de completar este experimento por temas de memoria. Adicionalmente, a modo de tener una mayor cantidad de datos que analizar, se creó un nuevo experimento llamado **Experimento 6**, el cual consta de procesar 10.000.000 simulaciones. Por último, la máquina 9 está compuesta de los dispositivos de *hardware* que actualmente poseen el *rating* más alto en términos de desempeño, para poder observar como se espera que el sistema se desenvuelva en este tipo de ambiente.

A continuación, se presentan los datos de predicción de desempeño en términos de tiempos de procesamiento de las diferentes máquinas, tanto las reales como las simuladas, en forma de gráficos, separados en función de los diferentes experimentos para visualizar de mejor manera los resultados obtenidos:

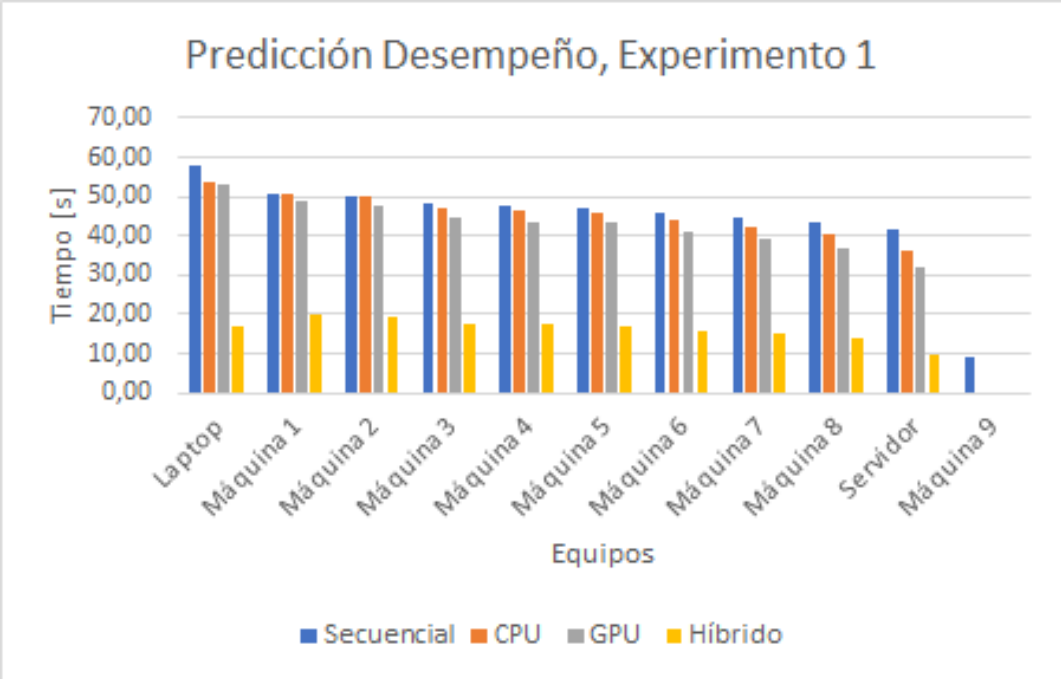



Figura 22: Predicción de Desempeño de los Equipos sobre el Experimento 1.
Fuente: Elaboración Propia.

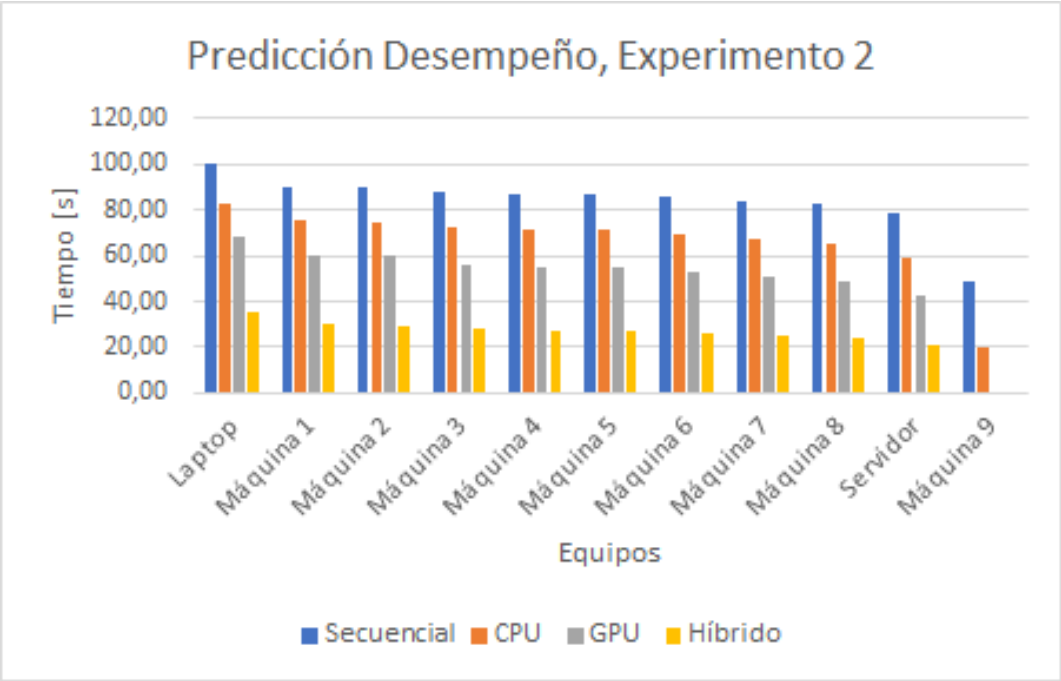



Figura 23: Predicción de Desempeño de los Equipos sobre el Experimento 2.
Fuente: Elaboración Propia.

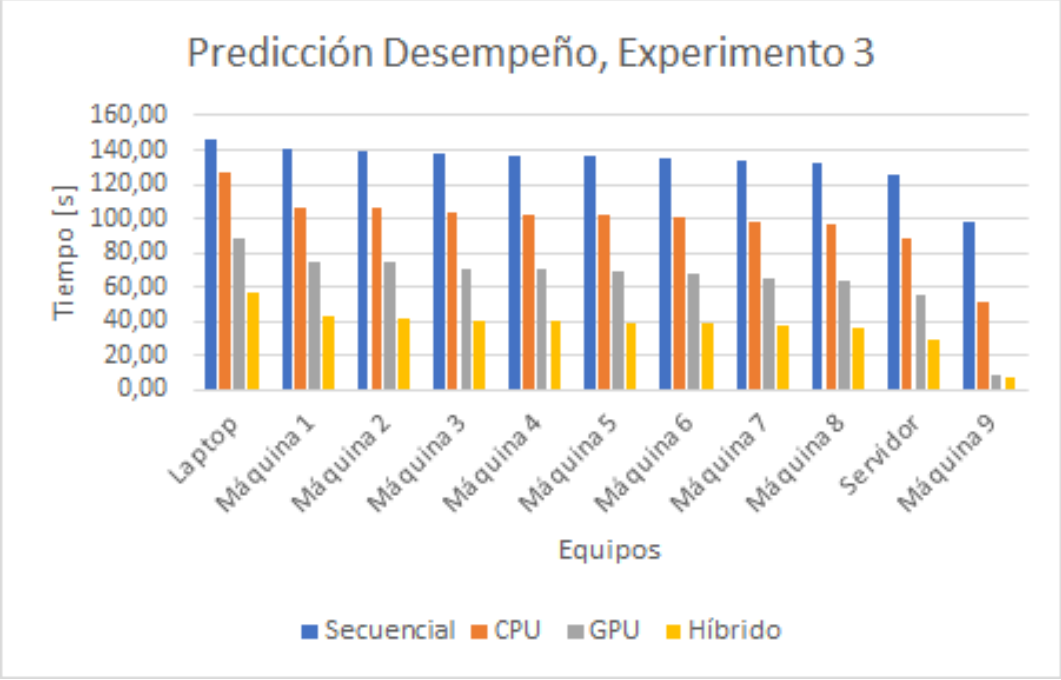



Figura 24: Predicción de Desempeño de los Equipos sobre el Experimento 3.
Fuente: Elaboración Propia.

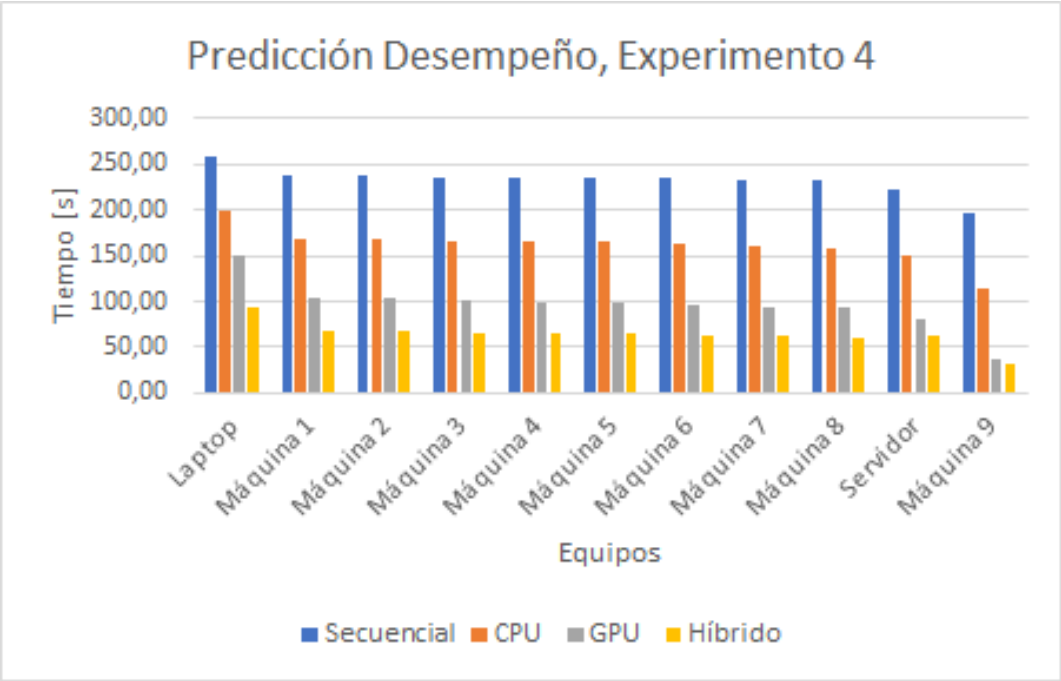



Figura 25: Predicción de Desempeño de los Equipos sobre el Experimento 4.
Fuente: Elaboración Propia.

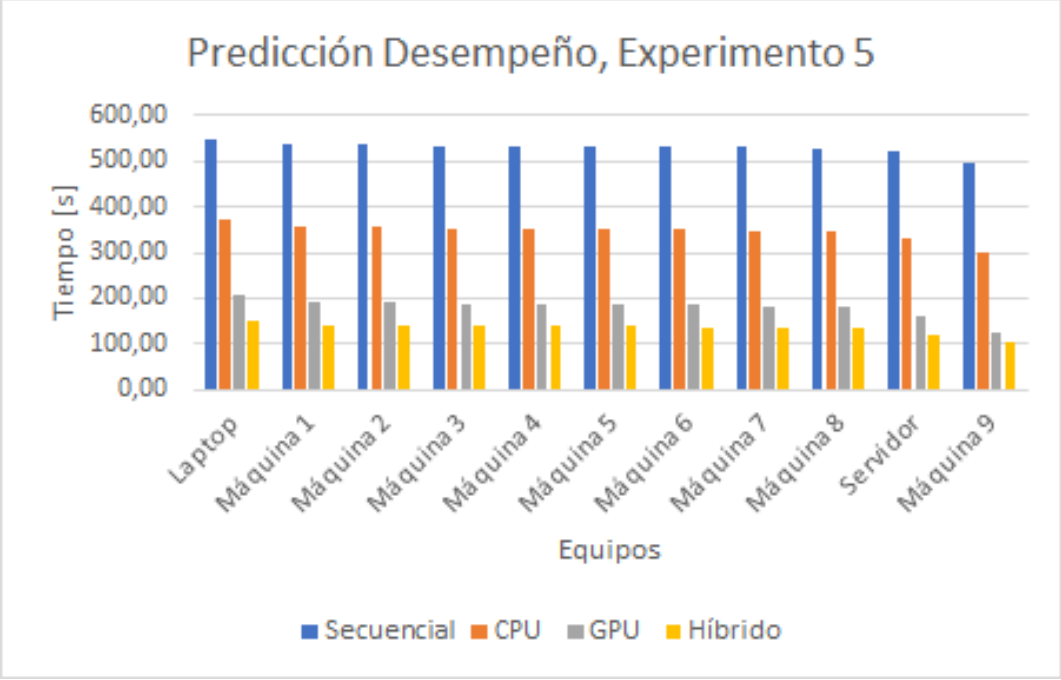



Figura 26: Predicción de Desempeño de los Equipos sobre el Experimento 5.
Fuente: Elaboración Propia.

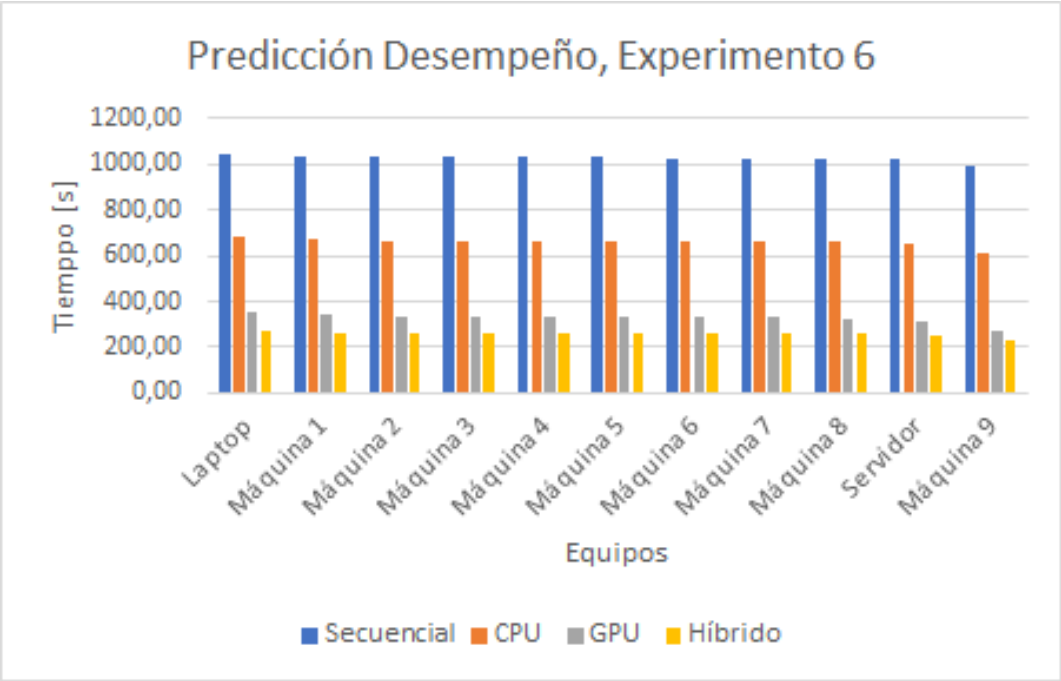



Figura 27: Predicción de Desempeño de los Equipos sobre el Experimento 6.
Fuente: Elaboración Propia.

Como se puede apreciar en los datos simulados, muchas de las conclusiones obtenidas al analizar las pruebas realizadas sobre los equipos *Laptop* y Servidor, se mantienen alrededor de todos los equipos probados. Queda claro que la versión secuencial es, sin lugar a dudas, la que peor desempeño tiene a lo largo de todos los experimentos realizados, y que cualquier tipo de paralelización que se aplique reducirá el tiempo de procesamiento de forma sustancial comparado con esta versión. Además, se puede observar como la versión paralela en GPU, a pesar de no paralelizar el ordenamiento de los datos, tiene un desempeño consistentemente mejor que el de su contraparte en CPU, y que la versión paralela híbrida es la que mejor desempeño tiene a lo largo de todos los experimentos, en todas las máquinas analizadas.

Por otra parte, en lo que respecta a la máquina 9, en los experimentos 1 y 2 se desempeñó tan rápido que el modelo predijo que se tardaría un tiempo negativo en completar el procesamiento, por lo cual, para efectos de mantener la lógica, se dejaron con un valor de tiempo cero. Pero esto entrega dos piezas de información muy relevantes: la primera es que, frente a un *hardware* tan poderoso, el procesamiento de casos de prueba tan pequeños es básicamente inmediato, y segundo que, como se preveía que ocurriera, el modelo que se obtuvo, al nacer a partir de tan pocos datos, no es lo suficientemente preciso como para estimar el tiempo real que una máquina con capacidades que van más allá de lo normal, se tarda en completar esta tarea, por lo tanto queda espacio para mejorar este modelo, lamentablemente por limitaciones técnicas fue imposible obtener mas datos para seguir alimentando al mismo.

Por último, y a modo de poder analizar de forma más precisa la diferencia de tiempo entre las versiones paralelas en GPU e Híbrida, es que se presenta otro conjunto de gráficos, que sólo miden la comparación de desempeño entre estas dos versiones del sistema en los últimos tres experimentos, ya que, producto de la gran diferencia de tiempo que existe entre estas versiones y la versión secuencial del sistema, no se puede apreciar con detalle la diferencia real de tiempo que se da entre estas dos versiones en los últimos experimentos realizados:

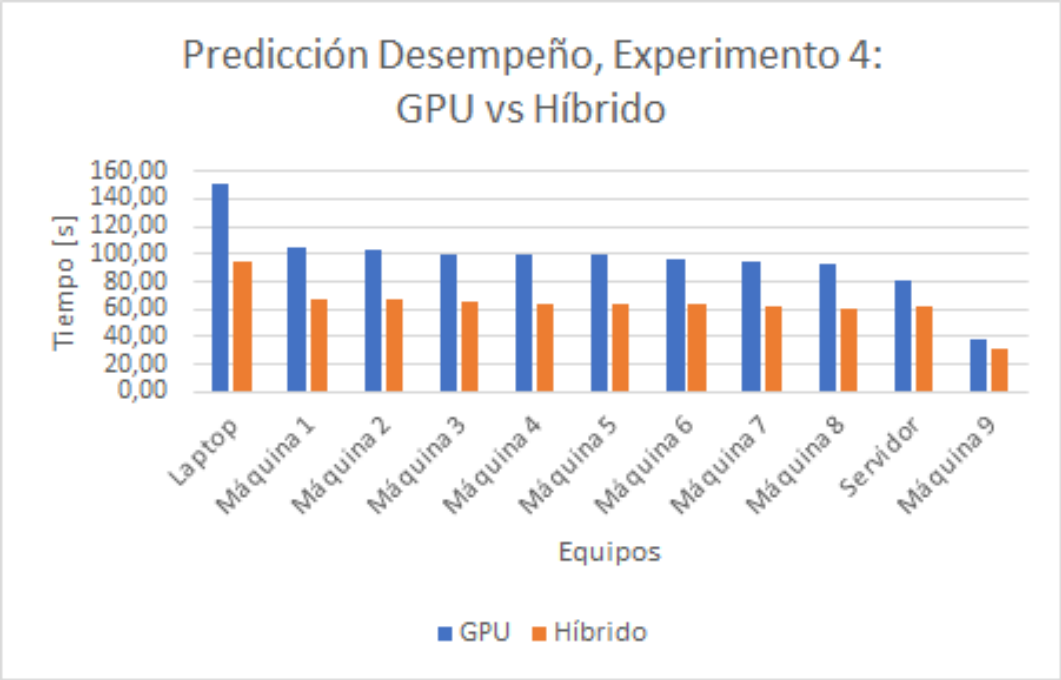



Figura 28: Predicción de Desempeño, GPU vs Híbrido, Experimento 4.
Fuente: Elaboración Propia.

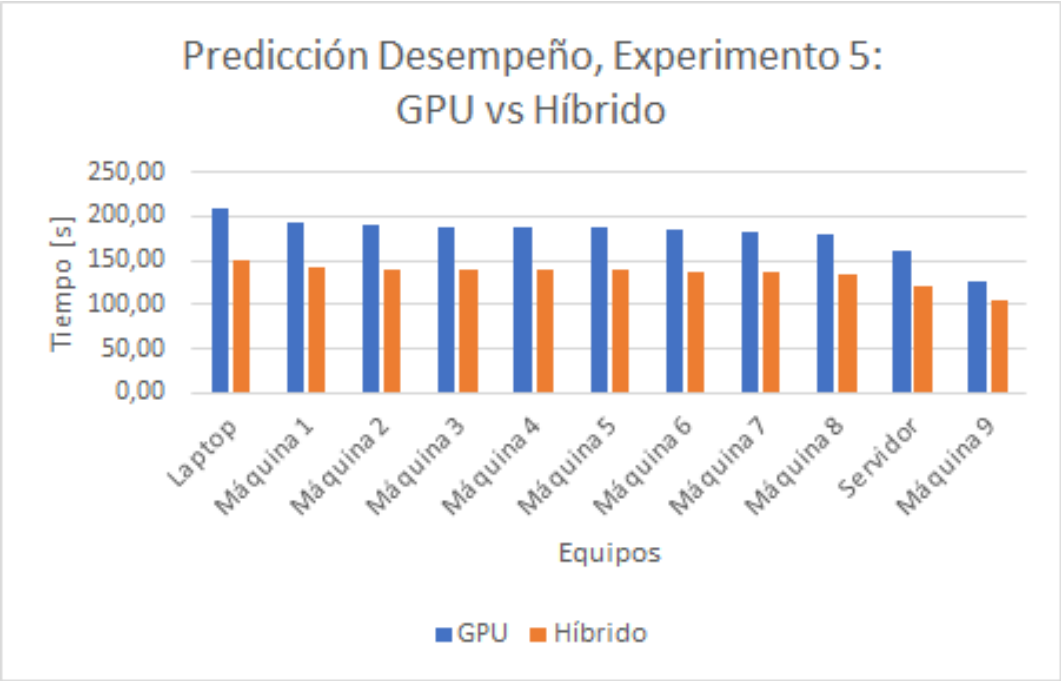



Figura 29: Predicción de Desempeño, GPU vs Híbrido, Experimento 5.
Fuente: Elaboración Propia.

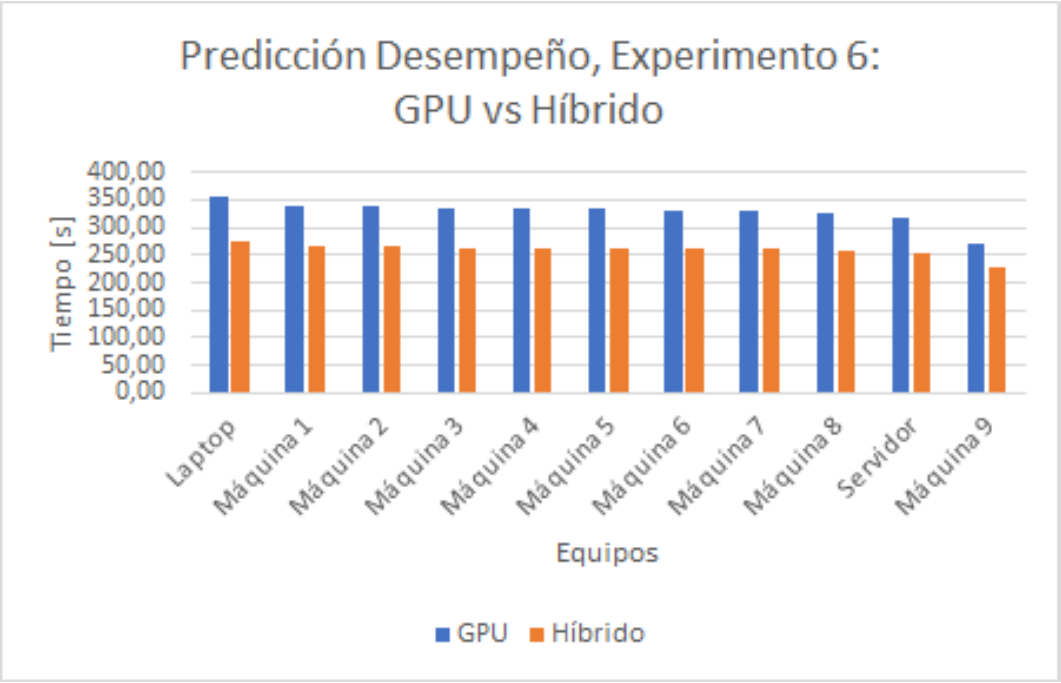



Figura 30: Predicción de Desempeño, GPU vs Híbrido, Experimento 6.
Fuente: Elaboración Propia.

Como se puede apreciar, a pesar de que el desempeño de la versión híbrida es consistentemente mejor que el de la versión paralela en GPU, esta no escala de forma perceptible a medida que se mejora el *hardware* sobre el cual se está ejecutando, y su comportamiento es bastante regular, y solo se ve alterado cuando los cambios en el *hardware* utilizado son sustanciales, por lo cual se puede asumir que la versión paralela híbrida tiene un cierto grado de independencia con respecto al *hardware* utilizado, y su desempeño no depende tanto de este, y más bien depende de las propiedades del problema que está intentando resolver.

Esto tiene dos matices a considerar, por un lado, si se dispone de un equipo de gama media, la versión paralela híbrida, al no depender tan intensamente de las características de la máquina en donde se ejecuta, escalará de mucho mejor manera que una implementación paralela únicamente a nivel de GPU. Pero por otro lado, a pesar de que sus resultados siguen siendo mejores, la versión paralela en GPU tiende a comportarse de manera sustancialmente mejor a medida que se incrementa la calidad del *hardware* utilizado, y puede que llegue a ser mucho más útil si se dispone de computadores de una gama mucho más alta.

# CAPÍTULO 5
# CONCLUSIONES

A continuación, se presentan las conclusiones generales que se rescataron a lo largo del desarrollo del proyecto, las cuales están relacionadas tanto con el diseño y construcción del proyecto como tal, como con los resultados obtenidos durante las pruebas que se le realizaron al mismo. A través de estas conclusiones, se intentarán responder las interrogantes planteadas al inicio de este proyecto, además de dejar en claro las áreas en las que se considera se puede seguir mejorando el trabajo que se realizó, terminando con el trabajo a futuro que todavía queda por realizar.

## 5.1. CONCLUSIONES GENERALES

A lo largo de este proyecto se fue adquiriendo conocimiento nuevo en múltiples áreas del mismo, y a pesar de que existen muchas conclusiones que se pueden sacar respecto a él, sin lugar a dudas la primera de estas conclusiones es la siguiente: A pesar de depender de las propiedades del problema que se busca resolver y, aunque dificulte aún más el diseño y el desarrollo asociados a dicha solución, el uso de paralelización, sin importar el tipo de la misma, genera resultados que superan con creces los costos adicionales de estas limitaciones. Por lo tanto, siempre que el problema lo permita, la paralelización del código que lo resuelve debería, al menos, tomarse en consideración y evaluarse con anterioridad y a lo largo del proyecto, sobre todo durante los resultados observados del mismo, ya que se puede apreciar como el uso de paralelización ayuda enormemente a la resolución de la problemática, no sólo reduciendo de forma significativa los tiempos de cálculo asociados con la solución, sino que también aprovechando de mucho mejor manera los recursos de *hardware* disponibles, con lo cual un simple *laptop* de uso diario puede llegar a superar en desempeño a un servidor de cálculo, demostrando el poder que esta técnica tiene y lo mucho que ayuda a ahorrar, no sólo en términos de tiempo, sino que también en términos monetarios, consiguiendo resultados en tiempos iguales o incluso menores con un *hardware* objetivamente inferior.

Siguiendo la misma línea, podemos concluir que, frente a procesos computacionalmente exigentes pero matemáticamente sencillos, la paralelización con GPU es sumamente más poderosa que su contraparte en CPU, y hoy en día, con todos los avances en las herramientas que permiten su uso y la extensa documentación que existe con respecto a la misma, la paralelización en GPU debería ser una herramienta mucho más utilizada de lo que hoy en día es, ya que los beneficios que conlleva son sustanciales, y la dificultad de programar en ella es cada vez menor, siendo la herramienta CUDA particularmente simple de utilizar y mucho más intuitiva que las herramientas de paralelización a nivel de CPU más utilizadas, debido principalmente a que el usuario no debe preocuparse directamente de gestionar el como es-

ta paralelización se lleva a cabo, dejándole mucha de la complejidad del proceso a CUDA, y preocupándose sólo de definir la lógica detrás de la misma y los parámetros necesarios para que se ejecute, a diferencia de herramientas más clásicas como *pthread*, en las cuales es necesario definir de forma manual como la conmutación de tareas se llevará a cabo, mediante el uso de mutex definidos por el mismo usuario. Por lo cual la paralelización con CUDA no sólo es más eficiente en estos casos, tal y como se puede apreciar en los resultados obtenidos a lo largo de las pruebas que se llevaron a cabo, sino que también es mucho más intuitiva, fácil de aprender y de utilizar, y su potencial es mucho más grande que lo que se pudo explorar durante el desarrollo de este proyecto. En general es un campo de estudio muy interesante, aunque muy poco explorado, y a nivel nacional, sumamente poco conocido y utilizado.

Yendo más a lo específico, y como se pudo apreciar en la sección anterior con respecto a la versión híbrida del sistema, ésta es claramente superior en términos de tiempo de procesamiento promedio a todas sus contrapartes, llegando a ser hasta cuatro veces más rápida que su contraparte secuencial, siendo la versión paralela en GPU la única con un desempeño remotamente igualable, y aun así la diferencia en el tiempo de procesamiento estuvo a favor de la versión híbrida en todos los casos de prueba estudiados. Lo anterior hace parecer que, en caso de que la paralelización sea una opción viable, se debería optar siempre por una implementación híbrida de la solución, sin embargo, hay una serie de factores a tener al momento de evaluar el costo/beneficio de una implementación paralela híbrida, comenzando por el hecho de que la complejidad de desarrollar la misma es significativamente más alta que la de una solución paralela en GPU o en CPU, y dependiendo del proyecto puede que este periodo de desarrollo más largos no sea viable y se tenga que optar por una solución diferente, además, a pesar de que en este proyecto se demostró el potencial de esta implementación para resolver problemas de valorización y cálculo de *value-at-risk* para una cartera de negocios, es imposible, en base a los resultados obtenidos, predecir el comportamiento de este tipo de solución de implementarse para resolver un problema diferente, debido a que la paralelización en si depende mucho de las propiedades específicas del problema donde se aplica, y es muy probable que en otros casos no se comporte de forma tan favorable.

Otro punto interesante de la paralelización híbrida es como esta escala en relación a las capacidades de las piezas de *hardware* sobre las cuales se ejecuta, lo cual se puede apreciar con mayor detalle en las figuras 28, 29 y 30. Aquí, se ve claramente cómo a medida que el *hardware* incrementa en sus capacidades de cálculo los tiempos de procesamiento de la versión híbrida no tienden a variar demasiado, con fluctuaciones de no más de 5 segundos en promedio, mientras que la versión paralela únicamente en GPU tiene mejoras de tiempo mucho más perceptibles a medida que se mejora la calidad de las piezas de computo. Esto significa que, en caso de no poseer el capital para obtener piezas de *hardware* de alto nivel, la solución paralela híbrida resulta sumamente conveniente, ya que será capaz de utilizar de la mejor forma posible los elementos que ya se posee, y entregará resultados significativamente mejores que cualquier otro tipo de paralelización siempre que el problema lo permita, mientras que si se tiene acceso a una GPU de última generación puede que la versión paralela en GPU entregue mejores resultados que una solución híbrida, sobre todo

si la CPU de la que se dispone no está a la par con la tarjeta gráfica, generando un cuello de botella que la primera versión paralela no posee, y viceversa, si se tiene una CPU de gama alta pero no una GPU a la par, una solución paralela en CPU puede que entregue muchos mejores resultados.

Un punto interesante a tomar en cuenta es la comparación entre la versión del cálculo procesada en *Excel* con los resultados obtenidos de la solución propuesta. Como se pudo apreciar, dicha versión es sumamente lenta e ineficiente, siendo consistentemente más lenta y capaz de procesar una cantidad mucho menor de escenarios que incluso la versión secuencial de la solución propuesta, por lo que a primera vista pareciera no existir ventaja perceptible a procesar el cálculo con *Excel* en lugar que con este tipo de sistema. Sin embargo, esta herramienta le aporta al usuario una facilidad de uso que la solución propuesta no posee, al ser capaz de armar y editar diferentes escenarios de forma relativamente fácil y sin la ayuda de un programador externo, razón por la que este tipo de enfoque se suele utilizar bastante a nivel industrial para hacerle frente a los problemas de valorización y simulación, no obstante, la diferencia de tiempo y mayor precisión en el cálculo hacen que la opción de una solución dedicada, aunque sea una programada de forma secuencial, sea completamente válida, y muchas veces deseada al tratarse de operaciones que requieren de decisiones rápidas con tal de adquirir el mayor beneficio de las mismas.

Por lo tanto, tras el trabajo realizado a lo largo de este proyecto, se puede afirmar que el uso de técnicas de paralelización para agilizar procesos de análisis financieros, tales como la valorización de derivados y el cálculo del *Value-at-Risk*, es una práctica sumamente recomendada, ya que la naturaleza de estos cálculos se acomoda perfectamente a la lógica de la paralelización, y los resultados que estas técnicas entrega son más que deseados, generando en el peor de los casos una mejora cercana al 50 % en términos de velocidad de procesamiento y llegando a ser hasta 3 veces más rápido al utilizar paralelización en GPU, y 4 veces más al utilizar paralelización Híbrida, lo cual ayuda de sobremanera no sólo a los tomadores de decisiones para aprovechar mejores oportunidades de negocios, sino que la reducción en tiempo de procesamiento ayuda a los usuarios que hagan uso de este tipo de cálculo para tareas menos críticas, pero para los cuales el menor tiempo de procesamiento les significa una mayor agilidad y una mejor experiencia de uso con los sistemas.

En lo que respecta a la paralelización híbrida y su uso para este tipo de operaciones, la respuesta dependerá mucho de las condiciones en las cuales se aplique. Resulta ser que este tipo de paralelización no depende mucho de las piezas individuales de *hardware* sobre las que se ejecuta, sino que más bien depende de la diferencia entre las capacidades de cálculo que cada una tiene con respecto a la otra, obteniendo resultados mucho más consistentes mientras menor sea esta diferencia, haciendo que su comportamiento sea difícil de predecir, sin embargo, para casos en los que no se dispone de equipos con muy altas capacidades de cálculo, pero donde la diferencia entre las capacidades independientes de la CPU y la GPU no es tan grande, la paralelización híbrida puede ser la mejor opción, agilizando el comportamiento del sistema de forma mucho más significativa que métodos más clásicos de paralelización. A pesar de ser un campo muy amplio, y que aún posee muchos elementos

por estudiar y analizar, los cuales no se lograron cubrir por completo durante el desarrollo de este proyecto, la idea de un uso híbrido de las piezas de *hardware* parece ser sumamente prometedora, y puede que sea uno de los mejores métodos para agilizar el procesamiento de una solución.

## 5.2. TRABAJO FUTURO

A pesar de que se intentó cubrir de la mejor forma posible los temas tratados en este proyecto, los resultados obtenidos dejan ver que todavía existe un amplio campo que se puede seguir estudiando con respecto al mismo, y quedo mucho trabajo por hacer, el cual, producto del alcance de este proyecto, no se lograra cubrir, sin embargo se espera que a futuro este puede ser explorado con más detalle. Una de las áreas que quedaron al debe, es la cantidad de las pruebas que se llevaron a cabo ya que, a pesar de que se probó el sistema y se midió su desempeño de la mejor forma posible y en dos equipos diferentes, puede que utilizar sólo dos máquinas no proporcione la suficiente cantidad de diversidad como para obtener una muestra concluyente; aunque, por temas de recursos disponibles, fue imposible dar con otros equipos con las capacidades técnicas apropiadas como para seguir probando el desempeño del sistema. En este sentido, si se desea estudiar más a fondo las capacidades que este posee, y si se quiere comprobar o refutar de forma más definitiva las conclusiones a las que se llegaron durante el transcurso de este proyecto, lo primero que se debería hacer es probar el mismo en múltiples piezas de *hardware* diferentes, a modo de obtener un espacio de muestra mucho más complejo y que refleje de mejor manera el comportamiento real que tiene el sistema.

En ese sentido, una de las tareas que se pueden llevar a cabo en el futuro y que de cierto modo depende de la capacidad de llevar a cabo más pruebas sobre el sistema, es descubrir de qué forma se relacionan realmente la CPU y la GPU y, cómo esa relación afecta al funcionamiento de la versión híbrida sobre un cierto equipo, ya que, como se observó en los resultados obtenidos, a pesar de que se ejecutó la versión híbrida sobre un equipo objetivamente mejor que el *laptop*, las diferencias en términos de *speed-up* no son tan sustanciales entre sí, y las ganancias de tiempo de procesamiento obtenidas en el servidor tienden a ser irregulares, a pesar de ser significativas, no parecen regirse por ningún patrón fácilmente observable. Este comportamiento se le atribuye a la diferencia en la capacidad de cálculo de estos dos elementos ya que, en el servidor utilizado, la GPU disponible era significativamente más poderosa que la CPU, lo cual provocó estas discrepancias, sin embargo sería mucho más interesante llegar a determinar, en base a las características de las piezas, como es que estas se desempeñarán al funcionar en conjunto, a modo de predecir que tan bien escalará el sistema híbrido sobre un sistema sin necesidad de ejecutarlo directamente en el mismo, cosa que ayudaría a los posibles usuarios de un sistema así, recomendandoles que piezas de *hardware* adquirir.

En cuanto a tareas que quedaron por hacer, desde el punto de vista del funcionamiento del sistema, probablemente la más sencilla y directa que no se logró completar fue la incorpo-

ración de un sistema de base de datos para almacenar tanto los registros de los derivados a analizar cómo la data extra que se requiere para su análisis, esto es información tanto del mercado en el cual se transen como de las monedas que se ocupan para tranzarlas. La forma en que esto se aplicó en el sistema actual fue mediante el uso de *APIs*, a las cuales se les solicita la información relevante a las monedas que se están tranzando, a modo de dejar todo en términos de pesos chilenos. El problema que se dió con esto es que algunas monedas son un poco complejas de obtener mediante *APIs*, y su adquisición le significó entre 10 a 30 segundos extra de tiempo de procesamiento al sistema, dependiendo de la calidad de la red que utiliza, el cual se puede llegar a reducir de forma sustancial si en cambio se obtuviera una base de datos que ya tuviera cargada esta información. Por temas de tiempo y de priorización de tareas no se logró armar esta base de datos, y tampoco se logró obtener acceso alguna que ya tuviera esta información de antemano, pero será completamente recomendado llevar a cabo este proceso si se quiere mejorar de forma directa el funcionamiento de todas las versiones del sistema que hoy en día existen.

Ahora, en cuanto a cosas que se pueden mejorar con respecto al sistema, el método que se utilizó para paralelizar el algoritmo *Quicksort* en CPU deja bastante que desear, en particular a lo relacionado con la forma en la cual se llena la pila de tareas. Idealmente, el sistema debería ser lo suficientemente inteligente como para llenar esta pila de tareas de forma dinámica, esto es, a medida que realiza las iteraciones del algoritmo, siempre que los subarreglos estén dentro de un cierto largo, estos deberían automáticamente colocarse dentro de la pila de tareas, a modo de que esta se vaya expandiendo dinámicamente, sin embargo, no se logró llevar a cabo esta tarea, razón por la cual la pila se llena a priori de forma secuencial, lo cual reduce el grado de paralelización que posee el sistema, lo cual resulta en un desempeño temporal menor al esperado.

Relacionado con el algoritmo *Quicksort*, otra tarea interesante de llegar a cumplirse sería la capacidad de paralelizar este algoritmo a nivel de GPU, cosa que para datos con una densidad tan grande como los analizados en este proyecto, actualmente no se puede hacer en el *hardware* del cual se dispone, por lo cual, y si consideramos el hecho de que la versión paralela en GPU, sin utilizar un *Quicksort* paralelo, posee tan buen desempeño. Sería interesante ver como la agilización de este algoritmo afectaría en el desempeño del sistema en términos de tiempo de procesamiento, además de ser un aporte interesante para la ciencia en general poder dar con una versión paralela de este algoritmo que funcione independiente del tamaño del arreglo que se le entregue.

Finalmente, una mejora que se podría llegar a hacer sobre este sistema, la cual simplemente no se hizo por temas de tiempo, sería unificar las 4 versiones del mismo en un solo programa, el cual sea capaz de determinar cuál versión es la más apropiada dependiendo de las propiedades, tanto del dispositivo en el cual se corre como de la cantidad de datos a procesar, y dinámicamente sea capaz de seleccionar la versión del mismo a utilizar, cosa que ayudaría de sobremanera a los potenciales usuarios de este tipo de sistema, haciéndolo mucho más atractivo y simple de utilizar por los mismos.

# REFERENCIAS BIBLIOGRÁFICAS

[Amadeo, 2019] Amadeo, K. (2019). Mark to market accounting, how it works, and its pros and cons. https://www.thebalance.com/mark-to-market-accounting-how-it-works-3305942. Visitada: 2019-07-14.

[Barney, 2007] Barney, B. (2007). Introduction to parallel computing. Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/. Visitada: 2019-07-14.

[Brand, 2003] Brand, M. v. (2003). Análisis de la medida de riesgo value at risk (var) y aplicación práctica a los multifondos de pensiones chilenos.

[CBOE, 2019] CBOE, C. B. O. E. (2019). About cboe global markets, inc. https://www.cboe.com/aboutcboe/aboutcboe-main. Visitada: 2019-07-14.

[CMF, 2018] CMF, C. C. (2018). ¿qué son los derivados? https://www.clientebancario.cl/clientebancario/educacion-financiera?articulo=que-son-los-derivados. Visitada: 2019-07-14.

[Cormen *et al.*, 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., y Stein, C. (2009). *Introduction to algorithms*. MIT press.

[Harris, 2013] Harris, M. (2013). Cuda pro tip: Write flexible kernels with grid-stride loops. NVIDIA Developer, https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/. Visitada: 2019-07-14.

[Hoare, 1962] Hoare, C. A. (1962). Quicksort. *The Computer Journal*, 5(1):10–16.

[Hull, 2003] Hull, J. C. (2003). *Options futures and other derivatives*. Pearson Education India.

[Kent State University, 2019] Kent State University, U. L. (2019). Spss tutorials: Pearson correlation. SPSS Tutorials, https://libguides.library.kent.edu/SPSS/PearsonCorr. Visitada: 2019-07-21.

[Mathers, 2013] Mathers, W. S. (2013). A brief history of derivatives. http://www.realmarkits.com/derivatives/3.0history.php. Visitada: 2019-07-14.

[Metropolis y Ulam, 1949] Metropolis, N. y Ulam, S. (1949). The monte carlo method. *Journal of the American statistical association*, 44(247):335–341.

[Ramamoorthy y Li, 1977] Ramamoorthy, C. V. y Li, H. F. (1977). Pipeline architecture. *ACM Computing Surveys (CSUR)*, 9(1):61–102.

[Zeller, 2011] Zeller, C. (2011). Cuda c/c++ basics. NVIDIA Corporation, https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf. Visitada: 2019-07-14.

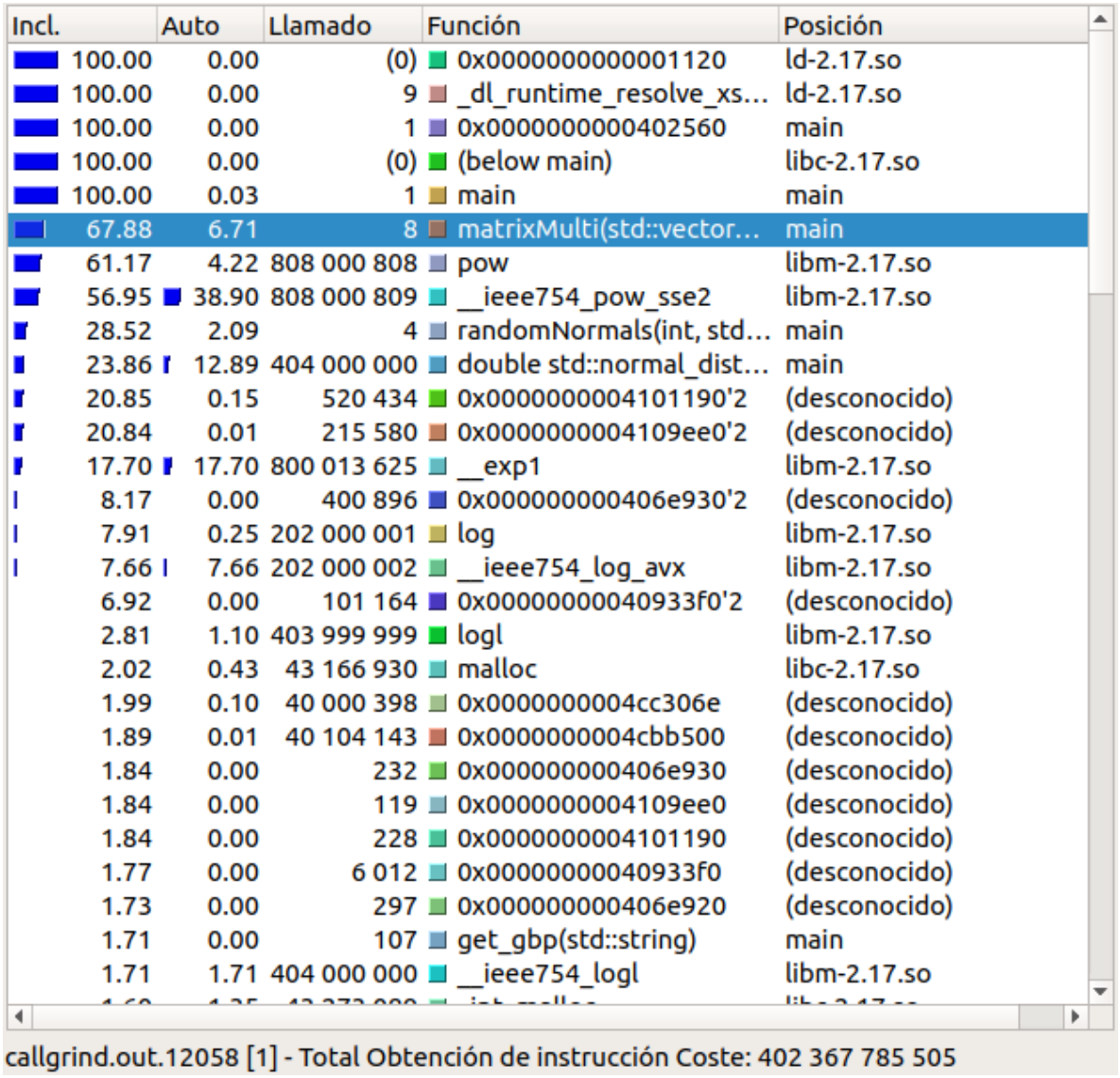
# ANEXOS

## ANEXO A: *Profiling* Versión Secuencial

Como se mencionó en la sección 3.2.5, se realizó un *profiling* sobre la versión secuencial del sistema, a modo de poder determinar las secciones del mismo que requieren mayor atención al momento de paralelizar.

Con tal de lograr esto, se ejecutó el programa con ***Callgrind***, herramienta de *profiling* para C++ la cual registra la cantidad de llamadas que se hacen para cada una de las funciones que se utilizan en un sistema dado, asignando de esta forma un cierto costo para cada una de ella, con lo cual es posible cuantificar el porcentaje de utilización de recursos de *hardware* de cada función que se implementa en un cierto sistema. Más aún, *Callgrind* tiene la particularidad de detectar dependencias en las llamadas a funciones, esto es, si una función hace uso de otra, normalmente se medirá el costo de utilización de recursos de cada una por separado, pero como existe dependencia (al ser una la que invoca a la otra), *Callgrind* toma en cuenta esto al momento de asignar el costo de cada una de estas funciones, incluyendo el costo de la segunda en la medición de la primera. A modo de llevar a cabo este *profiling* se ejecuto la aplicación realizando un total de 1.000.000 simulaciones, cifra que, a pesar de ser relativamente pequeña en comparación a las pruebas que se realizan en el sistema, es la única con la que se logró completar el proceso de *profiling* dentro de un tiempo razonable.

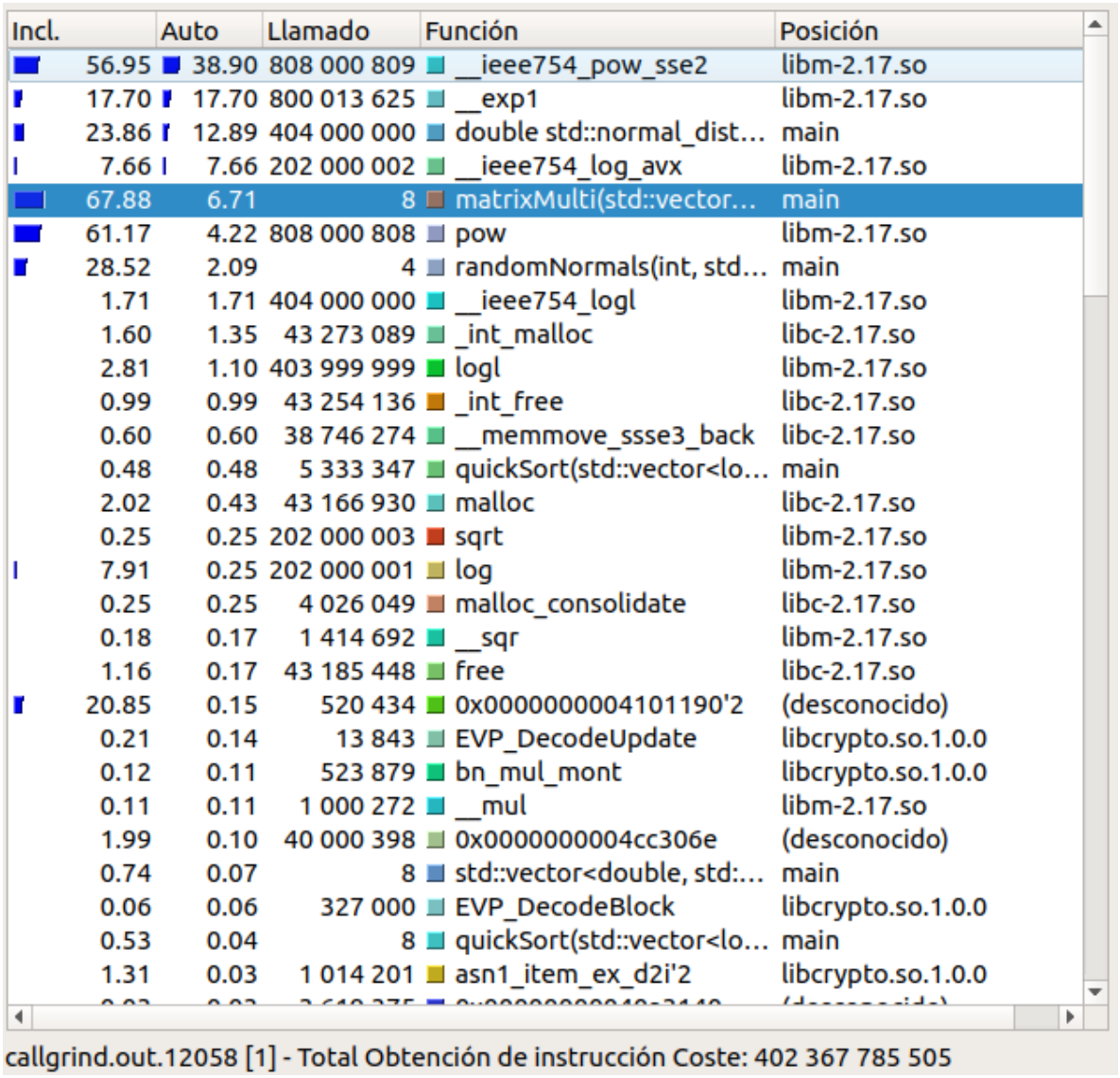
Al realizar el *profiling* mediante esta herramienta se termina generando un archivo **callgrind.out** (el cual se adjuntó con este documento). Dichos archivos requieren de herramientas externas para presentarlos de una forma más comprensible para un usuario, tal como **KCacheGrind**, programa que se utilizó para el análisis de dicho resultado. Se sugiere encarecidamente al lector obtener este programa para que pueda revisar el archivo de *profiling* por sí mismo, pero para su facilidad al momento de leer este documento, se incluyen capturas de pantallas del mismo a modo de dejar más en claro el análisis realizado.



| Incl. | Auto | Llamado | Función | Posición |
|---|---|---|---|---|
| 100.00 | 0.00 | (0) | 0x0000000000001120 | ld-2.17.so |
| 100.00 | 0.00 | 9 | _dl_runtime_resolve_xs… | ld-2.17.so |
| 100.00 | 0.00 | 1 | 0x0000000000402560 | main |
| 100.00 | 0.00 | (0) | (below main) | libc-2.17.so |
| 100.00 | 0.03 | 1 | main | main |
| 67.88 | 6.71 | 8 | matrixMulti(std::vector… | main |
| 61.17 | 4.22 | 808 000 808 | pow | libm-2.17.so |
| 56.95 | 38.90 | 808 000 809 | __ieee754_pow_sse2 | libm-2.17.so |
| 28.52 | 2.09 | 4 | randomNormals(int, std… | main |
| 23.86 | 12.89 | 404 000 000 | double std::normal_dist… | main |
| 20.85 | 0.15 | 520 434 | 0x0000000004101190'2 | (desconocido) |
| 20.84 | 0.01 | 215 580 | 0x0000000004109ee0'2 | (desconocido) |
| 17.70 | 17.70 | 800 013 625 | __exp1 | libm-2.17.so |
| 8.17 | 0.00 | 400 896 | 0x000000000406e930'2 | (desconocido) |
| 7.91 | 0.25 | 202 000 001 | log | libm-2.17.so |
| 7.66 | 7.66 | 202 000 002 | __ieee754_log_avx | libm-2.17.so |
| 6.92 | 0.00 | 101 164 | 0x00000000040933f0'2 | (desconocido) |
| 2.81 | 1.10 | 403 999 999 | logl | libm-2.17.so |
| 2.02 | 0.43 | 43 166 930 | malloc | libc-2.17.so |
| 1.99 | 0.10 | 40 000 398 | 0x0000000004cc306e | (desconocido) |
| 1.89 | 0.01 | 40 104 143 | 0x0000000004cbb500 | (desconocido) |
| 1.84 | 0.00 | 232 | 0x000000000406e930 | (desconocido) |
| 1.84 | 0.00 | 119 | 0x0000000004109ee0 | (desconocido) |
| 1.84 | 0.00 | 228 | 0x0000000004101190 | (desconocido) |
| 1.77 | 0.00 | 6 012 | 0x00000000040933f0 | (desconocido) |
| 1.73 | 0.00 | 297 | 0x000000000406e920 | (desconocido) |
| 1.71 | 0.00 | 107 | get_gbp(std::string) | main |
| 1.71 | 1.71 | 404 000 000 | __ieee754_logl | libm-2.17.so |



Figura 31: *Profiling* Sistema, ordenado según costo acumulado.
Fuente: Elaboración Propia.



| Incl. | Auto | Llamado | Función | Posición |
|---|---|---|---|---|
| 56.95 | 38.90 | 808 000 809 | __ieee754_pow_sse2 | libm-2.17.so |
| 17.70 | 17.70 | 800 013 625 | __exp1 | libm-2.17.so |
| 23.86 | 12.89 | 404 000 000 | double std::normal_dist… | main |
| 7.66 | 7.66 | 202 000 002 | __ieee754_log_avx | libm-2.17.so |
| 67.88 | 6.71 | 8 | matrixMulti(std::vector… | main |
| 61.17 | 4.22 | 808 000 808 | pow | libm-2.17.so |
| 28.52 | 2.09 | 4 | randomNormals(int, std… | main |
| 1.71 | 1.71 | 404 000 000 | __ieee754_logl | libm-2.17.so |
| 1.60 | 1.35 | 43 273 089 | _int_malloc | libc-2.17.so |
| 2.81 | 1.10 | 403 999 999 | logl | libm-2.17.so |
| 0.99 | 0.99 | 43 254 136 | _int_free | libc-2.17.so |
| 0.60 | 0.60 | 38 746 274 | __memmove_ssse3_back | libc-2.17.so |
| 0.48 | 0.48 | 5 333 347 | quickSort(std::vector<lo… | main |
| 2.02 | 0.43 | 43 166 930 | malloc | libc-2.17.so |
| 0.25 | 0.25 | 202 000 003 | sqrt | libm-2.17.so |
| 7.91 | 0.25 | 202 000 001 | log | libm-2.17.so |
| 0.25 | 0.25 | 4 026 049 | malloc_consolidate | libc-2.17.so |
| 0.18 | 0.17 | 1 414 692 | __sqr | libm-2.17.so |
| 1.16 | 0.17 | 43 185 448 | free | libc-2.17.so |
| 20.85 | 0.15 | 520 434 | 0x0000000004101190'2 | (desconocido) |
| 0.21 | 0.14 | 13 843 | EVP_DecodeUpdate | libcrypto.so.1.0.0 |
| 0.12 | 0.11 | 523 879 | bn_mul_mont | libcrypto.so.1.0.0 |
| 0.11 | 0.11 | 1 000 272 | __mul | libm-2.17.so |
| 1.99 | 0.10 | 40 000 398 | 0x0000000004cc306e | (desconocido) |
| 0.74 | 0.07 | 8 | std::vector<double, std:… | main |
| 0.06 | 0.06 | 327 000 | EVP_DecodeBlock | libcrypto.so.1.0.0 |
| 0.53 | 0.04 | 8 | quickSort(std::vector<lo… | main |
| 1.31 | 0.03 | 1 014 201 | asn1_item_ex_d2i'2 | libcrypto.so.1.0.0 |

callgrind.out.12058 [1] - Total Obtención de instrucción Coste: 402 367 785 505

Figura 32: *Profiling* Sistema, ordenado según costo individual.
Fuente: Elaboración Propia.

Es importante señalar que los valores de la tabla **Incl**. pertenecen al costo acumulativo del la función, tomando en cuenta los costos individuales de las funciones a las cuales invoca, mientras que los valores de la columna auto corresponden a los costos independientes de cada una de las funciones. Ambos costos se calculan en base a la cantidad de invocaciones que dicha función tiene durante la ejecución del programa, y para facilitar su interpretación se presentan en términos de porcentaje con relación al total de llamadas. Además, como lo principal de este análisis es determinar qué partes del sistema se van a paralelizar, se pasaran por alto los costos que pertenezcan a funciones sacadas desde librerías de C++.

Al analizar los costos acumulativos vemos que dos funciones destacan por su alto costo, estas

son **matrixMulti** y **randomNormals**, funciones que se encargan, respectivamente, de generar la simulación de escenarios futuros, y de calcular el conjunto de valores aleatorios en distribución normal que se utilizan para dicha simulación.

De inmediato se pensaría en intentar paralelizar ambas funciones, a modo de atacar directamente los dos elementos del programa que consumen la mayor cantidad de recursos, sin embargo, a pesar de que la simulación de escenarios futuros es completamente paralelizable, ya que solo involucra cálculos independientes, la generación de valores aleatorios no resulta tan fácil de abordar, principalmente debido al hecho de que se realiza utilizando funciones de la librería estándar de C++ que no son del todo compatibles con los procesos de paralelización en CPU, y son aún menos compatibles con GPU, debido al hecho de que esta última, al trabajar en lenguaje *kernel*, solo es capaz de entender operaciones matemáticas básicas, por lo tanto, y tras múltiples intentos fallidos, se optó por no paralelizar la generación de valores aleatorios.

Por otro lado, al analizar los costos individuales, se puede observar como la función **quickSort** es la que sigue en términos de costo, sin embargo, su costo es relativamente bajo en comparación a las funciones analizadas anteriormente. A pesar de esto, hay que recordar que estos datos sólo representan una ejecución de 1.000.000 simulaciones, y que, a medida que la cantidad de valores a simular incremente, la cantidad de datos a ordenar también incrementa, por lo que, no solo se puede estar bastante seguro de que al incrementar la cantidad de simulaciones el costo de esta función también incrementa de forma significativa, además, si este proyecto se enfocara en paralelizar únicamente la simulación de valores futuros para una cierta cartera de derivados y se mantuviera el ordenamiento de estos datos de forma secuencial, eventualmente se produciría un cuello de botella en este paso, aumentando de forma significativa el tiempo de procesamiento de este sistema.