Universidad de la República

Facultad de Ingeniería

Aceleración de una herramienta para la toma de decisiones en el mercado eléctrico

Informe de Proyecto de Grado presentado por

Marco Pérez y Camila Irachet

como requisito de graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Ernesto Dufrechou
Pablo Ezzatti

Montevideo, 23 de marzo de 2024

Resumen

SimSEE (Simulador de Sistemas de Energía Eléctrica) es una plataforma diseñada para realizar simulaciones personalizadas de sistemas de generación de energía eléctrica. Los altos costos computacionales que implica su uso motivó explorar el aprovechamiento de plataformas de Hardware masivamente paralelas como las GPUs.

En particular, se lleva a cabo un estudio exhaustivo a nivel de documentación, código y experimentación del proceso de simulación. El objetivo es evaluar qué partes de la rutina pueden ser procesadas de manera más eficiente por una GPU.

En el análisis experimental realizado, se identificó que una de las rutinas más exigente en términos de recursos para esta simulación es el método Simplex, el cual se realiza un gran número de veces (del orden de millones) y en el caso tiene un tamaño considerable, más de 120 variables por 20 restricciones generales y 45 restricciones de cota superior.

Este proyecto tiene como objetivo principal adaptar los algoritmos utilizados en SimSEE para aprovechar las capacidades de cómputo paralelo de las GPUs, centrado especialmente en el método Simplex.

Con base en esta premisa, se exploran diferentes implementaciones del algoritmo Simplex y sus variantes. Posteriormente, se implementan estas variantes en el lenguaje CUDA, se ejecutan en una GPU y se realiza una comparación de rendimiento entre las diferentes versiones en CPU y GPU.

Los resultados obtenidos tras la implementación y comparación de diversas versiones del algoritmo Simplex en la herramienta SimSEE son significativos. Se destaca que todas las variantes implementadas en GPU superaron en rendimiento a las versiones en CPU, siendo Simplex SimSEE la variante más destacada. En particular, en el caso de estudio denominado 'Extra Grande', se observó que el algoritmo Simplex SimSEE adaptado para GPU supera en un 96 % a su versión en CPU. Los resultados resaltan la eficacia del procesamiento en paralelo en GPU para mejorar el rendimiento de SimSEE.

Índice general

Capítulo 1

Introducción

En este capítulo, se proporciona una breve descripción del Sistema de Energía Eléctrica, se explora la utilidad de una herramienta de simulación del sistema eléctrico y se presentan las técnicas en las que se basa la implementación de SimSEE. Además, se explica la motivación detrás del uso de GPUs (Unidades de Procesamiento Gráfico) con el objetivo de introducir al lector en el tema.

El Sistema de Energía Eléctrica se refiere a la infraestructura que permite la generación, transmisión y distribución de energía eléctrica. Es un sistema complejo y vital para la sociedad moderna.

La simulación del sistema eléctrico es una herramienta fundamental para comprender su comportamiento y analizar diferentes escenarios. Permite evaluar el rendimiento, la eficiencia y la seguridad del sistema, así como diseñar y optimizar su operación (SimSEE, 2020).

La implementación de SimSEE se basa en diversas técnicas, como modelos matemáticos y algoritmos, que permiten simular el comportamiento del sistema eléctrico. Estos modelos y algoritmos capturan las interacciones entre los componentes del sistema, como generadores, líneas de transmisión, transformadores y cargas.

Además de su funcionalidad y utilidad, es importante considerar el aspecto del costo computacional al utilizar herramientas de simulación como SimSEE. A medida que las simulaciones se vuelven más precisas y complejas, la carga computacional también aumenta. Por ejemplo, a medida que se busca lograr una mayor precisión en el modelado del sistema eléctrico, se requieren más cálculos, lo que puede incrementar significativamente el tiempo de ejecución de las simulaciones. Del mismo modo, cuando se intenta simular escenarios más complejos, como la incorporación de nuevos factores en el sistema eléctrico, los recursos computacionales necesarios aumentan. Este crecimiento puede traducirse en mayores costos de tiempo y recursos, lo que afecta la eficiencia general de la simulación y limita la capacidad de analizar múltiples escenarios en un tiempo razonable.

Es en este punto donde la paralelización es una solución para aprovechar las capacidades de procesamiento en paralelo de las GPUs, permitiendo abordar de

manera más eficiente la creciente carga de trabajo, acelerando considerablemente los tiempos de simulación y reduciendo los costos computacionales asociados.

La motivación para utilizar GPUs en el contexto de SimSEE radica en su capacidad para realizar cálculos paralelos de manera eficiente. Las GPUs están diseñadas para procesar grandes volúmenes de datos y ejecutar operaciones simultáneas en múltiples núcleos. Al aprovechar esta capacidad de cómputo paralelo, es posible acelerar significativamente la simulación del sistema eléctrico y reducir el tiempo de ejecución.

1.1. Sistemas de Energía Eléctrica

Un sistema de energía eléctrica es una infraestructura diseñada para generar, transmitir, distribuir y suministrar energía eléctrica a los consumidores finales. Estos sistemas están compuestos por una red interconectada de componentes y equipos que trabajan en conjunto para asegurar un suministro constante y confiable de electricidad a hogares, industrias, comercios y otros usuarios. Los principales componentes de un sistema de energía eléctrica son:

- Generación: es el proceso de convertir diferentes fuentes de energía en electricidad. Esto puede lograrse mediante centrales eléctricas que utilizan diversas fuentes como carbón, gas natural, petróleo, energía hidroeléctrica, energía eólica, energía solar, etc.
- Transmisión: la electricidad se transporta a largas distancias a través de líneas de transmisión de alta tensión para minimizar las pérdidas de energía. Estas líneas pueden ser torres o cables subterráneos.
- Distribución: En esta etapa, la electricidad se reduce de alta a media tensión y se distribuye a través de subestaciones y líneas de distribución para abastecer a áreas más pequeñas, como barrios y comunidades.
- Subestaciones: Son instalaciones que regulan y transforman los niveles de voltaje de la electricidad en diferentes puntos del sistema. Estas también se utilizan para conectar fuentes de energía renovable y otros generadores a la red.
- Red de Distribución Local: Esta parte del sistema entrega la electricidad directamente a los hogares, empresas y otros usuarios finales a través de líneas de baja tensión.
- Consumidores: Los usuarios finales de la electricidad, como hogares, comercios, industrias y servicios públicos, reciben la energía eléctrica y la utilizan para una variedad de propósitos, como iluminación, calefacción, refrigeración, operación de equipos y más.

Los sistemas de energía eléctrica son esenciales para el funcionamiento de la sociedad moderna, ya que la mayoría de los dispositivos y tecnologías cotidianas

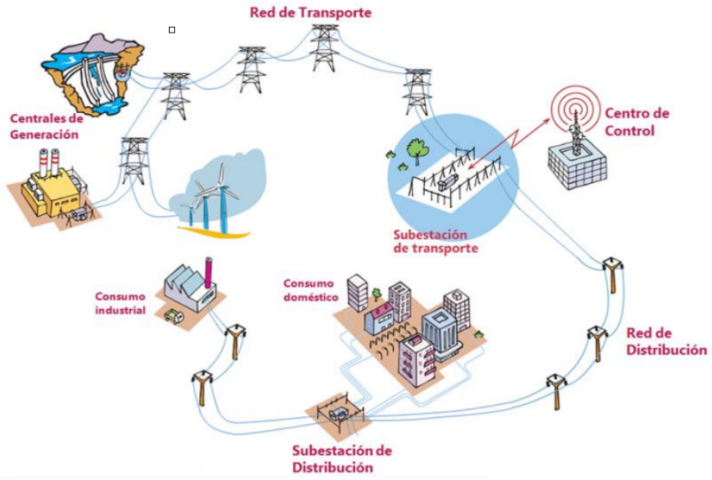


Figura 1.1: Generación, transmisión y distribución de energía eléctrica. Extraído de (CFN, Corporación Financiera Nacional, 2017)

dependen de la electricidad. La planificación, operación y mantenimiento efectivos de estos sistemas son cruciales para garantizar un suministro confiable y seguro de energía eléctrica.

La generación y distribución de electricidad en Uruguay esta a cargo de la empresa estatal UTE (Administración Nacional de Usinas y Trasmisiones Eléctricas). UTE posee una variedad de instalaciones de generación de energía renovable, incluyendo centrales hidroeléctricas, parques eólicos y plantas solares. Estas fuentes de energía son respaldadas por una capacidad de generación térmica constante para garantizar un suministro estable. Además, UTE se beneficia de la electricidad generada por la Central Hidroeléctrica de Salto Grande, que es un proyecto conjunto entre Argentina y Uruguay, tiene acuerdos a largo plazo con proveedores privados de energía renovable y compra electricidad en el mercado abierto según sea necesario y, para aumentar la flexibilidad de su sistema eléctrico, también está conectada a Argentina a través de la Estación de Salto Grande y a Brasil mediante estaciones de conversión de frecuencia en Rivera y Melo, lo cual fortalece la capacidad de Uruguay para gestionar su suministro eléctrico. (Portal UTE, 2023).

Uruguay obtiene una gran parte de su electricidad de fuentes renovables, como la energía hidroeléctrica, eólica y solar. En 2008, en Uruguay se lanzó una estrategia energética para el período 2005-2030, donde se marcó un enfoque a largo plazo centrado en la diversificación de fuentes de generación y abastecimiento, incorporando la energías renovables y mejorando la eficiencia en el consumo de energía.

Esta estrategia permitió lograr en poco tiempo la transición hacia una ge-

neración de energía eléctrica libre de carbono. Entre los años 2017 y 2021, las fuentes renovables representaron en promedio el 94 % en la matriz eléctrica del país. De este porcentaje, un 52 % provino de fuentes eólicas, solares y de biomasa, mientras que un 42 % fue generado por energía hidroeléctrica. Esta transformación tuvo un efecto significativo en la reducción de emisiones de gases de efecto invernadero originados en el sector energético (Uruguay XXI, Octubre 2022).

1.2. Utilidad de una herramienta de simulación

La simulación en general se lleva a cabo con diversas motivaciones. Una de ellas es comprender mejor cómo funcionan los sistemas y los procesos, lo que ayuda, por ejemplo, a los planificadores y operadores (en este caso, el sistema eléctrico) en diferentes escenarios. También se utiliza para optimizar operaciones, es decir, encontrar la mejor manera de operar un sistema eléctrico o un mercado en tiempo real, por ejemplo, optimizando la generación y distribución de electricidad para satisfacer la demanda de manera eficiente y al menor costo posible.

La simulación también facilita la identificación y evaluación de riesgos, lo que a su vez ayuda a desarrollar estrategias de mitigación. Además, se utiliza en la planificación a largo plazo y en la evaluación del impacto de diferentes políticas energéticas y regulaciones en el mercado eléctrico. Esto puede incluir la introducción de fuentes de energía renovable, cambios en las tarifas eléctricas y la implementación de medidas de eficiencia energética.

Simular un sistema de generación de energía eléctrica supone simular la operación futura del sistema incorporando distintos tipos de pronósticos. Esto se realiza en base a un tiempo, que dependiendo del tipo de estudio que se quiera realizar, puede ser del orden de años (para planificaciones de largo plazo), meses (planificaciones de mediano plazo) o lapsos de tiempo de días y horas (consideradas como operaciones de corto plazo).

Un Simulador del Sistema evalúa cuáles serán los valores de determinadas variables durante la operación del sistema permitiendo, probar las diferentes posibilidades de operaciones evaluando sobre el modelo las consecuencias asociadas a cada elección.

Existen métodos que se utilizan para resolver problemas de optimización complejos como lo es este de simular la generación de energía eléctrica. Lo que se busca es encontrar una regla que permita decidir cuándo utilizar cada uno de los recursos del sistema, y en qué cantidad, para lograr minimizar una función de costo objetivo, que puede ser por ejemplo el costo de ingresos a minimizar o maximizar.

1.3. SimSEE

En el 2007 surge la implementación (Simulador de Sistema de Energía Eléctrica), en el marco del proyecto de desarrollo tecnológico PDT-47-12 financiado

por el BID, bajo la dirección del Dr. Ing. Gonzalo Casaravilla y con el apoyo de Facultad de Ingeniería, MIEM, URSEA, ADME y UTE. (Chaer., 2008) (Ings. Felipe Palacio, 2019)

La herramienta SimSEE permite la simulación y optimización de sistemas de energía eléctrica incorporando distintos tipos de pronósticos, como la generación de energía renovable, demanda eléctrica, precios de las tecnologías y combustibles, entre otras cosas. De esta forma se puede analizar por ejemplo la rentabilidad de proyectos específicos, cálculos de precios y riesgos, por lo que SimSEE brinda elementos imprescindibles para la toma de decisiones en forma eficiente en el sector de generación de energía eléctrica.

Desarrollada íntegramente en el lenguaje de programación Pascal, esta herramienta es de código abierto y libre, lo que significa que su código fuente está disponible para el público. Además, es compatible con múltiples plataformas, permitiendo su ejecución tanto en sistemas operativos Windows como en Linux.

La implementación adoptó la formulación recursiva de la función CF (Costo Futuro). El Costo Futuro de una etapa k, es el costo de operar del sistema desde la etapa k (un estado o instante conocido) hasta el final. La función de Costo Futuro se puede plantear como: $CF(k) = ce(k) + CF(k+1)$, donde $ce(k)$ representa el costo directo en la etapa k.

Se puede decir que las "formulaciones recursivas"son herramientas o métodos que ayudan a abordar o analizar situaciones que involucran eventos aleatorios o inciertos. Se descompone el problema en etapas sucesivas, y en cada etapa, se asume que se conoce toda la información pasada relacionada con el proceso estocástico, es decir, lo que ha ocurrido hasta ese momento. Sin embargo, lo que sucederá en el futuro a partir de esa etapa se considera desconocido o incierto.

La Programación Dinámica Estocástica (SDP, siglas en inglés), es la técnica en la que se basa SimSEE. En SDP, la optimización se realiza de forma recursiva, asumiendo conocidos los valores de la función CF en la última etapa del horizonte de tiempo estudiado y resolviendo paso a paso desde el futuro hacia el presente. Uno de los problemas de esta técnica es el crecimiento exponencial de la cantidad de variables de estado (conocido como la maldición de la dimensionalidad de Bellman), lo que hace que el algoritmo SDP no sea aplicable a sistemas con muchas variables de estado (Bellman, 1957). Estas variables de estado representan la información necesaria para describir completamente el estado de un sistema en un momento dado. Son aquellas que evolucionan con el tiempo y se utilizan para tomar decisiones óptimas o predecir el comportamiento futuro del sistema. El fenómeno de Bellman, consiste en que a medida que aumentamos el número de dimensiones, el volumen del espacio crece exponencialmente y provoca que los datos estén más dispersos, lo que produce que la cantidad de datos de entrenamiento necesarios para extraer resultados fiables crezca también exponencialmente. Cuando tenemos muchas dimensiones, o variables, surgen problemas a nivel estadístico. Se desea que el modelo sea siempre lo más preciso posible, pero al tener tantas variables algunas pueden dar características poco importantes para el modelo (aportando poco a la creación del modelo y a la solución del problema).

En la siguiente ilustración se muestra cómo evoluciona de forma general la eficiencia de los algoritmos de aprendizaje automático con el aumento de las dimensiones. Se observa que se alcanza un máximo en la eficiencia con un número concreto de dimensiones y a medida que aumentamos las dimensiones el rendimiento empieza a decaer.

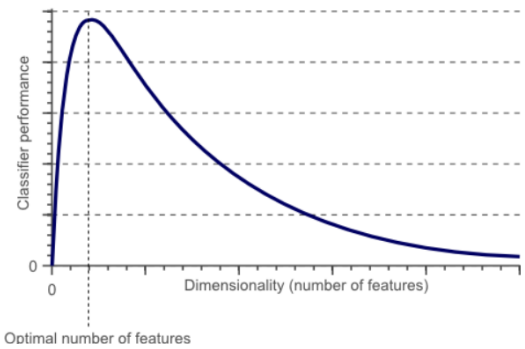


Figura 1.2: Gráfica relación dimensionalidad eficiencia.Extraído de AI Planet (2023)

Una solución posible es la Descomposición, que permite abordar problemas complejos dividiéndolos en partes más manejables o componentes más simples (por lo que el algoritmo va a tener menos problemas). Lo que se hace es reducir las dimensiones de los datos al crear nuevos valores que representan a múltiples dimensiones (llegando a tener datos que son abstractos) pero que se comportan de forma muy similar a los originales, teniendo menor cantidad de dimensiones. Existen muchas técnicas de descomposición que usan diferentes estrategias para llevar a cabo la reducción de dimensionalidad. El Análisis de Componentes Principales (PCA, por sus siglas en inglés, Principal Component Analysis) es relevante en el contexto de la descomposición de problemas que involucran múltiples variables de estado. Es una técnica de análisis de datos que se utiliza comúnmente en estadísticas y aprendizaje automático para reducir la dimensionalidad de un conjunto de datos. PCA permite identificar las combinaciones lineales de variables originales que explican la mayor variabilidad en los datos. Al proyectar los datos en un espacio de menor dimensionalidad, es posible reducir la cantidad de variables de estado mientras se retiene la mayor cantidad posible de información relevante. Esto simplifica el problema y lo hace más tratable. (Joaquín Amat Rodrigo, Junio, 2017)

El método conocido como Programación Dinámica Estocástica Dual (SDDP, siglas en inglés), es una resolución del problema mediante aproximaciones sucesivas que evitan calcular CF (Costo Futuro) sobre una discretización del espacio de estado. En SDDP, el problema original se divide en etapas y escenarios, lo que reduce enormemente el costo computacional de resolución, sin embargo, para asegurar la convergencia y eficiencia del algoritmo se requiere que la dependencia temporal (o entre etapas) de las variables aleatorias sea (o se modele)

lineal.

La ventaja está en que por el planteo del problema, los valores de CF (Costo Futuro) de cada etapa son funciones lineales a tramos y son entonces rápidamente aproximadas por el algoritmo SDDP. Esto elimina la necesidad de discretizar las variables de estado y de calcular el Costo Futuro para cada punto de la discretización evitando así la Maldición de la Dimensionalidad que sufre la SDP.

1.4. Motivación del uso de GPUs

La motivación para utilizar GPUs en la herramienta SimSEE es impulsada por la necesidad de mejorar significativamente el rendimiento y la eficiencia de la simulación eléctrica.

El costo computacional en el contexto de SimSEE, o cualquier otra herramienta de simulación y análisis de sistemas eléctricos, puede ser un problema dado que las simulaciones eléctricas a menudo involucran una gran cantidad de datos y cálculos complejos. La latencia introducida por la transferencia de datos entre componentes de la simulación, así como también mantener la sincronización y la coherencia de los datos entre los componentes de la simulación, puede ser un desafío, especialmente cuando se trabaja con sistemas en tiempo real. Los problemas de sincronización pueden llevar a resultados incorrectos o incoherentes.

La decisión de aprovechar una GPU en SimSEE se basa en la hipótesis de que la herramienta tiene el potencial de paralelismo suficiente para aprovechar al máximo el poder de procesamiento de una GPU. Algunas de las razones por las que se piensa que SimSEE puede tener paralelismo suficiente para explotar una GPU son:

1. Cálculos intensivos: SimSEE realiza cálculos intensivos en términos computacionales, especialmente al simular sistemas eléctricos de gran tamaño. Estos cálculos suelen involucrar operaciones matemáticas complejas y la manipulación de grandes conjuntos de datos.

2. Operaciones en paralelo: Muchas de las operaciones realizadas en la simulación de sistemas eléctricos se pueden realizar en paralelo. Esto significa que se pueden dividir en tareas más pequeñas e independientes que se pueden ejecutar simultáneamente en una GPU, lo que acelera significativamente el proceso de simulación.

3. Escalabilidad: A medida que se aumenta la complejidad de la simulación o se trabaja con sistemas eléctricos más grandes, la necesidad de paralelismo se vuelve aún más necesaria. Las GPUs ofrecen una considerable capacidad de procesamiento, lo que las hace adecuadas para manejar simulaciones grandes.

Considerando lo anterior, la distribución de los cálculos utilizando GPUs puede permitir realizar simulaciones de realidades más grandes que las actuales, acompañando la expansión del mercado eléctrico, así como también realizar simulaciones con mayor nivel de precisión y en tiempo más cortos.

1.5. Objetivos

El objetivo principal de este proyecto es desarrollar un prototipo y llevar a cabo un análisis cualitativo que permita estimar las ventajas potenciales de la integración de GPUs en el contexto de SimSEE. Para lograr este objetivo, se han establecido los siguientes objetivos específicos:

1. Realizar un estudio exhaustivo de SimSEE para comprender en profundidad su funcionamiento.
2. Evaluar el rendimiento de la herramienta en diversos escenarios, identificando los procesos computacionales más costosos.
3. Relevar el estado del arte relacionado con el problema que se busca resolver.
4. Desarrollar algoritmos y técnicas de programación específicas para aprovechar al máximo la capacidad de procesamiento paralelo de las GPUs.
5. Mejorar significativamente la eficiencia de estos procesos mediante la utilización de plataformas altamente paralelas, como las GPUs.
6. Realizar pruebas exhaustivas y comparativas de rendimiento entre la versión actual de SimSEE y la versión en GPU para una variedad de casos de uso y escenarios de simulación.
7. Investigar y aplicar estrategias de administración de datos eficientes para minimizar la transferencia de datos entre la CPU y la GPU, reduciendo así la latencia y optimizando el uso de la memoria.

Capítulo 2

Conceptos preliminares

En este capítulo del informe, se abordarán conceptos preliminares que son fundamentales para comprender el estudio y análisis del modelo SimSEE. En primer lugar, se explicará en detalle el modelo SimSEE y su aplicación en el campo de la optimización. Este modelo se basa en el método Simplex, que es un algoritmo ampliamente utilizado para encontrar soluciones óptimas en problemas lineales y no lineales. Se examinarán diferentes variantes de este método como el Simplex clásico, Simplex Big M y Simplex Two Phases, con el fin de comprender su funcionamiento y aplicaciones prácticas.

A continuación, se explorarán los conceptos relacionados con las GPUs (Unidades de Procesamiento Gráfico). Se describirá la arquitectura de hardware de una GPU, incluyendo los componentes clave y su funcionamiento. Además, se introducirá el modelo de programación para GPUs, que permite aprovechar el paralelismo masivo y la capacidad de procesamiento de estas unidades en aplicaciones computacionales intensivas.

Otro aspecto importante a considerar son las herramientas de profiling. Estas herramientas permiten analizar y medir el rendimiento de un programa o sistema en términos de uso de recursos, tiempos de ejecución y eficiencia. Se examinarán diferentes herramientas de profiling utilizadas en el desarrollo con el objetivo de obtener información detallada sobre el desempeño y posibles cuellos de botella.

Finalmente, se presenta el trabajo relacionado en el campo de la optimización y el uso de GPUs.

Este capítulo de conceptos preliminares sienta las bases teóricas y prácticas necesarias para comprender el funcionamiento y aplicaciones del sistema SimSEE, así como su integración con GPUs y técnicas de optimización. A través de un estudio exhaustivo de estos conceptos, se sentará el marco necesario para el análisis posterior del sistema y el desarrollo de soluciones óptimas.

2.1. El modelo SimSEE

Los componentes principales de SimSEE incluyen:

1. Actores: Estos son los elementos fundamentales que representan los diferentes componentes del sistema eléctrico, como generadores térmicos, eólicos, solares, hidráulicos y otros. Cada actor tiene su propio comportamiento y características específicas.

2. Sala de Juego: Es el entorno de simulación que se crea mediante un archivo de texto. La sala de juego o sala es un archivo de texto donde se describe el sistema eléctrico a simular y optimizar. En este archivo se especifican datos como la cantidad y tipo de generadores, las demandas, mantenimiento de los generadores, pronósticos, entre otros.

3. Optimización: Esta etapa se centra en la búsqueda de la "Política Óptima de Operación (POO)", que implica la construcción de una función de costo futuro basada en el estado actual y el tiempo. La POO es esencial para tomar decisiones informadas durante la simulación.

4. Simulación: Durante esta fase, se ejecutan simulaciones basadas en la POO para explorar diferentes escenarios y posibles realizaciones de procesos estocásticos que afectan al sistema, como la variabilidad de la velocidad del viento, la radiación solar y las posibles averías en las máquinas.

SimSEE permite crear simuladores a medida de un sistema de generación de energía eléctrica, simplemente agregando los diferentes tipos de actores a una Sala de Juego. Estos actores se comportan en la Sala de acuerdo con su tipo, y la cantidad de variables de estado de una sala está dada por la suma de la cantidad de variables de estado de sus fuentes y actores.

En la determinación de la operación óptima de un sistema con SimSEE se pueden distinguir dos procesos, Optimización y Simulación. Durante la Optimización se resuelve el problema de encontrar "La Política Óptima de Operación (POO)", que minimice los costos y maximice la eficiencia del sistema eléctrico. En particular, el método Simplex se puede aplicar en esta etapa con el objetivo de encontrar la POO resultante es esencial para la toma de decisiones informadas durante la siguiente etapa, donde se utiliza la POO para ejecutar múltiples simulaciones, y así evaluar el rendimiento del sistema bajo diferentes condiciones y escenarios.

La simulación de una Sala puede ser de una o más crónicas, y para que pueda ser ejecutada es necesario que previamente se haya ejecutado la etapa de Optimización de la misma Sala. Cuando se habla de "crónicas", se refieren a las secuencias de etapas de Optimización/Simulación que se realizan en SimSEE. Es la corrida completa en SimSEE, que incluye tanto la etapa de Optimización como múltiples simulaciones en la etapa de Simulación. Esto permite evaluar el comportamiento del sistema en diversos contextos y condiciones.

2.1.1. Funciones del Simplex de SimSEE

A continuación se explican algunas de las funciones destacadas del Simplex de SimSEE. Las funciones descritas forman parte de la clase $TSimplex$ en la unidad $usimplex$:

1. PASOBUSCARFACTIBLEIGUALDAD4

   Recibe como parámetros $IgualdadesNoResueltas$ (tipo entero), $nCeros-Filas$ y $nCerosCols$ (arrays de enteros pasados por referencia). Busca el elemento con mayor valor absoluto en la caja entre $cnt\_RestriccionesRedu-ndantes + 1$ y $cnt\_RestriccionesRedundantes + nIgualdadesNoResueltas$. A su vez cuenta la cantidad de ceros en filas y columnas para cada valor en esa caja. Luego hace una segunda pasada comparando la cantidad de ceros en filas y columnas para los valores relativamente cercanos al máximo encontrado, si algún valor esta cerca y tiene más ceros será elegido como pivote. Guarda la cantidad de ceros en filas y columnas $nCerosFilas$ y $nCerosCols$ y devuelve el entero 1 en caso de que haya encontrado un valor distinto de cero para utilizar como pivote y -1 en caso contrario.

2. INTERCAMBIAR

   Recibe como parámetros dos enteros $kfil$ y $jcol$. Intercambia la fila que se encuentre en $kfil$ por la columna que se encuentre en $jcol$ utilizando como pivote el valor que se encuentra en $(kfil, jcol)$

3. LOCATE_ZPOS

   Busca la columna que en la última fila (fila z) tenga el valor positivo más grande. Para eso recorre todas las columnas exceptuando las ya fijadas observando el valor en el lugar $kfila_z$ (entero que se recibe como parámetro en la función). Devuelve un entero que es el índice de la columna del máximo valor positivo en el lugar dado por $kfila_z$. Si ningún valor es positivo, devuelve -1.

4. MEJORPIVOTE

   Esta función toma como parámetros q y $kmax$ enteros, $filaFantasma$ y $colFantasma$ de tipo booleano pasados por referencia y $checkearFilaOpt$ también booleano. La variable q es el índice de la columna candidata a pivotear con una fila, $kmax$ es el índice máximo de fila con la cual se puede pivotear, ya que en la resolución del simplex, las restricciones (filas) ya resueltas se ordenan al final de la matriz. La función recorre las filas entre 1 y$kmax$ - 1 y busca la mejor fila para pivotear, el resultado es el índice (p) de fila o -1 si no hay ningún candidato a pivote $a(p, q)$ que sea negativo. La búsqueda se realiza entre las filas 1 y $kmax$ - 1. Si la fila en $kmax$ es una variable con restricción de cota superior, revisa también que esta no se este violando.

5. CAMBIO_VAR_COTA_SUP_EN COLUMNA

Esta función acepta un parámetro q de tipo entero, que representa el índice de una columna. Cuando esta columna está asociada a una variable con cota superior, la función realiza una operación que implica cambiar la variable correspondiente en esa columna. Como resultado de esta operación, la función devuelve un valor booleano que indica si se ejecutó exitosamente el cambio de variable o no.

2.2. Problema de programación lineal - Método Simplex

Los problemas de programación lineal (PL) consisten en maximizar (o minimizar) una función objetivo lineal sujeta a un conjunto de restricciones lineales. Más formalmente, consideramos el siguiente problema:

$$\begin{aligned} \text{máx } & x_0 = cx \\ \text{s.t. } & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{2.2}$$

Donde:

$$c = (c_1, c_2, ..., c_n) \in \mathbb{R}^n$$

$$\text{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \in \mathbb{R}^{mxn}$$

$$x = (x_1, x_2, ..., x_n)^T$$

n y m son el número de variables y restricciones, respectivamente.

El Método Simplex, publicado en 1974 por George Dantzig resuelve el problema de programación lineal 2.2, (Hillier F.S., 1991) y consiste en un algoritmo iterativo que secuencialmente se va aproximando al óptimo del problema de Programación Lineal en caso de existir este último.

Es un método de resolución de problemas de PL que parte de una primera solución factible de punto extremo (solución que se encuentra en uno de los vértices del conjunto factible del problema de PL y no se puede mejorar en términos de la función objetivo sin abandonar el conjunto factible) y se desplaza a otra solución factible utilizando manipulaciones de matrices llamadas operaciones de pivoteo, de tal manera que aumenta continuamente el valor objetivo.

Un método alternativo de reaolver problemas de PL sencillos es el método gráfico que permite visualizar las soluciones factibles y encontrar la solución

óptima al problema. La idea principal de este método es representar gráficamente las restricciones del problema en un plano cartesiano (2D), donde los ejes representan las dos variables de decisión. Cada restricción se dibuja como una línea o un límite en el plano, y la región donde todas las restricciones se superponen es el conjunto factible, es decir, el conjunto de soluciones que cumplen con todas las restricciones del problema. Luego, se traza la función objetivo en el mismo plano y la solución óptima se encuentra en el punto donde esta línea intersecta el conjunto factible. Por lo tanto, este método es una técnica visual para resolver problemas de PL con dos variables, pero se vuelve ineficiente para problemas con un mayor número de variables, para los cuales se utiliza el Método Simplex.

El problema Simplex que necesita resolver SimSEE contiene además de las restricciones de $\leq$, restricciones de $\geq$. Es por esto que, en la siguiente sección, además de los conceptos fundamentales y una ilustración del método Simplex, se presentan sus dos variantes Big M y Two Phases que extienden el algoritmo Simplex básico para considerar en sus cálculos estas restricciones de $\geq$.

2.2.1. Descripción del algoritmo Simplex

El Método Simplex hace uso de la propiedad de que la solución óptima de un problema de Programación Lineal se encuentra en un vértice o frontera del dominio de puntos factibles (esto último en casos muy especiales), por lo cual, la búsqueda secuencial del algoritmo se basa en la evaluación progresiva de estos vértices hasta encontrar el óptimo. Para aplicar el Método Simplex a un modelo lineal, este debe estar en un formato especial conocido como formato estándar.

El modelo debe cumplir las siguientes condiciones:

1. El objetivo es de la forma de maximización o de minimización.
2. Todas las restricciones son de igualdad.
3. Todas las variables son no negativas.
4. Las constantes a la derecha de las restricciones son no negativas.

El algoritmo Simplex comienza con un problema de PL que se define mediante una función objetivo lineal (que queremos maximizar o minimizar) y un conjunto de restricciones lineales que limitan las variables de decisión. En un espacio de n dimensiones (donde n es el número de variables de decisión), el conjunto de soluciones factibles (puntos que cumplen con todas las restricciones) forma un poliedro convexo (conjunto de puntos que satisface propiedades de convexidad y linealidad). El algoritmo Simplex se inicia en uno de los vértices de este poliedro, que se llama punto extremo (como se explicó anteriormente). El algoritmo Simplex se mueve de un punto extremo a otro, mejorando continuamente el valor objetivo en cada paso. Para hacerlo, realiza operaciones de pivoteo que cambian el conjunto de variables básicas (variables que no son cero) y no básicas (variables que son iguales a cero) para encontrar un nuevo punto

extremo que mejore la función objetivo. El algoritmo continúa iterando hasta que no puede mejorar más la función objetivo, momento en el cual se detiene y se obtiene la solución óptima.

En su forma algebraica original, tomamos el siguiente problema de ejemplo:

Maximizar:

$$z = 7x_1 + 4x_2$$

Sujeto a:

$$\begin{aligned} 2x_1 + x_2 + s_1 &= 20 \\ x_1 + x_2 + s_2 &= 18 \\ 2x_1 + s_3 &= 8 \end{aligned}$$

Dado que la función objetivo y las restricciones de no negatividad no participan explícitamente en la mecánica del procedimiento de solución, solo necesitamos presentar los coeficientes en las ecuaciones de restricción. En forma tabular, este problema se representará de la siguiente manera:

	x_1	x_2	s_1	s_2	s_3	
z	-7	-4	0	0	0	0
s_1	2	1	1	0	0	20
s_2	1	1	0	1	0	18
s_3	1	0	0	0	1	8

Así, cada ecuación de restricción se traduce en una fila de coeficientes; y los coeficientes de una variable en diferentes ecuaciones se enumeran en la misma columna, con el nombre de esa variable especificado en la parte superior de esa columna como encabezado. Además, para facilitar la lectura, hemos delimitado la tabla en varias áreas, por ejemplo, (i) la columna z a la izquierda, (ii) los coeficientes en la ecuación objetivo en la fila superior y (iii) constantes del lado derecho de las ecuaciones en la columna más a la derecha.

La tabla anterior se denominará tabla Simplex inicial. Para simplificar las declaraciones, nos referiremos a las filas sucesivas del cuadro como R0, R1, etc.; esta numeración, por supuesto, corresponde a la de las ecuaciones originales. Además, nos referiremos a la última columna como la columna RHS (ya que proviene de las constantes del lado derecho de las ecuaciones).

Asociadas con este cuadro inicial, las variables no básicas son x_1 y x_2 y las variables básicas son s_1, s_2 y s_3. Por lo tanto, la solución factible básica inicial (o actual) es: (x_1, x_2, s_1, s_2, s_3) = (0, 0, 20, 18, 8). Esta solución tiene un valor de función objetivo 0, que es el número más a la derecha en R0.

Dado que los coeficientes de x_1 y x_2 (las variables no básicas) en la primera fila son negativos (positivos en el caso de minimización), la solución actual no es óptima. Además, dado que el coeficiente de x_1, es decir -7, es más negativo

que el de x_2 (más positivo en el caso de minimización), seleccionaremos a x_1 como la variable de entrada.

Ahora nos vamos a referir a la columna x_1 como la columna pivote.

Para determinar el aumento máximo posible en x_1, realizamos una prueba de razón (evaluación que se realiza para determinar cuánto puede aumentar una variable antes de que se violen restricciones, lo que ayuda a decidir qué variable debe ingresar a la base en el siguiente paso del algoritmo). La prueba de razón involucrará los coeficientes en la columna pivote y en la columna que se encuentra más a la derecha . Esto se resuelve en el margen derecho del cuadro, como se muestra a continuación.

variables	z	x_1	x_2	s_1	s_2	s_3		prueba	
básicas	1	-7	-4	0	0	0	0	razón	
s_1	0	2	1	1	0	0	20	20/2	
s_2	0	1	1	0	1	0	18	18/1	
s_3	0	1	0	0	0	1	8	8/1	(mínimo)

Se toma el 1 como el elemento pivote ya que 8/1 es el mínimo valor. La elección del mínimo valor positivo entre las razones (razón de cambio) asegura que al aumentar la variable entrante, la función objetivo se mejore de la manera más eficiente posible sin exceder los límites de las restricciones. Esto muestra que la nueva fila pivote será la R3 (0,1,0,0,0,1,8), y la variable básica asociada con esa fila, s_3, será la variable saliente.

Luego se hace una combinación lineal que consiste en la actualización de las filas para lograr que los elementos en la columna del elemento pivote, excepto el elemento pivote, se vuelvan cero. Esta operación garantiza que el algoritmo Simplex siga explorando el espacio de soluciones factibles y se acerque a la solución óptima.

Luego de la combinación lineal la tabla queda de la siguiente forma:

z	x_1	x_2	s_1	s_2	s_3	
1	0	-4	0	0	7	56
0	0	1	1	0	-2	4
0	0	1	0	1	-1	10
0	1	0	0	0	1	8

Como todavía quedan valores negativos dado que el coeficiente de x_2 (variable no básica) en la primera fila es negativo, la solución actual no es óptima. Por lo tanto se repite el procedimiento anterior, seleccionando x_2 como variable saliente.

variables	z	x_1	x_2	s_1	s_2	s_3		prueba	
básicas	1	0	-4	0	0	7	56	razón	
x_2	0	0	1	1	0	-2	4	4/1	(mínimo)
s_2	0	0	1	0	1	-1	10	10/1	
x_1	0	1	0	0	0	1	8	8/0	

Luego se hace la combinación lineal y la tabla queda de la siguiente forma:

z	x_1	x_2	s_1	s_2	s_3	
1	0	0	4	0	-1	72
0	0	1	1	0	-2	4
0	0	0	-1	1	1	6
0	1	0	0	0	1	8

El algoritmo sigue ya que hay valores negativos

variables	z	x_1	x_2	s_1	s_2	s_3		prueba	
básicas	1	0	0	4	0	-1	72	razón	
x_2	0	0	1	1	0	-2	4	4/-2	(mínimo)
s_3	0	0	0	-1	1	1	6	6/1	
x_1	0	1	0	0	0	1	8	8/1	

Como todavía quedan valores negativos, se sigue con el algoritmo hasta que ya no haya valores negativos en la función objetivo, llegando finalmente a la siguiente tabla:

z	x_1	x_2	s_1	s_2	s_3	
1	0	0	3	1	0	78
0	0	1	-1	2	0	16
0	0	0	-1	1	1	6
0	1	0	1	-1	0	2

La solución factible básica asociada con este nuevo cuadro es (2, 16, 0, 0, 6), con un valor de función objetivo correspondiente de 78.

Aquí se llega al fin del algoritmo dado que ya no hay valores negativos en la función objetivo.

2.2.2. Problema a Resolver

Con el propósito de sustituir el Algoritmo Simplex desarrollado por SimSEE con una versión optimizada para GPU, es fundamental que el nuevo algoritmo tenga la capacidad de gestionar no solo restricciones de $\leq$, sino también de $\geq$ y $=$. Asimismo, debe ser capaz de manejar tanto restricciones de cota superior como de cota inferior.

A continuación, se analizarán diversas variantes del Algoritmo Simplex que pueden gestionar eficazmente restricciones de $\geq$ y $=$. Cabe destacar que las restricciones de cota inferior se pueden abordar de manera sencilla mediante un cambio de variables para cada variable sujeta a una cota inferior. Asimismo, las restricciones de cota superior pueden representarse como restricciones regulares, añadiendo una restricción adicional al problema por cada variable con límite superior.

2.2.3. Método Simplex Big M

El método Simplex Big M es una extensión del método Simplex clásico que permite resolver un problema con restricciones mixtas ($\geq, \leq$ o $=$). Para ello se realizan varias transformaciones del problema hasta llevar el problema a la forma estándar que el algoritmo Simplex puede resolver.

Primero, se debe conseguir una solución básica factible (BFS en sus siglas en inglés). Esta se obtiene cuando en la matriz del problema ciertas columnas conforman la identidad, entonces las variables asociadas a esas columnas conforman la BFS y el sistema presenta una solución trivial. Al asignar el valor 0 al resto de las variables, la solución del sistema esta dada al igualar las variables de la base a las entradas del vector b.

Entonces para transformar las restricciones $\geq$ en restricciones de igualdad agregamos una variables de holgura, la cual es positiva. Pero cuando intentamos transformar en igualdad una restricción de $\leq$ nos encontramos que la holgura es negativa, generando una una entrada -1 en la matriz, que hace que la base sea infactible. Esto se da porque tomando el resto de variables iguales a 0, tenemos que $-1 \times s_i = b_i$, si s_i y b_i deben ser mayores que 0 la ecuación no tiene solución. Entonces para transformar esta restricción en una restricción de igualdad que derive en una BFS se agrega una variable extra, llamada variable artificial, quedando la restricción con la forma $v_1 + .. + v_n + .. - s_i + a_i = b_i$.

Y aquí es donde el término gran M cobra significado, este término refiere a que para lograr que el método Simplex tradicional converja en la solución buscada, penalizaremos en la función objetivo estas variables artificiales con un valor muy grande, llamado M, de forma que en el óptimo de la función todas las variables artificiales tengan necesariamente el valor 0.

Un último requisito de este método es que el costo relativo de las variables artificiales en la función objetivo debe ser 0. Es por esto que para cada variable artificial agregada, restamos -M veces la fila de la restricción a la función objetivo, consiguiendo así que la entrada M de la variable en la función objetivo sea 0.

Se recuerda que para transformar un problema de maximización a minimización se debe multiplicar la función objetivo por -1.

En resumen los pasos del algoritmo son los siguientes:

1. Cualquier restricción que tenga constantes negativas a la derecha, se multiplica por -1 para asegurarse de que el lado derecho sea positivo.
2. Para restricciones $\leq$, se introducen variables de holgura s_i para que todas las restricciones sean igualdades.
3. Para cada restricción $=$, se introduce una variable de holgura s_i.
4. Para cada restricción $\geq$, se introduce una variable de holgura s_i y una variable artificial a_i.
5. Para cada una de las variables artificiales que se introduzca al problema, se elije un valor M positivo grande e introduce un término en el objetivo de la forma $(-M)$ multiplicando las variables artificiales $(-Ma_i)$.
6. Reducir en z las entradas M de las variables artificiales a 0 usando operaciones de fila ($z = z - c_b \times A$).
7. Por último se resuelve el problema usando el método Simplex usual.

2.2.4. Método Simplex Two Phases

El método de las dos fases, como el método Big M, tiene la ventaja de ser un algoritmo más robusto donde las restricciones pueden ser desigualdades de $\geq, \leq o =$ (mixtas). De igual forma se agregan variables de holgura y artificiales, aunque el coeficiente en z de las variables se setea en 0. Luego para lograr eliminar las variables artificiales de la base y obtener una BFS se procede a la primer fase en la cual se utiliza el método Simplex para minimizar la función auxiliar $z' = a_1 + ... + a_n$, siendo a_i las variables artificiales del problema. La solución a este problema se encuentra cuando todas las variables artificiales son 0. Entonces al resolver este problema auxiliar se obtiene una BFS la cual es usada en la segunda fase para resolver el problema original usando el método Simplex usual. (Dr. Hiba G. Fareed, 2022)

Los pasos en el algoritmo son los siguientes:

1. Cualquier restricción que tenga constantes negativas a la derecha, se multiplica por -1 para asegurarse que el lado derecho sea positivo.
2. Para cada restricción $=$, se introduce una variable de holgura s_i.
3. Para cualquier restricción $\geq$, se introduce una variable excedente (s_i) y variables artificiales a_i.
4. Para cada una de las variables artificiales que se introduzca al problema, se introduce el valor 0 en el objetivo.
5. Para restricciones $\leq$, se introducen variables de holgura s_i para que todas las restricciones sean igualdades.

6. Fase 1: Se construye la función auxiliar $z' = a_1 + ... + a_n$. Se reducen las entradas de z' (1s) a 0 usando operaciones de fila ($z' = z' - c_b \times A$).
   Se utiliza el método Simplex para minimizar z'.

7. Fase 2: Se reduce a 0 en z las entradas de las variables de la base obtenida en el paso anterior usando operaciones de fila ($z = z - c_b \times A$).
   Se utiliza el método Simplex para resolver el problema original usando la BFS obtenida en el paso anterior.

2.3. GPUs

Una Unidad de Procesamiento Gráfico, conocida como GPU (por sus siglas en inglés, Graphics Processing Unit), es un componente de hardware diseñado para llevar a cabo tareas relacionadas con el procesamiento de gráficos y operaciones de punto flotante. Su función principal es aliviar la carga de trabajo del procesador central o CPU en una variedad de aplicaciones, permitiendo un rendimiento más eficiente y rápido en el procesamiento de imágenes, videos y cálculos matemáticos intensivos en gráficos.

Además de su contribución en el ámbito de los gráficos, las GPUs también han demostrado ser vitales en la aceleración de aplicaciones científicas y técnicas, como la simulación computacional, la inteligencia artificial, el aprendizaje automático y la criptografía. Estas unidades de procesamiento paralelo están diseñadas para manejar simultáneamente una gran cantidad de cálculos, lo que las convierte en una herramienta esencial para tareas que requieren un alto poder de procesamiento y rendimiento, fuera del ámbito puramente gráfico. Su capacidad de paralelismo ha revolucionado numerosos campos de la informática y la investigación científica.

Una GPU se basa en un chip de silicio que alberga una serie de millones de transistores bajo una arquitectura específica y con unas prestaciones dedicadas para cada modelo en concreto. Tienen hasta miles de núcleos para procesar cargas de trabajo en paralelo de forma eficiente.

La computación acelerada por GPU permite asignar a dicho dispositivo el trabajo de la aplicación donde la computación es más intensiva, mientras que el resto del código se ejecuta en la CPU.

En la Figura 2.1 se puede apreciar la diferencia entre una CPU, que tiene unos cuantos núcleos optimizados para el procesamiento secuencial, y una GPU que cuenta con una arquitectura masivamente paralela que consiste de miles de núcleos más pequeños y simples, diseñados para abordar varias tareas simultáneamente.

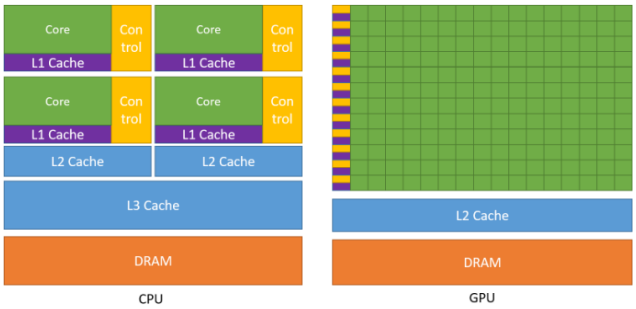


Figura 2.1: Comparación de arquitecturas de CPU vs GPU. Extraído de (*NVIDIA, CUDA C++ Programming Guide 12.2*, 2023)

2.3.1. Arquitectura de Hardware

La arquitectura se fundamenta en una disposición escalable de varias unidades de procesamiento llamadas Streaming Multiprocessors (SM), que son los componentes de procesamiento esenciales en la GPU. La cantidad de SM varía según el modelo de GPU específico. Los SM son unidades de cómputo altamente paralelas que contienen una serie de núcleos de procesamiento, así como memoria compartida y caché. Son los encargados de administrar eficazmente los recursos de hardware (como núcleos de procesamiento, unidades de operaciones en punto flotante, etc.). Esto se hace para asegurar una ejecución eficiente y equitativa de los hilos en paralelo.

Un grid, en el contexto de la programación con GPUs, se organiza en bloques de hilos que pueden ser configurados en una, dos o tres dimensiones, adaptándose al formato de los datos y a las exigencias del problema a resolver. Cada bloque, a su vez, se compone de hilos dispuestos en una estructura que puede igualmente ser de una a tres dimensiones. Estos hilos tienen la capacidad de colaborar entre sí mediante el uso de memoria compartida y sincronización, permitiéndoles ejecutar distintas tareas de manera coordinada para contribuir a la solución de una sección del problema computacional global.

Un grid se inicia a través de un solo programa CUDA llamado kernel. La ejecución de un kernel en la GPU comienza con una llamada dese el host (CPU). Cuando todos los subprocesos de un kernel completan su ejecución, el grid correspondiente finaliza y la ejecución continúa en el host hasta que se invoca otro kernel.

Cuando un programa CUDA en la CPU del host invoca un grid, los bloques de la grid se enumeran y distribuyen a los multiprocesadores con capacidad de ejecución disponible. Los subprocesos de un bloque de subprocesos se ejecutan simultáneamente en un multiprocesador, y varios bloques de subprocesos se pueden ejecutar simultáneamente en un multiprocesador. A medida que terminan los bloques de subprocesos, se lanzan nuevos bloques en los multiprocesadores

desocupados.

Un multiprocesador está diseñado para ejecutar cientos de subprocesos al mismo tiempo. Para administrar una cantidad tan grande de hilos, este emplea un modelo de ejecución único llamado SIMT (Single Instruction Multiple Thread). En un modelo SIMT, todos los hilos en un grupo (warps en el caso de NVIDIA) ejecutan la misma instrucción en un ciclo de reloj dado. Esto significa que todos los hilos en el grupo siguen el mismo flujo de programa.

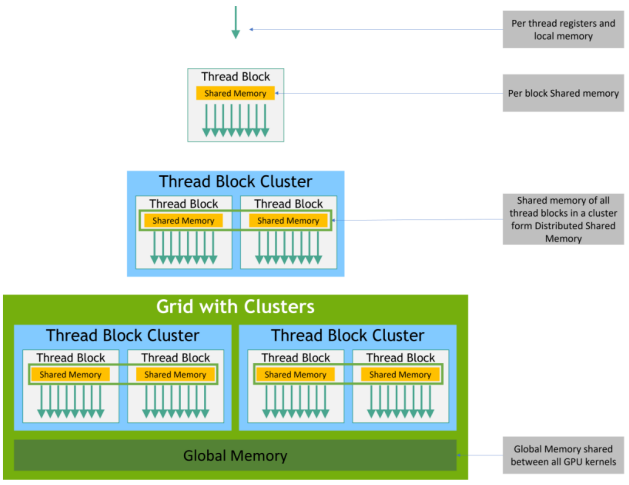


Figura 2.2: Thead y jerarquía de memoria en GPU. Imagen extraída de (*NVIDIA, CUDA C++ Programming Guide 12.2*, 2023)

Como se muestra en la Figura 2.2, un grid representa un conjunto de bloques donde cada bloque contiene hasta 1024 subprocesos. Cuando se inicia el kernel, cada multiprocesador procesa un bloque ejecutando subprocesos en grupos de 32 subprocesos paralelos (warps). El warp se considera como la unidad de ejecución en paralelo, ya que todos los hilos de un mismo warp se ejecutan físicamente en paralelo y por lo tanto comienzan en la misma instrucción (aunque después son libres de bifurcarse y ejecutarse independientemente). Así, cuando se selecciona un bloque para su ejecución, el bloque se divide en warps, se selecciona uno que esté listo para ejecutarse y se emite la siguiente instrucción a todos los hilos que forman el warp. Dado que todos ellos ejecutan la misma instrucción simultáneamente, la máxima eficiencia se consigue cuando todos los hilos coinciden en su ruta de ejecución (sin bifurcaciones).

La jerarquía de memoria en las GPUs NVIDIA, esencial para el alto rendimiento en el procesamiento paralelo, abarca desde registros individuales por hilo, memoria caché L1 y L2, memoria compartida accesible por todos los hilos de un bloque, hasta la memoria global disponible para todos los hilos. La memoria compartida se distingue por facilitar la cooperación entre hilos dentro de

un mismo bloque, organizada en 32 bancos para permitir accesos simultáneos rápidos y eficientes, optimizando así el acceso de cada uno de los 32 hilos del warp de forma individual. Esta disposición no solo reduce la latencia sino que también maximiza el rendimiento al disminuir los cuellos de botella en el acceso a los datos. Entender y programar con un enfoque en la arquitectura de la memoria compartida y su configuración de bancos es crucial. Una gestión eficaz de estos recursos puede prevenir conflictos de bancos, los cuales degradan el rendimiento del programa cuando múltiples hilos intentan acceder al mismo banco de memoria simultáneamente, causando serialización de los accesos y, por ende, retrasos en la ejecución. Diseñar algoritmos que maximicen el uso de la memoria compartida y eviten conflictos de bancos es clave para aprovechar al máximo las capacidades de cálculo de las GPUs NVIDIA.

Para más detalles sobre la arquitectura de las tarjetas NVIDIA y cómo optimizar el código, se hace referencia a (*NVIDIA, CUDA C++ Programming Guide 12.2*, 2023).

2.3.2. Modelo de programación

Se emplea un modelo de programación escalable. Con la presencia de CPUs multinúcleo y GPUs con numerosos núcleos, los procesadores principales han evolucionado hacia sistemas paralelos, y su capacidad de paralelismo continúa expandiéndose. El reto reside en desarrollar software de aplicación que pueda aumentar su paralelismo de manera transparente, permitiendo así una utilización más eficiente del número de procesadores disponible

A continuación se presenta los conceptos principales detrás del modelo de programación CUDA al describir cómo se exponen en C++.

Kernel

La definición de un kernel se realiza mediante el uso del especificador de declaración __global__ . La cantidad de subprocesos CUDA que ejecutan dicho kernel en una llamada específica se determina mediante una nueva sintaxis de configuración de ejecución «...». El subproceso que lleva a cabo la ejecución del kernel recibe una identificación única, la cual puede ser accedida dentro del propio kernel a través de variables especiales dedicadas a ese fin.

Grid y Thread Block

En CUDA, los hilos se organizan en una jerarquía compuesta por un grid y bloques.

El grid es la estructura de nivel superior y representa una colección de bloques. Cada grid tiene tres dimenciones, lo que significa que puede contener múltiples bloques organizados en las direcciones X, Y y Z.

Cada grid se divide en bloques, también tridimensionales. Los bloques son la unidad de programación en CUDA, y los hilos dentro de un bloque pueden

cooperar y compartir datos a través de una memoria compartida de alto rendimiento. Los bloques están diseñados para trabajar juntos en tareas paralelas y son la unidad básica de asignación de hilos.

En la Figura 2.3 se ilustra como es la organización de los bloques e hilos en un grid.

Existe un límite para la cantidad de hilos por bloque, ya que se espera que todos los hilos de un bloque residan en el mismo SM y deben compartir los recursos de memoria limitados del multiprocesador. En las GPUs actuales, el número máximo de hilos que puede contener un bloque es de 1024 hilos por bloque.

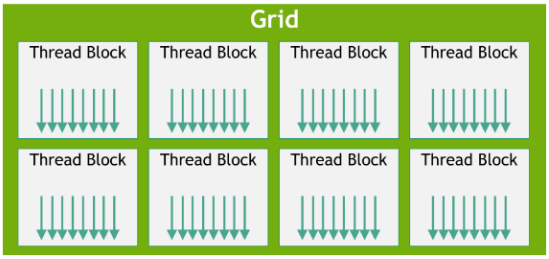


Figura 2.3: Ejemplo de grid de bloques de hilos. Extraído de (*NVIDIA, CUDA C++ Programming Guide 12.2*, 2023)

Cada bloque dentro del grid se puede identificar mediante un índice único unidimensional, bidimensional o tridimensional accesible dentro del kernel a través de la variable *blockIdx* incorporada. Se puede acceder a la dimensión del bloque de subprocesos dentro del kernel a través de la variable *blockDim* incorporada.

2.3.3. Jerarquía de memoria

Al igual que el host, la memoria del dispositvo se compone de varias memorias de distinta tecnología, tamaño, latencia y ancho de banda, las cuales se organizan en una jerarquía.

Memoria Global Es una memoria de alta capacidad y ancho de banda que se utiliza para almacenar datos que serán utilizados por múltiples hilos de ejecución en un kernel (función) de GPU. La memoria global es accesible por todos los hilos en un programa en paralelo y sus datos persisten en la GPU durante toda la ejecución del programa. Los datos almacenados en la memoria global pueden ser leídos y escritos por cualquier hilo en cualquier bloque de hilos en ejecución en la GPU.

Proporciona una forma de compartir datos entre hilos y bloques, lo que permite la cooperación y la sincronización entre ellos. Por ejemplo, si varios hilos necesitan acceder a los mismos datos o compartir resultados parciales,

pueden utilizar la memoria global para comunicarse y compartir información. Además permite intercambiar datos con el host.

Por otro lado, la memoria global tiene una latencia relativamente alta en comparación con otras formas de memoria en una GPU, como la memoria compartida o la memoria caché.

El acceso coalesced a la memoria global es una optimización clave que mejora el rendimiento agrupando los accesos de memoria de los hilos de un warp. Cuando estos accesos son a direcciones secuenciales y contiguas, la GPU puede fusionarlos en una sola transacción, lo cual reduce el ancho de banda necesario y aumenta la eficiencia. Para aprovechar este mecanismo, es crucial que los desarrolladores estructuren los datos de manera que los accesos por parte de los hilos sean alineados.

Memoria Compartida La memoria compartida en una GPU es un tipo de memoria de acceso rápido y baja latencia que se encuentra dentro de cada SM de la GPU. Esta memoria se comparte entre los hilos de un mismo bloque y se utiliza para facilitar la comunicación y la colaboración entre estos hilos mientras ejecutan tareas en paralelo en un bloque.

La memoria compartida en una GPU, caracterizada por su rápido acceso y baja latencia, se ubica dentro de cada SM de la GPU. Aunque su capacidad es significativamente menor en comparación con la memoria global, esta memoria es compartida entre los hilos de un mismo bloque, optimizando la comunicación y colaboración entre ellos durante la ejecución de tareas en paralelo en un mismo bloque. Su organización en bancos facilita el acceso concurrente, pero requiere una gestión cuidadosa para evitar conflictos de banco.

Memoria Local El espacio de la memoria local reside en la memoria global del dispositivo, por lo que los accesos a la memoria local tienen la misma latencia alta y ancho de banda bajo que los accesos a la memoria global y están sujetos a los mismos requisitos para la optimización de memoria. Sin embargo, la memoria local está organizada de tal manera que se accede a palabras consecutivas de 32 bits mediante ID (identificador único) de subprocesos consecutivos.

Memoria Constante La memoria constante es un tipo de memoria que se utiliza para almacenar datos que son constantes durante la ejecución de un kernel. Estos datos son accesibles por todos los hilos en un bloque y tienen la característica de ser solo de lectura, no se pueden modificar durante la ejecución del kernel.

Archivo de registros El archivo de registros en una GPU representa una forma de almacenamiento extremadamente rápido y de baja latencia. Cada hilo dispone de un conjunto propio de registros para almacenar variables locales y temporales, cruciales para su rendimiento. Dada su proximidad al núcleo de procesamiento, el acceso a los registros es mucho más rápido que a cualquier otro tipo de memoria en la GPU, haciéndolos ideales para operaciones intensivas y de

alta velocidad. Sin embargo, el espacio limitado en el archivo de registros exige una gestión cuidadosa por parte del desarrollador para minimizar el recurso a memorias más lentas, como la local o global. Optimizar el uso de los registros es clave para maximizar la eficiencia computacional.

Calificadores de tipo variable Los calificadores de tipo de variable especifican la ubicación de memoria en el dispositivo de una variable. Una variable automática declarada en código de dispositivo sin ninguno de los calificadores __device__, __shared__ y __constant__ generalmente reside en un registro. Sin embargo, en algunos casos, el compilador puede optar por colocarlo en memoria local.

El calificador *__constant__* , usado opcionalmente junto con __device__, declara una variable que reside en un espacio de memoria constante, tiene el tiempo de vida de una aplicación y es accesible desde todos los subprocesos dentro del grid y desde el host a través de la biblioteca de tiempo de ejecución.

El calificador *__shared__*, usado opcionalmente junto con __device__, declara una variable que reside en el espacio de memoria compartida de un bloque de subprocesos, tiene la duración del bloque y solo es accesible desde todos los subprocesos dentro del bloque.

2.4. Herramientas de profiling

Las herramientas de profiling son fundamentales en el desarrollo de software, proporcionando una visión detallada del comportamiento de las aplicaciones en términos de rendimiento y uso de recursos. Gprof, por ejemplo, es un profiler clásico que inserta código en las funciones para medir el tiempo de ejecución, generando un archivo `gmon.out` con los datos recopilados. Sin embargo, existen muchas otras herramientas que ofrecen diferentes enfoques y capacidades de análisis.

Valgrind, por ejemplo, es ampliamente utilizado para detectar fugas de memoria y problemas de gestión de recursos, mientras que Perf proporciona un análisis de rendimiento a nivel de sistema operativo, capturando estadísticas de ejecución de todo el sistema. Nsight, desarrollado por NVIDIA, se especializa en el profiling de aplicaciones CUDA para GPUs, ofreciendo una visión profunda sobre la ejecución paralela, uso de memoria y cuellos de botella de rendimiento específicos de la arquitectura de GPU.

En el proceso de desarrollo de los diversos prototipos para este proyecto, Nsight se ha utilizado efectivamente para ajustar parámetros críticos de configuración, como los tamaños de grilla y bloque. Este ajuste fino es esencial para

optimizar el rendimiento de las aplicaciones CUDA, permitiendo a los desarrolladores identificar la configuración óptima.

Gprof es un programa de creación de perfiles que recopila y organiza estadísticas sobre los programas. Analiza cada una de sus funciones e inserta código al principio y al final de cada una para recopilar información de tiempo. Luego, cuando ejecuta su programa normalmente crea un archivo llamado "gmon.out".

La creación de perfiles es un aspecto importante ya que permite determinar las partes del código del programa que consumen mucho tiempo y necesitan ser analizadas en detalle.

Para usar la heramineta gprof:

- Tener la creación de perfiles habilitada mientras se compila el código (compilar y vincular con la opción -pg).
- Ejecutar el código del programa para producir los datos de perfil.
- Ejecute la herramienta gprof en el archivo de datos de generación de perfiles, se escribe gprof exec > out , donde exec se reemplaza por el nombre de su ejecutable y out por algún nombre significativo para la información de creación de perfiles.

La decisión de utilizar Gprof esta relacionada por su facilidad de uso y su capacidad para proporcionar un análisis detallado del tiempo de ejecución.

2.5. Trabajos relacionados

En el artículo de Lalami (Mohamed Esseghir Lalami, 2011) se ofrece una descripción detallada del algoritmo Simplex y su implementación en GPU. Este artículo se concentra en la implementación paralela del algoritmo Simplex estándar en sistemas CPU-GPU para problemas de PL densos.

Los autores observan que al identificar las tareas factibles de paralelizar y con una buena gestión de las memorias de las GPU, se puede obtener una buena aceleración con respecto a la implementación secuencial (que es lo que queremos lograr con este proyecto de aceleración de SimSEE). Este trabajo es parte de un estudio sobre la paralelización de métodos de optimización.

La implementación en paralelo se realiza optimizando los diferentes pasos del algoritmo Simplex y los resultados computacionales mostraron que su implementación en doble precisión en el sistema CPU-GPU es eficiente, ya que para grandes problemas de programación lineal no dispersa han obtenido aceleraciones estables.

He et al. (He, Bai, Jiang, y Ouyang, 2018) se centran en la adaptación del algoritmo Simplex revisado (RSA, por sus siglas en inglés) para Unidades de

Procesamiento Gráfico (GPUs) utilizando CUDA, con el objetivo de aprovechar las capacidades de procesamiento en paralelo inherentes de las GPUs.

El RSA puede ser más numéricamente estable y eficiente en ciertos escenarios, y se desarrolló para abordar algunas de las limitaciones del Algoritmo Simplex original, especialmente en casos en los que la solución cambia con frecuencia o cuando ocurre la degeneración (una situación en la que una o más variables permanecen en cero).

Una de las principales diferencias entre el RSA y el Algoritmo Simplex original es el uso de técnicas de factorización de matrices para actualizar la matriz de base. Esto permite que el RSA aproveche la dispersión en la matriz de restricción y reduzca el esfuerzo computacional.

Los investigadores transfirieron las manipulaciones de matrices computacionalmente intensivas tradicionalmente asociadas con el RSA a la GPU, aprovechando efectivamente sus potentes capacidades de procesamiento en paralelo. A través de experimentos rigurosos, demostraron que el RSA basado en GPU no solo producía de manera consistente soluciones óptimas correctas, sino que también exhibía notables ventajas en velocidad, superando el rendimiento del RSA basado en CPU en hasta 100 veces. Este enfoque innovador promete mejorar significativamente la eficiencia y velocidad en la resolución de problemas de programación lineal, especialmente en plataformas informáticas modernas equipadas con GPUs.

Otro artículo o trabajo relacionado es (Alfredo G. Escobar Portillo, enero-febrero 2011). En este artículo, se exploró la utilización del procesamiento paralelo en tarjetas gráficas, que operan bajo el modelo SIMD (Single Instruction Multiple Data). El enfoque se centró en aplicar esta técnica a un algoritmo matricial común conocido como el RSA.

El artículo aprovechó el crecimiento de las tarjetas gráficas cada vez más potentes disponibles en los entornos domésticos. Se utilizó la tecnología CUDA de NVIDIA para desarrollar una herramienta que permitiera realizar cálculos en paralelo y se llevaron a cabo experimentos para evaluar su eficiencia.

Los resultados mostraron que este enfoque de cálculo paralelo es factible para la mayoría de las personas, ya que se pueden utilizar tarjetas gráficas convencionales que suelen estar presentes en las computadoras personales sin requerir una inversión económica significativa.

Se realizaron pruebas comparativas entre una CPU y dos GPU diferentes: una con 192 núcleos y otra con 480 núcleos. Se encontró que los beneficios de la paralelización comienzan a ser evidentes cuando el tamaño del problema supera un cierto umbral, que se estimó en alrededor de 600 restricciones. En problemas más pequeños, no resulta rentable utilizar GPUs.

Aunque se demostraron beneficios al paralelizar el algoritmo símplex revisado en GPUs, especialmente para problemas grandes, el aumento en rendimiento no fue tan marcado como se podría anticipar considerando el mayor número de núcleos de las GPUs. Este resultado indica que la paralelización del algoritmo tiene espacio para optimizaciones adicionales. Este hallazgo resalta la oportunidad de futuras investigaciones para mejorar la eficiencia de las GPUs en este

contexto, marcando una dirección prometedora para trabajos subsiguientes.

Durante este relevamiento, se identificaron numerosos estudios relacionados con la aplicación de diversas técnicas para resolver el Algoritmo Simplex Revisado RSA. Sin embargo, no se encontraron investigaciones que aborden la resolución en GPU del Algoritmo Simplex cuando se manejan cotas superiores sin la necesidad de añadir restricciones adicionales. Este enfoque se conoce en la literatura como la "Técnica de la cota superior" (Bounded Simplex Method en inglés), como se describe en (Hillier F.S., 1991) en el capítulo 7.3.

Es importante destacar que aunque nuestro problema implica tanto restricciones de cota superior como de cota inferior, abordar las restricciones de cota inferior se simplifica mediante un cambio de variables que ajusta la cota inferior a cero. Dado que el algoritmo Simplex requiere que las variables sean mayores que cero, este cambio de variables se aplica a todas las variables con cota inferior, manteniendo así el problema en su forma estándar.

Además, se observa que la mayoría de los trabajos relacionados se enfocan en mejorar el rendimiento para problemas de gran escala, que implican decenas de miles de variables y miles de restricciones, ejemplo (Usman Ali Shah y Ahmad, 2020). Esto contrasta significativamente con nuestro enfoque, que se centra en la resolución paralela de un gran número de problemas Simplex de pequeña escala, típicamente con 100 a 400 variables y 20 a 100 restricciones, en el rango de 1000 a 3000 problemas simultáneos.

Capítulo 3

Estudio del desempeño del modelo SimSEE

Es importante conocer los principales cuellos de botella en el desempeño de SimSEE en escenarios realistas para luego poder diseñar, implementar y validar un prototipo que incorpore cómputo en GPU a la herramienta SimSEE y mejore su desempeño. Para medir este desempeño son útiles las herramientas de profiling que permiten identificar las partes de un programa que están consumiendo una cantidad significativa de tiempo o recursos del sistema, luego de identificar los cuellos de botella, comprender dónde y por qué se está produciendo una degradación del rendimiento.

La primera etapa del trabajo consiste en la instalación de la herramienta SimSEE y el entorno de desarrollo, buscar mejoras y avances en la forma en que se utilizan las GPUs para abordar el problema, y luego proceder a realizar el profiling de la herramienta SimSEE.

Además, se presenta una descripción del proyecto simplificado en el que se trabajó, así como una explicación de la implementación en lenguaje C del algoritmo Simplex.

3.1. Ambiente de Trabajo

Como SimSEE está implementado en Delphi (Pascal) es necesario instalar el compilador Free Pascal (Free Pascal, 2020), y una herramienta de desarrollo de aplicaciones Pascal en el entorno de desarrollo

Lazarus (Lazarus, 2020) es un entorno de desarrollo integrado (IDE) de código abierto y gratuito para programación en Pascal. Permite crear una amplia variedad de aplicaciones de software, desde aplicaciones de escritorio simples hasta aplicaciones más complejas y robustas. Es compatible con múltiples plataformas (como Windows, Linux, macOS).

Pascal XE es otra IDE para el desarrollo en Pascal que ofrece un entorno de desarrollo integrado y características para el desarrollo de aplicaciones en Pascal, pero se eligió Lazarus, principalmente por presentar mayor documentación y una mayor comunidad de desarrolladores, por ser de código abierto y gratuito.

3.2. Profiling

Para llevar a cabo el profiling, se utilizaron algunos servidores de la Facultad de Ingeniería equipados con unidades de procesamiento gráfico (GPU). Los servidores utilizados fueron labgpu01, labgpu02 y labgpu03.

La fase de evaluación se centró principalmente en el servidor labgpu01. Este servidor está equipado con dos GPU GTX 980 (las GTX 980 son tarjetas gráficas de la serie GeForce GTX de NVIDIA), una GTX 1060 con 3 GB de memoria RAM, y un procesador i7-6700 CPU @ 3.40 GHz con 64 GB de RAM. Con el objetivo de analizar el rendimiento del simulador y detectar posibles cuellos de botella, se descargaron los archivos fuente versión 27 (SimSEE_viie27.7z) [1]

Para compilar el proyecto asociado (cmdsim.dpr) se debe ejecutar:

- fpc -B -dRELEASE @fp64.cfg ./cmdsim.dpr

El archivo `fp64.cfg` contiene la configuración necesaria para la compilación de `cmdsim`. Este archivo de configuración proporciona instrucciones detalladas al compilador de Pascal sobre cómo compilar el proyecto, incluidas las rutas para buscar dependencias, las directivas de compilación condicional, los niveles de optimización y otros parámetros de compilación. Estas configuraciones ayudan a personalizar la compilación para adaptarse a las necesidades específicas del proyecto y su entorno de ejecución.

Al ejecutar la línea anterior, se generará el archivo binario correspondiente al simulador en el directorio `/bin`.

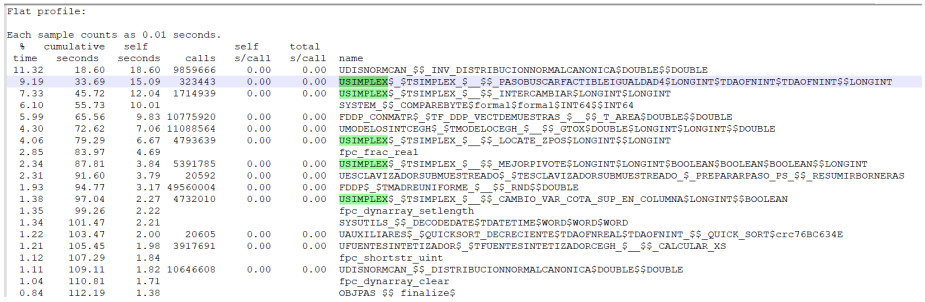
Luego se ejecuta el simulador de la siguiente forma:

- ./cmdsim sala='direccion-completa/sala-ejemplo.ese'

Antes de la ejecución del simulador, es necesario correr el optimizador indicando la sala con las fuentes deseadas y las condiciones iniciales para generar los archivos que permiten realizar la simulación correspondiente a la sala indicada.

En la Figura 3.1 se puede observar que los tiempos están ordenados en forma descendiente en cuanto al porcentaje de tiempo de las funciones que utilizan más tiempo de ejecución. En el Cuadro 3.1 se puede ver un resumen de los porcentajes de tiempo correspondiente a las funciones de SimSEE. En este estudio se puede observar que la implementación del método Simplex lleva casi un cuarto del tiempo total de la ejecución. Por este motivo, nos enfocamos en mejorar ese algoritmo y lograr ejecutarlo en una GPU. La mayor parte del tiempo se dedica a las operaciones de pivoteo. Funciones como: `PASOBUSCARFACTIBLEIGUALDAD`,

[1] https://adme.com.uy/db-docs/Docs_secciones/nid_1206/SimSEE_viie27.7z



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 11.32     18.60    18.60  9859666    0.00     0.00  UDISNORMCAN_$$_INV_DISTRIBUCIONNORMALCANONICA$DOUBLE$$DOUBLE
  9.19     33.69    15.09   323443    0.00     0.00  USIMPLEX$_$TSIMPLEX_$__$$_PASOBUSCARFACTIBLEIGUALDAD4$LONGINT$TDAOFNINT$TDAOFNINT$$LONGINT
  7.33     45.72    12.04  1714939    0.00     0.00  USIMPLEX$_$TSIMPLEX_$__$$_INTERCAMBIAR$LONGINT$LONGINT
  6.10     55.73    10.01                             SYSTEM_$$_COMPAREBYTES$formal$formal$INT64$$INT64
  5.99     65.56     9.83 10775920    0.00     0.00  FDDP_CONMATR$_$TF_DDP_VECTDEMUESTRAS_$__$$_T_AREA$DOUBLE$$DOUBLE
  4.30     72.62     7.06 11088564    0.00     0.00  UMODELOSINTCEGH$_$TMODELOCEGH_$__$$_GTOX$DOUBLE$LONGINT$LONGINT$$DOUBLE
  4.06     79.29     6.67  4793639    0.00     0.00  USIMPLEX$_$TSIMPLEX_$__$$_LOCATE_ZPOS$LONGINT$$LONGINT
  2.85     83.97     4.69                             fpc_frac_real
  2.34     87.81     3.84  5391785    0.00     0.00  USIMPLEX$_$TSIMPLEX_$__$$_MEJORPIVOTE$LONGINT$LONGINT$BOOLEAN$BOOLEAN$BOOLEAN$$LONGINT
  2.31     91.60     3.79    20592    0.00     0.00  UESCLAVIZADORSUBMUESTREADOS_$TESCLAVIZADORSUBMUESTREADO_$_PREPARARPASO_PS_$$_RESUMIRBORNERAS
  1.93     94.77     3.17 49560004    0.00     0.00  FDDP$_$TMADREUNIFORME_$__$$_RND$$DOUBLE
  1.38     97.04     2.27  4732010    0.00     0.00  USIMPLEX$_$TSIMPLEX_$__$$_CAMBIO_VAR_COTA_SUP_EN_COLUMNA$LONGINT$$BOOLEAN
  1.35     99.26     2.22                             fpc_dynarray_setlength
  1.34    101.47     2.21                             SYSUTILS_$$_DECODEDATE$TDATETIME$WORD$WORD$WORD
  1.22    103.47     2.00    20605    0.00     0.00  UAUXILIARES$_$QUICKSORT_DECRECIENTES$TDAOFNREAL$TDAOFNINT_$$_QUICK_SORT$crc76BC634E
  1.21    105.45     1.98  3917691    0.00     0.00  UFUENTESINTETIZADOR$_$TFUENTESINTETIZADORCEGH_$__$$_CALCULAR_XS
  1.12    107.29     1.84                             fpc_shortstr_uint
  1.11    109.11     1.82 10646608    0.00     0.00  UDISNORMCAN_$$_DISTRIBUCIONNORMALCANONICA$DOUBLE$$DOUBLE
  1.04    110.81     1.71                             fpc_dynarray_clear
  0.84    112.19     1.38                             OBJPAS_$$_finalize$
```

Figura 3.1: Resultados profiling SimSEE

Porcentaje	Función
9.19	`USIMPLEX - PASOBUSCARFACTIBLEIGUALDAD4`
7.33	`USIMPLEX - INTERCAMBIAR`
4.06	`USIMPLEX - LOCATEZPOS`
2.34	`USIMPLEX - MEJORPIVOTE`
1.38	`USIMPLEX - CAMBIOVARCOTASUPENCOLUMNA`

Cuadro 3.1: Tabla profiling

`INTERCAMBIAR`, `LOCATEZPOS`, `MEJORPIVOTE`, `CAMBIOVARCOTASUPENCOLUMNA`. Si sumamos todos los tiempos de estas funciones nos da que se dedica un 24,3 % del tiempo total de ejecución en el simplex. Por este motivo es que se decidió enfocarse en la aceleración del método Simplex.

Los datos de ejecución del algoritmo Simplex varían según el problema de optimización lineal que se esté resolviendo, incluyendo el número de variables, restricciones y la estructura de las matrices.

En una investigación previa (Marichal y cols., 2021), se abordaron diversos casos de prueba realistas para evaluar SimSEE en distintos escenarios, incluyendo horarios, diarios y semanales, que representan diferentes plazos de simulación (corto, medio y largo plazo), como se puede ver en la Figura 3.2.

TABLE I
HOURLY DRY PLAYROOM.

Procedure	Perc. (%)	CPU Cycles	Calls	Unit
PASOBUSC...	5.7	2.902×10^9	4.786×10^5	USIMPLEX
INTERCAMBIAR	5.5	2.772×10^9	$1,640 \times 10^6$	USIMPLEX
RND	4.8	2.421×10^9	7.457×10^7	FDDP

TABLE II
HOURLY MID-HUMID PLAYROOM.

Procedure	Perc. (%)	CPU Cycles	Calls	Unit
PASOBUSC...	6.8	3.612×10^9	5.942×10^5	USIMPLEX
INTERCAMBIAR	4.9	2.584×10^9	1.611×10^6	USIMPLEX
RND	4.6	2.421×10^9	7.457×10^7	FDDP

TABLE III
HOURLY HUMID PLAYROOM.

Procedure	Perc. (%)	CPU Cycles	Calls	Unit
EROGADO...	10.5	6.949×10^9	2.037×10^5	UHIDRO...
VAL_X	9.0	5.929×10^9	2.044×10^8	COMPOL
PASOBUSC...	5.1	3.392×10^9	4.977×10^5	USIMPLEX
INTERCAMBIAR	4.6	3.030×10^9	1.678×10^6	USIMPLEX
RND	4.6	2.421×10^9	7.457×10^7	FDDP

Figura 3.2: Resultados profiling SimSEE. Extraído de (Marichal y cols., 2021)

El objetivo principal de dichos estudios fue identificar los procedimientos que requerían más recursos durante la simulación. Luego de varios experimentos demostraron que ninguno de los procedimientos individualmente representaba un cuello de botella crítico, ya que una única ejecución de estos procedimientos no ejercía una carga significativa en los recursos. Sin embargo, se identificó que una parte sustancial del tiempo de ejecución se consumía en las rutinas del método Simplex.

Para obtener datos de ejecución específicos para escenarios y tablas particulares en relación con el algoritmo Simplex, es necesario realizar experimentos y pruebas en un entorno específico con datos concretos. El algoritmo Simplex se implemento en C como paso intermedio antes de pasar a CUDA, porque hace que pasar a CUDA sea más fácil y con menos errores, así luego ejecutarlo en diferentes escenarios para recopilar información sobre el tiempo de ejecución y otras métricas de rendimiento.

3.3. Adaptación del diseño de SimSEE para el procesamiento en paralelo

Luego de analizar el problema se pudo observar que el cálculo del método del Simplex es una parte importante e interesante a enviar a la GPU, debido a la cantidad de llamados a este algoritmo, del orden de millones, y el peso de su resolución. Dado que el costo de transmitir los datos a la GPU y lanzar un millón de instancias es demasiado elevado debido a la arquitectura del dispositivo, en los artículos (Marichal y cols., 2021) y (Marichal, Seveso, Dufrechou, y Ezzatti, 2022), se propone reestructurar el orden de ejecución, avanzando todas las crónicas (trayectorias) hasta el paso del cálculo del Simplex y lanzar a la GPU todos los Simplex de todas las crónicas a la vez por paso de tiempo.

"The main idea is to compute many independent trajectories concurrently, executing multiple instances of resolver or, equivalently, solving multiple simplex matrices in parallel."(Marichal y cols., 2021)

La reestructuración implica un cambio en la forma en que se aborda el cálculo en SimSEE, priorizando la ejecución simultánea de múltiples trayectorias independientes en lugar de la optimización de operaciones individuales. Esto se logra mediante la paralelización de la solución de múltiples matrices simplex asociadas con diferentes trayectorias y la optimización de las transferencias de datos. La estrategia demuestra ser efectiva en términos de rendimiento, particularmente cuando se manejan grandes cantidades de crónicas.

Entonces este proyecto se focaliza en implementar una buena resolución del algoritmo Simplex en GPU, acorde a las necesidades de SimSEE y que esté orientado a resolver lotes de problemas de gran tamaño, del orden de 1000 o 2000 crónicas a simular, y comparar estos resultados con los resultados obtenidos por el algoritmo actual de SimSEE que corre en CPU.

Para esto se instaló y probó una versión específica de SimSEE, denominada "simsee-hcl,"diseñada para generar un controlador de SimSEE que permita probar la implementación del algoritmo Simplex de forma aislada.

En el directorio */SimSEE_GPUs/SimSEE_src/src/trunk/rchlib/testSimplexGPUs* del proyecto de simsee-hcl[2], se encuentran los archivos fuente que facilitan la experimentación con el uso de una GPU en relación con las unidades asociadas al algoritmo Simplex (como umipsimplex, usimplex, etc.). Específicamente el script *make_test.sh* se usa para limpiar los binarios existentes y compilar nuevamente el proyecto. La compilación aborda dos aspectos clave, las rutinas implementadas en CUDA (que se encuentran en el directorio */cuda* y se declaran como *external* para su uso posterior en el programa Pascal) y el proyecto Pascal que utiliza las unidades relacionadas con el Simplex

[2]https://gitlab.fing.edu.uy/raul.marichal/simsee-hcl

(busca lograr una versión paralela de la rutina *resolver* que se encuentra en *rchlib/mat/umipsimplex.pas*.).

- ./testSimplexGPUs.o simplexPruebaGpus_0.xlt 16

Para la ejecución se utiliza el programa utilizando el archivo *testSimplexGPUs.lpr*. Este programa crea la memoria necesaria, instancia el Simplex y la memoria compartida con la GPU. Además llama a *resolver_cuda* que realiza copias de memoria necesarias, invoca el kernel *kernel_resolver* (que resuelve múltiples instancias de Simplex en paralelo en la GPU). Finalmente, se realizan copias de memoria para devolver los cambios a las estructuras de datos de la matriz y del Simplex.

La parte del kernel "*kernel_resolver*" es donde se busca lograr la paralelización eficiente de las instancias de Simplex en la GPU.

Capítulo 4

Desarrollo de una variante masivamente paralela

Este capítulo presenta el desarrollo de la variante masivamente paralela propuesto. Se describe de forma general el diseño masivamente paralelo del Simplex y se dan detalles sobre la implementación.

Como primer paso a resolver, se realizó una versión en C del algoritmo Simplex implementado por SimSEE en Pascal. Este algoritmo está basado en algoritmo Simplex descrito en (Heinz, 1990) el cual funciona de la siguiente forma :

- Regla 1: El elemento pivote debe ser negativo.
- Regla 2: El elemento pivote debe elegirse por encima de un elemento positivo de la fila z.
- Regla 3. El elemento pivote a_{pq}, entre todos los elementos negativos del misma columna q, debe tener la propiedad de que a_{it}/a_{iq} (t es el índice de la columna i) es máximo para $i = p$ (es decir, mínimo en valor absoluto).

No obstante este algoritmo tiene varias modificaciones con motivo de mejorar su rendimiento, como por ejemplo reordenaciones en la matriz, búsqueda de paso factible y manejo de cotas superiores sin ser agregadas como restricciones en la matriz del problema. Como guía inicial para entender la implementación del método Simplex en el SimSSE se puede usar el manual de Usuario de la clase TSimplex (Pablo Alfaro, 2009).

Para la implementación en C del Simplex, se tuvo que manejar los índices en los arrays top y left con un corrimiento (-1) dentro algoritmo. Se tomó esta decisión del corrimiento del índice ya que en la estructura del Simplex original, los arreglos top y left, que guardan la información de las permutaciones

de variables y restricciones, estaban indexados desde 1 (el primer elemento se desperdiciaba). Esto es porque top[i] guarda la posición de la entrada i y el signo indica si la variable está en la base. Si estuvieran indexadas desde 0, al no poder distinguir entre +0 y -0 esta información se perdería. Esto conlleva a que la matriz esté indexada desde 1 con lo cual la primera fila y columna se desperdician.

Luego de implementar en C y analizar esta primera resolución del Simplex, notamos que es bastante compleja, con muchas bifurcaciones, copias de datos, reordenaciones de la matriz del Simplex, cambio de los límites de las iteraciones, etc. Este tipo de códigos pueden no ser adecuado para ejecutar eficientemente en una GPU, los códigos complejos que involucran bifurcaciones, copias de datos y reordenamientos pueden dificultar la implementación de un paralelismo efectivo, e intentar paralelizar secciones de este código puede no aprovechar todo el potencial de una GPU.
Por esto se propone dos variantes. La primera está basada en la implementación secuencial y se enfoca en paralelizar las secciones computacionalmente más intensivas. Y la segunda, realizar otra versión del algoritmo Simplex que busca ser más compatible con la arquitectura de una GPU y así poder realizar una comparación.

4.1. Descripción variante 1 Simplex SimSEE

Antes de comenzar con la implementación, se realizó un diseño esquemático de la solución, de modo de ver en qué forma se van a repartir los hilos en los bloques, sobre qué datos va a trabajar cada hilo, qué cosas se realizarán en paralelo, cuáles serán secuenciales y qué datos se podrán reutilizar (utilizando por ejemplo la memoria compartida).

Para el diseño primero se tomó en consideración que se trabajará sobre una GPU NVIDIA con 1000 CUDA cores o más, y se calculó un Simplex de tamaño considerable para cada una de 1000 trayectorias.

En estas condiciones se estimó que una buena opción sería tomar bloques de 32 hilos y asignar cada Simplex a un bloque, de forma de balancear las operaciones que se pueden hacer en paralelo (como por ejemplo la función intercambiar) y las veces que se debe de usar un solo hilo de los 32 que tiene el bloque (secuencial). El diseño seguido permite tomar N hilos para cada bloque-Simplex y que el funcionamiento no se vea afectado. Esto luego se confirmó experimentalmente variando la cantidad de hilos por bloque.

Las posibles distribuciones de hilos que se planteó fue la de utilizar tantos bloques como trayectorias, y que cada bloque resuelva un Simplex. Respecto a la organización y el tamaño de los bloques, la disposición de los datos se corresponden mejor con un arreglo de hilos unidimensional.

Para determinar el tamaño de bloque se debe tomar en cuenta la arquitectura de la GPU. Cada GPU tiene una arquitectura específica con un número diferente de multiprocesadores (SM, Streaming Multiprocessors), CUDA cores, y memoria compartida por SM. En particular, si se utiliza memoria compartida de manera intensiva, el tamaño de bloque debe ajustarse para que esta sea suficiente para almacenar los datos de todos los bloques ejecutando al mismo tiempo, cada SM tiene una cantidad limitada de memoria compartida, y asignar demasiado a un solo bloque puede limitar la cantidad de bloques que pueden ejecutarse simultáneamente.

Opción 1 - 32 en general no es una buena opción debido al máximo de bloques por SM. Suele dar una occupancy por debajo del óptimo. La occupancy se refiere a la ocupación de los recursos de un multiprocesador (SM) en una GPU por parte de los bloques de hilos que están ejecutándose en ese SM. Es la medida de cuánto se está utilizando un SM en términos de hilos activos en comparación con su capacidad máxima de procesamiento. El objetivo al elegir el tamaño de bloque adecuado es equilibrar el número de hilos por bloque y el número de bloques por SM para alcanzar la occupancy óptima.

Opción 2 - (NumeroVariablesPromedio divEnt 32) x 32. Osea el múltiplo de 32 más cerca del promedio. Al elegir un tamaño de bloque que esté cerca del promedio del número de variables, es más probable que se obtenga una occupancy óptima de la GPU. Esto significa que la GPU estará bien utilizada, con la cantidad correcta de bloques e hilos para mantener ocupados los multiprocesadores.

Las siguientes funciones son ejecutadas por un único hilo:

- Resolver_multihilo (antes resolver_cpu) llamada dentro de kernel_resolver por cada bloque.
- posicionarPrimeraLibre
- fijarVariables
- resolverIgualdades
- pasoBuscarFactible
- darpaso

Mientras que las funciones que se consideró que se podrían ejecutar en forma paralela son las que se muestran el cuadro 4.1.

Alta	Media	Baja
intercambiar	buscarMejorPivoteEnCol	Locate_zpos
fijarCajasLaminares	pasoBuscarFactibleIgualdad4	Mejorpivote
intercambioColumnas	reordenarPorFactibilidad	Cambio_var_cota_sup_en_columna
intercambioFilas		locate_qOK
Cambiar_borde_de_caja		test_qOK

Cuadro 4.1: Los títulos de las columnas indican para cada función del algoritmo Simplex SimSEE la aptitud para ser paralelizado

La estimación de si un segmento de código es apto para ser paralelizado en una GPU y la clasificación de las prioridades de paralelización se basan en una evaluación del código en función de ciertas características y requisitos que son relevantes para el procesamiento en una GPU. Las características no convenientes para el cálculo en GPU generalmente se relacionan con una alta sincronización de los hilos, la cual complejiza el desarrollo en c e impacta negativamente en el tiempo de ejecución.

Los criterios que se utilizaron para evaluar la aptitud para la paralelización son los siguientes:

- Naturaleza de las operaciones: Se analiza qué tipo de operaciones realiza el código. Las operaciones que pueden realizarse en paralelo son más adecuadas para la GPU. Por ejemplo, operaciones matriciales o vectoriales suelen ser altamente paralelizables.

- Dependencias de datos: Se evalúan las dependencias entre datos y operaciones en el código. Si hay muchas dependencias que requieren sincronización, puede ser un programa menos adecuado para la GPU. Se examina la cantidad de sincronización necesaria entre hilos. Las GPU funcionan mejor cuando los hilos pueden ejecutarse de manera independiente y no requieren una sincronización frecuente.

- Tamaño de los datos: Se considera el tamaño de los conjuntos de datos involucrados. Las GPU son eficientes cuando se procesan grandes conjuntos de datos.

- Paralelismo de datos: Se busca la presencia de paralelismo de datos, lo que significa que múltiples elementos de datos se procesan de la misma manera. Esto es común en operaciones vectoriales o matriciales.

- Paralelismo de tareas: Se evalúa si hay tareas independientes que pueden ejecutarse en paralelo. Esto es especialmente relevante para problemas que se pueden dividir en subproblemas independientes.

Teniendo en cuenta estos criterios se definen tres categorías de prioridad de paralelización:

- Alta prioridad: Las partes del código que tienen un alto potencial de paralelismo y pueden ejecutarse en la GPU de manera eficiente se consideran de alta prioridad. Esto podría incluir operaciones matriciales, cálculos intensivos en términos de cómputo y grandes conjuntos de datos.

- Media prioridad: Las partes del código que tienen cierto potencial de paralelismo pero pueden requerir alguna sincronización o tienen menos datos para procesar se consideran de prioridad media. Esto podría incluir operaciones con alguna dependencia de datos o paralelismo de tareas limitado.

- Baja prioridad: Las partes del código que tienen dependencias de datos significativas, sincronización frecuente o no se benefician significativamente de la paralelización se consideran de baja prioridad. Estas partes pueden ser menos adecuadas para la GPU.

La estrategia utilizada fue portar las funciones directamente y ejecutarlas con un solo hilo, y luego ir paralelizando secciones.
Para la primera versión paralelizamos la función `fijarCajasLaminares` y la función `intercambiar`, las cuales, según los resultados del profiling podrían ser clave para la mejora de los tiempos debido a su uso, su peso computacional, y su aptitud para ser paralelizada en GPU.

Para este caso se usó una grilla unidimensional de tamaño igual al número de trayectorias y dimensión del bloque igual a 32 hilos, como se puede ver en 4.1. El kernel *kernel_resolver* se encargará de realizar operaciones en paralelo en la GPU en función de la configuración de la grilla y el tamaño del bloque.

```
#define BLOCK_SIZE 32

// Configuro la grilla
const dim3 DimGrida(NTrayectorias, 1);
const dim3 DimBlocka(BLOCK_SIZE, 1);

// Ejecuto el kernel
kernel_resolver<<< DimGrida, DimBlocka, 0, 0 >>>(d_simplex_array,
    NTrayectorias);
```

```
__global__ void kernel_resolver(TDAOfSimplexGPUs simplex_array, int
     NTrayectorias) {

        if (threadIdx.x == 0)
            resolver_gpu(simplex_array[blockIdx.x]);

}
```

Figura 4.1: Primera versión donde solo se usó el primer hilo del bloque.

```
__global__ void kernel_resolver_etapa_cnt_igcount( simplex_array,
    d_svars) {
        resolver_etapa_cnt_igcount(simplex_array[blockIdx.x],
            d_svars[blockIdx.x]);
}

__global__ void kernel_resolver_etapa_fijar_cajlam( simplex_array,
     d_svars) {
        resolver_etapa_fijar_cajlam(simplex_array[blockIdx.x],
            d_svars[blockIdx.x]);
}

__global__ void kernel_resolver_fijar_variables( simplex_array,
    d_svars, NTrayectorias) {
        int index = blockIdx.x*blockDim.x + threadIdx.x;
        if (index < NTrayectorias) resolver_fijar_variables(
            simplex_array[index], d_svars[index]);
}

__global__ void kernel_resolver_enfilar_variables_libres(
    simplex_array, d_svars, NTrayectorias) {
        int index = blockIdx.x*blockDim.x + threadIdx.x;
        if (index < NTrayectorias)
            resolver_enfilar_variables_libres(simplex_array[index],
             d_svars[index]);
}

__global__ void kernel_resolver_igualdades( simplex_array, d_svars,
      NTrayectorias) {
        int index = blockIdx.x*blockDim.x + threadIdx.x;
        if (index < NTrayectorias) resolver_igualdades(
            simplex_array[index], d_svars[index]);
}

__global__ void kernel_resolver_reordenar_por_factibilidad(
    simplex_array, d_svars, NTrayectorias) {
        int index = blockIdx.x*blockDim.x + threadIdx.x;
        if (index < NTrayectorias)
            resolver_reordenar_por_factibilidad(simplex_array[index
            ], d_svars[index]);
}

__global__ void kernel_resolver_paso_iterativo( simplex_array,
    d_svars) {
        resolver_paso_iterativo(simplex_array[blockIdx.x], d_svars[
            blockIdx.x]);
}
```

Figura 4.2: Segunda versión donde se paralelizan todas las funciones.

Luego está la versión donde se paralelizan todas esas funciones. El código de la figura 4.2, es parte de la implementación en CUDA, donde se están definiendo varios *kernels* que ejecutan en la GPU y realizan operaciones relacionadas con

la resolución del método Simplex.

4.1.1. Resumen implementación Simplex SimSEE

- `intercambiar`

  La función realiza las operaciones necesarias para llevar la matriz a su siguiente iteración y avanzar hacia una solución óptima en el método Simplex, mediante el intercambio de filas y la realización de operaciones de combinación lineal para transformar la matriz en cada paso.

  Las acciones principales que realiza son las siguientes:

  1. Se obtiene el valor del elemento pivote de la matriz ubicado en la fila *kfil* y columna $jcol$. El pivote es el elemento que se utiliza para realizar las operaciones de intercambio y transformar la matriz.
  2. Se calcula el inverso del valor del pivote para su uso en las operaciones de intercambio.
  3. Se recorren las filas de la matriz antes de la fila $kfil$. Para cada fila, se calcula un factor de multiplicación m utilizando el valor del elemento en la columna $jcol$ y el inverso del pivote. Luego, se realizan operaciones de combinación lineal en todas las columnas de la fila utilizando este factor m para transformar la matriz y hacer ceros en la columna $jcol$ de estas filas.
  4. Se repite el paso 3 para las filas después de la fila $kfil$.
  5. Se completa la fila $kfil$ multiplicando todos los elementos de la fila por el inverso del pivote, de modo que el valor del pivote quede igual a -1.
  6. Se actualizan los índices *top* y $left$ para mantener el seguimiento de las filas y columnas pivote.

- `fijarCajasLaminares`

  Esta función busca identificar y fijar las variables que cumplen con las características de cajas laminares, lo que permite optimizar y simplificar el proceso de resolución del problema de programación lineal en el algoritmo Simplex.

  El término «cajas laminares» se refiere a un conjunto de variables que están restringidas por límites muy cercanos entre sí, lo que hace que su rango de valores posibles sea extremadamente pequeño.

  1. Recorre todas las variables (columnas) de la matriz Simplex, excepto la última que representa la función objetivo (variable Z).
  2. Verifica si el indicador `flg_x` de la variable es igual a 0, 1 o -1. Si el valor absoluto de `flg_x` es menor a 2, significa que la variable aún no ha sido procesada como una caja laminar.

3. Compara la diferencia entre los límites superior e inferior (`x_sup` y `x_inf`) de la variable. Si esta diferencia es menor que un valor predefinido (`CasiCero_CajaLaminar`), se considera que la variable es una caja laminar.
4. Actualiza el indicador `flg_x` de la variable para marcarla como fija (2 o -2) dependiendo si el límite superior o inferior es cercano al valor de la variable.
5. Incrementa el contador `cnt_varfijas` que lleva la cuenta de las variables fijas.
6. Retorna la cantidad de variables que se han fijado como cajas laminares.

- `intercambioColumnas`

  Función auxiliar que se utiliza para realizar intercambios de columnas en la matriz del algoritmo Simplex, lo que permite realizar operaciones de pivote y buscar soluciones óptimas de manera eficiente.

  1. Recorre todas las filas de la matriz y realiza el intercambio de los valores de las columnas pasadas como parámetro.
  2. Intercambia los índices de las variables en el vector `top` que representa la estructura de la tabla Simplex.

- `intercambioFilas`

  Función auxiliar que se utiliza para realizar intercambios de filas en la matriz del algoritmo Simplex, lo que permite realizar operaciones de pivote y buscar soluciones óptimas de manera eficiente.

  1. Recorre todos los elementos de las filas pasadas como parámetro y realiza el intercambio de sus valores.
  2. Intercambia los índices de las restricciones en el vector `left`, que representa la estructura de la tabla Simplex.

- `Cambiar_borde_de_caja`

  Se encarga de realizar un cambio de variable en el algoritmo Simplex, específicamente en el contexto de una caja laminar. En resumen, realiza las siguientes acciones

  1. Calcula el índice `ix` correspondiente a la variable `x` relacionada con la restricción que se está analizando.
  2. Recorre todas las variables de la matriz (excepto la última) y multiplica sus valores por -1. Esto es parte del proceso de cambio de variable.

3. Modifica el último valor de la fila `k_fila` de la matriz, para reflejar el cambio de variable. El nuevo valor será igual a la diferencia entre el límite superior `x_sup` de la variable `x` y el valor anterior de esa posición en la matriz.
4. Verifica si el indicador `flg_x` de la variable `x` es diferente de 1 (es decir, si está marcado de alguna forma). Si no es igual a 1, muestra un mensaje de advertencia.
5. Finalmente, cambia el indicador `flg_x` de la variable `x` al valor opuesto (por ejemplo, de 1 a -1 o de -1 a 1).

.

- `buscarMejorPivoteEnCol`

  El algoritmo recorre las filas de la matriz y compara el valor absoluto de cada elemento de la columna con una variable `a`, que inicialmente está en 0. Si encuentra un elemento con un valor absoluto mayor que `a`, actualiza `a` con el valor absoluto del nuevo elemento y guarda el índice de la fila en la variable `res`.

  Al finalizar la búsqueda, la función devuelve el índice de la fila que contiene el mejor pivote, o -1 si no se encontró un pivote válido (todos los elementos de la columna son iguales a cero).

  En resumen, esta función encuentra el mejor pivote en una columna dada y lo devuelve para su posterior uso en el algoritmo del Simplex.

- `pasoBuscarFactibleIgualdad4`

  Esta función se utiliza en el algoritmo del Simplex para buscar un pivote factible en el conjunto de igualdades no resueltas. El objetivo es encontrar el mejor pivote (mayor valor absoluto) dentro de las filas y columnas que contienen ceros y valores cercanos a cero.

  El algoritmo comienza inicializando contadores y variables, y luego realiza un recorrido por las filas y columnas en el rango correspondiente a las igualdades no resueltas. Durante este recorrido, se cuentan la cantidad de ceros en cada fila y columna, y se busca el elemento con el mayor valor absoluto, que será el posible pivote.

  Luego, se verifica si el valor absoluto del posible pivote es lo suficientemente grande para ser considerado un pivote válido. Si es así, se realiza un intercambio de filas y columnas para ubicar el posible pivote en la posición adecuada. En caso contrario, se devuelve un valor negativo indicando que no se encontró un pivote factible.

- `reordenarPorFactibilidad`

  Esta función se utiliza en el algoritmo del Simplex para reordenar las restricciones según su factibilidad. Primero, recorre todas las restricciones para verificar si alguna de ellas viola alguna restricción de cota superior y,

en ese caso, realiza los cambios necesarios para que la restricción violada quede explícita y no afecte la factibilidad del problema.

Luego, se mueven todas las restricciones violadas al final del conjunto de restricciones para que queden agrupadas. Se lleva un contador `cnt_RestrInfactibles` que registra la cantidad de restricciones violadas.

- `Locate_zpos`

  Esta función se utiliza en el algoritmo del Simplex para buscar la columna que tiene el valor positivo más grande en la última fila (fila Z) de la tabla Simplex. Retorna el número de columna si encuentra algún valor positivo, o -1 si todos los valores en la última fila son negativos.

- `Mejorpivote(`q`,` k_{max}`)`

  Se encarga de buscar el mejor pivote en el algoritmo del Simplex.

  La función recorre las filas de la tabla Simplex y busca el mejor pivote en la columna q para optimizar la fila hasta k_{max}. También verifica si la variable correspondiente a la columna q tiene restricción de cota superior y si es así, agrega una fila fantasma como candidato a pivote.

  Los valores del pivote se comparan usando la relación b_i/a_{iq}, donde b_i es el término independiente y a_{iq} es el coeficiente de la columna q en la fila i. Se elige el pivote que maximiza esta relación, siempre que b_i/a_{iq} y a_{iq} sean negativos. Si se encuentra una fila con restricción de cota superior, también se compara con el pivote actual para verificar si es mejor.

  La función retorna el índice de la columna que contiene el mejor pivote. También establece los booleanos filaFantasma y colFantasma para indicar si se ha encontrado una fila o columna fantasma respectivamente.

  La función también realiza algunas verificaciones y considera casos especiales para variables con restricciones de cota superior.

- `Cambio_var_cota_sup_en_columna`

  Se encarga de realizar un cambio de variable en la columna q cuando corresponde a una variable con restricción de cota superior en el algoritmo del Simplex.

- `locate_qOK`

  Se encarga de encontrar el valor óptimo de la variable q en una fase específica del proceso. El resultado es el índice de la mejor variable q que cumple ciertas condiciones específicas.

- `test_qOK`

  Ayuda a determinar si una columna q es factible para resolver una fila p específica, lo que es esencial para encontrar la solución óptima del problema de programación lineal.

4.2. Descripción variante 2 Simplex Big M

Se construye el problema de forma estándar basándonos en el libro (H.S. Kasana, 2004).

Para esta versión se hace foco en los dos siguientes aspectos:

- Lograr un algoritmo con operaciones más simples y más paralelizables. En esta versión se usa una mayor cantidad de datos, por ejemplo representar restricciones de cota superior en la matriz del problema, y usar variables artificiales para lograr que el algoritmo soporte restricciones de igualdad o mayor igual.
- Trabajar sobre una estructura de datos menos fragmentada en la memoria de forma que sea más apta para una GPU, ya sea para su transferencia como para el acceso.

Considerando que:

- Se desea estudiar experimentalmente si en este caso la paralelización de operaciones simples sobre un conjunto de datos más extenso tiene mejor performance que operaciones complejas (con baja aptitud para ser paralelizadas) sobre un conjunto de datos más reducido en una GPU.
- Se utilizará una estructura de datos tabular (tabloide) para representar todos los datos del problema, esto permite la transferencia de todos los datos hacia la GPU en un paso, lo cual es mucho más rápido que transmitir los diferentes datos (configuración, vectores de cotas, vectores de flags y matriz principal) de a uno (Mark Harris, 2012a).
- Tener todos los datos que usará un bloque de forma consecutiva en la memoria podría mejorar los tiempos de acceso.

En este proyecto se implementa la función TMIPSimplex.toStruct en umsimplex.pas (Archivo de SimSEE), la cual mapea la estructura usada actualmente por SimSEE para representar el problema en esta estructura de tabloide. Esta función también describe cómo se deben ubicar los datos a partir del problema que se quiere representar.

Se espera que en una etapa de producción, SimSEE complete esta nueva estructura de tabloide directamente desde la carga de la sala y así ahorrar este paso extra que es crear la estructura (actual) del problema y luego traducirla a esta nueva estructura de tabloide.

Por su parte, el parámetro M se obtiene de forma experimental, y para todos los distintos ejemplos de casos reales que se probaron, el valor 1.0E+15 resulta en el mejor desempeño del programa, por lo tanto podemos esperar no tener que

reajustar este parámetro para nuevos problemas, al menos dentro del espacio de problemas que se está abordando.

4.2.1. Resumen implementación Simplex Big M

- `moverseASolFactible`

  Realiza los cambios necesarios en la solución actual para llevarla hacia una solución factible dentro del algoritmo Simplex. Estos cambios incluyen ajustar los valores de las variables de holgura y la orientación de las restricciones para que la solución cumpla con todas las restricciones del problema y pueda avanzar hacia una solución óptima.

- `agregarRestriccionesCotaSup`

  Se encarga de agregar restricciones de cota superior a la estructura de datos TSimplexGPUs para adaptar el problema a la formulación requerida por el algoritmo Simplex. Cada restricción de cota superior se agrega como una fila adicional en la matriz del problema, y se ajustan los valores de las variables y los flags correspondientes para asegurar que el algoritmo pueda resolver adecuadamente el problema en búsqueda de una solución óptima.

- `agregarVariablesHolguraArtificiales`

  Se encarga de agregar las variables de holgura y artificiales necesarias a la estructura de datos TSimplexGPUs, ajustando la matriz del problema y las funciones objetivo para adaptar el problema a la formulación requerida por el algoritmo Simplex. Estas variables adicionales permiten que el algoritmo resuelva problemas de programación lineal con restricciones en igualdad y desigualdad y encuentre una solución óptima.

- `resolver_simplex_big_m`

  Implementa el algoritmo Simplex con la técnica Big M para encontrar la solución óptima de un problema de programación lineal. Realiza iteraciones para seleccionar y cambiar las variables que formarán la base hasta alcanzar una solución óptima o hasta que se cumpla una condición de parada definida por el número máximo de iteraciones permitidas.

  Explicación paso a paso:

  1. La función declara e inicializa tres variables: `zpos`, `qpos` e `it`.
     `zpos` y `qpos` representan las posiciones de las variables que serán seleccionadas en la siguiente iteración para optimizar la función objetivo y realizar el intercambio, respectivamente. `it` es un contador que lleva el registro del número de iteraciones realizadas en el algoritmo. Se realiza un bucle `do-while` que ejecuta el algoritmo Simplex hasta que se cumpla una condición de parada.

2. Dentro del bucle, se llama a la función `locate_min_dj(simplex)` para obtener la posición de la variable con el valor más negativo en el vector de costos reducidos (coeficientes de la función objetivo). Esta variable se selecciona para optimizar la función objetivo en la siguiente iteración.
3. Se verifica si `zpos` (la posición de la variable seleccionada para la optimización) es menor que 0. Si es así, significa que se ha encontrado una solución óptima (los costos reducidos son todos no negativos o nulos), y el algoritmo termina con éxito.
4. Si `zpos` es mayor o igual a 0, se procede a buscar la posición de la variable que será la siguiente a entrar en la base (variable a intercambiar) utilizando la función `locate_min_ratio(simplex, zpos)`. Esta función busca la posición de la variable con el cociente más pequeño entre el término independiente y el coeficiente de la columna correspondiente en la fila `zpos`. Esta variable será intercambiada con la variable que sale de la base (la variable correspondiente a la posición `zpos`).
5. Se verifica si `qpos` (la posición de la variable seleccionada para el intercambio) es menor que 0. Si es así, significa que no se encontró una posición válida para el cociente, y el algoritmo termina con un error.
6. Si se encontraron posiciones válidas tanto para la variable a optimizar como para la variable a intercambiar, se llama a la función `intercambiarvars(simplex, qpos, zpos)`, que realiza el intercambio de variables en la estructura Simplex.
   Después del intercambio, se incrementa el contador de iteraciones `it`.
7. Se verifica si se alcanzó el límite máximo de iteraciones. Si es así, se imprime un mensaje indicando que se ha alcanzado el número máximo de iteraciones permitidas, y el algoritmo termina.
8. Si el límite de iteraciones no se ha alcanzado, el bucle `do-while` continúa y se repiten los pasos desde la búsqueda de la próxima variable a optimizar.

4.2.2. Implementación en GPU

A continuación se describe la metodología utilizada para paralelizar los distintos pasos del algoritmo descriptos en la sección 2.2.3.

El **primer paso** del algoritmo, moverse a una solución factible, haciendo b (el vector de términos independientes) positivo. Para ello se debe chequear si algún b_i es negativo y en ese caso multiplicar toda la fila por -1 y cambiar la restricción de $\leq$ a $\geq$ o viceversa. Este es un paso altamente paralelizable y es atendido por un kernel independiente que utiliza una grilla de hilos de tamaño 32×8, tamaño obtenido experimentalmente, y un bloque por cada trayectoria

(Simplex a resolver).

Los **pasos 2, 3, 4 y 5** del algoritmo, agregar variables de holgura y artificiales, y para cada variable artificial el coeficiente -M en la función objetivo, son pasos que tienen baja aptitud para ser paralelizados, debido a que el algoritmo tiene dependencias de datos, lo que significa que las iteraciones de los pasos (bucles) posteriores dependen de los resultados de las iteraciones anteriores. Es por esto que se decidió que cada Simplex a resolver sea atendido por un único hilo, paralelizando así a nivel de trayectorias pero secuencial dentro de cada Simplex.

El **paso 6**, reducir los coeficientes (-M) de las variables artificiales a 0 usando operaciones lineales sobre la matriz, es altamente paralelizable, ya que en forma es una reducción de toda la matriz sobre la función objetivo, similar a sumar todos los valores de cada columna y guardarlos en el vector función objetivo. Para ser más precisos $z = z + c \times A$, siendo c un vector en donde la entrada c_i es $-M$ si la variable básica de la fila i es una variable artificial y 0 en otro caso. Paralelamente, esto se resuelve utilizando una grilla de hilos de tamaño 32×8 (tamaño obtenido experimentalmente) y un bloque por cada trayectoria. Cada proceso lee el c_i correspondiente y los valores de la columna de la matriz correspondiente acumulando los valores en un registro y luego escribe estos registros en un tile, que luego se reduce utilizando una reducción intercalada y finalmente el resultado se escribe en la función objetivo en memoria global.
Esta recorrida tiene la ventaja de que el acceso es coalesced en memoria global y no tiene conflicto de bancos en memoria compartida, tiene buena carga de trabajo por procesador y usa un registro (rápido acceso) para llevar la reducción (suma) de los valores que le corresponden según su posición en el bloque.
Los algoritmos de reducción que evaluamos se encuentran en (Mark Harris, 2012b).

El **paso 7** Corresponde a resolver el algoritmo Simplex usual. A continuación se detalla la metodología usada para resolver las tres etapas dentro de cada bucle del algoritmo:

- **Búsqueda en z del mínimo positivo**, se utiliza una reducción Secuencial en memoria compartida que mejora un poco los tiempos respecto a la reducción Intercalada. Se decidió no profundizar más en mejorar esta reducción sustituyéndola por otra más compleja, dado que el esfuerzo es considerable y la potencial mejora en rendimiento en este caso es marginal.

- **Búsqueda del ratio mínimo**, se utiliza la misma reducción que el anterior caso y la diferencia de rendimiento es aún menor, dado que la reducción es más pequeña y que lo pesado de esta búsqueda son los accesos no coalesced (por columna) necesarios que se hacen sobre la matriz localizada en memoria global.

La **función intercambiar**, es la encargada de implementar la situación en la cual una variable sale de la base y otra entra a la base. Esta función tiene alta aptitud para la paralelización debido a que se trata de transformaciones lineales sobre matrices con bifurcaciones mínimas. La matriz extendida (con la fila de términos independientes b) se divide en bloques del tamaño del bloque de hilos y cada hilo atiende una entrada, ya que al tomar el tamaño en la dimensión X del bloque de hilos múltiplo de 32 se permite un acceso coalesced a la matriz en memoria global. Se entiende que el uso de un tile en memoria compartida no conlleva ninguna ventaja en este paso ya que las entradas de la matriz obtenidas de memoria compartida se acceden solo una vez (se realiza una operación de suma y luego se guarda este resultado en la entrada). En cambio, sí se usa un vector de memoria compartida para guardar la fila a intercambiar porque sus datos son accedidos repetidas veces para realizar las transformaciones lineales, tantas veces como columnas de la matriz por cada entrada de este vector, lo cual vuelve conveniente cargarlos en una memoria de acceso más rápido como memoria compartida.

Se ejemplifica la estructura de la tabla con el siguiente problema

Maximizar:

$$z = -x_1 - 3x_2 - 2x_3$$

Sujeto a:

$$x_1 + x_2 + x_3 \ >= 0{,}5$$

$$x_1 + x_2 \ = 0{,}7$$

$$-x_1 + x_3 \ >= 2{,}1$$

$$x_1 \ <= 12, x_2 \ <= 12, x_3 \ <= 10$$

$$x_1, x_2, x_3 \ >= 0$$

A continuación se brinda información sobre el significado de las variables representadas en la tabla de la figura 4.4:

- NV (Número de Variables): Indica que hay 3 variables en el problema para este ejemplo.
- NR (Número de Restricciones): Indica que hay 3 restricciones en el problema para este ejemplo.
- TL: Largo de la tabla.
- Columnas de x_1, x_2, y x_3: Estas columnas representar las variables de decisión del problema, que son x_1, x_2 y x_3 en este caso.

NV	NR	TL		x_1	x_2	x_3	
3	3	15		-1	-3	-2	función a minimizar
				1	1	1	flag cota superior
				12	12	10	cota superior
				0	-6	-5	cota inferior
							tipo de variable
							top
	0		0.5	1	1	1	
	2		0.7	1	1	0	
	0		2.1	-1	0	1	
left	flagy	CB	Xb	x_1	x_2	x_3	

Figura 4.3: Ejemplo de la estructura de la tabla

- Fila de coeficientes en la función objetivo: Esta fila representa los coeficientes de la función objetivo que se está minimizando. Por ejemplo, el coeficiente de x_1 es -1, el de x_2 es -3, y el de x_3 es -2.
- Función a minimizar: Esta fila representa la función objetivo que se está minimizando.
- Flag cota superior: Toma los valores 0 y 1 indicando si la variable tiene cota superior.
- Cota superior: Aquí están las cotas superiores específicas para las variables.
- Cota inferior: Similar a la fila anterior, pero para cotas inferiores.
- Tipo de variable: Toma los valores 0, 1 y 2 para representar los 3 tipos de variables.
- Top y left: Se utilizan para llevar conteo de las variables que pertenecen a la base.
- Flagy: Toma los valores 0, 1 y 2 para representar las 3 tipos de restricciones.
- Cb: Su valor es M si la variable de la base asociada a esa fila es artificial y 0 en otro caso, mantener este valor ayuda a reducir accesos a memoria en los cálculos.
- Xb: Es el término independiente (es un vector) y además es solución de las variables de la base.

Luego de creada esta estructura en Pascal, se pasa la misma para que la retome la GPU y termine de cargar el resto de los datos a la misma, agregando

las variables de holgura y artificiales antes mencionadas.

En la figura 4.4 se puede observar un ejemplo de la tabla con todos los valores cargados

NV	NR	TL		x_1	x_2	x_3									
3	3	15	0	-1	-3	-2	0	-M	-M	0	-M	0	0	0	función a minimizar
11	6	0	0	1	1	1	0	0	0	0	0	0	0	0	flag cota superior
0	0	0	0	12	12	10	0	0	0	0	0	0	0	0	cota superior
0	0	0	0	0	-6	-5	0	0	0	0	0	0	0	0	cota inferior
0	0	0	0	0	0	0	1	2	2	1	2	1	1	1	tipo de variable
0	0	0	0	4	5	6	7	8	9	10	11	12	13	14	top
8	0	M	0.5	1	1	1	-1	1	0	0	0	0	0	0	
9	2	M	0.7	1	1	0	0	0	1	0	0	0	0	0	
11	0	M	2.1	-1	0	1	0	0	0	-1	1	0	0	0	
12	1	0	12	1	0	0	0	0	0	0	0	1	0	0	
13	1	0	12	0	1	0	0	0	0	0	0	0	1	0	
14	1	0	10	0	0	1	0	0	0	0	0	0	0	1	
left	flagy	Cb	Xb	x_1	x_2	x_3									

Figura 4.4: Ejemplo de la estructura de la tabla

La implementación se basa en 6 funciones principales. La primera a ser ejecutada tiene como propósito manejar mejor la estructura. Se define la estructura de datos TSimplexGPUs y la función `desestructurarTabloide` la cual mapeará las diferentes partes del tabloide a la estructura, de modo que sea más ordenado, legible y modificable para el programador y de más fácil acceso para los procesos. Esta estructura y el mapeo se puede ver en la figura 4.4.

Luego de armar la matriz (Tabloide), es necesario moverse a una solución factible. Para esto ejecutamos `moverseASolFactible`, que mueve las restricciones de mayor igual (>=) a restricciones de menor igual (<=) y viceversa de modo de que la columna b sea positiva.

Si las restricciones de un problema de optimización son mutuamente contradictorias no hay puntos que satisfagan todas las restricciones y, por lo tanto, la región factible es el conjunto nulo. En este caso, el problema no tiene solución y se dice que es inviable.

Luego se agregan las restricciones de cota superior `agregarRestricciones-CotaSup` y las variables de holgura y artificiales `agregarVariablesHolguraArtificiales`. También se introduce en la función objetivo con coeficiente cero ya que no influye en el valor de la función objetivo.

Luego usando la función `resolverVariablesArtificialesEnZ` se realizan transformaciones lineales sobre la función objetivo de forma de reducir los co-

```
void resolver_cpu(TabloideGPUs &tabloide) {
        TSimplexGPUs simplex = desestructurarTabloide(tabloide);
        moverseASolFactible(simplex);
        agregarRestriccionesCotaSup(simplex);
        agregarVariablesHolguraArtificiales(simplex);
        resolver_simplex_big_m(simplex);
}
```

Figura 4.5: Parte del código de versión Big M CPU, ejecución secuencial de las funciones

```
const dim3 DimGrid(NTrayectorias, 1);

const dim3 DimBlock_e1(32, 8);
kernel_resolver_etapa_moverse_a_sol_factible<<< DimGrid,
    DimBlock_e1, 0, 0 >>>(d_simplex_array);

const dim3 DimBlock_e2(1, 1);
kernel_resolver_etapa_agregar_restricciones_cota_sup<<< DimGrid,
    DimBlock_e2, 0, 0 >>>(d_simplex_array);

const dim3 DimBlock_e3(1, 1);
kernel_resolver_etapa_agregar_variables_holgura_artificiales<<<
    DimGrid, DimBlock_e3, 0, 0 >>>(d_simplex_array);

const dim3 DimBlock_e4(32, 16);
kernel_resolver_etapa_resolver_variables_artificiales_en_z<<<
    DimGrid, DimBlock_e4, 0, 0 >>>(d_simplex_array);

const dim3 DimBlock_e4(32, 16);
kernel_resolver_simplex_big_m<<< DimGrid, DimBlock_e4, 0, 0 >>>(
    d_simplex_array);
```

Figura 4.6: Parte del código de versión Big M GPU, donde se paralizan todas las funciones

eficientes M de las variables artificiales a 0.

Por último, se llama a la función `resolver_simplex_big_m`. Esta función es la encargada de ejecutar el algoritmo Simplex.
Primero se busca el pivot, que consiste en encontrar la columna que en la última fila (fila z) tenga el valor positivo más grande y retorna el número de columna si lo encontró (-1 en el caso de que sean todos < 0). Este paso se da en el Simplex para minimizar; en el de maximizar busca el menos negativo. Luego se pasa a buscar la posición en la fila así finalmente poder hacer una combinación lineal entre las filas.

4.3. Descripción variante 3 Simplex Two Phases

El problema se expresa de forma estándar como en la versión anterior basándose en el libro (H.S. Kasana, 2004).

Esta versión mantiene el mismo enfoque, pero tiene la ventaja de que no requiere la introducción del parámetro M.

Dado que esta versión, al igual que Big M, usa la forma estándar, el tabloide también se construye con la función `TMIPSimplex.toStruct`.

4.3.1. Implementación en GPU

A continuación se describe la metodología utilizada para paralelizar los distintos pasos del algoritmo descriptos en la sección 2.2.4.

El **primer paso** del algoritmo, moverse a una solución factible, y los **pasos 2, 3, 4 y 5** del algoritmo, agregar variables de holgura y artificiales, se realizan utilizando la metodología descrita en la Sección 4.2.

Los **pasos 6 y 7** phase 1 y phase 2 se realizan en el mismo kernel ya que ambas grillas pueden compartir las mismas dimensiones y ahorrarse los tiempos de carga de las instrucciones de un kernel.

Para la función `resolverZj_Cj_ph()`, la cual reduce la matriz en el vector Z, se utiliza una reducción Intercalada, la cual, como mencionamos antes en la Sección 4.2, no incurre en conflictos de bancos (Mark Harris, 2012b).

Búsqueda en z del mínimo positivo, **Búsqueda del ratio mínimo**, y La **función intercambiar** utilizan el mismo enfoque que el paso 6 de la sección 4.2.2.

Capítulo 5

Evaluación experimental

Este capítulo detalla las evaluaciones experimentales de los prototipos, analizando y comparando su desempeño. La evaluación abarca las variantes GPU y CPU. Todos los tiempos reportados en este informe se expresan en milisegundos (ms) y corresponden al promedio de 10 ejecuciones independientes.

5.1. Evaluaciones Simplex

Para la evaluación de los tiempos, tanto para la versión CPU como la GPU, se usó la función gettimeofday y las funciones de CudaEvent. Se utilizan dos métodos diferentes, CudaEvent y gettimeofday, para medir los tiempos con el propósito de obtener mediciones precisas y completas de los tiempos de ejecución de las diferentes versiones de código en GPU. Cada método tiene sus ventajas y desventajas, y al combinar ambos métodos se puede verificar si los resultados son consistentes. Si hay una diferencia significativa entre los dos métodos, podría indicar un problema en la medición.

CudaEvent describe las funciones de gestión de eventos de la interfaz de programación de aplicaciones de tiempo de ejecución de CUDA. Los eventos CUDA son temporizadores de GPU que permiten medir el tiempo de ejecución de una operación en la GPU con alta precisión. Se definen cuatro funciones:

- `CLK_CUEVTS_INIT`: Se crean dos objetos de evento. El evento de inicio y el de fin (`evt_start, evt_stop`)
- `CLK_CUEVTS_START`: Se graba el evento `evt_start` antes de la llamada a la función que se quiere medir.
- `CLK_CUEVTS_STOP`: Se graba el evento `evt_stop` después de la llamada a la función que se quiere medir.

- `CLK_CUEVTS_ELAPSED`: Calcula el tiempo transcurrido entre dos eventos (en milisegundos con una resolución de alrededor de 0,5 microsegundos).

Por otro lado el método que usa la función gettimeofday consiste en las siguientes funciones:

- `CLK_POSIX_INIT`: Se declaran las variables.
- `CLK_POSIX_START`: Se usa la función `gettimeofday()` que retorna el número de segundos y microsegundos transcurridos desde un punto en el pasado. Se ejecuta antes de la función que se quiere medir.
- `CLK_POSIX_STOP`: Se ejecuta después de la función que se quiere medir.
- `CLK_POSIX_ELAPSED`: Calcula el tiempo obtenido entre el tiempo de inicio y el tiempo final.

El uso de CudaEvent proporciona una medición precisa y detallada del tiempo de ejecución en la GPU, mientras que gettimeofday() ofrece una alternativa más general para medir el tiempo. Al utilizar ambos métodos, se puede verificar si los resultados obtenidos son consistentes entre sí.

Para la evaluación experimental en el presente estudio, se utilizaron tres tamaños diferentes de matrices como ejemplos de problema a resolver. Estos tamaños se denominaron Base, Grande y Extra Grande, y se diferencian en la cantidad de variables y restricciones que contienen.

La matriz Base consiste en un problema de optimización con 51 variables y 13 restricciones. Este tamaño se seleccionó como punto de partida para evaluar el desempeño del sistema SimSEE y los métodos de optimización empleados.

La matriz Grande se utiliza como una ampliación del tamaño de la matriz base y presenta un problema con 92 variables y 16 restricciones. Esta matriz más grande se incluye en el estudio para analizar cómo el sistema SimSEE y los métodos de optimización se comportan y escalan en problemas más complejos.

Por último, se emplea la matriz Extra Grande, que representa un problema de mayor magnitud. Esta matriz contiene 162 variables y 21 restricciones, lo que plantea un desafío adicional en términos de complejidad y requerimientos computacionales.

Estos tres tamaños de matriz proporcionan una variedad de casos de prueba para evaluar el rendimiento del sistema SimSEE en diferentes niveles de complejidad. Al utilizar matrices con distintas dimensiones, se busca comprender cómo el sistema responde y se comporta en problemas de diferentes tamaños y estructuras.

Además, cada versión de matriz se ejecutó para 100, 500, 1000, 2000, y 4000 trayectorias.

En las siguientes tablas se presentan los tiempos obtenidos utilizando la función gettimeofday, con el fin de evitar sobrecargar el informe con una gran cantidad de valores. Sin embargo, es importante destacar que también se realizaron mediciones de tiempo utilizando CudaEvent para verificar que los resultados fueran similares. La utilización de ambas funciones de medición permitió asegurar la precisión y consistencia de los tiempos registrados.

5.2. Evaluación variante 1: Simplex SimSEE

5.2.1. CPU

La versión `v1_simplex_simsee_cpu_prog` implementa el algoritmo Simplex usado en SimSEE en una biblioteca para CPU en lenguaje C.

Matriz \Trayectorias	100	500	1000	2000	4000
Base	9,82	50,31	92,02	203,52	461,79
Grande	28,12	133,01	268,94	745,75	2.782,59
Extra Grande	98,98	513,41	1.486,27	4.488,95	9.141,48

Cuadro 5.1: Tiempos de ejecución en milisegundos SimSEE CPU

5.2.2. GPU

La versión `v2_simplex_simsee_gpu_prog` implementa el algoritmo Simplex usado en SimSEE para GPU en lenguaje CUDA. El programa contiene ejemplos usados por SimSEE que se pueden correr directamente ejecutando el programa sin parámetros.

Matriz \Trayectorias	100	500	1000	2000	4000
Base	13,00	8,06	11,00	23,70	42,41
Grande	31,70	52,81	73,56	149,62	204,38
Extra Grande	75,52	112,99	140,21	185,95	261,36

Cuadro 5.2: Tiempos de ejecución en milisegundos SimSEE GPU

Se puede observar, analizando los tiempos que se obtuvieron, que a medida que aumenta el tamaño de la matriz y el número de trayectorias, la ventaja de la GPU sobre la CPU se vuelve más evidente. En la configuración Extra Grande, la GPU es significativamente más rápida que la CPU. El aumento en el tamaño de la matriz afecta más los tiempos de ejecución que el incremento en el número

de trayectorias. Esto se debe a que un problema de mayor tamaño incrementa el trabajo secuencial necesario. Por otro lado, el aumento en el número de trayectorias se beneficia de la ejecución paralela en las GPU, siempre y cuando la GPU no esté saturada.

5.3. Evaluación variante 2: Simplex Big M

5.3.1. CPU

La versión `v3_simplex_big_m_cpu_prog` implementa el algoritmo Simplex variante Big M en un programa C para CPU que contiene ejemplos usados por SimSEE que se pueden correr directamente ejecutando el programa. Y los tiempos que se obtuvieron se muestran el cuadro 5.3

Matriz \Trayectorias	100	500	1000	2000	4000
Base	134,59	470,40	1.636,98	2.023,16	4.078,80
Grande	673,20	2.310,82	4.521,99	9.188,17	18.482,20
Extra Grande	2.427,52	11.509,51	22.251,57	45.905,83	89.536,71

Cuadro 5.3: Tiempos de ejecución en milisegundos Big M CPU

5.3.2. GPU

La versión `v9_simplex_big_m_improves_gpu_prog` implementa una mejora del algoritmo Simplex Big M Versión 2 en un programa en CUDA para GPU, que incluye los ejemplos utilizados por SimSEE.

Para determinar la mejor configuración en términos de tamaño de bloques para la versión Simplex Big M, se llevaron a cabo pruebas de rendimiento utilizando el ejemplo Extra Grande con 1000 trayectorias. Se evaluaron diferentes tamaños de bloques, incluyendo 32 x 4, 32 x 8, 32 x 16 y 32 x 32.

Tras analizar los resultados de estas pruebas, se llegó a la conclusión de que la versión del Simplex Big M con un tamaño de bloque de 32 x 8 ofrecía los mejores tiempos de ejecución. Este tamaño de bloque específico permitió una distribución eficiente de las tareas de cálculo en la GPU, logrando un equilibrio óptimo entre la cantidad de hilos de ejecución y la carga de trabajo.

La elección de un tamaño de bloque adecuado es crucial para aprovechar al máximo las capacidades de paralelismo de la GPU y optimizar el rendimiento de los algoritmos. En este caso particular, se determinó que el tamaño de bloque de 32 x 8 fue la configuración óptima para la versión del Simplex Big M utilizada

en el ejemplo Extra Grande con 1000 trayectorias.

En la tabla 5.4 se presentan los tiempos obtenidos para la versión del Simplex Big M con la configuración de bloques previamente mencionada, considerando diferentes cantidades de trayectorias.

Matriz \Trayectorias	100	500	1000	2000	4000
Base	7,05	9,07	12,16	28,90	55,04
Grande	22,22	106,31	212,71	427,452	857,75
Extra Grande	47,68	231,53	461,30	919,29	1.832,79

Cuadro 5.4: Tiempos de ejecución en milisegundos Big M GPU

Los resultados de que a medida que se incrementa la cantidad de trayectorias, los tiempos de ejecución tienden a aumentar de forma lineal, no es el comportamiento esperado, ya que se esperaba un aumento menos pronunciado de los tiempos de ejecución para las instancias con menor cantidad de trayectorias, es decir, se esperaba que el porcentaje de aumento en el tiempo de ejecución fuera menor que el porcentaje de aumento de la cantidad de trayectorias. Este comportamiento puede deberse a diversas razones, como la limitación de recursos de la GPU (la GPU podría estar alcanzando sus límites de procesamiento a medida que se incrementa la carga de trabajo) o la forma en que se distribuye la carga de trabajo entre los hilos de ejecución, ya que podría no ser eficiente para manejar grandes cantidades de trayectorias.

Con el objetivo de identificar las posibles causas de este comportamiento y buscar mejoras en el algoritmo Simplex Big M, se realizó un análisis más detallado del rendimiento. Este análisis incluyó un desglose por función del algoritmo para determinar en qué parte del proceso se consume más tiempo y detectar posibles cuellos de botella.

A continuación, se presenta una tabla que muestra el tiempo de ejecución en milisegundos dedicado a cada función del algoritmo para el problema Extra Grande con configuración de de bloques 32x8 y para 1000, 2000 y 4000 trayectorias. Se optó por utilizar la matriz de mayor tamaño y cantidades grandes de trayectorias con el fin de evaluar los tiempos de ejecución en condiciones más desafiantes.
1. kernel_resolver_etapa_moverse_a_sol_factible
2. kernel_resolver_etapa_agregar_restricciones_cota_sup
3. kernel_resolver_etapa_agregar_variables_holgura_artificiales
4. kernel_resolver_etapa_resolver_variables_artificiales_en_z
5. kernel_resolver_simplex_big_m

	Trayectorias		
Función	1000	2000	4000
1	0,16	0,27	0,50
2	1,18	1,59	2,37
3	0,47	0,82	1,52
4	2,42	4,81	9,52
5	457,94	912,98	1818,19

Cuadro 5.5: Tiempos de ejecución en milisegundos de las funciones del Big M GPU ejemplo Extra Grande

El análisis de este desglose permite identificar las funciones que consumen más tiempo durante la ejecución del algoritmo. En este caso particular, se puede observar que la función `kernel_resolver_simplex_big_m` es la que requiere más tiempo. Esta función podría ser el cuello de botella en el rendimiento del algoritmo.

Después se analizó la función `kernel_resolver_simplex_big_m` en el algoritmo Simplex Big M. Se observa que la función `intercambiar`, en la cual una variable sale de la base y otra entra a la base, es la que requiere la mayor parte del tiempo de ejecución. Esto indica que la operación de intercambio de variables tiene un impacto significativo en el rendimiento general del algoritmo.

5.4. Evaluación variante 3: Simplex Two Phases

5.4.1. CPU

La version v7_simplex_two_phase_cpu_prog implementa el algoritmo Simplex Two Phase, en un programa C para CPU que contiene ejemplos usados por SimSEE que se pueden correr directamente ejecutando el programa.

Matriz \Trayectorias	100	500	1000	2000	4000
Base	49,26	286,91	618,70	800,99	1.866,75
Grande	133,38	616,41	1.225,74	2.433,84	4.869,86
Extra Grande	1.621,25	7.908,45	15.748,15	31.312,90	62.732,43

Cuadro 5.6: Tiempos de ejecución en milisegundos Two Phases CPU

5.4.2. GPU

La implementación denominada `v8_simplex_two_phase_gpu_prog` es una versión del algoritmo Simplex Two Phase desarrollada en CUDA para GPU. Esta

implementación incluye ejemplos que son compatibles con SimSEE, permitiendo la ejecución directa de los casos de prueba a través del programa.

Matriz \Trayectorias	100	500	1000	2000	4000
Base	13,67	44,42	91,90	176,78	354,56
Grande	41,87	188,05	363,71	714,91	1.427,79
Extra Grande	172,76	744,13	1.472,19	2.993,55	8.323,23

Cuadro 5.7: Tiempos de ejecución en milisegundos Two Phases GPU

El cuadro 5.7 muestra los tiempos de ejecución de la variante Two Pahses. Tras analizar, se observa que la versión del algoritmo Simplex Two Phases no proporcionó el rendimiento esperado, ya que se esperaba que utilizar Two Phases, una para encontrar una solución factible inicial y otra para optimizarla, fuera beneficioso en situaciones en las que es difícil encontrar una solución factible de inmediato, ya que separa el proceso en dos etapas más manejables. Contrariamente a lo anticipado, los tiempos de ejecución no mejoraron, sino que, en comparación con la versión anterior del algoritmo (Simplex Big M), fueron incluso mayores.

Este resultado sugiere que la versión Simplex Two Phases no logró aprovechar eficientemente las características y capacidades del sistema SimSEE en la resolución de problemas de optimización. La falta de mejora en los tiempos de ejecución puede atribuirse a diversos factores, como la complejidad adicional introducida por el proceso de dos fases o la interacción entre las etapas del algoritmo. La eficacia del Simplex Two Phases a veces depende de la elección de la base inicial. Entonces, si la elección de esta base no es adecuada, el algoritmo puede tener dificultades para encontrar la solución factible y por lo tanto no lograr la mejora de los tiempos de ejecución. Otra posibilidad es que las características del problema no sean ideales para aprovechar la estrategia de las dos fases.

5.5. Evaluación de las tres variantes juntas

En las siguientes gráficas se puede observar los tiempos de ejecución para las configuraciones de SimSEE, Big M y Two Phases para el caso Base, Grande y Extra Grande en función de la cantidad de trayectorias.

Cada configuración se representa mediante una línea en la gráfica, y los puntos en la línea representan los tiempos de ejecución para cada cantidad de trayectorias.

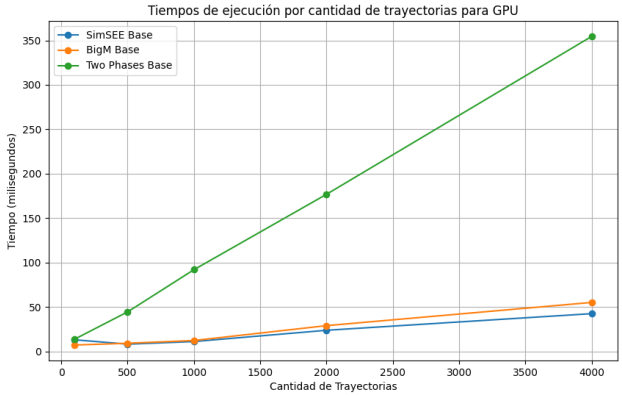


Figura 5.1: Tiempos de ejecución para Matriz Base GPU

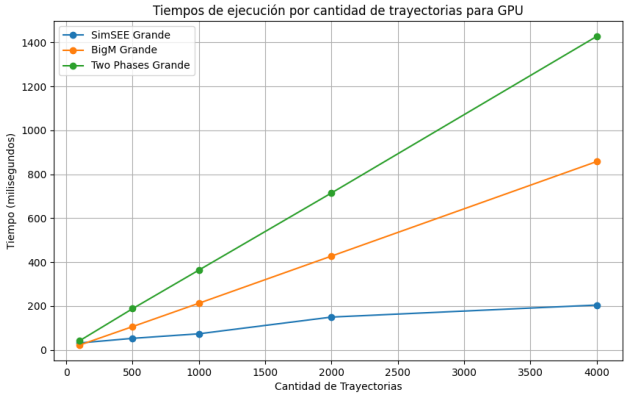


Figura 5.2: Tiempos de ejecución para Matriz Grande GPU

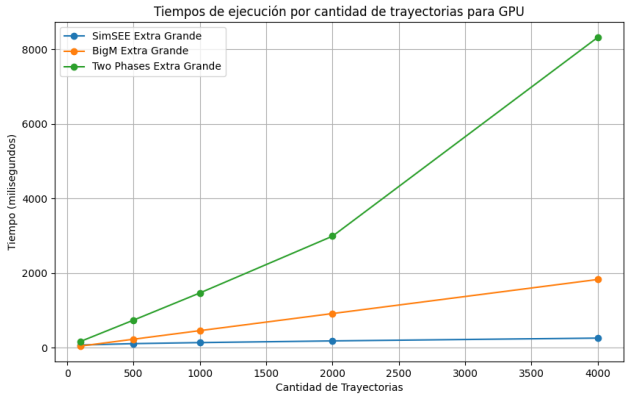


Figura 5.3: Tiempos de ejecución par Matriz Extra Grande

En la primera gráfica, que representa los tiempos de ejecución obtenidos para la matriz Base, se observa que los tiempos de ejecución del Big M y SimSEE presentan un comportamiento similar. A medida que aumentan las trayectorias, los tiempos de ejecución del Big M se alejan gradualmente . Además, se observa que la solución del Two Phases muestra un crecimiento lineal, lo que indica que es la menos performante de las tres soluciones. Otra observación importante es que, a medida que aumenta el tamaño de la matriz (caso de la matriz Grande y ExtraGrande), el rendimiento del Big M empeora. Esto se puede ver analizando las pendientes de las gráficas y calculando la diferencia entre los tiempos de ejecución y las trayectorias. Las pendientes están representando la velocidad de cambio en los valores a medida que aumenta la cantidad de trayectorias.

Entonces si analizamos el caso del algoritmo Big M, tenemos que para la matriz Base, la pendiente es $\frac{50}{4000} = 0,0125$. Esto significa que por cada aumento de 1 en las trayectorias (un aumento del 1 %), el valor del algoritmo Big M en la matriz Base aumenta en $0,0125$ms, lo que indica un cambio relativamente pequeño en los valores a medida que aumentan las trayectorias. Luego en el caso de la matriz Grande, la pendiente es $\frac{800}{4000} = 0,2$. Esto significa que por cada aumento de 1 en las trayectorias, el valor del algoritmo Big M en la matriz Grande aumenta en un $0,2$ms. Para el caso que se usa de la matriz Extra Grande, la pendiente es $\frac{2000}{4000} = 0,5$, por lo que se puede comprobar que el valor del algoritmo Big M en la matriz Extra Grande aumenta en $0,5$ms para cada trayectoria. Esta es la pendiente más pronunciada de las tres configuraciones (Base, Grande y Extra Grande), lo que indica un cambio más grande en los valores a medida que aumentan las trayectorias en la configuración Extra Grande. Por eso se llega a la conclusión que el rendimiento del Big M empeora a medida que aumenta el tamaño de la matriz.

Simplex SimSEE aborda las restricciones de cota superior mediante lógica

adicional, en contraposición a la estrategia de incorporar estas restricciones en la matriz del problema, como lo hace Simplex Big M.

Para ilustrar este punto, consideremos un caso con 50 variables y 15 restricciones. Si solo la mitad de las variables, que es una situación común según los ejemplos disponibles, tienen cotas superiores, el tamaño de la matriz de datos se incrementaría a 75 variables y 40 restricciones, es decir, cuatro veces más grande.

5.6. Variante Big M con matriz en memoria compartida

Luego de observar que los tiempos de ejecución del algoritmo Big M no son mejores que los de SimSEE, se decidió explorar una nueva variante en el algoritmo Simplex Big M. Esta variante consiste en almacenar toda la matriz en memoria compartida. Sin embargo, nos encontramos con la limitación de que la matriz más grande que puede ser almacenada completamente en la memoria compartida es de 36 variables y 13 restricciones.

Una vez incorporadas las variables adicionales, las restricciones de límite y las variables necesarias para los cálculos del algoritmo, el tabloide resultante tiene un tamaño de 96 columnas y 55 filas, con un total de 92 variables y 49 restricciones. Además, esta implementación utiliza las siguientes estructuras en memoria compartida:

- Un arreglo de números en punto flotante (double) para acumuladores con un tamaño igual al número de variables y otro para número de restricciones.
- Una matriz de acumuladores con un tamaño de 8 filas por 32 columnas, también en punto flotante (double).
- Dos arreglos de números enteros, uno con un tamaño igual a la cantidad de de variables y otro con un tamaño igual a la cantidad de de restricciones, además de dos variables adicionales.

En resumen, el cálculo del espacio de memoria requerido es el siguiente:
Total: $8 \times (96 \times 55 + 92 + 8 \times 32 + 49) + 4 \times (92 + 55 + 2) = 46012$ bytes.

Esta limitación implica que si deseamos trabajar con matrices de mayor tamaño, deberemos buscar otras estrategias de optimización y aprovechar otras características de la GPU para mejorar el rendimiento del algoritmo Big M.

También esta la posibilidad de reducir el tamaño de las matrices que se almacenan en la memoria compartida utilizando diversas estrategias. La reducción de la cantidad de datos en la memoria compartida puede ser necesaria cuando las limitaciones de tamaño de la memoria compartida de la GPU son un obstáculo para el rendimiento de un algoritmo. Una de las estrategias que se podrían usar es la eliminación de redundancia. Por ejemplo, si una restricción es una combinación lineal de otras restricciones, esta se podría eliminar. También se podría reducir la precisión (dependiendo del problema a resolver). Por ejemplo, si los valores en la matriz son números de punto flotante de doble precisión, se

podría considerar usar precisión simple (float) para reducir el uso de memoria. Otra estrategia podría ser la de particionar la matriz. En lugar de almacenar toda la matriz en la memoria compartida, se pude dividir la matriz en bloques más pequeños y cargar solo los bloques necesarios en la memoria compartida en cada iteración del algoritmo.

La exploración de esta nueva variante y la búsqueda de estrategias de optimización adicionales serán parte de los trabajos futuros para mejorar el desempeño y la eficiencia del algoritmo Big M en el contexto del sistema SimSEE.

En respuesta a la necesidad de comparar el rendimiento del algoritmo simplex Big M con el algoritmo Simplex SimSEE en un escenario diferente, se creó una nueva matriz de prueba llamada matriz XS (36 variables y 13 restricciones). Se ejecutaron los tiempos de ejecución tanto para el algoritmo Big M como para SimSEE, con el fin de comparar los resultados y determinar si en este caso particular el rendimiento del algoritmo Big M podría ser superior al del algoritmo Simplex SimSEE.

Matriz \Trayectorias	100	500	1000	2000	4000
XS SimSEE	6,39	9,07	12,15	19,49	33,15
XS Big M	1,74	5,56	10,29	20,01	39,39

Cuadro 5.8: Tiempos de ejecución en milisegundos SimSEE y Big M GPU

Después de analizar los datos, se observa que el algoritmo Big M muestra una mejora significativa en los tiempos de ejecución a menor tamaño de la matriz. Sin embargo, a medida que aumenta la cantidad de trayectorias, este es superado por SimSEE XS, que tiene crecimiento sublineal.

Este comportamiento sugiere que el algoritmo Big M puede ser más eficiente en matrices más pequeñas, pero puede encontrar dificultades para escalar y manejar un mayor volumen de datos. Esto podría estar relacionado con la complejidad y la cantidad de operaciones requeridas por el algoritmo en función del tamaño de la matriz y la cantidad de trayectorias.

Estos hallazgos resaltan la importancia de evaluar cuidadosamente el rendimiento de los algoritmos en diferentes escenarios y considerar estrategias de optimización y paralelización para mejorar el rendimiento en situaciones de mayor demanda computacional, además de permitir seleccionar el algoritmo más rápido según el número de trayectorias.

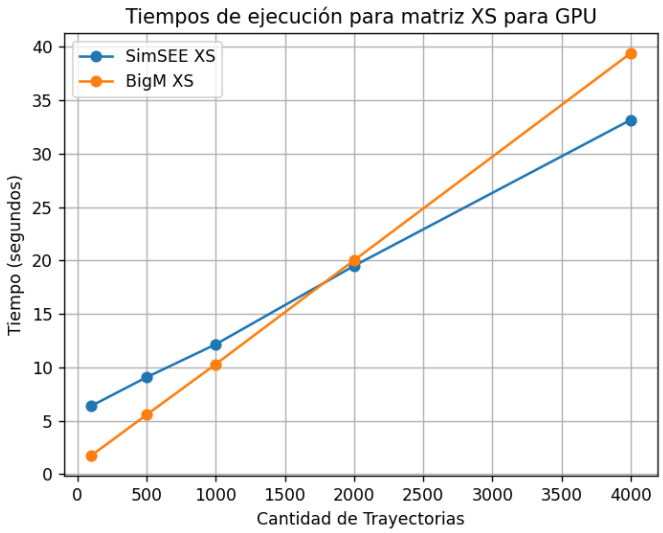


Figura 5.4: Tiempos de ejecución para Matriz XS GPU

En futuros trabajos, será importante explorar en profundidad las causas del deterioro en el rendimiento del algoritmo Big M a medida que aumenta la cantidad de trayectorias y buscar posibles soluciones y mejoras para abordar este desafío.

En la Tabla 5.9, se toman los tiempos de las funciones que se encargan del manejo de memoria entre CPU y GPU. La función `ini_mem` prepara los datos del tabloide para ser utilizados en la GPU, reservando y copiando memoria según las necesidades para una ejecución paralela en la GPU utilizando la librería CUDA, mientras que la función `copy_mem_back`, se encarga de copiar los datos de vuelta desde la GPU a la memoria RAM de la CPU.

Matriz	Trayectorias	ini_mem	copy_mem_back
Extra Grande SimSEE	2000	706,402	92,214
Extra Grande Big M	2000	860,265	246,525

Cuadro 5.9: Tiempos de ejecución en milisegundos de SimSEE y Big M GPU por cada paso de tiempo

Haciendo una comparación de estos tiempos con los tiempos de resolución del algoritmo, observamos que los tiempos de transferencia de memoria son considerables. Por ejemplo, para el método Big M, el tiempo de resolución de 2000

trayectorias es de 912ms, pero se emplea 860ms para inicializar la memoria y transferir los datos a la GPU, seguido de otros 246ms en copiar la memoria devuelta a la RAM de la CPU. En el caso ideal, la simulación se realizaría enteramente en la GPU, y solo sería necesario transferir los datos entre CPU y GPU antes del primer paso de tiempo y luego del último.

También observamos que el manejo de memoria que utiliza Big M es bastante más eficiente, ya que la memoria que utiliza el método Big M es del orden de cuatro veces mayor, y el tiempo de inicio y copia es casi el mismo. Esto se debe a que la estructura de datos de Simplex SimSEE está mucho más fragmentada y no resulta tan conveniente para el procesamiento en GPU.

Todos los tiempos de resolución de los algoritmos son comparados sin tener en cuenta los tiempos de manejo de memoria entre CPU y GPU. Aunque estos tiempos son considerables, se pueden reducir con técnicas de streams que provee el lenguaje CUDA y las GPU Nvidia, pudiendo paralelamente ir copiando los datos a GPU en streams mientras ésta realiza cálculos con los streams ya procesados, reduciendo significativamente estos tiempos de manejo de memoria, referencias (Steve Rennich, s.f.) (*NVIDIA, CUDA C++ Programming Guide 12.2*, 2023). En la imagen 5.5 se puede observar un esquema con varias técnicas para paralelizar la transferencia de memoria con los cálculos realizados en el *kernel*. En SimSEE se podría dedicar un stream a la ejecución de cálculos en la GPU y otro a las operaciones de transferencia de datos entre la CPU y la GPU. Aplicar esta técnica de manejo de memoria y streams en CUDA, se puede reducir significativamente los tiempos de transferencia de datos entre la CPU y la GPU, lo que permitirá realizar simulaciones de SimSEE de manera más eficiente en un entorno GPU.

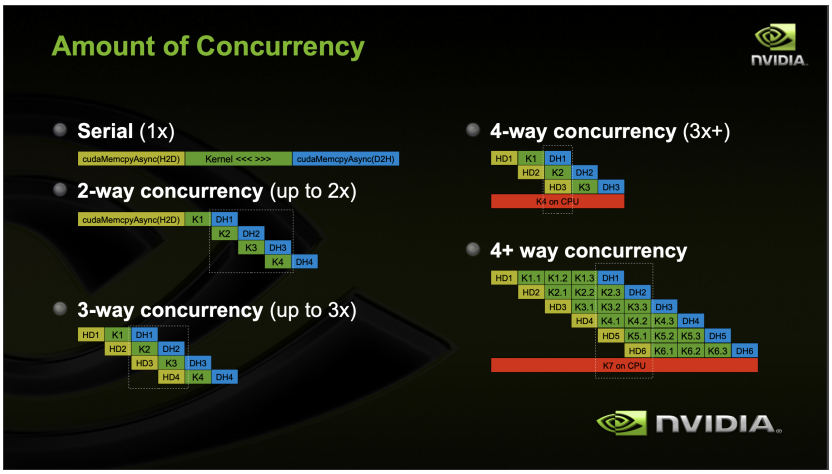


Figura 5.5: Esta imagen presenta un esquema con diferentes técnicas para paralelizar las transferencias de memoria entre la CPU y la GPU simultáneamente con la ejecución de los *kernels*. Fuente (Steve Rennich, s.f.).

No incluir estos tiempos de manejo en memoria entre CPU y GPU, permite analizar el comportamiento en un escenario donde los datos residen en GPU y las transferencias no son necesarias.

Luego la función `free_mem` se encarga de liberar la memoria asignada previamente en la GPU para los datos del tabloide que fueron utilizados durante la ejecución del programa. Esto se puede realizar de forma asincrónica con la función `cudaFreeAsync` por lo que estos tiempos no impactan en el tiempo final de la rutina.

Capítulo 6

Conclusiones y Trabajo Futuro

Este proyecto tuvo como objetivo principal analizar la posibilidad de mejorar el rendimiento de la herramienta SimSEE mediante la implementación de GPUs. Para lograr este cometido, se planearon una serie de objetivos específicos, tales como el estudio exhaustivo de SimSEE, la evaluación de su rendimiento en diferentes escenarios, y la identificación de los procesos computacionales más costosos. Además, se llevó a cabo una investigación del estado del arte relacionado con el problema a resolver, desarrollando algoritmos y técnicas de programación específicas para aprovechar la capacidad de procesamiento paralelo de las GPUs.

Tras el análisis, se pudo concluir que la tarea óptima para llevar a cabo en la GPU es el cálculo del método Simplex. Esto se debe a la alta cantidad de ejecuciones requeridas en una simulación (del orden de millones), la carga computacional asociada y la idoneidad de la GPU para resolver este tipo de problema.

Luego de llevar a cabo un análisis teórico exhaustivo del Método Simplex, sus diversas variantes, la implementación específica de SimSEE y los trabajos relacionados, se pudo identificar varias vías para abordar el problema. Sin embargo, no se encontró literatura que abordara de manera específica el desafío de lograr un alto rendimiento en la ejecución paralela de miles de algoritmos Simplex de tamaño pequeño (del orden de 200 variables x 100 restricciones) con restricciones mixtas, incluyendo restricciones de cota superior. Dentro de un alcance razonable en términos de complejidad y extensión, se logró implementar y comparar el rendimiento de diversas versiones.

Se desarrollaron tres versiones del método Simplex:

- **Simplex SimSEE:** Es una adaptación a GPU del algoritmo actualmente utilizado por SimSEE, que corre en CPU. Esta versión tiene la capacidad

de manejar las restricciones de cota superior.

- **Simplex Big M y Simplex Two Phases:** Ambos manejan las restricciones de cota superior como restricciones ordinarias. En una primera etapa, se implementó una versión para CPU y posteriormente su versión paralela para GPU.

Luego de realizar un análisis exhaustivo de los tiempos obtenidos en las diversas versiones del algoritmo, utilizando los problemas de ejemplo disponibles y ejecutando diferentes cantidades de trayectorias, se pudo constatar que todas las versiones implementadas en GPU superaron ampliamente en rendimiento a las versiones en CPU, siendo Simplex SimSEE la variante que ofrece un mejor rendimiento.

Por otro lado las versiones Simplex Big M y Simplex Two Phases son susceptibles de mejoras, como se describe en los artículos mencionados en la Sección 2.5, aplicando técnicas asociadas a la versión Revisada del Algoritmo Simplex (RSA).

También se considera que Simplex SimSEE tiene margen de mejora, ya que contiene segmentos que no son óptimos para ejecución en una GPU debido a su alta cantidad de bifurcaciones, dependencia de datos y necesidad de sincronización, estos segmentos se podrían reescribir considerando la arquitectura de la GPU para luego ser paralelizados. Además, no utiliza una estructura de datos compacta y matricial, que sería más adecuada para una GPU, principalmente por la transferencia de datos entre la RAM y la memoria global de la GPU.

Uno de los objetivos detrás de mejorar el rendimiento de SimSEE es poder llevar a cabo simulaciones más complejas en un plazo razonable. El crecimiento del número de actores en el sistema eléctrico, resultado del cambio de la matriz energética uruguaya, incrementa el número de actores, fuentes de energía, variables y restricciones, requiriendo así más tiempo de procesamiento.

Es por esto que queremos hacer un foco en el análisis de los resultados para el caso de estudio denominado “Extra Grande”, creado artificialmente a partir del caso real de mayor magnitud disponible, incrementando su tamaño en casi el doble. Para este caso, observamos los resultados para un número razonable de trayectorias, concretamente 2000 (siendo 1000 la configuración estándar en una ejecución de SimSEE).

En este contexto, el algoritmo Simplex SimSEE adaptado para GPU supera en un 96 % a su versión en CPU, lo que se traduce en un tiempo de ejecución 24 veces menor (4.488,95 ms frente a 185,95 ms). De manera similar, la versión Simplex Big M en GPU supera a su contraparte en CPU en un 98 %, reduciendo el tiempo de ejecución en 50 veces (aunque la ventaja sobre la mejor version en CPU de SimSEE es de unas 4 veces).

Estos resultados enfatizan claramente la capacidad del algoritmo Simplex Big M para ejecutarse en paralelo en una GPU. Nuestra convicción es que el rendimiento superior de Simplex SimSEE se debe, en gran parte, al tamaño de los datos que maneja, el cual es considerablemente menor.

Basados en este análisis, planteamos que una versión óptima del algoritmo se encuentra en un punto medio entre las dos variantes. Sería beneficioso incorporar más lógica para eliminar las restricciones de cota superior de la matriz del problema en el algoritmo "Simplex Big M", sin incorporar el resto de las mejoras de rendimiento utilizadas en "Simplex SimSEE", diseñadas específicamente para un óptimo funcionamiento en una CPU. Además de utilizar los datos del problema integrados en un tabloide como lo hace "Simplex Big M", que por lo visto en el capítulo anterior, tiene mejores tiempos de transferencia de datos entre la GPU y la CPU.

La técnica propuesta para lograr esta mejora en el algoritmo "Simplex Big M" se denomina "Técnica de la cota superior"(Hillier F.S., 1991).

Otro hilo de investigación puede ser integrar en el algoritmo Simplex Big M el método RSA adaptado a GPUs de acuerdo a los artículos presentados en la Sección 2.5.

También se resalta la importancia de adaptar el enfoque del algoritmo según las características del problema y los recursos disponibles. El método del Simplex SimSEE en GPU es altamente efectivo para problemas de mayor tamaño y mayor demanda computacional, mientras que la solución del Simplex Big M con matriz en memoria compartida es más adecuada para problemas más pequeños y con requisitos de memoria específicos.

En futuras iteraciones del sistema SimSEE, se desarrollará una estrategia adaptativa que utilice el método del Simplex SimSEE en GPU como la opción predeterminada, pero considerando la opción del Simplex Big M con matriz en memoria compartida para problemas más pequeños y con restricciones de recursos.

Como trabajo futuro, se plantean diferentes líneas de investigación y desarrollo para mejorar el sistema SimSEE y su rendimiento en la resolución de problemas de optimización.
Algunas líneas de investigación y desarrollo son:

1. Lograr una versión de "Simplex Big M" integrando la "Técnica de la cota superior".

2. Lograr una versión de "Simplex Big M" integrando el método RSA adaptado a GPUs.

3. Investigación de nuevas técnicas y enfoques: Explorar nuevas técnicas de optimización y enfoques de resolución que puedan ser aplicados en el contexto del sistema SimSEE. Investigar algoritmos avanzados, métodos heurísticos o técnicas de aprendizaje automático que puedan mejorar la eficiencia y precisión de este problema.

4. Evaluación de otros problemas y conjuntos de datos: Extender el análisis y las pruebas del sistema SimSEE a otros problemas y conjuntos de datos, para evaluar su rendimiento en diferentes escenarios. Esto permitirá comprender cómo se comporta el sistema en situaciones variadas y determinar áreas de mejora específicas para cada tipo de problema.

Estas líneas de investigación y desarrollo proporcionan una base sólida para futuros trabajos en el sistema SimSEE, con el objetivo de mejorar su rendimiento, eficiencia y capacidad de resolución de problemas de optimización en diversos contextos.

Referencias

AI Planet. (2023). *Maldición de la dimensión.* https://aiplanet.com/learn/unsupervised-learning-es/introduccion-a-la-reduccion-de-la-dimensionalidad-y-sus-tecnicas/1679/reduccion-de-la-dimensionalidad.

Alfredo G. Escobar Portillo, J. M. L. C. (enero-febrero 2011). *Cálculo en paralelo empleando tarjetas gráficas. aplicación al algoritmo símplex revisado.* https://repositorio.comillas.edu/xmlui/bitstream/handle/11531/5124/IIT-11-076A.pdf?sequence=1&isAllowed=y.

Bellman, R. (1957). *Dynamic programming,*. Princeton University Press,.

CFN, Corporación Financiera Nacional. (2017). *Generación, transmisión y distribución de energía eléctrica.* https://www.cfn.fin.ec/wp-content/uploads/2018/04/Ficha-Sectorial-Generacion-transmision-y-distribucion-de-energia-electrica.pdf. (Accedido en enero de 2023)

Chaer., I. R. (2008). *Simulación de sistemas de energía eléctrica.* https://simsee.org/simsee/tesischaer.pdf. (Tesis para optar al grado de Magister en Ingeniería Eléctrica)

Dr. Hiba G. Fareed. (2022). *The two-phase simplex method.* https://uomustansiriyah.edu.iq/media/lectures/6/6_2022_01_08!08_05_56_PM.pdf. (Accedido en julio de 2022)

Free Pascal. (2020). *Free pascal.* https://www.freepascal.org/. (Accedido en noviembre de 2020)

He, L., Bai, H., Jiang, Y., y Ouyang, D. (2018). Revised simplex algorithm for linear programming on gpus with cuda. *Multimedia Tools and Applications*, *77*, 30035–30050.

Heinz, R. (1990). *Lectures on numerical mathematics.* Birkhäuser Boston, MA. (Linear inequalities (optimization))

Hillier F.S., L. G. (1991). *Introdución a la investigacion de operaciones.* https://frh.cvg.utn.edu.ar/pluginfile.php/54151/mod_resource/content/1/Introducci%C3%B3n%20a%20la%20Investigaci%C3%B3n%20de%20Operaciones%20%289na%20ed%29%20-%20Hillier%20%20Lieberman.pdf. (Programacion lineal)

H.S. Kasana, K. K. (2004). *Introductory operations research.* (Chapter 3 The Simplex Algorithm)

Ings. Felipe Palacio, P. S. y. R. C. (2019). *"manuales de usuario de simsee volumen 1.* https://simsee.org/simsee/verdoc/vol1.php. (Editor y Simulador)

Joaquín Amat Rodrigo. (Junio, 2017). *Análisis de componentes principales.* https://cienciadedatos.net/documentos/35_principal_component_analysis. (Accedido en diciembre de 2020)

Lazarus. (2020). *Lazarus.* https://www.lazarus-ide.org/. (Accedido en noviembre de 2020)

Marichal, R., y cols. (2021). *Towards a massively-parallel version of the simsee.* https://www.colibri.udelar.edu.uy/jspui/bitstream/20.500.12008/36664/1/MVDE21.pdf. (Accedido en mayo de 2022, Marichal, Raúl, Vallejo, Damián, Dufrechou, Ernesto, Ezzatti, Pablo)

Marichal, R., Seveso, F., Dufrechou, E., y Ezzatti, P. (2022). *Refactoring an electric-market simulation software for massively parallel computations.* https://www.colibri.udelar.edu.uy/jspui/bitstream/20.500.12008/36663/1/SMDE22.pdf. (Accedido en enero de 2023)

Mark Harris. (2012a). *Cuda: How to optimize data transfers.* https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc. (Accedido en julio de 2022)

Mark Harris. (2012b). *Cuda: Optimizing parallel reduction.* https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf. (Accedido en julio de 2022)

Mohamed Esseghir Lalami, D. E.-B., Vincent Boyer. (2011). *Efficient implementation of the simplex method on a cpu-gpu system.* https://www.researchgate.net/publication/220951550_Efficient_Implementation_of_the_Simplex_Method_on_a_CPU-GPU_System. (IEEE International Parallel Distributed Processing Symposium)

Nvidia, cuda c++ programming guide 12.2. (2023). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

Pablo Alfaro. (2009). Manual de usuario de la clase tsimplex [Manual de software informático]. https://simsee.org/simsee/por_dentro/SimSEE_pordentro004_TSimplex.pdf. Montevideo - Uruguay.

Portal UTE. (2023). *Portal ute.* https://portal.ute.com.uy/. (Accedido en enero de 2023)

SimSEE. (2020). *Simsee.* https://simsee.org/. (Accedido en noviembre de 2020)

Steve Rennich. (s.f.). *Cuda: Streams and concurrency.* https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf. (Accedido en enero de 2022)

Uruguay XXI. (Octubre 2022). *Energías renovables en uruguay.* https://www.uruguayxxi.gub.uy/uploads/informacion/95cfe0ad951a9828f3d434931a8fd0de2740b64c.pdf. (Accedido en enero de 2023)

Usman Ali Shah, I. A.-S. U. R., Suhail Yousaf, y Ahmad, M. O. (2020). *Accelerating revised simplex method using gpu-based basis update.*

https://www.researchgate.net/publication/220951550_Efficient_Implementation_of_the_Simplex_Method_on_a_CPU-GPU_System.