UNIVERSIDAD DE CUENCA

Facultad de Ingeniería

Carrera de Electrónica y Telecomunicaciones

Análisis de factibilidad del uso de unidades de procesamiento gráfico en algoritmos de optimización meta-heurísticos

Trabajo de titulación previo a la obtención del título de Ingeniero en Electrónica y Telecomunicaciones. Artículo científico.

Autor:

Manuel Mesias Guiracocha Yuquilima
C.I: 010479480-5
manuel.guiracocha94@gmail.com

Director:

Ing. Darwin Fabián Astudillo Salinas, PhD
C.I:010390703-6

Codirector:

Ing. Santiago Patricio Torres Contreras, PhD
C.I: 010244895-8

Cuenca - Ecuador
14 de octubre de 2020

Resumen

Actualmente, varios problemas de optimización del mundo real se han modelado matemáticamente. El proceso de modelado toma en cuenta la mayor cantidad de información posible para que el modelo obtenido sea lo más próximo a la realidad. Sin embargo, a medida que aumenta la información, la complejidad también aumenta. En consecuencia, se necesita una mayor capacidad computacional para resolver problemas complejos y escalables. Los métodos de optimización exactos o matemáticos son considerados de búsqueda exhaustiva, lo que para problemas multi-dimensionales no resulta práctico. Como resultado, se han desarrollado algoritmos meta-heurísticos para resolver problemas complejos de optimización. Estos algoritmos se usan comúnmente para problemas de dos o más dimensiones, en los que intervienen operaciones de vectores y matrices. Por lo tanto, para resolver este tipo de problema, es útil llevar a cabo procesos paralelos que reducen el tiempo de ejecución. Actualmente, existen unidades de procesamiento central (*Central Processing Unit* (CPU)) de múltiples núcleos que logran resolver fácilmente pequeños problemas con cálculos paralelos. Sin embargo, la unidad de procesamiento de gráficos (*Graphics Processing Unit* (GPU)) ofrece la posibilidad de mejorar el rendimiento porque incorpora una mayor cantidad de núcleos que la CPU, lo cual es muy útil para problemas con varios procesos en paralelo. Un problema clásico en la comunidad de investigación de sistemas de energía eléctrica es el Planeamiento de la Expansión del Sistema Eléctrico de Transmisión (PET). En el departamento de Ingeniería Eléctrica, Electrónica y Telecomunicaciones, en el proyecto "Uso del modelo de corriente alterna para la planificación integrada multietapa de la expansión de líneas de transmisión/subtransmisión y fuentes de potencia reactiva en sistemas de energía eléctrica" se trata de mejorar el desempeño del problema PET. En el proyecto, se busca reducir el tiempo de cálculo mejorando los algoritmos meta-heurísticos. En este trabajo de titulación, como complemento al proyecto mencionado precedentemente, se analizó la factibilidad de reducir el tiempo de ejecución, implementando los algoritmos meta-heurísticos usando una GPU. Para lo cual, previamente a la implementación del problema PET, se analizó el uso de la GPU en problemas paralelizables y la implementación de un algoritmo meta-heurístico en un problema bidimensional. Lo cual ha sido favorable para motivar el uso de la GPU.

Palabras clave : CUDA. GPU. Meta-Heurística. Modelo AC. Optimización. Planificación Expansión de Transmisión.

Abstract

Currently, several real-world optimization problems have been mathematically modeled. The modeling process takes into account as much information as possible so that the model obtained is as close to reality. However, as information increases, complexity also increases. Consequently, larger computational capacity is needed to solve complex and scalable problems. Exact or mathematical optimization methods are considered exhaustive, which for multi-dimensional problems is not practical. As a result, meta-heuristic algorithms have been developed to solve complex optimization problems. These algorithms are commonly used for problems of two or more dimensions, involving vector and matrix operations. Therefore, to solve this type of problem, it is useful to carry out parallel processes that reduce the runtime. Currently, there are multi-core CPU that can easily solve small problems with parallel calculations. However, the GPU offers the potential to improve performance by incorporating a larger number of cores than the CPU, which is very useful for problems with multiple processes in parallel. A classic problem in the electrical power systems research community is the Transmission Expansion Planning (TEP). In the Department of Electrical, Electronics, and Telecommunications Engineering, in the project "Use of the AC model for multi-stage integrated planning of transmission / sub-transmission line expansion and reactive power sources in electric power systems", It is tried to improve the performance of the TEP problem. The project seeks to reduce the calculation time by improving the meta-heuristic algorithms. In this final degree project, as a complement to the aforementioned project, the feasibility of reducing the execution time was analyzed, implementing the meta-heuristic algorithms using a GPU. For which, prior to the implementation of the TEP problem, the use of the GPU in parallelizable problems and the implementation of a meta-heuristic algorithm in a two-dimensional problem were analyzed, in which favorable results have been obtained to motivate the use of the GPU.

Keywords: AC Model. CUDA. GPU. Meta-heuristic. Optimization. Particle Swarm Optimization. Transmission Expansion Planning.

Índice general

Índice de figuras

Índice de tablas

Cláusula de Propiedad Intelectual

Yo, Manuel Mesias Guiracocha Yuquilima, autor del trabajo de titulación "Análisis de factibilidad del uso de unidades de procesamiento gráfico en algoritmos de optimización meta-heurísticos", certifico que todas las ideas, opiniones y contenidos expuestos en la presente investigación son de exclusiva responsabilidad de su autor.

Cuenca, 14 de octubre de 2020

Manuel Mesias Guiracocha Yuquilima
0104794805

Cláusula de licencia y autorización para publicación en el Repositorio Institucional

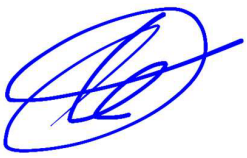
Yo, Manuel Mesias Guiracocha Yuquilima en calidad de autor y titular de los derechos morales y patrimoniales del trabajo de titulación "Análisis de factibilidad del uso de unidades de procesamiento gráfico en algoritmos de optimización meta-heurísticos", de conformidad con el Art. 114 del CÓDIGO ORGÁNICO DE LA ECONOMÍA SOCIAL DE LOS CONOCIMIENTOS, CREATIVIDAD E INNOVACIÓN reconozco a favor de la Universidad de Cuenca una licencia gratuita, intransferible y no exclusiva para el uso no comercial de la obra, con fines estrictamente académicos. Asimismo, autorizo a la Universidad de Cuenca para que realice la publicación de este trabajo de titulación en el repositorio institucional, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación Superior.

Cuenca, 14 de octubre de 2020

Manuel Mesias Guiracocha Yuquilima
0104794180-5

Certifico

Que el presente proyecto de tesis: Análisis de factibilidad del uso de unidades de procesamiento gráfico en algoritmos de optimización meta-heurísticos, fue dirigido y revisado por mi persona.



Ing. Darwin Fabián Astudillo Salinas, PhD
Director

Certifico

Que el presente proyecto de tesis: Análisis de factibilidad del uso de unidades de procesamiento gráfico en algoritmos de optimización meta-heurísticos, fue dirigido y revisado por mi persona.

Ing. Santiago Patricio Torres Contreras, PhD
Co-director

Dedicatoria

Dedico mi proyecto de titulación a todas aquellas personas que fueron un pilar fundamental en el desarrollo de mi carrera profesional. En primer lugar a mi familia, que siempre estuvo ahí para apoyarme y guiarme hasta cumplir mis metas. Ellos siempre supieron darme la fuerza necesaria para continuar. También dedico este proyecto a mis amigos, que día a día estuvieron a mi lado, volviendo a la vida universitaria, un recuerdo que no se podrá olvidar.

Manuel Guiracocha

Agradecimientos

Agradezco inmensamente a mis padres por permitirme estudiar y seguir el camino que siempre he querido. Me han enseñado que las decisiones que uno toma pueden trascender en la vida. Agradezco también a mis hermanos. Ellos me han motivado desde pequeño a auto-educarme, lo cual considero un pilar fundamental en mi vida y que ha forjado lo que ahora soy. Agradezco a mi compañera de vida y a mi hijo. Quienes me han demostrado que para ser feliz no hace falta tener mucho, sino tener muchas ganas de ser feliz. También agradezco a mis abuelos, quienes me han enseñado que una persona hábil no es aquella que tiene la capacidad de obtener buenas calificaciones, sino la persona que es capaz de aplicar el conocimiento y sacarle provecho a sus talentos. Por último, agradezco a mis tutores, quienes me han guiado a través del proyecto y han aportado a mi formación profesional.

Manuel Guiracocha

Abreviaciones y Acrónimos

AC Corriente alterna. 1, 7–11, 21
ACO *Ant Colony Optimization*. 5, 9
API *Application Programming Interface*. 7, 9

CPU *Central Processing Unit*. 1–3, 6, 7, 9, 10, 13, 15, 18, 20, 23
CUDA *Compute Unified Device Architecture*. 7, 9–11, 14, 15, 23

DC Corriente continua. 1, 7–9

GPU *Graphics Processing Unit*. 1–3, 6, 7, 9–11, 13–16, 18, 20, 23, 25, 31

HPC *High-Performance Computing*. 1, 2

JIT *Just-in-time*. 10

PET Planeamiento de la Expansión del Sistema Eléctrico de Transmisión. 1–3, 7, 9–14, 16, 20, 23, 25
PNLEM Programación No Lineal Entera Mixta. 1, 3, 8
PSO *Particle Swarm Optimization*. 2, 5, 6, 9–13, 16, 18, 20, 23

Capítulo 1

Introducción

Este capítulo presenta la identificación del problema (sección 1.1), justificación (sección 1.2) y los objetivos (sección 1.4) del proyecto.

1.1. Identificación del problema

Los problemas de optimización con alta dificultad requieren herramientas potentes para analizar y procesar una cantidad importante de datos. En problemas con cálculos repetitivos e independientes, los científicos e ingenieros tienen dificultades. Por lo tanto, es necesario paralelizar los procesos de cálculo. Para este propósito, se pueden usar la CPU y la GPU. Una CPU tiene múltiples núcleos y permite que los procesos se lleven a cabo en paralelo. Sin embargo, una GPU tiene una mayor cantidad de núcleos y a menudo se usa en problemas de *High-Performance Computing* (HPC).

En la actualidad, existe sistemas complejos en los que se evalúa su comportamiento ante factores externos. Estos factores son capaces de alterar el rendimiento o fin para el que está siendo utilizado el sistema. Por lo tanto, es necesario plantear un modelo robusto que ayude a encontrar soluciones para problemas futuros [1].

En el campo de la ingeniería es común encontrar sistemas modelados matemáticamente. La mayoría de estos sistemas tienen una alta complejidad computacional, lo que representa un reto para las herramientas actuales de hardware. Uno de ellos es el conocido problema del PET. Este problema representa la estructura eléctrica necesaria que debe adicionarse al sistema de transmisión para que pueda satisfacer el incremento de la demanda. Este incremento es debido a la expansión poblacional. Por lo cual, su planificación se debe realizar tomando en cuenta un horizonte de tiempo. El problema del PET tiene dos modelos importantes: el modelo AC y el modelo DC. El modelo AC considera parámetros que el modelo DC no considera, haciéndolo más complejo y preciso [2].

Sin embargo, el problema computacional va más allá del modelo a usarse. El problema del PET se considera un problema complejo de Programación No Lineal Entera Mixta (PNLEM) que puede contener múltiples óptimos locales. Por esto, es fundamental resolverlo con técnicas de optimización que ayuden a simplificar el proceso de búsqueda [3, 4].

En la actualidad, los algoritmos meta-heurísticos han ayudado de gran manera a optimizar problemas similares al PET. En los cuales se necesita una búsqueda rápida de la solución óptima entre un gran número de posibles soluciones. Los algoritmos meta-heurísticos están basados en comportamientos naturales como la búsqueda de alimento por parte de grupos de abejas u hormigas. Estos algoritmos son relativamente sencillos y logran determinar una solución óptima en poco tiempo de ejecución [5, 6] .

Las meta-heurísticas parten de la creación de un banco o cúmulo de posibles soluciones. Sin embargo, mientras más soluciones candidatas hayan, más tardará la meta-heurística en encontrar una solución óptima. El tiempo que tarda la meta-heurística en encontrar la solución, también está relacionado a las herramientas de hardware y de software que se estén utilizando para ejecutar el algoritmo. Es común el uso de la CPU para ejecutar algoritmos en un computador. Sin embargo, en los últimos años es frecuente el uso de una GPU para ejecutar algoritmos que contengan procesos repetitivos y paralelizables [7, 8], sobre todo en problemas con operaciones matriciales, como el problema del PET.

De esta manera, se considera el uso de la GPU para la ejecución de la meta-heurística enfocada a la solución del PET [7, 8].

1.2. Justificación

Por lo general, los computadores actuales integran en su arquitectura dos herramientas de hardware que permiten ejecutar procesos de forma independiente. Por una parte, la CPU es el encargado de ejecutar los procesos de una gran mayoría de software de cálculo y programación. Por otro lado, la GPU es comúnmente usada para procesos gráficos que se realizan en paralelo a gran velocidad. Este último permite ejecutar una mayor cantidad de procesos en paralelo que la CPU.

La GPU ha sido considerada para mejorar problemas de cálculo en los que intervengan procesos repetitivos e independientes, como los que comúnmente se encuentran en las operaciones matriciales. Aquellos cálculos que, en lugar de realizarlos en serie, sería mejor ejecutarlos en paralelo para disminuir el tiempo en que se obtiene una solución. Dentro del contexto de problemas con cálculos independientes, se ha dado cabida a los algoritmos meta-heurísticos. La formulación de estos algoritmos básicamente se da con vectores y matrices. Por esta razón, se usa un procesador gráfico como una herramienta para mejorar el rendimiento [9].

La GPU ha ayudado considerablemente a problemas relacionados con sistemas artificiales, redes neuronales y bioinformática. Todos estos problemas contienen operaciones matriciales que pueden escalar en complejidad y necesitan previamente HPC. Sin embargo, diferentes investigaciones apuntan a que un uso conjunto de la CPU y la GPU mejora el desempeño [10].

1.3. Alcance

En este trabajo de investigación se analizará la factibilidad de disminuir el tiempo de ejecución de los algoritmos de optimización meta-heurísticos mediante el uso de una GPU. Para lo cual será necesario evaluar un grupo de librerías que permiten ejecutar código en la GPU. A partir de su comparación, se seleccionará la librería con mejor desempeño y se adecuará el algoritmo meta-heurístico *Particle Swarm Optimization* (PSO) esa librería.

Consiguientemente se usarán dos problemas específicos que puedan ser resueltos mediante la ayuda

de algoritmos meta-heurísticos. El primer problema se lo conoce como "*Salesman problem*" de dos dimensiones y el segundo es el problema del PET de varias dimensiones. Este segundo problema es considerado complejo debido a que es de PNLEM. Para lo cual se analizará la estructura del problema y se identificará las partes que puedan ser traducidas al lenguaje entendible por la GPU. En el problema del PET las pruebas se realizarán mediante el uso del sistema garver de 6 nodos.

Cada uno de los problemas serán implementados usando la CPU y la GPU. Por lo tanto, se podrá comparar los tiempos de ejecución para cada herramienta de hardware, permitiendo determinar si el uso de la GPU representa una mejora en el rendimiento de los problemas planteados.

1.4. Objetivos

1.4.1. Objetivo general

Analizar la factibilidad de mejorar el desempeño de algoritmos de optimización meta heurísticos usando una GPU.

1.4.2. Objetivos específicos

El presente trabajo tiene los siguientes objetivos específicos:

- Determinar la mejora en el desempeño de los algoritmos de optimización meta-heurísticos enfocados en la solución del PET con respecto a trabajos similares [3, 11].
- Determinar cuales son las características necesarias para ejecutar un algoritmo de optimización meta-heurística en una GPU.

Capítulo 2

Marco teórico y estado del arte

En este capítulo se detalla cada uno de los conceptos que se involucran en el desarrollo del proyecto y todos los conocimientos anteriormente analizados en este campo de estudio.

2.1. Algoritmos Meta-heurísticos

En la actualidad se han modelado varios problemas de diseño estructural, como la distribución de redes eléctricas, redes de fibra óptica o redes de alcantarillado. En estos problemas interviene una gran cantidad de elementos, para los cuales se necesita de diseños que garanticen precisión, eficiencia y rapidez. Estos problemas comúnmente requieren encontrar la mejor opción entre una gran cantidad de posibles opciones. Para lo cual, se requiere de un proceso de optimización que permita encontrar la solución requerida sin la necesidad de probar cada una de las posibles opciones. En este contexto, se han desarrollado los algoritmos meta-heurísticos como una alternativa de optimización [6].

La optimización hace referencia al estudio de un problema con el objetivo de encontrar los valores máximos y mínimos de su función objetivo. En el pasado el uso de métodos matemáticos han permitido encontrar soluciones óptimas. Sin embargo, eran útiles para procesos simples y no para procesos escalables debido a la alta complejidad [12]. Por otra parte, los algoritmos de optimización meta-heurísticos están basados en procesos naturales y físicos, estos algoritmos han permitido implementar procesos que requieren de alta escalabilidad. Entre los procesos naturales en que se han basado los algoritmos meta-heurísticos se encuentran: comportamiento social de especies zoológicas, sonido biológico de los delfines, comportamiento de abejas y hormigas. Y para el caso de procesos físicos, tales como: colisión de cuerpos en una dimensión, ley de refracción de la luz por Snell y evaporación de moléculas de agua [13].

El éxito de este tipo de algoritmos se ha dado en varios campos, tales como: ingeniería, comercio, física, química e incluso política. A estos algoritmos se los puede clasificar en dos tipos, constructivos y de búsqueda local. Los algoritmos constructivos parten de un punto inicial y en cada iteración van mejorando hasta encontrar una solución óptima. Una de las ventajas de este tipo de algoritmos es su velocidad. Por otra parte, los algoritmos de búsqueda local, tratan de moverse hacia soluciones

vecinas esperando que éstas sean mejores. Si no se encuentra una mejor solución, el algoritmo concluye su proceso de evaluación en un óptimo local. Los algoritmos evolutivos son utilizados comúnmente en el diseño estructural. Estos algoritmos parten de un grupo de soluciones candidatas y finalizan cuando llegan a un número definido de iteraciones o cuando convergen a una misma solución. Entre los algoritmos meta-heurísticos evolutivos se encuentran el de PSO y el de *Ant Colony Optimization* (ACO).

2.1.1. Optimización por enjambre de partículas

El algoritmo de optimización por enjambre de partículas (PSO, por sus siglas en inglés) fue creado por Kennedy y Eberhart en 1995. Este está basado en el comportamiento de aves, peces y abejas. Para comprender el funcionamiento del algoritmo, el proceso de búsqueda de alimentos es uno de los más sencillos. Los miembros del enjambre, o partículas, se distribuyen en una posición inicial aleatoria y comparten información entre ellos. Si un individuo encuentra posible comida, esta información se transmite a los otros miembros para que puedan dirigirse a ese sector [6, 14].

El algoritmo PSO se considera una técnica de optimización estocástica. Especialmente utilizado en funciones no lineales. Una de las dificultades que presenta PSO es que su complejidad incrementa en función del número de partículas analizadas. Por otro lado, las soluciones son más eficientes si el número de partículas incrementa [14].

En cuanto a cada partícula, el algoritmo inicializa una población de partículas con soluciones candidatas. Cada partícula posee una velocidad y posición inicial. A medida que el algoritmo avanza, se encuentra una partícula con la mejor solución temporal. Y basado en esa partícula, las otras evolucionan y modifican su posición y velocidad. Este proceso se realiza hasta llegar a un número determinado de iteraciones o hasta que se evidencia la convergencia a una misma solución candidata por parte del enjambre [15]. La velocidad y posición de las partículas del enjambre se expresan en las ecuaciones (2.1) y (2.2) respectivamente.

$$\begin{aligned} V_i(t+1) &= V_i(t) + c_1 \times r_1 \times (P_{ibest} - P_i(t)) + \\ &\quad c_2 \times r_2 \times (P_{gbest} - P_i(t)) \end{aligned} \tag{2.1}$$

- $V_i(t+1)$: Velocidad de la partícula para la siguiente iteración
- $V_i(t)$: Velocidad actual de la partícula
- c_1 *y* c_2 : Constantes de atracción (Pre-definidas)
- r_1 *y* r_2 : Valor aleatorio entre 0 y 1
- P_{ibest} : Mejor posición personal de la partícula
- $P_i(t)$: Posición actual de la partícula
- P_{gbest}: La mejor posición global de todo el enjambre de partículas

$$P_i(t+1) = P_i(t) + V_i(t+1) \tag{2.2}$$

- $P_i(t+1)$: Posición de la partícula para la siguiente iteración

La representación vectorial de la Figura 2.1 muestra la evolución de la velocidad y posición de una partícula en dos dimensiones debido a la mejor partícula temporal y global. El algoritmo meta-heurístico PSO es relativamente simple de programar y se expresa en el Algoritmo 1.

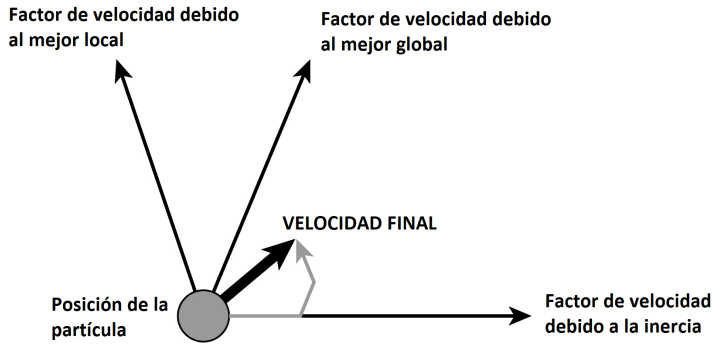


Figura 2.1: Movimiento de la partícula

Algoritmo 1 Meta-heurística PSO

Input: Topología inicial

Output: Mejor topología

1: Inicialización de posiciones aleatorias
2: Inicialización de velocidades aleatorias
3: **while** Criterio de parada **do**
4: **for** i in *cada partícula* **do**
5: Actualizar velocidad de la partícula
6: $V_i(t+1) = V_i(t) + c_1 \times r_1 \times (P_i best - P_i(t)) + c_2 \times r_2 \times (P_g best - P_i(t))$
7: Actualizar posición de la partícula
8: $P_i(t+1) = P_i(t) + V_i(t+1)$
9: Actualizar mejor posición personal
10: Actualizar mejor posición global
11: La mejor partícula obtenida es el resultado

2.2. Unidad de Procesamiento Gráfico y CUDA

La unidad de procesamiento gráfico (GPU, por sus siglas en inglés) ha tenido un gran desarrollo en la última década. En un principio, esta herramienta de hardware fue utilizada para el procesamiento gráfico bidimensional y tridimensional en procesos gráficos. Sin embargo, en los últimos años ha demostrado ser una alternativa viable para la computación científica. Esto debido a que la GPU es un procesador masivo en paralelo con un gran ancho de banda de memoria [16].

Actualmente, se ha determinado que hay un gran impacto en el rendimiento por parte de la GPU. Incluso en el estudio [17] se mostró un incremento de rendimiento de hasta 7 veces el de la CPU. Estas comparaciones se han realizado tomando en cuenta el ancho de banda y gigaflops en cálculos teóricos. Las causas de esta diferencia son su arquitectura y las restricciones físicas de los núcleos.

La principales diferencias entre los dos procesadores es el número de núcleos, arquitectura, ancho de banda de memoria y modelo de programación (SIMT). Por una parte, la GPU está compuesta por cientos de núcleos sencillos que realizan tareas específicas y al mismo tiempo. Por otra parte, la CPU está compuesta por un número menor de núcleos complejos que realizan tareas que demandan mayor rendimiento y pocos procesos en paralelo. Sin embargo, lo recomendable es realizar un uso conjunto de GPU y CPU, lo que se conoce como un computador heterogéneo [9].

Actualmente se sigue mejorando la comunicación entre el programador y la GPU. En las generaciones pasadas de la GPU, era necesario usar una *Application Programming Interface* (API) con lenguajes de programación específicos para comunicarse con la GPU. Sin embargo, la compañía NVIDIA desarrolló una interfaz que permite comunicarse con la GPU mediante lenguajes de programación conocidos. Esta herramienta de software se la conoce como *Compute Unified Device Architecture* (CUDA) y está disponible para la gran mayoría de GPU desarrolladas por NVIDIA. CUDA puede ser usada mediante el lenguaje de programación C++ e incluso por lenguajes como Python, Fortran y Java [18].

Estos algoritmos, al estar constituidos básicamente por operaciones matriciales y vectoriales, son los candidatos idóneos para la optimización mediante la GPU. Tomando en cuenta lo antes mencionado, en este proyecto se analiza el aporte de la GPU al rendimiento de la meta-heurística.

2.3. Planeamiento de la expansión en un sistema eléctrico de transmisión

El problema del PET se basa en la exploración de nuevas formas que permitan expandir el sistema de transmisión respetando criterios pre-establecidos. El sistema eléctrico de transmisión comprende toda la infraestructura necesaria para distribuir energía desde los generadores hasta la carga. La infraestructura debe garantizar la calidad y eficiencia en la que se suministra la energía. De tal forma que satisfaga la demanda requerida sin interrupciones de carga o daños en los equipos. El objetivo del PET es encontrar la manera más económica de expandir correctamente el sistema de transmisión. El PET puede ser considerado como un problema con decisiones estocásticas en el cual se pueden involucrar parámetros como: el tiempo, la ubicación, costo y tipo de líneas de transmisión a añadirse. Adicionalmente, al ser un problema de optimización de enteros mixtos de carácter no lineal. Este puede incrementar su dificultad a medida que la demanda aumenta [19].

El problema matemáticamente está conformado por una función objetivo y las correspondientes restricciones pre-establecidas. La función objetivo está expresada en función de los costos por elemento añadido y la carga conectada.

Para dar solución al problema del PET se pueden usar dos modelos. El primero, llamado modelo DC, es uno de los más utilizados por la comunidad de ingenieros eléctricos debido a que su formulación se basa en ecuaciones lineales. Por lo tanto, resolverlo es relativamente sencillo y obtiene buenas aproximaciones a la solución óptima. Por otro lado, existe el modelo AC con ecuaciones no lineales debido a los parámetros de potencia reactiva que involucra. Este modelo es más complicado de resolver que el modelo DC, sin embargo, obtiene soluciones más eficientes debido a que considera parámetros adicionales.

El PET puede ser dividido en tres horizontes de tiempo. Largo plazo, mediano plazo y corto plazo. Para el largo plazo se plantea un número promedio de 20 años, en el que se analiza un aumento considerable de interconexiones y nuevas fuentes de energía. Para el mediano plazo se plantea un

número promedio de 10 años, en el cual se analiza los niveles de voltaje tanto AC como DC que se podría requerir. Y para el corto plazo se plantea proyectos de menos de 5 años que analicen ajustes finales de acuerdo a alternativas elegidas con anterioridad [20].

2.3.1. Modelo AC

El modelo AC en los sistemas eléctricos es considerado de gran precisión debido a que incorpora el análisis de las potencias reactivas y la compensación de Shunt. Ésta es la principal diferencia con el modelo DC. Sin embargo, al considerar estos nuevos parámetros lo convierten en un PNLEM de carácter complejo. A este conjunto de ecuaciones se lo conoce como flujo de potencia [4].

El modelo AC tiene sus ventajas frente al modelo DC. Este modelo tiene una influencia más real sobre la planificación a futuro, pues al considerar las potencias reactivas se pueden realizar los cambios necesarios eficientemente. Sin embargo, en las investigaciones realizadas hasta el momento es muy común el uso del modelo DC debido a que su resolución presenta menor dificultad. El modelamiento matemático AC se lo puede dividir en 2 problemas:

- Problema de expansión.
- Problema de operación.

El problema de expansión busca minimizar el costo de la inversión al añadir nuevas lineas de transmisión. Este problema se expresa mediante la Ecuación (2.3). Ésta se basa principalmente en el costo por línea, costo por desconexión de carga y restricciones en la cantidad de circuitos admitidos. Como resultado se obtiene el costo total de la inversión.

$$\begin{aligned} \text{mín } \ & v = \textstyle\sum_{(k,l)\in\Omega} (c_{kl} n_{kl} + w) \\ & \textit{sujeto a} \\ & 0 \leq n \leq \bar{n} \qquad\qquad \text{n es entero} \end{aligned} \tag{2.3}$$

- v: Costo de la inversión total con los nuevos circuitos de prueba
- c_{kl} : Costo del circuito entre las barras $k-l$
- n_{kl} : Número de circuitos adicionales entre las barras $k-l$
- n : : Número de circuitos totales entre las barras $k-l$
- $\bar{n}$: : Número máximo de circuitos que pueden ser agregados entre barras
- w : Costo de desconexión de carga
- Ω: Conjunto de caminos de la red

El problema operacional hace referencia a la evaluación de la desconexión de carga en una topología determinada para saber si los generadores existentes satisfacen la demanda, como resultado se obtiene un valor w. Si la evaluación no converge, w toma un valor elevado para que dicha topología no sea considerada. En cuanto a las barras o nodos, son los puntos en donde se conectarán las cargas y generadores, para un mejor entendimiento ver Figura. 3.3. Actualmente existe paquetes de software que permite ejecutar el flujo óptimo AC y conocer el costo de desconexión de carga mediante funciones incorporadas en su código. Los paquetes más popular son MATPOWER, PyPOWER y Pandapower.

2.4. Trabajos relacionados

Una GPU se usa constantemente en proyectos que necesitan mejorar el rendimiento. Procesos en los que se necesita calcular una gran cantidad de datos en paralelo. En una amplia variedad de investigaciones, el uso de una GPU ha sido la solución. Los proyectos que generalmente usaban una CPU ahora se ejecutan a través de una GPU [16].

La comunicación entre el usuario y la GPU es posible debido a las APIs. CUDA es una plataforma que interactúa entre el usuario y la GPU. Adicionalmente, se requiere un lenguaje de programación para usar CUDA, los más comunes son Fortran, C ++ y Python [21]. Aunque el uso de una GPU parece simple, su integración puede ser complicada en ciertos problemas. La transferencia de datos entre la CPU (host) y la GPU toma tiempo. Por lo tanto, su uso puede estar limitado a ciertos casos [22].

En investigaciones previas el uso de una GPU para la ejecución de algoritmos meta-heurísticos ha resultado positiva. Estos algoritmos pueden ser relativamente fáciles de programar y se utilizan para una amplia variedad de problemas con una gran cantidad de posibles soluciones candidatas. Algoritmos meta-heurísticos como ACO y PSO han sido probados y optimizados para ciertos problemas [13, 23, 24].

El problema del vendedor es un claro ejemplo en el que cualquiera de los algoritmos mencionados se puede usar [25]. El problema del vendedor se basa en una lista de ciudades a las que debe ir un vendedor, cada ciudad debe visitarse solo una vez, y con la distancia más corta recorrida, finalmente, el vendedor debe regresar a la ciudad de origen. El uso de la meta-heurística facilita la evaluación de una cantidad determinada de posibles soluciones, llegando a obtener la respuesta óptima con la menor distancia recorrida. Por lo tanto, en este tipo de problemas, ha sido importante traducir la meta-heurística a un lenguaje comprensible por una GPU con el objetivo de mejorar el rendimiento y obtener soluciones rápidas y eficientes [14].

Por otro lado, el problema del PET tiene varios métodos de solución. Por lo general, un PET con modelo DC es menos complejo de resolver. Sin embargo, si se requiere una solución más realista, el modelo de AC es la mejor opción. El modelo AC considera las potencias reactivas lo que lo hacen eficiente. Hasta el momento, no existe mucha investigación en relación al uso de algoritmos meta-heurísticos que consideren el modelo AC para dar solución al problema del PET. El PSO ha demostrado ser uno de los pocos algoritmos útiles en este problema y se detalla en varias investigaciones [3, 11, 26].

Capítulo 3

Metodología de implementación

En este capítulo se presentan las herramientas que han permitido analizar la factibilidad de optimizar los algoritmos meta-heurísticos mediante el uso de la GPU. Como primer punto, ha sido fundamental encontrar el lenguaje de programación que pueda comunicarse con la GPU, ya sea mediante librerías adicionales o nativas. A partir del lenguaje seleccionado, se buscó la librería con mejor desempeño en cuanto a tiempo de ejecución. Por otro lado, fue necesario buscar una librería externa que permita ejecutar el flujo óptimo sobre un sistema de transmisión eléctrico. Además, la herramienta debe incorporar el modelo AC. Finalmente, con las herramientas seleccionadas se programó el algoritmo meta-heurístico PSO enfocado a la solución del problema del PET.

3.1. Comunicación con la GPU

Los lenguajes de programación que permiten la incorporación de CUDA son: Python, C++ y Fortran. De estos, se optó por Python debido a su facilidad de programación y a la gran comunidad que posee. A pesar de ser fácil de programar, Python es considerado una de las herramientas más potentes que existen en la actualidad [27]. Previamente a la selección de Python, se consideró Octave, el cual se dejó de utilizar debido a la incompatibilidad de herramientas, tanto para el problema del PET como para la GPU [28].

Python es compatible con CUDA, y además, posee una gran variedad de librerías que permiten comunicarse con la GPU mediante el uso de CUDA. Previamente se hizo un análisis de las librerías disponibles, de éstas se obtuvieron 3 sobresalientes: Cupy, Numba y Theano. Adicionalmente, se tomó en cuenta la librería dedicada a cálculos científicos de Python conocida como Numpy. Lo que sirvió como referencia del rendimiento de la CPU [29].

Cupy es una librería de uso libre que utiliza la plataforma de CUDA para comunicarse con la GPU. Lo que hace a Cupy especial es su interfaz de usuario, pues está basada en la librería Numpy y gran parte de sus funciones pueden usarse en Cupy con tan solo cambiar la palabra “numpy” por “cupy” [30].

Numba en cambio es un compilador *Just-in-time* (JIT) que permite traducir parte del código de

python y de la librería Numpy a un lenguaje de bajo nivel. Esta librería también es de acceso libre y utiliza la plataforma CUDA para comunicarse con la GPU. Numba posee una serie de "decoradores" o identificadores de función que se diferencian por el fin para el que se diseñaron. El decorador utilizado comúnmente para operaciones matriciales y vectoriales es "guvectorize" [31, 32].

Por otro lado, Theano es un compilador que fue diseñado para usarse en cálculos científicos. Por lo que es posible ejecutar operaciones matriciales y vectoriales mediante esta herramienta. Además, se comunica, al igual que las anteriores herramientas, con la GPU mediante CUDA [33].

En el capítulo 4 se muestra la comparación de rendimiento entre cada una de las herramientas.

3.2. Modelo AC y Pandapower

En investigaciones relacionadas al problema del PET. Una de las herramientas más utilizadas para ejecutar el flujo óptimo AC es MATPOWER. Esta herramienta es de gran utilidad actualmente para el área de la ingeniería eléctrica debido a la gran cantidad de funciones que posee. Además se la puede ejecutar mediante el software Matlab. Sin embargo, la preferencia de esta investigación ha sido orientarse al uso de software de código abierto. Por lo tanto, el uso de Octave fue una de las alternativas más atractivas debido a su similitud con Matlab. A pesar de este hecho, la herramienta MATPOWER no es completamente compatible con Octave, por lo que este software dejó de usarse [34].

Por otro lado, existen herramientas basadas en MATPOWER, inclusive mejoradas de cierta forma, que son compatibles con Python. Las más conocidas son Pypower y Pandapower. De estas dos la única que se mantiene en desarrollo activamente es Pandapower. Por tal motivo se optó por ésta. Pandapower en realidad combina la biblioteca de análisis de datos "Pandas" y el solucionador de flujo de energía "Pypower". Como resultado se ha obtenido un software de fácil programación destinado a la automatización de análisis y optimización de sistemas de energía. Además, posee una documentación completa en línea con ejemplos y su código es de libre acceso [35]. Pandapower también incorpora el modelo AC en su código, lo cual permite evaluar el problema operacional del PET.

3.3. Sistema de prueba

El problema PET usa el sistema de prueba "Garver de 6 nodos". Este sistema posee 6 nodos, 5 cargas que consumen en total 760 MW y 3 generadores. En total son 15 caminos candidatos, en los cuales la restricción es de máximo 5 circuitos. El número posible de topologías que pueden darse en este sistema se calcula con: $(5+1)^{15} = (6)^{15} = 470$ mil millones de posibles soluciones. Por tal motivo es considerado un problema de explosión combinatorial. La meta-heurística PSO ayuda a reducir el espacio de búsqueda a un número determinado de topologías a evaluarse y mejora las posibles soluciones en cada iteración. El diagrama del sistema de prueba se muestra en la Figura. 3.1.

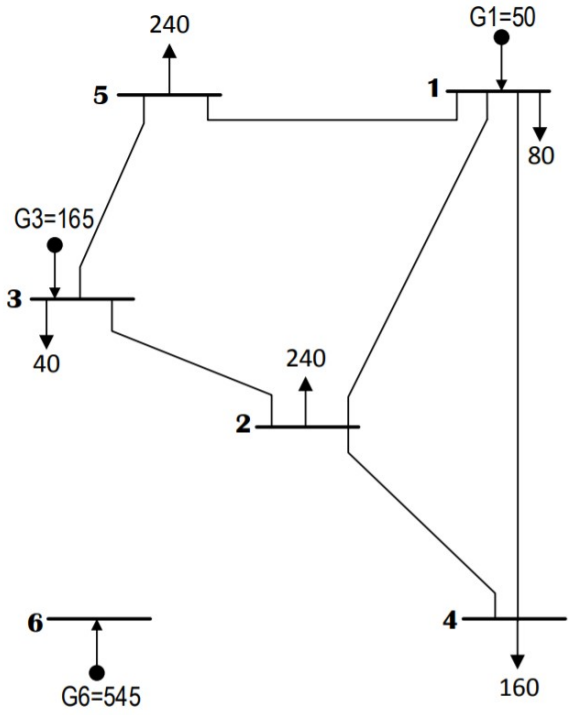


Figura 3.1: Diagrama del Sistema de prueba “Garver de 6 nodos”

3.4. Meta-heurística PSO y el problem PET

EL uso de la meta-heurística PSO está enfocado a tratar el problema de expansión descrito en la sección 2.3. Para ello cada topología candidata recibe el nombre de partícula en el algoritmo mencionado. Cada partícula es creada inicialmente de forma randómica y basándose en la sistema de prueba de la sección 3.3. La población inicial se acomoda en una matriz de tamaño $m \times n$. Donde m es el número de partículas iniciales y n la dimensión del problema PET, el cual depende del número de caminos disponibles que puedan añadir un circuito, es decir, en este caso 15 para el sistema Garver de 6 nodos. Esta matriz representa la posición inicial en el algoritmo meta-heurístico.

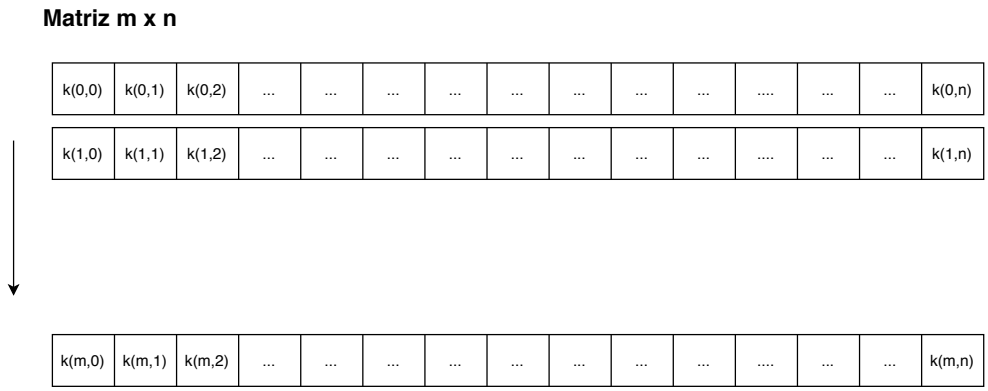


Figura 3.2: Matriz $m \times n$ de topologías

En los valores de $k(m,n)$ se colocan números del 0 al 5, dada la restricción del problema que limita a 5 circuitos como máximo en cada camino para este problema. De igual forma se crea una matriz de $m \times n$ correspondiente a las velocidades iniciales de cada partícula, tomando en cuenta parámetros constantes utilizados en los artículos [4, 11].

A partir de este proceso, el algoritmo meta-heurístico se desarrolla mediante las ecuaciones 2.1 y 2.2 con operaciones entre matrices y arreglos.

3.5. Paralelización de la meta-heurística

La paralelización es útil para algoritmos meta-heurísticos que se basan en la creación de una población de posibles soluciones, como es el caso del algoritmo PSO. El objetivo principal es acelerar el proceso de búsqueda, mejorar la solución obtenida y darle robustez a la meta-heurística. Al paralelizar, el número de partículas evaluadas puede incrementarse sin afectar la capacidad computacional disponible. De tal forma, las ecuaciones (2.1), (2.2) y (2.3), han sido paralelizadas.

La paralelización de las ecuación (2.1) y (2.2), permite actualizar las velocidades y posiciones del enjambre en menos tiempo. En este caso, es posible hacerlo porque las operaciones que intervienen en dichas ecuaciones son básicamente multiplicación y suma de matrices o arreglos. Sin embargo, se ha tenido que evaluar su eficiencia en problemas de dos y más dimensiones.

Por otro lado, la ecuación (2.3) se describe como la función objetivo del problema PET. De tal forma, al revisar su estructura, es notable que el valor w forma parte del problema operacional del PET y este no puede ser paralelizado, ya que depende directamente de la herramienta que evalúa la desconexión de carga de la topología. Para paralelizar la función objetivo debería obtenerse todos los valores de desconexión de carga y almacenarlos en un arreglo para acceder a ellos directamente. Sin embargo, esto no representa un cambio sustancial ya que la desconexión de carga de las topologías sigue evaluándose en serie. Una última solución es paralelizar los subprocesos de la herramienta que evalúa la desconexión de carga y así obtener el valor de w paralelamente. Para ello es necesario un análisis más extenso de la herramienta que se esté utilizando, en este caso Pandapower.

Por lo tanto, el diagrama general de paralelización de la meta-heurística PSO se muestra en la Figura. 3.3. Ciertas partes de la meta-heurística no necesariamente deben ser paralelizadas, ya que existen funciones optimizadas en la CPU, ahorrando tiempo adicional debido a la transferencia de datos entre la CPU y la GPU.

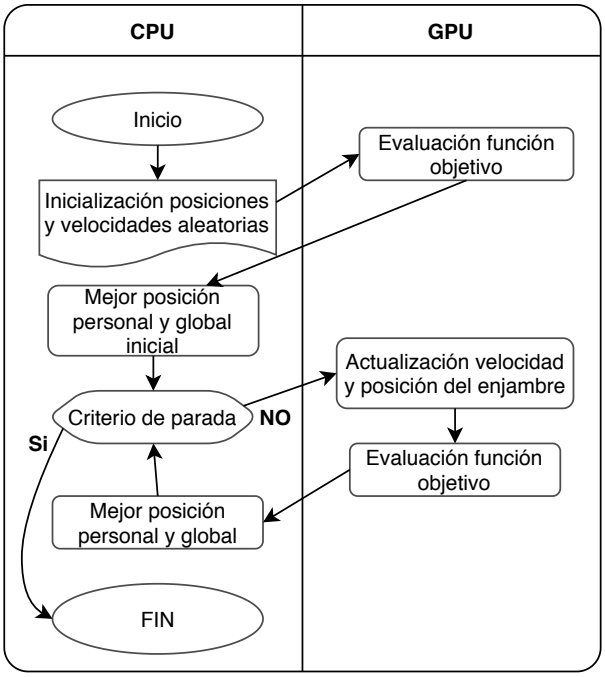


Figura 3.3: Diagrama GPU-CPU meta-heurística PSO

3.6. Integración de herramientas

La integración de las herramientas de software y hardware ocurre en las partes del algoritmo a las cuales aportan cada una de ellas. En el problema de expansión del PET, la meta-heurística se implementa mediante Python y las librerías que permiten ejecutar el código en la GPU. Esto en cuanto a la creación, manipulación y actualización de la posición y velocidad de las partículas del algoritmo. La misma metodología se aplica a cualquier algoritmo meta-heurístico. El objetivo es optimizar las ecuaciones que gobiernan el mismo. Dado que se basan en su mayoría en operaciones vectoriales y matriciales.

Por otra parte, el problema operacional se ejecuta mediante la herramienta Pandapower y los datos obtenidos del flujo de potencia para cada partícula se almacenan y se agregan en la ecuación del problema de expansión del PET.

3.7. Hardware y software

Las pruebas fueron realizadas en una laptop Predator Helios 300. El procesador integrado del ordenador es el i7 8750h desarrollado por la compañía Intel. Este procesador posee 6 núcleos y 12 hilos. También posee un procesador gráfico integrado en el chip, sin embargo, éste no fue utilizado.

Por otra parte, el ordenador también posee una unidad de procesamiento gráfico Geforce GTX 1060 desarrollado por la compañía NVIDIA. Esta unidad tiene 6 GB de VRAM, es compatible con CUDA, está conformada por 1280 núcleos CUDA con una arquitectura de tipo Pascal. Adicionalmente, en la tarjeta madre de la laptop se integran 16 GB de memoria RAM.

Es necesario que las versiones de las herramientas de software utilizadas sean compatibles. De tal forma, el sistema operativo usado fue Windows 10 puesto que la distribución Ubuntu/Linux presentó

incompatibilidad con la herramienta CUDA. La versión de CUDA que se instaló fue la 10.1.168. La versión de Python seleccionada fue la 3.7 con la plataforma Anaconda para la instalación del entorno, es decir, paquetes y librerías. Las herramientas utilizadas para conectar con la GPU mediante CUDA fueron: numba 0.5, cupy 7.6.0 y theano 1.0.5. Todas estas compatibles con la versión 3.7 de Python. En cambio como referente del rendimiento de la CPU se usó la librería numpy 1.16.

Capítulo 4

Pruebas y análisis de los resultados

En este capítulo se detallan los resultados obtenidos de cada una de las pruebas que se llevaron a cabo, con la finalidad de analizar la factibilidad del uso de la GPU para optimizar algoritmos meta-heurísticos. Los resultados se han dividido en 3 secciones. La primera comprende la comparación de las librerías que permiten ejecutar código en la GPU. En la segunda se analiza el uso de la GPU para mejorar el rendimiento del algoritmo meta-heurístico con la ayuda de un problema bidimensional. Por último, se analiza el rendimiento al incorporar en el algoritmo meta-heurístico el uso de la GPU aplicado al problema PET.

4.1. Comparación de librerías o compiladores

Las ecuaciones de posición y velocidad del algoritmo meta-heurístico PSO están basadas en operaciones matriciales elemento a elemento de multiplicación y suma. De tal forma, la comparación de librerías comprende la ejecución de estas operaciones con matrices cuadradas de diferentes dimensiones comenzando de menor a mayor dificultad. El objetivo es tomar en cuenta el tiempo de ejecución como parámetro principal de comparación y seleccionar la librería que se ejecute en el menor tiempo. Para asegurar que los resultados sean comparables, se utilizaron las mismas matrices para evaluar cada librería.

La comparación de las librerías en esta primera instancia no evalúa por completo el algoritmo meta-heurístico sino solo las operaciones elementales que lo conforman. Esto debido a que la paralelización se lleva a cabo en ciertas partes del algoritmo como se muestra en la Figura. 3.3. Además, los resultados obtenidos se pueden utilizar como referencia para otros algoritmos meta-heurísticos debido a que generalmente se utilizan operaciones matriciales en su estructura.

De esta forma, para la suma se ejecutaron operaciones en matrices de dimensión 100×100 hasta matrices de 5000×5000. No fue posible ejecutar una suma con una dimensión mayor a 5000 debido al límite de memoria de la GPU. Al evaluarse de forma conjunta para evitar diferencias en los resultados, la memoria disponible en la GPU se comparte entre las librerías analizadas. En concreto, debido a la deficiente optimización de la librería Numba, se obtuvo un error de limitación de memoria, lo que

impedía analizar el problema para una dimensión superior a 5000×5000.

En cuanto a los resultados similares para las librerías Numpy y Cupy en la operación suma de matrices, se debe a que ambas librerías están escritas en lenguaje C, por lo que superar a numpy no es algo sencillo.

Los resultados para la suma de matrices se muestran en la Figura 4.1 y la Tabla 4.1.

Dimensión	Theano	Cupy	Numba	Numpy	Matlab
100	12.90	0.99	7.01	0.96	0.03
500	24.90	1.99	74.80	0.97	0.37
1000	33.90	4.98	270	2.98	2.40
2000	96.80	17.90	929	13.90	8.00
3000	144	35.90	1,980	27.80	19.80
4000	234	57.80	–	52.80	35.60
5000	366	90.70	–	90.70	55.30

Tabla 4.1: Comparación de librerías (tiempo en milisegundos) - Suma

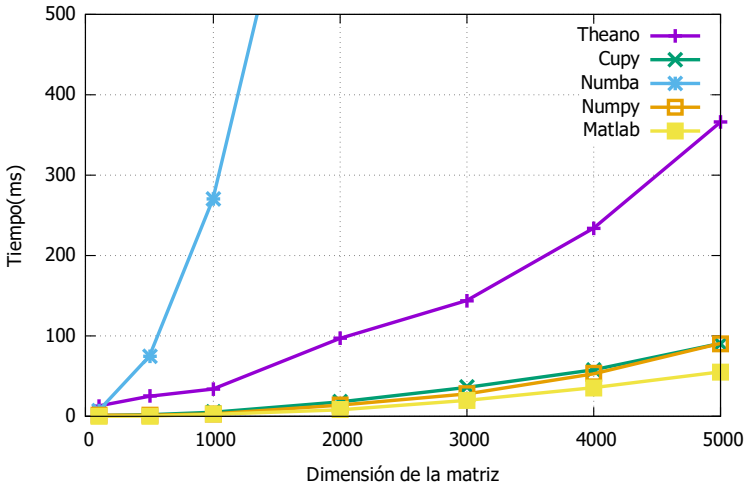


Figura 4.1: Comparación de librerías - Suma de matrices

Por otra parte, la multiplicación de matrices elemento a elemento se llevó a cabo desde una dimensión de 100×100 hasta una dimensión de 10000×10000. En este caso, la evaluación se realizó hasta una dimensión superior a la de la suma, debido a que actualmente la mayoría de paquetes de software utilizan algoritmos optimizados de multiplicación, puesto que la comunidad relacionada al cálculo matemático se ha enfocado a esta operación en específico. Además, la declaración del kernel influye significativamente en el tiempo de ejecución dado que al variar el uso entre memoria global, compartida y local, se obtienen resultados diferentes, lo cual no esta explícito por cada librería. También es notorio que las librerías Theano y Cupy tienen tiempos similares hasta cierta dimensión, esto es debido al modelo de paralelización con el cual han sido programadas. Es decir, hasta un cierto punto la librería tendrá el mismo tiempo de ejecución debido a que el umbral de operaciones posibles no se ha superado, lo cual no afecta su rendimiento. Esto es común en pruebas de estrés realizadas a software de cálculo.

Los resultados de esta operación se muestran en la Tabla 4.2 y la Figura 4.2.

Dimensión	Theano	Cupy	Numba	Numpy	Matlab
100	16.90	0.00	4	0.00	0.09
500	21.90	0.00	70	1.00	4.10
1000	13.90	1.00	264	4.02	2.50
2000	14.90	1.00	907	13.90	7.90
3000	19.90	1.00	1,903	37.80	16.50
4000	19.90	0.96	3,390	52.80	30.30
5000	13.90	1.99	5,242	83.80	59.60
6000	12.90	1.99	7,690	146	91.80
7000	18.90	2.99	10,168	206	117.90
8000	19.90	3.98	–	227	149.50
9000	13.90	4.99	–	302	194.60
10000	14.90	5.98	–	389	204.20

Tabla 4.2: Comparación de librerías (tiempo en milisegundos) - Multiplicación

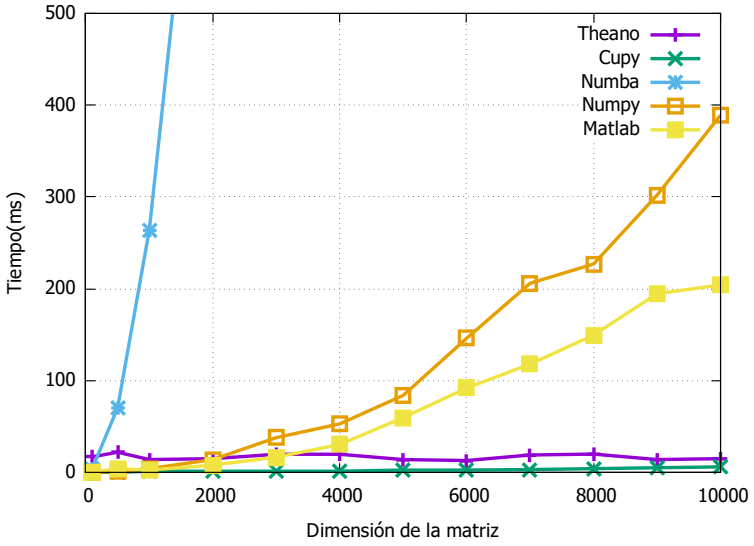


Figura 4.2: Comparación de librerías - Multiplicación de matrices

En el caso de la suma, Matlab y las librerías Numpy y Cupy muestran un rendimiento similar. Por otro lado, en la operación de multiplicación Cupy ejecuta la operación en menor tiempo. De tal forma, debido a su rendimiento y facilidad de instalación, se optó por usar la librería Cupy para optimizar los algoritmos mediante la GPU en este proyecto.

4.2. Meta-heurística y la GPU

En este caso, la meta-heurística PSO fue utilizada para solucionar el "Salesman Problem" bidimensional. Por tal motivo, se programó en Python dos versiones de este problema. La primera versión se ejecuta totalmente en la CPU mediante la librería Numpy. En la segunda versión se crearon y manipularon variables, vectores y matrices para usar la GPU mediante la librería Cupy de acuerdo al diagrama de la Figura. 3.3. Esto ha permitido evaluar el rendimiento de la meta-heurística por el uso de esta unidad de procesamiento.

Los parámetros de comparación en este caso han sido el número de iteraciones y el número de partículas versus el tiempo de ejecución. El número de iteraciones se varió desde 10 a 1000 y el número de partículas se mantuvo en 40. Los resultados se muestran en la Tabla 4.3 y la Figura 4.3. Por otro lado, el número de partículas se varió desde 40 a 300 y las iteraciones se mantuvieron en 20. Los resultados se muestran en la Tabla 4.4 y la Figura 4.4.

Iteración	PSO-CPU	PSO-GPU
10	0.21	0.37
50	1.08	1.61
100	2.16	3.14
200	4.35	6.21
300	6.54	9.29
400	8.65	12.65
600	12.96	18.62
800	17.66	25.21
1000	22.13	31.87

Tabla 4.3: Meta-heurística en CPU vs GPU (tiempo en segundos)

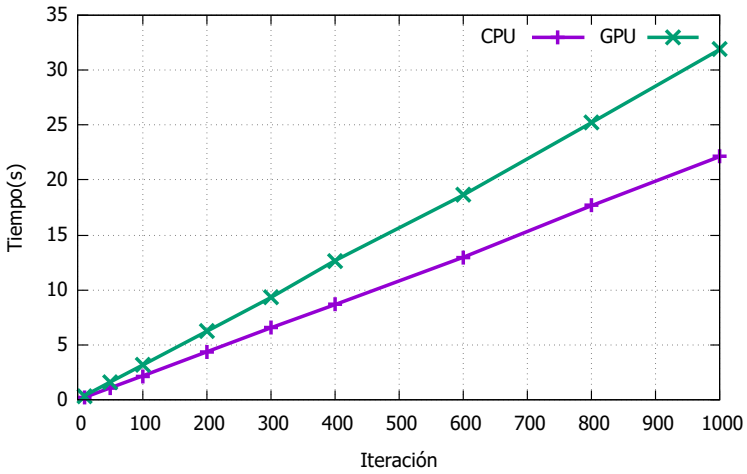


Figura 4.3: Comparación de la meta-heurística PSO - CPU vs GPU

# Partículas	PSO-CPU	PSO-GPU
40	0.42	0.61
80	1.71	1.21
120	3.99	1.82
200	10.63	3.07
300	24.45	4.63

Tabla 4.4: Meta-heurística en CPU vs GPU (tiempo en milisegundos)

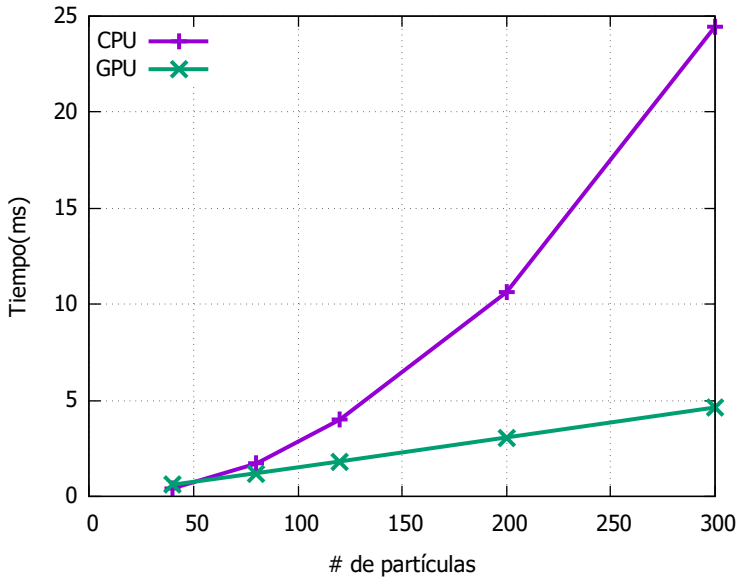


Figura 4.4: Comparación de la meta-heurística PSO por el número de partículas - CPU vs GPU

Al incrementar el número de iteraciones, el aporte de la GPU a la meta-heurística no es positivo. Pues el rendimiento es mejor en la CPU. Sin embargo, el objetivo no es incrementar el número de iteraciones debido a que la meta-heurística está diseñada para encontrar una solución en el menor número de repeticiones posibles. Por lo que no sería necesario una mejora de acuerdo a este parámetro. De igual forma, se consideró analizarlo.

Por otro lado, el incremento de partículas en una meta-heurística es de gran aporte. Esto debido a que se podrá hallar una mejor solución con un mayor número de soluciones candidatas analizadas. Para este caso, el uso de la GPU es una ayuda significativa. Los resultados apuntan a un incremento de partículas y un menor tiempo de ejecución en comparación con el algoritmo desarrollado en la CPU. Por tal motivo, se ha considerado a la GPU como una gran alternativa para mejorar el rendimiento del problema del PET en la sección 4.3.

4.3. PET, Meta-heurística y GPU

El problema del PET, como caso de estudio, se ha implementado usando la meta-heurística PSO y la librería Cupy para ejecutar el código en la GPU. De igual forma, en esta sección se han desarrollado dos versiones del código. La primera versión se ejecuta solo en la CPU y la segunda versión ejecuta la meta-heurística mediante el uso de la GPU de acuerdo al diagrama de paralelización de la Figura. 3.3. Es necesario recalcar que la función objetivo del problema PET no fue paralelizada debido a que depende directamente de la herramienta Pandapower y esta solo puede ser ejecutada secuencialmente. En este caso, los resultados obtenidos no han sido favorables para el uso de la unidad de procesamiento gráfico y se muestran en la Figura 4.5 y la tabla 4.5. Esto es debido a que al ser un problema de más de dos dimensiones los valores aleatorios de la ecuación de velocidad se incorporan mediante indexación de matrices, lo cual significa que se debe acceder a un segmento de la matriz de posibles soluciones y multiplicar los valores aleatorios por ese segmento. Esta función en las librerías analizadas no está bien optimizada, debido a que comúnmente la GPU es usada en problemas de “Una sola instrucción,

múltiples datos" SIMD o de menor dificulta en la que no sea necesario acceder a un determinado segmento. Por tal motivo, el tiempo de ejecución es afectado considerablemente.

El problema operacional fue solventado mediante el uso de la herramienta Pandapower. Esta herramienta ha permitido crear la red fácilmente debido a la gran cantidad de funciones que posee. Sin embargo, a pesar de ser una buena herramienta de diseño, el tiempo de creación de la red es mayor en comparación con otras herramientas como Matpower. La cual permite modificar la estructura de la topología rápidamente.

# Partículas	TEP-CPU	TEP-GPU
10	2.632	26
20	4.835	456
40	9.314	524
80	19.456	613
100	24.86	801.4

Tabla 4.5: TEP en CPU vs TEP en GPU - 10 iteraciones (tiempo en milisegundos)

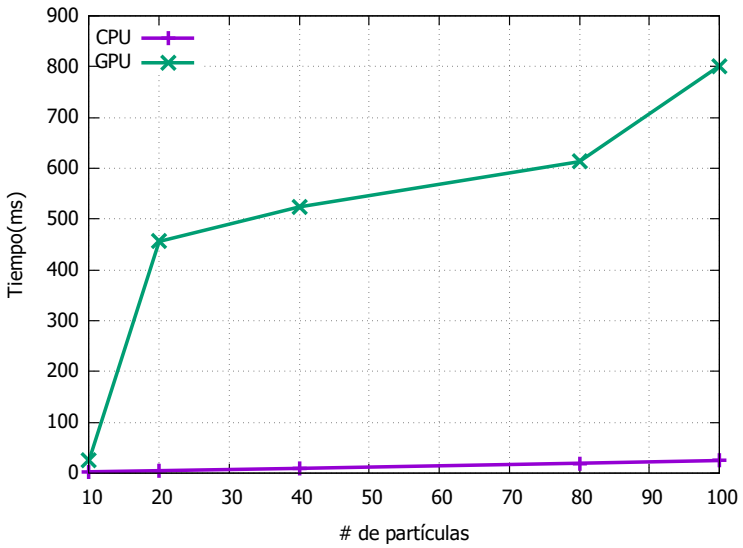


Figura 4.5: Rendimiento de la GPU en el problema TEP

Adicionalmente, se ha comparado la herramienta Matpower y Pandapower, las cuales solventan el problea operacional, en relación a la solución óptima obtenida por cada herramienta. Los resultados se muestran en la Figura 4.6. Es notorio que la meta-heurística en ambos casos funciona perfectamente, en la quinta iteración llegan a una solución óptima que no puede ser mejorada y en la décima iteración el proceso de búsqueda se detiene debido al criterio de parada. Esta similitud se debe a que la meta-heurística fue implementada usando los mismos valores constantes que se usaron en [11]. Sin embargo, de acuerdo a los costos obtenidos hay una gran diferencia, lo que se debe a la diferencia en la formulación del flujo óptimo AC y el código de implementación de cada herramienta, lo que significa una falta de homogeneidad en los parámetros considerados por cada una. A pesar de que las dos soluciones no sean iguales, no se puede considerar a una de ellas como incorrecta. Esto debido a que la meta-heurística converge a una posible solución óptima y no es un método exacto.

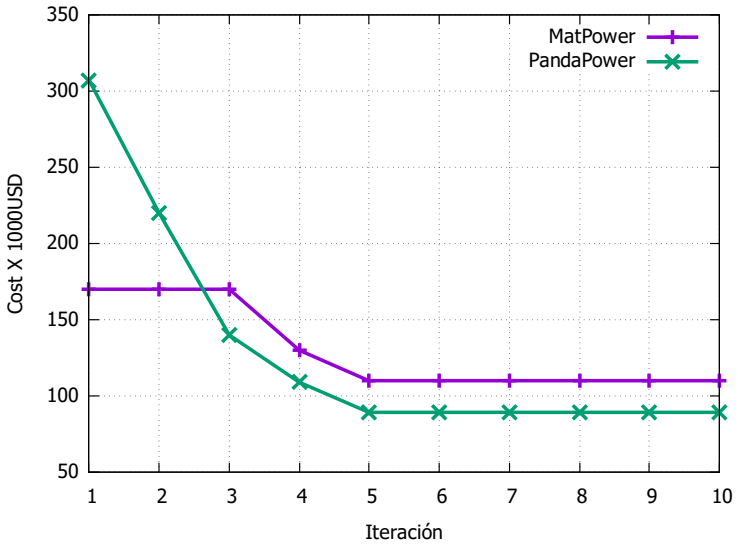


Figura 4.6: Convergencia de Pandapower y Matpower

Capítulo 5

Conclusiones y Recomendaciones

5.1. Conclusiones

De acuerdo a los resultados obtenidos en este proyecto, la GPU sí representa una mejora sustancial para el algoritmo meta-heurístico aplicado al "Salesman problem". Con la ayuda de la GPU la meta-heurística PSO puede evaluar un número mayor de partículas o soluciones candidatas en menor tiempo que la CPU. Debido a esto la solución que la meta-heurística obtiene es más eficiente.

Por otro lado, en el caso de la meta-heurística aplicada al problema del PET. La GPU no tuvo un impacto positivo. El tiempo de ejecución para el problema del PET es mayor para la GPU. Este problema es debido a que las librerías de Python para uso de la GPU no tienen optimizada la función de indexación de matrices. Por este motivo, el tiempo de ejecución se ve afectado considerablemente. Otra causa de la ralentización es debido a la comunicación entre funciones de la CPU y GPU. Al ser un problema complejo, ciertas funciones no están disponibles para la GPU, como transposición de matrices, creación de valores randómicos y obtención de valores mínimos en una matriz o arreglo, por lo que se deben usar funciones existentes en la CPU y la transferencia de datos entre estas unidades de hardware no aporta a la disminución del tiempo de ejecución.

Todo algoritmo meta-heurístico cumple con las características para ejecutarse mediante una GPU debido a que son altamente paralelizables. También utilizan vectores y matrices en sus ecuaciones. De esta manera, el número de partículas analizadas se incrementa con respecto a la CPU y en el mismo tiempo de ejecución. Sin embargo, el uso de la GPU se recomienda aplicarlo en problemas que no contengan operaciones con indexación de matrices.

5.2. Recomendaciones

Si bien el uso de Python ha mostrado ser una buena alternativa para ejecutar código en la GPU a través de múltiples librerías, además de ser un lenguaje de programación fácil, se recomienda considerar otro tipo de lenguajes. Una de las opciones más recomendables es C++, pues también es compatible con CUDA. La dificultad agregada para C++ es su programación debido a que es considerado un

lenguaje de bajo nivel. Por tal motivo, la estructura del código se asemeja más a un lenguaje de máquina.

Otra recomendación es usar tarjetas gráficas NVIDIA Quadro debido al número de núcleos que poseen. A diferencia de las tarjetas tradicionales, que comúnmente se usan para videojuegos. Las tarjetas de la serie Quadro son utilizadas específicamente para herramientas de diseño, donde es necesario paralelizar una mayor cantidad de datos. Lo que se traduce en un mayor número de cálculos simultáneos en problemas matemáticos. Esta recomendación es valida para problemas de optimización meta-heurísticos que no necesiten una indexación de matrices para ejecutar ciertos cálculos.

5.3. Trabajos futuros

Es necesario una evaluación a profundidad de las herramientas que se encargan de solventar el problema operacional, mismo que hace referencia al cálculo del costo por desconexión de carga en una topología de distribución eléctrica. Las herramientas a evaluarse deben ser MATPOWER y Pandapower debido a su importancia en el campo del planeamiento de sistemas eléctricos de transmisión. Además de buscar una tercera solución alternativa en código C++ e intentar paralelizarla.

Anexo A

Código de implementación del problema TEP

El código completo de implementación para el problema PET comprende la creación inicial del enjambre de partículas, incorporación del problema operacional al problema de expansión, algoritmo meta-heurístico enfocado al PET y ejecución de la meta-heurística mediante la GPU. La implementación completa en Python se encuentra en el Listado A.1.

```
import numpy as np
import cupy as cp
import random as rand
import matplotlib.pyplot as plt
import pandapower as pp
from numpy import array
import numpy as np
#Librería para GPU
from numpy import matlib
from numba import guvectorize, float64
from timeit import default_timer as timer
#DATOS IMPORTANTES
LineasIniciales = np.array([1,0,1,1,0,1,1,0,0,0,1,0,0,0,0])
PN = 40;      #Número de partículas
dim = 15;        #La dimensión es el número posible de vías que se pueden crear en la topología
Xmax = 5;        #Número máximo de líneas que se pueden agregar entre nodos.
Xmin = LineasIniciales     #Topología incial del problema

chi= 0.729;
c1 = 2.05;
c2 = 2.05;

#Creación Xmax
Xmax = Xmax;
Xmax = np.matlib.repmat(Xmax,dim,PN)
#Creación Xmin
Xmin= np.matlib.repmat(Xmin, PN,1)
Xmin = np.transpose(Xmin)
#Velocidad máxima
```

```
vmax = 0.5*(Xmax-Xmin)

TEPtotalgpu = 0

def CreacionInicial(LineasIniciales, PN, dim, Xmax, Xmin):

    from numpy import array

    #SE ASIGNA UN NÚMERO DE LÍNEAS PARA CADA DERECHO DE VÍA DE FORMA ALEATORIA,
    SIGUIENDO LOS SIGUIENTES PARÁMETROS.

    swarm=np.round(Xmin+(Xmax-Xmin)*(np.random.rand(dim,PN)));
    print(swarm)
    for i in range(0,PN):
        alpha=(0.6+0.25*np.random.rand());
        indices=np.random.randint( np.round(alpha*dim),size=15);
        indices = np.transpose(indices);
        swarm[indices,i]=Xmin[indices,i];

    print("Posicion inicial del enjambre 100 particulas")

    for j in range(0,PN):
        print(swarm[:,j])

    return swarm
#OBTENCIÓN DE LAS POSICIONES DE LAS PARTÍCULAS
swarm2 = CreacionInicial(LineasIniciales,PN, dim, Xmax, Xmin)
#INICIALIZACIÓN DE LA MEJOR POSICIÓN
bestpos = swarm2;
#Velocidad inicial
vel = (2*np.random.rand(dim, PN))*(Xmax/2)-(Xmax/2);

def EvaluacionParticulas(swarm, PN):
    import pandapower as pp
    costo =np.array([40,38,60,20,68,20,40,31,30,59,20,48,63,30,61])*1000
    coste = np.empty((PN,1))
    for i in range (0,PN):
        net = pp.create_empty_network()
        penalizacion = 0
        min_vm_pu = 0.95
        max_vm_pu = 1.05

    # TOPOLOGÍA PARA UN SISTEMA GARVER DE 6 NODOS

        bus1 = pp.create_bus(net, vn_kv = 110,geodata=(10,20), min_vm_pu = min_vm_pu,
    max_vm_pu = max_vm_pu)
        bus2 = pp.create_bus(net, vn_kv = 110, geodata=(5,15), min_vm_pu = min_vm_pu,
    max_vm_pu = max_vm_pu)
        bus3 = pp.create_bus(net, vn_kv = 110, geodata=(1,18), min_vm_pu = min_vm_pu,
    max_vm_pu = max_vm_pu)
        bus4 = pp.create_bus(net, vn_kv = 110, geodata=(10,10), min_vm_pu = min_vm_pu,
     max_vm_pu = max_vm_pu)
        bus5 = pp.create_bus(net, vn_kv = 110, geodata=(5,20), min_vm_pu = min_vm_pu,
    max_vm_pu = max_vm_pu)
        bus6 = pp.create_bus(net, vn_kv = 110, geodata=(1,10), min_vm_pu = min_vm_pu,
```

```
    max_vm_pu = max_vm_pu)

        vias = [[bus1,bus1,bus1,bus1,bus1,bus2,bus2,bus2,bus2,bus3,bus3,bus3,bus4,bus4
    ,bus5],
                [bus2,bus3,bus4,bus5,bus6,bus3,bus4,bus5,bus6,bus4,bus5,bus6,bus5,bus6
    ,bus6]]



        #Creación de las lineas a 110 Kw con un bucle
        #Asignación del derecho de vía correpondiente, dependiendo si existe o no la v
    ía.

        cont = 0;
        for j in swarm[:,i]:
            if j != 0:
                for k in range (np.int(j)):
                    exec('l{} = pp.create_line(net, vias[0][cont],vias[1][cont],
    parallel=4,max_loading_percent = 50, length_km = 1., std_type = "149-AL1/24-ST1A
    110.0")'.format(cont))
            cont = cont +1;
        print("Se crearon las lineas AC")


        #Creación de los generadores

        g1 = pp.create_gen(net,bus1, p_mw = 1000, min_p_mw =0, max_p_mw =450 ,
    controllable = True, slack= True)
        g2 = pp.create_gen(net,bus3, p_mw = 600, min_p_mw =0, max_p_mw =450 ,
    controllable = True)
        g3 = pp.create_gen(net,bus6, p_mw = 600, min_p_mw =0, max_p_mw =450 ,
    controllable = True)

        #Creación de las cargas

        pp.create_load(net, bus1, p_mw = 80)
        pp.create_load(net, bus2, p_mw = 240)
        pp.create_load(net, bus3, p_mw = 400)
        pp.create_load(net, bus4, p_mw = 180)
        pp.create_load(net, bus5, p_mw = 240)


        #Creación de costos por generador en mw/h

        pp.create_poly_cost(net, element = g1, et = "gen", cp1_eur_per_mw = 40)
        pp.create_poly_cost(net, element = g2, et = "gen", cp1_eur_per_mw = 20)
        pp.create_poly_cost(net, element = g3, et = "gen", cp1_eur_per_mw = 20)

        #Coste por derecho de vía
        costo =np.array([40,38,60,20,68,20,40,31,30,59,20,48,63,30,61])*1000
        #Imprime la topologia actual
        print(swarm[:,i])


        # SE EJECUTA EL SOLVER DE PUNTO INTERIOR AC
```

```
125         #pp.runopp(net, verbose=True) #imrprime todo los resultados
126         print("Comienza solver")
127         pp.runopp(net, numba = True, verbose= False, calculate_voltage_angles = False,
        check_connectivity=False)
128         #Penalizacion
129         from pandapower.optimal_powerflow import a
130         penalizacion = a;
131         print(penalizacion)
132         #Guarda los costes en un array
133         a= net.res_cost + np.sum(swarm[:,i]*costo) + penalizacion;
134         coste[i]= a;
135 #
136 #          plot.simple_plot(net)
137         print("Costo de la topologia")
138         print(np.sum(swarm[:,i]*costo))
139     return(coste)
140
141 #OBTENCIÓN DEL COSTE  DE LAS TOPOLOGÍAS
142 fswarm = EvaluacionParticulas(swarm2, PN)
143 fbestpos = fswarm;    #Inicialización de los costos de la mejor posición
144 #ACTUALIZACIÓN DE ÍNDICES DE LAS MEJORES PARTÍCULAS
145
146 fxopt = np.min(fbestpos);
147 ifxopt = np.argmin(fbestpos);
148
149 #SE EXTRAE LA TOPOLOGÍA DE LA MEJOR POSICIÓN CON EL ÍNDICE DEL MEJOR COSTE
150 xopt =  bestpos[:,ifxopt]
151
152 #Banderas de parada e iteración.
153 success =0;
154 iteracion = 0;
155 fevalcount = 0;
156 STOP= 0;
157
158 MaxIt = 10; # Numero maximo de iteraciones
159 #Array para almacenamietno de los costes de las mejores topologías.
160 FOPT = np.zeros((MaxIt+1,1))
161
162 #LAZO DE EVOLUCIÓN DEL ENJAMBRE
163
164 while STOP == 0:
165     #ACTUALIZACIÓN DE VELOCIDAD
166     #Ecuación de movimiento del enjambre
167
168     for i in range(0,PN):
169         #Variables randomicas de PSO
170         R1 = np.random.rand(dim,1)
171         R2 = np.random.rand(dim,1)
172         R1 = np.transpose(R1)
173         R2 = np.transpose(R2)
174         R1 = cp.asarray(R1)
175         R2 = cp.asarray(R2)
176         bestpos = cp.asarray(bestpos)
177         swarm2 = cp.asarray(swarm2)
178         vel = cp.asarray(vel)
```

```
179         #Cálculo de la velocidad dependiendo de la mejor posicion
180        # G = chi*(vel[:,i] + c1*R1*(bestpos[:,i]-swarm2[:,i])+c2*R2*(bestpos[:,ifxopt
    ]-swarm2[:,i]))
181         start = timer()
182         G = chi*(vel[:,i] + cp.multiply(R1,(bestpos[:,i]-swarm2[:,i]))*2.05+cp.
    multiply(R2,(bestpos[:,ifxopt]-swarm2[:,i]))*2.05)
183         cp.cuda.Stream.null.synchronize()
184
185         #Actualización del valor de la velocidad
186         TEPtime = timer() - start
187         TEPtotalgpu = TEPtotalgpu+TEPtime
188         print(TEPtotalgpu)
189         vel = cp.asnumpy(vel)
190         G = cp.asnumpy(G)
191         vel[:,i] = G;
192     #RESTRICCIÓN DE VELOCIDADADES
193     for i in range(0,dim):
194         for k in range(1,PN):
195             if vel[i,k] > vmax[i,k]:
196                 vel[i,k] = vmax[i,k];
197             elif vel[i,k] < -vmax[i,k]:
198                 vel[i,k] = -vmax[i,k];
199
200     #Actualización del enjambre
201     swarm2 = cp.asnumpy(swarm2)
202     swarm2 = np.round(swarm2 + vel);
203
204     #RESTRICCIÓN DEL ENJAMBRE
205
206     for i in range(0,PN):
207         for k in range(0,dim):
208             if swarm2[k,i] > Xmax[k,i]:
209                 swarm2[k,i] = Xmax[k,i];
210             elif swarm2[k,i] < Xmin[k,i]:
211                 swarm2[k,i] = Xmin[k,i];
212
213     #Evaluar partículas Actualizadas
214     fswarm = EvaluacionParticulas(swarm2, PN)
215
216     if np.min(fswarm) < np.min(fbestpos):
217         bestpos = swarm2;
218         fbestpos = fswarm;
219
220     #ACTUALIZACIÓN DE ÍNDICES DE LAS MEJORES PARTÍCULAS
221
222     fxopt = np.min(fbestpos);
223     ifxopt = np.argmin(fbestpos);
224
225     #SE EXTRAE LA TOPOLOGÍA DE LA MEJOR POSICIÓN CON EL INDICE DEL MEJOR COSTE
226     xopt =  bestpos[:,ifxopt]
227
228
229     if iteracion >= MaxIt:
230         STOP = 1;
231
```

```
232
233     #Almacenamiento de costes de las mejores topologías
234     FOPT[iteracion] = fxopt;
235     iteracion = iteracion +1;
236     print("Vector de Costos de la mejor topologia")
237     print(FOPT)
238     print("Mejor Topologia")
239     print(xopt)
240 print("tiempo PSO en CPU %f seconds." %TEPtotalgpu)
```

Listado A.1: Implementación del problema TEP

ANEXO B

Comparación de librerías en Python

Las librerías que se compararon fueron: Numpy, Cupy, Numba y Theano. Inicialmente se han definido las matrices cuadradas a utilizarse. Para que los resultados sean comparables las matrices son iguales para cada librería. Si bien los elementos de las matrices son iguales, la declaración en el código es diferente para cada una de ellas. De tal forma que en el caso de Cupy, Numba y Theano es necesario registrar la matriz en la GPU.

Como siguiente paso, se define la operación. Existen funciones específicas que ayudan a programar la multiplicación y la suma matricial. La más sencilla es Cupy, ya que es similar a Numpy. Finalmente, se ejecutan la operaciones secuencialmente para que las librerías puedan aprovechar todo el hardware disponible. La implementación completa en Python para la suma se encuentra en el Listado B.1 y para la multiplicación en el Listado B.2.

```
from theano import *
import cupy as cp
import time
import numpy as np
from cupy import dot
from numba import guvectorize, int64, void, float64
from numba import cuda
import pkg_resources, os

import os
os.environ['NUMBAPRO_NVVM']      = r'C:\Program Files\NVIDIA GPU Computing Toolkit\
    CUDA\v10.1\nvvm\bin\nvvm64_31_0.dll'
os.environ['NUMBAPRO_LIBDEVICE'] = r'C:\Program Files\NVIDIA GPU Computing Toolkit\
    CUDA\v10.1\nvvm\libdevice'

stream = cuda.stream()
# MATRIZ CUADRADA


dim = 1000
x = np.random.randint(dim,size=(dim,dim))
y = np.random.randint(dim,size=(dim,dim))
```

```
A = np.random.rand(dim,dim)*1000
A = np.round(A)
w = cp.asarray(x)
z = cp.asarray(y)
cp.cuda.Stream.null.synchronize()

print("Comparación de suma elemento a elemento de las librerías")

#SUMA EN THEANO
start = time.time()
#a = theano.tensor.vector() # declare variable
a = tensor.dmatrix()
out = a+a # construye una expresión simbólica
f = theano.function([a], out) # compila la función
out1 = f(x)
print(out1)
end = time.time()
print("Tiempo con theano - Suma = %s" %(end - start))

#SUMA EN CUPY
start = time.time()
out2= w + w
cp.cuda.Stream.null.synchronize()
print(out2)
end = time.time()
print("Tiempo con Cupy - Suma = %s" %(end - start))

#SUMA EN NUMBA
@guvectorize([void(float64[:,:], float64[:,:], float64[:,:])], '(m,n),(n,p)->(m,p)',
    target='cuda')
def matsum(A,B,C):
        m,n = A.shape
        n,p = B.shape
        for i in range(m):
            for j in range(p):
                C[i,j] = A[i,j]+B[i,j]

#matsum.max_blocksize = 32
d_x = cuda.to_device(A, stream=stream)

start = time.time()
C = matsum(A,A)
print(C)
end = time.time()
print("Tiempo con guvectorize (Numba) - Suma = %s" %(end - start))

#SUMA EN NUMPY

start = time.time()
out3= x + x
print(out3)
end = time.time()
print("Tiempo con Numpy - Suma = %s" %(end - start))
```

Listado B.1: Comparación de de librerías de acuerdo a la suma matricial

```
from theano import *
import cupy as cp
import time
import numpy as np
from cupy import dot
from numba import guvectorize

# MATRIZ CUADRADA

dim = 100
x = np.random.randint(dim,size=(dim,dim))
y = np.random.randint(dim,size=(dim,dim))
A = np.random.rand(dim,dim)*1000
A = np.round(A)
w = cp.asarray(x)
z = cp.asarray(y)
cp.cuda.Stream.null.synchronize()
#

print("Comparación de multiplicación elemento a elemento de las librerías")

#MULTIPLICACIÓN EN NUMBA
start = time.time()
#a = theano.tensor.vector() # declare variable
#
a = tensor.dmatrix()
out = a*a # Construye una expresión simbólica
f = theano.function([a], out) # compila la función
end = time.time()
out1 = f(x)
print(out1)

print("Tiempo con theano - Multiplicación = %s" %(end - start))

#MULTIPLICACIÓN EN CUPY
start = time.time()
out2= cp.multiply(w,w)
cp.cuda.Stream.null.synchronize()
end = time.time()
print(out2)
print("Tiempo con Cupy - Multiplicación = %s" %((end - start)))

#MULTIPLICACIÓN EN NUMBA
#Guvectorize

@guvectorize(['void(int64[:,:], int64[:,:], int64[:,:])'], '(m,n),(n,p)->(m,p)',target
    = 'cuda')
def matmul(A,B,C):
        m,n = A.shape
        n,p = B.shape
        for i in range(m):
            for j in range(p):
                C[i,j] = A[i,j] * B[i,j]

```

```
54
55 start = time.time()
56 C = matmul(A,A)
57 end = time.time()
58 print(C)
59 print("Tiempo con guvectorize (numba) - Multiplicación= %s" %(end - start))
60
61 #MULTIPLICACIÓN EN NUMPY
62
63 start = time.time()
64 out3= np.multiply(x,x)
65 end = time.time()
66 print(out3)
67 print("Tiempo con Numpy - Multiplicación = %s" %(end - start))
```

Listado B.2: Comparación de de librerías de acuerdo a la multiplicación matricial

Bibliografía

[1] A. Arseven, “Mathematical Modelling Approach in Mathematics Education,” ujer, vol. 3, no. 12, pp. 973–980, Dec. 2015, doi: 10.13189/ujer.2015.031204.

[2] M. J. Rider, A. V. Garcia, and R. Romero, “Power system transmission network expansion planning using AC model,” IET Gener. Transm. Distrib., vol. 1, no. 5, p. 731, 2007, doi: 10.1049/iet-gtd:20060465.

[3] S. P. Torres and C. A. Castro, "Parallel particle swarm optimization applied to the static Transmission Expansion Planning problem,"2012 Sixth IEEE/PES Transmission and Distribution: Latin America Conference and Exposition (T&D-LA), Montevideo, 2012, pp. 1-6.

[4] Richard M. Sanchez and Luis R. Villacres, “Planeamiento de la expansión de los sistemas de transmisión considerando incertidumbre,” 150, Universidad de Cuenca, Cuenca-Ecuador, Bachelor Thesis, 2019.

[5] J. Kurniasih, E. Utami and S. Raharjo, "Heuristics and Metaheuristics Approach for Query Optimization Using Genetics and Memetics Algorithm,"2019 1st International Conference on Cybernetics and Intelligent System (ICORIS), Denpasar, Bali, Indonesia, 2019, pp. 168-172.

[6] A. Kaveh, Advances in Metaheuristic Algorithms for Optimal Design of Structures. Springer International Publishing, 2017.

[7] T. V. Luong, N. Melab and E. Talbi, "GPU Computing for Parallel Local Search Metaheuristic Algorithms,ïn IEEE Transactions on Computers, vol. 62, no. 1, pp. 173-185, Jan. 2013

[8] J. M. Domínguez, A. J. C. Crespo, and M. Gómez-Gesteira, “Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method,” Computer Physics Communications, vol. 184, no. 3, pp. 617–627, Mar. 2013.

[9] M. G. Knepley and D. A. Yuen, “Why Do Scientists and Engineers Need GPU’s Today?,” in Lecture Notes in Earth System Sciences, Springer Berlin Heidelberg, 2013, pp. 3–11.

[10] G. Teodoro et al., “Coordinating the use of GPU and CPU for improving performance of compute intensive applications,” in 2009 IEEE International Conference on Cluster Computing and Workshops, 2009, doi: 10.1109/clustr.2009.5289193.

[11] Torres, Santiago P.; Castro, Carlos A.: Éxpansion planning for smart transmission grids using AC model and shunt compensation’, IET Generation, Transmission & Distribution, 2014, 8, (5), p. 966-975, DOI: 10.1049/iet-gtd.2013.0231 IET Digital Library, https://digital-library.theiet.org/content/journals/10.1049/iet-gtd.2013.0231

[12] G. C. Onwubolu and B. V. Babu, New Optimization Techniques in Engineering. Springer Berlin Heidelberg, 2004

[13] M. Dorigo and T. Stützle, “The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances,” in Handbook of Meta-heuristics, Springer US, 2003, pp. 250–285.

[14] Souza, Daniel & Monteiro, Glauber & Martins, Tiago & Dmitriev, Victor & Teixeira, Otávio Noura. (2011). PSO-GPU: accelerating particle swarm optimization in CUDA-based graphics processing units.. 837-838. 10.1145/2001858.2002114.

[15] Ioan Cristian Trelea,The particle swarm optimization algorithm: convergence analysis and parameter selection, Information Processing Letters, Volume 85, Issue 6, 2003, Pages 317-325, ISSN 0020-0190, https://doi.org/10.1016/S0020-0190(02)00447-7.

[16] H. Anzt, T. Hahn, V. Heuveline, and B. Rocker, “GPU Accelerated Scientific Computing: Evaluation of the NVIDIA Fermi Architecture; Elementary Kernels and Linear Solvers,” 2010, doi: 10.11588/EMCLPP.2010.04.11677.

[17] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, “Graphics processing unit (GPU) programming strategies and trends in GPU computing,” Journal of Parallel and Distributed Computing, vol. 73, no. 1, pp. 4–13, Jan. 2013, doi: 10.1016/j.jpdc.2012.04.003.

[18] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA,” presented at the the 13th ACM SIGPLAN Symposium, 2008, doi: 10.1145/1345206.1345220.

[19] Niharika, S. Verma and V. Mukherjee, "Transmission expansion planning: A review,"2016 International Conference on Energy Efficient Technologies for Sustainability (ICEETS), Nagercoil, 2016, pp. 350-355.

[20] Rodriguez, Jorge & Djalma, Falcao & Taranto, G.N.. (2008). Short-Term Transmission Expansion Planning with AC Network Model and Security Constraints.

[21] N. Kasmi, M. Zbakh, S. A. Mahmoudi and P. Manneback, "Taking advantage of GPU/CPU architectures for sparse Conjugate Gradient solver computation,"2015 Third World Conference on Complex Systems (WCCS), Marrakech, 2015, pp. 1-5.

[22] N. V. Sunitha, K. Raju and N. N. Chiplunkar, "Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead,"2017 International Conference on Inventive Communication and Computational Technologies (ICICCT), Coimbatore, 2017, pp. 211-215.

[23] A. J. Nebro, J. J. Durillo, J. Garcia-Nieto, C. A. Coello Coello, F. Luna, and E. Alba, “SMPSO: A new PSO-based metaheuristic for multi-objective optimization,” in 2009 IEEE Symposium on Computational Intelligence in Milti-Criteria Decision-Making, 2009.

[24] Dan Simon, Évolutionary Optimization Algorithms’Biologically-Inspired and Population-Based Approaches to Computer Intelligence , Cleveland State University, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

[25] S. Lin, “Computer Solutions of the Traveling Salesman Problem,” Bell System Technical Journal, vol. 44, no. 10, pp. 2245–2269, Dec. 1965.

[26] Fonseka, J. and Miranda, V. (2004), .ª hybrid meta-heuristic algorithm for transmission expansion planning", COMPEL - The international journal for computation and mathematics in electrical and electronic engineering, Vol. 23 No. 1, pp. 250-262.

[27] R. Dogaru and I. Dogaru, “A Low Cost High Performance Computing Platform for Cellular Nonlinear Networks Using Python for CUDA,” presented at the 2015 20th International Conference on Control Systems and Computer Science (CSCS), May 2015, doi: 10.1109/cscs.2015.36.

[28] James Bergstra and Olivier Breuleux and Frédéric Bastien and Pascal Lamblin and Razvan Pascanu and Guillaume Desjardins and Joseph Turian and David Warde-farley and Yoshua Bengio, Theano: A CPU and GPU math compiler in python, Proceedings of the 9th Python in Science Conference, 2010

[29] Oliphant TE. A guide to NumPy. Vol. 1. Trelgol Publishing USA; 2006.

[30] Prefered Network, inc. and Prefered Ifraestructure, Cupy Documentation, release 7.0.0, inc., Dec 26, 2019.

[31] S. K. Lam, A. Pitrou, and S. Seibert, “Numba,” in Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM ’15, 2015, doi: 10.1145/2833157.2833162.

[32] J. Crist, "Dask & Numba: Simple libraries for optimizing scientific python code,"2016 IEEE International Conference on Big Data (Big Data), Washington, DC, 2016, pp. 2342-2343.

[33] Theano Development Team. “Theano: A Python framework for fast computation of mathematical expressions”, November 21, 2017.

[34] Leon Thurner and Alexander Scheidler and Julian Dollichon and Florian Schäfer and Jan-Hendrik Menke and Friederike Meier and Steffen Meinecke and others, pandapower - Convenient Power System Modelling and Analysis based on PYPOWER and pandas, University of Kassel and Fraunhofer Institute for Wind Energy and Energy System Technology, 2017, http://pandapower.readthedocs.io/en/v1.3.1/downloads/pandapower.pdf.

[35] Ray D. Zimmerman, Carlos E. Murillo-Sanchez and Robert J. Thomas, “ MATPOWER: Steady-State Operations, Planning, and Analysis Tools for Power Systems Research and Education,” IEEE Transactions on power systems, vol. 26, pp. 12–19, 2011.

[36] D. Sapra, R. Sharma and A. P. Agarwal, Çomparative study of metaheuristic algorithms using Knapsack Problem,"2017 7th International Conference on Cloud Computing, Data Science & Engineering - Confluence, Noida, 2017, pp. 134-137.

[37] Lambert-Torres G., Martins H.G., Coutinho M.P., Salomon C.P., Filgueiras L.S. (2008) Particle Swarm Optimization Applied to Restoration of Electrical Energy Distribution Systems. In: Kang L., Cai Z., Yan X., Liu Y. (eds) Advances in Computation and Intelligence. ISICA 2008. Lecture Notes in Computer Science, vol 5370. Springer, Berlin, Heidelberg

[38] Lopéz Lezama Jesús Maria, Gallegos Pareja Luis Alfonso, Flujo de potencia óptimo con restricciones de seguridad usando método de punto interior, Scientia Et Technica. 2008, XIV(39),pp 31-36. ISSN: 0122-1701. https://www.redalyc.org/articulo.oa?id=84920503007.