Universidad Nacional Autónoma de México

Facultad de Ciencias

Paralelización de Meta-heurísticas Bio-inspiradas

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
González del Cueto, Rodrigo

DIRECTOR DE TESIS:
Dra. Katya Rodríguez Vázquez

México, D.F. 2017

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Rodrigo González del Cueto:
Paralelización de Meta-heurísticas Bio-inspiradas. © 2017

Hoja de Datos del Jurado

1. Datos del alumno
González
del Cueto
Rodrigo
55 41 85 28 81
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
301626040

2. Datos del tutor
Dra
Katya
Rodríguez
Vázquez

3. Datos del sinodal 1
Dr
José David
Flores
Peñaloza

4. Datos del sinodal 2
Dr
Luis Miguel
De la Cruz
Salas

5. Datos del sinodal 3
Dr
José de Jesús
Galaviz
Casas

6. Datos del sinodal 4
M. en C.
Manuel
Alcántara
Juárez

7. Datos del trabajo escrito
Paralelización de Meta-heurísticas Bio-inspiradas
123p
2017

RESUMEN

En el campo de la *Inteligencia Computacional*, diversas meta-heurísticas basadas en poblaciones han resultado ser métodos de optimización que proveen una mayor eficiencia en gran variedad de aplicaciones. Sin embargo, estos algoritmos requieren de gran capacidad de cómputo para poder realizar la evaluación de la función objetivo [1] sobre cada uno de los individuos de la población, limitando su utilidad cuando no se cuenta con suficientes recursos de cómputo.

Los algoritmos de algunas de estas meta-heurísticas son inherentemente paralelizables, al igual que la evaluación de la función objetivo sobre el conjunto poblacional, por lo que sus algoritmos pueden ser portados a gran variedad de arquitecturas de cómputo paralelo. A pesar de ello, existen varios detalles tanto en la implementación como en los casos de uso de estas meta-heurísticas que van a determinar si es conveniente portar los algoritmos a cierta arquitectura de cómputo.

En esta tesis se realiza un estudio comparativo sobre la paralelización en *CPU*s y *GPU*s de tres meta-heurísticas bio-inspiradas: *Cúmulo de Partículas*, *Algoritmos Genéticos* y *Evolución Diferencial*. Para este trabajo se diseñó una biblioteca que implementa cada uno de los algoritmos de forma modular, logrando utilizar estrategias de optimización ad-hoc para cada una de las arquitecturas de cómputo y permitiendo analizar los costos de cada una de las operaciones involucradas.

1 Una función objetivo es una función matemática cuyos valores máximos o mínimos representan los objetivos de optimización en el dominio de un problema

LOCUSTA

A mi familia y amigos

AGRADECIMIENTOS

Agradezco:

- ... a Perla y Natalie, por su amor y compañía; ustedes me han dado la fuerza e ilusión para continuar y alcanzar mis logros.

- ... a mis padres, por todo su esfuerzo y dedicación. Por creer en mi y siempre colmarme de su cariño.

- ... a Fernando mi hermano, amigo y mentor; gracias por tu cuidado y ayudarme a levantarme siempre que lo necesité. Mucho más aún, por tu amistad, tu consejo y tu ejemplo.

- ... a Sergio González, por tu inigualable compañía y fraternidad durante las incontables horas que compartimos: quemando *rocket-candy* y pretender mandar a *Rayo McQueen* a la luna, jugando videojuegos, conversando de temas de programación y demás. Tu compartir hace ameno cualquier día de escuela o trabajo, además de ser una fuente de inspiración para siempre seguir aprendiendo algo nuevo e interesante.

- ... a Erika Burgos, por tu amistad y sincero cariño a lo largo de estos años, a pesar de mi distancia. Gracias por el sinfín de carcajadas que siempre contagias y por estar a mi lado cuando más lo necesitaba.

- ... a Cesar Poot, por tu amistad y sabiduría. Aunque fue efímera nuestra colaboración, tus palabras y consejos impactaron mi camino; gracias por inspirarme a enfrentar nuevos retos.

- ... al Dr. Joaquín del Bosque, por su ayuda y consejo. Sus sabias y humanas palabras me ofrecieron perspectiva para encontrar mi camino.

- ... a la Dra. Katya Rodríguez Vázquez, por su apoyo, comprensión e infinita paciencia durante el desarrollo de este trabajo.

- ... al Dr. José Galaviz Casas, por lo que me permitió aprender durante la carrera mediante los cursos que él impartió; particulamente en el Seminario de Programación, en el cual con el apoyo de Maximiliano Monterrubio se creó un espacio de colaboración y preparación sin igual.

- ... a todos los miembros del jurado: Dra. Katya Rodríguez Vázquez, Dr. José David Flores Peñaloza, Dr. Miguel la Cruz Salas, José Galaviz Casas y M. en C. Manuel Alcántara Juárez; por sus apreciados comentarios y sugerencias durante la elaboración de este documento.

ÍNDICE GENERAL

1 INTRODUCCIÓN 1
1.1 ¿Qué es la Inteligencia? . . . 1
1.2 ¿Qué es la Inteligencia Artificial? . . . 4
1.3 Optimización . . . 6
1.4 Motivación . . . 9
1.5 Trabajo Previo . . . 11
1.6 Objetivo . . . 12
2 CÓMPUTO BIO-INSPIRADO 14
2.1 Dificultades con IA Tradicional . . . 14
2.2 Inteligencia Computacional . . . 14
2.3 Cómputo Evolutivo . . . 15
2.3.1 Configuración de Parámetros . . . 17
2.3.2 Algoritmos Genéticos . . . 18
2.3.3 Evolución Diferencial . . . 27
2.4 Inteligencia de Enjambre . . . 30
2.4.1 Cúmulo de Partículas . . . 31
3 CÓMPUTO PARALELO 36
3.1 Arquitecturas de Computadoras . . . 41
3.1.1 Arquitectura de Von Neumann . . . 41
3.1.2 Arquitecturas Modernas . . . 41
3.2 Arquitecturas Paralelas . . . 47
3.2.1 Tipos de Paralelismo . . . 48
3.2.2 Clasificación de arquitecturas paralelas . . . 49
3.2.3 Arquitectura de un *GPU* moderno (NVIDIA) . . . 51
3.3 Lenguajes y Modelos de Programación Paralela . . . 55
3.3.1 CUDA . . . 57
3.3.2 Tipos de Memoria . . . 60
4 DISEÑO E IMPLEMENTACIÓN 63
4.1 Diseño Orientado a Datos (DOD) . . . 63
4.2 Optimización . . . 66
4.2.1 Optimizaciones para CPUs . . . 68
4.2.2 Optimizaciones para GPUs . . . 75
4.3 Implementación . . . 77
4.3.1 Prototipos . . . 77
4.3.2 Drosophila . . . 79
4.3.3 Locusta . . . 80
4.3.4 Profiling . . . 82
5 ANÁLISIS DE RESULTADOS 87
5.1 Rendimiento en el CPU . . . 88
5.1.1 Número de Dimensiones . . . 88

5.1.2 Paralelización en CPUs 92
5.2 Rendimiento en el GPU 98
5.2.1 Número de Dimensiones 98
5.2.2 CPUs vs. GPUs 100
5.3 Conclusiones 104
A FUNCIONES OBJETIVO 107

BIBLIOGRAFÍA 108

ÍNDICE DE FIGURAS

Figura 2.1 Pseudo código - Esquema de un Algoritmo Evolutivo 17
Figura 2.2 Pseudo código - Algoritmo Genético 21
Figura 2.3 Pseudo código - Algoritmo de mutación utilizando distribución de Bates 26
Figura 2.4 Gráficas de densidad de probabilidad e histogramas para distribución de Bates. 27
Figura 2.5 Pseudo código - Algoritmo de Evolución Diferencial 30
Figura 2.6 Pseudo código - Algoritmo de Cúmulo de Partículas 34
Figura 3.1 Procesadores Intel 37
Figura 3.2 Gráfica de tendencias en frecuencia y densidad de los transistores 40
Figura 3.3 Jerarquía de memoria 46
Figura 3.4 Comparación en la arquitectura de un núcleo de *CPU* y un núcleo de *GPU* 52
Figura 3.5 Diagrama de bloque de un SM de micro-arquitectura GP100 54
Figura 3.6 Comparación entre la definición de una función iterativa y la definición de un kernel paralelo . 58
Figura 3.7 Polinomio de direccionamiento dentro de un kernel 59
Figura 3.8 Llamada a un kernel 59
Figura 3.9 Configuración de hilos de ejecución en el GPU 60
Figura 3.10 Accesos de memoria alineados con un cache-line, requiriendo sólo una transacción de 128 bytes 61
Figura 3.11 Accesos de memoria desalineados dentro de L1 y L2, requiriendo dos transacciones de 128 bytes 62
Figura 3.12 Accesos de memoria desalineados dentro de L1, requiriendo 5 transacciones de 32 bytes 62
Figura 4.1 Barrido por renglones (Row Major) 69
Figura 4.2 Barrido por columnas (Column Major) 70
Figura 4.3 *Hotspots* en el Cúmulo de Partículas 83
Figura 4.4 *Hotspots* en el Algoritmo Genético 85
Figura 4.5 *Hotspots* en Evolución Diferencial 86
Figura 5.1 Sandy Bridge - Barrido en dimensiones, 256 islas. 89
Figura 5.2 Haswell - Barrido en dimensiones, 256 islas. . . 90
Figura 5.3 PSO - Speedup 4 vs 8 Hilos - 128 Dimensiones 94

Figura 5.4 GA - Speedup 4 vs 8 Hilos - 128 Dimensiones 95
Figura 5.5 DE - Speedup 4 vs 8 Hilos - 128 Dimensiones . 96
Figura 5.6 Máximo speedup en el *CPU* al comparar el tiempo de ejecución de 4 y 8 hilos vs. un único hilo. 97
Figura 5.7 Geforce GTX 1060 6GB - Barrido en dimensiones, 256 islas. 99
Figura 5.8 PSO - Haswell vs Geforce GTX 1060 6GB - 128 Dimensiones 101
Figura 5.9 GA - Haswell vs Geforce GTX 1060 6GB - 128 Dimensiones 101
Figura 5.10 DE - Haswell vs Geforce GTX 1060 6GB - 128 Dimensiones 102
Figura 5.11 Máximo speedup en el *GPU* al comparar el tiempo de ejecución vs. un único hilo en *CPU*. 103
Figura A.1 Gráfica Función Esférica (con Translación) . . 107
Figura A.2 Gráfica Función de Weierstrass (con Translación y Rotación) 107

ÍNDICE DE CUADROS

Cuadro 3.1 Comparación entre la última generación de arquitecturas paralelas y masivamente paralelas . 37
Cuadro 3.2 Comparación de magnitud en latencias. 46
Cuadro 5.1 Speedup utilizando instrucciones SIMD en el CPU. Configuración: 256-512-128 92
Cuadro 5.2 Tabla de valores - Máximo speedup en el *CPU* al comparar el tiempo de ejecución de 4 y 8 hilos vs. un único hilo. 97
Cuadro 5.3 Tabla de valores - Máximo speedup en el *GPU* al comparar el tiempo de ejecución vs. un único hilo en *CPU*. 103

1

INTRODUCCIÓN

We are survival machines, robot vehicles blindly programmed to preserve the selfish molecules known as genes.

— *Richard Dawkins, 1999*

1.1 ¿QUÉ ES LA INTELIGENCIA?

Denominamos Inteligencia a la capacidad de razonar y resolver problemas. Se pueden observar un sinfín de ejemplos en la naturaleza, en donde un organismo o grupo de organismos biológicos resuelven problemas de una manera en particular, como por ejemplo: encontrar alimento, cazar en conjunto, huir de un depredador, crear un refugio o estructuras, etc.

Frecuentemente se catalogan las distintas especies biológicas encontrando semejanzas antropomórficas de sus comportamientos. Cualquier similitud en comportamiento o en la fisonomía de una especie, es atribuida como un signo de inteligencia; se dice que un insecto es más inteligente que una bacteria, a su vez un reptil es más inteligente que un insecto, un mamífero es más inteligente que el reptil, etc. Pero esta manera de describir la inteligencia y complejidad de otras especies es una mera simplificación jerárquica, que surge al intentar diferenciar a los demás organismos del ser humano. La realidad, es que cada organismo resuelve problemas en dimensiones y contextos diferentes. Estas características han sido codificadas en el genoma de cada organismo como solución al problema de supervivencia y reproducción. Todo organismo que vive hoy en día, sea una bacteria unicelular o un ser humano, son el resultado de más de tres mil millones de años de evolución y adaptación, perteneciendo todos a la última generación de supervivientes del árbol evolutivo, ilustrando el sin número de distintas soluciones que este proceso ha creado [Darwin, 1859].

Analicemos el caso de las hormigas podadoras, un grupo de especies de insectos endémicos al continente americano que muestran características eusociales[1]. Como todas las hormigas, viven en colonias que se pueden describir como super-organismos, por la manera en que todos los individuos de la colonia trabajan de manera colectiva. Estas colonias de hormigas son capaces de crear las

1 La eusocialidad define el nivel más elevado en organización social animal: viven en grupos conformados por varias generaciones y muestran división de labores de forma cooperativa.[Wilson & Holldobler, 2005]

estructuras más grandes y complejas que conocemos en el reino animal, excluyendo al ser humano [Piper, 2007].

El vivir en conjuntos tan numerosos les permite colaborar en las diferentes tareas de la colonia: exploración, recolección y transporte. Ellas realizan estas tareas de manera distribuida con un nivel de organización sorprendente; se pueden observar distintos roles entre los varios tipos de individuos, llegando incluso a tener la división de labores más compleja entre todas las otras especies de hormigas. Cada tipo de hormiga está especializada en alguna clase de interacción en la colonia: recolectoras, excavadoras, jardineras, nodrizas, recolectoras de basura, guardias, soldados y la reina. [Wilson, 1980]

Para poder recolectar alimento, cada hormigas primero realizan una búsqueda en el entorno, y una vez que logra encontrar una fuente de alimento tendrá que comunicarlo de alguna manera al resto de la colonia. La ingeniosa solución involucra un complejo sistema de señales químicas mediante trazas de feromonas que definen una red de caminos. Estos caminos pueden verse afectados por diversos obstáculos, por lo que la red de comunicación es dinámica y está en constante mantenimiento, siendo ajustada por todos los individuos que participan en la exploración. De esta manera, las hormigas logran solucionar problemas de búsqueda y creación de rutas que minimizan las distancias recorridas, un problema que sin lugar a dudas no es ajeno a nuestra realidad.

Otro detalle particular en estas especies de *hormigas podadoras*, es que se especializan en podar — tal como su nombre lo indica — recolectando hojas de plantas y transportándolas hasta su hormiguero. Sin embargo las hormigas no utilizan estas hojas como alimento de forma directa, si no como composta para crecer sus mico-cultivos subterráneos, granjas de hongos de la familia *Lepiotaceae* [Hölldobler & Wilson, 2010]. El hongo no tiene la habilidad de producir esporas, depende de las hormigas para lograr reproducirse. Estas hormigas llevan 60 millones de años explotando esta relación de mutualismo, convirtiéndolas en agricultores precursores a los seres humanos. Gracias a esta fuente prácticamente inagotable de comida, las hormigas pueden continuar alimentando a nuevas generaciones de larvas y así sustentar el gran número de individuos que conforman la colonia.

Si bien las hormigas y los seres humanos somos organismos separados por cientos de millones de años de evolución, podemos encontrar semejanzas en los problemas que enfrentamos, y las estrategias en que estos organismos los resuelven pueden ser fuente de inspiración para ingeniar nuevas soluciones.

Así como las hormigas, existen otras especies como las abejas, termitas e incluso microorganismos tan simples como las amibas, cuyo comportamiento en conjunto muestra capacidad para organizarse y resolver problemas computacionalmente complejos. [Bonner, 2009]

La inteligencia humana es la característica más significativa de nuestra especie, la cual también emerge como una adaptación de nuestros ancestros para luchar y sobrevivir en nuestro entorno natural. En un principio nuestros problemas eran muy parecidos a los de cualquier otra especie: lograr sobrevivir y tener descendencia. Gracias a nuestro intelecto, fuimos encontrando maneras de modificar nuestro modo de vida, crear herramientas y estrategias que nos permitieran ser más productivos y eficientes.

Hasta la fecha, sólo sabemos del *homo sapiens* como la especie que logró crear un lenguaje complejo con representación oral y escrita, posiblemente la invención con más trascendencia en la historia de nuestra especie. Aunque se estima que tenemos aproximadamente 200,000 años, fue hace apenas 5,200 años — a finales del periodo *Neolítico* [2] — que logramos desarrollar una manera eficiente de comunicar ideas; las montañas, mares y el tiempo, ya no serán una limitante. Nunca antes el conocimiento humano podía ser transmitido con tanta facilidad y persistir por los años adelante.

Otro fruto de nuestro intelecto y capacidad de crear soluciones para sobrepasar las limitantes impuestas por la naturaleza, fue el desarrollo de la agricultura. Se sabe que a lo largo de la historia esta estrategia surgió en diferentes momentos y lugares de manera independiente, pero fue la primordial solución al problema de poder contar con una fuente de alimento constante y almacenable, y así evitar las migraciones en busca de ello. A raíz de esto, nacieron los primeros grandes asentamientos y el surgimiento de la civilización sedentaria, dando lugar a la época del *Antropoceno*[3].

El ser humano comenzó a construir sociedades en donde los individuos se especializaron en distintas actividades: el guerrero cuidaba y defendía, el agricultor cultivaba la comida, el maestro enseñaba y el filósofo podría dedicarse a pensar. La humanidad ahora tenía la capacidad tecnológica para controlar y transformar su entorno, y el lujo de dedicar tiempo a pensar en cómo solucionar nuevos problemas que nunca antes alguien se había cuestionado. Así

2 En el estudio Antropológico, se le conoce al Neolítico como la última etapa de la edad de piedra, abarcando del 10,200 al 2,000 AC

3 El Antropoceno es una época geológica propuesta para identificar el momento en el que la actividad humana comenzó a tener un impacto significativo sobre la geología de la tierra y los ecosistemas. En base a evidencia geológica esto pudo haber comenzado desde hace 15,000 años [Doughty et al., 2010]

nace el estudio formal de las ciencias, el ser humano se cuestionará su propia identidad y eventualmente la naturaleza de su propia inteligencia. ¿Qué es la inteligencia humana? ¿Existe otro tipo de inteligencia? ¿Podemos crear una entidad con inteligencia?

Preguntas muy simples de realizar son sumamente complejas de responder, y apenas en las últimas décadas de nuestra historia hemos sido capaces de crear las herramientas que nos permiten estudiar el tema y posiblemente contestar algunas de estas preguntas.

1.2 ¿QUÉ ES LA INTELIGENCIA ARTIFICIAL?

El estudio formal de la Inteligencia Artificial, nace junto con las Ciencias de la Computación y de cierta manera con la invención de las primeras computadoras. Esta historia comienza a mediados del siglo XVIII, al mismo tiempo que estaba por iniciar la guerra de independencia de los Estados Unidos de Norteamérica, con un ingeniero alemán llamado Johann Helfrich von Müller. Müller pertenecía a los mercenarios de Hesse — soldados alemanes que trabajaban para el gobierno británico. Durante esta época, era necesario realizar todos los cálculos aritméticos utilizando a personas como medio de cómputo. Naturalmente esto era lento y propenso a resultados con errores. Para poder agilizar este proceso y generar cálculos libres de errores humanos, Helfrich publica en 1786 el concepto de una máquina diferencial, con la cual podrían automatizar la generación de tablas para cualquier función polinomial. Sin embargo, Helfrich nunca logra obtener fondos para construir la máquina, fue treinta y seis años más tarde cuando Charles Babbage — un matemático, filósofo e inventor — propone utilizar el concepto de Helfrich para crear una versión más eficiente y poderosa. Esta máquina era la versión mecánica de una computadora electrónica moderna, contaba con: una unidad aritmética lógica (ALU), memoria y unidad de control de saltos. Fue el primer concepto en la historia de una máquina programable Turing-completa. A pesar de que Babbage sí logró recibir fondos para realizar su máquina analítica, nunca logró tener una versión funcional de esta. [Lindgren, 1990]

Al inicio del siglo XX, varios matemáticos alrededor del mundo trabajaban en crear una fundación sólida para las matemáticas, buscando definir un sistema completo y consistente. Sin embargo, todos se enfrentaron a una serie de dificultades llegado siempre a una serie de paradojas y contradicciones. A este periodo se le conoce como *la crisis de la fundación de las matemáticas* [Robic, 2015]. Esta crisis culmina con el trabajo de Kurt Gödel en 1931, con sus

teoremas de incompletitud, al demostrar que un sistema lógico — con cierta estructura y complejidad mínima — no puede ser consistente y completo [Nagel & Newman, 2001]. En el área de las matemáticas esto fue un callejón sin salida para gran parte de la investigación que se realizaba en la época. Sin embargo, todo el trabajo realizado creó nuevas herramientas y métodos matemáticos muy ingeniosos. Particularmente nos interesa el trabajo de Alonzo Church y su alumno Alan Turing, el cual será el cimiento de la teoría de la computabilidad y complejidad: conceptos fundamentales en las Ciencias de la Computación. Entonces, comienza un esfuerzo de manera simultánea alrededor del mundo para crear máquinas que nos permitan procesar información.

En 1936, Turing define el concepto de una *máquina universal* [Turing, 1936], la cual está compuesta de: una cinta dividida en celdas, un conjunto de símbolos que pueden existir en cada celda de la cinta, una cabeza con la capacidad de leer y reescribir estos símbolos en la cinta, un registro que define el estado actual de la máquina y una tabla de instrucciones que describen las operaciones y transiciones entre estados. A pesar de ser un concepto de una máquina sumamente simple, Turing demostró que tal máquina era capaz de ejecutar cualquier función computable que se codificara en la cinta. A cualquier máquina con una capacidad de cómputo análoga a la máquina universal de Turing se le denomina Turing-Completa.

Previo a la segunda guerra mundial, el ingeniero Konrad Zuse había estado trabajando en el concepto de una máquina electromecánica, capaz de realizar cómputo de punto flotante, la *Z1*. Konrad continuaría sus esfuerzos en los próximos años construyendo nuevas versiones de su concepto: comenzando con la *Z2* y por último la *Z3*, la cual se convertiría en la primera implementación física en la historia de una máquina Turing-Completa.

Una vez que inicia la segunda guerra mundial, nacen nuevas necesidades para crear herramientas que agilicen el cómputo de información. Konrad, financiado por el gobierno alemán, trabajó en el desarrollo de dos nuevas computadoras, la *S1* y *S2*, las cuales fueron utilizadas para crear modelos computacionales y que permitan optimizar la estructura de las primeras bombas dirigidas y misiles aéreos. Por otra parte, Turing trabajó en una serie de conceptos de computadoras: *Bomba* y posteriormente *Enigma*, con la intención de descifrar los códigos criptográficos Alemanes y con ello lograr interceptar sus mensajes.

Al final de la guerra, Konrad Zuse desarrolló la siguiente iteración de sus computadoras, la *Z4*, y creó el primer lenguaje de alto nivel

utilizado para programar una computadora, *Plankalkül* [Knuth et al., 1976]. Turing por su parte continuó el estudio relacionado con la computabilidad de las máquinas, aportando importantes fundamentos que dieron inicio a los estudios de la Inteligencia Artificial.

Si ahora contábamos con máquinas capaces de computar cualquier algoritmo, ¿Podríamos entonces programar inteligencia y aplicarla para resolver problemas?

En la biología, la objetivo del cerebro en un animal no es reproductiva, sino de proveer a su dueño con conocimiento y habilidades para colocarse en circunstancias que promuevan su reproducción. En el caso particular del homo sapiens, una especie social, el entender y comprender su entorno y asegurar la cooperación con los demás miembros en su comunidad. [Pinker, 1997]

Un modelo o algoritmo de Inteligencia Artificial es la técnica utilizada para solucionar alguna clase de problemas. En cada modelo, el programador tendrá que definir ciertos objetivos para que el sistema logre producir una buena solución al problema que se está intentando resolver. El propósito de la Inteligencia Artificial es crear un sistema computacional inteligente que pueda obtener soluciones de manera similar a como lo hacemos los seres humanos. [Heaton, 2013] Sin embargo esto no significa que el programa tenga que pretender ser biológico, ni emular el funcionamiento del cerebro humano.

Las técnicas que se exploran en este trabajo pertenecen a una rama de la Inteligencia Artificial, denominada Inteligencia Computacional. La definición moderna de Inteligencia Computacional está fuertemente influenciada por modelos biológicos del aprendizaje de máquinas. [Konar, 2005]

1.3 OPTIMIZACIÓN

Dado un modelo matemático de cierto proceso, podemos definir una función $f_0(x)$, la cual representa el costo o utilidad de utilizar una opción x. A esta función se le denomina *función objetivo*.

Desde problemas de diseño en ingeniería química, mecánica, aeroespacial, tanto en sistemas de automatización y toma de decisiones, podemos crear modelos matemáticos con los cuales es posible utilizar métodos de optimización para encontrar soluciones más eficientes.

Podemos ejemplificar con una aplicación del mundo real en la industria del diseño de circuitos integrados, en donde se desea resolver el problema de organización de los componentes electrónicos modificando variables como el tamaño y posición de cada uno de ellos, tomando en cuenta que las variables están limitadas por la eficiencia de los procesos de manufactura, así como tiempos y costos de producción. En este caso se desea:

- Minimizar el consumo energético del circuito
- Maximizar la frecuencia de reloj

Resulta sencillo plantear un problema como el anterior, pero el encontrar una solución óptima puede ser sumamente complejo. Por ejemplo, si las variables del problema son discretas entonces el número de posibles combinaciones se vuelve inmenso, al punto que puede resultar imposible encontrar la solución óptima bajo las restricciones de tiempo y cómputo que existen.

Un problema de optimización se puede definir de la siguiente forma:

$$\begin{aligned} &\text{Minimizar } f_0(\vec{x}) \\ &\quad \text{Sujeto a } f_i(\vec{x}) \leq b_i, i = 1, ..., m \end{aligned} \tag{1.1}$$

Donde el vector $\vec{x}$ es la *variable de optimización* al problema, descrito por la función objetivo $f_0 : \mathbb{R} \to \mathbb{R}$.

Las constantes $b_1, ..., b_m$ son los límites que tienen que satisfacer el conjunto de funciones de restricciones: $f_i : \mathbb{R} \to \mathbb{R}$, $i = 1, ..., m$.

Llamaremos al vector $\vec{x^\star}$ la solución óptima de minimización al problema (1.1) si tiene el valor objetivo más pequeño entre todos los vectores que satisfacen las restricciones, es decir:

$$\begin{aligned} \forall \vec{z} \text{ con } & f_1(\vec{z}) \leq b_1, ..., f_m(\vec{z}) \leq b_m \\ & f_0(\vec{z}) \geq f_0(\vec{x^\star}) \end{aligned} \tag{1.2}$$

Se puede generalizar esta definición para problemas de maximización, con sólo invertir el signo de la función f_0.

$$f_0(\vec{z}) \geq f_0(\vec{x^\star}) \quad \Rightarrow \quad -f_0(\vec{z}) \leq -f_0(\vec{x^\star}) \tag{1.3}$$

No existe una manera única de clasificar una función objetivo, pero existen ciertos atributos que determinan los métodos que se pueden utilizar, así como el rendimiento que podemos esperar de cada método. Algunos de estos atributos son:

- *Cuantización de los parámetros*: ¿Qué tipo de variables utiliza la función objetivo? ¿Son continuas o discretas?
- *Diferenciable*: ¿La función objetivo es derivable en todos los puntos del dominio que se desea optimizar?
- *Dependencia entre parámetros*: ¿La función objetivo es separable? Esto quiere decir si las variables se pueden optimizar de manera independiente.
- *Dependencia temporal*: ¿El valor óptimo de la función es estático o dinámico?
- *Dimensionalidad*: ¿De cuántas variables define la función objetivo?
- *Modalidad*: La función objetivo tiene un óptimo local (unimodal) o varios óptimos locales (multimodal).
- *Ruido*: ¿La evaluación de la función es el mismo cada vez que se evalúa o fluctúa?
- *Optimización vectorial*: Se optimiza una sola función objetivo o son múltiples funciones las que se desean optimizar.

Utilizando la definición (1.1) llamaremos a un problema un *programa lineal*, si la función objetivo satisface la siguiente ecuación:

$$\begin{aligned} &f_0(\alpha x + \beta y) = \alpha f_0(x) + \beta f_0(y) \\ &\quad \forall x, y \in \mathbb{R} \text{ y } \alpha, \beta \in \mathbb{R} \end{aligned} \tag{1.4}$$

En caso de no satisfacer esta condición llamamos al problema un *programa no lineal*.

Existen muchos problemas cuya complejidad de encontrar la solución óptima es tan costosa que resulta inviable y no hay razón que justifique la inversión de tiempo y recursos. Pero, ¿realmente es necesario encontrar el valor óptimo en el codominio de la función? ¿Existirán soluciones suficientemente cercanas a lo que buscamos y que se puedan encontrar en el marco de tiempo disponible que justifiquen atacar el problema?

A pesar de que existen muchos métodos de optimización, los problemas más complejos que enfrentamos implican restricciones que limitan las herramientas que podemos utilizar para resolverlos.

Contamos con otra categoría de métodos de optimización llamados *heurísticas*, los cuales pueden ser utilizados de manera aislada o en conjunto con los métodos de optimización tradicionales. El implementar heurísticas es análogo a utilizar atajos o decisiones informadas. Conocer la naturaleza de un problema nos puede ayudar a reducir considerablemente el tiempo de cómputo o descartar secciones del espacio de búsqueda, en donde podemos suponer que existen o no soluciones de nuestro interés. Una heurística puede definir un criterio con el cual:

- Se toman decisiones con base en datos observados.
- Introducir elementos estocásticos en el algoritmo de búsqueda.
- Decidir en qué momento terminar el proceso de búsqueda.

Una meta-heurística es un proceso iterativo generacional, el cual guía una heurística subordinada al combinar de manera inteligente conceptos para la exploración y explotación del espacio de búsqueda [Osman & Laporte, 1996]. Las meta-heurísticas no son específicas a un problema. Al final de la década de los ochenta renació el interés por la investigación de meta-heurísticas inspiradas en procesos naturales, particularmente se obtuvo gran atención en el área de cómputo evolutivo. Los *algoritmos evolutivos* son meta-heurísticas que pueden aprender del conocimiento colectivo, y resultan ser una buena estrategia para resolver problemas de optimización complejos, donde utilizar métodos tradicionales no es factible.

1.4 MOTIVACIÓN

Muchos de los problemas de optimización en ciencias e ingeniería presentan características que limitan las opciones sobre métodos de optimización que podemos utilizar. Para atacar un problema de optimización, primero se tiene que definir una función objetivo que describa cuál es el valor que se desea maximizar o minimizar, en función de las variables presentes. Posteriormente, se van a identificar las restricciones sobre los valores de cada una de las variables, así como las relaciones que pueden existir entre estas variables y el dominio de la función objetivo.

Cada vez que se adapta una implementación de un método de optimización computacional a un nuevo problema, se tendrá que

definir una representación sobre la cuál se va a operar. La mayoría de los problemas que se ven en la industria tratan de optimizar modelos que ya han sido probados y atacados utilizando métodos tradicionales de optimización. El volver a implementar o migrar el problema a una nueva representación puede ser una opción no viable en la mayoría de los casos. Existen clases de meta-heurísticas que requieren realizar el cambio a una representación alternativa, como una representación binaria en el caso de los Algoritmos Genéticos, lo cual a su vez introduce nuevos obstáculos al problema de optimización: cardinalidad limitada, costo computacional de codificación y decodificación, así como otros problemas que son inherentes a este tipo de representación. Esto resulta especialmente importante cuando se quiere incorporar una meta-heurística como método de optimización híbrida a una infraestructura de optimización ya existente, en donde ya hubo una inversión en implementar una solución específica al dominio del problema.

A pesar de mostrar resultados interesantes en contraste con los métodos de optimización basados en gradientes[4], las meta-heurísticas de cómputo evolutivo demandan un alto nivel de poder computacional para poder realizar las operaciones de evaluación sobre toda la población en cada generación, lo que limita la practicidad de sus implementaciones así como las configuraciones que se pueden ejecutar.[Morse & Stanley, 2016]

En la última década, se ha observado un estancamiento en la velocidad de los microprocesadores, limitando el rendimiento de todas las implementaciones que habían sido diseñadas para ser ejecutadas de manera serial, y por ende restringiendo las herramientas a nuestra disposición para explorar y lograr resolver problemas cada vez más complejos.

Desde el punto de vista de consumo energético, también resulta más eficiente el utilizar múltiples procesadores de manera distribuida, que un solo procesador funcionando a una velocidad equivalente a la suma de varios procesadores (véase 3). Esta estrategia resulta importante para varios casos de uso actuales: granjas de cómputo conformadas por decenas de miles de nodos, así como en dispositivos operados por baterías o fuentes limitadas de energía. Por estas razones, ahora más que nunca resulta crucial el desarrollar nuevos algoritmos paralelos e implementaciones escalables que nos permitan utilizar en su totalidad y de manera eficiente los recursos existentes.

Las meta-heurísticas que estudiaremos son inherentemente paralelizables en varios niveles: individuos, poblaciones y operadores. Aún más relevante es la paralelización al evaluar la función objetivo

4 Gradiente de descenso estocástico

sobre cada individuo de una población, comúnmente siendo la sección más costosa en tiempo de cómputo. Dado que en cada paso de estas meta-heurísticas las operaciones de transformación sobre la población puede suceder de manera independiente, el paralelismo podrá ser explotado en diferentes granularidades. Esto nos permite crear implementaciones con gran eficiencia, dirigidas a supercomputadoras y procesadores gráficos de propósito general, *GPGPUs*[5].

Tradicionalmente se han realizado implementaciones sobre *clusters* de supercómputo, los cuales son sumamente caros de operar y su disponibilidad es muy limitada. Con los avances en *GPGPUs* es ahora posible implementar algoritmos masivamente paralelos en una computadora personal. Estas implementaciones a su vez pueden escalar en múltiples dimensiones: con múltiples *GPU*s en un mismo nodo, ejecutando en un *cluster* de múltiples nodos con *GPU*s.

1.5 TRABAJO PREVIO

En un trabajo de investigación previo [González del Cueto, 2011], se desarrolló una implementación de un algoritmo genético con la intención de estudiar los beneficios de utilizar arquitecturas masivamente paralelas para efectuar la evaluación de la función objetivo. El proceso involucró estudiar la arquitectura de los nuevos coprocesadores gráficos, utilizando estrategias específicas a las características de estos procesadores al portar un Algoritmo Genético a este nuevo modelo de cómputo. Posteriormente, se portó el código para que pudiera ser ejecutado de manera paralela en varios hilos sobre núcleos de *CPU*s, manteniendo el código y los pasos implementados lo más parecido posible entre arquitecturas.

Los resultados que se obtuvieron fueron positivos, particularmente en configuraciones en donde se utilizaban múltiples conjuntos poblacionales cada uno con un tamaño mayor a 32 individuos, donde se pudieron observar incrementos en rendimiento desde $14x$ hasta $887x$ sobre los tiempos de ejecución serial en el *CPU*.

Aunque el objetivo de este estudio era poder realizar una comparación de rendimiento sobre los mismos operadores, intentando mantener la misma secuencia de instrucciones de cómputo, esta táctica comparativa sufre de un sesgo importante, ya que las dos arquitecturas en el estudio son fundamentalmente

5 General-purpose computing on Graphics Processing Units

diferentes y el realizar optimizaciones para una de ellas tendrá un impacto negativo en la otra.

En otros trabajos se han implementado soluciones en *GPU*s reportando factores de aceleración de magnitud similar al compararlas con el rendimiento de *GAlib*, la cual es una biblioteca diseñada para ser flexible y no particularmente de alto rendimiento. [Pospichal & Jaros, 2009]

Para realizar una comparación significativa sobre el rendimiento de las meta-heurísticas es importante contar con una implementación cuyo enfoque también sea proporcionar alto-rendimiento y así poder entender los costos y beneficios de cada implementación. [Jaros & Pospichal, 2012]

Por la naturaleza de la arquitectura de un *GPU*, las implementaciones tienen más restricciones sobre la clase de operaciones y datos que pueden efectuar de manera eficiente. Es importante tomar esto en cuenta para entender qué tipo de aplicaciones pueden tener beneficios y qué limitaciones van a existir al ser portados a estas arquitecturas.

En los últimos años las arquitecturas de *GPU* han madurado, y con ello las herramientas que proveen para programarlos. Implementaciones realizadas anteriormente aún utilizaban gran parte de la infraestructura de gráficas para realizar cómputo de propósito general. Herramientas dirigidas a las primeras generaciones de *GPU*s, junto con el *hardware* ahora ya son obsoletas [Debattisti et al., 2009].

1.6 OBJETIVO

El propósito principal de esta tesis es realizar un estudio comparativo sobre los beneficios en rendimiento al paralelizar un conjunto de meta-heurísticas basadas en poblaciones, dirigido a dos arquitecturas paralelas: *CPU* multinúcleo y *GPUs*.

Para ello se desarrolló la implementación de *Locusta*[6] [González del Cueto, 2014], una biblioteca de programación de código abierto; escrita en *C++*, utilizando *OpenMP* y *CUDA C*; publicada bajo la *Licencia Pública Reducida de GNU*. *Locusta* provee una infraestructura para asistir en el desarrollo de implementaciones de meta-heurísticas basadas en poblaciones, que pueden ser ejecutadas en *CPU*s y *GPU*s *Nvidia*. La biblioteca

6 https://github.com/rdelcueto/Locusta

incluye implementaciones canónicas de tres meta-heurísticas: *Algoritmo Genético*, *Cúmulo de Partículas* y *Evolución Diferencial*.

En este trabajo se estudian las dos arquitecturas paralelas, describiendo sus características y diferencias más significativas. Utilizando *Locusta* como plataforma de desarrollo, se exploran estrategias de optimización específicas a cada una de las arquitecturas, permitiendo así explotar los recursos de cómputo de la manera más eficiente posible y documentando los principales obstáculos encontrados durante el desarrollo.

2

CÓMPUTO BIO-INSPIRADO

Variation is the raw material of evolutionary change. It represents the fundamental reality of nature, not an accident about a created norm. Variation is primary; essences are illusory.

— *Stephen Jay Gould, 1987*

2.1 DIFICULTADES CON IA TRADICIONAL

En los métodos tradicionales de Inteligencia Artificial, dado un problema se define una máquina de estados; conjunto de posibles estados y reglas de transición. A cada estado se le asigna un símbolo y se etiqueta como: inicial, transitorio o final. El programa comienza en alguno de los estados iniciales, y en cada iteración siguiendo las reglas de transición se decide cuál es el nuevo estado; a este conjunto de reglas se le denomina la base de conocimiento del programa. El programa termina cuando se alcanza un estado final.

Estos métodos tradicionales resultan ser muy efectivos para aprendizaje de tipo inductivo. Sin embargo, entre más reglas se requieran definir en el conjunto de conocimiento, cada transición involucra una búsqueda más extensa. Con problemas suficientemente complejos esto se convierte en una limitante. Con la excepción a los algoritmos de búsqueda heurística, la familia de métodos tradicionales de Inteligencia Artificial no resultan ser competentes para atacar problemas de optimización. [Konar, 2005]

2.2 INTELIGENCIA COMPUTACIONAL

Ante la poca efectividad de los métodos tradicionales de Inteligencia Artificial y la gran necesidad de crear nuevas herramientas de optimización, surgieron nuevas ramas de estudio, entre ellas la Inteligencia Computacional.

La Inteligencia Artificial clásica, se basa en utilizar métodos simbólicos para el razonamiento y requiere de una base de conocimiento proveniente de seres humanos. En contraparte, los métodos en Inteligencia Computacional son capaces de aprender de los datos que consume y crear nuevo conocimiento el cuál puede ser post-procesado e interpretado.

Un sistema computacionalmente inteligente, según la definición de James Bezdek [Konar, 2005], deberá tener las siguientes características:

- Información numérica de bajo nivel: El sistema debe de tener un componente capaz de reconocer patrones.
- Capacidad de adaptación: El sistema deberá poder cambiar los criterios de optimización en función de cambios en los datos de entrada y salida.
- Tolerancia a errores: Una característica importante de sistemas distribuidos, donde los recursos de cómputo tienen redundancia, y el fallo de una instancia no implica que todo el sistema deje de funcionar.
- Alto rendimiento con alta precisión: El sistema tiene que poder producir respuestas rápidamente sin sacrificar precisión en los resultados, además de reflejar adaptación inmediata a cambios en los datos de entrada.

Llamaremos Inteligencia Computacional al sistema e infraestructura de métodos que magnifican la inteligencia humana mediante el aprendizaje y descubrimiento de nuevos patrones, relaciones o estructuras, en complejos contextos dinámicos.

En años recientes se han innovado gran variedad de métodos computacionalmente inteligentes inspirados en imitar ciertos aspectos que observamos en la inteligencia de organismos biológicos. Entre ellos: las redes neuronales, sistemas de lógica difusa y los algoritmos evolutivos.

2.3 CÓMPUTO EVOLUTIVO

Una de las formas en que se puede implementar Inteligencia Computacional, es utilizando una familia de algoritmos inspirados en el proceso biológico de evolución mediante selección natural. La esencia del proceso evolutivo, es que los rasgos y características que permiten a cada individuo sobrevivir y reproducirse tienen más probabilidades de trascender a nuevas generaciones. De manera opuesta, características que obstaculizan la supervivencia tienden a desaparecer. Todas las características de un individuo están codificadas en su genotipo. Mediante la reproducción sexual, diferentes genotipos son recombinados para generar el material genético de la nueva generación de individuos, heredando las características de sus padres y al mismo tiempo pero de manera esporádica, introduciendo variaciones mediante eventos de mutación.

En el Cómputo Evolutivo se utilizan operaciones análogas a este proceso biológico. Se define un individuo como un grupo de variables

que codifica una solución, analogía al genotipo de un ser vivo. A estas soluciones les asignamos un valor de aptitud, al evaluar las variables que codifica sobre la función que deseamos optimizar. Utilizando estos valores de aptitud, se realiza un proceso de selección con un sesgo por las soluciones de mayor aptitud, de manera que estas tengan mayor probabilidad de trascender sus genes a las futuras generaciones, análogo a la presión selectiva que existe en la evolución mediante selección natural. Los operadores de cruza y mutación simulan la reproducción sexual, realizando una transformación que recombina y perturba la información genética de los individuos en una población. En cada generación se crea un nuevo conjunto de soluciones candidato, explotando la información de la generación anterior, aunado a los cambios aleatorios introducidos por las mutaciones genéticas que ocurren durante el proceso de recombinación genética. [Michalewicz, 1999]

Denotemos el conjunto poblacional en un instante de tiempo g, como P_g. Comenzando en la primera generación $g = 0$, se requiere inicializar a los individuos de la población. Una vez determinada la población inicial, se itera sobre las siguientes operaciones hasta alcanzar un criterio de terminación o el límite en el número de generaciones permitidas.

1. Se evalúa la aptitud del conjunto poblacional A_g.

2. Se realiza la selección de individuos cuya información genética se recombina, esto basado en el valor de aptitud obtenido previamente. Individuos con mayor aptitud tienen mayor probabilidad de ser seleccionados. S_g

3. Una vez definido el conjunto de recombinación se procede a aplicar la función de transformación sobre sus variables, obteniendo la nueva generación de individuos.

4. Incremento del contador de generaciones g.

En la Figura 2.1 se muestra el pseudo-código de un algoritmo evolutivo.

Un aspecto interesante de la familia de algoritmos basados en evolución y conjuntos poblacionales, es que las transformaciones y cálculos requeridos para cada individuo son independientes a los demás, permitiendo desarrollar implementaciones paralelas de las meta-heurísticas. Bajo esta perspectiva, los algoritmos evolutivos han sido un ámbito de gran interés desde el surgimiento de las computadoras masivamente paralelas. Tomando en cuenta las tendencias de la industria en la actualidad en crear nuevas arquitecturas de microprocesadores con múltiples núcleos, se ha

```
inicio
    g ⟵ 0
    P_g ← Inicializa(P_g)
    A_g ← Evalua(P_g)

    while no se cumpla el criterio de terminación do
        S_g ← Selecciona(P_g, A_g)
        P_{g+1} ← Transforma(P_g, S_g)
        A_{g+1} ← Evalua(P_{g+1})
        g ⟵ g + 1
    end
fin
```

Figura 2.1: Pseudo código - Esquema de un Algoritmo Evolutivo

encendido nuevamente el interés por explorar algoritmos evolutivos como métodos de optimización.

2.3.1 *Configuración de Parámetros*

Un algoritmo evolutivo define un conjunto poblacional inicial, el cual puede ser generado aleatoriamente o estar basado en conocimiento a *priori*, como información del espacio de búsqueda y soluciones al problema ya conocidas. En general para todo algoritmo evolutivo el tamaño del conjunto poblacional será uno de los parámetros más importantes y críticos por definir [Hu et al., 2010]. Utilizar poblaciones muy numerosas tendrá como consecuencia el requerir almacenar y computar más datos, lo cual reduce el rendimiento de la evolución simulada. En el caso inverso, el utilizar poblaciones muy pequeñas limita la diversidad en la población, reduciendo la capacidad de explotar y explorar, haciendo que el algoritmo sea propenso a converger prematuramente, arrojando soluciones de baja calidad. [1]

Una estrategia que se puede implementar de manera general en los algoritmos evolutivos basados en poblaciones, es extender el uso de un único conjunto poblacional a múltiples instancias, denominadas *islas*. Esta estrategia nos permite ampliar el tamaño de la búsqueda, fomentando la evolución independiente de cada isla y al mismo tiempo previniendo la monopolización de ciertas soluciones. Encima del concepto de islas se puede implementar el concepto de

1 En este trabajo se estudia el efecto en los tiempos de ejecución al modificar la configuración del tamaño poblacional en cada meta-heurística implementada.

migraciones, el cual habilita la colaboración de información entre los distintos conjuntos poblacionales de manera controlada. Esta estrategia provee de un parámetro que permite controlar indirectamente la diversidad genética entre islas.

Al utilizar un algoritmo evolutivo para optimizar una función, habrá que definir la configuración de los parámetros de la meta-heurística, comenzando con el tamaño poblacional y el número de islas. En cada algoritmo evolutivo existirán diferentes operadores de transformación y recombinación, para los cuales, como veremos más adelante, también habrá que ajustar parámetros que controlen el balance entre explotación y exploración.

Es importante subrayar que dependiendo de la función que se desea optimizar, estos valores tendrán efectos diferentes sobre el rendimiento de la meta-heurística. No existe una configuración universal que resulte eficiente y efectiva para cualquier problema. La selección de parámetros es en sí un problema de optimización complejo, dado que la interacción entre estos valores suele ser no lineal y no es posible optimizarlos de manera aislada. Para las meta-heurísticas mencionadas en este trabajo, la selección de los parámetros y sus operadores para atacar diferentes aplicaciones, son problemas abiertos para la investigación. A este proceso de optimización sobre los parámetros de otro método de optimización se le conoce como *meta-optimización*.

2.3.2 *Algoritmos Genéticos*

Un algoritmo genético (*GA*, del inglés: *Genetic Algorithm*) es una meta-heurística que modela el proceso evolutivo de las especies biológicas, donde información codificada en los individuos de la población se recombina utilizando el concepto de selección natural y transformaciones genéticas mediante los operadores de recombinación y mutación, obteniendo nuevas soluciones candidato en cada generación. [Mitchell, 1996]

En la década de 1970, John Holland, un profesor de la Universidad de Michigan, trabajaba junto con sus colegas estudiando fenómenos de adaptación en la naturaleza y cómo desarrollar mecanismos en sistemas computacionales que tuvieran el mismo comportamiento. En 1975, John Holland presentó en su libro, *Adaptation in Natural and Artificial Systems*, el concepto de un algoritmo genético como una abstracción de la evolución biológica. [Holland, 1975]

Definición 1. *Definamos P_g como el conjunto poblacional de N cromosomas en la generación g, a lo largo de $g_{max}-1$ generaciones.*

$$P_g = (\vec{x}_{i,g}),\ i = 0, 1, ..., N-1,\ \ g = 0, ..., g_{max}-1 \tag{2.1}$$

Definición 2. *Cada cromosoma en la población está representado por un vector $\vec{x}_{i,g}$ de dimensión D.*

$$\vec{x}_{i,g} = (\vec{x}_{i,j,g}),\ j = 0, 1, ..., D-1. \tag{2.2}$$

En la primera generación $g = 0$, se inicializan cada uno de los cromosomas asignando valores aleatorios uniformemente distribuidos dentro del intervalo que define el dominio de búsqueda.

$$\begin{aligned}\vec{x}_{i,j,0} = {} & rand_j[0,1) \cdot (limite_{j,superior} - limite_{j,inferior}) \\ & + limite_{j,inferior}\end{aligned}$$

En cada ciclo del algoritmo genético se produce una nueva generación de posibles soluciones. Cada individuo de la población codifica un cromosoma, compuestos a su vez por una serie de genes. El tipo de codificación o representación que se utilice jugará un papel importante en la definición del algoritmo genético.

En cada nueva generación se aplica el proceso de evaluación utilizando la función objetivo, la cual debe reflejar las restricciones del problema que se desea optimizar, asignando un valor de aptitud o índice de la calidad del prospecto de solución.

Definición 3. *Definimos al vector A_g de tamaño N, que contiene los valores de aptitud para cada individuo en la población, al evaluar la función objetivo.*

$$\begin{aligned}\vec{A}_g &= (\vec{A}_{i,g}),\ \ i = 0, 1, ..., N-1,\ \ g = 0, ..., g_{max}-1 \\ \vec{A}_{i,g} &= Evalua(\vec{x}_{i,g})\end{aligned} \tag{2.3}$$

Al contar con la evaluación de aptitud sobre cada individuo de la población, el siguiente paso en el algoritmo genético será realizar un proceso de selección sobre la población, definiendo cuales serán los cromosomas que participarán en el proceso de recombinación genética para generar el conjunto poblacional de la siguiente generación. La función principal del operador de selección es introducir presión selectiva sobre la población, al inducir un sesgo en la selección por los genomas que hayan obtenido un mayor índice de aptitud.

Definición 4. *Definimos el vector S_g de tamaño N, el cual codifica la asignación de parejas para cada uno de los cromosomas que serán recombinados. La función de selección asignará a cada $\vec{x}_{i,g}$ una pareja $\vec{S}_{i,g}$.*

$$\vec{S}_g = (\vec{S}_{i,g}),\ \ i = 0, 1, ..., N-1,\ \ g = 0, ..., g_{max} - 1 \tag{2.4}$$
$$\vec{S}_{i,g} = Selección_i(P_g, A_g)$$

Una vez seleccionados los cromosomas que participarán en la reproducción, se realiza la manipulación de la población, utilizando dos operadores: recombinación y mutación.

Definición 5. *La población de la nueva generación P_{g+1} se crea al transformar el conjunto anterior mediante los operadores de recombinación y mutación.*

$$P_{g+1} = Mutacion(Recombinacion(P_g)) \tag{2.5}$$

El objetivo del operador de recombinación será utilizar la información contenida en los genomas seleccionados para ser recombinados. Dependiendo de la representación que se esté utilizando en el algoritmo genético se pueden definir diferentes operadores de recombinación.

La última transformación ocurre sobre el cromosoma resultante de la recombinación de sus padres, llamado *operador de mutación.* Esta transformación va a alterar los genes del nuevo cromosoma, introduciendo nuevas variaciones que posiblemente no existían anteriormente en el conjunto poblacional.

En la Figura 2.2 se muestra el pseudo-código de un algoritmo genético canónico:

2.3.2.1 *El Algoritmo Genético Real o Continuo*

En los Algoritmos Genéticos como en cualquier método de aprendizaje y búsqueda, la selección de una representación adecuada será de primordial importancia para lograr encontrar buenas soluciones. En base a la codificación seleccionada también se definen los operadores de recombinación y mutación. Originalmente, Holland ideó el algoritmo utilizando una codificación binaria, construyendo los genomas como cadenas de bits con un tamaño fijo. Sin embargo, se pueden utilizar otros tipos de representaciones:

- Cadenas de múltiples caracteres, utilizados comúnmente para describir árboles de sintaxis abstracta en programación genética.

```
inicio
    g ⟵ 0
    P_g ← Inicializa(P_g)
    A_g ← Evalua(P_g)

    while no se cumpla el criterio de terminación do
        S_g ← Seleccion(P_g, A_g)
        P'_g ← Recombinación(P_g, S_g)
        P_{g+1} ← Mutacion(P'_g)
        A_{g+1} ← Evalua(P_{g+1})
        g ⟵ g + 1
    end
fin
```

Figura 2.2: Pseudo código - Algoritmo Genético

- Valores reales para representar conjuntos de condiciones, pesos sinápticos en redes neuronales, o los ángulos de torsión en las estructuras de proteínas.

En la codificación de un cromosoma, además de incluir la información respectiva a la solución candidato, se pueden añadir meta-datos que controlen o manipulen la manera en que se aplican los operadores genéticos.

En este trabajo, solamente utilizaremos una representación con valores reales. La razón fundamental se debe a que el cómputo de punto flotante es uno de los recursos que se desean explotar al trabajar con *GPU*s, además de ser común a la representación utilizada en las otras meta-heurísticas de este trabajo.

Existen múltiples formas de codificar nuestras variables en un dominio real. Al igual que en una representación de valores enteros, se puede utilizar una cadena de bits para codificar un rango de valores, separando el espacio de búsqueda en valores discretos. Al utilizar esta estrategia de codificación, se mantiene una precisión uniforme a lo largo del dominio de búsqueda, pero el tamaño de la codificación crecerá rápidamente, teniendo un costo muy alto en espacio de almacenamiento y en el tiempo de cómputo de los operadores genéticos sobre el cromosoma. También tendrá un impacto negativo en el rendimiento de la evaluación de aptitud, ya que se requiere de un paso adicional para decodificar las variables, cuya complejidad crece con el tamaño del cromosoma. Por estas razones resulta más eficiente utilizar una representación real con variables de punto flotante. [Herrera et al., 1998]

Un cromosoma que codifica D variables, será representado por un arreglo de D dimensiones. Habrá que subrayar que esto trae consigo restricciones y problemas de precisión, inherentes a la representación y aritmética que implementa el procesador [Goldberg, 1991]. Una vez seleccionada nuestra representación, se tienen que definir los operadores de recombinación y mutación *ad-hoc* a esta codificación.

2.3.2.2 *Métodos de selección*

Con la representación y codificación de los datos definidos, podemos elegir cómo realizar la selección de genomas para el proceso de reproducción.

Como ya había sido mencionado anteriormente, la presión selectiva tiene que ser balanceada por la variación que introducen los operadores genéticos. En el proceso de búsqueda de un algoritmo genético, existen dos imperativos:

- La explotación de la información existente en los genomas de una población.
- La exploración de nuevas soluciones, mediante la mutación de los genes.

Será sumamente importante mantener diversidad en el conjunto poblacional para que el algoritmo genético continúe haciendo progreso e ir evolucionando nuevas soluciones. Si se reduce la diversidad genética de toda la población a un solo cromosoma con mínimas variaciones entre individuos, el algoritmo genético sólo estará explorando y explotando la información de esta localidad, sin poder acceder al resto del espacio de búsqueda. Bajo esta observación: una presión selectiva demasiado alta puede resultar en la convergencia prematura de la búsqueda, quedando atrapada en un óptimo local. El caso contrario, al utilizar una presión selectiva débil resultará en que el algoritmo genético no logre aprovechar la información genética, alentando el proceso evolutivo.

2.3.2.3 *Selección Mediante Torneo*

Existen múltiples algoritmos para realizar el proceso de selección en un conjunto poblacional, y cada método tiene características y variables que pueden introducir diferentes niveles de presión selectiva [Sivanandam & Deepa, 2007]. Por las restricciones que implica el implementar una versión paralela del algoritmo genético, este trabajo

utilizó el método de selección mediante torneo, con el cual se podrán calcular todas las operaciones de selección de manera independiente.

El método de selección por torneo utiliza dos parámetros mediante los cuales se puede controlar la presión selectiva: El tamaño del torneo n y el parámetro estocástico de selección p.

De manera resumida la selección por torneo involucra realizar los siguientes pasos:

1. Se puebla el conjunto de individuos que participarán en el torneo T, seleccionando n individuos de la población de manera aleatoria.

2. Si $|T| >= 2$, tomamos los primeros dos elementos en el torneo T_0 y T_1 y comparamos sus valores de aptitud. Dada una variable aleatoria uniforme distribuida $r \in [0, 1)$, si $r < p$ removemos de T al individuo con menor valor de aptitud, en caso contrario removemos al individuo con mayor valor de aptitud.

3. Si $|T| = 1$, el único individuo que queda en T es seleccionado para la recombinación genética. En caso contrario, regresamos al paso 2.

Otros métodos de selección requieren de múltiples rutinas ejecutadas de manera secuencial para poder obtener estadísticas de la aptitud del conjunto poblacional en cada generación, y de esta manera sesgar la selección en función de la aptitud relativa a los demás individuos. A diferencia a la selección por *ranking*, la selección mediante torneo provee una manera de ajustar la presión selectiva, además de ser computacionalmente más eficiente y paralelizable. [Sivanandam & Deepa, 2007]

2.3.2.4 *Recombinación*

El operador de recombinación es la característica más importante que distingue a un algoritmo genético de otras meta-heurísticas. La idea detrás del operador se inspira en el concepto biológico de la reproducción sexual; en donde el intercambio genético entre padres produce un nuevo genotipo, heredando características de ellos. La meta de este operador, será explotar la información ya existente en el conjunto poblacional, generando nuevas variaciones genéticas a partir de la recombinación de dos o más cromosomas.

Al utilizar una representación real existen diferentes formas en las que se puede realizar una recombinación de los cromosomas. A

continuación se describe el método de cruza promedio, implementado en este trabajo.

2.3.2.5 *Cruza Promedio (Davis, 1991)*

Este método utiliza los valores de cada uno de los genes en los cromosomas padre como base, para así producir una combinación en el intervalo definido por los genes padres.

Para cada cromosoma en la población $\vec{x}_i$, dada una variable aleatoria uniformemente distribuida $rand_i \in [0,1)$ y el factor de recombinación $CO_r \in [0,1)$, si $rand_i <= CO_r$ entonces se realiza el proceso de recombinación con el cromosoma seleccionado $\vec{x}_s$, de lo contrario se mantiene el cromosoma $\vec{x}_i$ en la siguiente generación sin alteraciones.

Definición 6. *En cada instancia de recombinación tendremos el cromosoma del primer padre $\vec{x}_{i,g}$, el cromosoma seleccionado para recombinación $\vec{x}_{s,g}$ y el cromosoma resultante $\vec{c}$*

$$\begin{aligned}\vec{x}_{i,g} &= (\vec{x}_{i,j,g}),\\ \vec{x}_{s,g} &= (\vec{x}_{s,j,g}),\\ \vec{c} &= (\vec{c}_{j,g})\end{aligned}$$

Si $\beta = 1$, el gen del padre A se propaga sin ser modificado y el gen del padre B desaparece. Cuando $\beta = 0$, sucede el caso contrario, propagando sólo el gen del padre B. Cuando $\beta = 0.5$ el gen resultante es el promedio de ambos padres. [Haupt & Haupt, 1998]

$$\vec{c}_{j,g} = \beta \cdot \vec{x}_{i,j,g} + (1-\beta) \cdot \vec{x}_{s,j,g} \tag{2.6}$$

Para ilustrar la operación de la *cruza promedio*, veamos un ejemplo:

$$\begin{aligned}\vec{x}_i &= [4.0]\\ \vec{x}_s &= [8.0]\\ \beta &= 0.75\end{aligned}$$

$$\begin{aligned}\vec{c} &= [(0.75 \cdot 4.0) + ((1.0 - 0.75) \cdot 8.0)]\\ \vec{c} &= [(3) + ((0.25) \cdot 8.0)]\\ \vec{c} &= [(3) + (2)]\\ \vec{c} &= [5]\end{aligned}$$

2.3.2.6 *Operador de Mutación*

En un algoritmo genético el operador de mutación será la clave para explorar el espacio de búsqueda, y es importante mantener un balance entre la tasa del operador de recombinación y el de mutación. Una taza alta de mutación puede ser destructiva para la información que trasciende mediante la recombinación, convirtiendo la transformación en una búsqueda aleatoria. Por otro lado, en una función multimodal, una taza de mutación muy baja tendrá como efecto la convergencia prematura en un mínimo global.

Al utilizar una representación real existen diferentes formas en las que se pueden perturbar los datos para crear el efecto de una mutación en el gen.

Para la implementación del algoritmo genético de este trabajo, se diseñó el algoritmo descrito en la Figura 2.3. El objetivo de este método es poder obtener un cromosoma mutante $\vec{c}'$ perturbando el valor de cada uno de los genes de $\vec{c}$. La perturbación debe de tener las siguientes características: con alta probabilidad el nuevo valor será cercano a la variable original, y con baja probabilidad el nuevo valor será lejano al original. Los pasos del algoritmo son los siguientes:

1. Se calcula B como el promedio de la suma de n variables aleatorias uniformemente distribuidas en el intervalo $[0, 1)$.

$$B = \frac{1}{n} \sum_{k=1}^{n} U_k$$
$$B \in [0, 1)$$

2. Definimos B' al transformar B para tome valores con magnitud del rango de la variable $\vec{c}_j$ en el dominio definido para la función objetivo, en el sentido negativo y positivo.

$$rango_j = limite_{j,superior} - limite_{j,inferior}$$
$$B' = (B - 0.5) \cdot (2 \cdot rango_j)$$

3. Ahora aplicamos la perturbación, centrados sobre la variable original $\vec{c}_j$, utilizando el valor de B' escalado por el factor de desviación p, para generar la variable mutante $\vec{c'}_j$.

$$\vec{c}'_j = \vec{c}_j + (B' \cdot p)$$

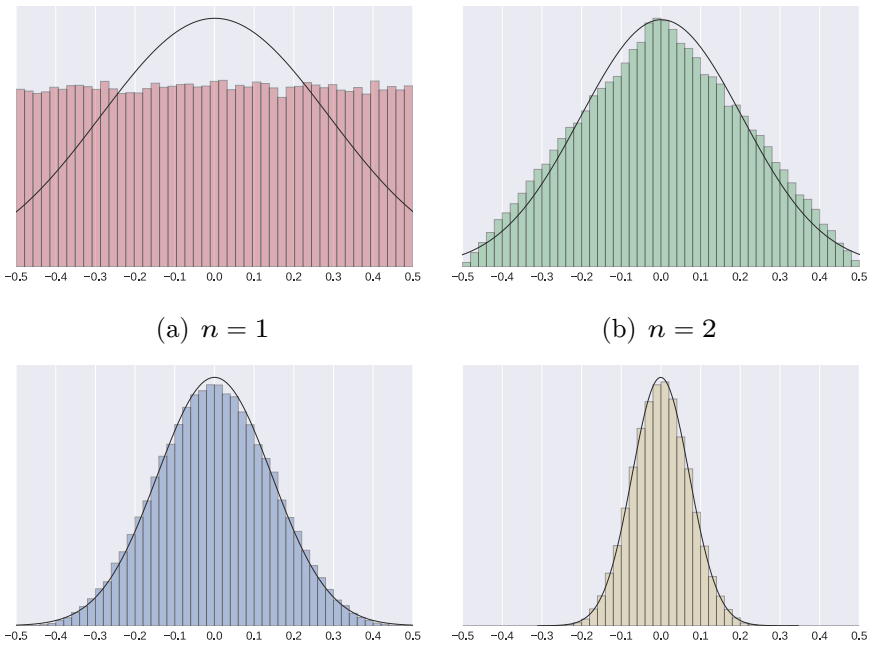
El valor de B tiene una distribución de *Bates* en el intervalo $[0, 1)$ [Bates, 1955]. Dado que la distribución de *Bates* converge rápidamente a una distribución normal al incrementar el número de iteraciones n,

se puede utiliza eficazmente para perturbar el valor del gen original $\vec{c}_j$ en una vecindad delimitada por el rango de la variable. Utilizando el factor de desviación p, se puede acotar la perturbación a un porcentaje del $rango_j$.

En la Figura 2.3 se muestra el pseudocódigo del algoritmo de mutación implementado, y en la Figura 2.4 se muestran los histogramas de la variable B para diferentes valores de n, ilustrando la convergencia de la distribución.

```
inicio
    j ⟵ 0
    while j < D do
        m_j ← rand(0, 1)
        if m_j ≤ M_r then
            B ← 0
            while i < n_b do
                B+ = rand(0, 1)
            end
            B ← B × 1/n_b
            B′ ← (B − 0.5) × rango_g
            c′_j = c_j + (B′ × p)
        end
        j ⟵ j + 1
    end
fin
```

Figura 2.3: Pseudo código - Algoritmo de mutación utilizando distribución de Bates



(a) $n = 1$ (b) $n = 2$

(c) $n = 4$ (d) $n = 16$

Figura 2.4: Gráficas de densidad de probabilidad e histogramas para distribución de Bates.

2.3.3 *Evolución Diferencial*

En 1995, Kenneth Price y Rainer Storn, estaban trabajando en problemas de optimización combinatoria con otra meta-heurística evolutiva llamada *Recocido Genético* [Schneier, 1994], la cual utiliza una representación de cadenas de bits. Al ver que el algoritmo convergía muy lentamente decidieron modificarlo para utilizar una representación de punto flotante y con ello definir nuevas operaciones para esta representación. Fue en este proceso que descubrió el método para optimización numérica de donde surge la idea de evolución diferencial (*DE*: del inglés *Differential Evolution*). [Storn & Price, 1997]

Como otros algoritmos evolutivos, la técnica de evolución diferencial es un optimizador basado en poblaciones donde los individuos primero pasan por un proceso de transformación y posteriormente son seleccionado en función del índice de aptitud.

Definición 7. *Definamos P_g como el conjunto poblacional de N individuos en la generación g, a lo largo de $g_{max} - 1$ generaciones.*

$$P_g = (\vec{x}_{i,g}),\ i = 0, 1, ..., N-1,\ \ g = 0, ..., g_{max} - 1 \tag{2.7}$$

Definición 8. *Cada individuo en la población está representado por un vector* $\vec{x}_{i,g}$ *de dimensión* D*.*

$$\vec{x}_{i,g} = (\vec{x}_{i,j,g}),\ j = 0, 1, ..., D-1. \tag{2.8}$$

Los pasos del algoritmo de Evolución Diferencial son los siguientes:

1. Inicialización del conjunto poblacional: En la primera generación $g = 0$, para cada uno de los individuos de la población se asignan valores aleatorios uniformemente distribuidos dentro del intervalo que define el dominio de búsqueda.

$$\begin{aligned}\vec{x}_{i,j,0} = \ & rand_j[0,1) \cdot (limite_{j,superior} - limite_{j,inferior}) \\ & + limite_{j,inferior}\end{aligned}$$

2. Operador de mutación diferencial: Aplicando un operador de mutación diferencial, sobre tres vectores de la población seleccionados de manera aleatoria: $\vec{x}_{r0}, \vec{x}_{r1}, \vec{x}_{r2}$, con $r_0 \neq r_1,\ r_0 \neq r_2,\ r_1 \neq r_2$.

   Definición 9. *Para* $F \in \mathbb{R},\ F \geq 0$*, se define un nuevo vector de mutación:*

$$\vec{v}_{i,g} = \vec{x}_{r0,g} + F \cdot (\vec{x}_{r1,g} - \vec{x}_{r2,g})$$

   F es el factor de escalamiento que controla la velocidad a la cual evoluciona la población. Aunque F puede ser cualquier valor positivo, se ha observado que los valores que producen una perturbación efectiva rara vez son mayores a 1.0. [Storn & Price, 1997]

   El operador de mutación introduce perturbaciones en la información que codifican los agentes; fomentando diversidad en la población y la exploración del espacio de búsqueda. Resulta crucial el utilizar un factor de escalamiento F con magnitud suficiente para contrarrestar la presión selectiva y evitar la convergencia prematura. [Zaharie, 2002]

3. Operador de recombinación: Se aplica un operador de recombinación para generar un nuevo vector $\vec{u}_{i,g}$, tomando elementos de manera aleatoria entre el vector original de la población $\vec{x}_{i,g}$ y el vector que se obtuvo en el paso de la perturbación $\vec{v}_{i,g}$. El operador de *recombinación uniforme* define el proceso de recombinación de la siguiente manera:

$$\vec{u}_{i,j,g} = \begin{cases} \vec{v}_{i,j,g} & \text{si } rand_j(0,1) \leq Cr \text{ y } j \neq j_{rand} \\ \vec{x}_{i,j,g} & \text{en el caso contrario} \end{cases}$$

Cr es el factor de recombinación y j_{rand} un índice de las D dimensiones, seleccionado de manera aleatoria por cada recombinación. Con una probabilidad $Cr \in [0,1)$ el operador toma el elemento $\vec{x}_{i,j,g}$ y en caso contrario toma $\vec{v}_{i,j,g}$, recorriendo las D dimensiones para generar $vecu_{i,g}$.

El operador siempre copia el elemento $\vec{x}_{i,j_{rand},g}$ en $\vec{u}_{i,j_{rand},g}$, para garantizar la persistencia de una dimensión del vector original en el vector de recombinación.

4. Proceso de selección: Se utiliza un esquema de supervivencia 1:1, en donde el vector candidato $\vec{u}_{i,g}$ es comparado con el vector original $\vec{x}_{i,g}$:

$$\vec{x}_{i,g+1} = \begin{cases} \vec{u}_{i,g} & \text{si } Aptitud(\vec{u}_{i,g}) > Aptitud(\vec{x}_{i,g}) \\ \vec{x}_{i,g} & \text{en el caso contrario} \end{cases}$$

El vector con mayor aptitud pasará a formar parte del conjunto poblacional de la siguiente generación.

5. Si aún no se cumple el criterio de terminación, regresar al paso número 2.

En la Figura 2.5 se muestra el pseudo-código del algoritmo de Evolución Diferencial.

```
inicio
    g ⟵ 0
    Inicializa(P_g)
    Evalua(P_g)

    while no se cumpla el criterio de terminación do
        // N es el número de individuos en P_g
        for i ← 0 to N do
            // Selección de tres individuos distintos en la población
            r_a ← rand(0,1) × N
            while r_a = r_b do r_b ← rand(0,1) × N
            while r_a = r_c OR r_b = r_c do r_c ← rand(0,1) × N

            v_{i,g} ← MutacionDiferencial(r_a, r_b, r_c)
            j_rand ← rand(0,1) × D

            // D es el número de dimensiones en cada vector
            for j ← 0 to D − 1 do
                if rand(0,1) ≤ Cr OR j_rand == j then
                    u_{i,j,g} = v_{i,j,g}
                end
                else
                    u_{i,j,g} = x_{i,j,g}
                end
            end
        end
        if Aptitud(u_{i,g}) ≥ Aptitud(x_{i,g}) then
            P_{i,g+1} = u_{i,g}
        end
        else
            P_{i,g+1} = x_{i,g}
        end
        g ⟵ g + 1
    end
fin
```

Figura 2.5: Pseudo código - Algoritmo de Evolución Diferencial

2.4 INTELIGENCIA DE ENJAMBRE

Existe otro grupo de meta-heurísticas también basadas en conjuntos poblacionales, las cuales toman inspiración en el comportamiento colaborativo de ciertos organismos sociales: colonias de hormigas, enjambres de abejas, parvadas de aves, etc. Aunque comparten ciertas características con los algoritmos evolutivos, este grupo de meta-heurísticas explota la información cognitiva del enjambre sin tener un énfasis en operadores de recombinación genética y presión selectiva.

La Inteligencia de Enjambre, es una disciplina dentro del campo de la Inteligencia Artificial, enfocada en la investigación del comportamiento colectivo de sistemas descentralizados y auto-organizados[2]. Estos sistemas consisten de un conjunto de agentes con la capacidad de comunicar información entre ellos y de reaccionar ante su entorno. Cada uno de los agentes sigue una serie de reglas muy simples y cada uno de ellos no tiene la capacidad de efectuar tareas complejas. Sin embargo, es mediante las interacciones entre todos agentes en el enjambre, que se induce un comportamiento complejo de donde emerge la inteligencia. [Blum & Merkle, 2008]

Mark M. Millonas, quien estudió modelos de enjambre en el contexto de vida artificial, enlista cinco principios en la Inteligencia de Enjambre [Millonas, 1994]:

- Proximidad: Capacidad de realizar cálculos sobre el espacio y tiempo.
- Calidad: Capacidad a responder a factores de calidad en el entorno.
- Diversidad: Evitar actividad excesiva sobre canales angostos.
- Estabilidad: Tolerancia a perturbaciones en el entorno sin desorganizar al enjambre.
- Adaptabilidad: Cambiar el comportamiento del enjambre cuando existe un beneficio.

2.4.1 *Cúmulo de Partículas*

El cúmulo de partículas (*PSO*: del inglés *Particle Swarm Optimization*) es una técnica de optimización estocástica basada en poblaciones y pertenece al conjunto de algoritmos dentro de la Inteligencia de Enjambre. El algoritmo fue propuesto por James Kennedy y Russell Eberhart, inspirado en el comportamiento social observado en parvadas de aves [Kennedy & Eberhart, 1995]. Utilizando un conjunto de partículas con la capacidad de comunicarse entre sí, cada una de ellas explora el espacio de búsqueda modificando su posición en base a dos factores: su experiencia individual y la experiencia colectiva de los demás individuos en la población. [Blum & Merkle, 2008]

2 La auto-organización es un conjunto de mecanismos dinámicos, por los cuales una estructura emerge en un sistema a partir de las interacciones de sus componentes más básicos. [Bonabeau et al., 1999]

Definición 10. *La descripción de cada partícula estará dada por una serie de vectores: posición ($\vec{x}$), cognición ($\vec{c}$) y velocidad ($\vec{v}$).*

$$Para \quad i = 0, 1, ..., N-1, \;\; j = 0, 1, ..., D-1, \;\; g = 0, ..., g_{max} - 1$$
$$\vec{x}_{i,g} = (\vec{x}_{i,j,g}), \tag{2.9}$$
$$\vec{c}_{i,g} = (\vec{c}_{i,j,g}), \tag{2.10}$$
$$\vec{v}_{i,g} = (\vec{v}_{i,j,g}). \tag{2.11}$$

Definición 11. *Definamos la población del enjambre P_g como el conjunto de N partículas, a lo largo de g_{max} generaciones.*

$$Para \quad i = 0, 1, ..., N-1, \;\; j = 0, 1, ..., D-1, \;\; g = 0, ..., g_{max} - 1$$
$$P_g = (\vec{x}_{i,g}, \vec{c}_{i,g}, \vec{v}_{i,g}) \tag{2.12}$$

La posición de cada partícula $\vec{x}$ representa una posible solución. Su vector de cognición $\vec{c}$, guarda la posición con mejor aptitud en la que ha estado anteriormente esa misma partícula. El vector de velocidad $\vec{v}$ define la dirección en la cual la partícula navega en el espacio de búsqueda.

Para que las partículas puedan realizar operaciones comparativas sobre su posición actual y posiciones anteriores, cada una almacena el valor de aptitud en $\vec{x}$ y el valor de aptitud de su vector de cognición $\vec{c}$.

Definición 12. *El vector social $\vec{s}_g$ codifica la mejor posición histórica de todas las partículas del enjambre hasta la generación g.*

$$\vec{s}_g = (\vec{s}_{j,g}) \;\; tal \;\; que \tag{2.13}$$

$$aptitud(\vec{s}_g) \geq aptitud(\vec{x}_{i,g})$$
$$\forall i \in [0, N), \;\; \forall g \in [0, g)$$

Las partículas navegan por el espacio de búsqueda, modificando su posición en base a su velocidad multiplicada por el factor de inercia ω. A su vez la velocidad de cada partícula es alterada por dos fuerzas:

- La fuerza de cognición individual con dirección a $\vec{c}$, multiplicada por el factor ϕ_c.
- La fuerza de cognición social con dirección a $\vec{s}$, multiplicada por el factor ϕ_s.

Los factores ω, ϕ_c y ϕ_s configuran el comportamiento del enjambre, dando distintos pesos a: la posición actual de cada

partícula, la mejor posición de la partícula en el pasado y la mejor posición del enjambre.

Los pasos del algoritmo son los siguientes:

1. Inicialización del conjunto poblacional: En la primera generación $g = 0$, para cada vector de posición, cognición, se asignan valores aleatorios uniformemente distribuidos dentro del intervalo que define el dominio de búsqueda.

$$\vec{x}_{i,j,0},\vec{c}_{i,j,0} = \\ rand_j[0,1) \cdot (limite_{j,superior} - limite_{j,inferior})$$

   De manera similar, para el vector de velocidad se genera un vector uniformemente distribuido sobre un intervalo de velocidad.

$$\vec{v}_{i,j,0} = \\ rand_j[0,1) \cdot (limiteVel_{j,superior} - limiteVel_{j,inferior})$$

   Al definir rango de velocidad permitido, se acota la distancia que puede recorrer una partícula en cada iteración, previniendo que salgan las partículas del espacio de búsqueda.

2. Actualiza los vectores de cognición individual y social en función de las posiciones de la generación actual.

$$\text{si } aptitud(\vec{x}_{i,g}) \geq aptitud(\vec{c}_{i,g}) \\ \text{entonces} \quad \vec{c}_{i,g} = \vec{x}_{i,g}$$

$$\text{si } aptitud(\vec{c}_{i,g}) \geq aptitud(\vec{s}_g) \\ \text{entonces} \quad \vec{s}_g = \vec{c}_{i,g}$$

3. Actualiza el vector de velocidad de cada individuo $\vec{v}_{i,j,g+1}$ utilizando la velocidad previa $\vec{v}_{i,j,g}$ multiplicada por el factor de inercia ω y añadiendo dos componentes de aceleración: el vector de cognición individual $\vec{c}_{i,j,g}$ multiplicado por el factor de peso ϕ_c y el vector de cognición social $\vec{s}_{i,j,g}$ por el factor de peso ϕ_s.

$$\vec{v}_{i,j,g+1} = \omega \cdot \vec{v}_{i,j,g} + \\ \phi_c \cdot rand_j[0,1) \cdot (\vec{c}_{i,j,g} - \vec{x}_{i,j,g}) + \\ \phi_s \cdot rand_j[0,1) \cdot (\vec{s}_{j,g} - \vec{x}_{i,j,g})$$

4. Actualizar el vector de posición de cada agente $\vec{x}_{i,j,g}$ en base a la nueva velocidad calculada $\vec{v}_{i,j,g+1}$

$$\vec{x}_{i,j,g+1} = \vec{x}_{i,j,g} + \vec{v}_{i,j,g+1}$$

5. Si no se ha cumplido el criterio de terminación, regresar al segundo paso.

En la Figura 2.6 se muestra el pseudo-código del algoritmo de Cúmulo de Partículas.

inicio
  $g \longleftarrow 0$
  `Inicializa(`P_g`)`
  `Evalua(`$x_{i,j,0}$`)`

  `Evalua(`$c_{i,j,0}$`)`

  $s_g \leftarrow x_{0,g}$

  while *no se cumpla el criterio de terminación* **do**
    for $i \leftarrow 0$ **to** N **do**
      for $j \leftarrow 0$ **to** K **do**
        $v_{i,j,g+1} = \omega \cdot v_{i,j,g} +$
        $\phi_c \cdot rand(0,1) \cdot (c_{i,j,g} - x_{i,j,g}) +$
        $\phi_s \cdot rand(0,1) \cdot (s_{j,g} - x_{i,j,g})$

        $x_{i,j,g+1} = x_{i,j,g} + v_{i,j,g+1}$
      end

      if `Aptitud(`$x_{i,g+1}$`)` $\geq$ `Aptitud(`$c_{i,g}$`)` **then**
        $c_{i,g+1} = x_{i,g+1}$
      end
      else
        $c_{i,g+1} = c_{i,g}$
      end

      if `Aptitud(`$c_{i,g+1}$`)` $\geq$ `Aptitud(`s_g`)` **then**
        $s_{g+1} = c_{i,g+1}$
      end
      else
        $s_{g+1} = s_{i,g}$
      end
    end
    $g \longleftarrow g + 1$
  end
fin

Figura 2.6: Pseudo código - Algoritmo de Cúmulo de Partículas

El tamaño del cúmulo de partículas también es un parámetro crítico en el rendimiento del algoritmo. Al igual que lo fue para las otras meta-heurísticas, si se utiliza un cúmulo con muy pocas partículas el algoritmo va a converger en mínimos locales rápidamente. En el caso opuesto, el utilizar demasiadas partículas

aumentará considerablemente el costo computacional del algoritmo [Chen et al., 2015]. Sin embargo, en cada enjambre el factor social afecta únicamente a los individuos de una población, siendo particularmente importante para esta meta-heurística el ajustar el número de partículas, así como los pesos de cada aceleración. Escalar el algoritmo para realizar una búsqueda más exhaustiva puede requerir dividir la ejecución en varios enjambres [Röhler & Chen, 2011].

3

CÓMPUTO PARALELO

It may be that brain hardware has co-evolved with the internal virtual worlds that it creates. This can be called hardware-software co-evolution.

— *Richard Dawkins, 1993*

Desde la invención de la primera computadora de propósito general, las computadoras se convirtieron en una herramienta indispensable para la investigación y el desarrollo de nuevas tecnologías. Por esta razón, hemos buscado incesantemente maneras de incrementar la capacidad de cómputo que tenemos a nuestra disposición.

En un artículo publicado en el año de 1965 [Moore, 1965] Gordon E. Moore describió como los avances tecnológicos permitirían duplicar la densidad de transistores en un circuito cada dos años, manteniendo la proporción de costo de manufactura constante; a esta proyección se le conoce como *Ley de Moore*.

Nueve años más tarde en un nuevo artículo, Robert H. Dennard formuló que el voltaje y corriente son proporcionales a las dimensiones lineales de un transistor; a este fenómeno se le conoce como el *escalamiento de Dennard*. Junto con la Ley de Moore esto permitiría crear procesadores con más transistores en cada generación.

Durante más de tres décadas, la industria de los microprocesadores logró mantener esta velocidad en el incremento de la densidad de transistores y la frecuencia de reloj a la que trabajan. Desde 1978 hasta el 2003, en promedio los procesadores aumentaron en un 52 % su rendimiento cada año. Para ilustrar este progreso realicemos la siguiente comparación: En el año de 1971 Intel introdujo el procesador *4004*, que contaba con 2,300 transistores corriendo a una velocidad de 740 KHz. Para el año 2002 el procesador *Pentium 4 Northwood* de Intel contaba con 55,000,000 transistores corriendo a 3.06GHz. En el 2015 Intel anunció el procesador *Broadwell-U*, el cual cuenta con 1,300,000,000 de transistores corriendo hasta 3.9GHz.

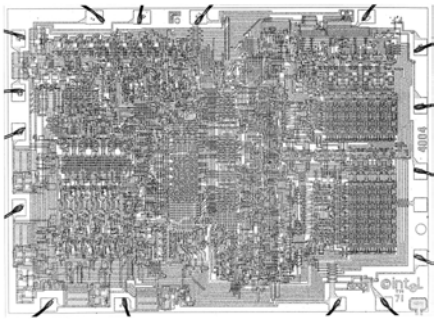
En la Tabla 3.1 se tiene una tabla comparativa mostrando las principales características de la última generación[1] de tres

1 Enero del 2017

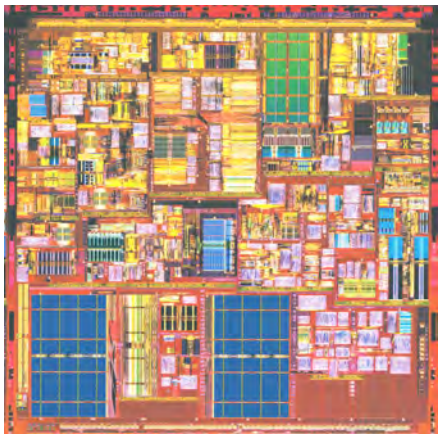
Micro-arquitectura	Núcleos	Reloj	Memoria Max.	Ancho de Banda	GFLOPs	TPD[4]
Broadwell x86	24	2.4GHz	3TB DDR4	85GB/s	~0.77	165W
Knights Landing x86	72	1.5Ghz	384GB DDR4	115GB/s	~2.41	250W
Pascal GP100	3840	1.1GHz	16GB GDDR5	720GB/s	~9.34	260W

Cuadro 3.1: Comparación entre la última generación de arquitecturas paralelas y masivamente paralelas

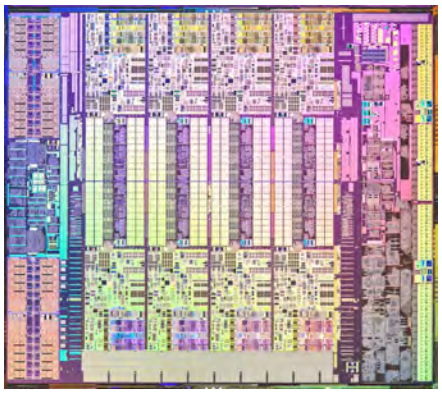
arquitecturas de cómputo paralelo: *Multi-core CPU*s, *MIC*s[2] y *GPU*s [3]



(a) Intel 4004, producido a 10 μm.



(b) Intel Pentium Northwood, producido a 130 nm.



(c) Intel Haswell-e, producido a $14nm$.

Figura 3.1: Procesadores Intel

Al incrementar la cantidad de transistores y su velocidad de reloj aumentamos de manera directamente proporcional la energía eléctrica que estos consumen. Una fracción importante de esta energía es transformada en calor por el circuito eléctrico. Cuando un

2 Many Integrated Core (MIC)
3 Graphics Processing Unit (GPU)
4 *Thermal Design Power (TDP)*

procesador alcanza temperaturas demasiado altas, el rendimiento del circuito se vuelve inestable. De ser suficientemente prolongada la exposición a altas temperaturas, las nano-estructuras del silicio llegan a desintegrarse, siendo la principal limitante por la cual desde el año 2003 se desaceleró el incremento en velocidad de reloj de los microprocesadores. [Sutter, 2005]

La manufactura de un microprocesador se realiza mediante procesos de foto-litografía sobre obleas de silicio. Para reducir el tamaño de un transistor se requiere un método de foto-litografía que permita realizar proyecciones con mayor resolución. Hasta la última década, esta había sido la estrategia más viable para producir microprocesadores cada vez más rápidos y eficientes. Sobra mencionar los obstáculos que implica trabajar a la escala de nanómetros[5]. A pesar de los grandes avances que la ingeniería ha logrado desarrollar en los últimos años, cada vez que se reduce el tamaño de manufactura se afrontan nuevos problemas que resultan más difíciles de resolver.

Existen límites físicos que restringen la resolución que se podría alcanzar, actualmente estamos al borde de estas barreras físicas. El proceso litográfico consta de proyectar un haz de luz sobre un compuesto fotosensible, técnica muy parecida a la de una fotografía análoga, pero en este caso funciona para construir los componentes del circuito capa por capa sobre las obleas de silicio. Los procesos para crear la nueva generación de microprocesadores utilizan el método de foto-litografía por inmersión, en el cual se utiliza un líquido con mayor índice de refracción que el aire. Este funciona como medio entre la fuente de luz y la superficie fotosensible, y de esta manera se logran mayores resoluciones en la proyección. En principio, la longitud de onda de la luz emitida define el tamaño de los componentes que se pueden crear.

Técnicas como *Multi-patterning*, en la cual se utiliza la sobreposición de múltiples máscaras de patrones mediante múltiples exposiciones, permite incrementar la resolución del proceso lito-gráfico. Utilizando *Extreme ultraviolet lithography* (EUV), en donde la longitud de onda de la luz emitida es de tan solo 13.5 nms, se han logrado crear componentes en escalas por debajo a $10nm$ [Borodovsky, 2012].

Aunque la magnitud de los componentes discretos del circuito son de tan sólo decenas de nanómetros, estos tienen elementos de dimensiones aún más pequeñas. Como referencia, el radio de un átomo de silicio es de 0.146 nm, en comparación un transistor creado en un

5 Un nanómetro es una billonésima de metro. $1nm = 10^{-9}m$

proceso de 14 nm está compuesto de una capa dieléctrica de tan sólo 0.5 nm, lo equivalente al radio de tan sólo 3 átomos.

En una compuerta lógica la función de un transistor es actuar como un interruptor, permitiendo o no el paso de electrones. Sin embargo, a estas escalas atómicas los electrones dejan de comportarse de manera usual y comienzan a experimentar fenómenos cuánticos. Particularmente existe un fenómeno conocido como *túnel cuántico*, en donde el electrón puede atravesar una barrera física. En un transistor esto introduce un grave problema, en donde el material dieléctrico de la compuerta lógica deja de funcionar como un aislante y comienza a fugar parte de la carga eléctrica. Como consecuencia de este fenómeno, el *escalamiento de Dennard* deja de ser cierto para transistores a estas escalas. [Wang, 2006]

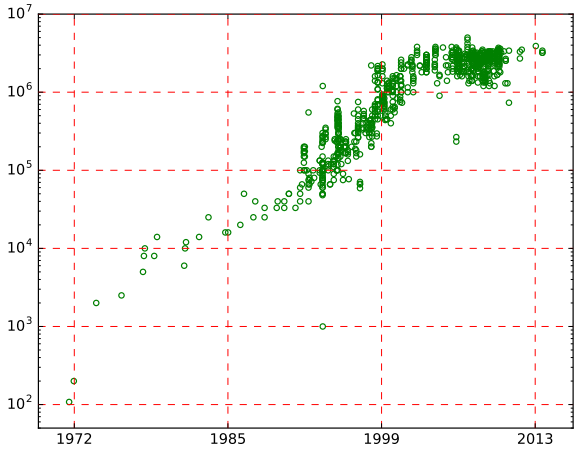
La industria de la manufactura de microprocesadores está realizando investigación de manera continua, buscando crear nuevos diseños de transistores que permitan mitigar los obstáculos causados por estos fenómenos. Sin embargo, para la tecnología de transistores tipo *MOSFET*[6], la desaceleración al incremento de densidad, velocidad y eficiencia es inevitable a largo plazo.

El panorama es menos alentador cuando realizamos el siguiente análisis: aún suponiendo que lográramos desarrollar técnicas que permitieran aumentar la densidad de cómputo en nuestros procesadores de manera indefinida, *las cotas de Bekenstein y Bremermann* [7] determinan el límite físico en el universo sobre el cómputo. En comparación con los procesadores que tenemos hoy en día sólo contamos con un factor de mejora en rendimiento de $10^{27}x$. Suponiendo que continuamos duplicando nuestro rendimiento cada 2 años, alcanzaríamos esta cota en 180 años. Se está al borde del límite físico de esta tecnología y pronto no será práctico seguir reduciendo el tamaño de los transistores. Bajo esta premisa, es necesario explorar otras estrategias para incrementar los recursos de cómputo a nuestra disposición.

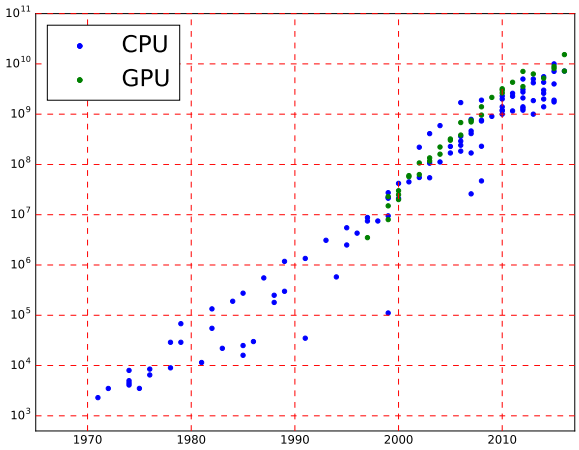
En oposición a los avances en procesos de manufactura, la evolución de micro-arquitecturas ha sido menos consistente durante la historia de los procesadores. El primer cambio de paradigma fue a inicio de la década de los ochenta, con el desarrollo y popularización de los lenguajes de programación de alto nivel, así como la creación de los sistemas operativos UNIX. Estos dos cambios en el ecosistema permitieron reducir el costo y riesgo de implementar nuevas

6 Metal-oxide-semiconductor field-effect transistor

7 La *Cota de Bekenstein* describe el límite físico de información que puede existir en una región finita de espacio, con una cantidad finita de energía. De esto se deriva otro límite, conocido como *cota de Bremermann*, la cual define el límite de velocidad de cómputo.



(a) Incremento exponencial en la frecuencia de reloj, alcanzando el punto de inflexión en el año 2003.



(b) Incremento lineal en el número de transistores.

Figura 3.2: Gráfica de tendencias en frecuencia y densidad de los transistores [Danowitz et al., 2012]

arquitecturas. Y es gracias a este contexto que nacen las arquitecturas con un número reducido de instrucciones, denominadas *RISC* (*Reduced instruction set computing*), cuyos diseñadores estaban interesados en implementar dos estrategias para aumentar el rendimiento:

- Explotar el paralelismo al nivel de instrucciones de un programa
- Introducción de jerarquía de memorias *caché*

3.1 ARQUITECTURAS DE COMPUTADORAS

3.1.1 *Arquitectura de Von Neumann*

También conocida como arquitectura de Princeton, es un modelo para el diseño de una computadora electrónica descrita en 1945 por el matemático John Von Neumann. El modelo está compuesto por una unidad central de procesamiento (CPU), un dispositivo de almacenamiento y la interconexión entre estos. El CPU a su vez se compone de dos unidades: la unidad de control, la unidad de lógica y aritmética (ALU).

La unidad de control se encarga de decidir qué instrucción se debe de ejecutar en cada momento, utilizando un registro especial denominado el contador del programa (CP), el cual almacena la dirección en memoria referente a la siguiente instrucción que será ejecutada en el CPU. Mientras la ALU se encarga de ejecutar cada una de las instrucciones ordenadas por la unidad de control.

La memoria del sistema está descrita por diferentes localidades, a las cuales se les puede hacer referencia mediante una dirección de Memoria RAM (Random Access Memory). En estas localidades de memoria se podrán almacenar tanto datos como la instrucciones del programa.

3.1.2 *Arquitecturas Modernas*

El modelo de Von Neumann sirvió como base en la implementación de las primera computadoras electrónicas. Sin embargo, sufre de varias limitantes; la más importante es el cuello de botella generado por la interconexión entre el CPU y la memoria. Aún al día de hoy, esta característica determina de manera

fundamental el rendimiento de cada arquitectura de cómputo. [Pacheco, 2011]

Al momento de diseñar la arquitectura de un procesador se analizan los costos y beneficios de integrar cada uno de sus componentes y funcionalidades. Dependiendo de los casos de usos que tendrá el procesador involucrará tomar decisiones que optimicen ciertas características. Por ejemplo, al diseñar un procesador de alto desempeño destinado para ser utilizado en súper-computadoras, se toma como prioridad maximizar el rendimiento y como objetivo secundario el reducir el consumo de energía. De manera diferente, al diseñar un procesador para dispositivos móviles que trabaja con baterías, el objetivo principal es el uso eficiente de energía y de manera secundaria el maximizar su rendimiento.

Cada arquitectura tendrá características particulares que a su vez implicarán restricciones específicas en la implementación de cualquier algoritmo. Comprender cómo en cada arquitectura el controlador de memoria pude calendarizar y realizar los accesos de lectura y escritura, será un aspecto fundamental para poder crear implementaciones que exploten las características para las cuales la plataforma fue diseñada.

En las primeras generaciones de microprocesadores, la velocidad del acceso a memoria RAM era comparable con la velocidad de acceso a los registros dentro del procesador. Fue a finales de la década de los ochenta que la brecha en el rendimiento de los procesadores y la memoria se convirtió en una limitante importante.

Grace Murray Hopper, una de las primeras personas en tener que desarrollar programas para una computadora y también pionera en el diseño de compiladores, comúnmente hacía la siguiente demostración a sus alumnos para ilustrar las limitantes físicas que existen al transformar y transmitir información. Primero mostraba un cable de teléfono de un pie [8] de largo, haciendo alusión a que esta era la máxima distancia que la luz podía recorrer en un nanosegundo. Después mostraba una bobina de mil pies de largo, siendo la máxima distancia que la luz podía recorrer en un microsegundo. Luego remarcaba lo imperativo que era el no desperdiciar un solo microsegundo. Este siendo el caso ideal, sin contar los costos implícitos en la transmisión de información por el circuito eléctrico, los protocolos de comunicación y el procesamiento de la información.

Desde el año 2000 contamos con procesadores que trabajan a frecuencias en el orden de miles de millones de ciclos por segundo, a esta frecuencia cada ciclo del reloj toma un nanosegundo.

8 Un pie equivale a 30.48 centímetros

Desearíamos que nuestras computadoras contaran con cantidades ilimitadas de memoria y que los accesos a esta a su vez fueran instantáneos. La realidad es que existen restricciones físicas que imponen obstáculos sobre el tamaño de la memoria que podemos utilizar, así como la velocidad a la cual se puede operar en ella y el costo que tiene integrarla. Independientemente de la tecnología utilizada, si se desea implementar una memoria con accesos de baja latencia [9] y un ancho de banda de gran capacidad, se requiere estar físicamente cerca del procesador. De manera contraria, si se desea implementar memoria con volúmenes de alta capacidad estos van a requerir una porción significante en área para poder ser integrados, lo cuál sólo podrían ser acoplados lejos del procesador implicando accesos a memoria más lentos. De esta manera, existe una relación inversa entre el tamaño de almacenamiento y la latencia implícita en los accesos entre los diferentes niveles de memoria.

3.1.2.1 *Jerarquías de Memoria*

El tipo de memoria utilizado por defecto desde la década de los setentas ha sido *DRAM* (Dynamic Random Access Memory), por que es la única tecnología que ha permitido año con año integrar volúmenes de memoria con mayor capacidad, velocidad y un costo práctico de integración. Sin embargo, tiene un gran inconveniente: la necesidad de refrescar las celdas de memoria. Las celdas en la *DRAM* utilizan pequeños transistores que almacenan una carga para definir el estado de la memoria. El realizar una lectura en *DRAM* es un proceso destructivo. Para leer el valor almacenado en las celdas de memoria, es necesario descargar el condensador, requiriendo restaurar la carga después de cada operación de lectura. También los condensadores no son perfectos, lo que significa que pierden parte de su carga con el tiempo, implicando tener que leer y recargar cada una de las celdas antes de que pierdan su información [10]. Estos requerimientos tienen una penalización tanto en consumo energético como en la latencia de ciertos patrones de accesos a memoria. [Bhati et al., 2016]

Para mitigar la gran brecha que existe entre la velocidad de la memoria y los procesadores se implementa una estrategia denominada *caching*, con la cual se intenta disminuir el costo de realizar accesos a localidades en memoria que ya han sido adquiridos de manera previa. El concepto de *caching* se puede implementar en toda la cadena de dispositivos utilizados en el almacenamiento de datos. Sin embargo, es de suprema importancia implementar dicha

9 Definiendo latencia como el intervalo de tiempo observado entre el inicio y término de un proceso

10 El las celdas de memoria DRAM, el tiempo constante de retención *tREFW* (Time Refresh Window) especificado por la industria es de 32 o 64 milisegundos

estrategia en la interconexión entre el procesador y la memoria, de donde se obtendrán las instrucciones y datos para operar. Una *jerarquía de memoria* es la implementación de varios niveles de *caché*, en donde en el nivel más bajo se tienen dispositivos con accesos de baja latencia y gran ancho de banda, pero con baja capacidad de almacenamiento. Estos primeros niveles en la jerarquía de memoria utilizan memoria *SRAM* (Static Random Access Memory), la cual puede ser integrada junto con el *CPU* en el mismo circuito. Conforme aumentan los niveles de la jerarquía, los dispositivos tienen mayor capacidad de almacenamiento a costo de mayor latencia en sus accesos.

Cada arquitectura implementa diferentes niveles y tamaños de *caches*, comúnmente teniendo hasta tres niveles en el circuito del procesador, denominados: *L1, L2 y L3* respectivamente. En las arquitecturas de multiprocesadores, los niveles más bajos de la jerarquía suelen ser específicos a cada núcleo, mientras que el último nivel es compartido entre todos los procesadores. Nuevas generaciones de computadoras también extienden nuevos niveles en la jerarquía utilizando *HBM* (High Bandwidth Memory) situada en el mismo empaquetado del procesador, lo que funciona como un nivel intermedio a los módulos de la memoria principal del sistema, *DIMM*s (*Dual In-line Memory Module*). Los detalles del proceso por el cual un procesador realiza accesos a memoria también son específicos a cada micro-arquitectura, la jerarquía que esta implementa y la tecnología de memoria que utiliza. A continuación daremos un ejemplo general de cómo opera un *controlador de memoria* (MC), la unidad en el procesador encargada de todas las transacciones de memoria.

Un *cache-line* es un bloque de múltiples *palabras* continuas en la memoria del sistema, y es la unidad mínima en la cual se pueden realizar transacciones de memoria. Por cuestiones de eficiencia un *MC* no opera en *palabras* individuales. El *MC* crea un diccionario en donde mantiene una relación de los *cache-lines* que existen en cada nivel de la jerarquía. Diferentes controladores de memoria utilizan distintas estrategias tanto para poblar como desalojar bloques en cada nivel de la jerarquía. Algunas implementaciones utilizan un mapeo 1:1 (*direct-mapped*), en donde cada *cache-line* es asignado a un lugar específico en el diccionario. Otras implementaciones como *n-way set associative*, agrupan bloques de tamaño n a los cuales se puede acceder de manera paralela. Independientemente de la implementación, eventualmente cuando se realizan referencias a nuevas localidades de memoria va a implicar el desalojar un *cache-line* que había sido previamente introducida al mapa del *caché*, a este proceso se le denomina *cache eviction*. Existen diferentes políticas y algoritmos de *cache eviction*, las diferentes

micro-arquitecturas implementan estrategias *ad-hoc* al número de procesadores que integran y la manera en que estos comparten la jerarquía de memoria. [Drepper, 2009]

Cada vez que el procesador realiza un acceso de lectura a cualquier dirección en memoria, el *MC* primero verificará si el *cache-line* al que pertenece la *palabra* en cuestión existe en alguno de los niveles de la jerarquía de memoria. En caso positivo, el *MC* puede pasar los datos para su consumo leyéndolo de este nivel, a esto se le denomina *cache-hit*. El caso contrario en donde no se encuentra la dirección se denomina *cache-miss*, para el cual el *MC* iterará sobre los niveles subsecuentes de la jerarquía hasta llegar al nivel en donde está presente el *cache-line* en cuestión. Una vez localizada la dirección se copia el *cache-line* al que pertenece, permeando los niveles inferiores de la jerarquía hasta alcanzar al *MC* donde puede ser consumido por el procesador.

El implementar una jerarquía de memoria permite explotar los *principios de localidad*, los cuales se definen a continuación. En la secuencia de instrucciones que describen a un programa suelen existir secciones de instrucciones en donde se itera un mayor número de ocasiones. De la misma manera existen ciertas direcciones que son utilizadas con más frecuencia. A esto se le denomina *localidad temporal*. También existe la misma relación con la *localidad espacial*, en donde secciones de memoria cercanas a aquellas que están siendo utilizadas, tienen alta probabilidad de tener referencias próximamente.

Por esta razón es una buena estrategia que el *MC* opere sobre bloques de varias direcciones de memoria contiguas a la vez, permeando la jerarquía de memoria para poder ser utilizado en referencias próximas. Esta característica permite explotar los dos *principios de localidad*.

En la tabla 3.2 se describen las latencias que tienen diferentes accesos a memoria, utilizando los distintos dispositivos que pueden existir en un sistema de cómputo actual.

Un concepto que es importante presentar es el de *false sharing*, lo cual ocurre cuando datos comparten el mismo *cache-line* dentro de su localidad en memoria. En arquitecturas de uniprocesadores esto no es un problema, sin embargo, en arquitecturas de multiprocesadores en donde varios núcleos pueden estar accediendo a un mismo *cache-line*, a pesar de no estar modificando el mismo dato esto crea contención sobre los mecanismos de coherencia de *caches* entre diferentes núcleos. La consecuencia es una desaceleración casi nula en el rendimiento de los núcleos. Una manera de evitarlo es manteniendo los datos manipulados aislados al contexto de cada hilo de ejecución. Otra

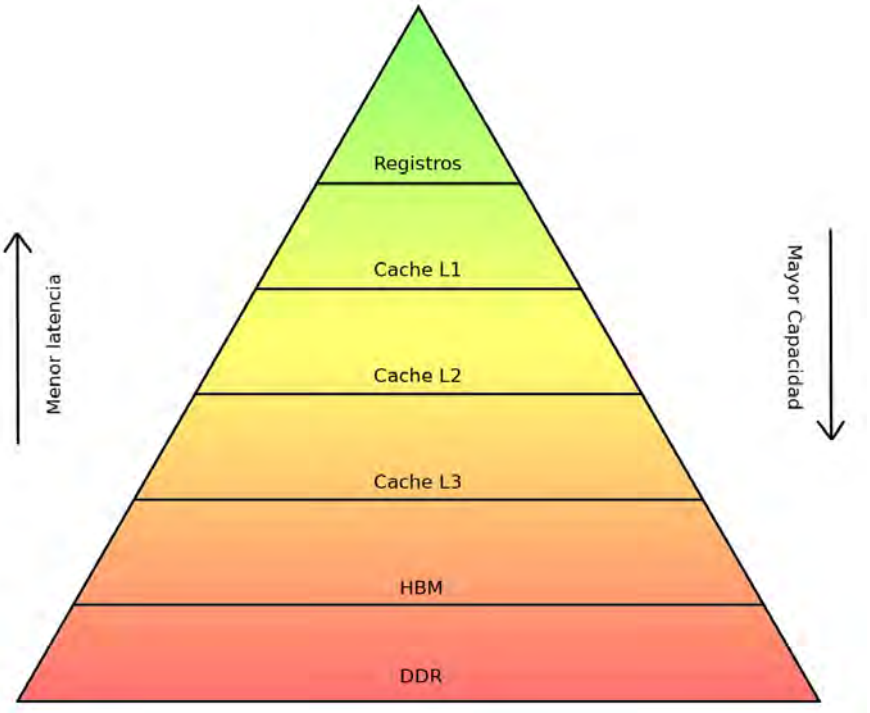


Figura 3.3: Jerarquía de memoria

Tipo de acceso	Latencia
Referencia a Cache L1	$0.5ns$
Falla en predicción de saltos	$5ns$
Referencia a Cache L2	$7ns$
Referencia a memoria principal	$100ns$
Transferencia de 1Kbyte por red 1Gbps	$10{,}000ns$
Lectura aleatoria de un disco SSD	$150{,}000ns$
Lectura secuencial de 1MByte de DDR	$250{,}000ns$
Lectura secuencial de 1MByte de un disco SSD	$1{,}000{,}000ns$
Lectura secuencial de 1MByte de un disco duro magnético	$20{,}000{,}000ns$

Cuadro 3.2: Comparación de magnitud en latencias.
[Norvig, 2001]

manera es introduciendo *padding*[11], para reservar el espacio en la localidad de cada variable que debe de ser manipulada de forma independiente sin bloquear los accesos a los otros núcleos. [Sutter, 2009]

11 El alineamiento de datos o estructuras está determinado por la manera en que los datos pueden ser almacenados y leídos en una computadora: esto está definido por el tamaño de la *palabra* de cada procesador. A la introducción de espacio entre los datos de una estructura para alinearse al tamaño de la palabra se le denomina *padding*.

3.1.2.2 *Memoria Virtual*

La *memoria virtual* es una técnica en el manejo de memoria, en donde se abstraen las localidades de memoria física, de tal manera que las tareas ejecutadas en un procesador trabajan sobre un rango continuo de direcciones virtuales. Uno de los múltiples beneficios de implementar memoria virtual, es que permite utilizar dispositivos de almacenamiento secundarios como parte de la memoria de un sistema, mientras que la memoria *RAM* se utiliza como un nivel adicional de *caché* en la jerarquía, que a su vez permite que los programas realicen alojamientos mayores al tamaño de memoria disponible en *RAM*. De manera muy similar a los *caches*, el sistema operativo utiliza una *tabla de paginación* para mapear los bloques de direcciones virtuales denominadas *páginas* a sus direcciones físicas en la memoria, manteniendo únicamente las páginas activas del programa en *RAM* y las páginas inactivas en dispositivos de almacenamiento secundario, denominado espacio *swap*.

Sin embargo, el utilizar direcciones virtuales de memoria introduce un nuevo problema, al requerir traducir con cada acceso a memoria entre direcciones virtuales y físicas. Para resolver este problema los procesadores que implementan memoria virtual tienen un *caché* dedicado para su uso llamado *TLB* (Translation Lookaside Buffer). En este *caché* el procesador puede almacenar ciertas entradas de la *tabla de paginación*. Esta estrategia también explota los principios de localidad, permitiendo accesos eficientes a direcciones que son utilizadas frecuentemente o cercanas a direcciones que ya fueron utilizadas.

Análogo a la funcionalidad de los niveles de *caché* en la jerarquía de memoria, si se tiene la página en cuestión dentro del *TLB* se le denomina *TLB-hit* y el caso contrario *TLB-miss*. De manera adicional, cuando la página no existe en memoria *RAM* se le denomina *page fault*, caso en donde se tendrá que extraer la página del dispositivo de almacenamiento secundario con un alto costo en ciclos de reloj inutilizados y tráfico adicional en los recursos de entrada/salida (*I/O*), externos a los subsistemas de memoria. A diferencia de los desalojos en *caché* y *TLB*, estas interacciones ya no son administradas por el hardware si no por el sistema operativo.

3.2 ARQUITECTURAS PARALELAS

El siguiente salto importante en el diseño de arquitecturas de cómputo ocurrió en los primeros años del siglo XXI, cuando la manufactura de uniprocesadores llegó al punto de inflexión de la Ley

de Moore y el escalamiento de Dennard, convirtiendo en un problema el depender únicamente en el avance de los procesos de manufactura para incrementar el rendimiento de los microprocesadores. Esto catalizó el enfoque en crear arquitecturas que integran múltiples núcleos como estrategia principal para incrementar el poder de cómputo.

El concepto de cómputo paralelo no es nuevo, tan pronto como existieron los primeros microprocesadores nació también el interés por crear súper-computadoras, las cuales integraban múltiples procesadores. Como caso singular, en el área de investigación de cómputo científico es común enfrentar problemas computacionalmente intensivos y que pueden ser resueltos de forma paralela, impulsando el desarrollo de nuevas arquitecturas masivamente paralelas.

En 1965, la organización de investigación CERN en Suiza recibió la primera súper-computadora comercial, la *CDC-6600* [Timelines, 1965]. El diseñador principal de esta máquina, Seymour Cray, estaba consciente de que la velocidad del procesador era sólo uno de los problemas a resolver si queríamos escalar los recursos de cómputo en órdenes de magnitud. Él estaba más interesado en atacar los problemas inherentes al diseño de todo el sistema computacional; interfaces de memoria, comunicación y almacenamiento. Únicamente resolviendo estos puntos de manera conjunta se podrían implementar eficientemente los métodos de coordinación y sincronización necesarios para explotar los recursos de computadoras masivamente paralelas. [Thornton, 1970] En 1982, Cray funda *Cray Research*, empresa que hasta la fecha se dedica exclusivamente al desarrollo de súper-computadoras, inicia una nueva industria y una nueva era del cómputo paralelo.

3.2.1 *Tipos de Paralelismo*

Desde la perspectiva de una aplicación, existen dos clasificaciones de paralelismo:

- A nivel de datos: Disponible cuando hay múltiples elementos de información que pueden ser procesados al mismo tiempo.
- A nivel de tareas: Disponible cuando se crean procesos de cómputo que pueden operar concurrentemente.

3.2.2 *Clasificación de arquitecturas paralelas*

Desde la perspectiva del hardware los dos tipos de paralelismo se pueden implementar utilizando alguno de los siguientes métodos de paralelización:

- **Paralelismo a nivel de instrucciones**: Arquitecturas con unidades de procesamiento en vectores pueden aplicar una instrucción a toda una colección de datos. Arquitecturas que implementan un pipeline de instrucciones pueden procesar varias instrucciones de manera paralela, intercalando las diferentes etapas en la cadena de instrucciones.

- **Hilos de ejecución concurrentes**: En arquitecturas que soportan ejecución e interacción entre hilos de ejecución concurrentes, se puede explotar tanto el paralelismo a nivel de datos como a nivel de tareas.

- **Peticiones concurrentes**: Es similar al paralelismo mediante hilos de ejecución, pero en tareas que no requieren de interacción logrando ejecutar cada una de ellas en manera concurrente, ya sea de forma explícita en la aplicación o delegando la coordinación al sistema operativo.

Existen múltiples maneras de clasificar los diferentes modelos de cómputo que implementa cada arquitectura. Michael Flynn, en su estudio sobre los diferentes tipos de paralelismo definió las siguientes cuatro categorías:

- **Una instrucción, un dato**(*SISD*): Esta categoría es lo que implementan los uniprocesadores. El programador ve al programa como una secuencia de instrucciones. Son capaces de explotar el paralelismo a nivel de instrucciones utilizando técnicas de ejecución súper-escalar[12] y ejecución especulativa[13].

- **Una instrucción, múltiples datos** (*SIMD*): Esta categoría es capaz de ejecutar una sola instrucción pero cada procesador utiliza un flujo de datos diferente. Estas computadoras pueden explotar el paralelismo al nivel de datos. A este grupo pertenecen

12 Un procesador superescalar implementa la ejecución de instrucciones de forma paralela entre las etapas del proceso de segmentación (*pipeline*), aumentando el número de instrucciones procesadas en un ciclo y por lo tanto el *throughput* total del procesador.

13 La ejecución especulativa es una técnica de optimización en un sistemas de cómputo, en la cual se utilizan los recursos de procesamiento disponibles — que en una ejecución ordinaria estarían sin utilizar — para efectuar operaciones que potencialmente serán requeridas más adelante en el tiempo, aumentando la capacidad de concurrencia del sistema

las unidades gráficas de procesamiento (*GPU*s), computadoras vectoriales y las extensiones al grupo de instrucciones x86[14].

- **Múltiples instrucciones, múltiples datos** (*MIMD*): Cada procesador es capaz de tomar su propia instrucción y operar en su propio flujo de datos, explotando el paralelismo a nivel de tareas y datos. Los procesadores multinúcleo y sistemas con múltiples procesadores pertenecen a esta categoría. A pesar de que este modelo es capaz de emular una arquitectura *SIMD*, la complejidad y costos de comunicación entre nodos no podrá tener un rendimiento comparable, además del costo que implica el integrar y escalar cientos o miles de nodos de cómputo.

- **Múltiples instrucciones, un sólo dato** (*MISD*): No existen procesadores comerciales que implementen este tipo de arquitectura. Las matrices sistólicas [15] son una arquitectura que comúnmente es identificada dentro de esta categoría. Sin embargo, en una matriz sistólica los datos son transformados al mismo tiempo que son transferidos entre nodos, de manera que los datos no tienen independencia. Por esta razón, las matrices sistólicas no son estrictamente una arquitectura *MISD*

A estas cuatro categorías se les conoce como *la taxonomía de Flynn*. Aunque estos grupos son muy generales nos servirán para clasificar el tipo de paralelismo disponible en cada problema y analizar cómo se puede explotar. Varias de las arquitecturas utilizadas hoy en día implementan más de una de estas categorías o alguna variación de estas.

Mientras un compilador esté informado de la arquitectura de la máquina objeto puede generar instrucciones que exploten el paralelismo a nivel de instrucciones. Los otros niveles de paralelismo requieren de una reestructuración algorítmica para poder ser explotados. En los casos en donde el paralelismo es inherente a los datos y sus transformaciones involucradas será sencillo utilizar los recursos disponibles de manera paralela. En las aplicaciones que requieran resolver problemas de sincronización y colaboración entre las tareas ejecutadas de forma concurrente, crear una implementación correcta y eficiente será una labor sumamente compleja. [McKenney, 2012]

14 Existen conjuntos de instrucciones adicionales, extensiones sobre el conjunto de instrucciones de *x86*, que añaden características para operar sobre registros con mayor número de bits, logrando realizar operaciones sobre múltiples datos en un sólo ciclo: *MMX*, *SSE**, *AVX**

15 Las matrices sistólicas (*systolic arrays*) son redes de unidades de procesamiento de datos (*DPU*s) independientes (nodos), utilizando transferencias síncronas entre sí.

3.2.3 *Arquitectura de un* GPU *moderno (NVIDIA)*

Antes de que existieran coprocesadores de gráficos, el *CPU* en una computadora también tenía que realizar los cálculos para poder generar y desplegar imágenes, lo cual requiere realizar múltiples operaciones de aritmética de punto flotante sobre cada uno de los *píxeles* que componen una imagen. La necesidad de poder crear visualizaciones interactivas[16], capaces de crear una secuencia de imágenes con una frecuencia de varios cuadros por segundo, fue la principal razón por la que surgen arquitecturas especializadas en este tipo de cómputo.

Cualquier programa que esté sintetizando imágenes con una resolución de 1920×1080 píxeles a $60Hz$[17], estará procesando 2,073,600 píxeles cada $16ms$. Utilizando un coprocesador especializado en realizar las mismas operaciones de punto flotante sobre grandes cantidades de datos en paralelo, se puede obtener el desempeño deseado con menor consumo de energía, además de liberar carga de trabajo al *CPU* y permitirle utilizar este tiempo para otras tareas.

Los primeros *GPU*s implementaban una estrategia fija, en donde sólo se podían realizar ciertas operaciones un orden determinado. La especialización de la arquitectura tenía como costo la pérdida de flexibilidad en su programación. Por décadas, el desarrollo de estos coprocesadores estuvo impulsado por los videojuegos y las aplicaciones de síntesis de gráficos. Cada año surgían nuevas necesidades, las cuales inducían cambios en el diseño de las arquitecturas de nuevas generaciones, con el propósito de poder proveer más poder de cómputo y la posibilidad de realizar nuevas operaciones para rasterizar[18] y procesar gráficos.

En el año 2001, Nvidia introduce una nueva arquitectura de *GPU*s que permite utilizar *Shaders*, pequeños programas que redefinen las rutinas para calcular el color de cada píxel de una imagen. Por primera vez se podía modificar la manera en que operaba el *GPU* sobre sus datos, mediante el cual se podían ejecutar nuevos procesos definidos por el programador dentro del *GPU*. [Kirk & Hwu, 2012]

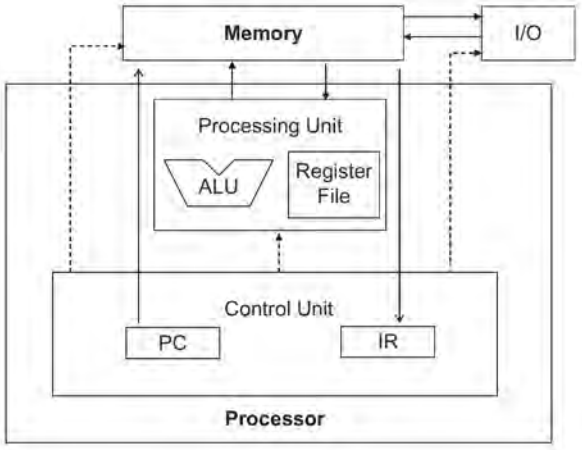
Ingenieros conscientes del poder de cómputo que existía en estos coprocesadores de gráficos comenzaron a implementar algoritmos

16 Una aplicación que ofrece al usuario una visualización en tiempo real (*real-time*), con la capacidad de interactuar en el entorno de manera dinámica.

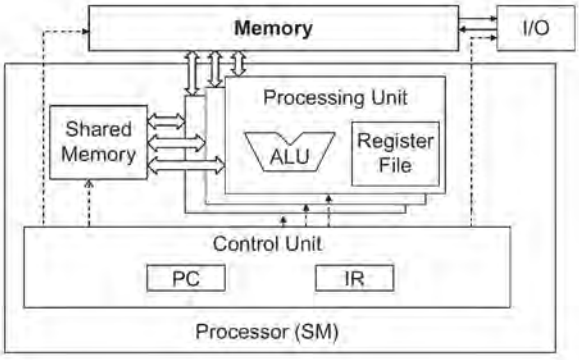
17 Diferentes aplicaciones tienen diferentes requerimientos. Comúnmente se utilizan $24Hz$ para codificar video, de $30Hz$ a $60Hz$ para el contenido de consolas de videojuegos y más de $90Hz$ en aplicaciones de realidad virtual

18 Rasterizar es el proceso de convertir un gráfico vectorial en un mapa de bits

masivamente paralelos, enmascarados como operaciones de gráficos en el *GPU*. Esta fue una tarea tediosa y compleja de implementar, ya que no se contaba con herramientas o lenguajes de programación que expusieran las abstracciones para comunicar y ejecutar rutinas en estos coprocesadores. Sin embargo, los resultados obtenidos fueron alentadores, logrando tiempos de ejecución cientos de veces más cortos que las implementación para *CPU*s. Estos esfuerzos dieron pie al nuevo paradigma *GPGPU Computing*, en donde los coprocesadores gráficos se convierten en procesadores de cómputo de propósito general.



(a) CPU



(b) GPU - SM

Figura 3.4: Comparación en la arquitectura de un núcleo de *CPU* y un núcleo de *GPU*
[Hennessy & Patterson, 2006]

Los *GPU*s forman parte de la trayectoria de diseño de arquitecturas de cómputo, denominada *Many Core Architecture*. Al igual que un *CPU*, un *GPU* puede contener múltiples procesadores, pero cada uno de estos está diseñado para integrar el mayor número de núcleos posibles. A estos procesadores se le denomina *Streaming Multiprocessor (SM)* y a cada uno de los núcleos *Streaming*

Processor (SP). Por diseño los núcleos que estos integran son esencialmente diferentes a los de un *CPU*, siendo más simples y limitados; gran parte de la lógica que tienen los núcleos de un *CPU* para predecir saltos y paralelizar instrucciones en el *pipeline* no existen en el *GPU*, pues todos los núcleos dentro de un procesador comparten el registro de control y *caché* de instrucciones.

Dado que los *GPU*s modernos pertenecen a la categoría *Single Instruction Multiple Threads (SIMT)* — una variante del grupo *SIMD* en la taxonomía de Flynn — tiene sentido compartir los recursos para el control de ejecución, ya que sólo una instrucción se utiliza para todos los distintos hilos de ejecución en un *SM*.

Un *GPU* moderno está diseñado exclusivamente para ser un motor de cómputo numérico *masivamente paralelo*, la mayoría de las tareas realizadas en un *CPU* no podrán ser ejecutadas eficientemente; incluso algunas son imposibles de ejecutar bajo las limitaciones actuales de los *GPU*s.[19,20,21]. Para ilustrar la diferencia en el grado de paralelismo entre estas dos arquitecturas, comparemos el *CPU* contra el *GPU* con mayor número de núcleos disponibles al momento en que se escribe este trabajo: El *CPU* E7-8894 v4 (Intel) contiene 24 núcleos, mientras el *GPU* GP100 (Nvidia) cuenta con 3584 núcleos. Esta cualidad es la que hace a un *GPU* tener una capacidad de cómputo paralelo órdenes de magnitud mayor a un *CPU*.

Una consecuencia de manejar una cantidad masiva de hilos de ejecución es el requerir de un sistema de memoria con suficiente ancho de banda, esto con el fin de lograr abastecer todas las peticiones de datos realizadas simultáneamente por cada uno de los procesadores. Por esta razón, los *GPU*s utilizan variantes de memoria *DDR*: *GDDR5* y *GDDR5X*, siendo las generaciones más actuales de esta tecnología. Estas variantes comparten la mayoría de las características de una memoria *DRAM*, pero utilizan interfaces de comunicación con tasas de transferencia y frecuencias de reloj mucho más altas.

19 En un *GPU* el marco de pila (*stack frame*) de cada hilo, vive en los registros del procesador; al estar compartidos entre múltiples hilos en la ejecución de un *kernel*, esto limita el uso de flujos recursivos o la capacidad de manejar excepciones por hilo de ejecución.

20 Existen limitantes importantes sobre el manejo de memoria, permitiendo sólo cierto tipo de alojamientos en partes específicas del flujo de un programa.

21 En contraste con la estandarización en los procesadores x86, cada una de las múltiples arquitecturas de *GPU* definen diferentes tipos de datos soportados, tamaños de memoria compartida, memoria constante, la organización de hilos en un *warp*, así como limitantes en la configuración de un *kernel*: el número máximo de hilos, las máximas dimensiones en un bloque y malla.

(a) Pascal - GP100 con 60 *SM*s

(b) Pascal - SM

Figura 3.5: Diagrama de bloque de un SM de micro-arquitectura GP100 [Whitepaper NVIDIA Tesla P100, 2016]

En la última generación de *GPU*s y *MIC*s[22], se está comenzando a utilizar memorias de circuitos integrados 3D: *Hybrid-memory cube*, *High Bandwidth Memory (HBM)* y su nueva versión *HBM2*. Estas tecnologías utilizan pilas verticales de *DRAM*, interconectadas utilizando vías a través del silicio. Estas tecnologías permite escalar la densidad de memoria que existe en un sólo dispositivo, permitiendo integrarlos dentro del mismo empaquetado del procesador. Al estar físicamente más cerca y no requerir de líneas externas de energía, estos tipos de memoria pueden alcanzar tasas de comunicación de hasta $256GB/s$ mientras consumen menos energía que *DDR*.

En el contexto de cómputo de propósito general en un *GPU*, a esta memoria se le llama *memoria global* y será el espacio de memoria sobre la cual se pueden realizar lecturas y escrituras desde las tareas ejecutadas en el *GPU*.

3.3 LENGUAJES Y MODELOS DE PROGRAMACIÓN PARALELA

Existen diferentes modelos y lenguajes de programación utilizados para poder implementar algoritmos paralelos. Cada tipo de paralelismo implica resolver problemas diferentes, debido a los recursos y restricciones que existen en cada contexto. Un aspecto importante al paralelizar una tarea, es definir la granularidad que será utilizada en la paralelización. La granularidad define la proporción entre el tiempo de cómputo y el tiempo de coordinación requerido para intercambiar información entre procesos. Describimos tres categorías de granularidad:

- Fina: Implica partir el problema en un gran número de pequeñas tareas asignadas a diferentes procesadores, permitiendo realizar balance entre cargas, pero con un alto costo en sincronización.
- Gruesa: Implica dividir el problema en un número reducido de tareas, cada una tomando tiempo significativo de cómputo. Aunque esto inhibe el poder realizar balance de cargas, disminuye la comunicación que existe entre procesos.
- Media: Es un nivel de granularidad intermedio, permitiendo una combinación de los beneficios de la granularidad fina y gruesa.

22 GP100 de Nvidia y Knights Landing de Intel

Estos son algunos de los modelos más relevantes en la actualidad:

- **MPI** (*Message Passing Interface*): Es utilizado en modelos donde se desea paralelizar sin la necesidad de memoria compartida. Toda coordinación, sincronización, así como intercambio de datos se realizar explícitamente mediante mensajes. La comunicación entre nodos es a base de conexiones de red, lo que implica alta latencia y bajo ancho de banda. Sin embargo, si el problema permite utilizar alto nivel de paralelismo con una granularidad gruesa; es posible escalar hasta cientos de miles de nodos de cómputo. Los nodos pueden ser de cómputo heterogéneo, cada uno teniendo *CPU*s y *GPU*s en un mismo sistema. Existe un nivel importante de complejidad implicada en portar algoritmos a este entorno, puesto que el algoritmo necesita realizar las labores de coordinación e integración de resultados, tomando en cuenta las restricciones de tráfico, velocidad y latencia que existen en una red de esta magnitud.

- **OpenMP**: Utilizado en modelos en donde se requiere explotar el paralelismo entre procesadores con memoria compartida. OpenMP se implementa como una extensión a los compiladores, permitiendo utilizar los recursos en una máquina con múltiples núcleos o procesadores para explotar paralelización de granularidad media. Los hilos de ejecución que se producen tienen un alto costo y sólo es práctico utilizarlos en rutinas de cómputo intensivo. Desde la versión 4.0 de *OpenMP* también se implementan expresiones para declarar secciones de instrucciones vectorizables, en donde se pueden utilizar las extensiones *SIMD* de un procesador.

- **OpenCL**: Nació como un esfuerzo para crear un modelo estandarizado de cómputo heterogéneo, utilizando extensiones en lenguajes de programación y un *API* que permite abstraer el manejo de memoria, así como la definición y ejecución de funciones paralelas. En contraste a los demás modelos, *OpenCL* nos permite definir y ejecutar programas de manera agnóstica a la arquitectura de la máquina objetivo, pudiendo ser un *CPU* con múltiples núcleos o cualquier arquitectura de *GPU*s, siempre y cuando soporten e implementen las extensiones de *OpenCL* utilizadas en el programa.

- **CUDA** (*Compute Unified Device Architecture*): Este modelo fue desarrollado por Nvidia, específicamente para su arquitectura de *GPU*s. A grandes rasgos los modelos de *CUDA* y *OpenCL* comparten gran parte de su diseño. La mayor diferencia es que *CUDA* expone las características

programables específicas a las arquitecturas de los *GPU*s de Nvidia, permitiendo crear implementaciones micro-optimizadas para explotar todos los recursos de cómputo, sincronización y comunicación disponibles en sus procesadores.

3.3.1 *CUDA*

En el modelo que implementa *CUDA* el sistema de cómputo está conformado por una máquina anfitrión, la cual está habilitada por uno o varios *CPU*s. Sobre esta máquina se conectan uno o varios coprocesadores utilizando PCI-e (*Peripheral Component Interconnect Express*). Estos son expuestos en el sistema anfitrión mediante la interfaz de *CUDA* como dispositivos de cómputo masivamente paralelo.

Al desarrollar una aplicación que utiliza las extensiones de *CUDA*, el ejecutable es indistinto del de otra aplicación. Incluso es posible que la misma aplicación puede ejecutar tareas paralelas utilizando *OpenMP* y *MPI*. El hilo de ejecución que inicia la aplicación en el *CPU* tendrá acceso a las funciones de *CUDA*, permitiendo iniciar y configurar el contexto de cómputo para los dispositivos. Esto puede involucrar buscar a los dispositivos disponibles, categorizarlos y seleccionar en base a su arquitectura o número de procesadores. Una vez seleccionados los dispositivos, se pueden realizar los alojamientos e inicialización de memoria en los *GPU*s activos.

Las funciones que configuran e invocan la ejecución en el *GPU* se les denomina *kernels*, los cuales están declarados de distinta manera a las funciones que sólo se ejecutan en el *CPU* de la máquina anfitrión. Para poder definir un *kernel* es necesario utilizar las extensiones de *CUDA*.

La definición de un *kernel* difiere en varios aspectos a la manera en que se define un función ordinaria de *C*. La primera, es el uso de la palabra clave de *CUDA* "`__global__`". Al incluir esta palabra clave, le estamos indicando al compilador que esta función debe de ser compilada para construir un *kernel*. Un *kernel* puede ser llamado desde el código del programa para iniciar una ejecución en el *GPU*. En las arquitecturas recientes de *Nvidia* con *Compute Capability 3.5+* es posible utilizar *Dynamic Parallelism*, permitiendo configurar y llamar nuevos *kernels* en el flujo de un hilo de ejecución de un *kernel* padre. De esta manera el código de un *kernel* puede generar más trabajo para el mismo *GPU*, permitiendo definir programas recursivos, realizar

nuevas transformaciones o aumentar la granularidad del proceso, en función de los datos.

El llamar un *kernel* implica la creación de una malla (*grid*), un arreglo de hilos de ejecución en el dispositivo seleccionado. La configuración que determina la cantidad de hilos y su organización no es parte de la definición de la función, si no de la manera en que se realiza la llamada al *kernel*. Esta *malla* está compuesta por *bloques* (*blocks*) que a su vez contienen *hilos* (*threads*) de ejecución. Dentro del contexto de un *kernel* existen variables implícitas pre-inicializadas, que identifican al hilo de ejecución y sus coordenadas dentro del *malla* al momento de ejecución: $blockIdx$, $threadIdx$.

Para ilustrar el contexto en el que se ejecuta un hilo al lanzar un kernel al *GPU* veamos el siguiente ejemplo:

```
1   // Función para calcular la suma de vectores de n dimensiones
2   //   suma c = a + b
3   void vecAdd(float* A, float* B, float* C, int n)
4   {
5     // Iteramos sobre las n dimensiones
6     for (i = 0; i < n; i++) {
7       C[i] = A[i] + B[i];
8     }
9   }
10
11  // Kernel para calcular la suma de vectores de n dimensiones
12  //   suma c = a + b
13  __global__
14  void vecAddKernel(float* A, float* B, float* C, int n)
15  {
16    // Cada hilo de ejecución calculará la suma
17    //   de una dimensión del vector
18    int i = threadIdx.x + blockDim.x * blockIdx.x;
19    if(i < n) {
20      C[i] = A[i] + B[i];
21    }
22  }
```

Figura 3.6: Comparación entre la definición de una función iterativa y la definición de un kernel paralelo

También existe otra palabra clave "`__device__`", en este caso se utiliza también para definir una función que será ejecutada en el *GPU*, pero que sólo puede ser llamada desde otra función de este mismo tipo o desde un *kernel*.

Recordemos que el paralelismo en un *GPU* es de tipo *SIMT*, esto quiere decir que al lanzar un *kernel* se están creando múltiples hilos

de ejecución, los cuales van a operar sobre las mismas instrucciones utilizando los mismos argumentos de entrada. La manera en la que se logra que cada hilo opere sobre datos diferentes, es definiendo en el código del *kernel* los índices de los datos en donde se opera, en función a los índices que identifican al hilo de ejecución dentro de la *malla* con la cual se llamó al *kernel*.

Un bloque iterativo sobre una variable de control, la cual recorre un rango de valores $[0, n)$ y en donde cada paso es independiente a los demás, se puede transformar en una llamada paralela a un *kernel*. La ejecución de un bloque iterativo sucede en pasos secuenciales, en donde cada paso define un valor sobre la variable de control. En contraste, al llamar un *kernel* cada posible valor de la variable de control define una instancia sobre un proceso de ejecución independiente. Ilustrando el ejemplo anterior Figura 3.6, se asigna el valor a la variable de control i utilizando el siguiente polinomio de direccionamiento Figura 3.7:

```
1 threadIdx.x + blockDim.x * blockIdx.x
```

Figura 3.7: Polinomio de direccionamiento dentro de un kernel

De esta manera se ejecutan todas las sumas sobre cada una de las dimensiones del vector de manera simultánea.

Dependiendo del dominio del problema tendrá sentido definir la configuración de los hilos de ejecución que se ajuste de manera óptima a la granularidad de paralelismo disponible y al orden en que están dispuestos los datos en memoria.

Tanto el arreglo de bloques como los bloquen en sí, son arreglos tri-dimensionales. A continuación se ejemplifica la definición de una *malla* y sus dimensiones, así como la dimensión de cada uno de los *bloques* y la llamada a un *kernel* con esta configuración.

```
1     dim3 dimBlock(2, 2, 4);
2     dim3 dimGrid(0, 2, 2);
3     KernelFunction <<<dimGrid, dimBlock>>> (...);
```

Figura 3.8: Llamada a un kernel

Dependiendo de la versión de la arquitectura del *GPU* existen restricciones sobre el número de dimensiones que se pueden declarar. Otro punto importante es la manera en que los procesadores del *GPU* ejecutan los *hilos*. Cada *bloque* que definimos es asignado a un procesador en el *GPU*, posteriormente el procesador subdivide cada *bloque* en *warps*. Un *warp*, es la unidad en la que un procesador del *GPU* puede lanzar hilos de ejecución paralela. El tamaño de un *warp* es dependiente de la arquitectura del *GPU* que se está utilizando, pero será de vital importancia para el rendimiento de un *kernel* el

utilizar *bloque*s con un número de *hilos* que sean múltiplos del tamaño del *warp* en la arquitectura objetivo. De no lanzar un múltiplo del tamaño de un *warp*, estos hilos son lanzados pero desperdiciados. Hasta la fecha todas las arquitecturas de *Nvidia* han utilizado *warps* de 32 hilos, pero esto podría cambiar en el futuro. Si se desea tener una implementación portable en arquitecturas futuras no se debe de asumir su valor.

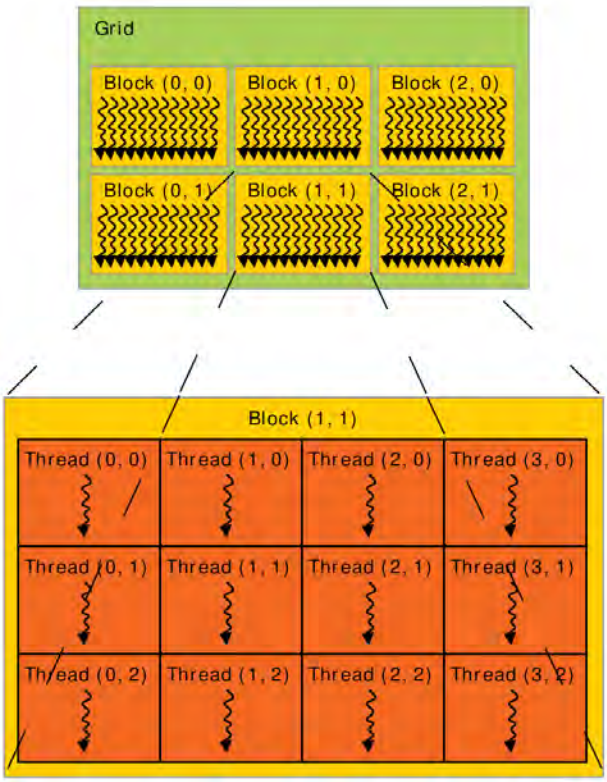


Figura 3.9: Configuración de hilos de ejecución en el GPU
[CUDA C Programming Guide, 2016]

Por defecto cada lanzamiento de un *kernel* es una operación síncrona. Esto quiere decir que el controlador del *GPU* iniciará la ejecución del *kernel* dejando en alto el hilo de ejecución del *CPU* hasta que la operación termine. Es posible lanzar *kernel*s de manera asíncrona y de hecho será un patrón común para alcanzar alta utilización en el *GPU*. El intercalar grupos de *kernel*s con otros *kernel*s o copias de memoria permite mantener los procesadores del *GPU* ocupados la mayor parte del tiempo.

3.3.2 *Tipos de Memoria*

En el ejemplo ilustrado en la Figura 3.6, todos los hilos de ejecución leen dos direcciones de memoria, de donde obtienen los

valores de los vectores a y b. Por último escriben a una dirección de memoria donde se encuentra el vector resultante c.

Estas direcciones de memoria viven en el espacio de *memoria global*. Como se mencionó anteriormente, este nivel de memoria está implementada utilizando *GDDR* y está localizada afuera del procesador, por lo que tiene alta latencia, y cualquier acceso implica desperdiciar ciclos de reloj para todos los *hilos* del *warp* en ejecución. A pesar de que los *GPU*s son capaces de paralelizar a gran escala, si los accesos a memoria no se realizan de una manera eficiente podríamos mantener detenidos a los procesadores. Mantener una proporción alta de operaciones de cómputo por cada acceso de memoria será vital para lograr una implementación de alto rendimiento en un *GPU*. Al igual que en los *CPU*s los *GPU*s cuentan con una jerarquía de memoria, con la cual se pueden mitigar los retardos ocasionados por estos accesos.

Las arquitecturas de *GPU*s de *Nvidia* actuales[23] cuentan con *caché* L1 y L2. Todo acceso a memoria global se copia a la memoria *caché*. Cada *cache-line* es de 128 bytes, al igual que en un *CPU* esto determina la unidad de tamaño de toda transacción a memoria global. Si datos previamente referidos están vivos en L1 y L2, entonces se puede realizar una transacción de 128 bytes al momento de volver a hacer referencia a ellos. Sin embargo, si los datos sólo están vivos en L2 las transacciones se reducen a 32 bytes.

Para explotar el principio de localidad espacial en un *GPU*, el patrón de accesos a memoria que realiza cada uno de los *hilos* dentro de un *warp*, necesitan ser contiguos y alineados entre los diferentes hilos. La idea es que todo acceso dentro de un *warp* esté encapsulado en un *cache-line* para reducir el número de transacciones necesarias.

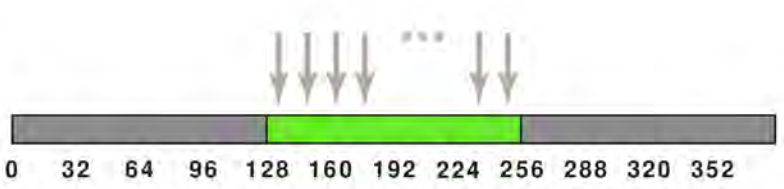


Figura 3.10: Accesos de memoria alineados con un cache-line, requiriendo sólo una transacción de 128 bytes

Otro tipo de memoria del cual se puede disponer en un *GPU* es la *memoria constante*. Esta memoria está restringida a 64KB por *GPU* y como su nombre lo indica, no es posible modificar su contenido dentro de un *kernel*. La ventaja de utilizar esta memoria es que tiene baja

23 Fermi, Kepler, Pascal

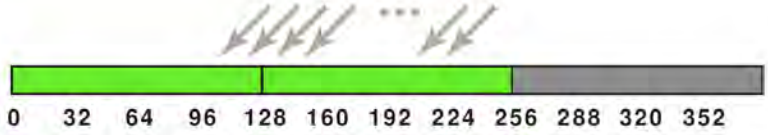


Figura 3.11: Accesos de memoria desalineados dentro de L1 y L2, requiriendo dos transacciones de 128 bytes

Figura 3.12: Accesos de memoria desalineados dentro de L1, requiriendo 5 transacciones de 32 bytes

latencia y alto ancho de banda, además de poder realizar emisiones a todos los hilos de ejecución en un *warp* de manera paralela.

En el nivel más bajo de la jerarquía tenemos la *memoria compartida* y los registros. Al igual que los registros de un *CPU* se encuentran integrados dentro del *CPU* como memoria *SRAM* y sus accesos son los más rápidos de toda la jerarquía de memoria. Los registros son asignados de manera privada a los diferentes hilos de ejecución, donde cada hilo guardará el contexto de las operaciones que realiza. Es importante tratar de mantener un número reducido de registros por hilo de ejecución, de tal manera que al lanzar la ejecución del *kernel* existan suficientes registros para atender a todos los hilos de manera paralela.[24] La memoria compartida es reservada por bloque de hilos, y es el medio por el cual se puede tener cooperación entre los diferentes hilos de un bloque. Esta memoria está dividida en grupos denominados *banks*, a los cuales se puede tener acceso de manera paralela. Si se intenta acceder a un mismo banco desde varios hilos de ejecución se crea un conflicto, realizando los accesos al banco de memoria de forma secuencial. Las arquitecturas actuales implementan 32 bancos de memoria. Si se logra explotar el uso de la memoria compartida para evitar accesos a memoria global, implementando un patrón que evite conflictos entre bancos, esto puede incrementar el rendimiento del algoritmo de forma considerable.

24 En el procesador GP100, de la arquitectura más reciente de Nvidia, se cuentan con 256KB de espacio en registros y 64KB de memoria compartida

4

DISEÑO E IMPLEMENTACIÓN

The purpose of all programs and all parts of those programs is to transform data from one form to another. If you do not understand the data you do not understand the problem.

— *Mike Acton, 2014*

En este capítulo se describen los puntos más importantes en el diseño de una aplicación de alto rendimiento, y como se utilizaron algunas de estas estrategias en la implementación y optimización del código de *Locusta*.

4.1 DISEÑO ORIENTADO A DATOS (DOD)

Todo programa está compuesto por instrucciones que definen transformaciones entre un conjunto de datos. Comenzando por las instrucciones del programa, estas tienen que ser extraídas de su respectivo lugar en memoria y pasar por un proceso de decodificación antes de poder ser consumidas por el procesador. De igual manera, los datos que serán transformados tienen que ser leídos de la memoria y cargados en los registros del procesador para ser manipulados. Por lo tanto, todo proceso implica interacciones entre el procesador y el sistema de memoria, siendo este el principal cuello de botella de una arquitectura de cómputo.

En las primeras generaciones de computadoras, la velocidad a la cual operaban los procesadores era equivalente a la velocidad con la cual accedían a los datos en memoria. Por ello, el realizar operaciones en memoria no inducía una penalización tan significativa en latencia. Actualmente la frecuencia a la que trabajan los procesadores es de órdenes de magnitud mayor a la velocidad en que se pueden efectuar las transacciones de memoria. Cada vez que hay una dependencia de datos en el flujo de un programa, el procesador no podrá continuar realizando operaciones hasta que concluya la transacción y los datos estén disponibles. A este estado en espera del procesador se le conoce como *starvation*. Sin importar qué tan rápido puede realizar operaciones un procesador, el desempeño de un programa siempre estará limitado por la velocidad a la cual se pueden realizar los accesos a memoria.

El diseño orientado a datos (*DOD*) es una estrategia utilizada en el desarrollo de software dirigido a entornos en donde existen restricciones sobre los recursos de cómputo disponibles y tiempo de ejecución, necesitando de un diseño especializado que logre cumplir

con los requerimientos de la aplicación. En una implementación que sigue un *DOD* se entiende que las características del hardware forman parte crucial del problema que se está resolviendo, guiando el diseño de los algoritmos y estructuras de datos que serán utilizados, así como estrategias *ad-hoc* a la micro-arquitectura del hardware en dónde se va a ejecutar el programa.

El *DOD* no es un concepto nuevo, dadas las importantes limitaciones en los recursos de memoria y cómputo de las primeras generaciones de computadoras, esta estrategia permeaba todo el desarrollo de aplicaciones computacionales. Con el tiempo, las nuevas generaciones de sistemas de cómputo contaron con mayor cantidad de recursos, y los obstáculos que habían sido un impedimento para el desarrollo de aplicaciones más complejas fueron desapareciendo. A partir de ello se crearon nuevas herramientas, como: compiladores y lenguajes de programación de alto nivel, los cuales a su vez ocultaron los detalles del hardware y con ello brindar la posibilidad de implementar programas que fueran más versátiles y portátiles entre diferentes arquitecturas de cómputo.

El contar con esta clase de herramientas resulta ser una gran ventaja en la productividad para los desarrolladores de software, ya que permite enfocarse en la implementación correcta de los algoritmos y estructuras de datos, delegando a los compiladores o intérpretes los detalles sobre cómo el hardware va a realizar estas operaciones. Es importante estar conscientes que estas abstracciones tienen un costo; entre menos información tengan los compiladores sobre los datos y sus transformaciones, menos inferencias pueden realizar, lo que conlleva a que la capacidad de realizar optimizaciones sea limitada. En ciertos contextos, como es el caso de *HPC*, donde el rendimiento tiene prioridad sobre la productividad, es necesario codificar el programa de manera descriptiva: para cada una de las transformaciones del programa se indica el tipo de dato, dependencias entre otros datos, si las variables hacen referencia a datos que pueden ser modificados durante la ejecución, etc. El desarrollador deberá verificar el código que produce el compilador y realizar un análisis sobre el número de instrucciones que está utilizando, y con ello corroborar que sea una secuencia de instrucciones eficiente para efectuar las transformaciones necesarias. En dado caso de que el compilador no esté logrando generar instrucciones optimizadas, el desarrollador puede utilizar funciones intrínsecas que permiten al compilador emitir instrucciones específicas a la arquitectura. Si aún así el compilador no logra generar el número de instrucciones esperadas, el desarrollador puede utilizar directamente lenguaje ensamblador, fijando el orden y las instrucciones óptimas para la arquitectura objetivo.

Los compiladores modernos tienen gran capacidad de inspección sobre el código de un programa, si el desarrollador provee suficiente información sobre las operaciones y los datos, así como datos específicos sobre la arquitectura objetivo, el compilador podrá realizan optimizaciones adicionales sobre las instrucciones generadas, así como reordenar las operaciones para obtener una mejor utilización de los recursos por su propia cuenta. Un compilador puede también optimizar la utilización de registros e instrucciones específicas a la arquitectura, sin necesidad de que el desarrollador los codifique explícitamente. Sin embargo, un compilador sólo puede tomar decisiones en base a la información disponible en el código, sin poder asumir condiciones sobre el contexto disponible durante su ejecución. Un compilador no podrá cambiar el esquema que utilizamos en nuestras estructuras de datos, la manera en que manipulamos la memoria, ni mucho menos los algoritmos que definimos en el código.

Dependiendo de la naturaleza del problema, el desarrollador tendrá que tomar ciertas decisiones durante el diseño de la implementación. Algunas tienen que ser definidas de manera previa por ser parte fundamental del problema y juegan un papel esencial en los algoritmos y estructuras de datos que serán utilizados, otras decisiones serán parte de micro-optimizaciones con un alto costo en productividad, que sólo tienen que ser realizadas una vez que se hayan hecho mediciones sobre las secciones críticas de un programa. Un *DOD* bien planificado, permite que podamos delegar a nuestras herramientas la mayor parte del trabajo al estar informadas sobre el contexto de las transformaciones, pero también nos da un profundo entendimiento sobre la arquitectura en la que estamos programando, la operación del programa en el hardware y las limitantes de nuestras herramientas; permitiendo realizar estrategias de macro y micro optimización en las diferentes etapas del proyecto.

Cada arquitectura de cómputo es diferente y por lo tanto se requieren de distintas estrategias al diseñar implementaciones específicas para cada caso. Existe documentación sobre las arquitecturas más comunes, así como de los principales compiladores y herramientas para medir rendimiento. Es importante conocer esta documentación y utilizarla como referencia al tomar decisiones sobre los detalles de implementación de cada arquitectura. [1] [2] [3]

1 https://developer.nvidia.com/cuda-zone/
2 http://developer.amd.com/resources/developer-guides-manuals/
3 http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

4.2 OPTIMIZACIÓN

El primer paso para mejorar el rendimiento de un programa es entender qué secciones del código pueden ser optimizadas y qué beneficios se pueden esperar de mejorar cada una de las rutinas. Prácticamente todo programa cumple con los principio de localidad, tanto para sus instrucciones como los datos que manipulan, por ello van a existir secciones que tendrá más valor optimizar que otras. De manera general, los programas tienden a correr el 90 % del tiempo de ejecución sobre secciones que corresponden al 10 % del código [Hennessy & Patterson, 2006], siendo estas secciones las candidatas para realizar trabajos de optimización.

La *ley de Amdahl* describe el beneficio que podemos esperar después de implementar una mejora a cierta sección de una tarea de cómputo correspondiente a una fracción del tiempo de ejecución. [Amdahl, 1967]

Definición 13. *Se define el* speedup *de una tarea de cómputo como la razón entre tiempo que toma ejecutar la tarea (T) y el tiempo que toma ejecutar la misma tarea implementando una mejora (T'):*

$$Speedup = \frac{T}{T'} \tag{4.1}$$

Una vez que se han realizado mediciones que capturen los tiempos de ejecución de las diferentes secciones del programa, podemos utilizar la *ley de Amdahl* para tener una idea clara de qué secciones conviene optimizar y qué resultados podemos esperar de dichas mejoras.

Definición 14. *Siendo T_{global} el tiempo de ejecución del programa original, definimos T'_{global} como el tiempo de ejecución después de implementar mejoras sobre una sección del programa que corresponde a $Fracción_{con\ mejora}$ del tiempo total de ejecución, y el Speedup de dicha sección como $Speedup_{Fracción_{con\ mejora}}$.*

$$T'_{global} = T_{global} \times \left((1 - Fracción_{con\ mejora} + \frac{Fracción_{con\ mejora}}{Speedup_{Fracción_{con\ mejora}}}) \right) \tag{4.2}$$

$$Speedup_{global} = \frac{T_{global}}{T'_{global}} = \frac{1}{\left(1 - Fracción_{con\ mejora} + \frac{Fracción_{con\ mejora}}{Speedup_{Fracción_{con\ mejora}}}\right)} \tag{4.3}$$

Una manera en que se puede mejorar el tiempo de ejecución de una tarea es mediante paralelización. Utilizando la ley de Amdahl se puede describir el *speedup* esperado al utilizar n procesadores para

ejecutar una sección de código paralelizable. Asumiendo el caso ideal, en donde n procesadores pueden ejecutar la sección paralela en T/n y la sección secuencial se ejecuta en tiempo $1-T$:

$$T_{paralelo} = (1 - T + \frac{T}{n}) \tag{4.4}$$

$$Speedup_{paralelo} = \frac{1}{(1 - T + \frac{T}{n})} \tag{4.5}$$

En cada algoritmo puede existir interdependencia entre cada uno de sus pasos, dando lugar a secciones que se podrán ejecutar en paralelo y otras que deberán ser ejecutadas de manera secuencial. Los puntos de sincronización se definen como aquellas barreras entre operaciones con dependencias de datos, debiendo integrar los resultados dentro de la dependencia antes de poder continuar con pasos subsecuentes en el algoritmo.

Paralelizar una tarea siempre implica dividir los datos y sus operaciones en particiones que pueden ser procesados de forma simultánea, así como crear y coordinar los procesos que van a realizar las operaciones sobre los datos. Estos pasos utilizan recursos y tiempo del procesador, lo que significa que paralelizar una sección de código siempre induce una penalización, comúnmente llamado *overhead*. Dependiendo del problema que se desea paralelizar y la estrategia que se implemente, pueden existir circunstancias que requieran más colaboración y comunicación entre procesos, como también la contención que estos generan sobre los recursos de memoria e *I/O*, incrementando la magnitud del *overhead* conforme aumenta el número de procesos que se ejecutan simultáneamente.

Podemos añadir el término respectivo al *overhead* (O_n) a nuestra representación del *speedup* paralelo para el caso de n procesos.

$$Speedup_{paralelo} = \frac{1}{(1 - T + \frac{T}{n})} - O_n \tag{4.6}$$

Por estas razones, al momento de diseñar una implementación se debe de elegir cuidadosamente las secciones que conviene paralelizar, el número de procesos concurrentes y los algoritmos utilizados, buscando maximizar el beneficio de una ejecución paralela y minimizando el costo implícito en la paralelización.

La ley de Amdahl nos dice que sin importar qué tan eficientemente logremos paralelizar ciertas secciones de un algoritmo,

existe una cota mínima en el tiempo de ejecución que determinan las partes secuenciales del mismo. Esta conclusión da un panorama pesimista de lo que pueden ofrecer las nuevas arquitecturas de cómputo para acelerar las implementaciones existentes diseñadas originalmente para ser ejecutadas de forma secuencial.

En contra parte, la ley de Gustafson describe cómo al aumentar los recursos de cómputo disponibles podemos incrementar el tamaño de los problemas. Por lo tanto, aunque sea imposible ejecutar en menor tiempo las secciones secuenciales, podemos crecer la fracción del programa que puede ser procesada de forma paralela.

$$Speedup_{paralelo} = 1 - T + nT - O_n \tag{4.7}$$

De cualquier manera, para utilizar de forma eficiente los recursos de cómputo disponibles en nuevas generaciones, es necesario crear nuevos algoritmos y estructuras de datos que nos permitan atacar problemas utilizando los nuevos paradigmas de cómputo.

En cada problema van a existir diferentes niveles a los cuales se pueden paralelizar las transformaciones de los datos. A estos niveles se les conocen como *granularidad*. Cada arquitectura de cómputo paralelo tiene capacidad de explotar diferentes tipos de granularidad, debiendo adaptar los algoritmos y estructuras de datos para que se puedan realizar las operaciones que provee la micro-arquitectura, así como la topología de los recursos de cómputo y la jerarquía de memoria. Como veremos más adelante, estos niveles no son mutuamente exclusivos, es posible paralelizar en diferentes dimensiones una misma implementación.

4.2.1 *Optimizaciones para CPUs*

4.2.1.1 *Patrones de acceso, Caching & Prefetching*

Como se describió en el capítulo anterior, los procesadores modernos utilizan arquitecturas orientadas a *caché* y utilizan varias estrategias para poder mitigar la latencia de los accesos a memoria. Para poder sacar provecho de estas características y lograr obtener un mayor rendimiento, es necesario mantener patrones de acceso a memoria que efectúen transacciones de manera eficiente, sirviéndose de cada nivel de *caché* en la jerarquía de memoria para reutilizar datos que ya fueron leídos anteriormente, así como evitar que estos datos sean desalojados si van a ser requeridos próximamente.

Analicemos con detalle el ejemplo de realizar un recorrido sobre un arreglo bi-dimensional de valores de punto flotante con precisión simple (32-bits por valor), en donde los datos están alojados de manera contigua y alineada en la memoria física del sistema. En principio el recorrer dicho arreglo elemento por elemento tendría el mismo costo independientemente del orden en que se realice la iteración. Sin embargo, en la práctica este no es el caso, ya que el patrón que describe las referencias a los datos determina el rendimiento de la operación en una arquitectura orientada a *caché*. [4].

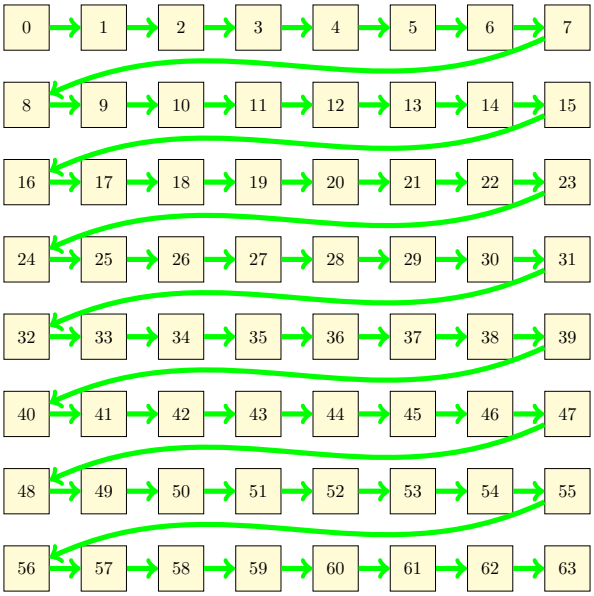


Figura 4.1: Barrido por renglones (Row Major)

En el recorrido mostrado en la Figura 4.1, se comienza iterando sobre el primer renglón del arreglo, recorriendo cada una de las columnas del renglón. Al llegar a la última columna se repiten los pasos sobre el renglón subsecuente. Las operaciones que realizará el procesador en este recorrido son las siguientes: cuando el programa hace referencia al primer elemento del arreglo, el controlador de memoria realizará una petición para obtener los datos de la localidad de memoria a la que pertenece este elemento, gestionando una transacción por el respectivo *cache-line*, obteniendo 64 bytes de memoria en la misma petición. Por esta razón, al recibir los datos del primer elemento en el arreglo también contamos con 3 valores

4 El orden en que se almacenan en memoria los elementos de un arreglo multidimensional es dependiente del lenguaje de programación o el tipo de direccionamiento que se implemente para codificar el arreglo multidimensional sobre memoria lineal. Para este ejercicio se describen los arreglos como serían almacenados al programar en los lenguajes *C* o *C++*, los cuales utilizan un orden por renglones (*row-major*). En contraste, en el lenguaje *Fortran* el orden de almacenamiento se realiza por *columnas* (*column-major*)

adicionales precargados en el *caché* $L1$, valores que el procesador puede ahora consumir sin requerir de nuevas interrupciones, aprovechando todos los ciclos disponibles para las próximas instrucciones. Al llegar al cuarto elemento se vuelve a repetir el proceso, adquiriendo cuatro nuevos valores por transacción. En algunas micro-arquitecturas de *CPU*s orientados a aplicaciones *HPC* se puede utilizar la función de *prefetching* de manera explícita, mediante instrucciones que gestionan nuevas transacciones de memoria a referencias que serán utilizadas próximamente, con el fin de maximizar la utilización del *pipeline* de la micro-arquitectura para efectuar estas transacciones mientras se está ejecutando el cómputo en las unidades aritméticas. También existen micro-arquitecturas en donde el procesador es capaz de detectar patrones simples en los accesos a memoria de manera automática, de modo que puede efectuar operaciones de *prefetching* de forma especulativa y explotar el paralelismo a nivel de instrucciones. Nuevamente, estas ventajas sólo serán visibles si la implementación de la aplicación aprovecha el uso de dichos patrones. En caso contrario, podría incluso resultar contraproducente inducir lecturas innecesarias, y por ende provocando el reemplazo de instrucciones o datos útiles en la jerarquía de *caché*.

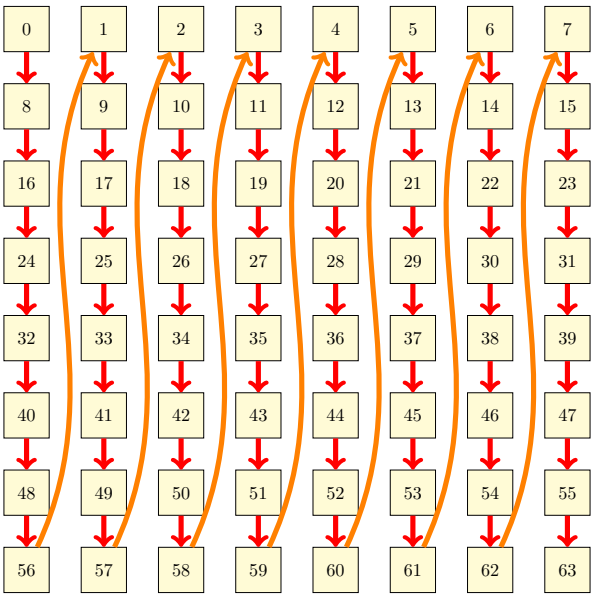


Figura 4.2: Barrido por columnas (Column Major)

En el recorrido alterno de la Figura 4.2, manteniendo el mismo esquema de datos en memoria se muestra otro orden en el recorrido, esta vez iniciando la iteración sobre cada elemento de la primera columna del arreglo. Al llegar al último renglón de la primera columna se continua en la segunda. De manera idéntica al caso anterior, el programa comienza haciendo referencia al primer elemento del arreglo,

obteniendo cuatro valores contiguos en memoria, esto corresponde a valores sobre el mismo renglón. Para el siguiente paso en la iteración, el procesador necesita el valor en el segundo renglón de la primera columna, pero el dato aún no se encuentra en *caché*, teniendo que realizar una nueva petición a memoria en cada paso.

Ambos recorridos utilizan la misma estructura de datos y describen un algoritmo de la misma complejidad sobre el tamaño del arreglo ($\mathcal{O}(n)$). Sin embargo, los detalles de la arquitectura determinan que uno de los patrones de acceso será mucho más eficiente que el otro. En un procesador moderno[5], dada la disposición de los datos y alineamiento con el tamaño del *cache-line*, el recorrer el arreglo por columnas resulta ser hasta $10x$ más lento que recorrerlo por renglones. En micro-arquitecturas de generaciones anteriores[6] la diferencia era aún más contrastante, teniendo una penalización de $40x$ al recorrer por columnas[7].

Los ejemplos anteriores muestran lo importante que es utilizar estructuras de datos en las cuales se pueda iterar de manera eficiente un conjunto de datos determinado. Cada problema define un conjunto de datos distintos, y para ciertas aplicaciones es necesario ajustar o rediseñar los algoritmos y estructuras para poder compactar los datos de forma alineada y contigua, evitando saltos en direcciones para maximizar la utilización de cada transacción de memoria y la localidad en los niveles de *caché*. Además, si las estructuras de datos cumplen con todos los puntos anteriores se puede hacer uso de instrucciones *SIMD* en el código, de manera que se puedan efectuar lecturas, operaciones y escrituras en forma paralela sobre múltiples valores en el mismo número de instrucciones, aumentado de manera significativa la eficiencia de las transformaciones. Bajo las condiciones anteriormente descritas, un multiprocesador *Intel* de última generación [8] puede efectuar hasta 32 operaciones de punto flotante en cada ciclo de reloj.

Existen otras características en cada micro-arquitectura las cuales también resultan de suma importancia en el diseño de implementaciones de alto rendimiento: número de núcleos por procesador, número de hilos de ejecución por núcleo, número de canales de memoria disponibles por procesador, el tamaño de las operaciones de memoria, número de transacciones concurrentes,

5 Intel Haswell

6 Intel Nehalem

7 Las nuevas generaciones de arquitecturas de procesadores implementan mejoras que mitigan la magnitud de esta penalización, al incluir volúmenes de *cache* con mayor capacidad, más niveles de *cache* en la jerarquía de memoria, así como también mejoras en los controladores de memoria y los canales de comunicación

8 Haswell, Broadwell y Skylake

ancho de banda, latencias, instrucciones *SIMD* y el número de niveles de *caché*, el tamaño de cada uno y la asociatividad de estos.

Utilizar multiprocesadores nos ofrece potencialmente mayor capacidad de cómputo, pero es importante comprender que en estas micro-arquitecturas la mayor parte de los recursos son compartidos entre los diferentes núcleos que integra el procesador: último nivel de *caché* (LLC), canales de memoria y canales de *I/O*. Esto implica que cuellos de botella que existían para un solo procesador ahora van a tener mayor contención, por lo que es de vital importancia cuidar la manera en que estos son utilizados para poder lograr escalar el rendimiento del programa al paralelizarlo. La cuestión primordial seguirá siendo cómo poder obtener los datos de memoria de la manera más eficiente para lograr mantener a todos los núcleos ocupados la mayor parte del tiempo.

El portar una implementación a una arquitectura de multiprocesadores con memoria compartida para ser ejecutada de manera concurrente puede introducir ciertos riesgos de coherencia en memoria, así como problemas de eficiencia. La manera en que la implementación coordina las interacciones entre procesos y resuelve los posibles conflictos de contención, influye de manera significativa en el rendimiento de la aplicación. Un ejemplo de un problema de contención que afecta el rendimiento es el fenómeno conocido como *false sharing*[Sutter, 2009], en donde dos núcleos están realizando modificaciones a localidades de memoria distintas pero que comparten el mismo *cache-line*. La arquitectura del sistema tendrá que detener cada una de las transacciones en memoria y retransmitir los *cache-lines* en cuestión para mantener la coherencia a lo largo de todos los niveles de *caché*. A pesar de que el algoritmo puede ser libre de *condiciones de carrera*[9], el hecho utilizar patrones de acceso que impliquen condiciones de *false sharing* en la arquitectura va a irrumpir el rendimiento de la ejecución. Para evitar estas condiciones, el desarrollador tiene que evitar realizar modificaciones a las mismas localidades de memoria que están siendo utilizadas por los otros hilos en ejecución. La solución más directa será siempre utilizar variables locales al hilo de ejecución para acumular resultados, escribiendo en localidades con alta contención solamente en los puntos de sincronización en donde es necesario realizar una integración o reducción de los resultados.

9 En un programa surge una condición de carrera (*race condition* en inglés) cuando existe una dependencia en la secuencia de ejecución de varios procesos concurrentes. Se les denomina *condiciones críticas de carrera* cuando los procesos concurrentes comparten recursos, requiriendo que toda operación sobre los recursos compartidos esté definida dentro de una *sección crítica*, cuya ejecución entre procesos concurrentes tenga exclusión mutua.

La regla general para evitar conflictos de memoria en implementaciones concurrentes es utilizar memoria compartida sólo si los datos accesibles por todos los procesos es inmutable. Datos que pueden ser alterados durante la ejecución deberán ser aislados para que sólo vivan en el contexto del mismo hilo de ejecución. Los problemas que requieren alteraciones a los datos compartidos necesitarán la introducción de controles de flujo y puntos de sincronización para lograr mantener un estado consistente de la memoria. Esta sincronización puede tener una penalización alta en el rendimiento de cómputo, ya que cada barrera será un punto de contención el cual requiere utilizar *operaciones atómicas*[10] para controlar la ejecución de secciones críticas de código. Si al diseñar una implementación paralela se evitan introducir las barreras y puntos de sincronización se pueden alcanzar mayores niveles de rendimiento, además de reducir la complejidad del programa y facilitando el mantener una implementación correcta del algoritmo.

En el caso de las meta-heurísticas implementadas en este trabajo, cada uno de los pasos del algoritmo definen una dependencia sobre cada una de las transformaciones, de manera que cada individuo tiene que ser evaluado antes de poder continuar con los operadores de selección, a su vez tiene que ocurrir el proceso de selección para poder proseguir a los operadores sociales o de recombinación. Por último, el proceso de cada generación tiene que concluir para poder iniciar las transformaciones de la siguiente generación. Cada uno de estos pasos definen barreras sobre las transformaciones aplicadas sobre una población.

Sin embargo, las transformaciones sobre cada individuo pueden ocurrir de manera independiente si separamos los datos que definen al conjunto poblacional original de los nuevos datos que describen a los individuos posteriores a las transformaciones. De esta manera, para una población de tamaño n mantenemos dos arreglos de tamaño estático n sobre los cuales vamos alternando: el primer conjunto referente a la población actual y el segundo describe a la población de la siguiente generación. Esta estrategia nos permite mantener inmutabilidad sobre el conjunto de la población original sobre la cual se van a realizar las transformaciones.

En la implementación de *Locusta* se define a cada uno de los individuos de la población como un arreglo de k variables de punto flotante, cada variable de 32 o 64 bits dependiendo la precisión utilizada. Cada conjunto poblacional está definido por los n individuos, de los cuales podemos utilizar un número i de islas

10 Una operación atómica es ejecutada por un solo proceso, de manera en que ningún otro proceso pueda intervenir y cambiar el estado de la operación.

poblacionales[11]. Con esta estrategia garantizamos mantener segmentos de memoria contigua sobre todo el conjunto poblacional, independientemente de la granularidad sobre la cual se va a operar.

4.2.1.2 *Selección de Granularidad*

La disposición de los datos en memoria de cierta manera determinan los niveles de granularidad con los cuales podemos paralelizar de manera eficiente. Dado el esquema de memoria utilizado en *Locusta* se puede paralelizar en varios niveles.

Primero en el nivel más fino, el tener los datos de cada uno de los individuos dispuestos de manera contigua, hace sumamente eficiente el proceso de iterar a lo largo del conjunto de variables, habilitando varias estrategias para explotar los aspectos de la arquitectura de un *CPU*. Los accesos a memoria están aprovechando eficientemente el contenido de cada *cache-line*, además de poder realizar *prefetching* sobre los siguientes elementos que se van a utilizar en iteraciones próximas. Finalmente, utilizamos instrucciones *SIMD* para explotar el paralelismo a nivel de datos, al efectuar las operaciones comunes a todas las variables en un solo paso. Esto se compone con los beneficios de optimización de un compilador que explota el paralelismo a nivel de instrucciones para mitigar la latencia de los accesos a memoria mientras se están computando los datos. Todos estos beneficios mejoran el rendimiento de cada hilo de ejecución, que se podrá ver reflejado en todos los niveles de granularidad subsecuentes.

La idea original era paralelizar la evaluación de la función objetivo, lo cual se beneficia del mismo esquema de memoria descrito anteriormente. Sin embargo, todas las transformaciones efectuadas al nivel de individuos pueden ser ejecutadas de manera paralela utilizando múltiples hilos de ejecución. Como el levantar múltiples procesos y coordinarlos tiene un costo adicional, el tiempo de vida de cada hilo y la densidad de cómputo que involucran tiene que ser suficiente para mitigar el costo de su creación. El punto clave es distribuir eficientemente el conjunto de individuos para poder mantener a todos los hilos de ejecución ocupados. La implementación de *Locusta* minimiza este costo al utilizar instrucciones que indican al compilador el desenroscado los ciclos iterativos para dividir la carga de cómputo equitativamente a lo largo de los procesos disponibles en tiempo de ejecución; el ciclo exterior siendo el que itera sobre poblaciones y el del ciclo interior el que itera sobre los individuos de cada población.

11 Estas variables de configuración son definidas al momento de crear una instancia de la meta-heurística

4.2.2 *Optimizaciones para GPUs*

Prácticamente todos los conceptos mencionados en las estrategias para *CPU*s también son aplicados en el contexto de los *GPU*s. Sin embargo, existen diferencias elementales en el diseño entre las dos arquitecturas, por lo que los detalles de implementación y las estrategias para optimizar cada aspecto serán distintas.

4.2.2.1 *Patrones de acceso y Caching*

Por la naturaleza de un *GPU*, existen múltiples y mayores limitantes sobre los patrones de acceso a memoria que pueden realizar cada uno de los hilos de ejecución de un programa. Los patrones de acceso a memoria, así como los puntos de divergencia en el código van a determinar el nivel de paralelismo. El utilizar patrones y accesos a memoria de manera eficiente será un aspecto primordial para habilitar la ejecución masivamente paralela.

Recordemos que en un *GPU* se lanzan hasta 32 hilos de ejecución en grupos llamados *warps*, los cuales son ejecutados en un mismo *stream processor* y comparten un solo registro (PC) para determinar la instrucción que se podrá ejecutar sobre todos los hilos en el *warp* de manera paralela. La arquitectura está diseñada para realizar eficientemente transacciones de memoria y operaciones asumiendo lo siguiente: los datos deben de estar dispuestos de manera contigua y alineada en la memoria, intercalando los datos que accede cada uno de los hilos. En caso de no cumplir con estas características, cada acceso de memoria dentro del *warp* se efectuará de manera secuencial, deteniendo a todos los hilos restantes hasta que las transacciones pendientes hayan concluido. Este requerimiento implica cambiar el esquema de las estructuras de datos anteriormente utilizadas para el *CPU* y la manera en que cada hilo de ejecución colabora al cómputo de la transformación. Por esta razón el tamaño de cada nivel de *caché* y los *cache-lines* son distintos en un *GPU*. Para el primer nivel en la jerarquía (L1) un *cache-line* es de sólo 128-bytes, lo equivalente a que cada hilo en un *warp* realice una lectura de 32-bits. Para el segundo y último nivel de *caché* (L2) cada transacción es de 32-bytes. (véase Sección 3.3.2) Esto permite ante un *cache-miss* en L1, relativo a una porción de hilos dentro del *warp*, que se logren restaurar sus valores haciendo transacciones pequeñas sin necesidad de irrumpir en los datos que ya existían en L1 para los demás hilos.

En cada nueva generación de procesadores *Nvidia*, se ha optimizado la implementación de la jerarquía de memoria para

relajar las restricciones anteriores y permitiendo realizar patrones de acceso más versátiles, evitando penalizaciones altas en el rendimiento de aplicaciones que lo requieran. Sin embargo, por el diseño de la arquitectura de un *GPU*, los algoritmos implementados tienen que poder hacer uso del modelo de memoria antes descrito.

4.2.2.2 *Utilización y Granularidad*

En contraste a las arquitecturas de *CPU*s, los *GPU*s están diseñados para manejar un nivel de paralelismo masivo, en donde lanzar hilos de ejecución tiene un *overhead* insignificante. Sin embargo, esto no necesariamente implica que debemos de utilizar la granularidad más fina, ya que existen múltiples factores que determinan el rendimiento de estos hilos de ejecución. Al igual que en el caso de los *CPU*s, entre más hilos se ejecuten simultáneamente la contención por los recursos compartidos del sistema será mayor, particularmente el sistema de memoria.

Los dos niveles de *caché* son compartidos entre todos los hilos, por lo que a mayor número de hilos simultáneos menos espacio estará disponible para cada hilo y posiblemente involucrará el reemplazo prematuro de datos que podían ser reutilizados en la jerarquía de memoria que comparten todos los núcleos.

En cada *stream processor* existe un número limitado de registros, los cuales son distribuidos entre todos los hilos en un *warp* que serán ejecutados en un respectivo procesador. En caso de que no existan suficientes registros para abastecer a todos los hilos, puede ocurrir lo que se denomina *spills* de registros. En una condición en donde ocurre un *spill*, los datos serán transferidos temporalmente a los siguientes niveles de la jerarquía de memoria, introduciendo instrucciones adicionales al código para poder realizar estas manipulaciones en memoria, además del costo implícito en efectuar este movimiento en memoria y el costo futuro al tener que volver a cargar los datos hacia los registros.

En el caso de los *CPU*s, el costo de utilizar instrucciones de saltos condicionales se puede ocultar gracias a que los procesadores implementan ejecución especulativa para aplicar predicción de saltos. Este no es el caso para los *GPU*s, donde ante cualquier salto condicional sólo los hilos que cumplan la condición serán ejecutados mientras los demás hilos dentro del *warp* se detienen, reduciendo el nivel de paralelismo efectivo en tiempo de ejecución. Para lograr una paralelización eficiente se requiere mantener a todos los hilos dentro de cada *warp* ocupados, evitando ramas en el flujo, además de

efectuar transformaciones con alta densidad de cómputo por cada referencia a memoria global.

Si el algoritmo necesita realizar operaciones de reducción sobre un conjunto de datos será conveniente efectuar el cómputo dentro de un mismo *warp* y de esta manera poder utilizar la memoria compartida para integrar resultados de manera eficiente.

Tomando en cuenta los puntos anteriores, el nivel de granularidad en una implementación para *GPU*s debe de orientarse a la dimensión en donde existe el mayor nivel de *paralelismo de datos*, siendo esta la clase de paralelismo que se puede explotar en las arquitecturas *SIMT*.

4.3 IMPLEMENTACIÓN

Locusta, la implementación realizada para este trabajo, es la tercera iteración en una serie de experimentos destinados a explorar las diferentes estrategias de paralelización de alto rendimiento disponibles en las dos arquitecturas de cómputo: x86-64 y *GPU*s de Nvidia. En tan sólo cinco años el entorno de los *GPU*s ha cambiado drásticamente; las arquitecturas en donde se realizaron los primeros prototipos ya han sido clasificadas como obsoletas[12]. Así como la arquitectura del hardware ha evolucionado en cada generación, también las herramientas disponibles. Las características en las nuevas generaciones tienen menos restricciones, permitiendo implementar nuevos algoritmos que pueden ejecutarse de manera eficiente en el modelo *SIMT*. Esto ha iniciado un interés importante en aplicaciones distintas a simulaciones; como es el caso de aprendizaje de máquina, particularmente en aprendizaje profundo.

4.3.1 *Prototipos*

Como primer paso, se desarrolló una implementación de un algoritmo genético canónico para ser ejecutada en un *CPU*. La intención fue tener una plataforma base sobre la cual extender y poder realizar comparaciones. Posteriormente, se extendió para poder paralelizar las secciones referentes a transformaciones genéticas y evaluación de la función objetivo, utilizando un modelo de memoria compartida utilizando *OpenMP*.

El siguiente paso fue adaptar la implementación para utilizar un modelo heterogéneo de cómputo y aprovechar el poder de cómputo de

12 Tesla y Fermi

un *GPU* solamente para acelerar el paso de evaluación de la función objetivo, manteniendo el resto de la ejecución en el *CPU*.

Esta estrategia sufre de limitantes importantes al requerir transferir los datos del sistema anfitrión (*host*) al *GPU* en cada generación: todos los genomas de la población se tienen que copiar, para una vez realizadas las evaluaciones transferir los resultados de vuelta al *host*. Las transferencias entre *host* y *GPU* utilizan el puerto *PCI-Express* donde está montado el coprocesador. En la ultima generación de *GPU*s que incorporan la tercera generación de *PCI-Express* logran un ancho de banda de hasta $12GBps$. El comparar esta cifra con los $320GBps$ de ancho de banda que tiene la memoria de un *GPU*, indica que se pudiera tener un mayor aprovechamiento si la información residiera en el *GPU* completamente.

El problema de comunicación también existe cuando se crean implementaciones para arquitecturas *NUMA* [13], en donde al distribuir el trabajo a otros nodos de cómputo se debe minimizar la comunicación y coordinación, para que los canales de comunicación no sean el cuello de botella de rendimiento de la implementación. En el caso particular del cómputo heterogéneo, existen estrategias para mitigar los costos de comunicación y coordinación; realizar operaciones de transferencias y operaciones de cómputo en el *GPU* de manera concurrente sobre conjuntos independientes de datos. De esta manera, en cada generación se puede intercalar el procesamiento del primer conjunto con la transferencia del segundo conjunto.

Hay otro problema aún más grave con esta estrategia; la implementación del algoritmo genético para el *CPU* determina cierta organización de los datos en memoria. Al transmitir los datos al *GPU* se hereda este mismo esquema para su evaluación en el *GPU*. Cada arquitectura requiere el uso de patrones de acceso a memoria distintos para lograr alcanzar el máximo rendimiento. En este caso en particular, ninguno de los patrones permitirá a los hilos dentro de un *warp* efectuar las lecturas de memoria de forma paralela teniendo un alto impacto en el rendimiento del cómputo en el *GPU* (Figura 3.10). Si se intenta reorganizar los datos durante la transferencia, tendría que ser realizado en cada generación para todos los individuos de la población. Tomando en cuenta que reorganizar datos en memoria es una operación lenta y que genera alta contención en el subsistema de memoria, esto se añade al *overhead* de utilizar el coprocesador, lastimando cualquier beneficio que este podría ofrecer.

13 Non-Uniform Memory Architecture

4.3.2 *Drosophila*

Drosophila es una implementación de un algoritmo genético que reside completamente en el *GPU*, lo que implica algunas restricciones sobre las aplicaciones de la implementación, pero permite aprovechar el poder de cómputo de un *GPU*. A continuación se enumeran los beneficios más importantes de tener una implementación que reside en el *GPU*.

- Se eliminó la necesidad de coordinación con el *host* y con ello el cuello de botella de este canal de comunicación.
- Se rediseñaron las estructuras de datos para utilizar los patrones de acceso necesarios para maximizar el rendimiento en el *GPU*.
- Al efectuar todas las operaciones dentro del subsistema de memoria del *GPU* se puede aprovechar el ancho de banda que este provee, el cual puede llegar a ser 5 veces mayor al disponible en un *CPU*.
- Dado que las meta-heurísticas en cuestión son algoritmos estocásticos, el rendimiento de la implementación estará fuertemente relacionado con la capacidad de generar números pseudoaleatorios rápidamente. Utilizando *cuRAND* para la generación de números pseudoaleatorios se alcanza un *speedup* de $75x$ en esta porción del algoritmo. Al mantener toda la implementación de la meta-heurística en el *GPU* se pueden generar todos los números pseudoaleatorios requeridos en cada generación de manera eficiente.

Una vez concluida la implementación de *Drosophila* para el *GPU*, se portaron los algoritmos para poder ser ejecutados de manera paralela en el *CPU* utilizando *OpenMP*, de manera similar a como se hubiera efectuado utilizando *OpenCL* para ejecutar las mismas funciones en diferentes arquitecturas de cómputo paralelo.

Los resultados obtenidos al comparar los tiempos de ejecución entre estas dos implementaciones, fueron muy favorables para la implementación del *GPU*, alcanzando un *speedup* de hasta $800x$. Sin embargo, es importante subrayar que este no es el *speedup* para cualquier ejecución; para alcanzar tales niveles de rendimiento se necesita de configurar la meta-heurística, particularmente en este caso el algoritmo genético, utilizando conjuntos poblacionales de más de $10,000$ individuos, con un número de variables que sea múltiplo de 32 y menor a 512. Además, como veremos con más detalle en el siguiente capítulo, estos resultados ilustran la complejidad en comparar el rendimiento de una aplicación en diferentes

arquitecturas de cómputo. Optimización específicas a una de las arquitecturas podría tener un resultado negativo en otra y vice versa. En el caso particular de *Drosophila* se realizaron muchas optimizaciones ad-hoc a la arquitectura de un *GPU*, creando un sesgo en el rendimiento de la aplicación para un *CPU*.

En el caso particular de aplicaciones de alto rendimiento, el desarrollar una biblioteca dirigida a distintas arquitecturas de cómputo implica resolver problemas fundamentalmente diferentes. Reutilizar la misma implementación de los algoritmos o estructuras de datos puede ser una estrategia contraproducente.

4.3.3 *Locusta*

Locusta extiende la implementación del algoritmo genético en *Drosophila*, abstrayendo los elementos comunes a cualquier meta-heurística basada en poblaciones y agrega dos nuevas implementaciones: cúmulo de partículas y evolución diferencial.

En esta versión, cada una de las arquitecturas utiliza una implementación independiente, con optimizaciones y estrategias adecuadas a cada caso para lograr alcanzar el máximo rendimiento disponible en cada arquitectura de cómputo paralelo. Esta perspectiva nos permite realizar un estudio objetivo, analizando los beneficios y restricciones de cada una de las arquitecturas, y entendiendo los casos de uso en donde se pueden aprovechar dichos recursos de cómputo.

El cambio más significativo en la implementación, es la organización de los datos para cada una de las arquitecturas. En ambos casos se utiliza un esquema de estructuras de arreglos para mantener los datos en conjuntos contiguos y alineados, pero describiendo el patrón de acceso a datos adecuado a cada arquitectura.

Para la implementación en *CPU* este cambio tuvo un impacto significativo en rendimiento, teniendo en promedio un *speedup* de $10x$ para la ejecución serial al compararlo con *Drosophila*, en donde se utilizaban las mismas estructuras de datos optimizadas para el *GPU* en la implementación para el *CPU*.

En el caso de la implementación para el *GPU* también hubo un cambio importante sobre el nivel de la granularidad de paralelización. En *Drosophila*, la granularidad de paralelización era al nivel de cada variable. Aunque esta estrategia permite ejecutar mayor volumen de hilos en paralelo, tiene tres importantes desventajas. Dado que en un

GPU la unidad de ejecución de un *kernel* es sobre *warps*, el utilizar cualquier número de variables distintas a un múltiplo del tamaño del *warp* [14] implica lanzar *warps* parcialmente utilizados, teniendo un impacto importante en el rendimiento de la ejecución. La segunda desventaja es respecto a las configuraciones válidas que existen para lanzar la ejecución de un *kernel*. Cada arquitectura define ciertas limitantes, particularmente el número máximo de hilos por *bloque* que es de 512 para arquitecturas *Pre-Fermi* y 1024 para las nuevas arquitecturas. La tercera desventaja tiene que ver con el número de registros disponibles en cada *stream processor* del *GPU*, donde el tener un mayor nivel de ocupación de un procesador no necesariamente implica mejor rendimiento. [Volkov, 2010] Con esto en mente, la implementación de *Locusta* utiliza una granularidad a nivel de cada individuo de la población, dando mayor flexibilidad en las configuraciones disponibles.

Los otros cambios tienen que ver con el diseño de *Locusta* y las herramientas que se pudieron utilizar. El entorno de desarrollo para *GPU*s ha evolucionado con gran rapidez en los últimos años. Ahora existen múltiples herramientas que permiten depurar, así como realizar análisis con gran precisión sobre la ejecución de los procesos y aprovechamiento de recursos. También los compiladores para *GPU*s han madurado considerablemente, añadiendo la posibilidad de utilizar nuevas características de los lenguajes de programación, permitiendo codificar aplicaciones utilizando la misma capacidad de expresión que existen al codificar programas en el *CPU*. Para la creación de *Locusta* se aprovechó esta oportunidad para recodificar haciendo uso de *C++* con *templates*, permitiendo reutilizar código para compilar versiones que utilizan aritmética de punto flotante con precisión simple o doble, esto disponible tanto para la implementación del *CPU* como para el *GPU*. Gracias a la última versión de compilador *nvcc* con soporte para *C++11*, la infraestructura de *Locusta* abstrae cada una de las transformaciones sobre la población como *objetos función* (*functors*), los cuales pueden ser redefinidos, manipulados e intercambiados en tiempo de ejecución.

Estas características permiten utilizar la implementación de *Locusta* como una herramienta para analizar de forma modular cada uno de los elementos de una meta-heurística, así como su comportamiento y rendimiento en una escala masivamente paralela.

14 Hasta la última generación disponible a principios del 2017, todas las arquitecturas de GPU de Nvidia utilizan *warps* con 32 hilos, pero no necesariamente será el caso en futuras generaciones.

4.3.4 *Profiling*

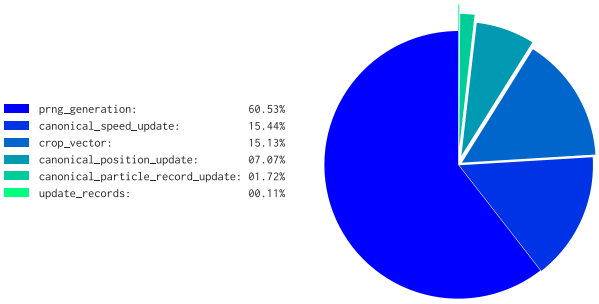
Para lograr entender el comportamiento de un programa es necesario realizar un análisis sobre los tiempos de ejecución, recursos utilizados e interacciones entre los subsistemas tanto de *hardware* como *software* involucrados. A este proceso se le conoce como *profiling* y existen múltiples técnicas para obtener estas mediciones. En este trabajo se utilizaron dos *profilers*, cada uno especializado en acumular y analizar indicadores disponibles en cada entorno de ejecución:

- *vTune Amplifier*: Análisis para aplicaciones ejecutadas en arquitecturas de Intel x86.
- *NVIDIA CUDA Profiler*: Análisis para aplicaciones desarrolladas en CUDA dirigido específicamente a *GPU*s de *Nvidia*.

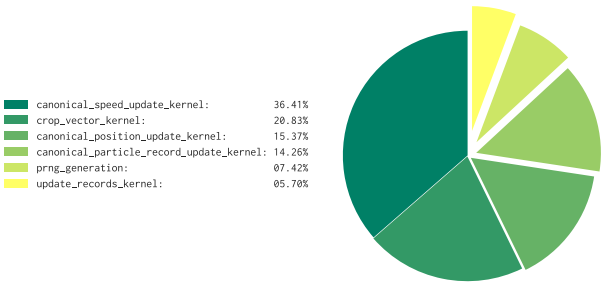
Un aspecto importante sobre la manera en que funcionan estas herramientas es el hecho que utilizan contadores embebidos en el *hardware* para obtener información detallada sobre el comportamiento en cada paso del programa. Como en cualquier otro proceso de medición, los métodos y herramientas utilizadas determinan la resolución y precisión disponible en las mediciones. La naturaleza de estos métodos de medición es imprecisa, ya que los registros de los contadores son leídos de manera periódica durante la ejecución del programa y posteriormente se realiza una correlación entre las instrucciones y el valor de los contadores. El incrementar la frecuencia con la que se leen los contadores nos ofrece mayor resolución en la medición, pero a su vez esto afecta el rendimiento del programa, ya que la captura y proceso de la información de los contadores requiere de interrupciones del sistema, lo que introduce ruido en las mediciones. A pesar de estas limitantes inherentes a los métodos de medición, el utilizar *profilers* nos permite realizar una introspección para comprender el comportamiento del programa y con ello definir estrategias para optimizar la implementación para cada arquitectura.

A continuación se realiza un análisis del comportamiento de cada una de las implementaciones utilizando la información obtenida por las herramientas de *profiling*. Dado que el paso de evaluación de la función objetivo es común para todas las implementaciones, no se tomará en cuenta dentro de la descomposición. Para este ejercicio todas las meta-heurísticas se configuran con 256 islas, 512 individuos y 128 variables.

En la Figura 4.3, Figura 4.4 y Figura 4.5, se presentan gráficas ilustrando los puntos críticos (*hotspots*) de cada una de las meta-heurística implementadas, mostrando la descomposición del tiempo total de ejecución entre el tiempo dedicado en cada transformación de la implementación. Para cada figura se muestra el comportamiento en la ejecución en el *CPU* y en el *GPU*, dando una clara idea más clara del costo que tiene cada una de las transformaciones en cada una de las arquitecturas de cómputo.



(a) CPU



(b) GPU

Figura 4.3: *Hotspots* en el Cúmulo de Partículas

El algoritmo de cúmulo de partículas tiene características clave que permiten una ejecución eficiente en ambas arquitecturas. La más importante es el hecho de que el algoritmo define un único flujo, sin saltos condicionales ni accesos a memoria a localidad aleatorias. A pesar de tener elementos estocásticos estos son representados de manera independiente para cada instancia, manteniendo el paralelismo de datos. Esto es particularmente importante para poder explotar el paralelismo a nivel de datos de forma eficiente al utilizar instrucciones *SIMD* o en una arquitectura *SIMT*.

En el caso de la ejecución en el *GPU*, en donde existe un único *program counter* que comparten todos los hilos de ejecución dentro de cada *warp*, esta característica permite realizar todas las operaciones de forma paralela, logrando mantener una eficiencia alta sobre cada instrucción programada para todo el *warp*. Los patrones que describen los accesos a memoria son óptimos en ambas arquitecturas. En el *GPU*, los accesos son contiguos y alineados sobre cada uno de los elementos

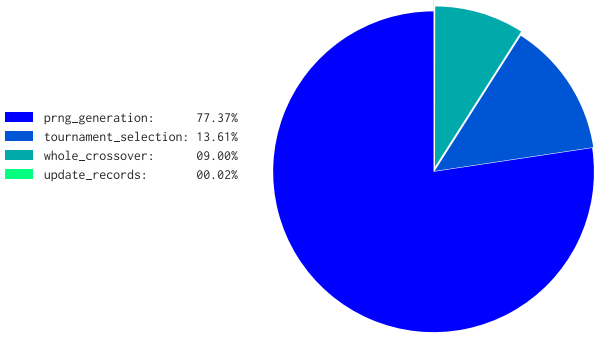
que se distribuyen entre cada hilo de ejecución, requiriendo un número mínimo de transacciones a la *memoria global*. Para los individuos en el enjambre que mantienen la mejor solución de la población, utilizamos una especie de *caché* administrado de forma manual, que guarda los datos de estos elementos en la *memoria compartida* del *GPU*, la cual ofrece mayor ancho de banda y evitamos contención adicional sobre la *memoria global*. Esta estrategia permite servir de forma eficiente una misma dirección en *memoria compartida* a todos los hilos dentro del *warp* sin causar conflictos entre los bancos de memoria, gracias a la propiedad de *multicasting* [CUDA C Best Practices, 2016, 9.2.2.1 Shared Memory and Memory Banks].

Para el caso de ejecución sobre el *CPU* esta característica también ofrece beneficios sobre las otras meta-heurísticas. Aunque los saltos condicionales no tienen una repercusión negativa tan significativa como en los *GPU*s. El mantener patrones de acceso a memoria que son deterministas, contiguos y alineados permite aprovechar con máxima eficiencia cada *cache-line* y utilizar instrucciones *SIMD* para efectuar de forma paralela las operaciones de punto flotante sobre las unidades vectoriales del procesador. El uso de instrucciones *SIMD* es importante en el algoritmo de cúmulo de partículas, ya que es la meta-heurística cuyos operadores requieren el mayor número de operaciones de punto flotante por variable.

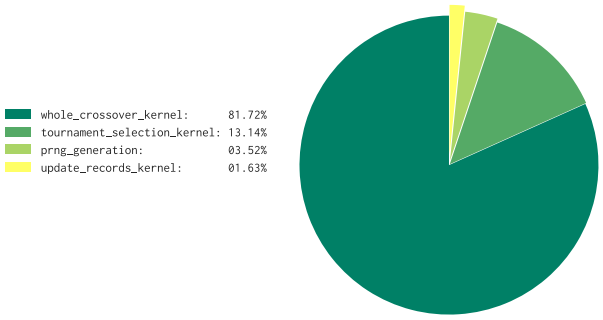
Como veremos más adelante, la cardinalidad de números pseudoaleatorios requeridos por la meta-heurística será un elemento importante en el rendimiento del algoritmo. El cúmulo de partículas es la meta-heurística implementada que requiere el menor número de números pseudoaleatorios. Figura 4.3

A pesar de tener un alto grado de paralelismo a nivel de datos, de las tres meta-heurísticas implementadas el algoritmo genético es el caso cuyos operadores utilizan el mayor número de saltos condicionales en su lógica, a parte de requerir de la mayor cantidad de números pseudoaleatorios comparado con las otras meta-heurísticas.

El algoritmo de evolución diferencial posee un alto grado de paralelismo a nivel de datos con pocos saltos condicionales, siendo así un buen candidato para utilizar instrucciones *SIMD* o una arquitectura *SIMT*. De manera similar al algoritmo genético, los operadores de la meta-heurística utilizan accesos a memoria poco predecibles, lo que irrumpe en los patrones de memoria requeridos para una ejecución eficiente en el *GPU*. Esta penalización se puede observar al comparar el costo relativo de los operadores de cruza y mutación (*whole crossover*) entre las arquitecturas de *CPU*s y



(a) CPU



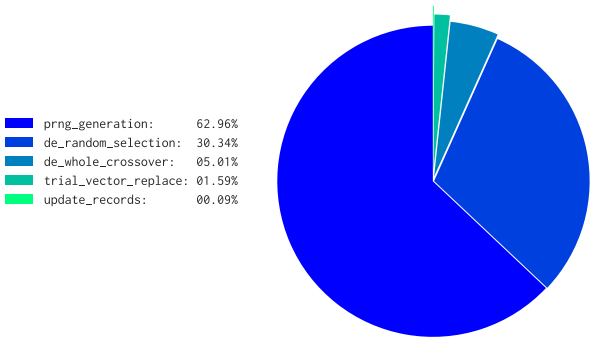
(b) GPU

Figura 4.4: *Hotspots* en el Algoritmo Genético

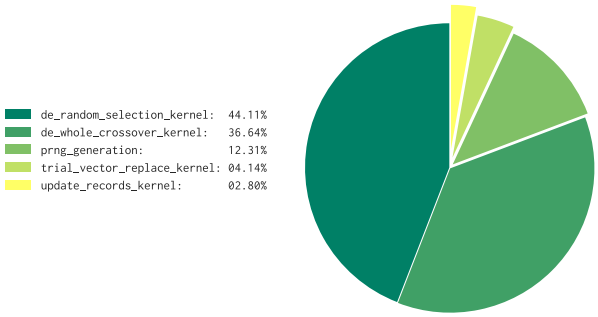
*GPU*s. (véase Figura 4.4a vs. Figura 4.4b y Figura 4.5a vs. Figura 4.5b)

En un *GPU*, la divergencia en el flujo de un programa entre los hilos de ejecución de un *warp* impactan significativamente el rendimiento de forma negativa. Debido a que todos los hilos dentro del *warp* comparten un mismo *program counter*, los saltos condicionales implican detener a todos los hilos que no son parte de la condición, evitando la ejecución paralela de las instrucciones. Para poder iniciar la ejecución de las instrucciones programadas para un *warp* se requiere satisfacer todas las dependencias de datos para la sección de código; los operadores de un algoritmo genético requieren de accesos a memoria a localidades aleatorias, definiendo patrones no contiguos que implican realizar transacciones independientes para cada hilo del *warp*. Por estas razones los operadores de un algoritmo genético no tienen características que permitan una ejecución eficiente en una arquitectura de *GPU*.

El costo de los saltos condicionales puede ser mitigado en una arquitectura con predicción de saltos y ejecución especulativa, este siendo el caso para la ejecución en procesadores *x86*. El acceso a localidades aleatorias no impacta de manera importante el rendimiento, ya que la codificación de los individuos dentro de la



(a) CPU



(b) GPU

Figura 4.5: *Hotspots* en Evolución Diferencial

población sigue siendo contigua, y esto permite mantener una alta eficiencia de utilización de *cache-lines* por transacción de memoria. A pesar de ser paralelizable, el operador de selección por torneo utilizado en el algoritmo genético y evolución diferencial, realiza accesos aleatorios sobre variables individuales [15], lo que define un patrón de accesos poco eficiente. Para estas dos meta-heurísticas el operador de selección es el más caro en cada generación.

Todas las meta-heurísticas implementadas son algoritmos estocásticos que requieren consumir un conjunto de números pseudoaleatorios en cada generación. En el caso de la ejecución en el *CPU* (véanse Figura 4.3a, Figura 4.4a, Figura 4.5a) la regeneración de este conjunto tiene el mayor impacto en el tiempo de ejecución para todas las meta-heurísticas; 57.47 % en el cúmulo de partículas, 74.83 % en el algoritmo genético y 68.63 % en evolución diferencial. En contraste, la generación de números pseudoaleatorios en el *GPU* puede realizarse de manera más eficiente (véanse Figura 4.3b, Figura 4.4b, Figura 4.5b), teniendo un costo relativamente menor en las meta-heurísticas; 6.17 % en el cúmulo de partículas, 3.37 % en el algoritmo genético y 10.54 % en evolución diferencial.

15 4 bytes para precisión simple y 8 bytes para precisión doble

5

ANÁLISIS DE RESULTADOS

En este capítulo se presenta una serie de análisis comparando el rendimiento de cada una de las meta-heurísticas implementadas sobre las diferentes arquitecturas de cómputo paralelo. Se mostrará cómo el escalamiento en el rendimiento del cómputo depende de los métodos de paralelización aplicados, así como la configuración específica de cada meta-heurística.

A continuación se describe la configuración utilizada en cada meta-heurística[1]:

- Cúmulo de Partículas: Inercia 0.5, Constante Cognitiva: 2.0 y Constante Social: 2.0
- Algoritmo Genético: Cruza: 90 % y Mutación: 10 %
- Evolución Diferencial: Cruza: 90 % y Mutación: 10 %

Los experimentos fueron ejecutados en los sistemas con las siguientes configuraciones:

- Intel Core i5-2500K (Sandy Bridge):
  - Proceso litográfico a $32nm$
  - 4 núcleos @ 3.3 GHz
  - Ancho de banda memoria *RAM DDR3*: 21.0$GBps$
  - Consumo energético: 95W
- Intel Core i7-4710HQ (Haswell):
  - Proceso litográfico a $22nm$
  - 4 núcleos con *Hyper-Threading* @ 3.1GHz
  - Ancho de banda memoria *RAM DDR3*: 25.6$GBps$

1 Se seleccionaron estas configuraciones, ya que son una buena referencia de una configuración neutra de cada una de las meta-heurísticas. Cada aplicación puede requerir ajustes particulares a cada uno de los parámetros, dependiendo de la función objetivo.

  - Consumo energético: $47W$

  Ambas arquitecturas de *CPU* comparten las siguientes características en su jerarquía de *caché*:

  - L1: $64kB$ *8-way set associative* por núcleo
  - L2: $256kB$ *8-way set associative* por núcleo
  - L3: $6MB$ *12-way set associative* compartido entre 4 núcleos

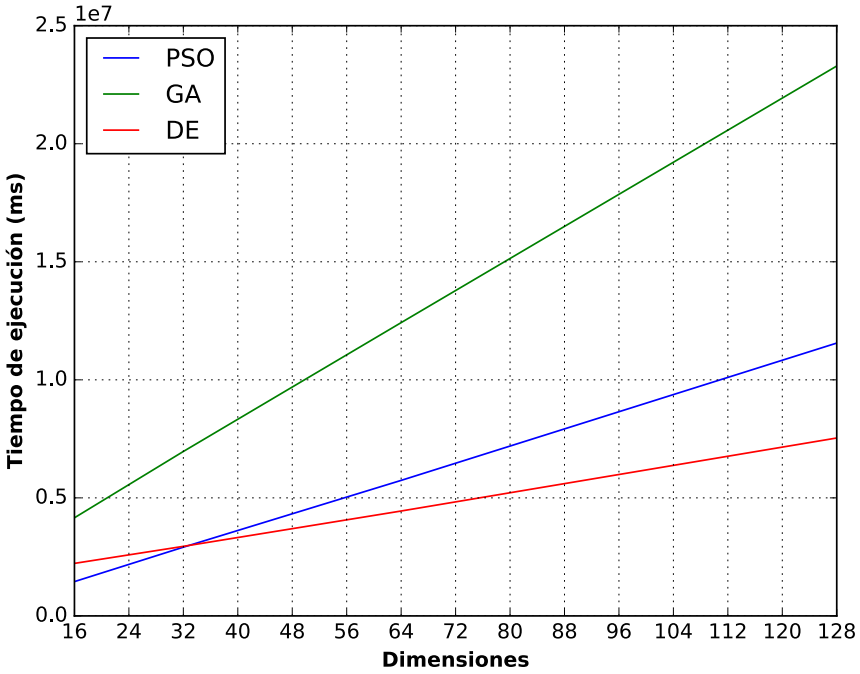
- NVIDIA Geforce GTX 1060 6GB (*GP106-400*)
  - Proceso litográfico a $16nm$
  - 1280 núcleos *CUDA* @ $1.5GHz$ (128 núcleos $\times$ $10SM$)
  - Ancho de banda memoria *RAM GDDR*: $192GBps$
  - Consumo energético: $120W$
  - L1: $48kB$
  - L2: $1536kB$ ($256kB$ por *iMC*)
  - Registros: $256kB$
  - Memoria Compartida: $96kB$

5.1 RENDIMIENTO EN EL CPU

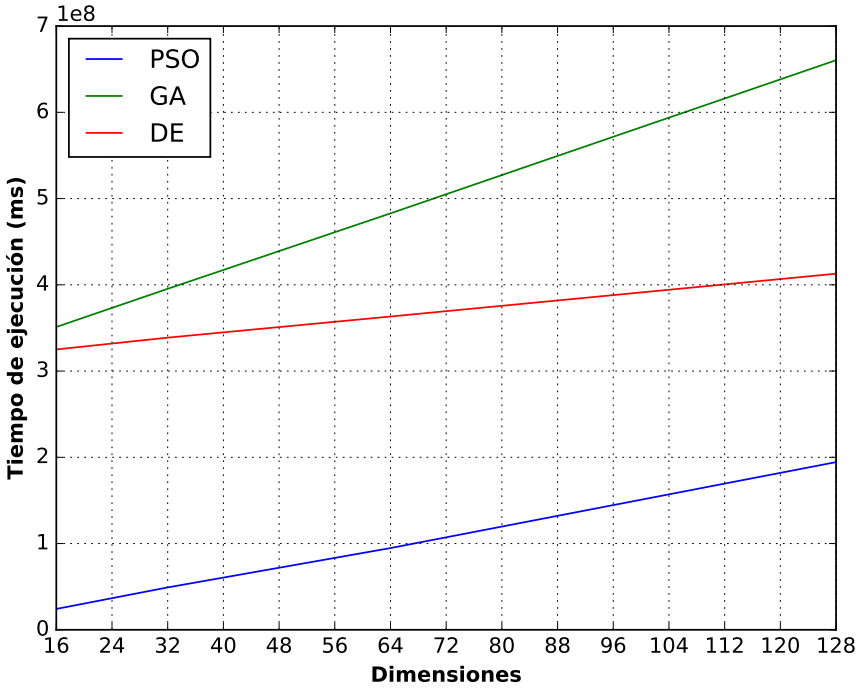
5.1.1 *Número de Dimensiones*

Para el primer análisis se capturaron los tiempos de ejecución en cada micro-arquitectura de *CPU* utilizando un sólo hilo de ejecución. Para cada meta-heurísticas se hace un barrido sobre el número de dimensiones del problema en incrementos de 8. Tanto la complejidad en los operadores de las meta-heurísticas como la evaluación de la función objetivo crece conforme aumenta el número de dimensiones del problema, incrementando el cómputo y transferencias de memoria requeridos. En los resultados se observa como el escalamiento en rendimiento es distinto para cada meta-heurística. (véanse Figura 5.1, Figura 5.2)

La implementación de cúmulo de partículas tiene los menores tiempos de ejecución en función al número de individuos en la configuración. El algoritmo genético tiene los tiempos de ejecución

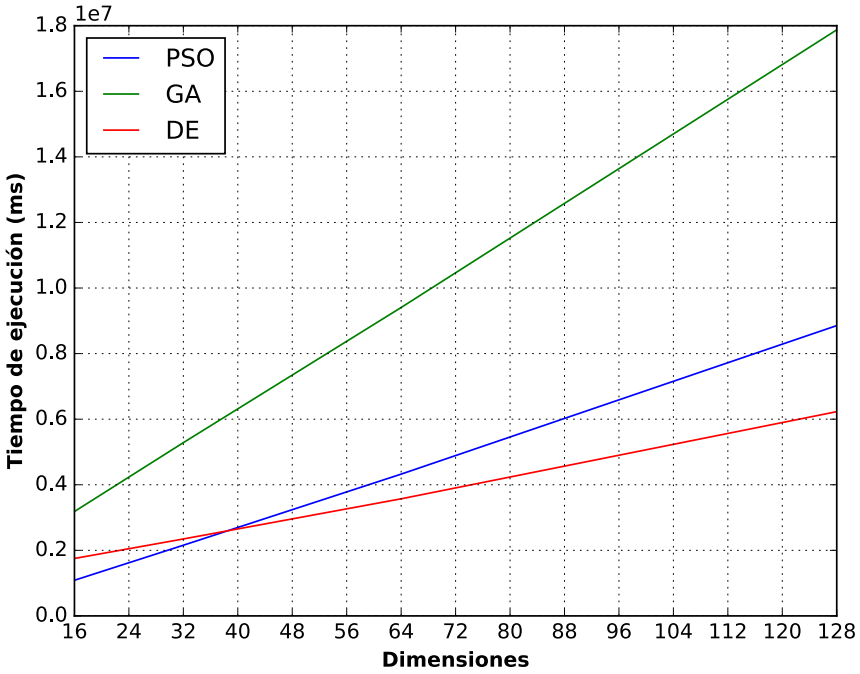


(a) 32 individuos por isla

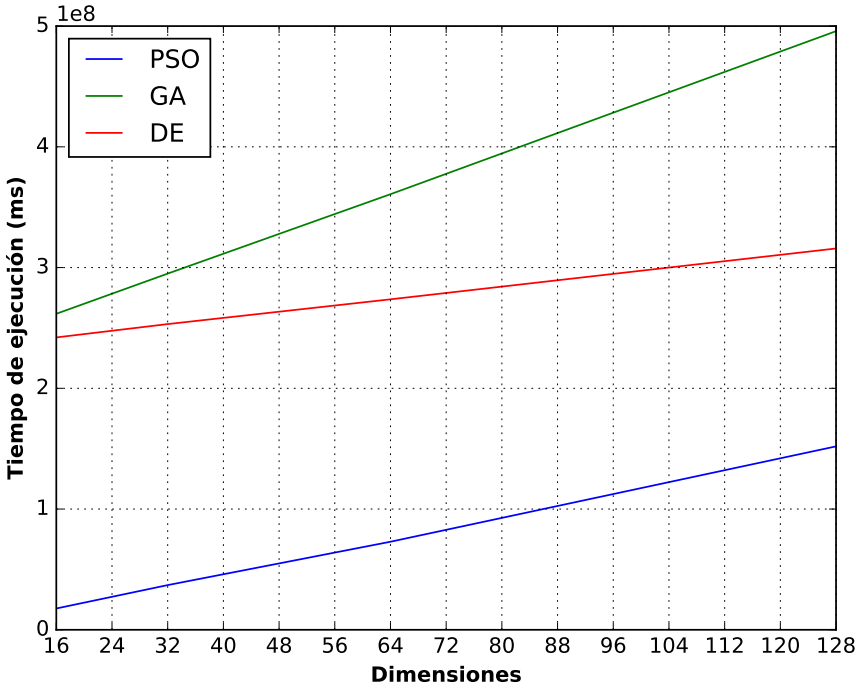


(b) 512 individuos por isla

Figura 5.1: Sandy Bridge - Barrido en dimensiones, 256 islas.



(a) 32 individuos por isla



(b) 512 individuos por isla

Figura 5.2: Haswell - Barrido en dimensiones, 256 islas.

más altos y la mayor pendiente en escalamiento en función de el número de variables.

Por otro lado, la implementación de evolución diferencial muestra ser un punto medio entre el algoritmo genético y el cúmulo de partículas, con la interesante cualidad de escalar mejor conforme se incrementa el número de variables del problema, teniendo un costo de cómputo por generación menor que las otras meta-heurísticas para configuraciones de conjuntos de poblaciones pequeños con altas dimensiones. Sin embargo, al utilizar configuraciones con poblaciones más numerosas, la brecha inicial en rendimiento aumenta considerablemente.

Al comparar el rendimiento entre las dos micro-arquitecturas de *CPU*s, se observan comportamientos muy similares, notando que la nueva micro-arquitectura logra mejores tiempos de ejecución, menores pendientes escalamiento y con un menor consumo energético. Dado que las dos micro-arquitecturas comparten las mismas características en sus niveles de *caché*, utilizan la misma tecnología de memoria (*DDR3*) y el *CPU* de micro-arquitectura *Haswell* trabaja a una frecuencia menor; indicando que el incremento en rendimiento proviene de las mejoras en el diseño de la micro-arquitectura: decodificación de instrucciones, ejecución de operaciones aritméticas por ciclo y controladores de memoria que ofrecen mayor ancho de banda.

Para realizar cálculos de punto flotante de forma eficiente en un procesador moderno, es crucial el utilizar instrucciones *SIMD* para explotar el paralelismo de datos y maximizar el número de operaciones por ciclo. El utilizar un *diseño orientado a datos* permite organizar los datos en memoria de tal manera que se aproveche cada lectura de *cache-line* con máxima eficiencia, y cuyos datos estén ya alineados para ser escritos directamente a los registros *SIMD* del procesador.

Para este experimento se utilizó la función de *Weierstrass* (véase Figura A.2) como función objetivo, para comparar las mejoras en rendimiento al vectorizar en cada micro-arquitectura. En la Tabla 5.1 se muestra el *speedup* al utilizar instrucciones *SIMD* para acelerar cada una de las meta-heurísticas.

Meta-heurística	Speedup
PSO	2.44x
GA	2.48x
DE	2.43x

(a) Sandy Bridge

Meta-heurística	Speedup
PSO	2.89x
GA	2.91x
DE	2.57x

(b) Haswell

Cuadro 5.1: Speedup utilizando instrucciones SIMD en el CPU. Configuración: 256-512-128

5.1.2 *Paralelización en CPUs*

En esta sección se presentan los resultados al paralelizar entre los múltiples núcleos disponibles en un *CPU*. En el caso particular del procesador *4710hq* existen cuatro núcleos y la capacidad de utilizar *hyper-threading*[2]; característica que permite la ejecución de múltiples hilos en un mismo núcleo.

Al utilizar *multithreading* en un núcleo físico del procesador, se están ejecutando varias tareas de forma simultanea en el mismo procesador, logrando esconder la latencia de los accesos a memoria, incrementando el rendimiento computacional y eficiencia energética. El utilizar *hyper-threading* permitió incrementar de forma significativa el rendimiento para ciertas configuraciones, en algunos casos obteniendo un *speedup* mayor a $4x$ para un procesador con cuatro núcleos (véase Tabla 5.2). Este resultado es signo de que el rendimiento de todas las meta-heurísticas en este estudio, está acotado por la latencia en los accesos a memoria.

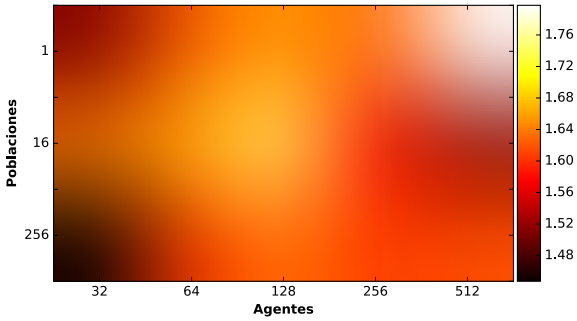
Se generaron mapas de calor para poder estudiar el comportamiento del factor de escalamiento al utilizar diferentes configuraciones en cada una de las meta-heurísticas (véanse Figura 5.3, Figura 5.4, Figura 5.5). De manera general, las configuraciones con menor número de agentes tienen el nivel más bajo de escalamiento, mientras que las configuraciones con mayor número de agentes muestran el nivel más alto. Esto es esperado, ya que al incrementar el número de agentes se incrementa el nivel de paralelismo y la densidad de cómputo sobre todas las operaciones realizadas, ayudando a esconder el *overhead* que induce el paralelizar sobre múltiples hilos de ejecución.

Al comparar los diferentes mapas de calor se pueden observar fluctuaciones específicas a cada meta-heurística y función objetivo. La función esférica muestra el mayor nivel de variación, especialmente al paralelizar sobre 8 hilos. Por el hecho de tener una

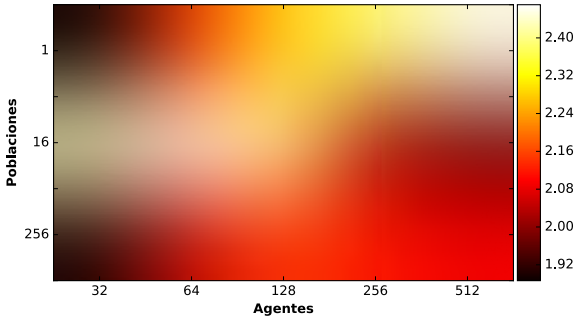
2 Hasta dos hilos por núcleo en el caso del procesador *4710hq*

baja densidad de cómputo, el rendimiento de la función esférica depende en gran parte del número de agentes en cada población. Mientras, para las configuraciones con un número de poblaciones mayor a 16, el continuar incrementando el número de poblaciones no afecta de manera significativa el nivel de escalamiento. Los mapas de calor para la función de Weierstrass tienen menos variación, obteniendo un escalamiento más uniforme, incluso en los casos con poblaciones más reducidas. Esto se debe al alto costo computacional de la función que mantiene los núcleos del *CPU* operando sin requerir accesos adicionales a la memoria principal del sistema, logrando una paralelización más eficiente que en la función esférica.

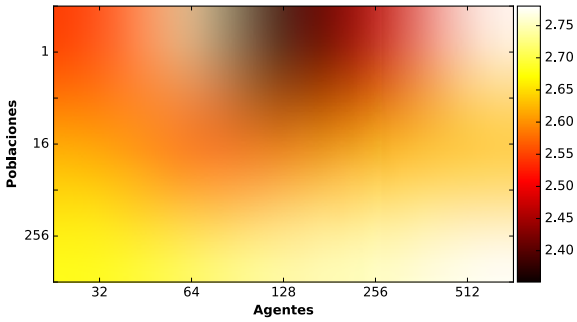
La evaluación de la función objetivo es la parte crucial en el rendimiento de todas las meta-heurísticas, dominando gran parte del tiempo de ejecución. De manera independiente, cada meta-heurísticas muestra comportamientos muy diferentes con cada una de las funciones. Los mapas ilustran que el escalamiento no siempre describe un comportamiento monótono conforme se incrementa el número de poblaciones y agentes. Existen ciertos nodos en los mapas de calor, en donde disminuye o aumenta el rendimiento de manera aislada. Estos artefactos probablemente surgen debido a detalles de la micro-arquitectura: asociatividad y tamaño de los *caches*, lo que puede incrementar el costo de ciertos patrones de acceso a memoria [Ostrovsky, 2010].



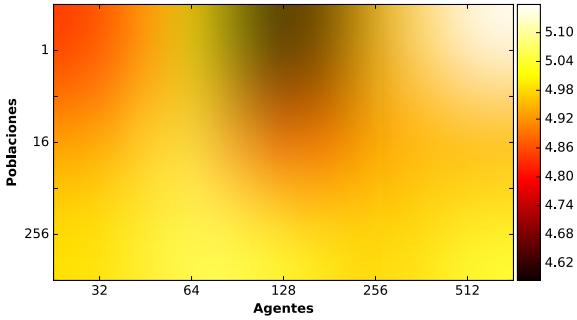
(a) 4 hilos
Función esférica



(b) 8 hilos (*Hyper-threading*)
Función esférica

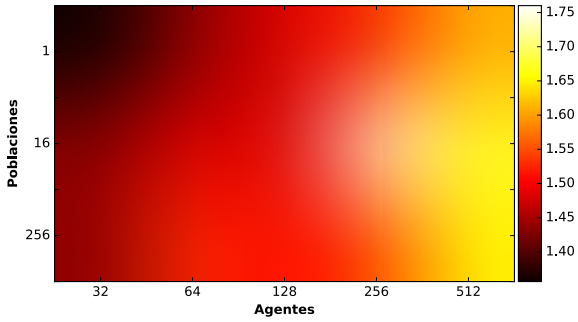


(c) 4 hilos
Función de Weierstrass

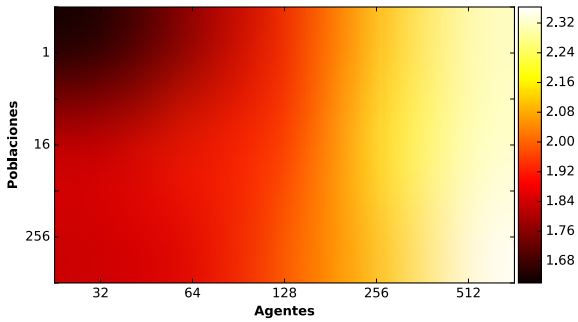


(d) 8 hilos (*Hyper-threading*)
Función de Weierstrass

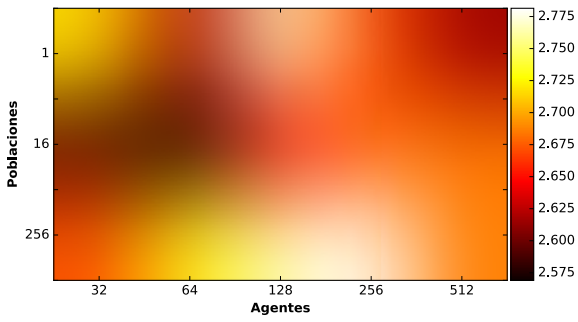
Figura 5.3: PSO - Speedup 4 vs 8 Hilos - 128 Dimensiones



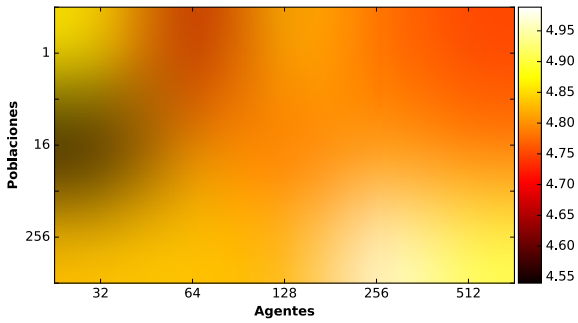
(a) 4 hilos
Función esférica



(b) 8 hilos (*Hyper-threading*)
Función esférica

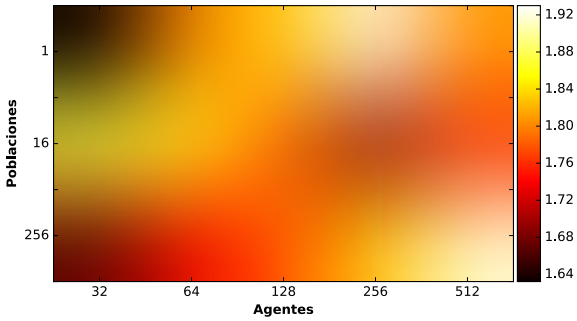


(c) 4 hilos
Función de Weierstrass

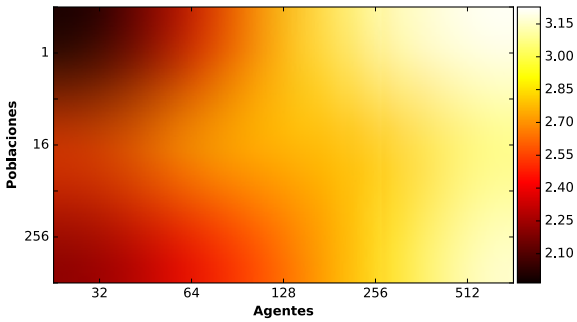


(d) 8 hilos (*Hyper-threading*)
Función de Weierstrass

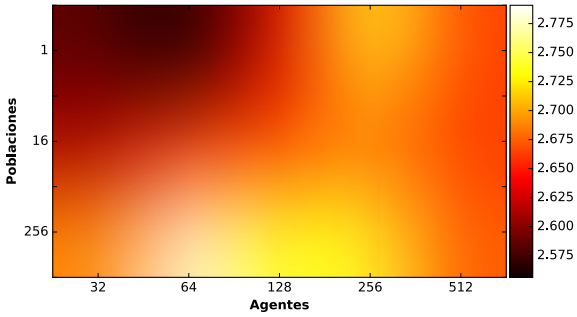
Figura 5.4: GA - Speedup 4 vs 8 Hilos - 128 Dimensiones



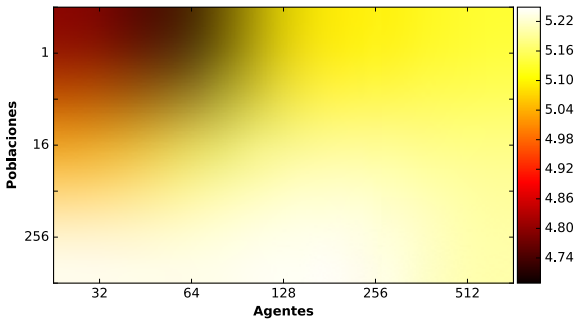
(a) 4 hilos
Función esférica



(b) 8 hilos (*Hyper-threading*)
Función esférica



(c) 4 hilos
Función de Weierstrass



(d) 8 hilos (*Hyper-threading*)
Función de Weierstrass

Figura 5.5: DE - Speedup 4 vs 8 Hilos - 128 Dimensiones

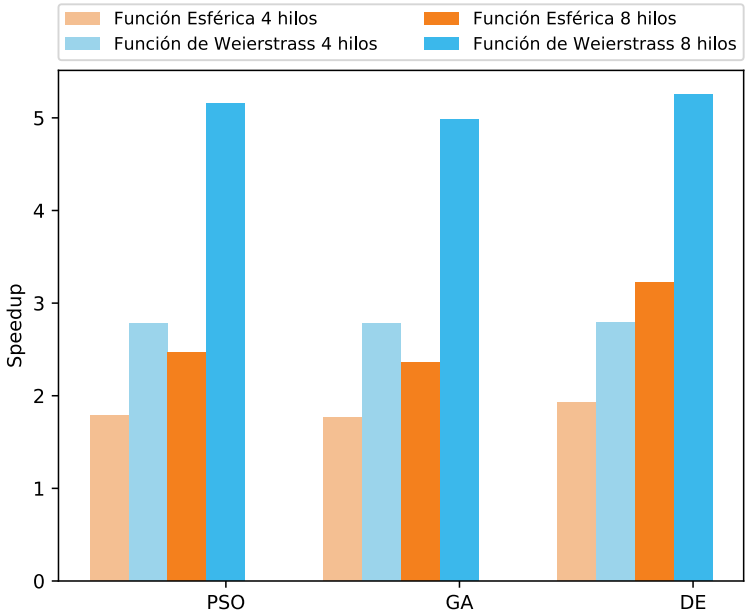


Figura 5.6: Máximo speedup en el *CPU* al comparar el tiempo de ejecución de 4 y 8 hilos vs. un único hilo.

Meta-heurística	4 hilos	8 hilos (*Hyper-threading*)
PSO	1.79x	2.47x
GA	1.76x	2.36x
DE	1.93x	3.22x

(a) Función Esférica

Meta-heurística	4 hilos	8 hilos (*Hyper-threading*)
PSO	2.78x	5.15x
GA	2.78x	4.98x
DE	2.79x	5.25x

(b) Función de Weierstrass

Cuadro 5.2: Tabla de valores - Máximo speedup en el *CPU* al comparar el tiempo de ejecución de 4 y 8 hilos vs. un único hilo.

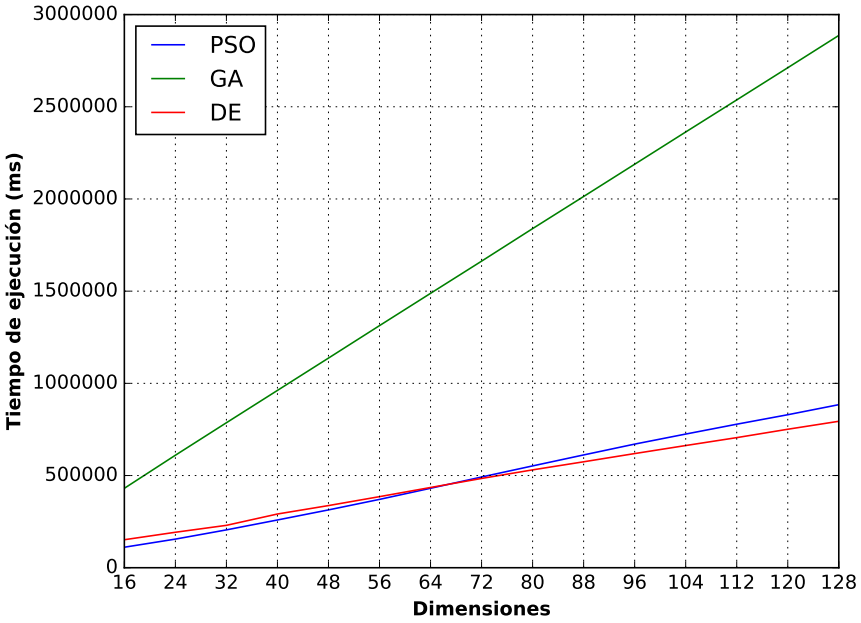
5.2 RENDIMIENTO EN EL GPU

5.2.1 *Número de Dimensiones*

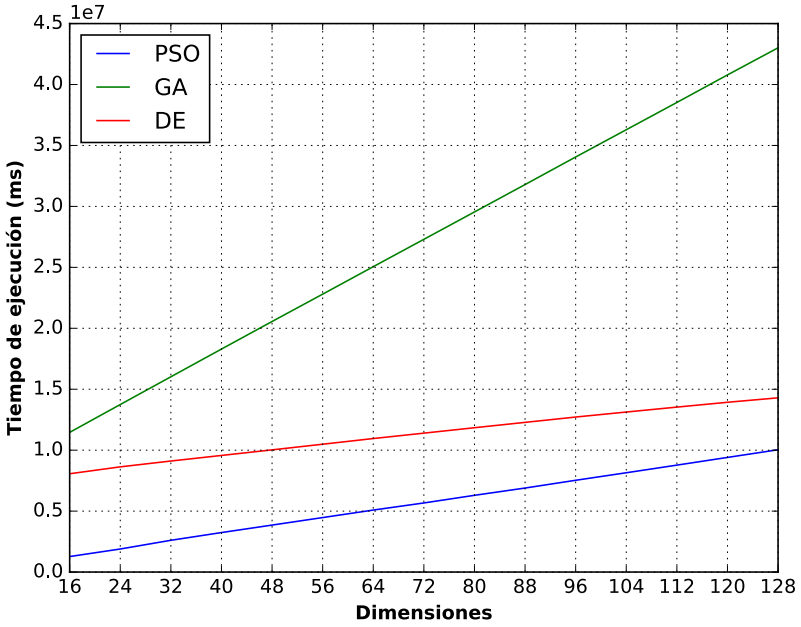
En esta segunda sección hacemos la misma comparación de escalamiento, realizando un barrido sobre las dimensiones del problema en incrementos de 8, en este caso para la ejecución del *CPU*s (véase Figura 5.7).

La implementación del algoritmo genético tiene el peor rendimiento de las tres meta-heurísticas en el *GPU*, debido a la alta incidencia de saltos condicionales y accesos aleatorios a memoria involucrados en los operadores genéticos. Esto era esperado, ya que la arquitectura de un *GPU* no permite la ejecución paralela de los hilos en un *warp*, cuando estos siguen ramas divergentes de código.

Similar al caso del *CPU*, la implementación del cúmulo de partículas y evolución diferencial en el *GPU*, muestran tiempos de ejecución muy parecidos. Para configuraciones con poblaciones más numerosas, el cúmulo de partículas muestra mejor rendimiento, mientras que el algoritmo de evolución diferencial tiene mejor escalamiento en función al número de dimensiones del problema.



(a) 32 individuos por isla



(b) 512 individuos por isla

Figura 5.7: Geforce GTX 1060 6GB - Barrido en dimensiones, 256 islas.

5.2.2 *CPUs vs. GPUs*

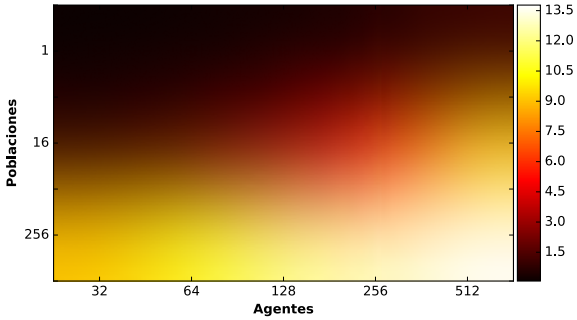
Por último, se muestran mapas de calor para ilustrar la aceleración que puede ser alcanzada en el *GPU* de micro-arquitectura *Pascal*, al comparar los tiempos de ejecución de cada meta-heurística contra las mediciones obtenidas con el *CPU* de micro-arquitectura *Haswell*, utilizando un solo hilo de ejecución (véanse Figura 5.8, Figura 5.9, Figura 5.10).

Estos mapas de calor para las tres meta-heurísticas en el *GPU* muestran prácticamente el mismo comportamiento. Tienen gran variabilidad a lo largo de las diferentes configuraciones, comenzando con niveles de aceleración prácticamente nulos para configuraciones con un número reducido de agentes, hasta llegar al máximo factor de escalamiento al alcanzar la configuración con mayor número de agentes. El utilizar una configuración con sólo una población no es suficiente para ver los beneficios de paralelizar sobre un *GPU*, únicamente las configuraciones con múltiples poblaciones y gran número de agentes producen niveles de aceleración superiores a los obtenidos en la implementación paralela del *CPU*.

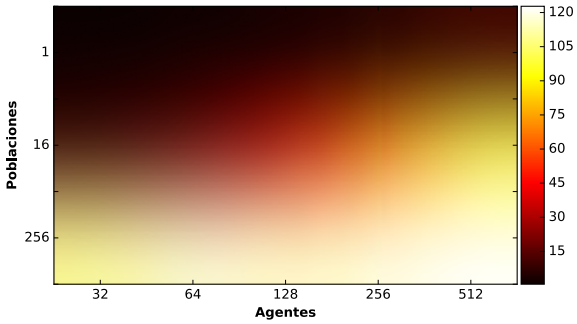
Al analizar los mapas de calor para la función esférica, se observa que los niveles de aceleración que logra el *GPU* son muy limitados. Incluso en el mapa de evolución diferencial, la meta-heurística que mostró el mayor grado de aceleración para esta función apenas alcanza un *speedup* cuatro veces mayor al obtenido en la implementación paralela en el *CPU*. Para todos los mapas de calor de estad función, el máximo nivel de aceleración está localizado en la esquina inferior derecha, correspondiente a la máxima configuración[3].

Los mapas de la función de Weierstrass muestran más uniformidad en su mitad inferior, consiguiendo niveles de aceleración superiores a los *CPU* para todas las configuraciones con múltiples poblaciones, a parte de alcanzar niveles de aceleración más altos que la función esférica, existe varias regiones en el mapa con valores de aceleración cercanos al máximo. Para esta función la implementación de cúmulo de partículas logró un *speedup* veinte veces mayor al rendimiento obtenido en la implementación paralela del *CPU*.

3 Utilizando 256 poblaciones, 512 agentes y 128 dimensiones

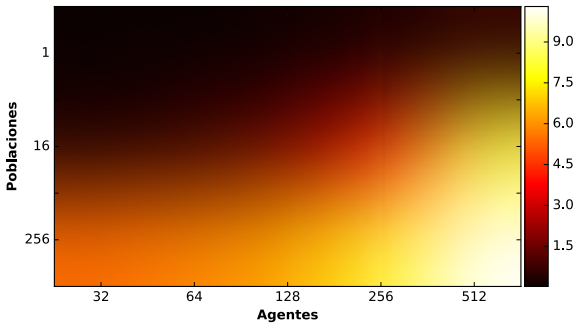


(a) Función esférica

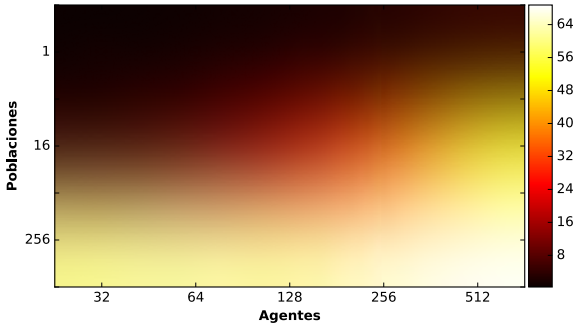


(b) Función de Weierstrass

Figura 5.8: PSO - Haswell vs Geforce GTX 1060 6GB - 128 Dimensiones

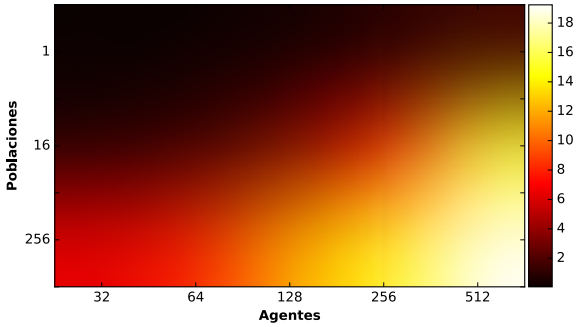


(a) Función esférica

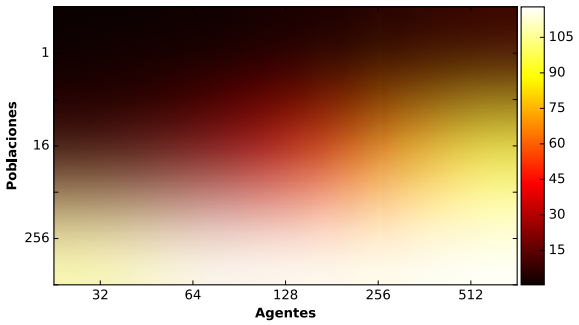


(b) Función de Weierstrass

Figura 5.9: GA - Haswell vs Geforce GTX 1060 6GB - 128 Dimensiones



(a) Función esférica



(b) Función de Weierstrass

Figura 5.10: DE - Haswell vs Geforce GTX 1060 6GB - 128 Dimensiones

En contraste con la paralelización en un *CPU*, los recursos de cómputo en un *GPU* son más difíciles de explotar. A pesar de que las tres meta-heurísticas describen procesos inherentemente paralelos, para lograr alcanzar altos niveles de aceleración se necesita utilizar una función objetivo con alta densidad de cómputo y una configuración de la meta-heurística que pudiera mantener ocupados los miles de núcleos disponibles en un *GPU*.

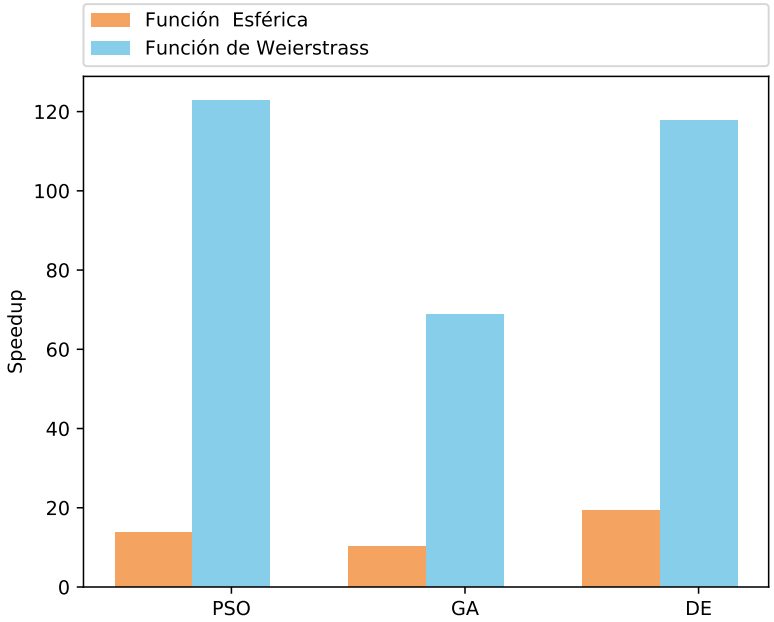


Figura 5.11: Máximo speedup en el *GPU* al comparar el tiempo de ejecución vs. un único hilo en *CPU*.

Meta-heurística	Speedup
PSO	13.77x
GA	10.30x
DE	19.23x

(a) Función Esférica

Meta-heurística	Speedup
PSO	122.75x
GA	68.75x
DE	117.86x

(b) Función de Weierstrass

Cuadro 5.3: Tabla de valores - Máximo speedup en el *GPU* al comparar el tiempo de ejecución vs. un único hilo en *CPU*.

5.3 CONCLUSIONES

En este trabajo se estudió cómo realizar una implementación de alto rendimiento para dos arquitecturas de cómputo distintas: *GPU*s y *CPU*s *x86*.

Un resultado preliminar que es interesante, fue cómo se logró optimizar la implementación paralela del algoritmo genético en el *CPU*, utilizando estrategias simples que permiten escalar eficientemente la ejecución de la meta-heurística. Para este caso particular fue necesario el evitar casos de *false sharing* en la generación de números pseudoaleatorios, mantener los patrones de acceso a memoria adecuados que disminuyan la contención sobre los recursos de memoria, y añadir instrucciones de vectorización al cómputo de la función objetivo, logrando obtener un *speedup* de $2.91x$. En un procesador de 4 núcleos con 2 procesadores lógicos por núcleo, al aumentar el número de hilos concurrentes para ejecutar de manera paralela las transformaciones de los operadores genéticos y evaluaciones de la función objetivo, se obtuvo un *speedup* de $5x$ adicional. Realizar optimizaciones y adaptaciones sobre implementaciones ya existentes, puede ser una solución más práctica que migrar a nuevas arquitecturas, particularmente si el problema en cuestión tiene características que serán difíciles de implementar eficientemente en una nueva arquitectura.

Con la presente desaceleración en la velocidad de los procesadores y el poder de cómputo que puede ofrecer un procesador, es claro que necesitamos cambiar el enfoque de nuestras aplicaciones para que puedan explotar los recursos de cómputo paralelo disponibles. Dependiendo de la aplicación, hoy en día existen diferentes arquitecturas de cómputo paralelo que pueden utilizarse. Es importante estar conscientes que al trabajar en implementaciones dirigidas a *GPU*s, *FPGA*s y *MIC*s, existen una serie de costos implícitos, además del costo de adopción del *hardware*. Se requiere capacitar a los desarrolladores para que puedan utilizar las herramientas disponibles en estos entornos. Muchas de estas herramientas no tienen la madurez ni flexibilidad que se cuenta ya en herramientas para arquitecturas *x86*. El hecho de que estas arquitecturas y las herramientas que las habilitan estén en su infancia, conlleva a que estén en constante cambio. Tanto el *hardware* como las implementaciones pueden ser obsoletas en cuestión de un par de años.

En ciertas aplicaciones se pueden obtener importantes beneficios al utilizar estas nuevas arquitecturas de cómputo, pero antes de iniciar el esfuerzo de portar un algoritmo a una arquitectura habrá

que entender si este puede ser ejecutado de forma eficiente en dicho entorno. Cada arquitectura provee un modelo de cómputo con diferentes características y por lo tanto capacidades de cómputo distintas. El crear una implementación de alto rendimiento para una aplicación específica, implica tomar en cuenta las limitaciones y fortalezas del *hardware* como parte del problema que se está resolviendo. Este proceso involucrará la adaptación de los algoritmos y estructuras de datos para ajustarse a la arquitectura objetivo, incluso el crear nuevas soluciones *ad-hoc* al contexto de aplicación en cuestión.

Existen aplicaciones en las que utilizar otras arquitecturas de cómputo es justificable y puede rendir grandes beneficios. En aplicaciones en donde se requiere realizar cómputo de punto flotante sobre un conjunto masivo de datos, el uso de *GPU*s puede ofrecer grandes beneficios: operaciones de álgebra lineal, procesamiento de señales, codificación y procesamiento de imágenes, etc. Tan solo para el caso de estudio del cúmulo de partículas, la implementación de *Locusta* logró alcanzar un *speedup* de $122.75x$ al compararlo con una implementación de alto rendimiento para el *CPU*.

Sin embargo, por la naturaleza de una arquitectura *SIMT* no es posible computar de manera eficiente cualquier tipo de función, limitando el rango de aplicaciones que pueden aprovechar estos beneficios. Otro obstáculo importante es la cantidad limitada de memoria con la que cuentan los *GPU*s al compararlos con los volúmenes de memoria que existen en arquitecturas de *CPU*s. El requerir de transferencias entre el sistema anfitrión y los diferentes nodos de *GPU*s añade complejidad a la implementación y muchas veces el costo de esta comunicación y coordinación limita los demás beneficios que estas arquitecturas pudieran ofrecer.

Si la función objetivo de un problema es computacionalmente intensiva, los datos necesarios para su cómputo pueden mantenerse dentro de la memoria *GPU*, incluyendo los datos requeridos por la meta-heurística, respetando los patrones de acceso a memoria que se pueden efectuar en un *GPU* de forma eficiente, entonces y sólo entonces se pueden esperar una aceleración significativa en comparación a la ejecución en un *CPU*.

La gran mayoría de las aplicaciones a problemas interesantes de optimización no cumplen con alguna de estas características. El obtener una solución implicará el desarrollo de nuevas técnicas y estrategias que permitan utilizar de manera eficiente los recursos de cómputo paralelo. La circunstancias demandan que conozcamos los recursos disponibles e ingeniemos nuevas soluciones. De acuerdo con el artículo "*No Free Lunch Theorems for Optimization*" [Wolpert &

Macready, 1997], no existe algoritmo de optimización que sea mejor que otro, para cualquier problema de optimización. Esto también aplica para los operadores y parámetros de configuración de las meta-heurísticas, por lo que no existe una única configuración que obtenga buen rendimiento para cualquier función objetivo. La implementación de *Locusta* para *GPU*s ofrece una interfaz programable que funciona como mesa de trabajo para investigar y probar la eficiencia de nuevos operadores sobre las meta-heurísticas implementadas. Al utilizar funciones de *benchmarking* como las presentes en este estudio, se puede lograr un grado de aceleración significativo en comparación a la ejecución en un *CPU* (véase Tabla 5.3).

Los mayores niveles de aceleración alcanzados en el *GPU*, sólo se logran al utilizar conjuntos poblacionales con un número masivo de individuos, lo cual impone una limitante importante sobre la aceleración que se puede obtener en las configuraciones que comúnmente se utilizan en estas meta-heurísticas. Sin embargo, una aplicación que requiere de dichas configuraciones, es la meta-optimización de las meta-heurísticas. La implementación de *Locusta* puede ser utilizada para problemas de meta-optimización, ejecutando varias instancias de forma paralela, cada una utilizando diferentes operadores y parámetros.

Aún existen gran número de optimizaciones y extensiones que pueden ser integradas a la infraestructura que *Locusta* provee: la interacción entre meta-heurísticas para crear soluciones híbridas, extensión para escalar el cómputo sobre múltiples *GPU*s y también múltiples nodos de cómputo utilizando *OpenMPI*. También queda abierto el explorar otras arquitecturas masivamente paralelas, que no tienen las mismas limitantes que los *GPU*s, y que pueden habilitar el atacar un conjunto diferente de problemas de optimización.

A

FUNCIONES OBJETIVO

1. Función Esférica (con Translación)

$$f(x) = \sum_{i=1}^{D} z_i^2 + f_{bias},$$
$$z = x - o, \quad x = [x_1, x_2, ..., x_D]$$

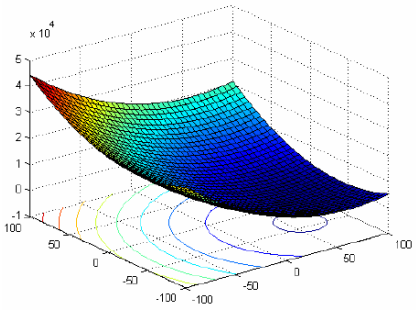


Figura A.1: Función Esférica

2. Función de Weierstrass (con Translación y Rotación)

$$f(x) = \sum_{i=1}^{D}(\sum_{k=0}^{k_{max}} (a^k cos(2\pi b^k (z_i + 0.5)))) - D \sum_{k=0}^{k_{max}} (a^k cos(2\pi b^k 0.5)) + f_{bias},$$
$$z = (x - o) \cdot M, x = [x_1, x_2, ..., x_D]$$

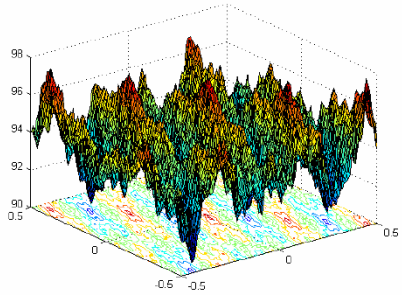


Figura A.2: Función de Weierstrass (con Translación y Rotación)

BIBLIOGRAFÍA

Acton, M. (2014). Data-oriented design and C++. `http://cppcon.org/third-keynote-2014`. (Citado en página 63.)

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring) (pp. 483–485). New York, NY, USA: ACM. (Citado en página 66.)

Bates, G. E. (1955). Joint distributions of time intervals for the occurrence of successive accidents in a generalized polya scheme. *The Annals of Mathematical Statistics*, 26(4), 705–720. (Citado en página 25.)

Bhati, I., Chang, M.-T., Chishti, Z., Lu, S.-L., & Jacob, B. (2016). Dram refresh mechanisms, penalties, and trade-offs. *IEEE Trans. Comput.*, 65(1), 108–121. (Citado en página 43.)

Blum, C. & Merkle, D. (2008). *Swarm Intelligence: Introduction and Applications*. Springer. (Citado en página 31.)

Bonabeau, E., Dorigo, M., & Theraulaz, G. (1999). *Swarm intelligence: From natural to artificial systems*. Santa Fe Institute Studies in the Sciences of Complexity. Oxford: Oxford University Press. (Citado en página 31.)

Bonner, J. T. (2009). *The Social Amoebae: The Biology of Cellular Slime Molds*. Princeton University Press. (Citado en página 3.)

Borodovsky, Y. (2012). International workshop on euv lithography. In *Intel*: Intel. (Citado en página 38.)

Chen, S., Montgomery, J., & Bolufé-Röhler, A. (2015). Measuring the curse of dimensionality and its effects on particle swarm optimization and differential evolution. *Applied Intelligence*, 42(3), 514–526. (Citado en página 35.)

CUDA C Best Practices (2016). *CUDA C Best Practices Guide*. nVidia. (Citado en página 84.)

CUDA C Programming Guide (2016). *CUDA C Programming Guide*. nVidia. (Citado en página 60.)

Danowitz, A., Kelley, K., Mao, J., Stevenson, J. P., & Horowitz, M. (2012). Cpu db: Recording microprocessor history. *Queue*, 10(4), 10:10–10:27. (Citado en página 40.)

Darwin, C. (1859). *On the origin of species by means of natural selection, or, The preservation of favoured races in the struggle for life*, volume -1859. London :John Murray,. — Freeman, R.B. Charles Darwin (2nd ed.). (Citado en página 1.)

Dawkins, R. (1993). The evolutionary future of man. (Citado en página 36.)

Dawkins, R. (1999). *The Selfish Gene*. Oxford paperbacks. Oxford University Press. (Citado en página 1.)

Debattisti, S., Marlat, N., Mussi, L., & Cagnoni, S. (2009). Implementation of a Simple Genetic Algorithm within the CUDA Architecture. *Gecco '09*. (Citado en página 12.)

Doughty, C. E., Wolf, A., & Field, C. B. (2010). Biophysical feedbacks between the pleistocene megafauna extinction and climate: The first human-induced global warming? *Geophysical Research Letters*, 37(15), n/a–n/a. (Citado en página 3.)

Drepper, U. (2009). What Every Programmer Should Know About Memory. `http://people.redhat.com/drepper/cpumemory.pdf`. (Citado en página 45.)

Goldberg, D. (1991). What every computer scientist should know about floating point arithmetic. (Citado en página 22.)

González del Cueto, R. (2011). Drosophila. `http://github.com/rdelcueto/Drosophila`. (Citado en página 11.)

González del Cueto, R. (2014). Locusta. `http://github.com/rdelcueto/Locusta`. (Citado en página 12.)

Gould, S. (1987). *The Flamingo's Smile: Reflections in Natural History*. Includes index. Norton. (Citado en página 14.)

Haupt, R. L. & Haupt, S. E. (1998). *Practical Genetic Algorithms*. John Wiley & Sons. (Citado en página 24.)

Heaton, J. (2013). *Artificial Intelligence for Humans, Volume 1: Fundamental Algorithms*. CreateSpace Independent Publishing Platform. (Citado en página 6.)

Heaton, J. (2014). *Artificial Intelligence for Humans, Volume 2: Nature Inspired Algorithms*. CreateSpace Independent Publishing Platform.

Hennessy, J. L. & Patterson, D. A. (2006). *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc. (Citado en páginas 52 y 66.)

Herrera, F., Lozano, M., & Verdegay, J. (1998). Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artificial Intelligence Review*, 12(4), 265–319. (Citado en página 21.)

Holland, J. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press. (Citado en página 18.)

Hölldobler, B. & Wilson, E. (2010). *The Leafcutter Ants: Civilization by Instinct*. W. W. Norton. (Citado en página 2.)

Hu, T., Harding, S., & Banzhaf, W. (2010). Variable population size and evolution acceleration: a case study with a parallel evolutionary algorithm. *Genetic Programming and Evolvable Machines*, 11(2), 205–225. (Citado en página 17.)

Jaros, J. & Pospichal, P. (2012). A fair comparison of modern cpus and gpus running the genetic algorithm under the knapsack benchmark. In *Proceedings of the 2012T European Conference on Applications of Evolutionary Computation*, EvoApplications'12 (pp. 426–435). Berlin, Heidelberg: Springer-Verlag. (Citado en página 12.)

Kennedy, J. & Eberhart, R. C. (1995). Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks* (pp. 1942–1948). (Citado en página 31.)

Kennedy, J. & Eberhart, R. C. (2001). *Swarm Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Kirk, D. B. & Hwu, W. W. (2012). *Programming Massively Parallel Processors: A Hands-On Approach*. Elsevier Science (book series). (Citado en página 51.)

Knuth, D. E., Pardo, L., & Department, S. U. C. S. (1976). *The Early Development of Programming Languages*. STAN-CS-76-562. Stanford University, Computer Science Department. (Citado en página 6.)

Konar, A. (2005). *Computational Intelligence: Principles, Techniques and Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. (Citado en páginas 6 y 14.)

Lindgren, M. (1990). *Glory and Failure: The Difference Engines of Johann Müller, Charles Babbage and Georg and Edvard Scheutz*. History of computing. MIT Press. (Citado en página 4.)

McKenney, P. E. (2012). Is parallel programming hard, and, if so, what can you do about it? (Citado en página 50.)

Michalewicz, Z. (1999). *Genetic Algorithms + Data Structures = Evolution Programs.* Springer. (Citado en página 16.)

Millonas, M. M. (1994). Swarms, phase transitions, and collective intelligence. In *Santa Fe Institute Studies in the Sciences of Complexity - Proceedings*, volume 17 (pp. 417–417).: addison-wesley. (Citado en página 31.)

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press. (Citado en página 18.)

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8). (Citado en página 36.)

Morse, G. & Stanley, K. O. (2016). Simple evolutionary optimization can rival stochastic gradient descent in neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16 (pp. 477–484). New York, NY, USA: ACM. (Citado en página 10.)

Nagel, E. & Newman, J. R. (2001). *Godel's Proof*. NYU Press. (Citado en página 5.)

Norvig, P. (2001). Teach yourself programming in ten years. http://norvig.com/21-days.html. http://norvig.com/21-days.html. (Citado en página 46.)

Osman, I. H. & Laporte, G. (1996). Metaheuristics: A bibliography. *Annals of Operations Research*, 63(5), 511–623. (Citado en página 9.)

Ostrovsky, I. (2010). Gallery of processor cache effects. `http://igoro.com/archive/gallery-of-processor-cache-effects/`. (Citado en página 93.)

Pacheco, P. (2011). *An Introduction to Parallel Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st edition. (Citado en página 42.)

Pinker, S. (1997). *How the Mind Works*. New York, NY: W. W. Norton & Company. (Citado en página 6.)

Piper, R. (2007). *Extraordinary Animals: An Encyclopedia of Curious and Unusual Animals*. Greenwood Press. (Citado en página 2.)

Pospichal, P. & Jaros, J. (2009). The GPU-based Acceleration of the Genetic Algorithm. *Gecco '09*. (Citado en página 12.)

Robic, B. (2015). *The Foundations of Computability Theory*. Springer Publishing Company, Incorporated, 1st edition. (Citado en página 4.)

Röhler, A. B. & Chen, S. (2011). An analysis of sub-swarms in multi-swarm systems. In D. Wang & M. Reynolds (Eds.), *AI 2011: Advances in Artificial Intelligence: 24th Australasian Joint Conference, Perth, Australia, December 5-8, 2011. Proceedings* (pp. 271–280). Berlin, Heidelberg: Springer Berlin Heidelberg. (Citado en página 35.)

Schneier, B. (1994). Algorithm alley. `http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1994/9410/9410k/9410k.htm`. (Citado en página 27.)

Sivanandam, S. N. & Deepa, S. N. (2007). *Introduction to Genetic Algorithms*. Springer. (Citado en páginas 22 y 23.)

Storn, R. & Price, K. (1997). Differential evolution; a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, 11(4), 341–359. (Citado en páginas 27 y 28.)

Sutter, H. (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. `http://www.gotw.ca/publications/concurrency-ddj.htm`. (Citado en página 38.)

Sutter, H. (2009). Eliminate False Sharing: Stop your CPU power from invisibly going down the drain. `http://www.drdobbs.com/parallel/eliminate-false-sharing/217500206`. (Citado en páginas 46 y 72.)

Thornton, J. (1970). *Design of a computer: the Control Data 6600*. Scott, Foresman. (Citado en página 48.)

Timelines, C. (1965). The CDC 6600 arrives at CERN. `https://timeline.web.cern.ch/the-cdc-6600-arrives-at-cern`. (Citado en página 48.)

Turing, A. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1), 230–265. (Citado en página 5.)

Volkov, V. (2010). Better Performance at Lower Occupancy. *GTC2010*. (Citado en página 81.)

Wang, L. (2006). *Quantum Mechanical Effects on MOSFET Scaling Limit*. PhD thesis, Georgia Institute of Technology. (Citado en página 39.)

Whitepaper NVIDIA Tesla P100 (2016). *Whitepaper: NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built*. nVidia. (Citado en página 54.)

Wilson, E. & Holldobler, B. (2005). Eusociality: Origin and consequences. *Proceedings of the National Academy of Sciences*, 102(38), 13367–13371. (Citado en página 1.)

Wilson, E. O. (1980). Caste and division of labor in leaf-cutter ants. *Behavioral Ecology and Sociobiology*, 7(2), 143–156. (Citado en página 2.)

Wolpert, D. H. & Macready, W. G. (1997). No free lunch theorems for optimization. *Trans. Evol. Comp*, 1(1), 67–82. (Citado en página 105.)

Zaharie, D. (2002). Critical values for the control parameters of differential evolution algorithms. In *Proceedings of MENDEL*, volume 2 (pp. 6267). (Citado en página 28.)