CENTRO DE INVESTIGACIÓN CIENTÍFICA Y DE EDUCACIÓN SUPERIOR DE ENSENADA, BAJA CALIFORNIA

Programa de Posgrado en Ciencias en Ciencias de la Computación

Un algoritmo de enumeración completa para la identificación de SCAPs y su implementación en GPUs

Tesis
para cubrir parcialmente los requisitos necesarios para obtener el grado de

Maestro en Ciencias

Presenta:

Najash Marrón Guluarte

Ensenada, Baja California, México

2014

Tesis defendida por

Najash Marrón Guluarte

y aprobada por el siguiente comité

Dr. Carlos Alberto Brizuela Rodríguez

Director del Comité

Dr. Israel Marck Martínez Pérez

Miembro del Comité

Dr. Andrey Chernykh

Miembro del Comité

Dr. Alexei Fedorovish Licea Navarro

Miembro del Comité

Dra. Ana Isabel Martínez García

Coordinadora del programa de
posgrado en Ciencias de la Computación

Dr. Jesús Favela Vara

Director de Estudios de Posgrado

Agosto, 2014

Resumen de la tesis de **Najash Marrón Guluarte**, presentada como requisito parcial para la obtención del grado de Maestro en Ciencias en Ciencias de la Computación. Agosto de 2014.

Un algoritmo de enumeración completa para la identificación de SCAPs y su implementación en GPUs

Resumen aprobado por:

Dr. Carlos Alberto Brizuela Rodríguez

Director de Tesis

Los péptidos catiónicos anfipáticos de corta longitud con actividad antimicrobiana son producidos de manera natural por una gran variedad de organismos como mecanismo de defensa. Debido a la creciente resistencia a antibióticos de los microorganismos patógenos, estos péptidos han llamado la atención de manera significativa como posibles fuentes de nuevos agentes antibacteriales. Aunque los péptidos antimicrobianos generalmente exhiben menor potencia contra las bacterias objetivo, en comparación con los compuestos antibióticos convencionales de bajo peso molecular, estos muestran una serie de ventajas compensatorias que incluye eliminación rápida, amplio rango de actividad, baja toxicidad y un mínimo desarrollo de resistencia en los organismos objetivo.

Para la identificación de los péptidos con potencial actividad selectiva antibacterial (SCAPs) se analizan cuatro propiedades fisicoquímicas previamente propuestas en la literatura. Debido a que el número de secuencias a explorar crece exponencialmente en función de la longitud de la cadena de aminoácidos que se desea analizar, desarrollamos un algoritmo paralelo de búsqueda exhaustiva que se implementa en una GPU. Esta implementación se logra utilizando la plataforma CUDA para generar así una alternativa rentable en la solución de este problema combinatorio. El diseño propuesto mejora el tiempo de ejecución de los trabajos de enumeración completa existentes en la literatura y hasta donde tenemos conocimiento, es la primera vez que se analizan en forma exhaustiva secuencias de 5 a 10 aminoácidos variables. Los resultados muestran que aproximadamente el 1 % de las secuencias evaluadas son candidatas a SCAPs, un número muy elevado si se requiere sintetizarlas para comprobar su actividad. Este resultado sugiere que es necesario incluir un mayor número propiedades para lograr una discriminación que arroje un número reducido de candidatos a SCAPs que puedan ser evaluados experimentalmente.

Palabras Clave: **Biocomputación, Diseño de péptidos, Péptidos antibacterianos, Algoritmos de optimización, GPUs, CUDA.**

Abstract of the thesis presented by **Najash Marrón Guluarte**, in partial fulfillment of the requirements of the Master in Sciences degree in Computer Science. August 2014.

A full enumeration algorithm for SCAPs identification and its implementation on GPUs

Abstract approved by:

Dr. Carlos Alberto Brizuela Rodríguez

Thesis director

Short cationic, amphipathic peptides with antimicrobial activity are produced by a wide variety of organisms as a defense mechanism. The increase in antibiotic resistance in pathogenic microorganisms has drawn significant attention towards the use of peptides as candidates for novel antibacterial agents. Although antimicrobial peptides (AMPs) generally exhibit lower potency against susceptible bacterial targets compared to conventional low molecular weight antibiotic compounds, they hold several compensatory advantages including fast killing, broad range of activity, low toxicity, and minimal development of resistance in target organisms.

To identify peptide sequences with potential selective antibacterial activity (SCAPs), four physicochemical properties previously proposed in the literature were analyzed. Because the number of sequences that needs to be explored grows exponentially with the length of the analyzed amino acid chain, we developed a parallel exhaustive search algorithm to be implemented in a GPU. The implementation is performed by using the CUDA platform to provide a cost effective alternative to solve this combinatorial problem. The proposed algorithm improves the running time of different complete enumeration analysis presented in the literature, and for the first time, to the best of our knowledge, sequences with 5 up to 10 variable amino acids are exhaustively analyzed. Results show that approximately 1 % of the evaluated sequences are SCAPs candidates, a large number of cases if synthesis is needed to verify whether or not they have antibacterial activity; this raises the need to include more properties in order to reduce the number of SCAPs candidates that will undergo experimental evaluation.

Keywords: **Bioinformatics, Peptide design, Antibacterial peptides, Optimization Algorithms, GPU, CUDA.**

Dedicatoria

Con todo mi cariño y mi amor para la persona que hizo todo lo posible para que yo pudiera lograr mis metas, por motivarme y darme la mano cuando sentía que el camino estaba muy pesado, a ti por siempre mi corazón y mi agradecimiento.

Mamá

Agradecimientos

Agradezco a mi familia, por todo su amor y apoyo incondicional.

A mi director de tesis, el Dr. Carlos Alberto Brizuela Rodríguez, quien despertó mi interés por el área de la bioinformática. Gracias por su guía y enseñanzas.

A los miembros de mi comité de tesis, el Dr. Israel Marck Martínez Pérez, el Dr. Andrey Chernykh y el Dr. Alexei Licea por su tiempo, observaciones y sugerencias durante el desarrollo de este trabajo.

Al Dr. Gabriel del Rio y su equipo del Instituto de Fisiología Celular por la contribución a este trabajo y las facilidades prestadas durante la estancia en la UNAM.

A mis amigos, en especial a David, Miriam, Armando y Hugo que me apoyaron en momentos importantes durante el proyecto, y a ti Esther Gamez, gracias por alentarme a comenzar y terminar este trabajo.

A Ricardo Guluarte y al Dr. Luis Gustavo Álvarez, por inspirarme.

A los investigadores y compañeros del posgrado del departamento de Ciencias de la Computación, por sus enseñanzas y experiencias durante esta estancia.

A CICESE por permitirme estudiar este posgrado, y al personal administrativo por la excelente atención que siempre me brindaron.

Al CONACyT por su apoyo económico para realizar mis estudios.

Contenido

Página

Resumen en español ii

Resumen en inglés iii

Dedicatoria iv

Agradecimientos v

Lista de Figuras ix

Lista de Tablas xii

1. Introducción 1
- 1.1. Antecedentes y motivación . . . 1
- 1.2. Definición del problema . . . 4
- 1.3. Objetivos . . . 4
  - 1.3.1. Objetivo general . . . 4
  - 1.3.2. Objetivos específicos . . . 4
- 1.4. Metodología de la solución propuesta . . . 5
- 1.5. Organización de la tesis . . . 6

2. Conceptos básicos de biología 7
- 2.1. Péptidos . . . 7
- 2.2. Péptidos Catiónicos Antimicrobianos . . . 8
- 2.3. Características de los Péptidos Catiónicos Antimicrobianos . . . 8
- 2.4. Modo de acción de los Péptidos Antibacterianos . . . 8
  - 2.4.1. Péptidos permeabilizadores de membrana . . . 10
  - 2.4.2. Péptidos no permeabilizadores de membrana . . . 13
- 2.5. Prediciendo péptidos antimicrobianos utilizando la Relación Cuantitativa Estructura-Actividad (QSAR) . . . 14
- 2.6. Trabajos Relacionados . . . 16
- 2.7. Propiedades fisico-químicas seleccionadas en Polanco *et al.* (2011) . . . 24

3. Fundamentos de computación 29
- 3.1. Introducción . . . 29
- 3.2. La arquitectura de NVIDIA . . . 30
- 3.3. Plan de trabajo para los GPUs de NVIDIA . . . 33
- 3.4. Capacidad de cómputo . . . 35
- 3.5. Modelo de programación . . . 37
- 3.6. Kernel . . . 38
- 3.7. Hilos . . . 39
  - 3.7.1. Warp . . . 42
  - 3.7.2. Divergencia . . . 42
  - 3.7.3. Sincronización . . . 43
  - 3.7.4. Calendarización . . . 44
- 3.8. Jerarquía de las memorias . . . 45

Contenido (continuación)

Página

3.9. Funciones especiales . . . 46
3.9.1. Funciones Atomic . . . 46
3.9.2. Funciones Shuffle . . . 48

4. Algoritmos **51**
4.1. Estrategías de paralelización evaluadas en la arquitectura de GPU . . . 52
4.2. Diseño de paralelización propuesto . . . 54
4.2.1. Consideraciones para el diseño de paralelización propuesto . . . 55
4.2.2. Estructuras de datos para almacenar SCAPs . . . 56
4.3. Algoritmos . . . 57
4.3.1. Función principal . . . 57
4.3.2. Preparar *Kernel* . . . 59
4.3.3. Función *Kernel* . . . 62
4.3.4. Algoritmo para detectar SCAPs . . . 64
4.3.5. Algoritmo de conversión base 20 . . . 66
4.3.6. Cálculo de cuatro propiedades fisicoquímicas . . . 67
4.4. Estrategias para optimizar la escritura de SCAPs en la memoria global del dispositivo . . . 72
4.4.1. Suma de Prefijos . . . 73
4.4.2. atomicAdd . . . 78
4.5. Secuencias permutadas . . . 81
4.5.1. Función Hash . . . 81
4.5.2. Agrupando llaves . . . 83

5. Experimentos y Resultados **85**
5.1. Detalles preliminares . . . 85
5.2. Configuración de hardware y software . . . 86
5.2.1. Comparando ambientes de desarrollo . . . 87
5.3. Implementación secuencial . . . 88
5.3.1. Tiempos de ejecución . . . 89
5.4. Implementación del esquema paralelo propuesto . . . 91
5.4.1. Mejoras técnicas . . . 91
5.4.2. Propuesta de uso de espacio en memoria global . . . 94
5.4.3. Tiempos de ejecución de la estrategia paralela propuesta . . . 95
5.4.4. Escritura a archivo . . . 98
5.5. Análisis de Resultados . . . 99
5.5.1. Aceleración . . . 100
5.5.2. Comparación de las estrategias paralelas para la detección de SCAPs 104
5.5.3. 1 % de casos positivos . . . 105
5.5.4. Análisis por región con más casos positivos . . . 106
5.5.5. Permutaciones con más casos positivos . . . 107
5.5.6. Comparación con la base de datos CAMP . . . 109

Contenido (continuación)

Página

5.5.7. Análisis de 20^{10} secuencias . . . 110
5.6. Discusión . . . 111
5.6.1. Diferencia entre la estrategia propuesta y los trabajos relacionados 111
5.6.2. Filtrado de casos positivos . . . 112
5.6.3. Candidatos a SCAPs . . . 113

6. Conclusiones **115**
6.1. Sumario . . . 115
6.2. Conclusiones . . . 115
6.3. Trabajo futuro . . . 117

Lista de Referencias **118**

A. Apéndice **123**
A.1. Comparación de GPUs de diferente generación . . . 123
A.2. Comparación C2070 vs GTX 680 . . . 123
A.3. Características y especificaciones técnicas de las GPUs en función de su capacidad de cómputo . . . 123

Lista de Figuras

Figura Página

1. En A (izquierda) se presenta la estructura química general de un aminoácido con el átomo central C-Alfa, un grupo amino, grupo carboxilo y la cadena lateral (R). En B (derecha) se muestran dos aminoácidos conectados a través de un enlace peptídico (dipéptido). Se pueden generar péptidos de longitud arbitraria al agregar más aminoácidos de esta forma. 7
2. El modelo poro Toroidal. En este modelo los péptidos que llegan a la membrana se unen e inducen a las monocapas lipídicas a doblarse a través del poro, donde la pared del poro la forman tanto los péptidos como la cabeza hidrofílica de los lípidos. La región hidrofílica de los péptidos se muestra en rojo y la sección hidrofóbica en azul (Brogden, 2005). 11
3. Modelos tablas de barril. En este modelo, los péptidos se unen y se insertan en la bicapa de la membrana de manera tal que las regiones hidrofóbicas se alinean con el núcleo de los lípidos y las regiones hidrofílicas forman la parte interior del poro. Las regiones hidrofílicas de los péptidos se muestran en color rojo, las hidrofóbicas en color azul (Brogden, 2005). 12
4. Modelo Alfombra. En este modelo los péptidos quebrantan la membrana al posicionarse paralelamente a la superficie de la bicapa lipídica formando una capa extensiva o alfombra. La regiones hidrofílicas de los péptidos se muestran en rojo, las hidrofóbicas en color azul (Brogden, 2005). 13
5. Modelo generado por computadora. El péptido está compuesto por un dominio ‘de caza’ (azul) y un dominio que deshace la membrana (pro-apoptótico) (residuos hidrofílicos en rojo e hidrofóbicos en verde), unidos por un dominio de enganche (amarillo). 21
6. Comparación de rendimiento entre los CPUs y GPUs comerciales (NVIDIA, 2008). 31
7. Asignación de transistores en la arquitectura de CPUs y GPUs (NVIDIA, 2008). 31
8. Estructura y recursos computacionales de la tarjeta NVIDIA GeForce GTX 680. La tarjeta tiene 4 GPC, dende cada uno contiene 2 SMX (NVIDIA, 2012). 33
9. Multiprocesador de flujo (SMX) de la tarjeta GeForce GTX 680. Contiene su propia unidad de instrucciones y 192 núcleos (NVIDIA, 2012). 34
10. Hoja de ruta de los GPUs de NVIDIA. En la gráfica se observa un incremento en el número de operaciones flotantes de doble precisión por segundo (DP GFLOPS) por Watt en cada generación, lo que traduce en ahorro de energía (Chiappetta, 2013). 34
11. Lista de características para cada capacidad de cómputo (NVIDIA, 2008). . 36
12. Información general sobre la organización de los hilos en CUDA. 40

Lista de Figuras (continuación)

Figura | Página

13. Ejemplo de organización multidimensional de un grid de CUDA. 41

14. Información general sobre el modelo de memoria del dispositivo CUDA. . . . 47

15. Operación *shfl* donde todos los hilos obtienen el valor de la línea 2 (“c”). En a) se muestra el *warp* antes de aplicar la función *shfl*, en b) el *warp* después de aplicar la función. 49

16. Operación *shfl_up* con $delta = 2$. En a) se muestra el *warp* antes de aplicar la función *shfl*, en b) el *warp* después de aplicar la función. 49

17. Operación *shfl_down* con $delta = 1$. En a) se muestra el *warp* antes de aplicar la función *shfl*, en b) el *warp* después de aplicar la función. 49

18. Operación *shfl_xor*. En a) se muestra el *warp* antes de aplicar la función *shfl*, en b) el *warp* después de aplicar la función. 50

19. Estrategia de paralelización analizando propiedades en serie. Las propiedades se ejecutan de menor a mayor tiempo de ejecución, en cada paso avanzan solo las secuencias positivas, por esta razón el recuadro disminuye de forma simbólica con cada propiedad analizada. 52

20. Estrategia de paralelización con *Kernels* concurrentes. 53

21. Estrategia de paralelización propuesta. Cada hilo calcula las cuatro propiedades de una secuencia de aminoácidos y determina si es o no un SCAP. 55

22. Formato de una secuencia de 25 aminoácidos introducida en el programa. Los aminoácidos variables se representan con ‘X’. 56

23. Estructuras de datos para almacenar SCAPs. 57

24. El arreglo A muestra una secuencia de valores aleatorios. El arreglo A' muestra el resultado de la suma de prefijos sobre A. 73

25. El arreglo A muestra el contenido relevante (descrito por las letras A, B y C) y contenido irrelevante (representado por ‘-’). En el arreglo B se cambia la información relevante por ‘1’ y la irrelevante por ‘0’. En *B’* se muestra la suma de prefijos. La posición de la información relevante en *A’* se obtiene de las casillas donde ocurre un incremento de valor (casillas coloreadas) en *B’*. . 74

26. Diagrama que ilustra el concepto de *warp*. 74

27. Suma de prefijos por *warp*. 77

28. Llenado del arreglo *sums*. 77

29. Suma de de prefijos del bloque. 78

Lista de Figuras (continuación)

Figura | Página

30. Valor de la suma de prefijos del hilo con Id = 4 dentro del *warp* con Id = 2 del arreglo *sums*. 78

31. Representación de la instrucción atomicAdd para obtener el índice a partir del cual cada bloque almacenará los SCAPs en la memoria global del dispositivo. 80

32. Estructuras de datos para almacenar SCAPs con el parámetro "Llave Hash". 82

33. Tiempo promedio de cómputo por secuencia en función de la longitud de la secuencia analizada. 91

34. Crecimiento exponencial en el tiempo de ejecución del algoritmo secuencial en función de la longitud de la secuencia. 91

35. Estrategias Divergente y No Divergente. 93

36. Tiempo de ejecución en horas por número de secuencias analizadas. 97

37. Tiempo de cómputo en función de la longitud variable de las secuencias (L). 98

38. Comparación de tiempos entre los esquemas secuencial y paralelo. 102

39. Aceleración en función del número de secuencias que se analizan. 103

40. Tiempo de cómputo por secuencia, comparativo entre la implementación secuencial y paralela en función del número de secuencias que se analizan. . . . 104

Lista de Tablas

Tabla Página

1. Selección de conceptos y enfoques para el diseño de péptidos basados en computadora (Fjell *et al.*, 2011a). 18

2. Valor de pK_a y pI de los aminoácidos. 26

3. Valor de hidrofobicidad de los aminoácidos de acuerdo a Eisenberg *et al.* (1984), ordenados de forma descendente. 28

4. Terminología de CUDA. 32

5. Operaciones *atomic*. 48

6. Relación de los aminoácidos con su equivalente en la representación en base 20. 58

7. Ejemplos de cadenas para analizar (columna Cadena), la conversión a base 20 de estas cadenas (columna Base 20) y del arreglo con la posición de los aminoácidos variables ordenados de forma inversa a la que aparecen en la secuencia (columna Pos Variables). 59

8. Configuración del *kernel* para analizar secuencias que van desde 3 hasta 7 aminoácidos variables. 60

9. Relación entre el identificador del hilo y la cadena de aminoácidos que genera, tomando como base la secuencia de estudio. 67

10. Aminoácidos en código de 3 letras (columna 1) y 1 letra (columna 2) y la operación para asignarle un valor numérico a cada uno (columna 3). 83

11. Comparación de tiempos con diferentes configuraciones de software en la implementación secuencial. Análisis de 20^4 hasta 20^6 secuencias. . . 87

12. Comparación de tiempos con diferentes configuraciones de software en la implementación paralela. Análisis de 20^4 hasta 20^6 secuencias. . . . 87

13. Tiempo en segundos por cada propiedad en el análisis de 20^4 hasta 20^7 secuencias. 88

14. Tiempos de ejecución en el cálculo de las cuatro propiedades propuestas en la implementación secuencial con 20^4 hasta 20^7 secuencias. El tiempo mostrado con 20^8 y 20^9 secuencias son estimados basados en el tiempo promedio por secuencia. 89

15. Promedio y desviación estándar de los enfoques Divergente y No Divergente. La columna "Tiempo" muestra el promedio de 20 ejecuciones. 94

Lista de Tablas (continuación)

Tabla Página

16. Cantidad de SCAPs y tiempos de ejecución en la implementación de la estrategia paralela propuesta. En la columna "Enfoque" la abreviación 'A' es para el enfoque suma atómica únicamente, 'A&P' hace referencia al enfoque suma atómica en combinación con la suma de prefijos. . . . 96

17. Comparación de tiempos en la escritura al filtrar (Operación Atómica y Operación Atómica & Suma de Prefijos) y no filtrar (Divergente y No Divergente) los resultados positivos. 99

18. Número de GFLOPs por Dispositivo. 101

19. Comparación del tiempo de ejecución de 20^4 secuencias con los diferentes enfoques. 104

20. Secuencias de nueve aminoácidos con siete de ellos variables que generan más casos positivos. 107

21. Secuencias de nueve áminoacidos con siete de ellos variables que generan menos casos positivos 108

22. Multiconjuntos de aminoácidos que genera el mayor número de permutaciones que resultan ser SCAPs. 108

23. Principales características de los diferentes enfoques en el análisis de secuencias. 111

24. Características generales de tarjetas de las arquitecturas Tesla, Fermi y Kepler. 123

25. Especificaciones técnicas principales de las GPUs Tesla C2070 y GeForce GTX 680. 124

26. Especificaciones técnicas de las GPUs en función de su capacidad de cómputo. 124

27. Características ofrecidas por la GPU en función de su capacidad de cómputo. 125

Lista de Algoritmos

1. Principal (main) 57
2. Función prepararKernel 61
3. Kernel, detección y almacenamieto de SCAPs. 63
4. Detección de SCAPs. 65
5. Convertidor base 20. 66
6. Media de Carga Neta. 68
7. Media de Hidrofobicidad. 68
8. Punto Isoeléctrico. 69
9. Algoritmo para calcular el Momento Hidrofóbico Helicoidal. 71
10. Suma de Prefijos. 75
11. atomicAdd, subproceso del Algoritmo 3 (*Kernel*) 79
12. Algoritmo para agrupar las *llaveshash* asociadas a cada SCAP 84

Capítulo 1. Introducción

1.1. Antecedentes y motivación

En las últimas décadas la resistencia a antibióticos se ha ido extendiendo entre los microorganismos patógenos, por esta razón los péptidos antimicrobianos han llamado la atención de manera significativa como posible nueva fuente de agentes antibacteriales (Fjell *et al.*, 2009). Aunque generalmente los péptidos antimicrobianos exhiben menor potencia contra las bacterias objetivo, en comparación con los compuestos antibióticos de bajo peso molecular, estos muestran ventajas compensatorias que incluyen eliminación rápida, un amplio espectro de actividad, baja toxicidad, y un mínimo desarrollo de resistencia en los organismos objetivo (Hancock y Sahl, 2006).

Los péptidos antimicrobianos son producidos de manera natural por una gran variedad de organismos como mecanismo de defensa contra los microbios patógenos (Zasloff, 2002). Estudios recientes han llevado al descubrimiento de varios tipos de estos péptidos de defensa del huésped con diferentes estructuras y perfiles de bioactividad (Wang, 2010).

Típicamente los péptidos antimicrobiales son relativamente cortos (de hasta 100 aminoácidos), están cargados positivamente y son anfifílicos. Estos se han logrado aislar en microorganismos unicelulares, insectos y otros invertebrados, hasta en plantas, anfibios, aves, peces y mamíferos incluyendo el humano. Cientos de estos péptidos han sido identificados mostrando su importancia en el sistema inmune (Jenssen *et al.*, 2006).

Entre los péptidos antimicrobianos existe un subconjunto de péptidos que muestran acción selectiva contra las bacterias y tienen una baja toxicidad frente a células humanas (del Rio *et al.*, 2001). En este trabajo se hace referencia a estós péptidos como Péptidos Catiónicos Antibacterianos Selectivos o SCAPs, por sus siglas en inglés. Debido a esta selectividad, los SCAPs han sido útiles en el desarrollo de nuevos compuestos para el tratamiento del cáncer (Ellerby *et al.*, 1999), y es previsible que muchas otras enfermedades se podrían atacar a través del uso de SCAPs (Kolonin *et al.*, 2004; Ellerby *et al.*, 2008).

En años recientes se ha utilizado la relación cuantitativa estructura-actividad (QSAR por sus siglas en inglés) para identificar y predecir la actividad antibacerial de los péptidos (Fjell *et al.*, 2009). El análisis QSAR busca relacionar las propiedades cuantitativas de un compuesto, conocidos como descriptores, con otras propiedades, como son la actividad farmacéutica o la toxicidad. Este análisis depende de propiedades físicas que pueden ser convenientemente medidas o calculadas para predecir de una manera no trivial otras propiedades de interés como la actividad biológica (Hilpert *et al.*, 2008).

Actualmente existen diferentes enfoques que utilizan QSAR para la predicción de péptidos antibacteriales. Entre los más destacados se incluyen los métodos estocásticos de Fjell *et al.* (2011a). En estos trabajos se han logrado modelar péptidos antibacterianos por medio de redes neuronales artificiales y algoritmos genéticos (Fjell *et al.*, 2009, 2011b). En el caso de las redes neuronales, el modelado consiste en dos fases: aprendizaje y predicción. El aprendizaje estudia la estructura tridimensional de los péptidos, un total de 44 descriptores QSAR se calculan con base en estas estructuras. Una vez entrenada la red neuronal, esta clasifica a los péptidos como activos o inactivos con base en los valores de los descriptores y el nivel de actividad experimental observado. Los péptidos que arroja la red neuronal son los individuos que forman parte de la población inicial en el algoritmo genético propuesto posteriormente.

Bajo un esquema de búsqueda exhaustiva, Polanco *et al.* (2011) utilizan cuatro características fisicoquímicas (media de carga, media de hidrofobicidad, punto isoeléctrico y momento hidrofóbico helicoidal) como descriptores para determinar si un péptido es o no un SCAP. La búsqueda exhaustiva se hace para péptidos con longitud de hasta nueve aminoácidos donde solo los últimos cuatro de estos varían. Por cada aminoácido variable de la secuencia peptídica existen 20 aminoácidos a considerar, el universo de combinaciones analizadas en este trabajo es entonces de 20^4. Las características que se utilizan para describir a un SCAP fueron tomadas del trabajo de del Rio *et al.* (2001), estas fueron medidas de entre 30 péptidos antibacteriales conocidos en la literatura. La propuesta para la búsqueda exhaustiva es un sistema paralelo con base en los arreglos lógicos programables FPGAs (del inglés Field Programmable Gate Array) para proveer una solución rentable al problema combinatorio.

El cálculo exhaustivo de las propiedades físicoquímicas de los péptidos tiene diferentes aplicaciones, incluyendo el análisis de datos de la espectrometría de masa, la identificación de regiones desordenadas en proteínas, el análisis de proteínas trans-membrana y la identificación de péptidos antibacterianos, entre otros (Polanco *et al.*, 2011). Debido al vasto conjunto de secuencias peptídicas a analizar, por ejemplo, para las secuencias de péptidos con 25 residuos de longitud, el número de secuencias diferentes es aproximadamente 9.095×10^{27}. Si el cálculo de las propiedades tomasen un nanosegundo en un solo procesador y tuviésemos disponibles 1000 procesadores, entonces el análisis de todos los péptidos de longitud 25 tomaría aproximadamente 32 millones de años. Queda claro entonces que es inviable la búsqueda exhaustiva de péptidos de esta longitud. Resulta natural entonces preguntarse cuál es la máxima longitud viable con tecnología actual de bajo costo.

Existen diferentes soluciones técnicas para abordar situaciones en las que se requiere un gran número de cálculos. Las soluciones se pueden dividir en tres clases: *a)* soluciones locales (esto incluye diseño de hardware), *b)* soluciones distribuidas (clúster de computadoras, computación en la nube) y *c)* una combinación de las opciones anteriores.

La naturaleza del problema nos conduce a la idea de utilizar técnicas de computación paralela para analizar las propiedades de los SCAPs. Una solución alternativa emergente para este tipo de problemas son las GPGPU (del inglés, Unidad de Procesamiento Gráfico de Proposito General) equipados con un vasto ancho de banda y cientos de unidades programables que permiten operaciones vectoriales y operaciones de punto flotante con doble precisión (Luebke *et al.*, 2006). Es necesario considerar que los cálculos con un GPU requiere de algoritmos específicos ya que el paradigma de programación difiere substancialmente de los sistemas tradicionales basados en CPU.

Las propiedades mencionadas proveen a las GPUs la posibilidad de competir con la capacidad de cómputo de un pequeño clúster de computadoras, y es gracias a esto que se puede crear una solución rentable con la capacidad de incrementar el número de secuencias a evaluar en un menor tiempo de ejecución.

Como se mencionó anteriormente, las secuencias que se analizan en el trabajo de Polanco *et al.* (2011) tienen una longitud máxima de nueve aminoácidos donde cuatro de estos son variables, lo que genera 160 mil (20^4) péptidos diferentes, el máximo análisis exhaustivo reportado hasta hoy según nuestro conocimiento. Una de las principales preguntas que surge al atacar este problema es conocer hasta cuántos aminoácidos variables se pueden analizar utilizando GPUs de última generación, ya que el problema crece exponencialmente con cada aminoácido agregado. La respuesta depende en gran medida a la estrategia propuesta sujeta a las reglas de la arquitectura y a la velocidad y capacidad de almacenamiento del dispositivo utilizado.

1.2. Definición del problema

El problema abordado en esta tesis se puede definir como:

Dado un número entero l y un cierto número x de propiedades fisicoquímicas, se busca determinar cuántos y cuáles péptidos de longitud l cumplen con las x propiedades establecidas.

1.3. Objetivos

1.3.1. Objetivo general

El objetivo general de esta investigación es:

Diseñar algoritmos paralelos en la arquitectura de GPUs y la plataforma de CUDA, para calcular cuatro propiedades fisicoquímicas de todos los péptidos de longitud l.

1.3.2. Objetivos específicos

1. Reducir el tiempo de ejecución en la detección de SCAPs de hasta 9 aminoácidos de longitud en comparación con trabajos existentes.

2. Proponer una estrategia para filtrar los casos positivos dentro del proceso paralelo del GPU.

3. Incrementar el espacio de búsqueda para analizar cadenas con hasta 10 aminoácidos variables.
4. Realizar un análisis de las secuencias arrojadas por el algoritmo y compararlas con las existentes en la literatura para determinar la calidad de las características evaluadas.

1.4. Metodología de la solución propuesta

Para cumplir con los objetivos propuestos, el desarrollo del presente trabajo fue realizado con la siguiente metodología.

Para abordar este problema primero se optó por analizar a detalle la solución propuesta por Polanco *et al.* (2011) y las propiedades que utilizan en esta para la detección de SCAPs.

En el siguiente paso se diseña e implementa un algoritmo secuencial en C++ para detectar candidatos a SCAPs, usando las mismas propiedades propuestas por Polanco *et al.* (2011). Se hicieron varias revisiones del algoritmo y de las propiedades para mejorar el tiempo de ejecución y lograr la comparación más justa posible con la versión paralela (evitando reportar mejoras de 100x como refiere Lee *et al.* (2010)).

Posteriormente se contemplan diferentes modelos de paralelización para ser implementados en el GPU, utilizando las funciones para el cálculo de las propiedades de la versión secuencial. El diseño final propuesto es el que contempla menos llamadas al GPU y con menos transferencia de datos entre CPU y GPU.

En la estrategia paralela propuesta, a cada secuencia a evaluar le corresponde un hilo diferente, es decir, n secuencias son analizadas por n hilos. Dentro de cada hilo se realiza el cálculo de las propiedades propuestas para determinar si la secuencia correspondiente es o no un SCAP. Si la secuencia es positiva esta se almacena en la memoria global del dispositivo. El índice que corresponde a cada caso positivo se calcula también dentro de cada hilo. Al terminar de evaluar las n secuencias indicadas, el arreglo global con los m casos positivos se envía de vuelta al CPU.

El tiempo de ejecución de esta estrategia se compara con los reportados en los trabajos de Polanco *et al.* (2011) y Garcia-Ordaz *et al.* (2012), en donde se analizan péptidos de longitud nueve donde cuatro aminoácidos son variables y el resto permanece fijo.

Finalmente, pensando en una futura aplicación, la implementación en paralelo se modifica para evaluar secuencias de hasta 25 aminoácidos de longitud, donde hasta un máximo de 10 son de valor y posición variable.

1.5. Organización de la tesis

El presente trabajo está organizado de la siguiente manera.

En el Capítulo 2 se presenta una definición general de los péptidos y una descripción detallada de las características y modo de acción de los péptidos antimicrobianos y del concepto de QSAR. También se incluye una reseña de los trabajos previos y finalmente la definición de las características que utilizamos para la definición de un SCAP.

En el Capítulo 3 se describe brevemente la arquitectura de NVIDA así como los conceptos fundamentales de la plataforma y el modelo de programación CUDA. Además se incluye la explicación de un conjunto de instrucciones específicas de CUDA que se utilizaron para el desarrollo de la solución propuesta.

En el Capítulo 4 se describen las estrategias analizadas y la solución propuesta con el conjunto de algoritmos secuenciales y paralelos que la componen.

En el Capítulo 5 se muestran la configuración de hardware utilizada y los resultados obtenidos por los algoritmos implementados, estos se comparan con los reportados en trabajos previos. Además se incluye un breve análisis de las secuencias positivas arrojadas por el algoritmo.

En el Capítulo 6 se exponen las conclusiones obtenidas, así como las propuestas para la continuación de este trabajo de investigación.

En el Apéndice A se presenta una comparación entre dispositivos con diferentes arquitecturas NVIDIA. También se presenta una comparación entre el dispositivo Tesla C2070 utilizado en el trabajo de Garcia-Ordaz *et al.* (2012) y el GPU GeForce GTX680 de este trabajo. El apéndice incluye también la lista de características y especificaciones técnicas definidas por capacidad de cómputo.

Capítulo 2. Conceptos básicos de biología

En este capítulo se describen algunos conceptos básicos referentes a los péptidos para ayudar a la comprensión de algunas de las características de los algoritmos y de los métodos utilizados por estos algoritmos en capítulos posteriores.

2.1. Péptidos

Los péptidos son cadenas de aminoácidos. Los aminoácidos están constituidos por un grupo amino, un grupo carboxilo y una cadena lateral que se conecta a un átomo de carbono central denominado C_α (Fig. 1-A). Los péptidos están unidos por enlaces covalentes "peptídicos" entre el grupo carboxilo de un aminoácido y el grupo amino del siguiente. La Figura 1-B muestra un dipéptido, el péptido más pequeño posible. La cadena principal o "columna vertebral" formada por la sucesión C_α-C-N-C_α del péptido se puede prolongar arbitrariamente mediante la unión de más aminoácidos. Las cadenas laterales son las regiones variables de los aminoácidos, estas pueden contener una carga positiva o negativa, ser hidrofóbicas o polares, etc. Los péptidos y proteínas se componen de un repertorio de 20 aminoácidos, y la frecuencia y la secuencia de sus respectivas cadenas laterales determinan las propiedades físico-químicas y función biológica de los péptidos (Hohm y Hoffmann, 2005; Branden *et al.*, 1991).

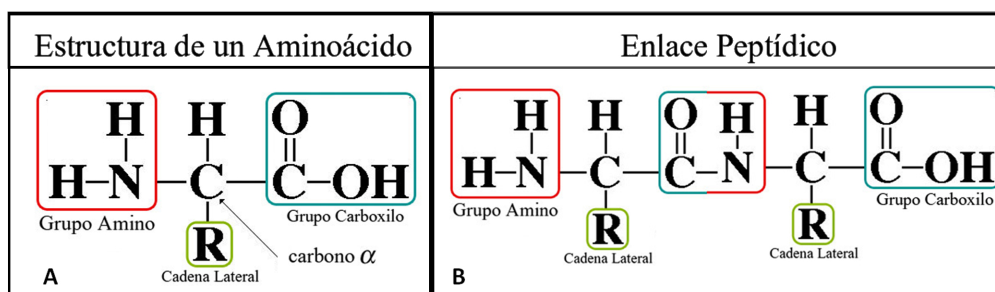

Figura 1: En A (izquierda) se presenta la estructura química general de un aminoácido con el átomo central C-Alfa, un grupo amino, grupo carboxilo y la cadena lateral (R). En B (derecha) se muestran dos aminoácidos conectados a través de un enlace peptídico (dipéptido). Se pueden generar péptidos de longitud arbitraria al agregar más aminoácidos de esta forma.

Muchas proteínas y péptidos adoptan una conformación específica estable y bien definida llamada "pliegue nativo", la cual se determina únicamente por la secuencia correspondiente de aminoácidos (Anfinsen *et al.*, 1973).

2.2. Péptidos Catiónicos Antimicrobianos

Los péptidos catiónicos antimicrobianos (CAPs por sus siglas en inglés) constituyen la primera línea de defensa de los seres vivos ante diferentes patógenos (Polanco *et al.*, 2011). Este diverso y abundante grupo de moléculas es producido por diferentes tejidos y tipos de células en una variedad de especies que van desde bacterias y hongos hasta invertebrados, plantas y animales (Jenssen *et al.*, 2006). Estos péptidos presentan actividad contra bacterias, hongos, virus y parásitos como insectos (Brogden, 2005).

2.3. Características de los Péptidos Catiónicos Antimicrobianos

Independientemente de las diferentes fuentes, secuencias y estructuras, todos lo péptidos catiónicos antimicrobianos comparten las siguientes características: pequeños en tamaño (menor a 50 aminoácidos), carga neta positiva (+2 hasta +9), anfifílicos ($>=$30 % de aminoácidos hidrofóbicos) y actividad antimicrobial y/o inmunomoduladora (sustancia que altera la respuesta inmune mediante el aumento o la reducción de la capacidad del sistema inmune para producir anticuerpos o células sensibilizadas que reconocen y reaccionan con el antígeno que inició su producción). Además de la actividad antibacteriana los péptidos catiónicos antimicrobianos también muestran actividad antiviral, anticancerígena, antifúngica y antiparasitaria, así como propiedades para sanar heridas (Hilpert *et al.*, 2008).

A pesar de las propiedades físicas generales similares, los péptidos antimicrobianos tienen una amplia gama de estructuras secundarias donde sobresalen cuatro grandes conjuntos. Las estructuras más prominentes son péptidos anfifílicos con dos o cuatro hebras β-láminas, α hélices anfipáticas, estructuras de ciclo y las estructuras extendidas (Jenssen *et al.*, 2006). En esta revisión se hace un resumen de la función directa del conjunto de los péptidos antibacterianos con estructura α-hélice en los que se centra la atención de esta investigación.

2.4. Modo de acción de los Péptidos Antibacterianos

Tener una buena comprensión de los mecanismos moleculares responsables de la actividad antibacteriana directa de los péptidos antibactereianos permitirá el desarrollo de mejores modelos de predicción, y de los diferentes mecanismos de acción propuestos (Fjell *et al.*, 2011a).

Se entiende bien que, independientemente de su objetivo real de acción, todos los péptidos catiónicos antibacterianos (y antimicrobianos en general) deben interactuar con la membrana citoplásmica bacteriana (Hancock y Rozek, 2002).

Aunque originalmente se propuso que la permeabilización de la membrana celular era el único modo de acción de los péptidos antibacterianos, hay evidencia que indica que algunos péptidos antimicrobianos ejercen su efecto a través de formas alternas de acción o que de hecho pueden actuar sobre múltiples objetivos de las células bacterianas. Independientemente del modo preciso de acción, la actividad de los péptidos antibacterianos depende mayormente de la interacción con la membrana de la célula bacteriana (Hancock y Rozek, 2002). El primer paso de esta interacción es la atracción inicial entre el péptido y la célula objetivo, que se cree ocurre a través de la unión electrostática entre el péptido catiónico y los componentes cargados negativamente presentes en la capa exterior de la bacteria, como son los grupos de fosfato dentro de los lipopolisacáridos de las bacterias gram-negativas o de los ácidos lipoteicoicos presentes en las superficies de los bacterias gram-positivas. En el caso de bacterias gram-negativas, los péptidos se insertan en la estructura externa de la membrana por un proceso impulsado entre la interacción hidrófobica y que posiblemente implica un plegado previo de los péptidos en una estructura asociada con la membrana; esto conduce a la perturbación de la estructura de la membrana externa y la permeabiliza para las siguientes moléculas peptídicas en un proceso denominado absorción auto-promovida. El resultado es que los péptidos llegan a la membrana citoplasmática, donde entran en la región interfacial de la membrana (la interfaz entre las porciones hidrofílica e hidrofóbica de la membrana) en un proceso impulsado por interacciones electrostáticas e hidrófobas. La mayor proporción de lípidos cargados negativamente sobre la superficie monocapa de la membrana citoplásmica bacteriana juega un papel importante en la selectividad de los péptidos antimicrobianos hacia las células bacterianas por encima de las células eucariotas, en el que los lípidos no cargados predominan en la superficie de la célula huésped. Los procesos que se producen en la superficie de la membrana son objeto de un intenso debate, varios modelos destacados (como son el tablas-barril, alfombra, poro toroidal, y agregado) han sido propuestos. Cada uno de estos indica un tipo diferente de intermediario que puede conducir a uno de tres eventos: la formación de un canal transitorio, disolución de la membrana, o la translocación a través

de la membrana. Como resultado, el péptido puede permeabilizar la membrana y/o translocarse a través de la membrana y llegar al citoplasma sin causar disrupción de la membrana principal. Por lo tanto, los modos de acción de los péptidos antibacterianos pueden ser clasificadas como interactuando sobre la membrana o bien no interactuando sobre la membrana. Mientras que los péptidos catiónicos antibacterianos más estudiados hasta ahora se han caracterizado como permeabilizadores de membrana, debe tenerse en cuenta que prácticamente cualquier péptido anfifílico catiónico causará perturbación de la membrana en los modelos si se aplica una concentración lo suficientemente alta (Patrzykat *et al.*, 2002), (Zhang *et al.*, 2001). Los estudios realizados con células bacterianas enteras han revelado que varios péptidos antibacterianos penetran en las células sin causar permeabilización de la membrana, si no que causan la muerte de la célula bacteriana al obstruir procesos intracelulares escenciales (Jenssen *et al.*, 2006). A continuación se muestra una visión general de los modelos actuales en que los péptidos antibacterianos "matan" a través de la membrana, ya sea por mecanismos de permeabilización o mecanismos no permeabilizadores de la membrana.

2.4.1. Péptidos permeabilizadores de membrana

Varios modelos se han propuesto para explicar cómo los péptidos antibacterianos se insertán en la membrana bacteriana para formar poros transmembrana que resultan en la permeabilización de la membrana. La naturaleza anfipática de los péptidos antimicrobianos es una característica clave de este proceso, debido a que las regiones hidrofóbicas son necesarias para interactuar directamente con los componentes lipídicos de la membrana, mientras que las regiones hidrofílicas interactúan ya sea con la cabeza de los grupos fosfolípidos o afrontan el lumen del poro. Generalmente, estos modelos pueden explicar la capacidad del péptido antibacteriano de formar el poro cuando presentan una estructura α hélice, sin embargo, la mecanismos utilizados por los péptidos β-lamina, tales como las defensinas no han sido tan ampliamente estudiados. Mientras los péptidos antimicrobianos β-lamina pueden adoptar pliegues anfipáticos, hay poca evidencia experimental que indique cuáles de los modelos que se describen a continuación son aplicables a defensinas (Jenssen *et al.*, 2006).

En todos los modelos los péptidos interactúan principalmente con los grupos de lípidos que contienen carga negativa en la cabeza y que se encuentran en la superficie de la membrana,

adoptando una orientación paralela a la membrana. A continuación se describen los modelos más destacados para la permeabilización de membrana.

- *Modelo poro toroidal*: En este modelo los péptidos se insertan en una orientación perpendicular a la membrana para formar un poro, con la membrana también formando una curva hacia adentro para crear un orificio con los grupos cabeza apuntando hacia el centro del poro, donde los péptidos se alinean con el orificio (Figura 2). Una predicción de este modelo es que la membrana exhibirá una tensión de curvatura positiva (debido a la flexión de la membrana para formar el agujero toroidal con los péptidos dentro). Aunque la formación de un canal toroidal formal no ha sido totalmente demostrada, este modelo ha sido estudiado en los trabajos de Hallock *et al.* (2003), Henzler Wildman *et al.* (2003) y Matsuzaki *et al.* (1996).

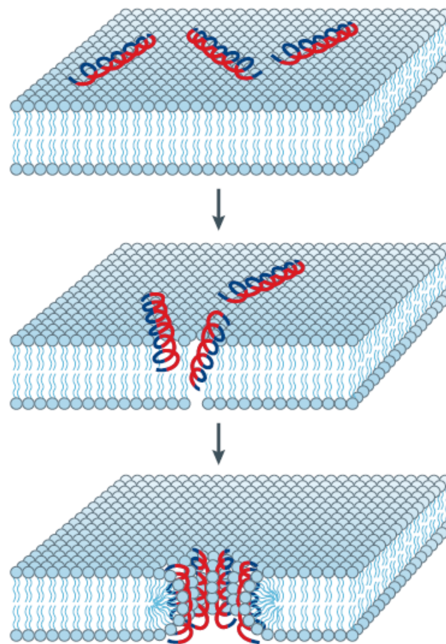

Figura 2: El modelo poro Toroidal. En este modelo los péptidos que llegan a la membrana se unen e inducen a las monocapas lipídicas a doblarse a través del poro, donde la pared del poro la forman tanto los péptidos como la cabeza hidrofílica de los lípidos. La región hidrofílica de los péptidos se muestra en rojo y la sección hidrofóbica en azul (Brogden, 2005).

- *Modelo tablas-barril:* En el modelo de *tablas-barril* los péptidos se acomodan de tal forma que aparentan ser "tablas" y formar un "barril" que se orienta perpendicularmente al plano de la membrana (Figura 3). Las regiones hidrofóbicas de cada péptido en el grupo se asocian con la capa lipídica, mientras que las regiones hidrofílicas apuntan

hacia el lumen del poro transmembrana recién formado. Este modelo predice que habrá un tamaño de canal consistente (o tamaños, también llamados subestados). La evidencia experimental que apoya este mecanismo de permeabilización de membrana se encuentra en los trabajos de (He *et al.*, 1996; Spaar *et al.*, 2004).

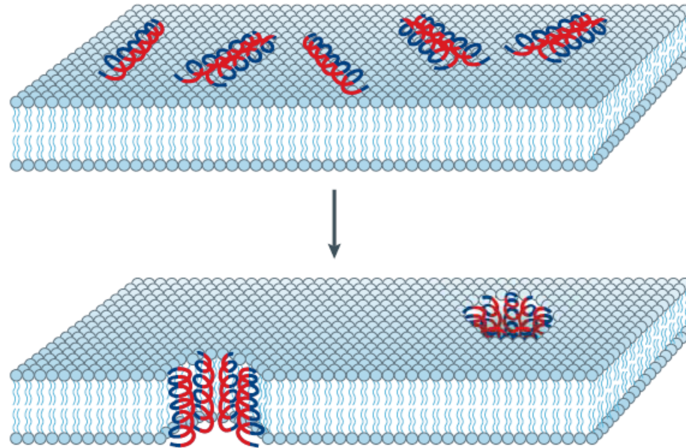

Figura 3: Modelos tablas de barril. En este modelo, los péptidos se unen y se insertan en la bicapa de la membrana de manera tal que las regiones hidrofóbicas se alinean con el núcleo de los lípidos y las regiones hidrofílicas forman la parte interior del poro. Las regiones hidrofílicas de los péptidos se muestran en color rojo, las hidrofóbicas en color azul (Brogden, 2005).

- *Modelo alfombra:* En contraste con los modelos tablas de barril y poros toroidales, el modelo de alfombra propone que los péptidos se alinean paralelamente a las bicapa lipídica, revistiendo áreas locales de manera similar a una alfombra (Figura 4). En una concentración suficientemente alta, se cree que logran una actividad de tipo detergente (causando parches en la membrana que se rompen en micelas), causando perturbaciones locales en la membrana que puede conducir a la formación de agujeros. Este comportamiento se logra usando péptidos antimicrobianos catiónicos en concentraciones suficientemente altas, debido a su carácter anfipático. Sin embargo, hay poca evidencia que demuestre que la mayoría de los péptidos causan la disolución membrana en concentraciones mínimas eficaces *in vivo* (Zhang *et al.*, 2001).

Es importante reconocer que cada uno de estos modelos puede tener validez bajo diferentes circunstancias, y la examinación de una amplia gama de péptidos con diferentes tamaños y estructuras ha indicado que estos dejan diferentes "firmas" en cuanto a la interacción con la membrana se refiere (Zhang *et al.*, 2001).

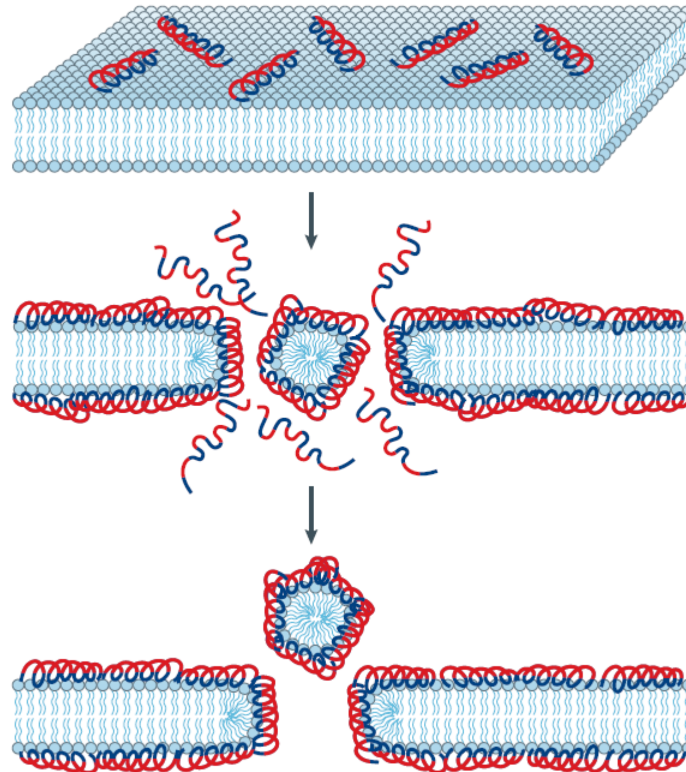

Figura 4: Modelo Alfombra. En este modelo los péptidos quebrantan la membrana al posicionarse paralelamente a la superficie de la bicapa lipídica formando una capa extensiva o alfombra. La regiones hidrofílicas de los péptidos se muestran en rojo, las hidrofóbicas en color azul (Brogden, 2005).

2.4.2. Péptidos no permeabilizadores de membrana

Todos los péptidos deben interactuar con la membrana citoplasmática, aunque sea sólo para llegar a su sitio de acción. Actualmente está bien establecido que varios péptidos antimicrobianos no causan permeabilización de la membrana en la concentración mínima efectiva y aún así logran la muerte de la bacteria. Un creciente número de péptidos han mostrado penetrar a través de la membrana y acumularse intracelularmente, en donde atacan diferentes procesos celulares esenciales y con esto propician la muerte celular. Nuevos modos de acción que han sido demostrados recientemente incluyen la inhibición de la síntesis del ácido nucleico, la síntesis de proteínas, la actividad enzimática, y la síntesis de la pared celular (Brogden, 2005).

El péptido antimicrobiano buforin II, proveniente de la rana, penetra a través de la membrana bacteriana sin causar permeabilización y se une al ADN y ARN dentro del citoplasma

de la bacteria *E. Coli* (Park *et al.*, 1998). Del mismo modo, péptidos con estructura α-hélice tales como los derivados de pleurocidina, un péptido antimicrobiano derivado del pescado, y dermaseptina, aislado de la piel de rana, causan la inhibición de la síntesis de ADN y ARN con la concentración mínima inhibitoria sin desestabilizar la membrana de las células de *E. coli* (Patrzykat *et al.*, 2002).

También se ha demostrado la inhibición de la síntesis del ácido nucleico usando péptidos antimicrobianos con diferentes clases estructurales, péptidos que interfieren con la síntesis de las proteínas, la inhibición de la actividad enzimática y otros que atacan la formación de componentes estructurales, tales como la pared celular (Otvos *et al.*, 2000; Kragol *et al.*, 2001; Brumfitt *et al.*, 2002).

Es probable que el modo de acción de los péptidos varíe individualmente según la célula bacteriana objetivo, la concentración en la que se examinaron y las propiedades físicas que interactúan en la membrana. También es probable que bajo el contexto de la infección, los péptidos antimicrobianos utilicen cierto mecanismo de acción, tales como la desestabilización de la membrana de la célula combinado con la inhibición de uno o más objetivos intracelulares. Se ha propuesto un mecanismo “multiobjetivo”, que plantea la hipótesis de que los péptidos con alta carga catiónica se pueden unir a moléculas aniónicas en el citoplasma, tales como los ácidos nucleicos o enzimas con superficies iónicas, interfiriendo así con procesos en los que estas moléculas están implicadas. Se ha demostrado recientemente que los péptidos catiónicos de diversas clases estructurales se pueden unir e inhibir la actividad de estas enzimas. Este alto grado de complejidad en el mecanismo es casi con certeza la causa de observación que hace extremadamente difícil seleccionar mutantes resistentes a los péptidos catiónicos (Fjell *et al.*, 2011a; Jenssen *et al.*, 2006; Stenlund *et al.*, 2002).

2.5. Prediciendo péptidos antimicrobianos utilizando la Relación Cuantitativa Estructura-Actividad (QSAR)

Los métodos de cribado virtual se pueden utilizar cuando la síntesis y pruebas exhaustivas son técnicas prohibitivamente costosas y cuando las técnicas de biología asistida tal como “phage display” no se pueden aplicar (Schneider y Böhm, 2002). Estos enfoques tienen la

ventaja de tener pocas suposiciones a priori, al intentar predecir las estructuras peptídicas basadas en las secuencias primarias. En contraste con los estudios de simulación computacional, los estudios de proyección virtual no necesariamente intentan crear modelos con resultados inmediatos fácilmente interpretables. En lugar de ello, se utilizan métodos numéricos para determinar las propiedades cuantificables de péptidos (descriptores), tales como la carga y la hidrofobicidad, a partir de la estructura primaria y de las características fisicoquímicas de los péptidos y se relacionan dichas propiedades con la actividad biológica utilizando modelos conocidos como de Relación Estructura-Actividad (SAR por sus siglas en inglés). El cribado virtual o dicho más específicamente, los modelos Cuantitativos SAR (QSAR por sus siglas en inglés) aplican análisis numéricos para describir la relación entre estos descriptores como variables de entrada y la actividad biológica como la variable de salida (Fjell *et al.*, 2011a).

QSAR trata de relacionar los descriptores de un compuesto con otras propiedades tales como la actividad farmacéutica o toxicidad. La suposición escencial de QSAR es que las cantidades que se pueden medir convenientemente o calcular para un compuesto se pueden utilizar para predecir de forma precisa otra característica de interés (por ejemplo, actividad antibacteriana) de una forma no trivial (Hilpert *et al.*, 2008). QSAR se ha convertido en una parte integral de los programas de soporte en el descubrimiento de fármacos y más recientemente en estudios toxicológicos (Perkins *et al.*, 2003). Sin embargo, el uso de modelos QSAR aplicado a la búsqueda de péptidos antimicrobianos es relativamente reciente.

Existen dos aspectos separados pero relacionados entre sí en el modelado de péptidos antibacteriales usando QSAR: la selección de los descriptores QSAR y la selección de técnicas de análisis utilizadas para relacionar estos valores con la actividad antibacteriana. Un ejemplo simple de un descriptor QSAR es la carga total de un péptido. En la literatura están disponibles un gran número de descriptores QSAR para pequeños compuestos, además de productos de software comerciales que pueden ser considerados. Un subconjunto más pequeño se utiliza en los estudios de QSAR de péptidos antibacterianos, que pueden ser separados en dos categorías: los descriptores basados en valores empíricos y los descriptores calculados. Un ejemplo de un valor empírico es el tiempo de retención de HPLC (High Performance

Liquid Chromatography o cromatografía líquida de alto desempeño), que es un sustituto de la medida de solubilidad o de hidrofilia / hidrofobia. Un ejemplo de un descriptor calculado es la carga total del péptido, por ejemplo a un pH = 7 (Hilpert *et al.*, 2008).

Además de la elección de los descriptores QSAR, existen métodos de aprendizaje estadístico para relacionar los descriptores con el valor predicho (Fjell *et al.*, 2011a). Hay dos categorías principales de predicción que responden a dos preguntas diferentes: modelos de regresión, modelos para la predicción de la actividad de un péptido como una variable continua o de clasificación donde el modelo es entrenado para clasificar simplemente como activo o inactivo. Históricamente, los métodos lineales han sido utilizados principalmente en la determinación de QSAR para péptidos antimicrobianos. La elección de una técnica de predicción también implica una compensación entre la exactitud del modelo y su significado. Para fines de clasificación, se considera que las técnicas no lineales de máquinas de vectores de soporte y RNAs ofrecen resultados superiores. Por otro lado los modelos lineales como la regresión lineal múltiple y el análisis de componentes principales, dan lugar a modelos que explícitamente relacionan los descriptores de entrada con la predicción de resultado de la actividad, con parámetros que son relativamente fácil de interpretar. Sin embargo, mientras más intuitivamente satisfactoria, esta claridad se produce a costo de un menor rendimiento (Weaver, 2004; Walters y Goldman, 2005).

Bajo el proceso QSAR y la selección de descriptores, se han presentado diferentes enfoques de diseño de péptidos basados en sistemas computacionales como los que a continuación se describen.

2.6. Trabajos Relacionados

El aspecto más importante del diseño de péptidos antimicrobiales asistidos por computadora, es la estimación precisa o predicción de la actividad biológica deseada a partir únicamente de la secuencia primaria (Fjell *et al.*, 2011a). En la década de los 80's, los modelos computacionales para péptidos basados en QSAR se utilizan como guía para la predicción de la actividad y de la optimización de secuencias para diferentes actividades biológicas (Tabla 1). En la década de los 90's, los métodos de aprendizaje máquina, específicamente las redes

neuronales artificiales, sustituyeron a las tradicionales funciones de regresión de los modelos QSAR para péptidos (Schneider y Wrede, 1998).

En la literatura encontramos dos corrientes principales para el diseño de péptidos asistido por computadora. En el primero de ellos se combinan un estimador de actividad con una técnica que permite la optimización estocástica (es decir, con la participación de procesos iterativos aleatorios). En el segundo enfoque se miden las posibles características que definen a los péptidos antimicrobiales y se hace un análisis sobre todas las secuencias de determinada longitud para encontrar aquellas que posean las características definidas.

El concepto general en ambos enfoques es utilizar los resultados que se obtienen mediante las pruebas de actividad o predicción para influir en la decisión de cuáles péptidos deberán ser diseñados, generados y probados, con el objetivo de mantener esfuerzos experimentales al mínimo.

Tabla 1: Selección de conceptos y enfoques para el diseño de péptidos basados en computadora (Fjell *et al.*, 2011a).

Año	Función de aptitud	Estrategia de diseño	Descripción
1994	Red neuronal artificial (Schneider *et al.*, 1998)	Algoritmo evolutivo	Los péptidos se generan *de novo* al simular el proceso de evolución molecular para maximizar el valor de puntuación de la función de la red neuronal.
2004	Estabilidad de plegado basado en probabilidad de energías (Klepeis *et al.*, 2004)	Reemplazo de aminoácido	El problema del plegado inverso se resuelve para identificar nuevas secuencias de péptidos que son compatibles con las plantillas de la columna vertebral de péptidos de referencia.
2005	Simulación de acoplamiento (Belda *et al.*, 2006)	Algoritmo evolutivo, heurística	Se calcula un puntaje de acomplamiento como una estimación de la interacción de energía de un péptido receptor, el cual sirve como una función de aptitud para un algoritmo evolutivo.
2006	Gramática Regular (Loose *et al.*, 2006)	Modelo lingüístico	Se generan nuevos péptidos antimicrobiales (AMPs) en base a reglas gramáticales derivadas de AMPs que ocurren de forma natural.
2007	Simulación de acoplamiento (Yagi *et al.*, 2007)	Algoritmo genético	Los péptidos se acoplan y después se alteran por mutación y cruzamiento.
2007	Alineamiento de matriz experimentalmente optimizada (Oren *et al.*, 2007)	Generación de secuencia aleatoria	Una matriz de puntuación se altera sistemáticamente en base a sustituciones que aprueban la diferenciación entre enlaces fuertes y débiles.
2007	Red neuronal artificial (Hiss *et al.*, 2007)	Algoritmo de optimización de colonia de hormigas	El diseño automatizado de péptidos se lleva a cabo de novo utilizando matrices de adaptación de peso posicionales.
2008	Modelo oculto de Markov (HMM) (Fjell *et al.*, 2008)	Generación de secuencia aleatoria	Se utiliza un modelo HMM para identificar y clasificar AMPs basado en proteínas naturales de la base de datos UniProt.
2008	Simulación de dinámica molecular (Gronwald *et al.*, 2008)	Algoritmo evolutivo	El algoritmo busca partes estables en un péptido mediante la generación de puntos mutantes, que se evalúan en simulaciones de dinámica molecular.
2009	Red neuronal artificial Fjell *et al.* (2009)	Generación de secuencia aleatoria	El algoritmo puntúa un gran conjunto de secuencias aleatorias de acuerdo a un modelo de red neuronal que ha sido entrenado por AMPs.
2011	Red neuronal artificial Fjell *et al.* (2011b)	Algoritmo genético	El diseño de péptidos se lleva a cabo de novo utilizando un algoritmo genético.

Con base en la primer corriente, los algoritmos evolutivos, en donde el proceso de evolución se lleva a cabo *in silico* a través de generaciones sucesivas de mutaciones, supresiones e intercambio de secuencias, se han utilizado en la búsqueda de péptidos con actividad mejorada. Los enfoques estocásticos son adecuados para la optimización del péptido cuando el espacio de búsqueda es vasto, sobre todo para secuencias largas donde los métodos deterministas son prohibitivos (Fjell *et al.*, 2011a).

Los trabajos más recientes (2009 y 2011) de la Tabla 1 fueron desarrollados por Fjell y Hancock, la motivación principal de estos es generar péptidos más potentes, rentables, con un amplio espectro como posibles antibióticos de nueva generación para sustituir a los antibióticos convencionales. Estos últimos son cada vez menos eficaces contra las bacterias pues estas se han hecho resistentes a diferentes fármacos, además los antimicrobianos naturales, conocidos como péptidos de defensa del huésped o péptidos antimicrobianos, defienden a los organismos huésped contra microbios pero la mayoría muestra una modesta actividad antibiótica directa.

Fjell *et al.* (2009) crearon un sistema de software utilizando redes neuronales artificiales, estas fueron entrenadas midiendo la actividad de 1400 péptidos de alto rendimiento. El modelo fue construido utilizando 44 descriptores QSAR con el que subsecuentemente se creó una librería de aproximadamente 100,000 péptidos. La validación *in vitro* del modelo mostró 94 % de precisión al identificar péptidos altamente activos. Los mejores péptidos identificados en la proyección probaron tener una actividad comparable o superior a cuatro antibióticos convencionales y superior al péptido más avanzado en desarrollo clínico que actúa frente a una amplia gama de patógenos humanos multirresistentes.

En 2011 Fjell *et al.* (2011b) presentan un método para generar secuencias de péptidos candidatas utilizando algoritmos genéticos (GA), que proporciona una gran mejora con respecto al trabajo anterior (19 veces) en la identificación de nuevos péptidos antibacterianos. Aproximadamente el 0.50 % de los péptidos evaluados durante el método de GA fueron clasificados como altamente activos, mientras que sólo el 0.026 % de las cerca de 100,000 secuencias que se proyectaron previamente fueron clasificados como altamente activas.

Por otro lado, el trabajo que da origen al enfoque determinista en el que se basa esta investigación comienza con Ellerby *et al.* (1999) a finales de los noventa. Al inicio de su investigación ya se conocían más de 100 péptidos antibacteriales naturales, y el diseño *de novo* de esta clase de péptidos empezaba a recibir mucha atención por parte de la comunidad científica (Bessalle *et al.*, 1990; Blondelle y Houghten, 1992). Muchos de estos péptidos presentaban las siguientes características: lineales, catiónicos y con una estructura α-hélice. Además, algunos de estos presentaban anfipaticidad, con residuos hidrofóbicos por un lado del eje helicoidal y residuos catiónicos por el otro.

En el trabajo que presentan en 1999 se diseñan péptidos con las características mencionadas los cuales tienen dos funciones principales. Una función es buscar un motivo en los vasos sanguíneos de los tumores y la otra es inducir la muerte celular programada al introducir una secuencia particular de aminoácidos en la célula. Esta función pro-apoptótica fue diseñada para no ser tóxica fuera de las células, pero tóxica una vez dentro de la célula objetivo y con esto deshacer la membrana mitocondrial. Los prototipos diseñados computacionalmente y después sintetizados contienen entre 21 y 26 residuos. Estos diseños se conforman de tres secciones: la primera de ellas es la sección selectiva, esta contiene un péptido con propiedades “selector de tumor” los cuales mostraban evidencia de introducirse en las células, la sección del péptido que provoca la apoptosis, y finalmente un pequeño segmento que une a estos dos. Los péptidos presentaron toxicidad contra las células endoteliales angiogénicas que suministran de sangre a los tumores y mostraron actividad anticancerígena en ratones. Un modelo generado computacionalmente de uno de los prototipos se muestra en la Figura 5.

Con la finalidad de mejorar el enfoque de seleccionar y eliminar a las células endoteliales angiogénicas que suministran de sangre a las células cancerígenas, del Rio *et al.* (2001) desarrollaron un método computacional para detectar péptidos potencialmente apoptóticos (APAP por sus siglas en inglés). Este enfoque fue diseñado para superar los problemas de toxicidad y síntesis asociado con el enfoque quimioterapéutico de 1999.

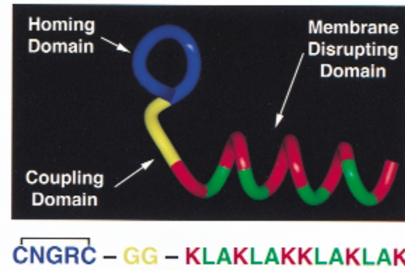

Figura 5: Modelo generado por computadora. El péptido está compuesto por un dominio 'de caza' (azul) y un dominio que deshace la membrana (pro-apoptótico) (residuos hidrofílicos en rojo e hidrofóbicos en verde), unidos por un dominio de enganche (amarillo).

El objetivo es detectar APAPs cargados positivamente que no sean tóxicos afuera de la célula, capaces de seleccionar la vasculatura de un tumor por medio de un péptido que reconoce a un receptor en la superficie de la célula, este se interna y deshace la mitocondria negativamente cargada ejerciendo su efecto apoptótico, causando una regresión en el tumor (Oren y Shai, 1998).

Basados en la relación postulada entre los mecanismos de la apoptosis dependiente de la mitocondria y la actividad antibacterial, buscaron APAPs que se asemejarán a los antibacterianos. Notaron que los péptidos antibióticos conocidos se ajustaban a un patrón: poca probabilidad de formar una hélice en solución acuosa, alta probabilidad de formar una hélice en presencia de membranas cargadas negativamente, y un valor de punto isoeléctrico alto. Hipotetizaron que estas características (descriptores) eran importantes al determinar la selectividad observada en estos péptidos hacia las membranas de las bacterias y membranas parecidas a las bacterias (como la de las mitocondrias).

Un subconjunto de 30 péptidos antibacteriales previamente reportados en la literatura se utilizaron para los cálculos de momento hidrofóbico, punto isoeléctrico y la probabilidad helicoidal de péptidos monoméricos en solución acuosa (llamado puntaje AGADIR) (Munoz y Serrano, 1997). Los valores que agrupaban a los péptidos con mayor índice terapéutico (TI) se consideraron como valores insignia de los PAPs.

El TI se define como la relación entre la concentración inhibidora observada en las células mamíferas y la concentración inhibidora observada en células bacterianas (del Rio *et al.*,

2001). Entre mayor sea el valor de la relación, el péptido será más específico hacia las membranas procariotas cargadas negativamente.

Las secuencias peptídicas con valores entre 0.4 y 0.6 para el momento hidrofóbico, punto isoélectrico entre 10.8 y 11.7 y puntaje AGADIR menor a 10 resultaron tener el mayor índice terapéutico. Estos valores insignia son los que se utilizaron como base para el enfoque computacional.

Los valores insignia se usaron para realizar una búsqueda en la base de datos SwissProt, versión 38, la cual contenía 80,000 secuencias de proteínas. Para el análisis, solo se tomaron en cuenta secuencias de menos de 40 aminoácidos, tampoco se tomaron en cuenta fragmentos de proteínas. Con este filtro un total de 2473 secuencias fueron analizadas y 14 de ellas fueron catalogadas como PAPs.

APAP provee entonces una herramienta para identificar PAPs independientemente de la similitud con cualquier otro péptido antibacterial o pro-apoptótico conocido. Además, APAP permite hacer búsquedas en bases de datos en forma sistemática.

En el trabajo de Polanco *et al.* (2011) se toman como base los parámetros de las propiedades fisicoquímcas descritas en el trabajo de del Rio *et al.* (2001) y plantean hacer una búsqueda exhaustiva de estas utilizando un arreglo de compuertas reprogramables (FPGAs por sus siglas en inglés) con la intención de encontrar Péptidos Catiónicos Selectivos Antibacteriales (SCPAs por sus siglas en inglés). El objetivo es ofrecer una solución rentable para el problema combinatorio el cual necesita cómputo de alto rendimiento.

Los parámetros que se calculan son: momento hidrofóbico y punto isoeléctrico como en (del Rio *et al.*, 2001), para la predicción del plegado se utilizó la media de carga neta y la media de hidrofobicidad en sustitución de AGADIR ya que este no discriminaba eficientemente en péptidos cortos pues más del 99.9 % de los péptidos de longitud 9 analizados de manera aleatoria cumplían con el requerimiento de este parámetro (AGADIR $<$ 10), lo que lo hacía irrelevante.

Para el cálculo de cada una de las propiedades se desarrollaron cuatro módulos separados dentro del FPGA, es decir, las operaciones para la predicción de SCAPs se hacen de manera paralela. El reporte en tiempo de la búsqueda exhaustiva se hace sobre la secuencia de nueve aminoácidos RAAAYXXXX, donde X representa a cualquiera de los aminoácidos esenciales y R, A, Y representan Arginina, Alanina y Tirosina, respectivamente. Aunque el artículo refiere que se analizan secuencias con hasta 9 aminoácidos, solo 4 aminoácidos fueron variables, es decir 20^4 secuencias diferentes.

Se reporta el tiempo promedio de análisis por propiedad y por secuencia. Para este cálculo se divide el tiempo total obtenido entre las 20^4 secuencias analizadas. El tiempo de procesamiento utilizando FPGAs se compara con la implementación secuencial en el lenguaje FORTRAN 77 implementado en CPU.

En 2012, Garcia-Ordaz *et al.* (2012) realizan el mismo estudio de cálculo exhaustivo de las propiedades fisicoquímicas para detectar SCAPs, pero esta vez utilizando GPUs. Este trabajo fue de nuestro conocimiento ya en una etapa avanzada de este proyecto de tesis. Los resultados obtenidos los comparan con los obtenidos en el trabajo de FPGAs del 2011.

El artículo refiere que gracias a la configuración en el GPU de hilos y bloques empleada, es posible analizar péptidos de hasta 9 aminoácidos, esto lo hacen analizando bloques de 20^4 secuencias por cada llamada al GPU, es decir, para lograr las 20^9 combinaciones posibles, el CPU debe hacer 20^5 llamados al GPU (kernel).

El trabajo incluye una tabla comparativa entre los tiempos reportados por los diferentes enfoques empleados analizando a lo más 20^4 combinaciones, donde el tiempo de ejecución de los GPUs es significativamente mejor al reportado con la implementación en FPGAs. El artículo no reporta el tiempo necesario para evaluar 20^9 péptidos aunque es posible estimarlo al multiplicar lo que tarda una sola llamada al GPU (con 20^4 secuencias) y multiplicarla por las 20^5 subsecuentes llamadas para lograr las 20^9 combinaciones.

Los trabajos de Polanco *et al.* (2011) y de Garcia-Ordaz *et al.* (2012) dejan espacio para la optimización, principalmente en el filtrado de los resultados negativos y en el caso de Garcia-Ordaz *et al.* (2012) además en el número de llamadas al GPU necesarias para calcular péptidos de longitud nueve. Es del trabajo de Polanco *et al.* (2011) de donde se obtienen las propiedades, media de carga neta, media de hidrofobicidad, punto isoeléctrico y momento hidrofóbico helicoidal que tomamos como base en nuestra investigación para la definición de un SCAP. La descripción de cada propiedad se hace en la siguiente sección.

2.7. Propiedades fisico-químicas seleccionadas en Polanco *et al.* (2011)

La predicción de SCAPs se realiza considerando tres características de las secuencias peptídicas: la propensión a no tener estructura de forma nativa, la anfipaticidad y la carga. Estas características se predicen con el cálculo de las siguientes cuatro propiedades físico-químicas.

- **Media de hidrofobicidad (MH)**. Es el valor normalizado de la media de hidrofobicidad sobre todos los aminoácidos en un péptido dado. A continuación se presenta la fórmula:

$$MH = \frac{1}{n-l}\sum_{i=1}^{n-l} H_i, \tag{1}$$

$$H_i = \frac{1}{l}\sum_{j=i}^{i+l} h_{aa}^{(j)}, \tag{2}$$

  donde H_i es la media de hidrofobicidad del segmento de longitud l de los residuos, $aa^{(j)}$ ... $aa^{(j+l)}$ y $h_{aa}^{(j)}$ es la hidrofobicidad del aminoácido aa en la posición j de la secuencia del péptido de longitud n. La hidrofobicidad de cada aminoácido está dada por la escala presentada por Eisenberg *et al.* (1984). Para que un péptido sea considerado SCAP su MH tiene que estar dentro del intervalo de 0.35 a 0.55 (Polanco *et al.*, 2011).

- **Media de carga neta (MC).** Es determinada por la siguiente expresión:

$$MC = \frac{1}{n}(\sum_i N_i - \sum_j N_j), \tag{3}$$

donde N_i es el número total de aminoácidos del tipo arginina (Arg) y lisina (Lys), que hay en la secuencia del péptido. N_j es el número de aminoácidos del tipo ácido aspártico (Asp) y ácido glutámico (Glu) que hay en la secuencia, el valor de n es la longitud de la secuencia. Para que un péptido sea considerado SCAP se debe cumplir la desigualdad $MC > -5.44MH + 2.66$ (Polanco *et al.*, 2011).

- **Punto isoeléctrico (PI).** Representa el pH al cual la carga neta (Z) del péptido es cero. El pH define la acidez y basicidad relativas de una sustancia acuosa, la escala del pH va de 0 hasta 14, siendo el pH de 7 el punto neutro, soluciones con valores inferiores a 7 son llamadas ácidas y mayores a 7 se denominan básicas o alcalinas.

Para calcular el punto isoeléctrico de la secuencia es necesario conocer primero el punto isoeléctrico de cada aminoácido, el cual se calcula con las constantes de disociación ácida pK_a. Estos valores indican la fuerza que tienen las moléculas para disociarse (2). Cada aminoácido tiene dos pK_a: pK_a1 para el ácido carboxílico y pK_a2 para el amino. El punto isoeléctrico es el promedio de estos dos: $pI = 1/2(pK_a1 + pK_a2)$.

Existen aminoácidos donde la cadena lateral 'R' introduce otros grupos ionizables, estos se describen con una tercera constante de disociación pK_a3. En este caso, para las cadenas laterales ácidas el pH será menor pues se agrega una carga negativa "extra", para estos casos la fórmula es: $pI = 1/2(pK_a1 + pK_a3)$. Para las cadenas laterales básicas el pH es mayor por la inclusión de una carga positiva "extra". La fórmula para este caso es $pI = 1/2(pK_a2 + pK_a3)$.

En el caso de los péptidos, solo algunos de los aminoácidos que componen la secuencia contienen cadenas laterales ionizables con valores de pK_a. Los aminoácidos esenciales que presentan estas constantes en sus cadenas laterales son: ASP, GLU, CYS, TYR, HIS, LYS y ARG. Los grupos amino y carboxilo deben ser tomados en cuenta en el cálculo ya que presentan valores de pK_a. Con esto se calcula la contribución de carga.

Tabla 2: Valor de pK_a y pI de los aminoácidos.

Aminoácido	$pK_a\{1\}$ -COOH	$pK_a\{2\}$ -NH3+	$pK_a\{3\}$ grupo R	**pI**
Alanina	2.34	9.69		6.00
Arginina	2.17	9.04	12.48	10.76
Asparagina	2.02	8.80		5.41
ácido aspártico	1.88	9.60	3.65	2.77
Cisteina	1.96	10.128	8.18	5.07
Fenilalanina	1.83	9.13		5.48
Glicina	2.34	9.60		5.97
ácido glutámico	2.19	9.67	4.25	3.22
Glutamina	2.17	9.13		5.65
Histidina	1.82	9.17	6.00	7.59
Isoleucina	2.36	9.60		6.02
Leucina	2.36	9.60		5.98
Lisina	2.18	8.95	10.53	9.74
Metionina	2.28	9.21		5.74
Prolina	1.99	10.60		6.30
Serina	2.21	9.15		5.58
Treonina	2.09	9.10		5.60
Tirosina	2.20	9.11	10.07	5.66
Triptófano	2.83	9.39		5.89
Valina	2.32	9.62		5.96

La fórmula usada para el cálculo del punto isoeléctrico del péptido completo es la siguiente:

$$Z = \sum_i N_i \frac{1}{1 + 10^{(pH - pK_{a_i})}} - \sum_j N_j \frac{1}{1 + 10^{(pK_{a_j} - pH)}}, \quad (4)$$

donde N_i es el número de argininas (Arg), lisinas (Lys) e histidinas (His) que hay en la secuencia del péptido, pK_{a_i} es el valor de pK_a de los residuos de argininas (Arg), lisinas (Lys) e histidinas (His). N_j es el número de ácido aspártico (Asp), ácido glutámico (Glu), cisteina (Cys) y tirosina (Tyr) y pK_{a_j} es el valor de pK_a de ácido aspártico (Asp), ácido glutámico (Glu), cisteina (Cys) y tirosina (Tyr). Un péptido es considerado SCAP si presenta un IP con valor en el intervalo de 10.8 a 11.8 (Polanco *et al.*, 2011).

- **Momento hidrófobico helicoidal (MHH).** El momento hidrofóbico mide la anfipaticidad de N aminoácidos en un segmento de un péptido. La periodicidad en la forma polar-apolar de la secuencia de aminoácidos de una proteína o péptido, puede

ser examinada al asignarle un valor numérico de hidrofobicidad a cada residuo (Tabla 3) y con esto buscar la periodicidad en la función unidimensional restante. La fuerza de cada componente periódico es la cantidad que se ha denominado momento hidrofóbico (Eisenberg *et al.*, 1984). Si la estructura tridimensional de la proteína es conocida, entonces el momento hidrofóbico se puede calcular a partir de la relación:

$$\mu_s = \sum_{i=1}^{N} H_n s_n \tag{5}$$

donde H_n es el valor de hidrofobicidad del n-ésimo residuo y, s_n es un vector unitario que va en dirección del núcleo del carbono alfa hacia el centro geométrico de la cadena lateral. Los segmentos de los péptidos con cadenas no polares de un solo lado de la estructura se caracterizan por un valor alto de μ_s (Eisenberg *et al.*, 1982).

El momento hidrofóbico de un segmento de una proteína se puede estimar cuando se conoce la estructura primaria, siempre que el segmento sea periódico y que el período sea conocido. Sea la estructura periódica especificada por m, es decir, el número de residuos por giro en la estructura, o alternativamente, $\delta = 2\pi/m$, donde δ es el ángulo en radianes en el que las cadenas laterales sucesivas emergen de la estructura principal, esto es cuando el segmento periódico es visto desde arriba del eje. Por lo tanto, para una α hélice, δ es 100° ($m = 3.6$), y para una estructura β lámina, se espera una δ en un intervalo entre 160° ($m = 2.3$) hasta 180° ($m = 2.0$) (Eisenberg *et al.*, 1984).

Cualquiera que sea el valor de δ, una estructura periódica que sea anfifílica arrojará un valor alto de μ, dado por la ecuación 6:

$$\mu(d) = \left\{ \left[\sum_{n=1}^{N} H_n sin(\delta n) \right]^2 + \left[\sum_{n=1}^{N} H_n cos(\delta n) \right]^2 \right\}^{1/2}, \tag{6}$$

donde δ se mide en radianes, por lo que la longitud, μ, esta dada por la suma de los componentes de los vectores de hidrofobicidad.

La forma de la ecuación 6 sugiere que podemos considerar al momento hidrofóbico de manera general; podemos pensar en δ como una variable que puede tomar cualquier

valor entre cero (para una hélice con repetición infinita) hasta 180° (para una cadena recta de estructura β lámina). Con esto, el momento hidrofóbico es el módulo de la transformada de Fourier de la función de la hidrofobicidad de una sola dimensión.

Con esto, $\mu(\delta)$ es la fuerza del componente de la periodicidad que tiene la frecuencia δ. Esta cantidad puede ser evaluada para cualquier secuencia de aminoácidos mediante la inserción de la hidrofobicidad de cada elemento. Un valor alto de μ indica un fuerte componente de la periodicidad en el δ dado (Eisenberg *et al.*, 1984).

Tabla 3: Valor de hidrofobicidad de los aminoácidos de acuerdo a Eisenberg *et al.* (1984), ordenados de forma descendente.

Aminoácido	**Hidrofobicidad de Eisenberg**
Isoleucina	0.73
Fenilalanina	0.61
Valina	0.54
Leucina	0.53
Triptófano	0.37
Metionina	0.26
Alanina	0.25
Glicina	0.16
Cisteina	0.04
Tirosina	0.02
Prolina	-0.07
Treonina	-0.18
Serina	-0.26
Histidina	-0.40
ácido glutámico	-0.62
Asparagina	-0.64
Glutamina	-0.69
ácido aspártico	-0.72
Lisina	-1.1
Arginina	-1.8

Un péptido es considerado SCAP si tiene un valor de μ dentro del intervalo 0.4 y 0.6 según el trabajo de del Rio *et al.* (2001); Polanco *et al.* (2011).

Una vez definidas las propiedades para la detección de SCAPs, estas se usaron para el desarrollo de los algoritmos que se presentan en el Capítulo 4.

Capítulo 3. Fundamentos de computación

En este capítulo se describe la arquitectura de los GPU de NVIDIA, así como el modelo de programación de CUDA. Además de explicar los conceptos generales del marco de trabajo, se incluye un conjunto de instrucciones específicas de CUDA que se utilizan en el desarrollo de la solución para detectar SCAPs.

3.1. Introducción

La idea general de la tarjeta de acelerador de gráficos es liberar al CPU de tareas de procesamiento masivo que están directamente relacionadas con la texturizado de imágenes y el procesamiento de video.

Por esta razón, este hardware normalmente tiene sus propios registros de memoria, separados de los registros de memoria del CPU principal. Gracias a esto, los dispositivos aceleradores implementan procesos independientes con la capacidad de manejar grandes cantidades de datos numéricos (Kirk y Hwu, 2010). Al liberar al CPU de la carga del procesamiento masivo de video, el uso del acelerador gráfico mejora el desempeño general de las aplicaciones gráficas. Al mismo tiempo, el hecho de tener hardware especializado y dedicado, da como resultado una mejor calidad de imagen (Patel, 2010).

El poder de procesamiento de estos aceleradores pronto fue utilizado para mejorar el desempeño de diferentes tipos de aplicaciones y no solo para aplicaciones gráficas (videojuegos principalmente). Sin embargo, para acceder a los recursos de procesamiento, era obligatorio codificar las instrucciones utilizando complejas declaraciones de gráficos y estructuras de datos, ya que el hardware había sido diseñado para cumplir con las restricciones de los videojuegos (Mor, 2013).

La compañía NVIDIA lanza su primer GPU (Unidad de Procesamiento Gráfico) en 1999. La tarjeta (NVIDIA GeForce 256) contenía cerca de 22 millones de transistores y era capaz de procesar 10 millones de polígonos por segundo (NVIDIA, 2010). Por primera vez un acelerador gráfico tenía tal poder de procesamiento, y gracias a su arquitectura, la familia de

GPUs GeForce se volvió un patrón a seguir para las siguientes generaciones de tarjetas de video.

Sin embargo, codificar para los GPUs seguía siendo un reto porque era obligatorio mantener la codificación mediante el uso de primitivas gráficas complejas como instrucciones (Kirk y Hwu, 2010). En 2006, después de varias generaciones de GPUs, NVIDIA lanza la primera versión de CUDA (Compute Unified Device Architecture). Por primera vez fue posible acceder a los recursos del GPU utilizando instrucciones sencillas, muy similares a las utilizadas en los lenguajes de programación de tercera generación. De hecho, la primera versión de CUDA era simplemente una extensión del lenguaje C. Pero su aparición introdujo el concepto de GPGPU (general-purpose computing on graphics processing units).

3.2. La arquitectura de NVIDIA

La arquitectura de los GPUs, diseñada específicamente para el procesamiento masivo de datos en paralelo, permite un mejor rendimiento sobre el desempeño de los CPUs multinucleos cuando el problema a resolver se basa fuertemente en la paralelización de datos (Mor, 2013). De hecho, el problema original por el que los aceleradores gráficos fueron diseñados (el procesamiento de imágenes y la renderización de video) se basa en la transformación de una gran cantidad de datos numéricos independientes, esta característica hace que los problemas basados en este tipo de organización de datos sean candidatos ideales para ser procesados por el GPU (Kirk y Hwu, 2010). La Figura 6 muestra la comparación de la reciente evolución del poder computacional de los CPUs y GPUs comerciales (NVIDIA, 2008).

Es importante comprender las razones que hacen que los GPUs muestren un rendimiento excepcional cuando se compara con el CPU convencional. En primer lugar, el diseño de los GPUs se enfoca en el procesamiento de datos masivos en paralelo también llamado *stream processing* (Jost *et al.*, 2010). Esto es posible ya que un área más amplia en el chip de silicio se reserva para unidades de procesamiento donde se asigna una mayor cantidad de transistores, mientras que la memoria “caché” y las estructuras de control se limitan al mínimo posible (NVIDIA, 2008), como se muestra en la Figura 7.

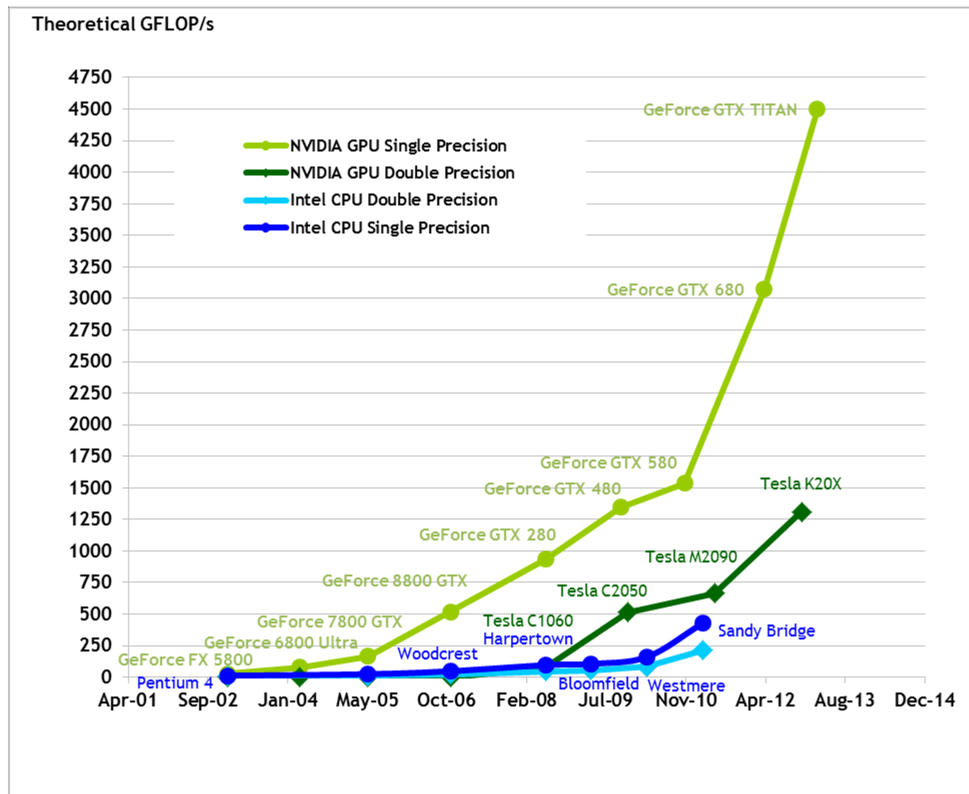

Figura 6: Comparación de rendimiento entre los CPUs y GPUs comerciales (NVIDIA, 2008).

Figura 7: Asignación de transistores en la arquitectura de CPUs y GPUs (NVIDIA, 2008).

La segunda razón se basa en el problema a resolver. Si el problema es fácilmente paralelizable, entonces puede ser un buen candidato para ser procesado en el GPU. Por otro lado, si el problema es difícil de paralelizar, una implementación híbrida (CPU y la GPU) puede ser una buena opción. De hecho, los CPUs convencionales utilizan paralelismo de tareas, y no paralelismo de datos. Múltiples tareas se asignan a múltiples hilos. Cada hilo ejecuta instrucciones diferentes, cada uno de ellos requiere ser administrado y calendarizado de manera explícita (Sanders y Kandrot, 2010). Esto significa que cada hilo necesita ser controlado por software y tiene que ser programado individualmente. Por otro lado, un GPU realiza

paralelismo de datos al tener una estructura SIMD (Single Instruction Multiple Data)(Kirk y Hwu, 2010). Esto significa que un GPU es capaz de ejecutar la misma instrucción sobre datos diferentes. Además, todos los hilos son administrados y calendarizados directamente por el hardware. El modelo de programación del GPU se basa en la programación de grupos de hilos independientes.

Los GPUs de CUDA se organizan en *Clusters de Procesamiento Gráfico* (GPC por sus siglas en inglés), estos a su vez agrupan varios *Procesadores de flujo* (Streaming Processors o SMX) que son la unidad básica de ejecución (Dematté y Prandi, 2010) (figuras 8 y 9). CUDA ejecuta el mismo programa en todos los GPCs: el código del programa (*kernel*) es el mismo para todos los hilos pero los datos y el flujo de ejecución pueden ser diferentes y diverger. CUDA lanza varias instancias del mismo *kernel* a través de los hilos. Los hilos se agrupan en *warps* (Tabla 4) para para ser ejecutados dentro de un GPC. Los hilos son instancias en tiempo de ejecución del mismo *kernel* que ejecuta el mismo código, además, todos los hilos de un *warp* son ejecutados por un GPC de manera SIMD, por lo tanto, deben ejecutar exactamente la misma instrucción, al mismo tiempo, aunque con diferentes datos (Mor, 2013). Si los hilos divergen (por ejemplo, al tomar diferentes caminos en una sentencia *if*), se dividirán en *warps* diferentes, y posiblemente no se aprovechen las capacidades de los GPCs. Estas restricciones ayudan a mantener una arquitectura simple pero poderosa.

Tabla 4: Terminología de CUDA.

Dispositivo/Huésped	GPU/CPU
Kernel	Función que se llama desde el huésped y se ejecuta en el dispositivo. Los Kernels se ejecutan de forma simultánea por n hilos.
Hilo	Flujo de instrucciones que se asigna a una unidad de ejecución.
Warp	Conjunto de hilos (32 actualmente). El warp es la unidad básica de calendarización (un warp se calendariza en un multiprocesador).
Bloque	Conjunto de hilos que pueden cooperar por medio de la memoria compartida y que son sincronizables entre si.
Grid	Estructura que agrupa a los bloques de hilos para su ejecución.

Por último, los GPUs son "tolerantes a la latencia" (A.Rage, 2011), mientras que en los CPUs convencionales se hace un gran esfuerzo con el fin de ocultar o minimizar la latencia, de hecho, es necesario el uso de memorias caché para minimizar este problema, además un CPU normalmente realiza instrucciones de captura previa (pre-fetch), control de flujo y ejecuciones

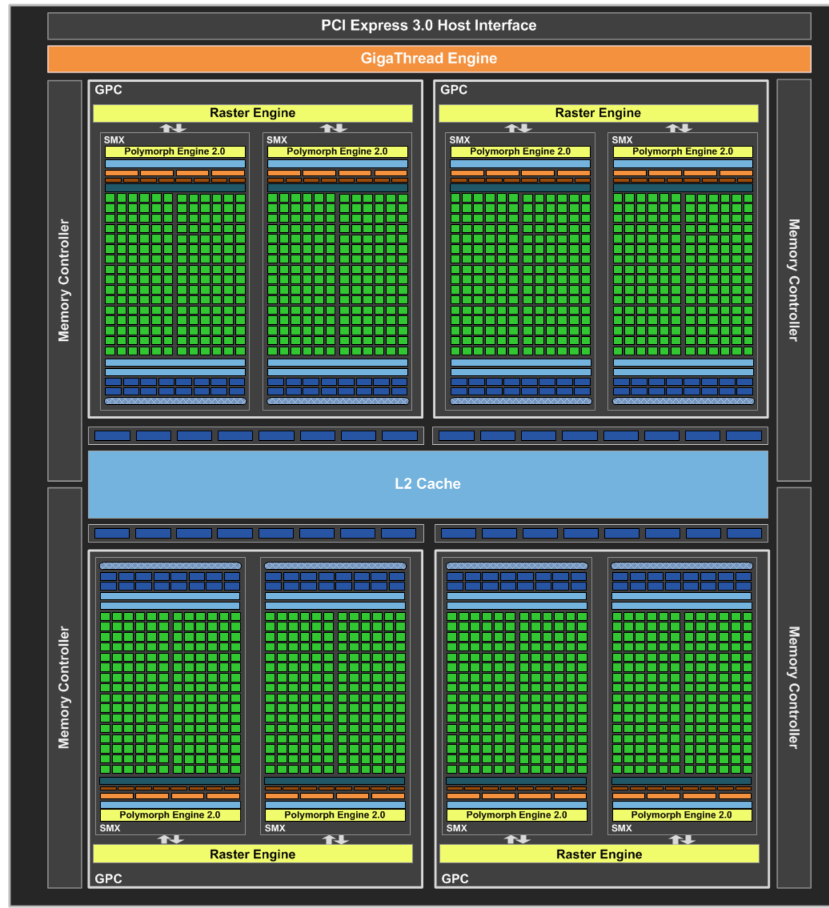

Figura 8: Estructura y recursos computacionales de la tarjeta NVIDIA GeForce GTX 680. La tarjeta tiene 4 GPC, dende cada uno contiene 2 SMX (NVIDIA, 2012).

fuera de orden. Por otro lado, los GPUs abandonaron el paradigma de tubería (pipeline), evitando problemas de sincronización de hilos, haciéndolos totalmente independiente entre si.

3.3. Plan de trabajo para los GPUs de NVIDIA

Desde el lanzamiento de su primer GPU GeForce, la compañia NVIDIA ha establecido una plan de trabajo que cubre la evolución de sus productos (Chiappetta, 2013). Cada paso en este plan de trabajo cubre (o cubrirá) un conjunto específico de capacidades, como se muestra en la Figura 10.

Las familias GeForce y Tesla formaron parte de la primera generación de GPUs que fueron cubiertas por el marco de programación CUDA. A pesar de que las dos familias pertenecen a la misma generación de GPUs, GeForce fue diseñada para el mercado de juegos y de uso

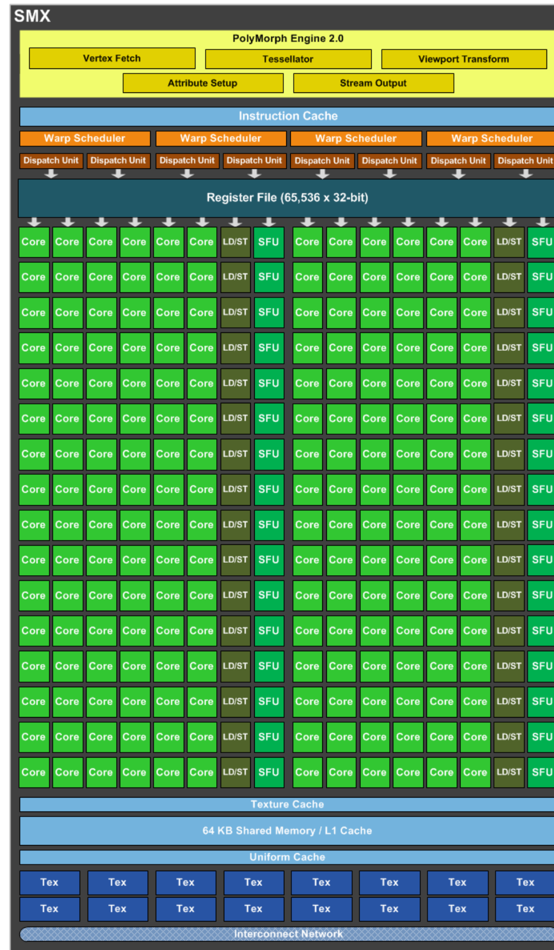

Figura 9: Multiprocesador de flujo (SMX) de la tarjeta GeForce GTX 680. Contiene su propia unidad de instrucciones y 192 núcleos (NVIDIA, 2012).

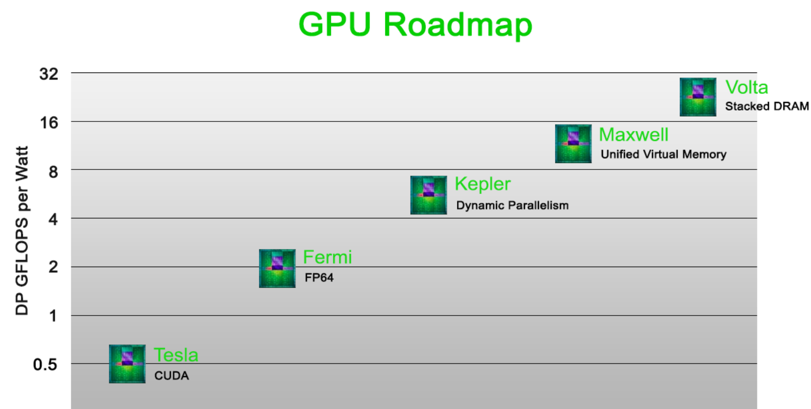

Figura 10: Hoja de ruta de los GPUs de NVIDIA. En la gráfica se observa un incremento en el número de operaciones flotantes de doble precisión por segundo (DP GFLOPS) por Watt en cada generación, lo que traduce en ahorro de energía (Chiappetta, 2013).

personal, mientras que la familia Tesla fue diseñada para el mercado de Cómputo de Alto Desempeño (HPC por su siglas en inglés).

La siguiente generación, la arquitectura Fermi, se caracterizó principalmente por el soporte de operaciones de punto flotante con doble precisión, además del aumento en la cantidad de memoria compartida disponible por cada bloque de hilos.

La arquitectura Kepler incluye paralelismo dinámico, lo que permite un aumento considerable en la autonomía de los códigos del *kernel* de CUDA (que se ejecutan en los dispositivos), en relación con la dependencia que tenían anteriormente con el controlador de tareas del CPU.

La arquitectura Maxwell (que aún no sale al mercado) promete un espacio de memoria unificado entre GPU y CPU, lo que deberá mejorar el rendimiento de las aplicaciones que necesitan comunicarse con los procesos que se ejecutan en la CPU. Esta función facilitará la implementación de códigos híbridos (que utiliza colaboración del CPU y GPU) que son comunes hoy en día en las aplicaciones de *clusters*.

Finalmente, la arquitectura Volta saldrá con mejoras en el diseño de hardware, principalmente en el uso de la memoria DRAM apilada. Esto debe representar mejoras en el consumo de energía, pero no necesariamente en la reducción de los tiempos de ejecución.

3.4. Capacidad de cómputo

La capacidad de cómputo (compute capability) describe las carácteristicas técnicas que contiene un GPU compatible con CUDA. Dispositivos que pertenecen a una misma familia pueden tener diferentes capacidades de cómputo. Esto ocurre porque se implementaron diferentes conjuntos de características durante el plan de trabajo de NVIDIA. Cada capacidad de cómputo se diferencia de las demás por implementar un conjunto de capacidades técnicas diferentes, como se muestra en la Figura 11.

La capacidad de cómputo de un dispositivo se define por un número de revisión dado co-

mo “X.Y”, donde ‘X’ hace referencia a la versión principal o revisión mayor y ‘Y’ a la revisión menor. El número de revisión mayor es 3 para los dispositivos basados en la arquitectura Kepler, 2 para los dispositivos basados en la arquitectura Fermi, y 1 para los dispositivos basados en la arquitectura Tesla. Los dispositivos con el mismo número de versión principal tienen la misma arquitectura en los núcleos (core). El número de revisión menor corresponde a una mejora incremental a la arquitectura del núcleo, posiblemente incluyendo nuevas características (NVIDIA, 2008).

<table>
<tr><th>Feature Support</th><th colspan="6">Compute Capability</th></tr>
<tr><th>(Unlisted features are supported for all compute capabilities)</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x, 3.0</th><th>3.5</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)</td><td rowspan="2">No</td><td colspan="5" rowspan="2">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)</td><td colspan="2" rowspan="4">No</td><td colspan="4" rowspan="4">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)</td></tr>
<tr><td>Warp vote functions (Warp Vote Functions)</td></tr>
<tr><td>Double-precision floating-point numbers</td><td colspan="3">No</td><td colspan="3">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)</td><td colspan="4" rowspan="7">No</td><td colspan="2" rowspan="7">Yes</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())</td></tr>
<tr><td>__ballot() (Warp Vote Functions)</td></tr>
<tr><td>__threadfence_system() (Memory Fence Functions)</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
__syncthreads_or() (Synchronization Functions)</td></tr>
<tr><td>Surface functions (Surface Functions)</td></tr>
<tr><td>3D grid of thread blocks</td></tr>
<tr><td>Funnel shift (see reference manual)</td><td colspan="5">No</td><td>Yes</td></tr>
</table>

Figura 11: Lista de características para cada capacidad de cómputo (NVIDIA, 2008).

Es posible conocer la capacidad de cómputo de un dispositivo mediante la ejecución de una instrucción específica de CUDA, lo que permite implementar código que se adapte

dinámicamente al GPU utilizado en tiempo de ejecución.

3.5. Modelo de programación

El modelo de programación CUDA trata al GPU como un dispositivo de cómputo que sirve como un coprocesador del CPU huésped. El GPU tiene su propia memoria en la tarjeta y es capaz de ejecutar muchos hilos en paralelo (Mor, 2013).

CUDA C extiende el lenguaje C al permitir al programador definir funciones llamadas *kernels*, que cuando son invocadas se ejecutan n veces en paralelo por n diferentes hilos de CUDA, a diferencia de las funciones regulares de C que solo se ejecutan una vez (NVIDIA, 2008).

El C de CUDA añade nuevas declaraciones y tipos de datos. Un ejemplo son los nuevos calificativos que se utilizan para invocar funciones (Mor, 2013):

- __ global__: la función se llama desde el código del huésped (CPU), pero se ejecuta en el GPU.

- __ device__: la función se llama desde las funciones del GPU y se ejecuta solo en el GPU.

- __ host__: la función se llama desde el huésped y corre solo en el CPU.

También se introdujo un nuevo conjunto de calificativos para las variables:

- __ device__: la variable se almacena en memoria global (estos datos no serán almacenados en el caché, lo que significa tener alta latencia).

- __ constant__: la variable se almacena en memoria global. El espacio de la memora constante se almacena en caché. Esta memoria es de solo lectura para los hilos y de escritura únicamente para los procesos del huésped (en el CPU).

- __ shared__: para almacenar en la memoria compartida. Es accesible para todos los hilos de un mismo bloque, pero invisible para hilos de otros bloques. Cada bloque tiene su propia cantidad de memoria compartida privada.

Las variables que no tienen un calificativo se almacenan en la memoria local de la siguiente manera: las variables de tipo escalar y vector se almacenan en los registros cuando es posible, de otra manera, serán puestas en la memoria global por el compilador. Los arreglos se almacenan en la memoria global de manera predeterminada (NVIDIA, 2011). En la sección 3.8 se describen a detalle los tipos de memoria contenidas en el GPU.

3.6. Kernel

La unidad de ejecución básica de un programa CUDA es el *kernel*. El *kernel* ejecuta un solo programa iniciando n hilos diferentes de manera concurrente. De acuerdo con el modelo de programación CUDA, existen miles de hilos para ser utilizados de forma simultánea. El *kernel* se puede iniciar de modo síncrono o asíncrono mediante el proceso huésped en el CPU. Dependiendo de la arquitectura, el *kernel* se puede ejecutar solo desde el huésped (capacidad de cómputo 3.0) o ser ejecutado desde una instancia previa del kernel (capacidad de cómputo 3.5 en adelante) (Mor, 2013).

El *kernel* se ejecuta por un arreglo de hilos. Cada hilo tiene su propio ID numérico, el cual se lee con una instrucción de CUDA. Existen tres tipos de organización disponibles para agrupar los hilos. Además del tipo “arreglo” que es la organización predeterminada, el bloque puede definirse como “matriz” (arreglo de hilos de dos dimensiones), o incluso como un “cubo” (arreglo de tres dimensiones) (Kirk y Hwu, 2010). Elegir uno u otro no afectará el tiempo de ejecución de la aplicación; esta elección sólo depende de las preferencias de los desarrolladores. Existen ciertos límites en la organización de tipo matriz o en las dimensiones del cubo, estas limitaciones son específicas para cada tarjeta y pueden ser consultadas directamente al dispositivo en tiempo de ejecución.

La palabra reservada __ global__ indica que la función que se está declarando en C es la función *kernel* de CUDA. La función se ejecutará en el dispositivo y sólo se puede llamar desde el huésped; en el *kernel* se configura el número de hilos y bloques a utilizar. La configuración se especifica mediante el uso de los símbolos <<< ... >>> (NVIDIA, 2011).

3.7. Hilos

La unidad fundamental para la ejecución en paralelo de CUDA es el hilo. Una función *kernel* de CUDA se ejecuta por un arreglo de hilos, este arreglo se crea al momento de invocar a la función *kernel* en tiempo de ejecución, todos los hilos ejecutan el mismo conjunto de instrucciones en C (Kirk y Hwu, 2010).

Debido al diseño de hardware, los hilos de CUDA son extremadamente ligeros, con muy poca sobrecarga al momento de su creación. El *kernel* lanza un grid de bloques de hilos, donde los hilos dentro de un bloque pueden compartir datos por medio de la memoria compartida y sincronizar su ejecución, mientras que hilos que se encuentren en bloques diferentes no se pueden comunicar directamente. La comunicación entre hilos de diferentes bloques se realiza por medio de llamados a la memoria global. Sin embargo, ese acceso es generalmente más costoso que el acceso directo a memoria compartida. De acuerdo con la documentación de NVIDIA, una operación de I/O en la memoria compartida toma 4 ciclos de reloj, mientras que la misma operación en la memoria global toma entre 300 y 400 ciclos de reloj (NVIDIA, 2011). Por esto, la falta de sincronización entre hilos de diferentes bloques puede convertiste fácilmente en una condición donde varios hilos compiten por acceder a un mismo recurso, conocida como condición de carrera.

Cada hilo tiene acceso a un identificador especial de CUDA lo cual permite al desarrollador identificar la ubicación específica de un hilo dentro de un bloque o dentro de un grid de bloques:

- threadIdx.x - Identificador de un hilo dentro de un bloque.
- blockIdx.x - Identificador de un bloque dentro de un grid.
- blockDim.x - Número de hilos por bloque

Como cada hilo en el arreglo ejecuta la misma función kernel, necesitan un identificador único para distinguirse unos de otros y para localizar la porción adecuada de los datos a procesar. Estos hilos se organizan en una jerarquía de dos niveles utilizando coordenadas únicas (*blockIdx* y *threadIdx*). Estas coordenadas son asignadas por el sistema de CUDA en tiempo

de ejecución. El *blockIdx* y *threadIdx* aparecen como variables embebidas y preinicializadas que se pueden acceder dentro de las funciones *kernel*. Adicionalmente al *blockIdx* y *threadIdx* existen dos variables embebidas más, *gridDim* y *blockDim* que proveen la dimensión del grid y del bloque, respectivamente (Kirk y Hwu, 2010).

La Figura 12 muestra un ejemplo sencillo de cómo se organizan los hilos en CUDA. El grid consiste de tres bloques, cada uno con un identificador *blockIdx.x* que va desde 0 hasta 2. Cada bloque contiene ocho hilos, cada uno con su indentificador *threadIdx.x* con valores de 0 a 7. Para obtener el índice global de cada hilo se utiliza la instrucción $threadID = blockIdx.x \times blockDim.x + threadIdx.x$.

Esta organización es unidimensional, tanto en los bloques como en los hilos. Por lo tanto, el grid contiene 3×8 hilos.

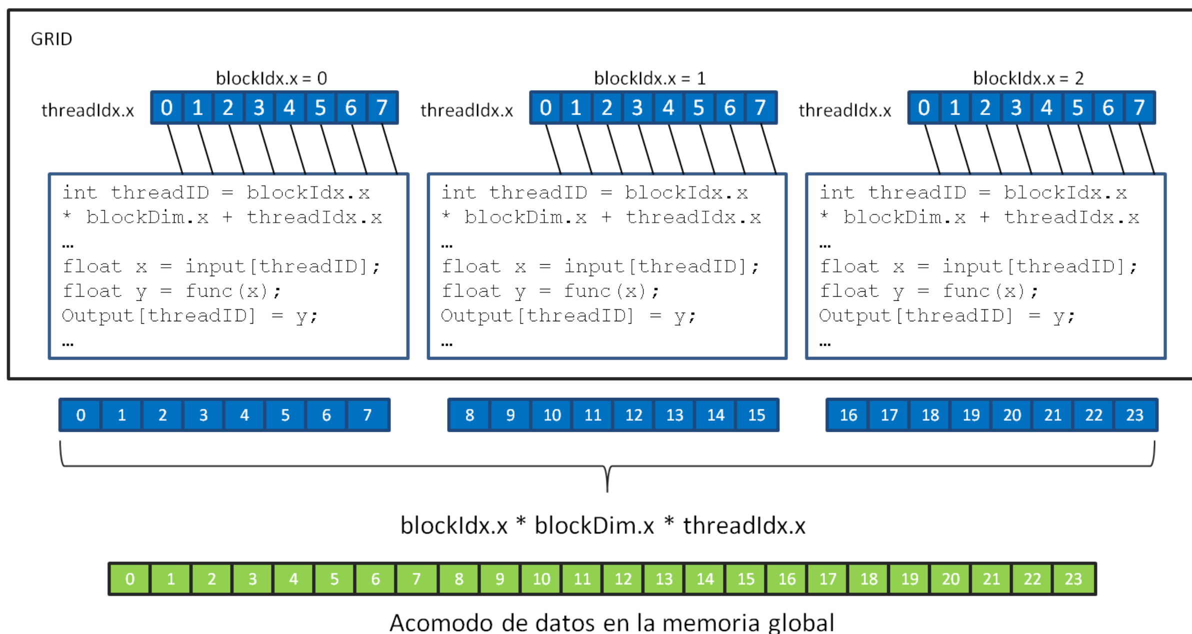

Figura 12: Información general sobre la organización de los hilos en CUDA.

El ejemplo anterior muestra la organización de los hilos en una dimensión, pero un grid tiene la capacidad de ser organizado en arreglos bidimensionales de bloques, a su vez, cada bloque contiene arreglos tridimensionales de hilos. El primer parámetro de la llamada a una función kernel especifica las dimensiones del grid en términos del número de bloques. El segundo especifica las dimensiones de cada bloque en términos de número de hilos.

Para acceder a estás dimensiones es necesario utilizar el tipo de dato “dim3” de C, que esencialmente es una estructura con tres campos enteros sin signo: x, y y z. Debido a que el grid es un arreglo bidimiensional de bloques, el tercer campo de la dimensión del grid se ignora, pero se le debe asignar el valor de 1 para mayor claridad.

La Figura 13 muestra un grid de 2D que se genera a partir del código de CUDA que se muestra en la parte superior.

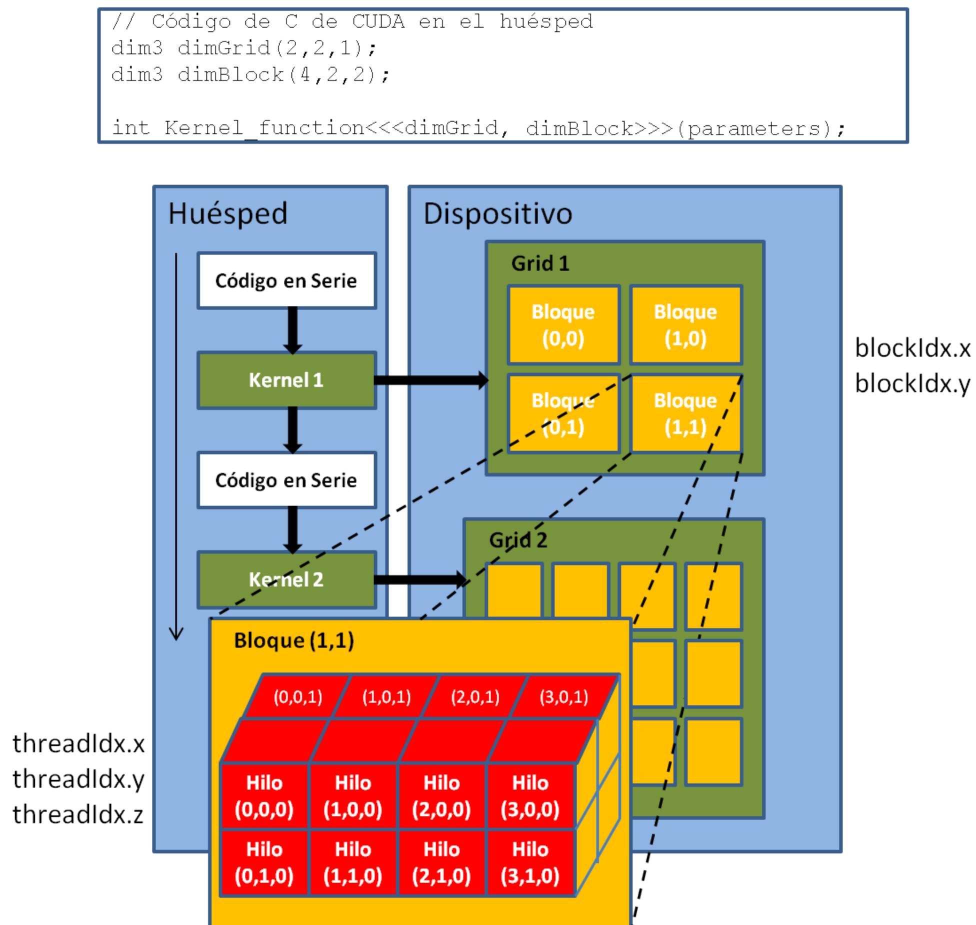

Figura 13: Ejemplo de organización multidimensional de un grid de CUDA.

El grid se compone de cuatro bloques organizados en un arreglo de $2{\times}2$. Cada bloque en la Figura 13 se accede por medio de las coordenadas $blockIdx.x$ y $blockIdx.y$; por ejemplo, Block (1,0) tiene los valores de $blockIdx.x = 1$ y $blockIdx.y = 0$.

En general, los bloques se organizan en conjuntos tridimensionales de hilos. Todos los blo-

ques de un grid tienen las mismas dimensiones. Cada threadIdx consta de tres componentes: la coordenada x en *threadIdx.x*, la coordenada y en *threadIdx.y* y la coordenada z en *threadIdx.z* (Kirk y Hwu, 2010). El número de hilos en cada dimensión de un bloque se especifica en el segundo parámetro de la llamada al *kernel*. El tamaño total de hilos por bloque está limitado por la arquitectura del GPU, en el caso de los dispositivos con arquitectura 2.x, el límite es de 512 hilos, y se tiene la flexibilidad de distribuir estos elementos en las tres dimensiones, siempre y cuando el número total de hilos no exceda el límite. Por ejemplo, en el caso de 512 hilos podemos distribuir como sigue: (512, 1, 1), (8, 16, 2), y (16, 16, 2), pero el blockDim (32, 32, 1) no es admisible porque el total de número de hilos sería 1024.

3.7.1. Warp

El multiprocesador crea, administra, programa y ejecuta los hilos en grupos de 32 hilos paralelos llamados *warps*. Los hilos individuales que componen un *warp* comienzan juntos en la misma instrucción del programa, pero cada uno tienen su propia dirección para el contador de instrucciones y registro de estados, por lo tanto son libres de hacer bifurcaciones y ejecutarse de manera independiente (Mor, 2013).

Cuando a un multiprocesador se le asignan uno o más bloques a ejecutar, este los particiona en *warps* y cada *warp* es calendarizado para su ejecución por el programador de *warps*. La manera en que un bloque se particiona en *warps* es siempre la misma; cada *warp* contiene hilos con IDs consecutivos donde el primer *warp* contiene al hilo 0. La jerarquía de los hilos describe cómo se relacionan los IDs de hilos con el índice que tienen estos dentro de un bloque.

3.7.2. Divergencia

Cualquier instrucción de control de flujo (if, switch, do, for, while) puede afectar significativamente el rendimiento al causar que los hilos de un *warp* diverjan, es decir, seguir diferentes rutas de ejecución. Si esto ocurre, las diferentes rutas de ejecución deben ser serializadas, aumentando el número total de instrucciones ejecutadas para ese *warp* en particular. Cuando las diferentes rutas de ejecución se han completado, los hilos convergen de nuevo a la misma ruta de ejecución (NVIDIA, 2011).

El hardware ejecuta una instrucción para todos los hilos de un *warp*, antes de pasar a la siguiente instrucción. Este método de ejecución SIMD está pensada por las restricciones del hardware, ya que permite que el costo y procesamiento de una instrucción sea compartido por un gran número de hilos. Esto funciona bien cuando todos los hilos dentro de un *warp* siguen la misma trayectoria de control de flujo. Por ejemplo, la instrucción *if-then-else* funciona bien cuando todos los hilos ejecutan la parte del *then* o, todos ejecutan la parte del *else*. Cuando los hilos dentro de un *warp* toman diferentes rutas de control de flujo, el método de ejecución SIMD ya no funciona de manera óptima. Esto es si en el *if-then-else*, algunos hilos ejecutan la parte del *then* y otros ejecutan la parte *else*. En tal situacion, la ejecución del *warp* requerirá múltiples pasadas a través de estos caminos divergentes. Se necesitará un pasada para los hilos que siguen al *then*, y otra para aquellos que siguen la parte del*else*. Estas pasadas son secuenciales, aumentando así el tiempo de ejecución. Cuando los hilos de un mismo *warp* siguen diferentes caminos de control de flujo, decimos que estos divergen en su ejecución (Kirk y Hwu, 2010).

Para obtener el mejor rendimiento en los casos en que el flujo de control dependa del ID del hilo, la condición de control debe ser escrita con el fin de minimizar el número de hilos divergentes. La eficiencia máxima se obtiene cuando los 32 hilos de un *warp* tienen el mismo camino a ejectuar. Un ejemplo trivial ocurre cuando la condición de control sólo depende de ($threadIdx$ / $warpSize$) donde $warpSize$ es el tamaño del *warp*. En este caso ningún *warp* diverge ya que la condición de control está perfectamente alineada con los hilos del *warp*. La divergencia entre hilos ocurre solamente dentro de un *warp*; cada *warp* se ejecuta independientemente sin importar si están ejecutando rutas de código en común o distintas (NVIDIA, 2011).

3.7.3. Sincronización

CUDA permite que los hilos de un mismo bloque coordinen sus actividades al utilizar la función *syncthreads()* como una barrera de sincronización. Cuando una función kernel manda llamar a *syncthreads()*, el hilo que ejecute la llamada a esta función se detendrá en ese punto hasta que todos los hilos dentro de un bloque lleguen a esta ubicación. Esto asegura que todos

los hilos en un bloque han completado una fase de su ejecución en el *kernel* antes de pasar a la siguiente fase (Kirk y Hwu, 2010). Con esto se evitan problemas de accesos a memoria global y compartida.

3.7.4. Calendarización

La calendarización de hilos es estrictamente un concepto de implementación, y debe ser discutido específicamente en el contexto del hardware utilizado para dicha implementación. Hasta el día de hoy, cada vez que un bloque es asignado a un multiprocesador de flujos (SMX), este se divide en unidades de 32 hilos llamados *warps*. El tamaño de los *warps* es específico de la implementación. De hecho, los *warps* no son parte de las especificaciones de CUDA, sin embargo, el conocimiento del número de hilos dentro de un *warp* puede ser útil en la comprensión y optimización del rendimiento de las aplicaciones desarrolladas en CUDA (Sanders y Kandrot, 2010).

El *warp* es la unidad de calendarización de hilos básica dentro de un SMX, ya que es la forma en que los procesadores CUDA ejecutan de manera eficiente las operaciones de alta latencia, como los accesos a la memoria global. Si una instrucción ejecutada por los hilos de un *warp* está en espera del resultado de una operación de alta latencia iniciada previamente, entonces dicho *warp* no es seleccionado por el calendarizador, en su lugar otro *warp* que no esté a la espera de resultados es seleccionado para su ejecución. Si más de un *warp* está listo para su ejecución, un mecanismo de prioridad se utiliza para seleccionar a uno de ellos. Este mecanismo de llenar la latencia de operaciones costosas con trabajo de otros hilos se conoce como *ocultamiento de latencia* (Kirk y Hwu, 2010).

La calendarización de *warps* se utiliza para cubrir otro tipo de operaciones con alta latencia como la aritmética de punto flotante, el “pipeline” y la bifurcación de instrucciones. Con suficientes *warps* esperando a ser atendidos, es muy probable que el hardware encontrará un *warp* para ser ejecutado en cualquier punto en el tiempo, haciendo con esto, pleno uso del hardware a pesar de las operaciones de alta latencia. La selección de *warps* que están listos para ser ejecutados no introduce ningún tiempo de inactividad en el tiempo de ejecución, lo que se conoce como *calendarización de hilos con sobrecarga cero.*

Esta capacidad de tolerar operaciones de alta latencia es la razón principal por la cual las unidades de procesamiento gráfico (GPU) no dedican tanta área del chip a memoria caché como sí lo hacen los CPUs. Como resultado, las GPU pueden dedicar más área del chip a recursos para la ejecución de punto flotante (Mor, 2013).

3.8. Jerarquía de las memorias

CUDA maneja varios tipos de memoria que se utilizan en diferentes escenarios para optimizar los tiempos de ejecución de los *kernels*. Cada hilo tiene memoria local privada. Cada bloque contiene memoria compartida que es visible para todos los hilos contenidos dentro de él y con la misma duración que la del bloque. Todos los hilos tienen acceso a la misma memoria global (NVIDIA, 2011). La Figura 14 muestra los diferentes tipos de memoria dentro del dispositivo CUDA. En la parte inferior de esta figura se muestra la memoria global y la memoria constante. El huésped puede escribir y leer en este tipo de memorias por medio de funciones de la interfaz de programación de aplicaciones (API) de CUDA.

La memoria global es de alta latencia, tiene un tiempo de vida igual al de la aplicación y es accesible para todos los hilos del *kernel*. Esta debe ser declarada en el huésped, fuera de la función *kernel* y se debe liberar de nuevo en el huésped al terminar el proceso. Es importante controlar la lectura y escritura a memoria global debido a que la ejecución de los hilos no puede ser sincronizada a través de diferentes bloques.

La memoria constante es de baja latencia, gran ancho de banda, y acceso de sólo lectura por el dispositivo. Esta memoria también debe ser declarada en un contexto global (fuera de la función *kernel*). Existe una cantidad limitada de memoria constante que puede ser declarada (64 Kb para todas las capacidades de cómputo). El valor contenido en esta memoria no cambiará entre diferentes llamados al *kernel* a menos que se especifique en el proceso del huésped. Al igual que la memoria global la memoria constante tiene un tiempo de vida igual a la aplicación.

Los registros y la memoria compartida en la Figura 14 son memorias contenidas dentro

del chip. Las variables que residen en este tipo de memoria se pueden acceder a muy alta velocidad de forma paralela. Los registros se asignan a cada hilo individualmente; cada hilo sólo puede acceder a sus propios registros. Una función *kernel* normalmente utiliza los registros para mantener a las variables escalares de acceso frecuente que son privadas para cada hilo (Kirk y Hwu, 2010).

La memoria compartida se declara dentro del contexto del *kernel* pero tiene el tiempo de vida de un bloque. Cuando el bloque termina su ejecución, la memoria que se definió en el *kernel* ya no se puede acceder. Todos los hilos de un bloque pueden acceder a las variables contenidas en la memoria compartida que se asigna al bloque. La memoria compartida es un medio eficaz donde los hilos cooperan al intercambiar datos de entrada y resultados intermedios de su trabajo. Al declarar una variable en uno de los tipos de memoria CUDA, una desarrollador determina la visibilidad y la velocidad de acceso a las variables.

En resumen, CUDA define registros, memoria compartida y memoria constante que se pueden acceder a más alta velocidad y de mejor forma paralela que la memoria global. Para utilizar estas memorias de forma eficiente es probable que sea necesario un rediseño del algoritmo implementado.

3.9. Funciones especiales

En esta sección se explican dos conjuntos de instrucciones que forman parte de la extensión del lenguaje C de CUDA. Estas funciones de uso específico no son necesarias para la comprensión del esquema general de CUDA pero sí para explicar detalladamente la solución propuesta para la detección de SCAPs en el Capítulo 4, ya que que se utilizaron en el desarrollo de esta. El contenido de esta sección tiene como referencia la guía de NVidia "CUDA C Programing Guide" (NVIDIA, 2008).

3.9.1. Funciones Atomic

Las funciones *atomic* de CUDA ejecutan operaciones sin la interferencia de otros hilos. Estas operaciones se utilizan frecuentemente para evitar los problemas de "condición de carrera" que son comunes en las aplicaciones con multihilos. Una operación *atomic* es capaz de

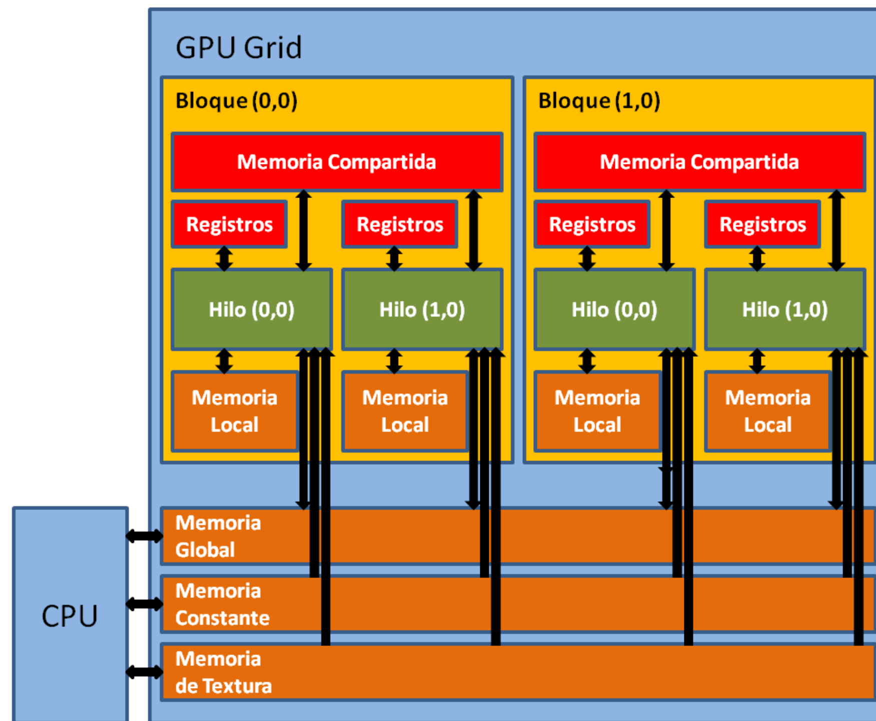

Figura 14: Información general sobre el modelo de memoria del dispositivo CUDA.

leer, modificar y escribir un valor en memoria sin la interferencia de otros hilos, lo que garantiza que la condición de carrera no ocurrirá. Las operaciones pueden operar en la memoria compartida o la memoria global. La instrucciones *atomic* dentro de la memoria compartida se utilizan para prevenir la condición de carrera entre hilos de un mismo bloque. En memoria global previenen la condición de carrera entre diferentes hilos sin importar a qué bloque pertenecen. Las operaciones *atomic* en memoria compartida son más rápidas que las que se ejecutan en memoria global. Las funciones atomic sólo se pueden utilizar dentro de las funciones del dispositivo.

Ejemplo:

int *atomicAdd*(int* *address*, int *val*);

Cuando un hilo ejecuta esta operación, se lee el valor *old* (no se muestra en los parámetros) contenido en la dirección de memoria *address*, se ejecuta la operación ($old + val$) y el resultado se escribe de nuevo en *address*. El valor original *old* contenido en la dirección *address* (antes de la suma) se asigna al hilo.

Además de la operación atómica Suma (Add), existen ocho operaciones aritméticas y tres de operación de bit (Tabla 5).

Tabla 5: Operaciones *atomic*.

Función	**Operación Aritmética**
atomicAdd()	(*old* + *val*)
atomicSub()	(*old* - *val*)
atomicExch()	*val* remplaza a *old*
atomicMin()	se almacena el mínimo entre *old* y *val*
atomicMax()	se almacena el máximo de *old* y *val*
atomicInc()	((*old* >= *val*) ? 0 : (*old*+1)),
atomicDec()	$(((old == 0) \mid (old > val))$? *val* : (old-1))
atomicCAS()	(*old* == compare ? *val* : *old*)
Función	**Operación de Bit**
atomicaAnd()	(*old* & *val*)
atomicOR()	$(old \mid val)$
atomicXOR()	(*old* ^ *val*)

3.9.2. Funciones Shuffle

Las funciones *Shfl* permiten el intercambio de una variable entre los hilos dentro de un *warp* sin la necesidad de utilizar la memoria compartida. El intercambio se produce de forma simultánea para todos los hilos activos dentro de un *warp*, pasando 4 bytes de datos por hilo. El intercambio de cantidades de 8 bytes debe ser dividido en dos llamadas separadas de la función shfl. Estas instrucciones solo son compatibles con dispositivos de capacidad de cómputo 3.x en adelante.

A los hilos dentro de un *warp* se les conoce como *líneas*. El índice de cada línea está entre los valores 0 y el tamaño del *warp*-1 (inclusive). Existen cuatro modos de direccionamiento de líneas:

- int __shfl(int *var*, int *srcLane*, int *width*=*warpSize*)

  Esta instrucción devuelve el valor *var* contenido en la línea objetivo *srcLane*. En el ejemplo de la Figura 15 se muestra como todos los hilos del *warp* intercambian su valor por el de la línea 2 con valor *c*, logrando con esto una transmisión o “broadcast”. Si *srcLane* está fuera del intervalo [0: *width*-1], entonces el valor del hilo que hizo la llamada es lo que devuelve esta función.

- int __shfl_up(int *var*, unsigned int *delta*, int *width*=*warpSize*)

  Esta instrucción devuelve el valor de un hilo contenido en una línea superior indicado

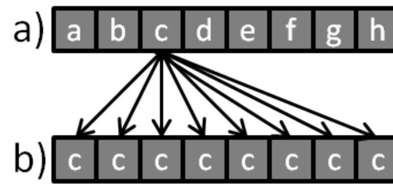

Figura 15: Operación *shfl* donde todos los hilos obtienen el valor de la línea 2 ("c"). En a) se muestra el *warp* antes de aplicar la función *shfl*, en b) el *warp* después de aplicar la función.

por el parámetro *delta*. Esto tiene el efecto de desplazar *var* hacia adelante *delta* número de líneas dentro del *warp*. Si el índice de la línea fuente excede el valor de *width* simplemente se desecha, así que las líneas en la parte inferior de *delta* permancerán sin cambios. En la Figura 16 el párametro $delta = 2$, lo que indica pasar el valor de cada hilo dos posiciones a la derecha.

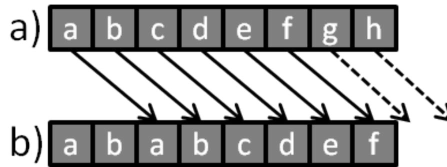

Figura 16: Operación *shfl_up* con $delta = 2$. En a) se muestra el *warp* antes de aplicar la función *shfl*, en b) el *warp* después de aplicar la función.

- int __shfl_down(int *var*, unsigned int *delta*, int *width=warpSize*)

  Esta instrucción devuelve el valor de un hilo contenido en una línea inferior indicado por el parámetro *delta*. Esto tiene el efecto de desplazar *var* hacia atrás *delta* número de líneas dentro del *warp*. Si el índice de la línea fuente excede el valor de *width* simplemente se desecha, así que las líneas en la parte superior de *delta* permancerán sin cambios. En la Figura 17 el párametro $delta = 1$, lo que indica pasar el valor de cada hilo una posición a la izquierda.

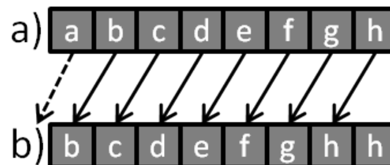

Figura 17: Operación *shfl_down* con $delta = 1$. En a) se muestra el *warp* antes de aplicar la función *shfl*, en b) el *warp* después de aplicar la función.

- int __shfl_xor(int *var*, int *laneMask*, int *width=warpSize*)

  Realiza un intercambio bit a bit entre el valor del hilo que hace el llamado y el indicado por el parámetro *laneMask* por medio de la compuerta XOR. Si el ID calculado se encuentra fuera del intervalo permitido por *width*, el propio valor de hilo es devuelto. A este patrón también se le conoce como “mariposa”. Un ejemplo de esta instrucción se muestra en la Figura 18 con el argumento *laneMask* = 2.

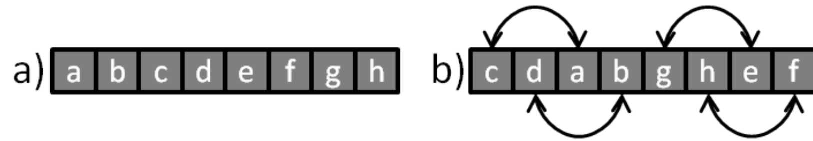

Figura 18: Operación *shfl_xor*. En a) se muestra el *warp* antes de aplicar la función *shfl*, en b) el *warp* después de aplicar la función.

Los hilos solo pueden leer datos de otro hilo que esté participando activamente en el comando *_shfl()*. Si el hilo objetivo está inactivo, el valor de retorno es indefinido.

Todas las funciones *_shfl()* reciben un parámetro adicional *width* el cual divide el *warp* en segmentos; por ejemplo, para intercambiar datos entre 4 grupos de 8 líneas de forma SIMD. Si *width* es menor que el tamaño del *warp* entonces cada sección del *warp* se comporta como una entidad independiente con un valor local inicial de línea igual a 0. Un hilo solo podrá intercambiar datos con otros hilos contenidos en la misma sección. El valor de *width* debe ser una potencia de 2 para que el *warp* pueda ser dividido de manera equitativa. Cuando *width* no es potencia de dos o si *width* es mayor que el tamaño del *warp* los resultados no están definidos.

Capítulo 4. Algoritmos

En este capítulo se explica la estrategia de paralelización diseñada e implementada para el problema de detección de SCAPs. Además se incluye un análisis breve de estrategias alternativas que se estudiaron. Posteriormente se describe a detalle la implementación de los algoritmos de la estrategia propuesta, estos incluyen la preparación de la secuencia de aminoácidos que se introduce en el huésped, la paralelización y escritura de los SCAPs en el dispositivo y finalmente el análisis de resultados arrojados al huésped.

El cálculo de las propiedades fisicoquímicas: Media de Hidrofobicidad (MH), Media de Carga Neta (MC), Punto Isoeléctrico (PI) y Momento Hidrofóbico Helicoidal (MHH) tienen diferentes aplicaciones en el diseño de péptidos (Polanco *et al.*, 2011). Los péptidos se componen de dos o más aminoácidos unidos por enlaces peptídicos, por cada posición de la cadena existen 20 aminoácidos a considerar, por esto, el número total de secuencias que se analizan para verificar si cumplen con las características del diseño crece de manera exponencial con la longitud de la secuencia peptídica (por ejemplo, un secuencia de 10 residuos genera 1.024 $\times$ 10^{13} péptidos diferentes). Como el espacio de búsqueda es muy grande resulta evidente la necesidad de desarrollar un enfoque diferente a la estrategia de enumeración secuencial para generar estos péptidos en el menor tiempo posible.

Un algoritmo secuencial analiza una cadena de aminoácidos en cada iteración dentro de un ciclo. Por cada cadena se realiza el cálculo de las propiedades propuestas para determinar si esta es o no SCAP. El tiempo de ejecución de dicho algoritmo verificador es de $O(n)$ para un péptido de n residuos.

Para poder calcular las propiedades en secuencias de hasta longitud 10, se diseñó un algoritmo paralelo que posteriormente se implementó en el procesador gráfico (GPU) aprovechando la capacidad de cómputo que estos presentan.

4.1. Estrategías de paralelización evaluadas en la arquitectura de GPU

Gracias a las herramientas que ofrece CUDA, se pueden plantear diferentes estrategias de paralelización y así agilizar los cálculos en la detección de SCAPs. A continuación se mencionan dos enfoques que fueron estudiados antes de lograr el diseño final.

En el primero de ellos las propiedades se analizan una a una utilizando *kernels* en serie. Para optimizar este proceso, las propiedades se ordenan de menor a mayor tiempo de ejecución con la intención de eliminar el mayor número de secuencias negativas en el menor tiempo posible tal y como se muestra en la Figura 19. Esto es, primero se calcula la media de carga y la media de hidrofobicidad (paso A de la misma figura), se almacenan solo los resultados positivos y estos se utilizan como parámetro para la siguiente llamada al *kernel*. Este calculará la siguiente propiedad con mayor tiempo de ejecución que en este caso, es el punto isoeléctrico (paso B). Una vez más, se pasa solo el conjunto de los resultados positivos hacia la siguiente llamada al *kernel* donde se calcula el momento hidrofóbico helicoidal (paso C). Al finalizar el cálculo de las cuatro propiedades, se tiene el conjunto de SCAPs.

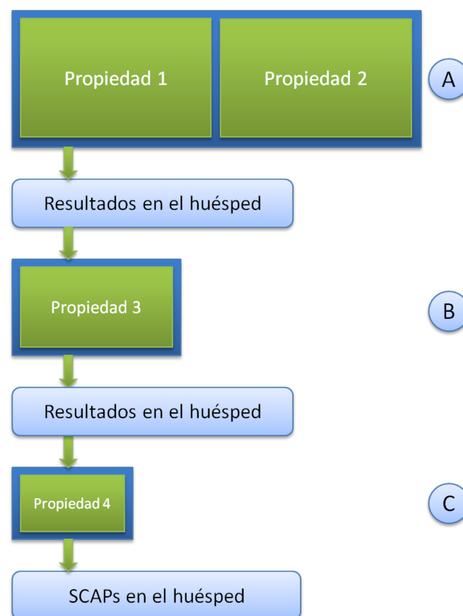

Figura 19: Estrategia de paralelización analizando propiedades en serie. Las propiedades se ejecutan de menor a mayor tiempo de ejecución, en cada paso avanzan solo las secuencias positivas, por esta razón el recuadro disminuye de forma simbólica con cada propiedad analizada.

La desventaja de este enfoque son las múltiples llamadas al *kernel*. La transferencia de datos es menor con cada iteración, la comunicación entre el dispositivo y el huésped está

activa durante un lapso apreciable de tiempo.

En el segundo enfoque se ejecuta múltiples *kernels* de manera simultánea, esto es posible gracias a la flexibilidad que CUDA provee. Con este enfoque se puede simular el trabajo de FPGAs de Polanco *et al.* (2011), creando un conjunto de hilos para cada una de las propiedades que definen un SCAP (MH, MC, PI, MHH). Estos hilos, se ejecutan en cuatro *kernels* diferentes de forma concurrente como se indica en el paso 'A' de la Figura 20, es decir un *kernel* por propiedad. Por lo tanto el tiempo que tarda el algoritmo en determinar si un péptido es o no SCAP, se asocia directamente con la propiedad que toma más tiempo en calcularse. En este estudio, la propiedad que determina el tiempo total de evaluación es el Momento Hidrofóbico Helicoidal (contenida en el *kernel* 4 de la misma figura).

El inconveniente que surge al calcular las propiedades por separado en el paso 'A', es que se necesita un paso adicional para unir los resultados; lo que implica que el huésped (paso B) debe reservar memoria de nuevo en el dispositivo para almacenar los resultados obtenidos y hacer una segunda llamada al *kernel*. En el paso 'C' el dispositivo combina los parámetros que recibe para obtener los SCAPs; estos resultados los envía de vuelta al huésped como se muestra en el paso 'D'.

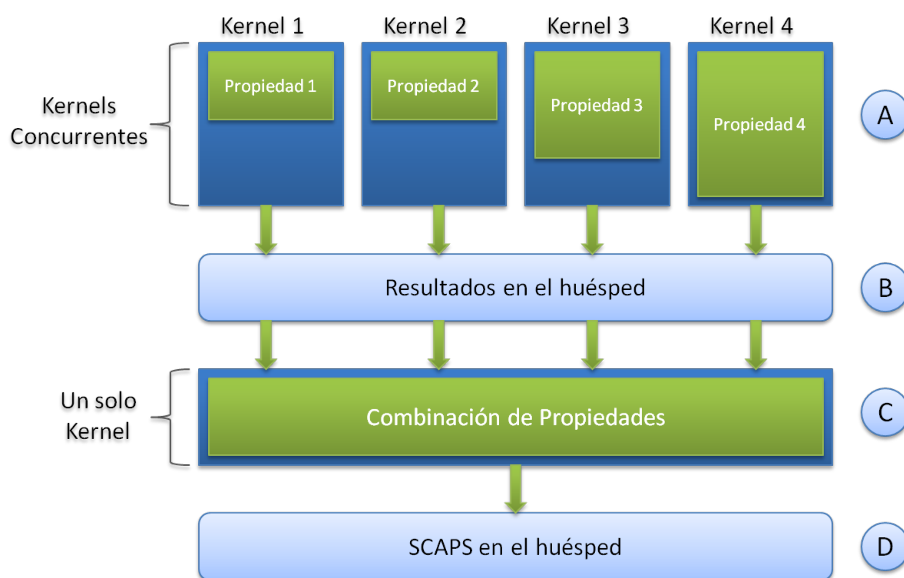

Figura 20: Estrategia de paralelización con *Kernels* concurrentes.

Aunque este segundo enfoque realiza menos llamadas al dispositivo que el esquema de

kernels en serie antes mencionado, era necesario buscar una estrategia donde de ser posible se hicieran todos los cálculos con una sola llamada al dispositivo, pues es justamente esta operación la que más tiempo de ejecución requiere, además una sola llamada al dispositivo evitaría la necesidad de traspasar resultados intermedios y reservar memoria para estos. Es con estos requerimientos que se diseñó el esquema que a continuación se presenta.

4.2. Diseño de paralelización propuesto

La estrategia propuesta es sencilla desde el punto de vista de la paralelización, pero más eficiente en cuanto a tiempo de ejecución. El esquema general se presenta en la Figura 21 y se describe de forma concisa en los siguientes puntos; los detalles de cada uno de estos pasos se explica más adelante con cada algoritmo.

- Preparar en el huésped la configuración y los parámetros que se utilizarán en el dispositivo (el parámetro principal es el arreglo h_SCAPs donde se almacenan los casos positivos).
- Llamar al dispositivo con los parámetros y la configuración de bloques e hilos establecida en el paso anterior.
- Generar las secuencias a partir de los índices de lo hilos. El índice del hilo en base 10 se convierte a base 20 donde cada dígito representa un aminoácido y cada hilo analiza una secuencia.
- Determinar si la secuencia es SCAP o no con base en las cuatro propiedades fisicoquímicas analizadas.
- Calcular mediante la suma de prefijos el índice en memoria global del dispositivo donde se almacenará la secuencia.
- Si la secuencia es SCAP almacenarla en la memoria global del dispositivo.
- Retornar el arreglo con casos positivos al huésped.
- Mostrar el arreglo de casos positivos al usuario.

La principal ventaja de este enfoque, es que el número de secuencias que se puede evaluar por llamada al dispositivo depende del índice máximo disponible, este se obtiene de multiplicar el número máximo de bloques por grid, por el número máximo de hilos por bloque, en nuestro caso $2^{31} - 1 \times 1024$ (Apéndice A.3). Además, no existe la necesidad de hacer transferencia de datos desde el huésped al dispositivo, es decir, la información que necesita el dispositivo para trabajar es mínima. Al termino de los cálculos, el dispositivo manda al huésped un arreglo que contiene únicamente casos positivos, por lo que el flujo de transferencia de datos se mantiene al mínimo.

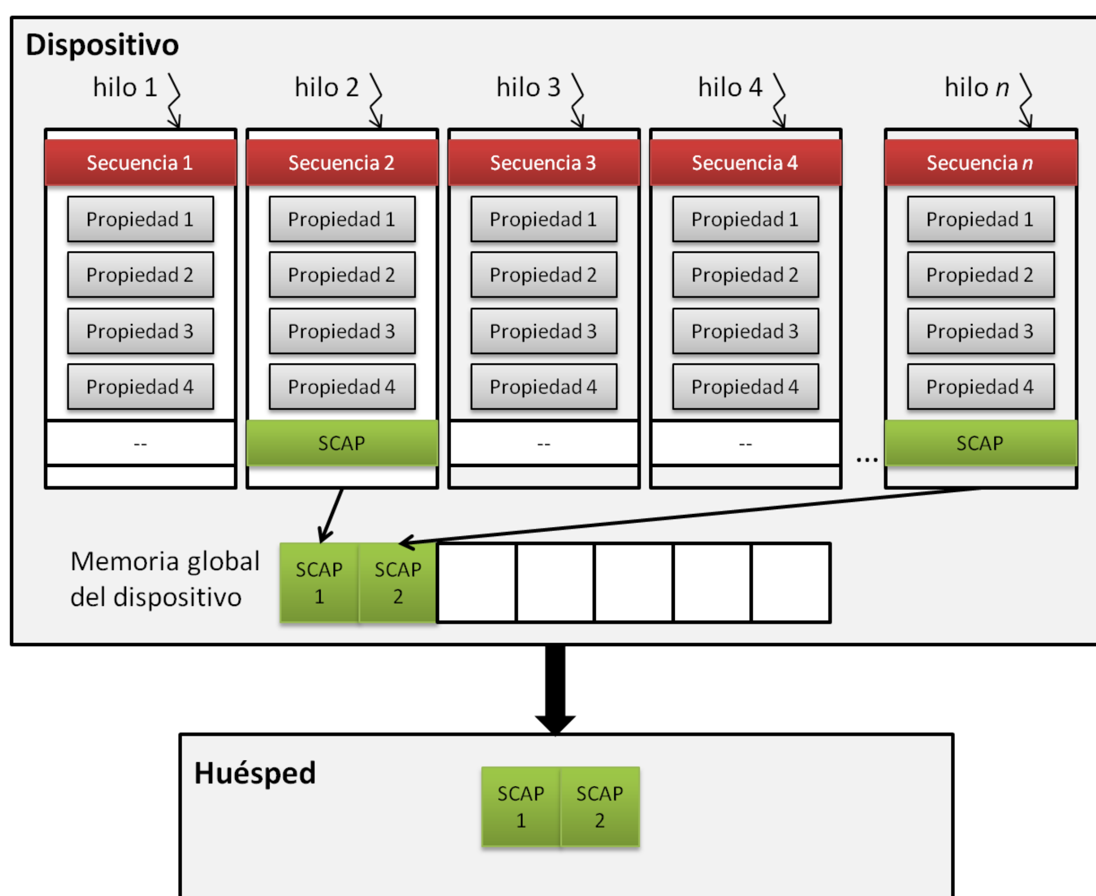

Figura 21: Estrategia de paralelización propuesta. Cada hilo calcula las cuatro propiedades de una secuencia de aminoácidos y determina si es o no un SCAP.

4.2.1. Consideraciones para el diseño de paralelización propuesto

Esta estrategia se configuró para analizar péptidos de 7 hasta 25 aminoácidos, con un máximo de 10 aminoácidos de posición y valor variable. La cota inferior lo impone el cálculo

del momento hidrofóbico helicoidal; la cota superior lo impone el dispositivo a utilizar.

La secuencia de aminoácidos a analizar se escribe en *código de una letra*, indicando los aminoácidos variables con la letra 'X' (Figura 22). Con esta información, el huésped prepara los bloques y configuración de hilos, así como los parámetros que se ejecutan dentro del *kernel*. La configuración del *kernel* depende del número de aminoácidos variables dentro de la secuencia a analizar. Con el hardware que se utilizó, el máximo posible de aminoácidos variables que se pueden analizar con una sola llamada al *kernel* es de siete. Las secuencias que contienen más de siete aminoácidos variables requieren múltiples llamadas al dispositivo.

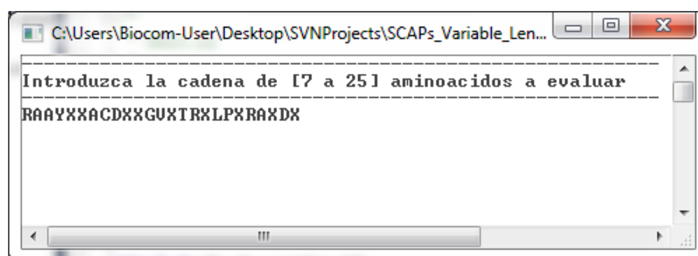

Figura 22: Formato de una secuencia de 25 aminoácidos introducida en el programa. Los aminoácidos variables se representan con 'X'.

Una vez configurado el *kernel* comienza el proceso paralelo. Cada hilo analiza una secuencia que se genera a partir del índice del hilo; dentro de cada hilo se calculan las cuatro propiedades mencionadas para determinar si la secuencia es SCAP o no. Si el resultado es positivo, la cadena se almacena en la memoria global del dispositivo; este último manda los resultados al huésped en un solo arreglo al terminar el análisis.

4.2.2. Estructuras de datos para almacenar SCAPs

Cuando una secuencia resulta ser SCAP, esta se almacena en una estructura de datos de C++. Según sea el caso de estudio existen dos estructuras diferentes a elegir: La primera, *Estructura Completa*, almacena todos los datos que contiene el SCAP, estos se muestran en la estructura de la izquierda de la Figura 23. Esta información es útil para verificar los valores de cada SCAP encontrado, esta estructura ocupa 48 bytes de memoria. El segundo caso, *Estructura Corta*, se muestra a la derecha en la misma figura; esta almacena únicamente el *Id* del hilo y el índice donde se almacenará en la memoria global, esta estructura ocupa 16 bytes. Esta última se propone con el objeto de usar menos espacio.

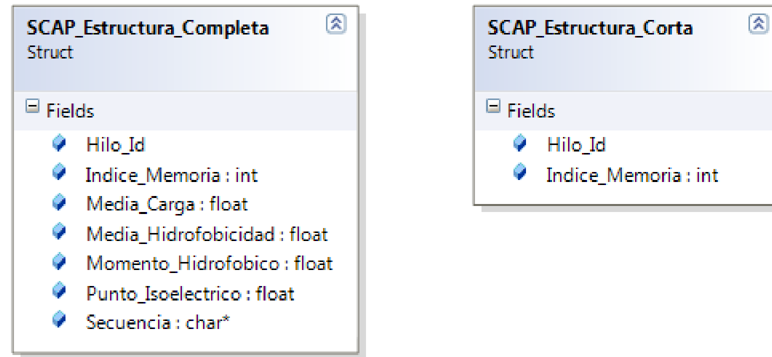

Figura 23: Estructuras de datos para almacenar SCAPs.

4.3. Algoritmos

Los algoritmos de la paralelización propuesta se describen a continuación.

4.3.1. Función principal

La función principal o "*main*" es el punto de inicio a los cálculos de SCAPs. El algoritmo que a continuación se describe se ejecuta del lado del huésped. Su objetivo principal es preparar la cadena que se analizará en el proceso paralelo.

Algoritmo 1 Principal (main)

Entrada: -
Salida: El arreglo h_SCAPs con los casos positivos.
1: $MAX_ANALISIS = 7$
2: $secuencia$ = Obtener secuencia a analizar
3: aa_var = número de aminoácidos variables dentro de $secuencia$
4: $secuencia20$ = Convertir $secuencia$ a base 20
5: pos_var = Guardar posición de los aminoácidos variables
{Declarar arreglo para almacenar SCAPs}
6: arreglo h_SCAPs[MAX_CAPACIDAD_GPU]
7: **si** $aa_var \leq MAX_ANALISIS$ **entonces**
8: $iteraciones = 1$
9: **si no**
10: $iteraciones$ = potencia(20, aa_var - $MAX_ANALISIS$)
11: **fin si**
12: **para** $i \leftarrow 0, iteraciones$ **hacer**
13: prepararKernel ($secuencia20, h_SCAPs, pos_var, iteraciones$)
14: **fin para**
15: Imprimir (h_SCAPs)

La constante del paso 1 indica el número máximo de aminoácidos variables que se pueden

analizar con una sola llamada al GPU. Esta constante se puede cambiar según la capacidad del dispositivo que se utilice.

En el paso 2 se obtiene la secuencia a analizar. Para introducir la secuencia es necesario escribir los aminoácidos en *código de una letra*, se utiliza "X" para indicar aminoácidos variables.

En el paso 3 se cuentan los aminoácidos variables dentro de la cadena propuesta.

En el paso 4 la secuencia se convierte a un número en base 20, donde cada dígito es equivalente a uno de los 20 aminoácidos esenciales. Esta conversión se hace utilizando la Tabla 6, la cual facilita los cálculos dentro del dispositivo.

Tabla 6: Relación de los aminoácidos con su equivalente en la representación en base 20.

AA C3	AA C1	Conv. base 20	AA C3	AA C1	Conv. base 20
Ala	A	0	Met	M	10
Cys	C	1	Asn	N	11
Asp	D	2	Pro	P	12
Glu	E	3	Gln	Q	13
Phe	F	4	Arg	R	14
Gly	G	5	Ser	S	15
His	H	6	Thr	T	16
Ile	I	7	Val	V	17
Lys	K	8	Trp	W	18
Leu	L	9	Tyr	Y	19

Para generar correctamente las secuencias que se analizarán dentro del dispositivo, es necesario guardar las posiciones de los aminoácidos variables en un arreglo distinto; estos se almacenan de forma inversa a la que aparecen en la secuencia. Este procedimiento se ejecuta en el paso 5.

Los resultados positivos que arroje el proceso paralelo se almacenan en el arreglo h_SCAP que se declara en el paso 6. En este algoritmo, se reserva toda la memoria del dispositivo para asegurar que el GPU almacene todos los resultados al terminar el procesamiento.

Si el número de aminoácidos variables es mayor que el número que puede analizar el dispositivo en una sola llamada entonces se hacen n llamadas al dispositivo. El número de iteraciones necesarias se calcula con la fórmula del paso 10. Las iteraciones están en potencia de 20 por la naturaleza del problema.

La función "prepararKernel" del paso 13 se encargará de preparar los parámetros para la llamada al GPU por medio del *kernel* y de recibir los resultados que este arroja.

Al terminar el proceso de detección de SCAPs, en el paso 15 se imprimen los resultados que se encuentran dentro del arreglo h_SCAPs. En este proceso los resultados se escriben en archivos con valores *separados por comas* (.csv por sus siglas en inglés). La función "Imprimir" de este paso se puede reemplazar fácilmente por otro método según las necesidades que se tengan para mostrar los resultados.

La Tabla 7 muestra dos ejemplos donde se hace la conversión de la secuencia a una cadena en base 20 y la configuración del arreglo con las posiciones de los aminoácidos variables. La primer secuencia de la columna "Cadena" contiene únicamente aminoácidos variables que se representan con la letra 'X'; la segunda secuencia (de longitud 25) se compone de aminoácidos fijos y variables. En la columna "Base 20" se muestran las mismas cadenas pero con la representación en base 20, la conversión se hace con la relación de la Tabla 6. En la columna "Pos Variables" se muestra la posición de los aminoácidos variables ordenados de forma inversa a la que aparecen en la secuencia (i.e., de derecha a izquierda).

Tabla 7: Ejemplos de cadenas para analizar (columna Cadena), la conversión a base 20 de estas cadenas (columna Base 20) y del arreglo con la posición de los aminoácidos variables ordenados de forma inversa a la que aparecen en la secuencia (columna Pos Variables).

Cadena	**Base 20**	**Pos Variables**
XXXXXXXXX	0.0.0.0.0.0.0.0.0	[0][1][2][3][4][5][6][7][8]
RAAYXXACDXXGV XTRXLPXRAXDX	14.0.0.19.0.0.0.1.2.0.0.5.17 0.16.14.0.9.12.0.14.0.0.2.0	[0][2][5][8][11][14][15][19][20]

Los aminoácidos variables, representados con la letra 'X' en la Tabla 7 tomarán el valor de cada uno de los 20 aminoácidos esenciales al momento de hacer el análisis exhaustivo de la cadena propuesta. La sustitución de X se hace de acuerdo a la Tabla 6, comenzando con el aminoácido *Alanina* que se representa con letra 'A' en código de una letra el cual tiene un valor de 0; por esta razón 'X' es igual a 'A', es decir ambos son 0 al inicio del análisis en la columna "Base 20" de la Tabla 7.

4.3.2. Preparar *Kernel*

La función *prepararKernel* ejecuta el trabajo principal del lado del huésped. Su función es ejecutar el *kernel* con los parámetros necesarios para las operaciones. Estos se almacenan en la memoria global del dispositivo, es necesario reservar memoria para cada uno de ellos. Además, es dentro de esta función donde se configura el número de hilos y bloques que

se utilizarán en la llamada al *kernel* dependiendo del número de aminoácidos variables que contenga la secuencia. La Tabla 8 muestra la configuración para analizar secuencias que van desde 3 hasta 7 aminoácidos variables.

Tabla 8: Configuración del *kernel* para analizar secuencias que van desde 3 hasta 7 aminoácidos variables.

AA variables	**Configuración *Kernel***	**Secuencias Analizadas**
3	kernel<<<8, 1024>>>	8000
4	kernel<<<157, 1024>>>	160000
5	kernel<<<3125, 1024>>>	3200000
6	kernel<<<62500, 1024>>>	64000000
7	kernel<<<1250000, 1024>>>	1280000000

En la primer columna de la Tabla 8 se muestra el número de aminoácidos variables que contiene la secuencia a evaluar, en la segunda columna, "Configuración *Kernel*", se establece el número de bloques y de hilos distribuidos uniformemente que se utilizarán para analizar el total de combinaciones creadas a partir de los aminoácidos variables; por ejemplo, una secuencia con 3 aminoácidos variables tiene 20^3 diferentes combinaciones, es decir, 8000 como se muestra en la tercer columna de la misma tabla, por lo que se necesitarán 8 bloques con 1024 hilos para cubrir el total de los casos (recordar que 1024 es el número máximo de hilos por bloque en la arquitectura *Kepler* utilizada para esta investigación).

En el Algoritmo 2 se definen los pasos de esta función. Para reservar memoria en el dispositivo se utiliza la función *CUDAMalloc* en los pasos 3 al 6, la copia de los valores del huésped hacia el dispositivo se hace con la función *CUDAMemcopy* en los pasos 7 y 8. Es necesario inicializar la variable $d_total_encontrados = 0$, esto se logra con la función CUDAMemset del paso 9. Con estos parámetros más la configuración de hilos y bloques se manda ejecutar el *kernel* en el paso 10. Al terminar el proceso paralelo de detección de SCAPs se sincroniza el dispositivo para asegurar que se finalizaron todas las tareas dentro del GPU. En el paso 12 se copian los resultados positivos que se almacenaron en el arreglo del dispositivo d_SCAPs hacia el arreglo del huésped h_SCAPs. Este arreglo contiene los resultados que se muestran al usuario.

Algoritmo 2 Función prepararKernel

Entrada: El arreglo h_SCAPs donde se almacenan los casos positivos que arroja el dispositivo, h_seq20 con la secuencia a analizar en formato base 20, el arreglo h_posVar con la posición de los aminoácidos variables y el número requerido de $iteraciones$ en el dispositivo.

Salida: El arreglo h_SCAPs con los casos positivos.

1: declarar d_SCAPs, $d_total_encontrados$, d_seq20, d_posVar
2: calclular el número de $BLOQUE$ e $HILOS$
{Reservar memoria para las variables que se utilizan en el dispositivo}
3: CUDAMalloc(d_SCAPs, tamaño = MAX_CAPACIDAD_GPU)
4: CUDAMalloc(d_seq20, tamaño = longitud de la secuencia)
5: CUDAMalloc(d_posVar, tamaño = número de aminoácidos variables)
6: CUDAMalloc($d_total_encontrados$, tamaño = entero)
{Copiar valores del huésped al dispositivo}
7: CUDAMemcopy(d_seq20, h_seq20, tamaño = longitud de la secuencia)
8: CUDAMemcopy(d_posVar, h_posVar, tamaño = número de aminoácidos variables)
{Iniciar $d_total_encontrados = 0$ en el dispositivo}
9: CUDAMemset($d_total_encontrados$, 0, sizeof(int))
{Llamar al $Kernel$}
10: SCAPsKernel$\lll$ $BLOQuES$, $HILOS$ $\ggg$(d_SCAPs, d_seq20, d_posVar, $d_total_encontrados$)
{Sincronizar el dispositivo}
11: CUDAStatus = CUDADeviceSynchronize()
{Copiar el arreglo d_SCAPs del dispositivo al arreglo del huésped h_SCAPs}
12: CUDAMemcpy(h_SCAPs, d_SCAPs, MAX_CAPACIDAD_GPU, CUDAMemcpy-dispositivoToHost)
13: **devolver** h_SCAPs

En este proceso se ejecutan todas las operaciones que tienen relación con CUDA, por esta razón la función se diseñó separada del proceso principal o “main”. Esto sirve para mantener el código de CUDA en un módulo separado del código estándar de C++. Esto hace que el mantenimiento del código sea más fácil para futuras modificaciones.

4.3.3. Función *Kernel*

Esta función es el punto de entrada en el dispositivo, es el algoritmo principal de la paralelización. La estrategia principal es analizar un péptido por hilo y almacenar únicamente aquellos péptidos que resulten ser SCAPs, para optimizar así el uso de la memoria global del GPU.

Al igual que otros enfoques en la literatura, se busca disminuir el tiempo de ejecución del algoritmo al encontrar una buena estrategia de paralelización. En este diseño además se pone particular interés en la optimización del espacio de memoria requerido. Esto implica menos llamadas al GPU, que al final se traduce en una mejora en el tiempo de ejecución total, ya que uno de los procesos más costosos en cuanto a tiempo se refiere es la comunicación CPU-GPU.

Para optimizar la memoria del GPU se almacenan solo SCAPs, desechando los resultados negativos que representan la gran mayoría de los casos. Aunque esta consideración implica un conjunto adicional de instrucciones, la optimización es notoria cuando el proceso que hace la llamada al GPU no necesita ejecutar operaciones para filtrar el arreglo de resultados que recibe.

Ya que no todos los hilos contendrán una secuencia de aminoácidos que sea SCAP, es necesario calcular el índice dentro de la memoria global donde se almacenarán los casos positivos, solo que, al estar en un ambiente de paralelización, este proceso es una tarea no trivial, pues no existe un proceso directo, como en el caso de un algoritmo secuencial, que determine dicho índice para cada caso positivo.

Para calcular el índice se utiliza la técnica conocida como “suma de prefijos” (JaJa, 1992b) que se describe en la sección 4.4.1 y la instrucción “atomicAdd” que proveé el SDK de CUDA.

Un aspecto a considerar es que, hacer una sola suma de prefijos de un número muy grande de elementos sería ineficaz, así que para aprovechar al máximo la arquitectura que provee NVidia, la suma de prefijos se divide en secciones, es decir, en bloques de hilos. Al terminar el cálculo del índice por bloque de cada caso positivo, estos se almacena en la memoria global. El pseudocódigo de esta función se muestra en el Algoritmo 3.

Algoritmo 3 Kernel, detección y almacenamieto de SCAPs.

Entrada: El arreglo d_SCAPs donde se almacenan los casos positivos, d_seq20 con la secuencia a analizar en formato base 20, el arreglo d_posVar con las posiciones de los aminoácidos variables, el contador de casos positivos $total_positivos$ y la variable $iteracion$ que indica el $i-$ésimo llamado al dispositivo.

Salida: El arreglo d_SCAPs con los casos positivos.

1: Id_hilo = blockDim.x × blockIdx.x + threadIdx.x
2: Id_hilo += $iteracion$ * 20^7
3: $peptido_local$ = validarSCAP(Id_hilo, d_seq20, d_posVar)
4: Sincronizar hilos por bloque
5: $indice_SCAP$ = SumaDePrefijos($peptido_local$, Id_hilo, threadIdx.x, max_SCAPs_bloque)
6: **si** threadIdx.x == blockDim.x - 1 **entonces**
7: $indice_bloque$ = atomicAdd($total_positivos$, max_SCAPs_bloque)
8: **fin si**
9: Sincronizar hilos por bloque
10: **si** $peptido_local == SCAP$ **entonces**
11: $d_SCAPS[indice_bloque + indice_SCAP] = peptido_local$
12: **fin si**

En el paso 1 se calcula el identificador global del hilo Id_hilo. Con este identificador se genera la secuencia que se analizará, la conversión de Id_hilo a secuencia se hace dentro de la función *validarSCAP*.

El paso 2 es útil para casos donde el problema requiere ejecutar el *kernel* n número de veces, es decir casos con más de 7 aminoácidos variables. En estos casos la solución del problema se divide en bloques de 20^7 secuencias. Cada bloque se identifica por medio de la variable $iteracion$. El índice del hilo en estos casos está en relación con el identificador del paso 1, el cual se multiplica por el bloque al que pertenece.

En el paso 3 la función *validarSCAP* determina si la secuencia que se analizó es SCAP o no mediante el cálculo de las cuatro propiedades propuestas. Dentro de esta función se genera la secuencia a evaluar con base en el identificador del hilo que recibe. El proceso completo de esta función se describe en el Algoritmo 4.

La instrucción "Sincronizar hilos por bloque" del paso 4 coordina la comunicación entre los hilos del mismo bloque, con esto se previenen problemas de lectura a memoria compartida en futuras operaciones.

A partir del paso 5 el diseño del algoritmo contempla operaciones por bloque, donde los hilos de un bloque n se coordinan para realizar las instrucciones indicadas.

La técnica de paralelización "suma de prefijos" del paso 5 se utiliza para calcular el índice que le corresponde a un hilo n dentro de un bloque m cuando el hilo contiene un SCAP con la intención de agruparlos en localidades contiguas de memoria, con esta operación se eliminan los hilos que no contienen un SCAP. Con la suma de prefijos se obtiene además el máximo número de SCAPs dentro del bloque m, este valor se guarda dentro de la variable max_SCAPs_bloque. Esta función se explica a detalle en la Sección 4.4.1.

La condición "if" del paso 6 indica que solo el último hilo de cada bloque ejecutará la instrucción del paso 7. En el paso 7 se obtiene el índice que le corresponde a cada bloque para escribir los SCAPs en la memoria global. Como solo se necesita un índice por bloque, un solo hilo dentro de este es suficiente para obtenerlo. La operación *atomicAdd* con la que se calcula el índice para cada bloque se explica en la Sección 4.4.2.

En el paso 9 de nuevo se hace una sincronización de hilos.

Finalmente, en el paso 10 se pregunta si el hilo contiene un SCAP, de ser así, en el paso 11 se almacena en el arreglo global d_SCAPs. El índice donde se almacena el SCAP s se obtiene de sumar los índices que se calcularon en los pasos 5 y 7, es decir, índice a partir del cual el bloque m escribirá SCAPs (*indice_bloque*) más el índice que le corresponde al hilo con el SCAP s dentro del bloque m ($indice_SCAP$).

4.3.4. Algoritmo para detectar SCAPs

Este algoritmo determina si la cadena de aminoácidos a es SCAP o no. El resultado se obtiene con el cálculo de las cuatro propiedades fisicoquímicas propuestas por Polanco *et al.* (2011). La cadena a se construye principalmente con índice del hilo que este algoritmo recibe como parámetro.

Primero se declara la varible *peptido_local* de tipo *SCAP_Estructura_ Completa* (ver Sección 4.2.2) donde se almacenan los valores de las cuatro propiedades.

Algoritmo 4 Detección de SCAPs.

Entrada: El índice del hilo ($hilo_Id$), la secuencia base que se desea analizar con los aminoácidos en base 20 (d_seq20) y el arreglo con la posición de los aminoácidos variables (d_posVar).

Salida: La variable $peptido_local$ del tipo $SCAP_Estructura_Completa$.

1: declarar $peptido_local$ de tipo $SCAP_Estructura_Completa$
2: $secuencia$ = GenerarSecuencia20($hilo_Id$, d_seq20, d_posVar)
3: $conteo_aa$ = Contar los aminoácidos con carga en la $secuencia$
4: $peptido_local.MH$ = Calcular el Momento Hidrofóbico de $secuencia$
5: $peptido_local.MC$ = Calcular la Media de Carga Neta con $conteo_aa$
6: **si** $peptido_local.MH$ y $peptido_local.MC$ están dentro de los intervalos de un SCAP **entonces**
7: $peptido_local.PI$ = Calcular el Punto Isoléctrico con $conteo_aa$
8: **si** $peptido_local.PI$ está dentro de los intervalos de un SCAP **entonces**
9: $peptido_local.MHH$ = Calcular Momento Hidrofóbico Helicoidal de $secuencia$
10: **si** $peptido_local.MHH$ está dentro del rango de un SCAP **entonces**
11: Marcar al péptido como SCAP
12: **fin si**
13: **fin si**
14: **fin si**
15: **devolver** $peptido_local$

En el paso 2 se construye la secuencia que se analizará, la variable $hilo_Id$ es el parámetro principal en este proceso. La función *GenerarSecuencia20* se encarga de construir la secuencia en base 20 con los parámetros que recibe; esta función se explica en la Sección 4.3.5.

En el paso 3 se hace un conteo de los aminoácidos con carga positiva y negativa en la secuencia, además de aquellos que contienen un valor asociado de pK_a (Sección 4.3.6). Los aminoácidos con carga positiva son arginina y cisteína. Para calcular la carga negativa se cuentan los ácidos glutámico y aspártico. Con valores de pK_a el conteo es de cisteinas, histidias y tirosinas. Este conteo es necesario para los cálculo posteriores de carga neta y punto isoeléctrico.

El cálculo de las propiedades se realiza de manera secuencial dentro de cada hilo, es por esto que se calculan primero las propiedades que toman menor tiempo en ejecutarse, como son los casos del Momento Hidrofóbico y la Media de Carga Neta de los pasos 4 y 5, respectivamente. Si los valores de estas dos propiedades están dentro de los intervalos que definen un SCAP, entonces se calcula el punto isoeléctrico en el paso 7 y finalmente el momento hidrofóbico helicoidal en el paso 9. Si estos dos últimos valores se encuentran dentro de los intervalos definidos por un SCAP, entonces la secuencia se marca como SCAP en el

paso 11.

Es importante notar que si un hilo diverge, todos los hilos de ese bloque deben esperar a que ese hilo termine, lo cual elimina la "optimización" de ejecutar las funciones de menor peso en cuanto al tiempo, pero la estrategia es efectiva cuando todos los hilos de un bloque ejecutan las mismas instrucciones, el ejemplo óptimo ocurre cuando ningún hilo dentro de un bloque cumple con la condición del paso 6.

Al final, el algoritmo retorna la variable $peptido_local$ con los valores de las cuatro propiedades más una bandera que identifica a la secuencia como un caso positivo o negativo de un SCAP.

4.3.5. Algoritmo de conversión base 20

Con este algoritmo se genera la secuencia s de aminoácidos que se analizará dentro del hilo h. La secuencia se construye con el identificador del hilo más la secuencia original o que se introduce desde el huésped, la secuencia o tiene la combinación de aminoácidos fijos y variables. Para la construcción de s es necesario también el arreglo con la posición de los aminoácidos variables.

Algoritmo 5 Convertidor base 20.

Entrada: El índice del hilo ($hilo_Id$), la secuencia base que se se desea analizar con los aminoácidos en base 20 (d_seq20) y el arreglo con la posición de los aminoácidos variables (pos_Var).

Salida: El arreglo "pos" con cada dígito del número en base 20

1: Clonar $temp_seq20 = d_seq20$
2: $cociente = hilo_Id$
3: $i = 0$;
4: **mientras** $cociente >= 1$ **hacer**
5: $temp_seq20[pos_Var[i]] = cociente$ Mod 20
6: $cociente = cociente$ / 20
7: $i++$
8: **fin mientras**
9: **devolver** $temp_seq20$

El primer paso consiste en clonar el arreglo que contiene la secuencia original d_seq20. El arreglo con la secuencia original está almacenado en la memoria global, modificarlo dentro de un hilo significaría modificarlo permanentemente, por lo cual hilos subsecuentes tendrían información errónea.

En el paso 2 el valor del hilo contenido en $hilo_Id$ se pasa a la variable $cociente$.

El ciclo **mientras** se encarga de construir la secuencia utilizando como plantilla al arreglo clon $temp_seq20$. En cada iteración se genera un dígito en base 20 mediante la operación módulo sobre la variable $cociente$. El índice i del ciclo determina el aminoácido variable que se reemplazará. El remplazo se hace con ayuda del arreglo pos_Var que contiene la posición de cada aminoácido variable dentro de la secuencia.

Finalmente, se retorna el arreglo con la nueva secuencia. El hilo que invocó esta función analizará la secuencia para determinar si es SCAP o no.

Ejemplo: Sea d_seq20 = "RXYXLXDXX" la secuencia a evaluar, las posiciones de los aminoácidos variables son, de derecha a izquierda, $pos_Var = [0, 1, 3, 5, 7]$. Sea el identificador del hilo $hilo_Id = 1000$; la representación en base 20 de d_seq20 es "14.0.19.0.9.0.2.0.0" y de $hilo_Id$ es "2.10.0" lo que equivale a los aminoácidos D.M.A. Estos tres aminoácidos reemplazan a los aminoácidos variables ubicados en las tres primeras posiciones del arreglo pos_Var ([0, 1, 3]). El resto de los aminoácidos variables ([5,7]) se sustituyen por A. El péptido resultante es: "RAYALDDM". La Tabla 9 muestra cinco ejemplos con la misma cadena de estudio "RXYXLXDXX".

Tabla 9: Relación entre el identificador del hilo y la cadena de aminoácidos que genera, tomando como base la secuencia de estudio.

Cadena de estudio: RXYXLXDXX				
Id hilo	**No. base 20**	**Aminoácido(s)**	**Arreglo interno (base 20)**	**Cadena resultante**
0	0	A	14.0.19.0.9.0.2.0.0	RAYALADAA
1	1	C	14.0.19.0.9.0.2.0.1	RAYALADAC
100	5.0	G.A	14.0.19.0.9.0.2.5.0	RAYALADGA
1000	2.10.0	D.M.A	14.0.19.0.9.2.2.10.0	RAYALDDMA
10000	1.5.0.0	C.G.A.A	14.0.19.1.9.5.2.0.0	RAYCLGDAA

4.3.6. Cálculo de cuatro propiedades fisicoquímicas

En esta sección se revisan los algoritmos referentes al cálculo de las cuatros propiedades fisicoquímicas; la definición formal de cada propiedad se encuentra en la Sección 2.7

Media de carga neta

El valor de la carga neta es igual a la suma del número de aminoácidos con carga positiva de la secuencia, arginina (Arg) y lisina (Lys), menos la suma de los aminoácidos con carga

negativa, ácido aspártico (Asp) y ácido glutámico (Glu); el resultado de esta operación se divide entre el total de aminoácidos que componen la secuencia (Polanco *et al.*, 2011).

Algoritmo 6 Media de Carga Neta.

Entrada: El arreglo $totalAA$ con el conteo por aminoácido del pépetido a analizar, la variable n con el total de los aminoácidos.
Salida: El valor de carga del péptido
1: **devolver** (($totalAA$[Arg] + $totalAA$[Lys]) - ($totalAA$[Glu] + $totalAA$[Asp]))/ n

Media de hidrofobicidad

El cálculo de la media de hidrofobicidad presentada por Polanco *et al.* (2011) se realiza por medio de *ventanas*, cada una de ellas contiene cinco aminoácidos; el valor por *ventana* se define como: $V_i = [a_i + a_{i+1} + \ldots + a_{i+4}]/5$, para $i = 1$ hasta n, donde a_i representa a la hidrofobicidad del i-ésimo aminoácido y V_i es la i-ésima *ventana*; es decir, cada *ventana* tiene el valor promedio de hidrofobicidad de los elementos que la componen.

En el ciclo **para** del paso 2 del Algoritmo 7 se realiza una sumatoria de ventanas; el valor de cada ventana se calcula en el paso 4; este valor se acumula en la variable mH en el paso 5; terminada la operación se incrementa el contador *ventanas* en el paso 6.

La media de hidrofobicidad es el valor de la sumatoria mH entre el número total de ventanas creadas, como se indica en el paso 8.

Algoritmo 7 Media de Hidrofobicidad.

Entrada: El péptido, la variable n con el total de los aminoácidos, el tamaño de la ventana en la variable *tamano_ventana*
Salida: El valor de la media de hidrofobicidad
1: *ventanas* = 0;
2: **para** $i \leftarrow 1$ hasta n - *tamano_ventana* **hacer**
3: *valor_ventana* = 0
4: *valor_ventana* = sumar la carga del aminoácido i hasta i+5
5: *mH* = *mH* + *valor_ventana* / *tamano_ventana*
6: *ventanas*++
7: **fin para**
8: **devolver** *mH* / *ventanas*;

Punto isoeléctrico

El Punto Isoeléctrico (PI) es el valor de pH en el cual una molécula en particular carece de carga eléctrica. La fórmula para calcularlo es (Polanco *et al.*, 2011):

$$Z = \sum_i N_i \frac{1}{1 + 10^{(pH - pK_{a_i})}} - \sum_j N_j \frac{1}{1 + 10^{(pK_{a_j} - pH)}} \tag{7}$$

En el Algoritmo 8 se realiza el cálculo del punto isoeléctrico con el método de Bisección (Weisstein, 2010). El proceso comienza evaluando la ecuación con el valor de pH neutro (6.5). Si la ecuación arroja un valor negativo (menor a cero) significa que el pH neutro es alto, entonces el intervalo de búsqueda se acota por valores menores a este y viceversa cuando el valor de la ecuación es positivo. El algoritmo termina cuando la diferencia entre la cota superior e inferior en la búsqueda del pH es menor a 0.1, lo que da como resultado el punto isoeléctrico.

Algoritmo 8 Punto Isoeléctrico.

Entrada: El arreglo $totalAA$ que contiene el conteo por aminoácido dentro del pépetido a analizar.
Salida: El valor del punto isoeléctrico.

```
1:  pHlow = 0.0
2:  pHup = 14.0
3:  pH = 6.5
4:  mientras pHup - PHlow > 0.1 hacer
5:     EQ = Ejecutar la ecuación del PI con el valor de pH actual
6:     si EQ < 0 entonces
7:        pHup = pH
8:        pH = 0.5 * (pH + pHlow)
9:     si no
10:       pHlow = pH
11:       pH = 0.5 * (pH + pHup)
12:    fin si
13: fin mientras
14: devolver valor de pH
```

En los pasos 1, 2, y 3 se establecen las cotas inferior ($pHlow$) y superior ($pHup$), y el valor de pH neutro.

El ciclo del paso 4 se ejecuta mientras exista diferencia entre el $pHup$ y $pHlow$, es decir, mientras no se encuentre el PI.

En el paso 5 se ejecuta la ecuación del PI.

Si el valor de la ecuación del paso 5 es menor a cero indica que el pH es muy alto, por lo que en el paso 7 se actualiza la cota superior ($pHup$) y en el paso 8 se "reduce" el valor de pH a la mitad. Es necesario tomar en cuenta el valor de la cota inferior ($pHlow$) de pH para hacer la reducción.

Los pasos 9 y 10 hacen el mismo proceso del paso anterior pero de manera inversa, es decir, se actualiza la cota inferior y se actualiza el valor de pH con base en la cota superior.

Para el cálculo del PI de Polanco *et al.* (2011) se utiliza un algoritmo secuencial. En dicha implementación el PI se calcula con incrementos de 0.01 en el valor de pH por cada iteración; el proceso termina cuando la ecuación arroja un valor $<= 0$. Para acelerar este cáclulo se cambió el algoritmo secuencial por el de bisección aquí descrito (Algoritmo 8).

Momento Hidrofóbico Helicoidal

El momento hidrofóbico helicoidal se calcula con la Ecuación 6 que se encuentra en al Sección 2.7. El objetivo es predecir la posibilidad de que la secuencia primaria forme una α-hélice, propiedad que caracteriza a los SCAPs (del Rio *et al.*, 2001; Hilpert *et al.*, 2008). Para formar una α-hélice, se espera que las cadenas laterales de cada aminoácido se encuentren entre 95° y 105° con respecto a la columna vertebral del péptido (Eisenberg *et al.*, 1984), es decir, el segmento periódico de la estructura contiene 3.6 residuos por cada giro.

El péptido a evaluar se divide en secciones o "ventanas", cada ventana contiene siete aminoácidos. En cada ventana se hace el cálculo del momento hidrofóbico helicoidal (Ecuación 6). Las ventanas se definen de la forma $V_i = [a_i + a_{i+1} + \ldots + a_{i+7}]$.

El Algoritmo 9 presenta los pasos para el cálculo del momento hidrofóbico helicoidal. Este recibe como entrada la tabla con los valores de hidrofobicidad de cada aminoácido (3), el ángulo inicial y final a evaluar, 95° y 105° respectivamente para una α-hélice, el último parámetro indica el tamaño de las ventanas que se utilizarán para el cálculo del momento hidrofóbico.

El primer ciclo del Algoritmo (pasos 1 al 9), prepara la primer ventana para el cálculo

Algoritmo 9 Algoritmo para calcular el Momento Hidrofóbico Helicoidal.

Entrada: La secuencia a evaluar que se almacena en *peptido*, el tamaño de la *ventana*, la región a evaluar delimitada por *angulo_Inicial* y *angulo_Final* y la tabla con los valores de hidrofobicidad de cada aminoácido contenidos en el arreglo *TablaH*.

Salida: El valor del momento hidrofóbico helicoidal del péptido.

```
1:  para i ← 0 hasta ventana − 1 hacer
2:     hidrofobicidad = TablaH[peptido[i]]
3:     sumHidro += hidrofobicidad
4:     para angPos ← 0 hasta angulo_Final − angulo_Inicial hacer
5:        angulo = angulo_Inicial * i
6:        sumSin[angPos] += seno[angulo] * hidrofobicidad
7:        sumCos[angPos] += coseno[angulo] * hidrofobicidad
8:     fin para
9:  fin para
10: para i ← ventana hasta longitud[peptido] hacer
11:    hidrofobicidad = TablaH[peptido[i]]
12:    sumHidro += hidrofobicidad
13:    promedioHidro += sumHidro/ventana
14:    momento_MAX[i] = 0;
15:    para angPos ← 0 hasta angulo_Final − angulo_Inicial hacer
16:       angulo = angulo_Inicial * i
17:       sumSin[angPos] += seno[angulo] * hidrofobicidad
18:       sumCos[angPos] += coseno[angulo] * hidrofobicidad
19:       momento = Ecuacion_Momento_Hidrofobico(sumSin, sumCos, promedioHidro)
20:       momento = momento / ventana
21:       si momento >= momento_MAX[i] entonces
22:          momento_MAX[i] = momento
23:          angulo_MAX[i] = angulo_Inicial + angPos
24:       fin si
25:    fin para
26:    hidrofobicidad = TablaH[peptido[i − window]]
27:    sumHidro -= hidrofobicidad
28:    para angPos ← 0 hasta angulo_Final − angulo_Inicial hacer
29:       angulo = angulo_Inicial * i
30:       sumSin[angPos] -= seno[angulo] * hidrofobicidad
31:       sumCos[angPos] -= coseno[angulo] * hidrofobicidad
32:    fin para
33: fin para
34: devolver  Promedio de Momento_MAX
```

del momento hidrofóbico. Esta primer ventana abarca desde el aminoácido en la posición ‘0’ hasta el tamaño de *ventana* - 1.

En el paso 2 se obtiene la hidrofobicidad del i-ésimo aminoácido y en el paso 3 se realiza la sumatoria con la hidrofobicidad de cada aminoácido.

El ciclo de los pasos 4 al 8 recorre el número de ángulos a evaluar, 10 en este caso (del 95° al 105°). Dentro de este se realiza una sumatoria con el seno y coseno por cada ángulo multiplicado por la hidrofobicidad del i-ésimo aminoácido. Los resultados se almacenan en los arreglos $sumSin$ y $sumCos$.

En el ciclo del paso 10 se agrega un nuevo elemento para completar la ventana. En el paso 11 se toma la hidrofobicidad del nuevo aminoácido que se añade, en el paso 11 se agrega este valor a la sumatoria de hidrofobicidad previamente iniciada y en el paso 13 se calcula el promedio de hidrofobicidad entre los elementos de la ventana.

Al inicio del ciclo de los pasos 14 al 22 se agregan a las sumatorias $sumSin$ y $sumCos$ el valor de cada ángulo del elemento recién agregado a la ventana, pasos 16 y 17. Para cada ángulo del ciclo se realiza el cálculo del momento hidrofóbico en el paso 18, este recibe las sumatorias $sumSin$ y $sumCos$ así como el promedio de hidrofobicidad de la ventana. El resultado se divide entre el número de elementos de la ventana en los pasos y se almacena en $momento$ en el paso 19.

Si $momento$ del ángulo evaluado es el máximo encontrado para del i-ésimo aminoácido, este se almacena en el arreglo $momento_MAX$ en la posición reservada para dicho aminoácido en los pasos 20 y 21.

El ciclo de los pasos 26 al 30 se encarga de remover el primer elemento de la ventana.

El momento hidrofóbico de la cadena se obtiene del valor promedio de los elementos almacenados en $momento_Max$, como se indica en el paso 33.

4.4. Estrategias para optimizar la escritura de SCAPs en la memoria global del dispositivo

Además de la estrategia para la detección de SCAPs en paralelo, es necesario considerar el método de escritura de los resultados en la memoria global. Se deben almacenar solo los casos positivos y desechar los negativos aún cuando estos se encuentran intercalados. El procedimiento para calcular los índices contiguos de cada caso positivo requiere de operaciones

que no son triviales cuando se trabaja en un proceso en paralelo, pero no considerar el filtrado de los casos positivos significaría faltar a una optimización obvia pues esto generaría un desperdicio de memoria global y tiempo de ejecución, principalmente al momento de pasar los datos del dispositivo hacia el huésped y cuando el huésped escriba los resultados hacia algún dispositivo de almacenamiento (presumiblemente disco duro), principalmente si el número de casos negativos es alto.

A continuación se describen las estrategias implementadas para la escritura de SCAPs dentro del dispositivo.

4.4.1. Suma de Prefijos

En esta sección se explica la técnica “suma de prefijos” que se utiliza en el paso 5 del Algoritmo 3 y los detalles de la implementación que utiliza instrucciones específicas de CUDA.

A continuación se presenta le definición de suma de prefijos (JaJa, 1992a). Consideremos una secuencia de n elementos $\{x_1, x_2, \ldots, x_n\}$ obtenidos de un conjunto S con una operación binaria asociativa, denotada por $\bigoplus$. La suma de prefijos de esta secuencia es igual a las sumas parciales (o productos) definidos por:

$$s_i = x_1 \bigoplus x_2 \bigoplus \ldots \bigoplus x_i, 1 \leq i \leq n.$$

Un algoritmo trivial secuencial calcula s_i con una sola operación mediante el uso de la identidad $s_i = s_{i-1} \bigoplus x_i$, para $2 \leq i \leq n$, y por lo tanto toma $O(n)$ de tiempo. Este algoritmo es inherentemente secuencial.

En la Figura 24 se muestra un ejemplo donde $\bigoplus$ representa la operación suma.

	0	1	2	3	4	5	6	7
A =	3	1	7	0	4	1	6	3
A′ =	3	4	11	11	15	16	22	25

Figura 24: El arreglo A muestra una secuencia de valores aleatorios. El arreglo A' muestra el resultado de la suma de prefijos sobre A.

En la teoría de algoritmos paralelos la herramienta utilizada para calcular la suma de prefijos es el árbol binario balanceado.

La suma de prefijos tiene diferentes usos como son la clasificación, análisis léxico, comparación de cadenas, evaluación de polinomios, compactación corriente e histogramas de construcción y estructuras de datos (gráficos, árboles, etc) en paralelo, entre otras (Harris, 2007).

En este problema se utiliza para optimizar la memoria requerida. El objetivo es ubicar los hilos que contienen un SCAP en posiciones contiguas de memoria y eliminar los casos negativos que se encuentran intercalados. La Figura 25 muestra un ejemplo de esta operación.

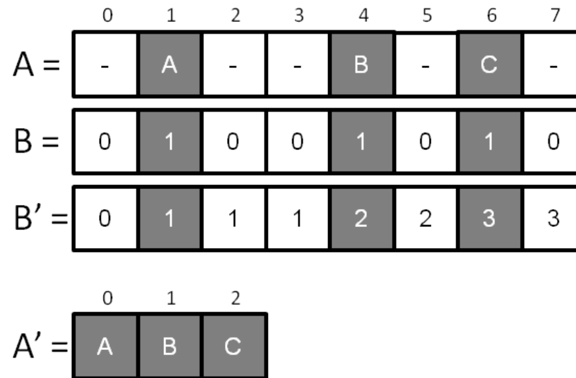

Figura 25: El arreglo *A* muestra el contenido relevante (descrito por las letras A, B y C) y contenido irrelevante (representado por '-'). En el arreglo *B* se cambia la información relevante por '1' y la irrelevante por '0'. En *B'* se muestra la suma de prefijos. La posición de la información relevante en *A'* se obtiene de las casillas donde ocurre un incremento de valor (casillas coloreadas) en *B'*.

Para resolver la suma de prefijos, CUDA proveé un conjunto de funciones llamadas "*shfl*" (en abreviación a la palabra *shuffle* o barajeo en español) que optimizan el cálculo bajo esta arquitectura (ver *shfl* en la Sección 3.9.2).

La unidad básica con que trabaja este algoritmo es el warp. Cada *warp* contiene 32 hilos, cada hilo tiene un valor numérico de 1 si contiene un SCAP o 0 de lo contrario. Un ejemplo de un *warp* se muestra en la Figura 26.

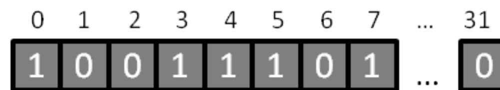

Figura 26: Diagrama que ilustra el concepto de *warp*.

En los pasos 1 al 4 (Algoritmo 10) se declaran cuatro variables necesarias para la expli-

Algoritmo 10 Suma de Prefijos.

Entrada: La variable *data* indica si el hilo contiene o no un SCAP, el *id* global del hilo, el índice del hilo dentro del bloque en la variable *threadIdx*, el ancho del *warp* contenido en *WarpSize*, una variable donde guardar el máximo elemento del bloque.

Salida: El valor de la suma de prefijos.

```
1: shared int sums[32]
2: laneId = id MOD WarpSize
3: warpId = threadIdx / WarpSize
4: value = data
5: para i ← 1 hasta width hacer
6:    n = shfl_up(value, i, width)
7:    si laneId >= i entonces
8:       value + = n
9:    fin si
10: fin para
11: si threadIdx % warpSize == warpSize - 1 entonces
12:    sums[warpId] = value
13: fin si
14: syncthreads()
15: si warpId == 0 entonces
16:    warpSum = sums[laneId]
17:    para i ← 1 hasta width hacer
18:       n = shfl_ up(warpSum, i, width)
19:       si laneId >= i entonces
20:          warpSum += n
21:       fin si
22:    fin para
23:    sums[laneId] = warpSum
24: fin si
25: syncthreads()
26: blockSum = 0
27: si warpId > 0 entonces
28:    blockSum = sums[warpId - 1]
29: fin si
30: value + = blockSum
31: si threadIdx == blockDim.x-1 entonces
32:    max_block_element = value
33: fin si
34: devolver value
```

cación del algoritmo:

1. El arreglo *sums* se utiliza para calcular la suma de prefijos por bloque, este contendrá el valor máximo de cada *warp*. El tamaño del arreglo se calcula con el número de hilos por bloque entre el tamaño del *warp*, en este caso: $1024/32 = 32$.

2. La variable laneId lleva el registro de la línea a la que pertence cada hilo, como se trabaja con *warps* existen 32 líneas.

3. La variable “warpId” maneja el índice del *warp* al que pertenece cada hilo, hay que recordar que se trabaja con múltiples hilos y *warps* de manera simultánea.

4. La variable *value* almacena el dato binario principal, 1 significa que el hilo contiene un SCAP y 0 lo contrario.

La suma de prefijos se ejecuta por *warp* en los pasos 5 al 10 con ayuda de la instrucción *shfl_up*, la suma de prefijos se ejecuta sobre múltiples *warps* de manera simultánea. La instrucción *shfl_up* mueve los datos de los hilos dentro de un *warp* hacia hilos que se encuentran en posiciones posteriores. En cada iteración los hilos de un *warp* mueven y suman el valor que contienen hacia los hilos que están a su derecha de manera simultánea y en potencias de dos, esto es, primero pasan su valor una posción, después dos, después cuatro hasta llegar al valor de la variable *width* (Figura 27). El índice i del ciclo del paso 5 indica cuántas posiciones a la derecha se debe mover el dato de un hilo en la instrucción shfl_up, el índice además dicta si la línea ($laneId$) a la que pertenece el hilo debe sumar el valor que se ha desplazado, a esto se le llama línea de corte, la cual divide el warp entre los que solo van a pasar su valor y los que lo pasan y además suman el valor que reciben.

Al terminar la suma de prefijos por *warp*, el total de cada uno se almacena dentro del arreglo local *sums*, cada arreglo *sums* alberga la suma total de 32 *warps* para representar a un bloque de 1024 hilos (Figura 29). La suma total de cada *warp* lo tiene el hilo en la última posición de este (casilla 31). Esta operación se hace en los pasos 11 y 12.

En los pasos 15 al 24 se efectúa el mismo procedimiento de suma de prefijos pero ahora sobre el arreglo *sums* que representa a un bloque entero. Múltiples bloques son atacados de manera simultánea. El arreglo *sums* es de memoria compartida pues 32 hilos operan sobre este de manera simultánea.

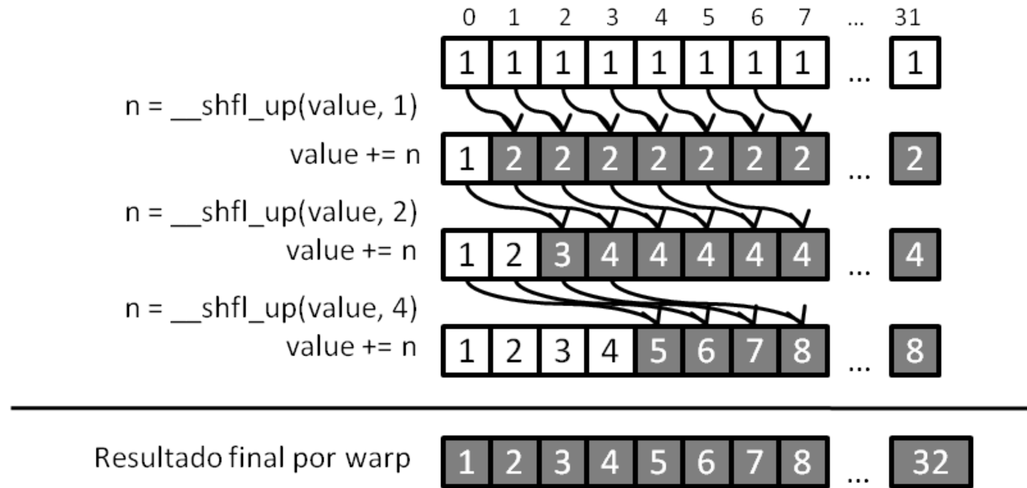

Figura 27: Suma de prefijos por *warp*.

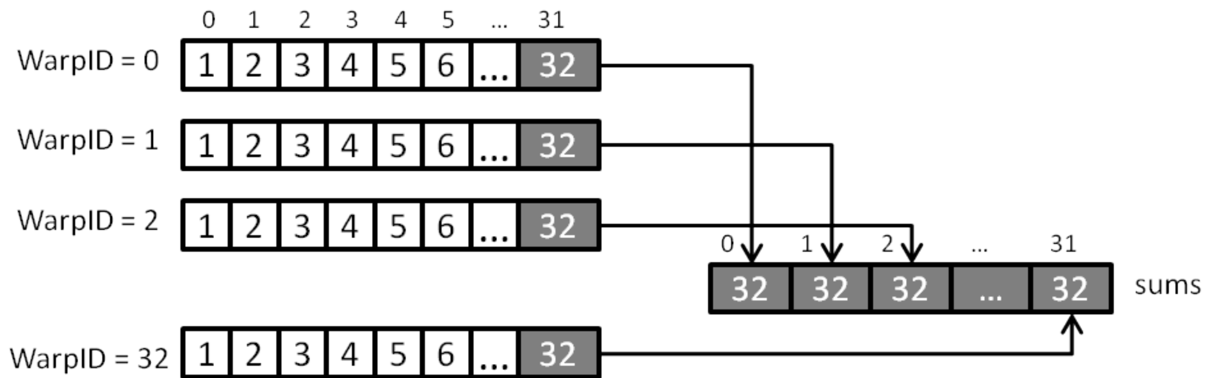

Figura 28: Llenado del arreglo *sums*.

Como el arreglo *sums* almacena los valores de 32 *warps*, es suficiente con los hilos de un solo *warp* para hacer los cálculos, en este caso se eligieron a los hilos que pertenecen al *warp* con índice 0 (condición del paso 15). El resultado de la suma se almacena dentro del mismo arreglo *sums* (29).

En el paso 27 se obtiene la suma total de *warp* anterior al *warp* al que pertenece el hilo actual. Este valor se utiliza en el paso 30 para cálculo de la suma de prefijos de un hilo en específico. Para calcularlo se suma el valor del *warp* anterior al que pertenece el hilo actual (que se obtuvo en el paso 27) más el valor que tiene el hilo actual dentro de su propio *warp*.

En la Figura 30, se muestra un ejemplo donde se obtiene la suma de prefijos de un hilo con la ayuda del valor máximo del *warp* anterior a donde se encuentra este.

Finalmente, en la condición del paso 31, el último hilo de cada bloque almacena su valor en la variable *max_block_element*, este representa la suma total del bloque, este valor se

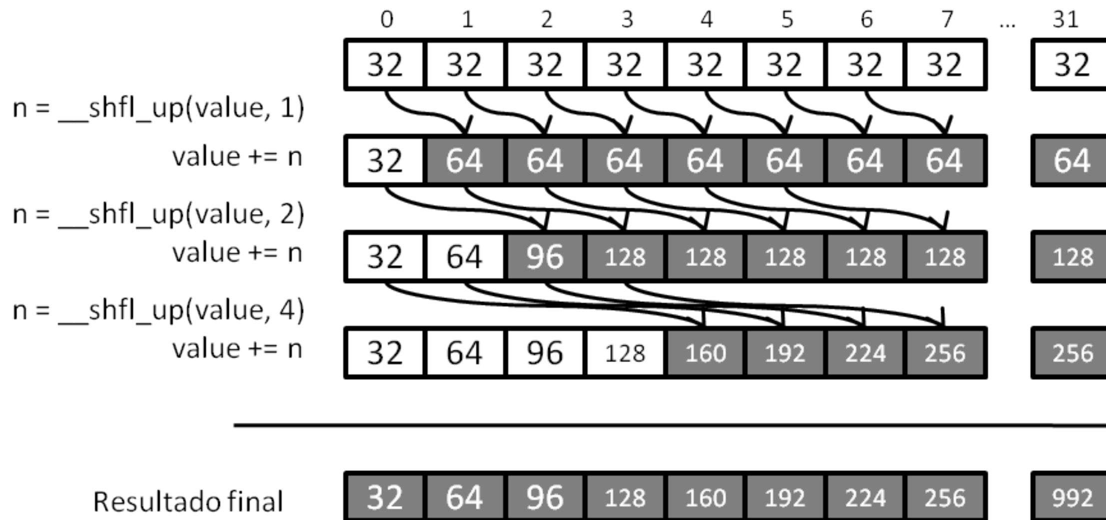

Figura 29: Suma de de prefijos del bloque.

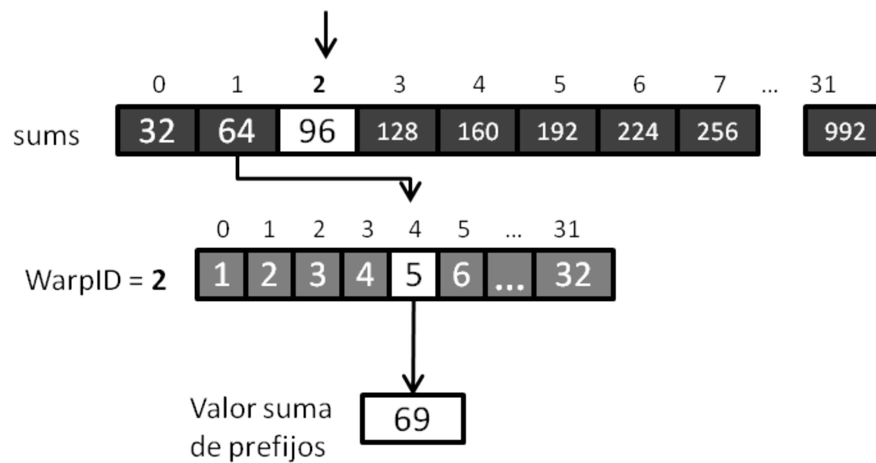

Figura 30: Valor de la suma de prefijos del hilo con Id = 4 dentro del *warp* con Id = 2 del arreglo *sums*.

utiliza posteriormente para reservar la memoria global necesaria para almacenar los SCAPs que se encontraron en cada bloque.

Al calcular la suma de prefijos se obtiene el índice para almacenar a los hilos que contienen un SCAP (valor de 1) en posiciones contiguas dentro de la memoria global.

4.4.2. atomicAdd

Uno de los procesos más importantes del Algoritmo 3 (*Kernel*) es el que incluye la instrucción *atomicAdd* o *suma atómica* de CUDA (descrita entre los pasos 6 y 8). A continuación se explica su funcionamiento.

Algoritmo 11 atomicAdd, subproceso del Algoritmo 3 (*Kernel*)

1: **si** threadIdx.x == blockDim.x - 1 **entonces**
2: $indice_bloque$ = atomicAdd($total_positivos$, max_SCAPs_bloque)
3: **fin si**

Las funciones *Atomic* de CUDA (descritas en la Sección 3.9.1 del Capítulo 3) realizan operaciones atómicas de lectura-escritura en la memoria global o compartida garantizando que la instrucción se ejecutará sin la interferencia de otros hilos. En otras palabras, ningún otro hilo puede acceder a esta dirección de memoria utilizada hasta que la operación se haya completado.

Previo a la instrucción *atomicAdd* del Algoritmo 3 se calcula la suma de prefijos, al terminar esta instrucción existen múltiples bloques de hilos "esperando" escribir en la memoria global los SCAPs que almacenan. Para que no haya traslapes es necesario que cada bloque conozca el índice a partir del cual hará la escritura, el cálculo de este índice se hace de manera atómica, es decir, un bloque a la vez, para esto se utiliza la instrucción *atomicAdd*.

Para obtener este índice solo se necesita que un hilo por bloque ejecute la instrucción *atomicAdd*. Convenientemente este hilo es el último de cada bloque porque este contiene la suma de prefijos total de dicho bloque, con este valor se asegura reservar espacio para todos los elementos positivos del bloque.

La función recibe dos parámetros: el primero indica la localidad de memoria que reserva para hacer la operación de suma y el segundo parámetro indica el valor que va a sumar. La función retorna el valor que se encuentra dentro de la localidad de memoria antes de hacer la suma.

Al comenzar el algoritmo, la localidad de memoria reservada para la instrucción *atomicAdd* tiene un valor inicial de 0. El primer bloque que realice la instrucción *atomicAdd* sumará sus n casos positivos en a y los almacenará desde la posición 0. El segundo bloque sumará sus m casos positivos y los almacenará a partir de la posición n. El tercero lo hará a partir de la posición $n+m$. La operación continua hasta que el último bloque es atendido.

En la Figura 31 se muestra un ejemplo donde el bloque 3 (B3) ya realizó la instrucción *atomicAdd* sobre la localidad de memoria *loc_memoria* reservada para la instrucción atomicAdd de la cual obtuvo el índice 0 y le sumó los cuatro SCAPs que contiene (recuadro en la parte superior derecha del bloque B3). Esto se traduce en que B3 almacenó 4 SCAPs a partir de la posición 0 de la memoria global.

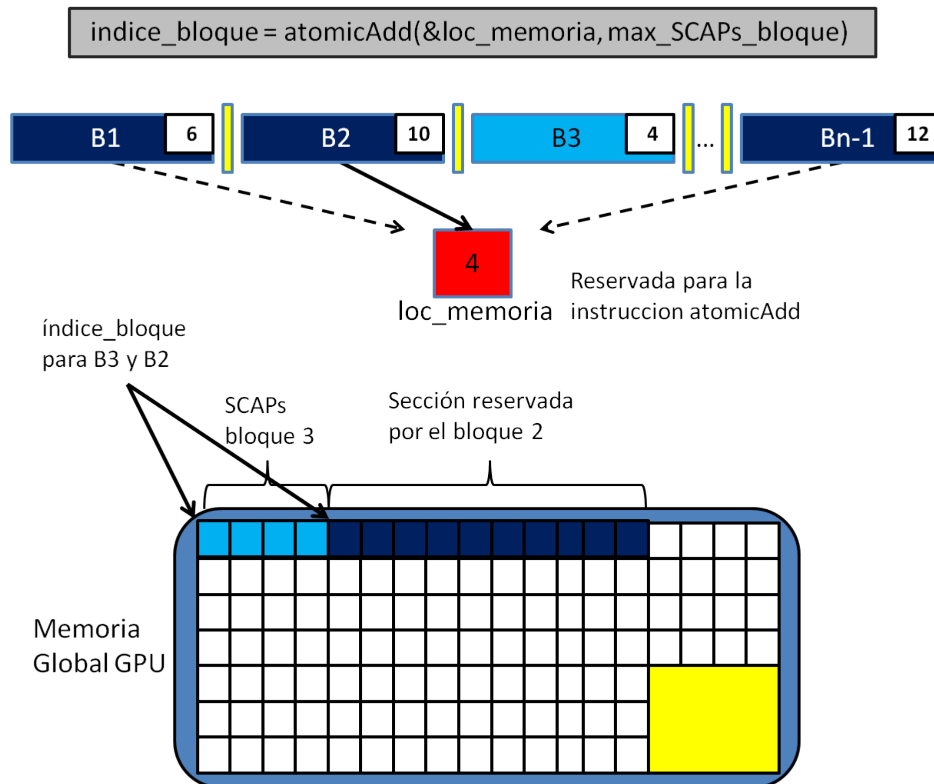

Figura 31: Representación de la instrucción atomicAdd para obtener el índice a partir del cual cada bloque almacenará los SCAPs en la memoria global del dispositivo.

Cuando B3 termina la consulta en *loc_memoria*, los bloques B1, B2 hasta Bn-1 compiten por accesarla de nuevo para obtener el índice a partir del cual escribirán los SCAPs en la memoria global. En el ejemplo, la línea continua del bloque 2 (B2) muestra que este obtiene acceso a *loc_memoria*. B2 recibe el valor 4 como índice inicial para almacenar los SCAPs y suma los 10 SCAPs que contiene, dejando listo el índice para el siguiente bloque que ejecute la operación *atomicAdd*.

Gracias a este proceso 'secuencial' cada bloque conoce el índice a partir de cual almacenará los SCAPs que contiene. Al terminar la instrucción *atomicAdd* los n bloques que ya tienen

índice escriben sus resultados de manera paralela.

4.5. Secuencias permutadas

Como parte de los experimentos de este estudio, se agruparon las secuencias que contienen el mismo multiconjunto de aminoácidos pero ordenados de forma distinta para detectar aquellas combinaciones que generaban el mayor número de casos positivos. Para hacer el análisis de las secuencias positivas arrojadas por el algoritmo *Kernel* (Algoritmo 3), se implementó un algoritmo adicional dentro de la solución original para detectar permutaciones.

Para crear los multiconjuntos el algoritmo añade una llave a cada secuencia que resulte ser SCAP con base en los aminoácidos que la conforman. Esto lo hace por medio de una función Hash. Con esta función, dos secuencias que tengan los mismos aminoácidos generan la misma llave, lo que se define como una colisión. Las colisiones son útiles para contar secuencias equivalentes.

El proceso de asignar llaves a las secuencias se realiza en el dispositivo, en el Algoritmo 2, dentro de la condición del paso 10. Cumplir con dicha condición significa que una secuencia cumple con los parámetros para ser SCAP, es entonces que se le asigna una *llave hash* como se indica en la siguiente expresión:

$SCAP.Llave_Hash = funcionHash(SCAP.secuencia)$

El método $funcionHash$ recibe como parámetro la secuencia SCAP, la llave se genera a partir de los aminoácidos de esta. Para realizar la asignación de la llave fue necesario añadir el campo $Llave_hash$ a la estructura $SCAPs_Estructura_Completa$ como se muestra en la Figura 32.

4.5.1. Función Hash

La llave de cada secuencia se genera con una sumatoria de los aminoácidos que la conforman. A cada aminoácido se le asigna un valor numérico como se indica en la Tabla 10 en donde se describen los aminoácidos con código de tres letras y de una letra en las columnas 'AA C3' y 'AA C1' respectivamente; la columna *Valor* × *aminoácido* es la que contiene el

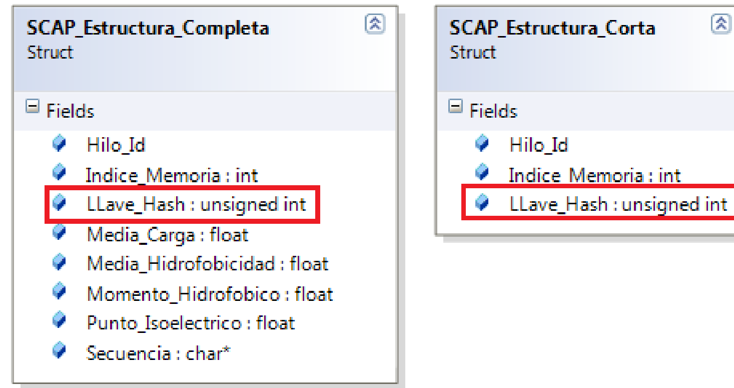

Figura 32: Estructuras de datos para almacenar SCAPs con el parámetro “Llave Hash”.

valor que se le asigna a cada aminoácido para realizar la sumatoria de la Ecuación 8 con la que se crea la $Llave\ Hash$.

$$LlaveHash = \sum_{i=1}^{n} aa_i, \tag{8}$$

donde n representa la longitud de la secuencia y aa_i indica el valor de cada aminoácido. Por ejemplo, la secuencia “RAAAY” genera la llave “54783272”, esto de acuerdo con los valores que se muestran de la Tabla 10, donde: [R $= (82 \times 47)^2$] + [A$\times$3 $= (65 \times 1)^2 \times 3$] + [Y $= (89 \times 71)^2$]. La misma llave se genera con la secuencias “ARAYA” y “AAAYR” lo que significa que todas ellas contienen los mismos aminoácidos y por lo tanto pertenecen al mismo conjunto según la sumatoria.

La ecuación que se muestra en la columna $Valor \times aminoácido$ se define con ayuda del código de una letra de los aminoácidos; cada letra se convierte a su equivalente en código ASCII; además a cada aminoácido se le asigna un número primo. Estos dos valores se multiplican y el resultado se eleva al cuadrado.

En la ecuación se utilizan los números primos para evitar que dos secuencias con aminoácidos diferentes generen la misma llave, lo que ocasionaría conteos erróneos. Como resultado de esta operación, cada aminoácido tiene un valor distante de los demás, lo que crea un mayor número de llaves exclusivas. Es necesario aclarar que esta ecuación desarrollada de forma empírica no garantiza de forma rigurosa que dos secuencias con diferentes aminoácidos generen la misma llave, lo que ocasionaría un error en el cálculo.

Una alternativa para generar llaves iguales con secuencias que contengan los mismos aminoácidos ordenados de forma diferente sería utilizar el *método de división* propuesto en Cormen *et al.* (2009). En este método se direcciona una llave k a uno de los m espacios disponibles al tomar el residuo de k dividido por m. Es decir, la función hash es $h(k) = k \ mod \ n$.

Para que este método sea efectivo en generar llaves con las características requeridas por este problema, Cormen *et al.* (2009) proponen el valor de $m = 2^p - 1$ y que k se interprete como una cadena de caracteres en *radix* 2^p. El libro refiere que esta configuración particular representa una pobre opción para generar llaves hash si se quiere evitar colisiones cuando solo existen permutaciones entre los caracteres que conforman una cadena, pero es exactamente lo que se busca para este trabajo. Con esta propuesta podría resultar más sencillo demostrar que dos secuencias con aminoácidos diferentes no generen la misma llave.

Tabla 10: Aminoácidos en código de 3 letras (columna 1) y 1 letra (columna 2) y la operación para asignarle un valor numérico a cada uno (columna 3).

AA C3	**AA C1**	**Valor × Aminoácido**	**AA C3**	**AA C1**	**Valor × Aminoácido**
Ala	A	$(65 \times 1)^2$	Met	M	$(77 \times 31)^2$
Cys	C	$(67 \times 3)^2$	Asn	N	$(78 \times 37)^2$
Asp	D	$(68 \times 5)^2$	Pro	P	$(80 \times 41)^2$
Glu	E	$(69 \times 7)^2$	Gln	Q	$(81 \times 43)^2$
Phe	F	$(70 \times 11)^2$	Arg	R	$(82 \times 47)^2$
Gly	G	$(71 \times 13)^2$	Ser	S	$(83 \times 53)^2$
His	H	$(72 \times 17)^2$	Thr	T	$(84 \times 59)^2$
Ile	I	$(73 \times 19)^2$	Val	V	$(86 \times 61)^2$
Lys	K	$(75 \times 23)^2$	Trp	W	$(87 \times 67)^2$
Leu	L	$(76 \times 29)^2$	Tyr	Y	$(89 \times 71)^2$

4.5.2. Agrupando llaves

El agrupamiento de los SCAPs que contienen la misma llave, es decir, aquellos que contienen los mismos aminoácidos pero en diferentes posiciones, se realiza del lado del huésped. Este proceso secuencial se ejecuta después de que el dispositivo transfiere el arreglo de SCAPs al huésped. Para el conteo de llaves se utiliza un "*Mapa*".

Los *Mapas* son contenedores asociativos de C++ que almacenan elementos formados por la combinación de una llave y el valor almacenado. En los *Mapas*, la llave se utiliza para

ordenar e identificar elementos únicos, mientras que los valores almacenados representan el contenido asociado a esta llave. El procedimiento para agrupar las llaves se describe en el Algoritmo 12.

Algoritmo 12 Algoritmo para agrupar las *llaveshash* asociadas a cada SCAP

Entrada: El arreglo $SCAPs$ que arroja el GPU al terminar su ejecución, la variable $SCAPs_encontrados$ que tiene el número total de secuencias positivas y un contenedor tipo $Mapa$ para contar las llaves repetidas (secuencias equivalentes).

Salida: El $Mapa$ con el conteo de las secuencias equivalentes.

1: **para** $i \leftarrow 0$ hasta $SCAPs_encontrados$ **hacer**
2: **si** la llave de $SCAPs[i]$ se encuentra en el $Mapa$ **entonces**
3: incrementar el contador de esta llave dentro de $Mapa$
4: **si no**
5: agregar la llave y la secuencia asociada en el $Mapa$
6: iniciar el contador de la llave en 1
7: **fin si**
8: **fin para**

En el ciclo del paso 1 se hace un recorrido por los elementos del arreglo SCAPs. La condición del paso 2 verifica si la llave del $SCAPs[i]$ existe en el $Mapa$. Si la llave ya existe solo se incrementa el contador de esa llave en el paso 3. Si la llave no existe, entonces se inserta en el $Mapa$ junto con la secuencia asociada en el paso 4; en el paso 5 se inicia el contador de la llave en '1'.

Al finalizar el ciclo, el $Mapa$ tiene un contador por cada llave diferente y una secuencia (que se registra en el paso 4) asociada a cada llave.

Capítulo 5. Experimentos y Resultados

En el capítulo anterior se describieron a detalle los algoritmos paralelos y secuenciales que se implementaron para la detección de los péptidos candidatos a SCAPs. Lograr la implementación definitiva requirió de ensayos con diferentes enfoques, estos se revisan de forma breve en este capítulo. También se compararon diferentes configuraciones de hardware y software para determinar la mejor en términos de tiempo de cómputo. Una vez elegida la estrategia de paralelización definitiva y la configuración de hardware a utilizar se midieron los tiempos de ejecución para compararlos con los que se logró con la implementación secuencial y posteriormente con los reportados en los trabajos en los que se basó esta investigación. Además de analizar la ganancia en tiempo se hizo un análisis de las características de los candidatos a SCAPs que arroja el algoritmo que se implementó.

5.1. Detalles preliminares

El proceso para la detección de SCAPs se lleva a cabo con el cálculo de cuatro propiedades fisicoquímicas: Media de Hidrofobicidad, Media de Carga Neta, Punto Isoeléctrico y Momento Hidrofóbico Helicoidal.

Como primer paso, se hizo una implementación secuencial en C++ que calcula las cuatro propiedades y se mide el tiempo de ejecución. El algoritmo secuencial se basa en la implementación, en lenguaje Fortran, del trabajo de Polanco *et al.* (2011).

Para los experimentos con la implemenación secuencial se tomó como base la secuencia de nueve aminoácidos que refiere el artículo de Polanco *et al.* (2011): RAAAYXXXX, donde R, A y Y representan la Arginina, Alanina y Tyrosina, respectivamente, mientras que X es una variable que representa a cualquiera de los 20 aminoácidos esenciales. Con esta secuencia se comenzó el análisis de 20^4 péptidos. Posteriormente se aumentó el número de aminoácidos variables para obtener más información en cuanto al tiempo de ejecución y al número de secuencias positivas, esto se hizo hasta llegar a siete aminoácidos como lo muestra la secuencia: RAXXXXXXX. Los resultados de estas pruebas arrojaron los datos necesarios para tener un punto de comparación con los que se logran por las subsecuentes implementaciones en

paralelo.

Se experimentó con los enfoques de paralelización descritos en la Sección 4.1 antes de lograr desarrollar el algoritmo con los mejores resultados en tiempo de ejecución y espacio requerido en memoria. Para lograrlo se implementaron las técnicas y prácticas sugeridas por los manuales de NVidia para el manejo óptimo de CUDA y con esto obtener el mejor desempeño del GPU con el que se trabajó.

Una vez definida la estrategia de paralelización definitiva esta se comparó con la implementación secuencial. La estrategia para las pruebas fue la misma que se utilizó en la implementación secuencial, se comenzó con la secuencia con cuatro aminoácidos variables: RAAAYXXXX. Se avanzó en el número de secuencias analizadas hasta llegar al límite del GPU que se utilizó, el cual resultó ser de 20^7 secuencias, por lo que a partir de este punto se particionó el problema en bloques de 20^7 para ser procesados por el GPU. Para los casos que van desde 20^8 secuencias en adelante se hicieron llamadas consecutivas al *kernel* en potencias de 20.

Terminados los experimentos con una cadena de 10 aminoácidos variables consecutivos, y analizando las limitaciones de este esquema, se decidió hacer la longitud de la cadena variable. La nueva longitud puede ser desde 7 hasta 25 aminoácidos, donde desde 1 y hasta 10 aminoácidos pueden ser de valor y posición variables, generalizando así la propuesta inicial de Polanco *et al.* (2011). Con esto cubrimos casos de usos más reales y diversos. El valor de 7 como número mínimo de amioácidos se debe a que esta es la longitud a partir de la cual es posible calcular el momento hidrofóbico helicoidal (ver Sección 4.3.6).

5.2. Configuración de hardware y software

Los experimentos se realizaron utilizando el procesador Intel Core i7-3820, velocidad del reloj de 3.60 Ghz y RAM de 8.00 GB. Para los experimentos en paralelo se utilizó la tarjeta de video NVidia GeForce GTX 680 con 1536 núcleos CUDA y 2 Gigabytes de memoria global. Las pruebas iniciales se hicieron en dos sistemas operativos, Windows 7 (64 bits) y Ubuntu versión 12.04 (64 bits). La implementación en Windows se realizo en C++ de Microsoft

Visual Studio 2010 con el complemento (pulg-in) Nsight de NVidia para programar el GPU. En Ubuntu se usó Eclipse, también con el complemento de NVidia Nsight. Además para el caso de Ubuntu se utilizaron dos compiladores diferentes, gcc (GNU C Compiler) y nvcc (NVidia C Compiler). Las especificaciones técnicas de la tarjeta están en el Apéndice A.1.

5.2.1. Comparando ambientes de desarrollo

Como experimento inicial se comparó el tiempo de ejecución de las configuraciones de software entre Windows y Ubuntu para determinar la existencia o no de una diferencia significativa entre estos sistemas operativos. Los resultados de dicha comparación en los algoritmos secuencial y paralelo se muestran en la tablas 11 y 12, respectivamente. Los valores mostrados en cada tabla representan el tiempo total que se tarda el algoritmo al recorrer todas las subsecuencias de longitud 4, 5 y 6. La secuencia que se utilizó como patrón es la RAAAYXXXX para las subsecuencias de longitud 4, RAAAXXXXX para subsecuencias de longitud 5 y RAAXXXXXX para subsecuencias de longitud 6.

Tabla 11: Comparación de tiempos con diferentes configuraciones de software en la implementación secuencial. Análisis de 20^4 hasta 20^6 secuencias.

Configuración para algoritmo secuencial (tiempo en segundos)			
#Sec.	**MS Visual Studio**	**Ubuntu & Eclipse gcc**	**Ubuntu & Eclipse nvcc**
20^4	8.7	16	13.5
20^5	167	322	272
20^6	3328	6400	5440

Tabla 12: Comparación de tiempos con diferentes configuraciones de software en la implementación paralela. Análisis de 20^4 hasta 20^6 secuencias.

Configuración para algoritmo paralelo (tiempo en segundos)		
#Sec.	**MS Viusal Studio**	**Ubuntu & Eclipse nvcc**
20^4	0.337	0.505
20^5	7.5	11.4
20^6	160.6	244.8

Los algoritmos secuencial y paralelo utilizados para la comparación de ambientes de desarrollo, los cuales arrojan los tiempos de las tablas 11 y 12, no son óptimos pero son idénticos en todas las configuraciones.

Como resultado de la comparación de rendimiento entre sistemas operativos y plataformas de desarrollo, se observó que la implementación secuencial en Windows fue 1.6 veces más rápida que la versión nvcc de Ubuntu y 2.0 veces mejor que la versión gcc. Igualmente la implementación en paralelo de Windows con MS Visual Studio fue 1.5 veces más rápida que la configuración de Ubuntu con nvcc. Se conjetura que la diferencia observada a favor de Windows se debe a que la configuración empleada de Ubuntu no está optimizada para explotar la capacidad del compilador nvcc de NVIDIA.

La configuración de Windows se utilizó como base en la implementación final de los algoritmos secuencial y paralelo.

5.3. Implementación secuencial

Esta implementación consiste en calcular de forma secuencial las cuatro propiedades fisicoquímicas para cada péptido. Si el valor de alguna propiedad no está dentro de los intervalos requeridos por esta, el péptido no es candidato a SCAP y no es necesario continuar con la revisión de las propiedades restantes. Por esta razón y para optimizar la implementación, las propiedades se ordenaron convenientemente de menor a mayor tiempo de ejecución. El tiempo medido para cada propiedad en secuencias de 4 hasta 7 aminoácidos variables se muestra en la Tabla 13.

Tabla 13: Tiempo en segundos por cada propiedad en el análisis de 20^4 hasta 20^7 secuencias.

	Tiempo promedio por propiedad (seg)			
#Sec	**MC**	**MH**	**PI**	**MHH**
20^4	0.005	0.004	0.225	0.774
20^5	0.111	0.099	4.532	15.392
20^6	2.224	2.132	90.998	299.192
20^7	44.360	44.430	1815.500	6108.810

En la Tabla 13 se puede observar que los cálculos de la media de carga y media de hidrofobicidad son los que menor tiempo requieren, seguidos del punto isoeléctrico y finalmente el mometo hidofóbico helicoidal. Estos resultados dictan el orden en que las propiedades son calculadas en la implementación secuencial optimizada. El análisis de tiempo por propiedad dió como resultado la estrategia de paralelización “Kernels Concurrentes” descrita en la Sección 4.1.

La implementación se desarrolló en C++ tomando como base el trabajo de Polanco *et al.* (2011). Las cuatro propiedades fisicoquímicas están escritas originalmente en Fortran y las cuales se reescribieron y mejoraron en C++ con el objetivo de emplearlas en las implementaciones secuencial y paralela. El cálculo del punto isoeléctrico fue modificado para hacerlo por el método de bisección ya que este se hacía con una búsqueda lineal. Con esto se mejoró el tiempo de ejecución de esta función de $O(n)$ a $O(log(n))$. Para esta implementación se utiliza un solo hilo dentro del procesador i7-3820.

5.3.1. Tiempos de ejecución

La Tabla 14 muestra los tiempos que tarda el algoritmo secuencial en analizar 20^4 hasta 20^7 péptidos. Para estos experimentos se tomó como base la secuencia de nueve aminoácidos propuesta por Polanco *et al.* (2011): "RAAAYXXXX"; comenzando con cinco aminoácidos fijos y cuatro variables; para aumentar el número de secuencias que se analizan se incrementa el número de aminoácidos variables y se disminuye el número de los aminoácidos fijos de derecha a izquierda.

Tabla 14: Tiempos de ejecución en el cálculo de las cuatro propiedades propuestas en la implementación secuencial con 20^4 hasta 20^7 secuencias. El tiempo mostrado con 20^8 y 20^9 secuencias son estimados basados en el tiempo promedio por secuencia.

#Sec.	**SCAPs**	**% Positivos**	**Promedio(seg)**	**DE(seg)**	**Prom. x Sec.(ns)**
20^4	3	0.0018 %	0.0498	0.012	311.25
20^5	2177	0.0680 %	1.4312	0.036	447.25
20^6	191,064	0.2985 %	41.3619	1.047	646.28
20^7	7,333,827	0.5729 %	1,042.991	8.369	814.84
20^8	n/a	n/a	25,157.087	n/a	982.69
20^9	n/a	n/a	589,087.88	n/a	1,150.56

La primera columna de la Tabla 14 muestra el número de secuencias que se analizan, la segunda el número de casos positivos, la tercera el porcentaje de casos positivos en relación con los analizados, la cuarta el tiempo promedio tras ejecutar 20 veces el algoritmo. Con esto se calcula la desviación estándar que se muestra en la columna 5 y finalmente se presenta el tiempo promedio por secuencia, este se obtiene al dividir el tiempo de ejecución de la columna 4 entre el número de secuencias que se analizan.

En el caso de 20^6 secuencias el algoritmo secuencial tarda un poco más de 40 segundos, la desviación estándar es de un poco más de un segundo y el tiempo promedio por secuencia es de 646.84 nanosegundos (cuarta fila de la Tabla 14).

Aunque solo se midió el tiempo de ejecución de 20^4 hasta 20^7 secuencias, es posible estimar el comportamiento de 20^{7+n} secuencias gracias al tiempo promedio por secuencia. El tiempo promedio se obtiene de dividir el número de secuencias que se analizan entre el tiempo de ejecución. Este promedio por secuencia crece de manera lineal como lo muestra la Figura 33. El incremento en tiempo por secuencia al aumentar el número de péptidos analizados en el intervalo de 20^4 hasta 20^7 es de 40 % aproximadamente. Con este porcentaje se estimó el tiempo para 20^8 y 20^9 secuencias que se muestra en la Tabla 14. Al ser una estimación, el número de SCAPs encontrados, porcentaje de casos positivos y desviación estándar (columnas 2, 3 y 5, respectivamente) no están disponibles (n/a).

El incremento en el tiempo promedio por secuencia se debe al aumento del porcentaje de secuencias positivas que se encuentran conforme aumenta el número de secuencias que se evalúan, este porcentaje se muestra en la columna 3 de la Tabla 14. El aumento del porcentaje de secuencias positivas indica que un mayor número de veces fue necesario el cálculo de las cuatro propiedades. Debido a que el tiempo de cada secuencia que se evalúa es acumulativo en el algoritmo secuencial, el tiempo de ejecución total crece, lo que se refleja en el promedio por secuencia. Este comportamiento se muestra en la Figura 33.

Es importante notar que el objetivo del algoritmo es determinar si la secuencia es o no SCAP. No cumplir con los valores requeridos por cualquier propiedad es condición suficiente para ser un caso negativo, por lo que en la mayoría de los casos no es necesario el cálculo de las cuatro propiedades. Por esta razón los tiempos reportados del algoritmo secuencial en la Tabla 14 son menores que los observados en el cálculo del punto isoeléctrico y momento hidrofóbico helicoidal dentro de la Tabla 13, pues en ese caso cada secuencia realizó obligadamente las operaciones requeridas por estas propiedades con mayor tiempo de ejecución.

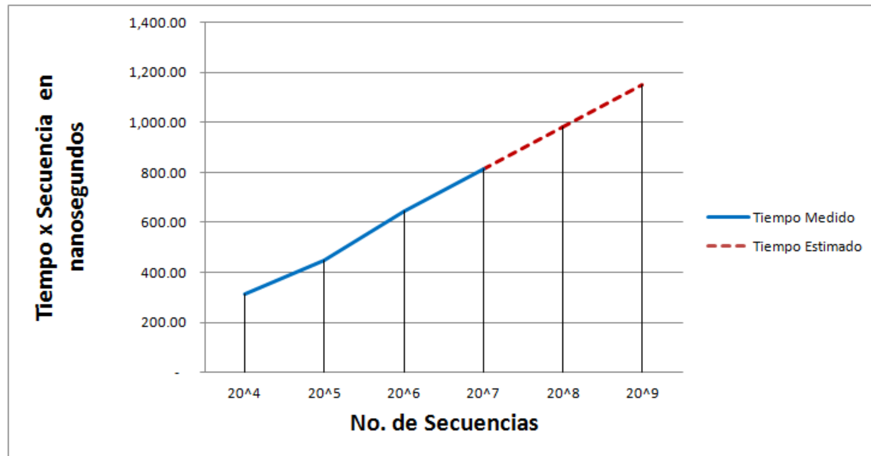

Figura 33: Tiempo promedio de cómputo por secuencia en función de la longitud de la secuencia analizada.

En la Figura 34 se aprecia cómo el incremento exponencial del número de secuencias que se analizan se manifiesta directamente en el tiempo de ejecución.

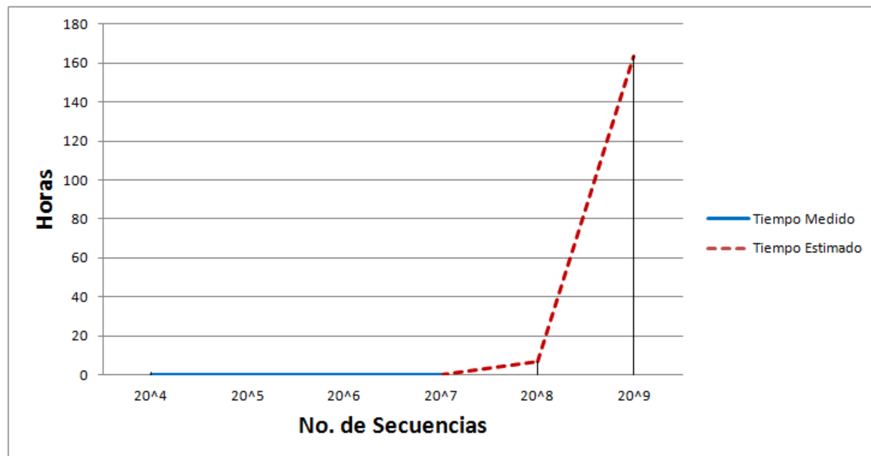

Figura 34: Crecimiento exponencial en el tiempo de ejecución del algoritmo secuencial en función de la longitud de la secuencia.

5.4. Implementación del esquema paralelo propuesto

5.4.1. Mejoras técnicas

Definido el diseño de paralelización general, se estudiaron las especificaciones técnicas de la arquitectura del GPU que se utilizó y de la plataforma CUDA para mejorar los tiempos de ejecución.

Un parámetro importante para optimizar un proceso paralelo es evitar la divergencia (descrita en la Sección 3.7.2 del Capítulo 3). La manera óptima de ejecutar instrucciones en paralelo es hacer que todos los hilos dentro de un *warp* recorran la misma bifurcación para mantener un comportamiento síncrono y eficiente.

El código es divergente, es decir, cada hilo tiene la opción de elegir diferentes caminos de ejecución. Esto se debe a que existe una serie anidada de instrucciones de control "if", ya que el cálculo de las propiedades es secuencial y escalonado, es decir, primero se calcula la propiedad A, si el resultado cumple con las condiciones establecidas entonces se calcula la propiedad B y así sucesivamente hasta llegar a la propiedad D. Las propiedades se ordenan de menor a mayor tiempo de ejecución para optimizar el cálculo.

Esquemas divergente y no divergente

Con la configuración divergente se calcula una propiedad a la vez, si el resultado de una propiedad cumple con las condiciones establecidas, entonces se calcula la siguiente propiedad. Esto se hace de manera sucesiva hasta calcular las cuatro propiedades en caso de ser necesario. La gran mayoría de los casos son negativos (no son SCAPs), donde un gran porcentaje de casos no cumple ni siquiera con la condición requerida para la primera propiedad que se evalúa, gracias a esto la divergencia entre los hilos dentro de un *warp* no es crítica. Además existe un ahorro significativo en el número de operaciones que se realizan ya que el porcentaje de secuencias que requerirán hacer el cálculo de las cuatro propiedades es muy bajo en comparación con el enfoque de "No Divergencia" a explicarse más adelante. Recordemos que las propiedades se ordenaron de menor a mayor tiempo de ejecución, o dicho de otra forma, de menor a mayor número de operaciones. El diagrama 'A' de la Figura 35 muestra el flujo de esta configuración.

En la configuración no divergente cada una de las secuencias realiza el cálculo de las cuatro propiedades fisicoquímicas, con esto se evita la divergencia entre hilos pero se realizan una gran cantidad de operaciones innecesarias, ya que la mayoría de los casos que se analizaron son negativos y basta con que una propiedad no cumpla con los parámetros que se establecieron para determinar si una secuencia es o no SCAP. En el diagrama 'B' de la Figura 35 se

muestra el flujo de esta configuración. En la siguiente sección se miden los tiempos de estos dos enfoques.

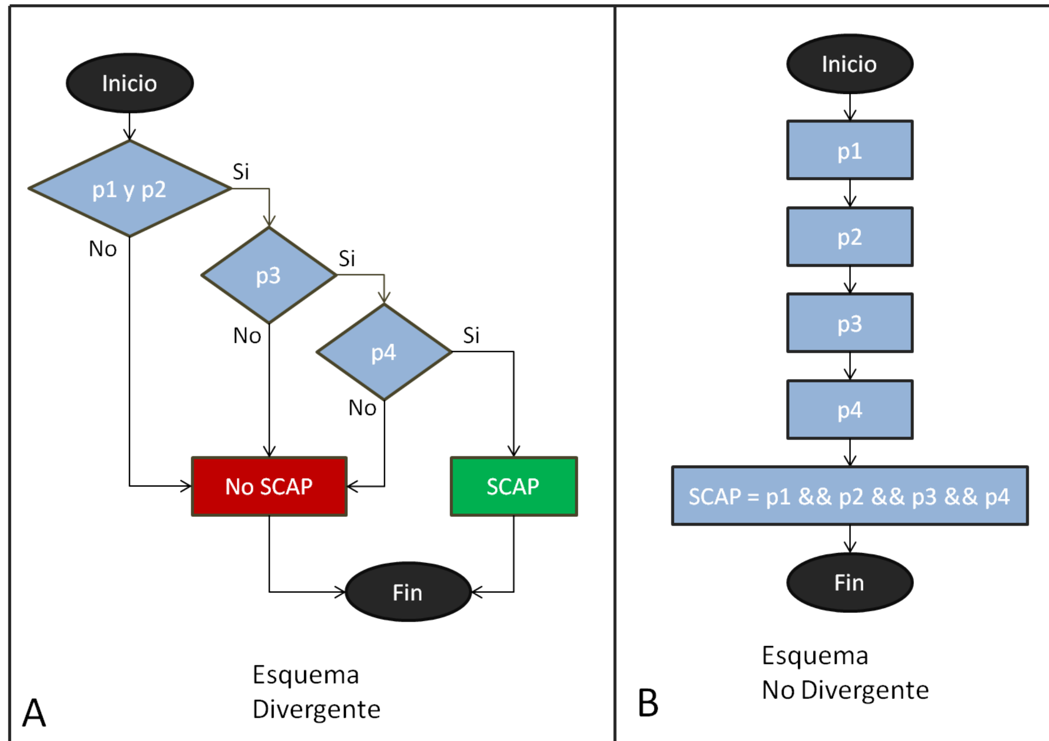

Figura 35: Estrategias Divergente y No Divergente.

Tiempos de ejecución de los esquemas con y sin divergencia

Comparando los tiempos de ejecución de 20^4 y 20^5 secuencias, el enfoque Divergente resultó 2.12 veces más rápido que el No Divergente, como se muestra en la Tabla 15. El resultado está de acorde con el razonamiento lógico de que en este problema en particular es más eficiente realizar un número menor de operaciones a pesar de la divergencia, ya que la mayoría de los casos, al ser negativos, siguen la misma bifurcación del diagrama de flujo que se muestra en la Figura 35 y por lo tanto la divergencia no es crítica.

Es necesario recalcar que estos enfoques almacenan todos los péptidos que analizan dentro de la memoria global del dispositivo, es decir, casos positivos y negativos, pues no es trivial discriminar los casos negativos cuando se trabaja en paralelo, por lo que el siguiente paso es filtrar y guardar en memoria solo los casos positivos.

Tabla 15: Promedio y desviación estándar de los enfoques Divergente y No Divergente. La columna "Tiempo" muestra el promedio de 20 ejecuciones.

#Sec.	Enfoque	Tiempo(ms)	Desviación Estandar(ms)
20^4	No Divergente	12.444	0.265
20^4	Divergente	5.865	2.515
20^5	No Divergente	226.612	6.32
20^5	Divergente	105.930	24.023

5.4.2. Propuesta de uso de espacio en memoria global

El problema de espacio en memoria es notorio cuando los resultados muestran que más del 99 % de los casos son negativos. Llenar la memoria global del dispositivo con estos casos es ineficaz, y a la postre se traduce en un mayor número de llamadas desde el huésped hacia el dispositivo.

Se consideró como meta importante diseñar un algoritmo que almacenará solo los casos positivos. El objetivo es optimizar el espacio de memoria global que se requiere sin afectar la meta principal que es el tiempo de ejecución total, es decir, que las operaciones de filtrado no tuvieran repercusiones significativas en los tiempos de ejecución. En caso contrario sería una filtrado estético pero poco útil para el objetivo primordial. Lograr optimizar el espacio que se requiere se traduce en menos llamadas al dispositivo y por lo tanto en una mejoría en el tiempo total del proceso.

La dificultad del filtrado radica en acomodar los casos positivos en posiciones sucesivas de la memoria global y desechar los casos negativos. Cuando un hilo detecta un SCAP no tiene manera directa de obtener el índice donde lo almacenará por ser un proceso en paralelo.

Para resolver el problema del índice se analizaron dos estrategias, la primera se basa en la instrucción "*atomicAdd*" que ofrece el SDK de CUDA. La segunda solución es utilizar también *atomicAdd* pero en conjunto con la técnica *suma de prefijos en paralelo* que se aplica por bloque de hilos. Estas estrategias se explican a detalle en las secciones 4.4.1 y 4.4.2, respectivamente.

El filtrado de los casos positivos en conjunto con la estrategia divergente representan el

total de las operaciones necesarias para detectar un candidato a SCAP en este trabajo. Más adelante se revisan los tiempos que este conjunto de instrucciones arroja.

Restricciones del hardware y partición del espacio de soluciones

Además de la estrategia de paralelización y los enfoques de filtrado para el ahorro de memoria existe otro parámetro que fue necesario tomar en cuenta, la capacidad máxima de almacenamiento del dispositivo.

El número máximo de secuencias que se analizan con una sola llamada al GPU lo dicta la memoria global con la que viene equipado el dispositivo. En el caso de la tarjeta NVidia GTX 680 su capacidad de memoria es de 2 Gigabytes (ver especificaciones técnicas de la tarjeta en el Apéndice A.1). Cada SCAP se almacena utilizando la estructura "SCAP_Structure" que ocupa 48 bytes (descrita en la Sección 4.2.2). Si consideramos que aproximadamente 1 % de las secuencias que se analizan serán SCAPs, al analizar 20^8 secuencias tendremos 256 millones de casos positivos, lo que genera 11.5 Gigabytes de datos, muy por encima de la capacidad del dispositivo. Para atacar esta limitación de hardware se particiona el espacio de soluciones del problema.

Esta partición se realiza mediante llamadas al *kernel* con la máxima capacidad de memoria disponible, el número necesario de veces. La capacidad máxima que se puede atacar con la GPU que se utilizó es 20^7, pues genera 1 Gigabyte de datos aproximadamente. Para calcular 20^8 secuencias en adelante es necesario ejecutar el *kernel* de 20^7 en múltiplos de 20. Por ejemplo, si se quiere calcular 20^9 secuencias es necesario ejecutar el *kernel* de 20^7 secuencias 20^2 veces, es decir 400 veces, para así explorar todas las secuencias de longitud 9.

5.4.3. Tiempos de ejecución de la estrategia paralela propuesta

A continuación se muestran la cantidad de SCAPs que existen para diferentes longitudes de secuencias así como los tiempos de ejecución de las dos estrategias de paralelización recientemente descritas, estos representan los resultados más importantes de esta investigación. Los resultados se comparan con los descritos en la Sección 5.3. Se comparan además con los obtenidos por los enfoques descritos en la literatura que dieron origen a este trabajo. La

Tabla 16 concentra estos resultados.

Tabla 16: Cantidad de SCAPs y tiempos de ejecución en la implementación de la estrategia paralela propuesta. En la columna "Enfoque" la abreviación 'A' es para el enfoque suma atómica únicamente, 'A&P' hace referencia al enfoque suma atómica en combinación con la suma de prefijos.

#Sec.	Enfoque	SCAPs	% Positivos	Promedio(ms)	DE(ms)
20^4	A	3	0.0018 %	5.687	2.520
20^4	A&P	3	0.0018 %	5.690	2.514
20^5	A	2,177	0.0680 %	109.778	33.111
20^5	A&P	2,177	0.0680 %	109.082	33.372
20^6	A	191,064	0.2985 %	2267.93	548.327
20^6	A&P	191,064	0.2985 %	2270.78	547.280
20^7	A	7,333,827	0.5729 %	45,822.1	9720.986
20^7	A&P	7,333,827	0.5729 %	46,008.5	9692.584
20^8	A	205,539,446	0.8028 %	919,311.8	n/a
20^8	A&P	205,539,446	0.8028 % %	923,004.3	n/a
20^9	A	4,808,964,660	0.9393 %	18,256,369.9	n/a
20^9	A&P	4,808,964,660	0.9393 %	18,260,127.1	n/a

La columna #Sec. indica el número de secuencias que se analizan. 'Enfoque' hace referencia a la estrategia paralela que se utiliza para detectar y filtrar SCAPs. La columna SCAPs muestra el número de casos positivos. '% Positivos' es el porcentaje de casos positivos. 'Promedio(ms)' es el tiempo promedio en milisegundos al ejecutar 20 veces cada configuración. 'DE(ms)' es la desviación estándar en milisegundos que se observa en el cálculo de la columna anterior. La desviación estándar no está disponible (n/a) para los casos con 20^8 y 20^9 secuencias debido al tiempo requerido para hacer dicho cálculo.

En la Tabla 16, los casos positivos con 20^9 secuencias analizadas alcanzan casi el 1 %. El tiempo para calcular este número de secuencias es un poco mayor a 5 horas, mientras que el tiempo de escritura en archivo de los casi 5 mil millones de casos positivos es superior a las 7 horas, lo que nos da un total de 12 horas de procesamiento. En la Sección 5.4.4 se revisará el proceso de escritura a archivo.

Comparando los tiempos promedio de la Tabla 16 con los tiempos promedios del enfoque 'Divergente' de la Tabla 15 se observa que el incremento en tiempo al filtrar los resultados

positivos es mínimo, alrededor del 1 %. Además, el espacio que se ahorra en memoria global es significativo, más del 99 % de las secuencias que se analizan no son SCAPs y se desechan. El incremento en tiempo al filtrar los casos positivos es despreciable ya que, a pesar de tener operaciones extras para el filtrado, los enfoques que filtran los resultados ejecutan significativamente menos operaciones de escritura en la memoria global. La diferencia en tiempo se verá en forma clara en la Sección 5.4.4.

El ahorro de espacio resultado de desechar los casos negativos se traduce en la capacidad de analizar un mayor número de secuencias por llamada al GPU. Analizar un mayor número de secuencias por llamada al GPU se convierte en un menor número de llamadas a este. Hacer menos llamadas al GPU mejora el tiempo de ejecución total.

La Figura 36 muestra el comportamiento exponencial de los tiempos de ejecución de las estrategias *Operación Atómica* y *Suma de Prefijos*. Se aprecia una sola línea ya que los tiempos de ejecución son prácticamente iguales.

Figura 36: Tiempo de ejecución en horas por número de secuencias analizadas.

En la Figura 37 se muestra el tiempo promedio por secuencia. La diferencia del tiempo promedio por secuencia entre el análisis de 20^4 secuencias y el de 20^9 es apenas de 5 nanosegundos. Si consideramos este tiempo promedio como una función lineal en L y sabiendo que el número de secuencias a analizar crece de manera exponencial con L, podemos concluir que el tiempo que toma analizar todas las secuencias crecerá también de manera exponencial con L. Con estos datos es posible predecir el costo de analizar un mayor número de secuencias

antes de hacer los experimentos.

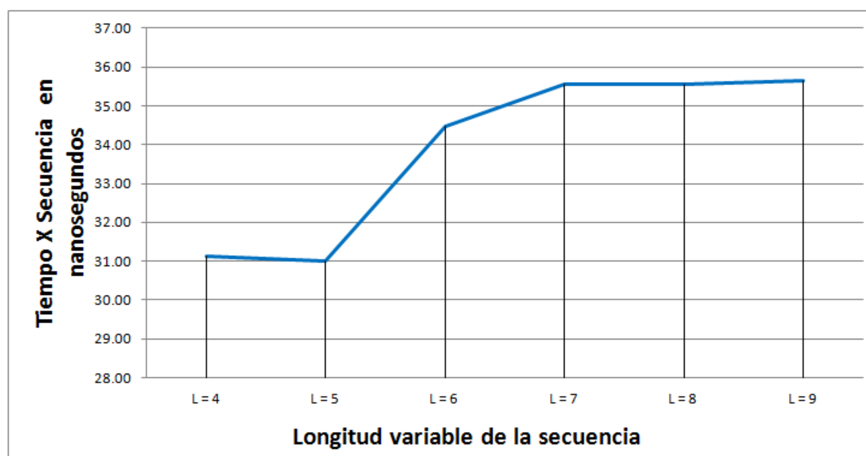

Figura 37: Tiempo de cómputo en función de la longitud variable de las secuencias (L).

5.4.4. Escritura a archivo

Los resultados arrojados por el dispositivo son escritos en archivo de manera secuencial del lado del huésped. Este proceso se optimizó para este trabajo, sin embargo, el filtrado de los resultados positivos con las estrategias *Operación Atómica* y *Operación Atómica con Suma de Prefijos* dentro del GPU se refleja directamente en el tiempo de escritura. Los enfoques Divergente y No Divergente no filtran las secuencias que se analizan, estos trasladan todos los péptidos, positivos y negativos, al huésped. Filtrar la información relevante del lado huésped resulta ineficiente.

En la columna ' % positivos' de la Tabla 16 se muestra que menos del 1 % de las secuencias que se analizan dan como resultado un candidato a SCAP, por esto, la diferencia en tiempo al momento de la escritura es de hasta 1390 veces mayor en los casos sin filtrar comparados con las estrategias que filtran; esta comparación se muestra en la Tabla 17.

La escritura de los resultados se hace en un archivo con el formato *Valores Separados por Comas* (CSV, por su siglas en inglés), el cual contiene las siguientes siete columnas: contador, Id del hilo que se analiza, secuencia de aminoácidos (se genera a partir del Id del hilo que se analizó), media de hidrofobicidad, media de carga neta, punto isoeléctrico y momento hidrofóbico helicoidal. Con este archivo es posible verificar el valor de cada propiedad de una

Tabla 17: Comparación de tiempos en la escritura al filtrar (Operación Atómica y Operación Atómica & Suma de Prefijos) y no filtrar (Divergente y No Divergente) los resultados positivos.

#Sec.	Enfoque	Escritura(seg)
20^4	No Divergente	0.984
20^4	Divergente	0.815
20^4	Operación Atómica	0.000
20^4	Operación Atómica & Suma de Prefijos	0.001
20^5	No Divergente	19.468
20^5	Divergente	17.197
20^5	Operación Atómica	0.013
20^5	Operación Atómica & Suma de Prefijos	0.014
20^6	Operación Atómica	1.173
20^6	Operación Atómica & Suma de Prefijos	1.197
20^7	Operación Atómica	45.186
20^7	Operación Atómica & Suma de Prefijos	46.6
20^8	Operación Atómica	1222.44
20^8	Operación Atómica & Suma de Prefijos	1221.91
20^9	Operación Atómica	25,988.1
20^9	Operación Atómica & Suma de Prefijos	25,994.3

secuencia y comprobar si estos cumplen o no con los parámetros establecidos.

Se resalta el proceso de filtrado y escritura ya que en los trabajos de Polanco *et al.* (2011) y Garcia-Ordaz *et al.* (2012) se analizan únicamente el tiempo dentro del dispositivo que se requiere para calcular las cuatro propiedades que definen un SCAP.

5.5. Análisis de Resultados

En esta sección se muestra la diferencia en tiempo entre la implementación secuencial y la implementación paralela. La comparación se hace con base en los resultados que se muestra en las Tablas 14 y 16. También se comparan los resultados con los reportados en los trabajos de Polanco *et al.* (2011) y Garcia-Ordaz *et al.* (2012) en los que se basó esta investigación. Finalmente, se hace un análisis sucinto de las secuencias que arroja el algoritmo.

5.5.1. Aceleración

En computación paralela, la aceleración S_p lograda por p procesadores se define como (JaJa, 1992b):

$$S_p = \frac{T_1}{T_p} \tag{9}$$

Donde: T_1 es el tiempo de ejecución del algoritmo secuencial y T_p es el tiempo de ejecución del algoritmo paralelo con p procesadores. La aceleración ideal teórica se obtiene cuando $S_p = p$.

Esta expresión se aplica en el esquema de paralelización "tradicional" donde el concepto de procesador es el mismo para la implementación secuencial que para los p procesadores de la implementación paralela. El concepto de procesador p es distinto entre CPU y GPU, donde la arquitectura de cada uno se optimiza para tareas diferentes, la aceleración no se puede calcular estrictamente con la fórmula definida en (9).

Una alternativa para calcular la aceleración consiste en dividir el tiempo del algoritmo secuencial entre el tiempo del algoritmo paralelo, especificando las características del hardware que se utiliza para ambos casos. La eficiencia de los algoritmos entonces se liga estrechamente a la arquitectura y los recursos que de estos se utilizan.

Es posible encontrar en la literatura algoritmos paralelos que reportan una aceleración en orden de magnitud, diferentes factores contribuyen en esta gran brecha. La manera en que se aprovechan las características del CPU y GPU y las optimizaciones que se hacen en el código son factores determinantes para una buena comparación (Lee *et al.*, 2010). Una de las causas principales para esta gran brecha en la aceleración que se reporta es por que explotan al máximo las características del GPU más no así con la implementación en el CPU.

FLOPS

Si la medición es puramente computacional, la unidad que se utiliza es el número de *Operaciones de Punto Flotante por Segundo* (FLOPs, por sus siglas en inglés). La medida en FLOPs de la arquitectura con p procesadores se calcula como (Fernandez, 2012):

$$pFLOPs = procesadores \times reloj \times \frac{FLOPS}{Ciclos}. \qquad (10)$$

La Tabla 18 muestra un comparativo de GFLOPs (10^6 FLOPs) entre los dispositivos que se utilizan en este trabajo además de la tarjeta Tesla C2070 que se utiliza en el trabajo de Garcia-Ordaz *et al.* (2012).

Tabla 18: Número de GFLOPs por Dispositivo.

Dispositivo	**GFLOPs**
CPU i7-3820	122
Tesla 2070	515
GeForce GTX 680	3080

Esta medida no toma en cuenta factores como el uso de la memoria por parte del algoritmo, el cual es uno de los factores más importantes para la aceleración. Es importante mencionar que del total del flujo de instrucciones solo una parte pequeña de la misma (aproximadamente 25 %) usa operaciones de punto flotante.

Ley de Amdahl

Para conocer la máxima aceleración esperada de un sistema cuando solo una parte de éste es mejorado se utiliza la ley de Amdahl (Hill y Marty, 2008). En computación paralela se utiliza para conocer la aceleración teórica máxima utilizando multiples procesadores y se calcula como:

$$s(N) = \frac{1}{(1-P) + \frac{P}{N}}, \qquad (11)$$

donde P es la sección paralelizable del sistema, $(1-P)$ representa a la sección que no se puede paralelizar y N es el número de proceadores utilizados. En el límite, cuando N tiende

a infinito la máxima aceleración tiende a $1/(1-P)$.

En nuestro caso la parte secuencial $(1-P)$ representa 0.015 del total del algoritmo, con esto podemos calcular que la aceleración máxima posible con el algoritmo propuesto es de hasta 66 veces.

La aceleración relativa entre el algoritmo paralelo propuesto y su versión secuencial es de hasta 32 veces. La aceleración se calcula en función del tiempo del algoritmo paralelo que se desarrolló en la tarjeta Geforce GTX 680 y el tiempo del algoritmo secuencial que se implementó en el procesador i7-3820 utilizando un solo hilo.

El incremento exponencial en tiempo de ejecución del algoritmo paralelo utilizando cada vez un mayor número de hilos se muestra en la Figura 38. En esta se aprecia un incremento en la aceleración de 5 veces por cada incremento en la longitud de las secuencias evaluadas.

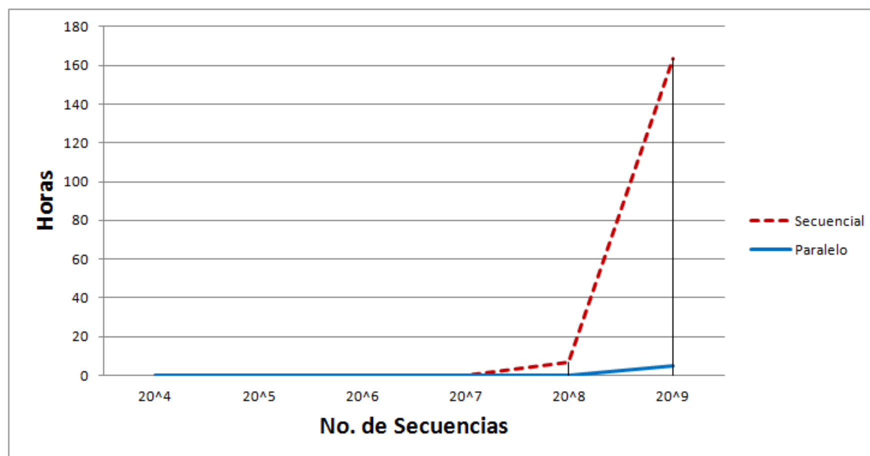

Figura 38: Comparación de tiempos entre los esquemas secuencial y paralelo.

La aceleración del enfoque paralelo no es constante, mejora en promedio 5 veces con cada incremento en el número de secuencias que se analizan de acuerdo a las mediciones que se muestran en las tablas 14 y 16. Con 20^4 secuencias el enfoque paralelo es 10 veces mejor que el secuencial. La aceleración con 20^9 secuencias incrementa a 32 veces. Esta aceleración se muesta en la Figura 39. Es importante recordar que el tiempo mostrado con 20^8 y 20^9 secuencias del algoritmo secuencial es con base en una estimación que se explica en la Sección

5.3.1.

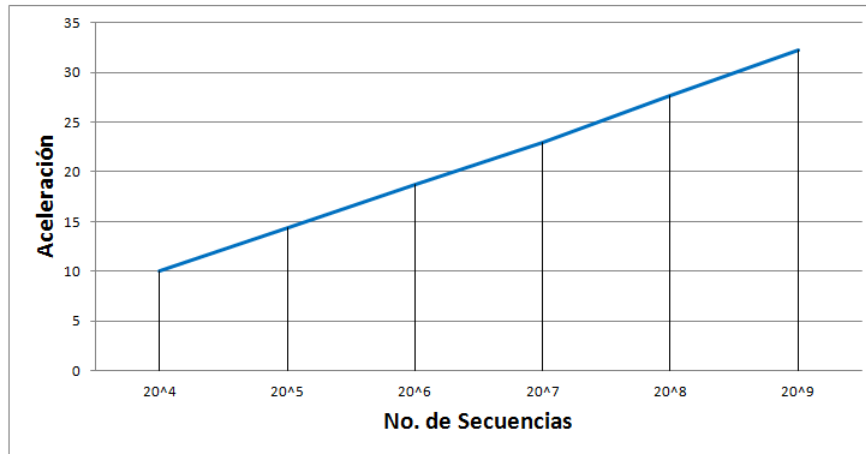

Figura 39: Aceleración en función del número de secuencias que se analizan.

La aceleración del algoritmo paralelo mejora en comparación con el algoritmo secuencial por el incremento en el porcentaje de casos positivos que se encuentran al aumentar el número de secuencias que se analizan.

En el algoritmo secuencial, el tiempo de cada caso positivo que requiere el cálculo de las cuatro propiedades se refleja directamente en el tiempo total del proceso. Por lo tanto cada caso positivo aumenta el tiempo promedio por secuencia.

En el algoritmo paralelo, el GPU está constantemente calendarizando *warps* de hilos de manera asíncrona, si un hilo dentro de un *warp* tiene una secuencia positiva, el *warp* completo espera a que este hilo termine, retrasando solo a 32 hilos. El resto de la calendarización de *warps* no se ve afectada. Por ser un proceso paralelo asíncrono, las secuencias positivas no tienen un gran impacto en el tiempo total del proceso como sí lo tienen en el algoritmo secuencial.

El tiempo promedio por secuencia se asocia directamente con la aceleración. La Figura 40 muestra el tiempo promedio por secuencia en el algoritmo secuencial y paralelo. En el caso del algoritmo secuencial, el tiempo promedio por secuencia para 20^8 y 20^9 diferentes péptidos se obtiene de la estimación del tiempo promedio por secuencia de la Sección 5.3.1.

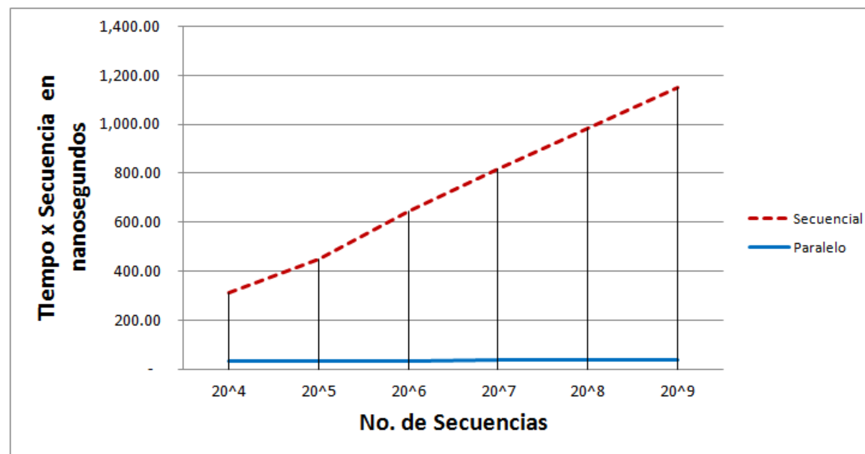

Figura 40: Tiempo de cómputo por secuencia, comparativo entre la implementación secuencial y paralela en función del número de secuencias que se analizan.

5.5.2. Comparación de las estrategias paralelas para la detección de SCAPs

En esta sección se comparan los tiempos de ejecución de las estrategias de Polanco *et al.* (2011) y Garcia-Ordaz *et al.* (2012) con los logrados en este trabajo.

La comparación es con el tiempo de análisis de 20^4 secuencias y con el tiempo promedio por secuencia. El promedio por secuencia es el tiempo total de las operaciones dividido entre el número de secuencias que se analizan.

Tabla 19: Comparación del tiempo de ejecución de 20^4 secuencias con los diferentes enfoques.

Año	**Dispositivo**	**Tiempo promedio por péptido**	**Análisis con 20^4 secuencias**
2011	FPGA Vertex II Pro	5.15×10^{-3}	3778 seg
2012	Telsa C2070	9.3×10^{-5}	10.6 ms
2013	i7-3820	311.1×10^{-9}	49.8ms
2013	GeForce GTX 680	31.1×10^{-9}	5.6ms

En la Tabla 19 se observa que el enfoque que se desarrolló en este trabajo utilizando la tarjeta GeForce GTX 680 toma 5.6 ms en analizar 20^4 secuencias, 1.86 veces más rápido que el enfoque desarrollado por Garcia-Ordaz *et al.* (2012) utilizando el dispositivo Tesla C2070. También se muestra que la implementación secuencial propuesta en este trabajo, utilizando el procesador Intel i7-3820, es 16,546 veces más rápida que la versión que se desarrolló con el FPGA Vertex II Pro en el 2011.

Para hacer una comparación justa desde el punto de vista algorítmico se requeriría que cada enfoque se analizara bajo las mismas condiciones de software y hardware. También hay que tomar en cuenta que la implemetación de cada algoritmo se liga estrechamente a la arquitectura y características específicas del dispositivo que se utiliza con el objetivo de obtener el mejor rendimiento posible.

El enfoque general de la propuesta de Garcia-Ordaz *et al.* (2012), que se describe en la Sección 2.6, utiliza una estrategia parecida a la que se desarrolló en este trabajo. En ambos casos se analiza una secuencia por hilo (Figura 21), pero la configuración de hilos y bloques que se manda al dispositivo es distinta. Por la estrategia que se utilizó en el trabajo de Garcia-Ordaz *et al.* (2012) el máximo número de secuencias que se pueden analizar con una llamada al GPU es de 20^4. Por lo tanto, con este enfoque, analizar 20^9 secuencias requeriría llamar 20^5 veces al dispositivo, manteniendo una comunicación huésped-dispositivo activa.

En nuestro trabajo el límite lo impone la capacidad de la memoria global del Hardware o el máximo índice que se le puede asignar a un hilo. Este índice máximo se calcula como *No. de Bloques* $\times$ *No. de Hilos*. El valor de cada parámetro de esta expresión se da por la configuración particular de cada tarjeta. El máximo índice disponible con la tarjeta que se utilizó en este trabajo tomando en cuenta solo la dimensión 'x' es de $2^{31} \times 1024$.

La mayor diferencia entre las dos estrategias es el filtrado de casos positivos. Garcia-Ordaz *et al.* (2012) analizan las secuencias y regresa los resultados al huésped con los resultados positivos y negativos juntos. Esto se traduce en más llamadas al dispositivo y en operaciones extras de filtrado en el huésped, lo que a la postre acabaría con la ganancia obtenida por la paralelización. Esto se vuelve aún más crítico cuando se observa que más del 99 % de los casos son negativos.

5.5.3. 1 % de casos positivos

Los resultados de la Tabla 16 muestran un incremento gradual de resultados positivos conforme aumenta el número de secuencias que se analizan, sin embargo, estos casos positivos

representan menos del 1 % del total de secuencias de una longitud dada. Con esta perspectiva se podría conjeturar que las cuatro propiedades que se utilizan para filtrar SCAPS son efectivas, pues descartan el 99 % de casos que se analizan. Por otro lado, si se toma en cuenta que el 1 % de 20^9 secuencias da un total de 512 x 10^7 casos positivos, los cuales se deben sintetizar para comprobar si tienen o no la actividad deseada, entonces se observa que aun teniendo una herramienta computacional de alto desempeño para detectar SCAPs, nuevas propiedades deben ser incluidas para que esta arroje resultados viables de implementar y así lograr un impacto más directo en el diseño de péptidos.

Otro aspecto a considerar es el almacenamiento de las secuencias positivas. Guardar en disco duro el 1 % de casos positivos de 20^9 secuencias que se analizan equivale a 200 Gigabytes de información con la estructura de 48 bytes que se manejó en la implementación.

Para optimizar el espacio que se necesita se puede almacenar únicamente el identificador del hilo (Id), (como se muestra en la Sección 4.2.2). Este Id se puede traducir a una secuencia de aminoácidos fácilmente en el huésped con el proceso inverso del algoritmo de conversión a base 20 descrito en la Sección 4.3.5. El identificador del hilo es un dato de tipo "long long" de 8 bytes dentro del lenguaje C++, el cual genera 28 Gigabytes de datos con 20^9 secuencias que se analizan.

5.5.4. Análisis por región con más casos positivos

Gracias a la información que arrojó el análisis de 20^9 secuencias, separadas en bloques de 20^7, límite impuesto por el hardware, fue posible observar las regiones con 7 aminoácidos variables que tienen más casos positivos. La primera secuencia que se analiza es "AAAAAAAAA" y la última "YYYYYYYYY", cada región contiene 7 aminoácidos variables y dos fijos como el ejemplo siguiente: $A_1A_2XXXXXXX$.

La Tabla 20 muestra las primeras 20 regiones con más casos positivos con base en las propiedades que se analizan. La Arginina "R", aminoácido cargado positivamente, es el que más casos positivos produce, le sigue el otro aminoácido con carga positiva, Lisina "K" y el polar sin carga Tirosina "Y". La tabla se ordenó con base en el porcentaje de casos positivos.

Tabla 20: Secuencias de nueve aminoácidos con siete de ellos variables que generan más casos positivos.

Pos	Secuencia Inicial	Casos Positivos	Cálculos(min)
1	RKxxxxxxx	5.9 %	1.26
2	KRxxxxxxx	5.9 %	1.26
3	RYxxxxxxx	5.4 %	1.09
4	RRxxxxxxx	4.8 %	1.28
5	YRxxxxxxx	4.3 %	1.11
6	RFxxxxxxx	3.8 %	1.06
7	FRxxxxxxx	3.4 %	1.11
8	KYxxxxxxx	3.3 %	1.00
9	KFxxxxxxx	3.2 %	1.01
10	KKxxxxxxx	3.1 %	1.16
11	RLxxxxxxx	3.0 %	1.04
12	RMxxxxxxx	3.0 %	1.07
13	RIxxxxxxx	3.0 %	1.03
14	LRxxxxxxx	2.9 %	1.11
15	IRxxxxxxx	2.9 %	1.10
16	MRxxxxxxx	2.9 %	1.12
17	RNxxxxxxx	2.8 %	1.14
18	RQxxxxxxx	2.8 %	1.10
19	VRxxxxxxx	2.8 %	1.11
20	RWxxxxxxx	2.7 %	1.13

En la Tabla 21 se muestran las 10 regiones de secuencias de longitud 9 con menor porcentaje de casos positivos que se calcularon en grupos de 20^7. En estas aparecen los aminoácidos cargados negativamente, ácido Aspártico “D” y ácido glutámico “E”, además del aminoácido polar sin carga Cisteína “C”.

5.5.5. Permutaciones con más casos positivos

Además de analizar los casos positivos por región, se estudiaron las permutaciones de los candidatos a SCAPs, donde lo único que cambia es el orden de los aminoácidos que lo componen. De las cuatro propiedades que se analizan, i.e. solo el Momento Hidrofóbico Helicoidal toma en consideración la posición que ocupa cada aminoácido en la secuencia.

Tabla 21: Secuencias de nueve áminoacidos con siete de ellos variables que generan menos casos positivos

Pos	Secuencia Inicial	Casos Positivos	Cálculos(min)
391	CTxxxxxxx	0.0768 %	0.54
392	CDxxxxxxx	0.0070 %	0.37
393	DDxxxxxxx	0.0066 %	0.37
394	EDxxxxxxx	0.0065 %	0.35
395	CExxxxxxx	0.0065 %	0.37
396	DExxxxxxx	0.0061 %	0.35
397	EExxxxxxx	0.0060 %	0.36
398	DCxxxxxxx	0.0057 %	0.35
399	ECxxxxxxx	0.0056 %	0.35
400	CCxxxxxxx	0.0046 %	0.38

Para analizar las permutaciones se extendió la implementación del lado del dispositivo. A cada secuencia positiva se le asigna un índice por medio de una función Hash, esta función genera la llave o índice con base en los aminoácidos que conforman la cadena, los detalles de la implementación se encuentran en la Sección 4.5. Con este proceso, si dos secuencias están compuestas por los mismos aminoácidos pero en diferente orden entonces generarán la misma llave, es decir, se toman como la misma secuencia.

Tabla 22: Multiconjuntos de aminoácidos que genera el mayor número de permutaciones que resultan ser SCAPs.

Pos	Multiconjunto	Casos Positivos
1	{A, K, L, L, N, R, Y, T, V}	153251
2	{E, F, G, G, K, R, R, W, Y}	152934
3	{A, K, L, L, Q, R, Y, T, V}	152467
4	{E, F, F, G, G, K, R, R, W}	148183
5	{A, H, K, L, L, R, Y, V, T}	146344
6	{A, F, K, L, L, N, R, T, V}	145423
7	{A, F, K, L, L, Q, R, T, V}	145263
8	{A, F, G, H, K, N, Q, Y ,R}	142345
9	{F, G, H, K, N, Q, P, Y, R}	142160
10	{F, G, H, K, N, Q, S, Y ,R}	141341
11	{E, F, G, G, K, L, W, R, R}	140856
12	{A, F, G, K, N, P, Q, R, Y}	140425
13	{A, F, H, K, L, R, T, L, V}	140087
14	{A, F, H, K, N, P, Q, R, Y}	139859
15	{F, G, H, K, N, Q, T, Y, R}	139523

El multiconjunto de aminoácidos {A, K ,L ,L ,N ,R ,Y ,T ,V} es el que más permutaciones positivas genera, 153,251. El máximo número de permutaciones con una secuencia de 9 aminoácidos es 9!, es decir, 362,880 permutaciones. Con este multiconjunto el 42.2 % de las permutaciones de los péptidos resultaron ser candidatos a SCAPs.

Que una secuencia contenga un alto porcentaje de permutaciones positivas podría ser un indicativo para mostrar el comportamiento que se desea con base en las propiedades que se analizan.

En el estudio de las permutaciones al igual que en el análisis por región descrito en la Sección 5.5.4, la Arginina “R” sigue apareciendo en todos los casos pero con a lo más dos instancias, le siguen la Lisina “K”, Tirosona “Y” y la Treonina “T”.

5.5.6. Comparación con la base de datos CAMP

De los resultados positivos que se obtuvieron con el algoritmo paralelo, se seleccionó un pequeño subconjunto de 500 SCAPs para evaluarlos como Péptidos Antimicrobiales (AP por sus siglas en inglés) con la herramienta de predicción de CAMP (Collection of Anti-Microbial Peptides).

CAMP es una base de datos en línea (http://www.camp.bicnirrh.res.in/) del Centro Informático Biomédico (BIC por sus siglas en inglés) que colecciona péptidos con actividad conocida, de patentes y péptidos previstos con actividad antimicrobial que no se ha validado.

La herramienta recibe como entrada un archivo con las secuencias a analizar en formato FASTA. Esta herramienta de predicción incorpora tres algoritmos a la base de datos. Estos se basan en las estrategias de Máquina de Soporte Vectorial, Bosque Aleatorio, Redes Neuronales Artificiales y Análisis de Discriminantes (Thomas *et al.*, 2009).

El subconjunto de péptidos etiquetados como SCAPs por los criterios de (del Rio *et al.*, 2001) fueron evaluados con la herramienta de predicción de CAMP, 45 % de estos cumplieron con las propiedades de AP de los algoritmos utilizados. El porcentaje de predicción es bajo

si se considera que el conjunto de los SCAPs debe ser un subconjunto de los APs.

5.5.7. Análisis de 20^{10} secuencias

Dedicamos esta sección al análisis de secuencias de longitud 10 ya que, hasta donde tenemos conocimiento, es la primera vez que se hace una búsqueda exhaustiva sobre péptidos de hasta esta longitud.

En los trabajos de Polanco *et al.* (2011) y Garcia-Ordaz *et al.* (2012) se especifíca que se hace búsqueda exhaustiva de SCAPs en secuencias de hasta 9 aminoácidos.

La secuencia de análisis de Polanco *et al.* (2011) es RAAAYXXXX, con cinco aminoácidos fijos y 4 de estos variables, es decir se analizan 160,000 secuencias. En el trabajo con GPUs de Garcia-Ordaz *et al.* (2012), teóricamente es posible analizar un péptido de longitud 9; por la configuración de kernel que se desarrolló, solo se reporta el análisis con hasta 4 aminoácidos variables.

En este trabajo, la longitud máxima que se evaluó con la tarjeta GTX 680 es de 10 aminoácidos variables: 1.024×10^{13} secuencias en 4.5 días, donde el 1 % de estas resultaron SCAPs. Ya que la capacidad máxima de almacenamiento del dispositivo utilizado se rebasa cuando se almacenan los casos positivos con 20^8 secuencias evaluadas, fue necesario invocar al GPU en 20^3 ocasiones y con cada llamado se analizan 20^7 péptidos. Por cada conjunto de 20^7 secuencias se almacenan únicamente los primeros 20 casos positivos encontrados, ya que almacenar el 1 % de casos positivos de cada conjunto con el que se logran las 20^{10} secuencias requiere de un dispositivo con 1.5 Terabytes de memoria. Esta cálculo se obtiene de multiplicar el número de casos positivos por la estructura de 16 bytes necesaria para almacenar cada caso, como se muestra en la Sección 4.2.2.

El número de secuencias que se pueden evaluar en el dispositivo está limitado al índice máximo que se le puede asignar a un hilo, y éste está dado por el valor del entero máximo que puede asignar el compilador de C++, $2^{64}-1$, lo que equivale a evaluar más de 20^{14} secuencias.

Tabla 23: Principales características de los diferentes enfoques en el análisis de secuencias.

Año	Dispositivo	Secuencia de Análisis	L. Max	AA Variables	Filtrado
2011	Vertex II Pro	RAAAYXXXX	9	4	No
2012	Telsa C2070	AAAAAXXXX	9	9 (Teórico)	No
2013	GeForce GTX 680	XXXXXXXXXX	10	10	Sí

En la Tabla 23 se muestra un resumen comparativo entre las secuencias que se analizan en cada enfoque. La columna "L. Max" indica la longitud máxima evaluada, "AA Variables" muestra el número de aminoácidos variables dentro de la longitud máxima que se reportó, finalmente, en la columna "filtrado" se especifica si el enfoque filtra o no los resultados positivos.

5.6. Discusión

En esta sección se discutirán sobre los alcances de la estrategia propuesta y las principales diferencias entre este trabajo y los trabajos relacionados. También se incluye un breve análisis del porcentaje de casos positivos que arroja el algoritmo y de la posible relevancia de estos casos en otros enfoques.

5.6.1. Diferencia entre la estrategia propuesta y los trabajos relacionados

Debido a la estrategia que se diseñó, el número máximo de secuencias que se pueden evaluar por llamado al dispositivo lo impone la capacidad de almacenamiento de este y no el algoritmo como sucede en el trabajo de Garcia-Ordaz *et al.* (2012). Esta característica hace que migrar la solución a un dispositivo de diferente arquitectura (de mayor o menor revisión) o incluso hacia un clúster de GPUs sea posible sin cambiar la estructura principal del programa. La estrategia propuesta permite además mantener la comunicación entre el dispositivo y el huésped al mínimo.

Ya que el número máximo de secuencias a analizar lo impone el dispositivo, la pregunta que surgió fue ¿Cuál es el máximo número de secuencias que se pueden evaluar dentro de la GPU GTX680 de arquitectura Kepler (última generación al momento del desarrollo de esta tesis) en un tiempo razonable? El máximo que se reporta en los trabajos de Polanco *et al.* (2011); Garcia-Ordaz *et al.* (2012) es de hasta nueve aminoácidos con solo cuatro de estos

variables, es decir 160 mil (20^4) secuencias diferentes.

Los resultados muestran que es posible evaluar secuencias con hasta 10 aminoácidos variables (1.024×10^{13} secuencias) en 4.5 días, el máximo análisis exhaustivo que se ha realizado según nuestro conocimiento (ver Sección 5.5.7). El tiempo promedio por secuencia es de 31.1 $\times 10^{-9}$ como se muestra en la Tabla 23, este promedio es 165,000 veces más rápido que los 5.15×10^{-3} reportados por (Polanco *et al.*, 2011) en el análisis de péptidos de longitud nueve pero con apenas 4 aminoácidos variables utilizando FPGAs.

Con el tiempo promedio por secuencia podemos estimar el tiempo requerido para secuencias más largas; con 11 aminoácidos variables el tiempo estimado es de un poco menos de 3 meses (84.5 días) y para 12 es un poco más de 4 años y medio. Estos cálculos solo toman en cuenta el procesamiento dentro de la GPU. Los estimados muestran el crecimiento exponencial en el tiempo de cómputo al agregar aminoácidos a la secuencia, y por este crecimiento se rebasa rápidamente la capacidad de cómputo de un solo dispositivo, lo que naturalmente lleva a considerar una solución distribuida.

5.6.2. Filtrado de casos positivos

Una característica que separa significativamente al diseño de la solución propuesta de los enfoques de búsqueda exhaustiva citados es el filtrado en paralelo de los casos positivos. Sin este filtrado se almacenarían el 1 % de casos positivos y el 99 % de negativos en la memoria global del dispositivo para después enviarlos al huésped, lo que implica un mayor número de invocaciones al GPU y un mayor tiempo de ejecución. Además el CPU sería el encargado de filtrar de manera secuencial a los candidatos a SCAPs, lo que acabaría en gran medida con el esfuerzo de la paralelización.

Filtrar dentro del dispositivo requiere de un conjunto de operaciones específicas pues es necesario calcular el índice de cada candidato a SCAP de forma paralela. Este proceso se realiza con la técnica de suma de prefijos en conjunto con instrucciones de CUDA que aprovechan las características de la arquitectura. El tiempo requerido para la suma de prefijos es despreciable como se muestra en la Tabla 16, y gracias a este la comunicación entre el GPU

y el CPU se mantiene al mínimo. Otro proceso que se beneficia del filtrado es la escritura de resultados en archivo, pues este llega a ser hasta 1390 veces considerando solo las primeras 3,200,000 (20^5) secuencias evaluadas (Tabla 17).

Aunque el proceso de filtrado no es trivial, resulta necesario por la naturaleza del problema, sin embargo los trabajos de Polanco *et al.* (2011) y Garcia-Ordaz *et al.* (2012) carecen de este procedimiento lo que da a nuestro enfoque una gran ventaja.

5.6.3. Candidatos a SCAPs

Desde la perspectiva computacional, el tiempo de ejecución es uno de los parámetros más importantes a considerar, pero esto debe estar ligado a la calidad de los resultados para tener una solución útil.

El conjunto de candidatos a SCAPs es resultado del cálculo de las cuatro propiedades (descriptores) que fueron obtenidas de los trabajos de investigación de Ellerby *et al.* (1999); del Rio *et al.* (2001); Polanco *et al.* (2011). En la Tabla 5.5 se muestra que al analizar péptidos con 9 aminoácidos variables cerca del 1 % resultan ser candidatos a SCAPs, un número muy alto si se considera la posibilidad de sintetizarlos para corroborar su comportamiento.

En la literatura existen cientos de descriptores que podrían ser utilizados para lograr un filtrado más preciso en caso de ser posible, pero por otro lado, predictores como CAMP (Sección 5.5.6) respaldan en casi un 50 % los resultados obtenidos por la búsqueda exhaustiva, lo que demuestra la gran cantidad de secuencias que cumplen con las propiedades, que según trabajos anteriores definen a un SCAP.

¿Qué tan útil es entonces la búsqueda exhaustiva? El 1 % de candidatos a SCAP de este análisis puede ser utilizado como punto de partida para un nuevo estudio donde se detecten patrones y características sobre este conjunto que ayuden a mejorar la calidad de los resultados, un pequeño análisis de estos patrones se muestra en las Sección 5.5. Además, una pequeña parte de este conjunto se puede utilizar como parámetro de entrada para otros enfoques, por ejemplo, para entrenar una red neuronal o como los individuos de la población

inicial de un algoritmo genético.

Capítulo 6. Conclusiones

6.1. Sumario

En este trabajo se diseñó un algoritmo paralelo de búsqueda exhaustiva para detectar candidatos a Péptidos Catiónicos Antibacterianos Selectivos (SCAPs). La implementación del algoritmo se hizo sobre una unidad de procesamiento gráfico utilizando la plataforma y modelo de programación CUDA. El diseño incluye cuatro propiedades fisicoquímicas que según del Rio *et al.* (2001); Polanco *et al.* (2011) definen un SCAP. La búsqueda exhaustiva se hace, hasta donde sabemos por primera vez, en cadenas con hasta 10 aminoácidos variables (20^{10} secuencias). El algoritmo paralelo filtra los candidatos a SCAPs del resto de las secuencias analizadas utilizando la técnica “suma de prefijos” en conjunto con la instrucción “Suma Atómica” que provee CUDA. El objetivo del filtrado es mejorar el tiempo y espacio requeridos para la escritura en memoria de los candidatos. Se realizó también un análisis breve del multiconjunto de aminoácidos que produce más candidatos a SCAPs. Se analiza además los aminoácidos más frecuentes dentro de los candidatos a SCAPs.

6.2. Conclusiones

A continuación se enuncian las principales conclusiones de este proyecto de tesis.

1. Los resultados experimentales muestran que el algoritmo paralelo es 32 veces más rápido que su versión secuencial. Además, la propuesta de esta tesis supera en aproximadamente 165,000 veces el tiempo reportado en el trabajo de Polanco *et al.* (2011) utilizando FPGAs y es dos veces mejor que el de Garcia-Ordaz *et al.* (2012) usando GPUs, esto de acuerdo al tiempo registrado en el análisis de 20^4 secuencias. Sin embargo, debido a que el harwdware y algoritmos utilizados en cada trabajo son diferentes, la comparación se hace estrictamente en términos de tiempo. Una comparación justa en términos del algoritmo paralelo se da con la ley de Amdhal, como se presenta en este trabajo, aunque esta no se reporta en los trabajos antes mencionados.

2. Con las propiedades utilizadas se desechan un poco más del 99 % de los casos que se analizan, sin embargo, aun el 1 % de casos positivos restante representa un conjunto vasto de péptidos, haciendo inviable la comprobación experimental de todos ellos.

Resulta entonces necesario encontrar un conjunto más amplio de propiedades con características específicas para que el filtrado arroje un número reducido de candidatos a SCAPs.

3. El tiempo de ejecución requerido para filtrar los casos positivos dentro del GPU es despreciable, y es gracias a este filtrado que se tiene la capacidad de almacenar más SCAPs en la memoria global por cada llamada al GPU, lo que implica menos comunicación entre CPU y GPU y una mejora en el tiempo de procesamiento total. Sin el filtrado, la memoria global se llenaría rápidamente de casos negativos pues estos representan el 99 % del total, lo que significa un desperdicio de memoria y la necesidad de más llamados al GPU, además, no filtrar los casos negativos merma el esfuerzo de la parelelización pues sería necesario proveer esta funcionalidad dentro del huésped.

4. Con las propiedades evaluadas se observa que la Arginina "R", aminoácido con carga positiva, es el que más casos positivos produce, esto está de acorde con la propiedad catiónica de los SCAPs. A la Arginina le sigue el otro aminoácido con carga positiva, Lisina "K" y el polar sin carga Tirosina "Y".

5. En el estudio de multiconjuntos de aminoácidos con el mayor número de permutaciones que resultan casos positivos se tiene en primer lugar al multiconjunto {A, K, L, L, N, R, Y , T, V} con un 42 % de sus permutaciones siendo casos positivos. Analizando los 15 primeros multiconjuntos con esta propiedad puede observarse que todos ellos incluyen al siguiente conjunto de aminoácidos {R, K, Y, T, V, L}.

6. Al comparar un conjunto muestra de candidatos a SCAPs arrojados por el algoritmo paralelo con un predictor de CAMP, cerca del 50 % de estos cumplieron con las propiedades de los antimicrobianos evaluadas en dicha colección. Este porcentaje nos indica que las propiedades utilizadas en este trabajo arrojan resultados que pueden ser respaldados en gran medida con este tipo de herramientas disponibles en línea. Para sustentar esta hipótesis se tendrá que analizar ejemplos de casos negativos según el criterio de las cuatro propiedades.

6.3. Trabajo futuro

El algoritmo propuesto aprovecha las características del GPU que utilizamos en esta investigación (NVidia GeForce GTX 680), pero gracias a las instrucciones que provee CUDA en principio sería posible hacer una implementación que se adapte a cualquier GPU que utilice esta plataforma y crear con esto una solución genérica.

Es imperativa la búsqueda de un mayor número de propiedades que describan de forma específica a los SCAPs propuestos por Polanco *et al.* (2011). Las cuatro propiedades que aquí se implementan son las básicas que se mencionan en la literatura para definir a un péptido antimicrobiano en general y, por los resultados obtenidos, estos no son suficientes para un filtrado útil en la búsqueda exhaustiva. Además, en la literatura se menciona la existencia de un gran número de descriptores disponibles para intentar identificar a un péptido catiónico antimicrobiano (Hilpert *et al.*, 2008; Fjell *et al.*, 2011a) que pueden ser tomados en cuenta en trabajos posteriores.

En cuanto al conjunto de resultados arrojados por el algoritmo, es posible hacer un segundo análisis, estocástico o determinístico, sobre las secuencias positivas para intentar identificar un patrón en estas o agrupar familias de soluciones con miras a reducir el espacio de búsqueda en trabajos futuros.

El algoritmo propuesto se puede escalar para implementarlo en un clúster de GPUs y permitir el análsis exhaustivo de secuencias de mayor longitud en un menor tiempo. Este clúster se puede utilizar para crear un servicio web que provea al usuario el conjunto de péptidos que cumplan con las propiedades introducidas por él. Además el sistema debe tener la flexibilidad de añadir nuevas propiedades para que el usuario tenga la posibilidad de elegir cuántas y cuáles de estas desea analizar, creando con esto un sistema capaz de detectar péptidos con diferentes características además de los SCAPs aquí propuestos.

Lista de referencias

Anfinsen, C. B. *et al.* (1973). Principles that govern the folding of protein chains. *Science*, **181**(4096): 223–230.

A.Rage (2011). An introduction to modern gpu architecture. En: *Presentation at Parallel Computing with CUDA seminar*. NVIDA Corporation.

Belda, I., Llorà, X., y Giralt, E. (2006). Evolutionary algorithms and de novo peptide design. *Soft Computing*, **10**(4): 295–304.

Bessalle, R., Kapitkovsky, A., Gorea, A., Shalit, I., y Fridkin, M. (1990). All-d-magainin: chirality, antimicrobial activity and proteolytic resistance. *FEBS letters*, **274**(1): 151–155.

Blondelle, S. E. y Houghten, R. A. (1992). Design of model amphipathic peptides having potent antimicrobial activities. *Biochemistry*, **31**(50): 12688–12694.

Branden, C., Tooze, J., *et al.* (1991). *Introduction to protein structure*, Vol. 2. Garland New York.

Brogden, K. A. (2005). Antimicrobial peptides: pore formers or metabolic inhibitors in bacteria? *Nature Reviews Microbiology*, **3**(3): 238–250.

Brumfitt, W., Salton, M. R., y Hamilton-Miller, J. M. (2002). Nisin, alone and combined with peptidoglycan-modulating antibiotics: activity against methicillin-resistant staphylococcus aureus and vancomycin-resistant enterococci. *Journal of Antimicrobial Chemotherapy*, **50**(5): 731–734.

Chiappetta, M. (2013). Nvidia ceo unveils volta graphics, tegra mobile roadmap, and grid vca virtualized rendering systems.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., y Stein, C. (2009). *Introduction to Algorithms, Third Edition*, capítulo 11 Hash Tables, pp. 253–285. The MIT Press, tercera edición.

del Rio, G., Castro-Obregon, S., Rao, R., Ellerby, H. M., y Bredesen, D. E. (2001). Apap, a sequence-pattern recognition approach identifies substance p as a potential apoptotic peptide. *FEBS letters*, **494**(3): 213–219.

Dematté, L. y Prandi, D. (2010). Gpu computing for systems biology. *Briefings in Bioinformatics*, **11**(3): 323–333.

Eisenberg, D., Weiss, R. M., Terwilliger, T. C., y Wilcox, W. (1982). Hydrophobic moments and protein structure. En: *Faraday Symposia of the Chemical Society*, Vol. 17, pp. 109–120. Royal Society of Chemistry.

Eisenberg, D., Weiss, R. M., y Terwilliger, T. C. (1984). The hydrophobic moment detects periodicity in protein hydrophobicity. *Proceedings of the National Academy of Sciences*, **81**(1): 140–144.

Ellerby, H. M., Arap, W., Ellerby, L. M., Kain, R., Andrusiak, R., Del Rio, G., Krajewski, S., Lombardo, C. R., Rao, R., Ruoslahti, E., *et al.* (1999). Anti-cancer activity of targeted pro-apoptotic peptides. *Nature medicine*, **5**(9): 1032–1038.

Ellerby, H. M., Bredesen, D. E., Fujimura, S., y John, V. (2008). Hunter- killer peptide (hkp) for targeted therapy. *Journal of medicinal chemistry*, **51**(19): 5887–5892.

Fernandez, M. R. (2012). Nodes, sockets, cores and flops.

Fjell, C. D., Jenssen, H., Fries, P., Aich, P., Griebel, P., Hilpert, K., Hancock, R. E., y Cherkasov, A. (2008). Identification of novel host defense peptides and the absence of α-defensins in the bovine genome. *Proteins: Structure, Function, and Bioinformatics*, **73**(2): 420–430.

Fjell, C. D., Jenssen, H., Hilpert, K., Cheung, W. A., Pante, N., Hancock, R. E., y Cherkasov, A. (2009). Identification of novel antibacterial peptides by chemoinformatics and machine learning. *Journal of medicinal chemistry*, **52**(7): 2006–2015.

Fjell, C. D., Hiss, J. A., Hancock, R. E., y Schneider, G. (2011a). Designing antimicrobial peptides: form follows function. *Nature reviews Drug discovery*, **11**(1): 37–51.

Fjell, C. D., Jenssen, H., Cheung, W. A., Hancock, R. E., y Cherkasov, A. (2011b). Optimization of antibacterial peptides by genetic algorithms and cheminformatics. *Chemical biology & drug design*, **77**(1): 48–56.

Garcia-Ordaz, D., Arias-Estrada, M., Nu no-Maganda, M., Polanco, C., y Del Rio, G. (2012). Acceleration of selective cationic antibacterial peptides computation: A comparison of fpga and gpu approaches.

Gronwald, W., Hohm, T., y Hoffmann, D. (2008). Evolutionary pareto-optimization of stably folding peptides. *BMC bioinformatics*, **9**(1): 109.

Hallock, K. J., Lee, D.-K., y Ramamoorthy, A. (2003). Msi-78, an analogue of the magainin antimicrobial peptides, disrupts lipid bilayer structure via positive curvature strain. *Biophysical journal*, **84**(5): 3052–3060.

Hancock, R. E. y Rozek, A. (2002). Role of membranes in the activities of antimicrobial cationic peptides. *FEMS Microbiology Letters*, **206**(2): 143–149.

Hancock, R. E. y Sahl, H.-G. (2006). Antimicrobial and host-defense peptides as new anti-infective therapeutic strategies. *Nature biotechnology*, **24**(12): 1551–1557.

He, K., Ludtke, S. J., Worcester, D. L., y Huang, H. W. (1996). Neutron scattering in the plane of membranes: structure of alamethicin pores. *Biophysical journal*, **70**(6): 2659–2666.

Henzler Wildman, K. A., Lee, D.-K., y Ramamoorthy, A. (2003). Mechanism of lipid bilayer disruption by the human antimicrobial peptide, ll-37. *Biochemistry*, **42**(21): 6545–6558.

Hill, M. D. y Marty, M. R. (2008). Amdahl's law in the multicore era. *IEEE Computer*, **41**(7): 33–38.

Hilpert, K., Fjell, C., y Cherkasov, A. (2008). Short linear cationic antimicrobial peptides: Screening, optimizing, and prediction. En: L. Otvos (ed.), *Peptide-Based Drug Design*, Vol. 494 de *Methods In Molecular Biology* ^{TM}, pp. 127–159. Humana Press.

Hiss, J. A., Bredenbeck, A., Losch, F. O., Wrede, P., Walden, P., y Schneider, G. (2007). Design of mhc i stabilizing peptides by agent-based exploration of sequence space. *Protein Engineering Design and Selection*, **20**(3): 99–108.

Hohm, T. y Hoffmann, D. (2005). A multi-objective evolutionary approach to peptide structure redesign and stabilization. En: *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pp. 423–429, New York, NY, USA. ACM.

JaJa, J. (1992a). *An Introduction to Parallel Algorithms*, capítulo List and Trees, pp. 43–51. Addison-Wesley, primera edición.

JaJa, J. (1992b). *An Introduction to Parallel Algorithms*, capítulo Introduction, pp. 1–4. Addison-Wesley, primera edición.

Jenssen, H., Hamill, P., y Hancock, R. E. (2006). Peptide antimicrobial agents. *Clinical microbiology reviews*, **19**(3): 491–511.

Jost, T., Contassot-Vivier, S., y Vialle, S. (2010). An efficient multi-algorithms sparse linear solver for gpus. En: B. Chapman, F. Desprez, G. Joubert, A. Lichnewsky, F. Peters, y T. Priol (eds.), *Parallel Computing: From Multicores and GPU's to Petascale*, Vol. 19 de *Advances in Parallel Computing*, pp. 546–553. IOS Press.

Kirk, D. B. y Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., primera edición. San Francisco, CA, USA.

Klepeis, J., Floudas, C., Morikis, D., Tsokos, C., y Lambris, J. (2004). Design of peptide analogues with improved activity using a novel de novo protein design approach. *Industrial & engineering chemistry research*, **43**(14): 3817–3826.

Kolonin, M. G., Saha, P. K., Chan, L., Pasqualini, R., y Arap, W. (2004). Reversal of obesity by targeted ablation of adipose tissue. *Nature medicine*, **10**(6): 625–632.

Kragol, G., Lovas, S., Varadi, G., Condie, B. A., Hoffmann, R., y Otvos, L. (2001). The antibacterial peptide pyrrhocoricin inhibits the atpase actions of dnak and prevents chaperone-assisted protein folding. *Biochemistry*, **40**(10): 3016–3026.

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., y Dubey, P. (2010). Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, **38**(3): 451–460.

Loose, C., Jensen, K., Rigoutsos, I., y Stephanopoulos, G. (2006). A linguistic model for the rational design of antimicrobial peptides. *Nature*, **443**(7113): 867–869.

Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M., y Buck, I. (2006). Gpgpu: General-purpose computation on graphics hardware.

Matsuzaki, K., Murase, O., Fujii, N., y Miyajima, K. (1996). An antimicrobial peptide, magainin 2, induced rapid flip-flop of phospholipids coupled with pore formation and peptide translocation. *Biochemistry*, **35**(35): 11361–11368.

Mor, Filipo Novo, d. A. N. P. (2013). Gpu programming with cuda. A brief overview.

Munoz, V. y Serrano, L. (1997). Development of the multiple sequence approximation within the agadir model of α-helix formation: Comparison with zimm-bragg and lifson-roig formalisms. *Biopolymers*, **41**(5): 495–509.

NVIDIA, C. (2008). NVIDIA CUDA Programming Guide 5.0. Reporte técnico, NVIDIA Corporation.

NVIDIA, C. (2010). Geforce 256 product overview.

NVIDIA, C. (2011). *CUDA C Best Practices Guide*. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara 95050, USA, cuarta edición.

NVIDIA, C. (2012). Nvidia geforce gtx 680. the fastest, most efficient gpu ever built. Reporte técnico, NVIDIA Corporation.

Oren, E. E., Tamerler, C., Sahin, D., Hnilova, M., Seker, U. O. S., Sarikaya, M., y Samudrala, R. (2007). A novel knowledge-based approach to design inorganic-binding peptides. *Bioinformatics*, **23**(21): 2816–2822.

Oren, Z. y Shai, Y. (1998). Mode of action of linear amphipathic α-helical antimicrobial peptides. *Peptide Science*, **47**(6): 451–463.

Otvos, L., O, I., Rogers, M. E., Consolvo, P. J., Condie, B. A., Lovas, S., Bulet, P., y Blaszczyk-Thurin, M. (2000). Interaction between heat shock proteins and antimicrobial peptides. *Biochemistry*, **39**(46): 14150–14159.

Park, C. B., Kim, H. S., y Kim, S. C. (1998). Mechanism of action of the antimicrobial peptide buforin ii: buforin ii kills microorganisms by penetrating the cell membrane and inhibiting cellular functions. *Biochemical and biophysical research communications*, **244**(1): 253–257.

Patel, H. (2010). Gpu accelerated real time polarimetric image processing through the use of cuda. En: *Aerospace and Electronics Conference (NAECON), Proceedings of the IEEE 2010 National*, pp. 177–180.

Patrzykat, A., Friedrich, C. L., Zhang, L., Mendoza, V., y Hancock, R. E. (2002). Sublethal concentrations of pleurocidin-derived antimicrobial peptides inhibit macromolecular synthesis in escherichia coli. *Antimicrobial agents and chemotherapy*, **46**(3): 605–614.

Perkins, R., Fang, H., Tong, W., y Welsh, W. J. (2003). Quantitative structure-activity relationship methods: Perspectives on drug discovery and toxicology. *Environmental Toxicology and Chemistry*, **22**(8): 1666–1679.

Polanco, C., Maganda, M. A. N., Arias-Estrada, M., y Del Rio, G. (2011). An FPGA implementation to detect selective cationic antibacterial peptides. *PloS one*, **6**(6): e21399.

Sanders, J. y Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, primera edición.

Schneider, G. y Böhm, H.-J. (2002). Virtual screening and fast automated docking methods. *Drug Discovery Today*, **7**(1): 64–70.

Schneider, G. y Wrede, P. (1998). Artificial neural networks for computer-based molecular design. *Progress in biophysics and molecular biology*, **70**(3): 175–222.

Schneider, G., Schrödl, W., Wallukat, G., Müller, J., Nissen, E., Rönspeck, W., Wrede, P., y Kunze, R. (1998). Peptide design by artificial neural networks and computer-based evolutionary search. *Proceedings of the National Academy of Sciences*, **95**(21): 12179–12184.

Spaar, A., Münster, C., y Salditt, T. (2004). Conformation of peptides in lipid membranes studied by x-ray grazing incidence scattering. *Biophysical journal*, **87**(1): 396–407.

Stenlund, P., Lindberg, M. J., y Tibell, L. A. (2002). Structural requirements for high-affinity heparin binding: alanine scanning analysis of charged residues in the c-terminal domain of human extracellular superoxide dismutase. *Biochemistry*, **41**(9): 3168–3175.

Thomas, S., Karnik, S., Barai, R. S., Jayaraman, V. K., y Idicula-Thomas, S. (2009). Camp: a useful resource for research on antimicrobial peptides. *Nucleic acids research*, p. gkp1021.

Walters, W. y Goldman, B. (2005). Feature selection in quantitative structure-activity relationships. *Current opinion in drug discovery & development*, **8**(3): 329–333.

Wang, G. (2010). *Antimicrobial peptides: discovery, design and novel therapeutic strategies*, Vol. 18. CABI.

Weaver, D. C. (2004). Applying data mining techniques to library design, lead generation and lead optimization. *Current opinion in chemical biology*, **8**(3): 264–270.

Weisstein, E. W. (2010). *CRC concise encyclopedia of mathematics*. CRC press, tercera edición.

Yagi, Y., Terada, K., Noma, T., Ikebukuro, K., y Sode, K. (2007). In silico panning for a non-competitive peptide inhibitor. *BMC bioinformatics*, **8**(1): 11.

Zasloff, M. (2002). Antimicrobial peptides of multicellular organisms. *Nature*, **415**(6870): 389–395.

Zhang, L., Rozek, A., y Hancock, R. E. (2001). Interaction of cationic antimicrobial peptides with model membranes. *Journal of Biological Chemistry*, **276**(38): 35714–35722.

Apéndice

A.1. Comparación de GPUs de diferente generación

La siguiente tabla proporciona una comparación de alto nivel entre la tarjeta GTX 680 de arquitectura Kepler y generaciones previas de GPUs de NVIDIA.

Tabla 24: Características generales de tarjetas de las arquitecturas Tesla, Fermi y Kepler.

GPU	**GT200 (Tesla)**	**GF110 (Fermi)**	**GTX 680 (Kepler)**
Especificaciones del motor			
Núcleos CUDA	240	512	1536
Reloj del núcleo gráfico	648 MHz	772 MHz	1006 MHz
Tasa de llenado de texturas	51.8 GB/sec	49.4 GB/sec	128.8 GB/sec
Especificaciones de memoria			
Velocidad de la memoria	2484 MHz	4008 MHz	6008 MHz
Capacidad de memoria	1024 MB	1536 MB	2048 MB
Ancho de banda de la memoria	141.7 GB/sec	192.4 GB/sec	192.2 GB/sec
Soporte de monitores			
Multi-monitor	2	2	4
Máxima resolución digital	2560x1600	2560x1600	4096x2160
Máxima resolución VGA	2048x1536	2048x1536	2048x1536
Especificaciones estándares			
Transistores	1.4 mil millones	3.0 mil millones	3.54 mil millones
GFLOPs	1063	1581	3090
Potencia de diseño térmico	183W	244W	195W

A.2. Comparación C2070 vs GTX 680

La siguiente tabla proporciona una comparación de alto nivel entre la GPU Tesla C2070 utilizado en el trabajo de Garcia-Ordaz *et al.* (2012) y la GPU GeForce GTX 680 empleado en este trabajo.

A.3. Características y especificaciones técnicas de las GPUs en función de su capacidad de cómputo

Tabla 25: Especificaciones técnicas principales de las GPUs Tesla C2070 y GeForce GTX 680.

Especificaciones estándares	**Tesla C2070**	**GeForce GTX 680**
Capacidad de cómputo	2.0	3.0
Número de núcleos	448	1536
Velocidad de reloj de los núcleos	1.15 GHz	1.0 GHz
Velocidad del reloj de memoria	1.50 GHz	6.0 GHz
Tamaño de la memoria	6 GB	2 GB
Ancho de banda de la memoria	144 GB/seg	192.2 GB/seg

Tabla 26: Especificaciones técnicas de las GPUs en función de su capacidad de cómputo.

<table>
<tr><th rowspan="2">Especificaciones técnicas</th><th colspan="8">Capacidad de cómputo (versión)</th></tr>
<tr><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th><th>3.0</th><th>3.5</th><th>5.0</th></tr>
<tr><td>Dimensión máxima del grid de bloques</td><td colspan="4">2</td><td colspan="4">3</td></tr>
<tr><td>Máxima dimensión en x, y o x de un grid de bloques</td><td colspan="4">65535</td><td colspan="4">$2^{31} - 1$</td></tr>
<tr><td>Máxima dimensión de un bloque de hilos</td><td colspan="8">3</td></tr>
<tr><td>Máxima dimensión en x o y por bloque</td><td colspan="4">512</td><td colspan="4">1024</td></tr>
<tr><td>Máxima dimensión en z por bloque</td><td colspan="8">64</td></tr>
<tr><td>Máximo número de hilos por bloque</td><td colspan="4">512</td><td colspan="4">1024</td></tr>
<tr><td>Tamaño del warp</td><td colspan="8">32</td></tr>
<tr><td>Número máximo de bloques residentes por multiprocesador</td><td colspan="5">8</td><td colspan="2">16</td><td>32</td></tr>
<tr><td>Número máximo de warps residentes por multiprocesador</td><td colspan="2">24</td><td colspan="2">32</td><td>48</td><td colspan="3">64</td></tr>
<tr><td>Número máximo de hilos residentes por multiprocesador</td><td colspan="2">768</td><td colspan="2">1024</td><td>1536</td><td colspan="3">2048</td></tr>
<tr><td>Número de registros de 32 bits por multiprocesador</td><td colspan="2">8 K</td><td colspan="2">16 K</td><td>32 K</td><td colspan="3">64 K</td></tr>
<tr><td>Número máximo de registros de 32 bits por hilo</td><td colspan="4">128</td><td>63</td><td colspan="3">255</td></tr>
<tr><td>Canitad máxima de memoria compartida por multiprocesador</td><td colspan="4">16 KB</td><td colspan="3">48 KB</td><td>64 KB</td></tr>
<tr><td>Número de bancos de memoria compartida</td><td colspan="4">16</td><td colspan="4">32</td></tr>
<tr><td>Cantidad de memoria compartida por hilo</td><td colspan="4">16 KB</td><td colspan="4">512 KB</td></tr>
<tr><td>Tamaño de la memoria constante</td><td colspan="8">64 KB</td></tr>
<tr><td>Número máximo de instrucciones por kernel</td><td colspan="4">2 millones</td><td colspan="4">512 millones</td></tr>
</table>

Tabla 27: Características ofrecidas por la GPU en función de su capacidad de cómputo.

<table>
<tr><th rowspan="2">Lista de soporte de características</th><th colspan="8">Capacidad de cómputo (versión)</th></tr>
<tr><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th><th>3.0</th><th>3.5</th><th>5.0</th></tr>
<tr><td>Funciones atómicas con enteros que operan en palabras de 32 bits en la memoria global</td><td rowspan="2">No</td><td colspan="7" rowspan="2">Si</td></tr>
<tr><td>atomicExch() que operan con valores de punto flotante de 32 bits en la memoria global</td></tr>
<tr><td>Funciones atómicas con enteros que operan en palabras de 32 bits en la memoria compartida</td><td colspan="2" rowspan="3">No</td><td colspan="6" rowspan="3">Si</td></tr>
<tr><td>atomicExch() que funcionan con valores de punto flotante de 32 bits en la memoria compartida</td></tr>
<tr><td>Funciones atómicas con enteros que operan en palabras de 64 bits en la memoria global</td></tr>
<tr><td>Operaciones de punto flotante con doble precisión</td><td colspan="3">No</td><td colspan="5">Si</td></tr>
<tr><td>Funciones atómicas que operan con valores enteros de 64 bits en la memoria compartida</td><td colspan="4" rowspan="4">No</td><td colspan="4" rowspan="4">Si</td></tr>
<tr><td>Suma atómica de punto flotante que opera en palabras de 32 bits en la memoria global y compartida</td></tr>
<tr><td>_syncthreads_count(), _syncthreads_and(), _syncthreads_or()</td></tr>
<tr><td>Grid 3D de bloques de hilos</td></tr>
<tr><td>Funciones de intercambio de valores</td><td colspan="5">No</td><td colspan="3">Si</td></tr>
<tr><td>Paralelismo dinámico</td><td colspan="6">No</td><td colspan="2">Si</td></tr>
</table>