Universitat Autònoma de Barcelona

escola d'enginyeria

2817 - OPTIMITZACIÓ D'UNA APLICACIÓ BIOINFORMÀTICA D'ALINIAMENT DE SEQÜÈNCIES EXECUTADA EN PROCESSADORS MANY-CORE (GPUs)

Memòria del Projecte Fi de Carrera
d'Enginyeria en Informàtica
realitzat per
Alejandro Chacón de San Baldomero
i dirigit per
Juan Carlos Moure Lopez
Bellaterra, 12 de setembre del 2011

Universitat Autònoma de Barcelona

etse
Escola Tècnica Superior d'Enginyeria

El sotasignat, **Juan Carlos Moure López**

Professor/a de l'Escola Tècnica Superior d'Enginyeria de la UAB,

CERTIFICA:

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en **Alejandro Chacón de San Baldomero.**

I per tal que consti firma la present.

Signat:

Bellaterra, 13 de setembre del 2011

A mi familia y a mi novia

AGRADECIMIENTOS

Primero, quisiera a agradecer a mi tutor, Juan Carlos Moure, sin el que el siguiente trabajo no hubiera sido posible, por su paciencia, sus consejos en esas intensas sesiones de seguimiento en el proyecto. También quisiera mencionar a los profesores Porfidio y a Toni por su apoyo incondicional.

A mi familia, que me ha apoyado durante todos estos meses y a los que le debo todo. Finalmente, quisiera agradecer a Pedro Erencia y a mi novia por su ayuda, leyendo y comentando esta memoria para lograr mejorarla.

Dado que la presentación de este proyecto supone el fin de la carrera, no quisiera dejar de agradecer a mis compañeros de curso por compartir con ellos todos estos años y ser un gran apoyo para mi.

ÍNDICE GENERAL

1- Introducción 1

1.1 Resumen 1

1.2 Estado del Arte de la Bioinformática 1

1.3 Cómputo GPGPU 3

1.4 Objetivos 4

1.5 Motivaciones 4

1.6 Resumen de la memoria 4

1.7 Planificación y Cost 5

2- Conceptos teóricos: bioinformática 7

2.1 Resumen 7

2.2 Conceptos Biológicos 7

2.3 Motivaciones del estudio de los genomas 9

2.4 Métodos existentes en la identificación de los genes 9

2.4.1 Método Cross-species Comparison 11

2.5 Alineamiento entre secuencias 12

2.6 Algoritmos para el problema Whole Genome Alignment 13

2.6.1 Métodos de alineamiento exactos 14

2.6.2 Métodos de alineamiento aproximados 15

2.6 Puntos clave del capítulo 17

3- Modelo de programación de CUDA y arquitecturas ManyCore 19

3.1 Resumen 19

3.2 Estado del arte en arquitecturas MultiCore y ManyCore 19

3.3 Diferencias entre arquitecturas MultiCore y ManyCore 20

3.4 CUDA: Compute Unified Device Architecture 21

3.5 Visión general de la arquitectura GPU Fermi 22

3.6 Modelo de ejecución en GPU 23

3.7 Modelo de paralelismo a nivel de threads 25

3.8 Jerarquía de memoria 26
3.9 Principales factores críticos en el rendimiento 28
3.9.1 Acceso a memoria de forma coalesced 28
3.9.2 Shared memory 29
3.9.3 Warps y colaboración entre threads........ 29
3.9.4 Divergencia 30
3.10 Puntos clave del capítulo 30

4- Aplicación MUMmer en arquitecturas MultiCore y ManyCore 33
4.1 Resumen 33
4.2 Definición y objetivos de la aplicación MUMmer 33
4.3 Procedimiento del algoritmo MUMmer para la resolución de la problemática WGA 34
4.4 Etapa de *Matching*: la búsqueda de coincidencias exactas 35
4.4.1 Representación de los datos 36
4.4.2 Concepto de Maximal Match en MUMmer 37
4.4.3 Proceso de String Matching en arquitectura MultiCore 38
4.4.4 Diferencias entre MUMmer1 , MUMmer2 y MUMmer3 38
4.4.5 Proceso de String Matching en arquitecturas ManyCore 39
4.4.6 Diferencias entre la implementación de *MUMmerGPU* y *MUMmerGPU++* 42
4.5 Puntos clave del capítulo 43

5- Estructuras de datos para indexación de textos 45
5.1 Resumen 45
5.2 Motivaciones de las estructuras de datos para la indexación de secuencias genómicas. 45
5.3 Suffix-Tree 46
5.3.1 String matching con Suffix-Tree 47
5.3.3 Suffix-Links 48
5.3.4 Complejidades del Suffix-Tree y Suffix-Links 50
5.4 Suffix-Array 51
5.4.1 String matching con Suffix-Array 52
5.4.2 String matching con Suffix-Array utilizando el LCP 55
5.4.3 Complejidades del Suffix-Array y su construcción 56
5.5 Enhanced Suffix-Array 56
5.5.1 String matching con Enhanced Suffix-Array 58

5.5.2 Complejidades del Enhanced Suffix-Array. 60
5.6 Indexación de textos en MUMmer: Búsqueda de nonUnique MEMs 61
5.7 Puntos clave del capítulo 63

6- Evaluación experimental de las aplicaciones 65
6.1 Resumen 65
6.2 Metodología utilizada en la adquisición de datos 65
6.3 Entorno y sistema de experimentación 66
6.4 Herramientas 67
6.5 Carga de trabajo utilizada 67
6.6 Parámetros de ejecución 69
6.7 Proceso de compilación 69
6.8 Proceso de ejecución 70
6.9 Métricas utilizadas 71
6.10 Resultados de la experimentación 72
6.10.2 Pruebas detalladas sobre MUMmerGPUpp 78
6.10.2 Pruebas detalladas sobre el ancho de banda 84
6.10.2 Pruebas detalladas sobre el impacto del acceso a disco en la aplicación 85
6.10.5 Pruebas comparativas entre MUMmerGPU++ y MUMmerGPU 86
6.10.6 Pruebas específicas de la arquitectura 88
6.11 Puntos clave del capítulo 90

7- Conclusiones y líneas abiertas 91
7.1 Líneas abiertas 91
7.2 Problemas encontrados 92
7.3 Valoración personal 93
7.4 Desarrollo del proyecto 93

Bibliografía 95

ÍNDICE DE IMÁGENES

Capítulo 1:

Figura 1.1- Número de secuenciaciones de genomas completos de GenBank. Recuento de los proyectos activos y finalizados de GOLD.

Figura 1.2- Incremento de la obtención de bases genómicas con el avance de las tecnologías de secuenciación. Relación coste de la secuenciación-coste del cómputo en la última década.

Figura 1.3- Comparativa de potencia de cómputo pico en GLOPS de simple precisión entre MultiCores y ManyCores. Comparativa de ancho de banda pico en GB/s MultiCores y ManyCores

Capítulo 2:

Figura 2.1- Esquema del segmento de una molécula de DNA.

Figura 2.2- Representación de los 25 primeros pbs (pares de bases) del cromosoma 22 del Homo sapiens

Figura 2.3- Esquema conceptual del DNA de un espécimen

Figura 2.4 – Esquema conceptual de las regiones que puede contener una secuencia

Figura 2.5: Predicción de la Ley de Moore en el tiempo

Figura 2.6- Correspondencia de similitud entre regiones génicas en 2 genomas de diferentes especies.

Figura 2.7- Representación matricial de 3 alineamientos entre secuencias A y B.

Figura 2.8 – Comparación de regiones en dos genomas de entrada A y B

Figura 2.9 – Cambios en los genomas que analiza el WGA

Figura 2.10 – Modelo de las 3 principales etapas presentas en la mayoría de algoritmos para el WGA

Figura 2.11 – Muestra como la tercera etapa agrupa las subcadenas encontradas en clústeres

Figura 2.12 – Esquema y organización de los temas más importantes tratados en el capítulo.

Capítulo 3:

Figura 3.1: Diagrama de ocupación de los recursos en el área del chip CPU y GPU.

Figura 3.2: Diagrama de la arquitectura CUDA Fermi.

Figura 3.3: Ejemplo del modelo de ejecución de un programa con 4 etapas que es paralelizado en CUDA.

Figura 3.4: Ejemplo del modelo de ejecución de un CUDA kernel y la sincronización que realiza la CPU y GPU.

Figura 3.5: Modelo jerárquico del paralelismo a nivel de thread.

Figura 3.6: Jerarquía de memoria de CUDA

Figura 3.7: Acceso coalesced, unificación de las peticiones a memoria consecutivas de un conjunto de threads.

Figura 3.8: Etapa de planificación y lanzamiento de las instrucciones de los warps en los recursos del SM.

Figura 3.9: Efecto de serialización debido a la divergencia en la ejecución de una estructura condicional

Capítulo 4:

Figura 4.1: Se muestra la secuencia de referencia acaaacatat$1 y su Suffix-Tree.

Figura 4.2: Ejemplo de match y maximal match entre las secuencia referencia S=ctatatc con la query T=gtatatta

Figura 4.3: Ejemplo de los 2 ficheros de entrada (izquierda) y el salida (derecha) que utiliza MUMmer.

Figura 4.4: Ejemplo de división del genoma B en un conjunto de queries Q.

Figura 4.5: Ejemplo de los diferentes maximal match división del genoma B en un conjunto de queries Q.

Figura 4.6: Ejemplo de búsqueda de todos los MEMs en una referencia S=acaaacatat de una query T=acatata con minmatch=3.

Figura 4.7: Diferentes etapas de procesado que presentan las implementaciones MUMmerGPU y MUMmerGPU++

Figura 4.8: Pseudocódigo del proceso de string matching que se ejecuta en GPU en rounds.

Capítulo 5:

Figura 5.1: Se muestra la secuencia de referencia acaaacatat$ y su Suffix-Tree.

Figura 5.2: Proceso de búsqueda de la query T0 acatata$ en el Suffix-Tree de la referencia S acaaaacatat$

Figura 5.3: Suffix-Tree de la referencia S acaaaacatat$ con los Suffix-Links añadidos.

Figura 5.4: Uso de los Suffix-Links en el proceso de búsqueda de la query T1 catata$ en el Suffix-Tree de la referencia S acaaaacatat$ después de realizar la búsqueda de T0 acatata$

Figura 5.5: Generación de la estructura del Suffix-Array a partir de la secuencia de referencia S acaaaacatat$.

Figura 5.6: La imagen muestra la correlación entre los nodos hijos del Suffix-Tree y los elementos del array del Suffix-Array

Figura 5.7: Pseudocodigo del proceso de string matching utilizando una búsqueda binaria del Suffix-Array

Figura 5.8: Muestra el proceso de string matching utilizando una estructura de indexación Suffix-Array mediante una búsqueda binaria de los sufijos

Figura 5.9: Pseudocódigo del proceso de string matching utilizando una búsqueda binaria en el Suffix-Array aprovechando las propiedades del LCP internal tree.

Figura 5.10: Ejemplo de ESA a partir de la referencia S acaaacatat$, a la derecha se representa la estructura que simula el LCP-Array

Figura 5.11: Rank-Array de la estructura ESA.

Figura 5.12: Ejemplo de String Matching en ESA.

Figura 5.13: Pseudocódigo del proceso de string matching utilizando Enhanced Suffix-Arrays

Figura 5.14: Pseudocódigo de una parte del proceso de string matching en ESA utilizando en la fase 3 del proceso.

Figura 5.15: Ejemplo en el post procesado, realizando una búsqueda Bottom-Up de MEMs en el Suffix-Tree

Figura 5.16: Postprocesado en el proceso de búsqueda de coincidencias en estructuras ESA. Proceso de búsqueda de MEMs

Figura 5.17: Pseudocódigo del proceso de búsqueda de MEMs en las estructuras de datos ESA.

Capítulo 6:

Figura 6.1: Pseudocódigo de los scripts de lanzamiento de los workloads y mediciones

Figura 6.2: Gráfica que muestra el tiempo de procesado de MUMmer, MUMmerGPU y MUMmerGPU++ para el workload A

Figura 6.3: Rendimiento de MUMmer, MUMmerGPU y MUMmerGPU++ para el workloadA.

Figura 6.4: Rendimiento de MUMmer, MUMmerGPU y MUMmerGPU++ para el workload B.

Figura 6.5: Rendimiento de MUMmer, MUMmerGPU y MUMmerGPU++ para el workload C0.

Figura 6.6: Rendimiento de MUMmer, MUMmerGPU y MUMmerGPU++ para el workload C0 y C1.

Figura 6.7: Rendimiento de MUMmer, MUMmerGPU y MUMmerGPU++ para el workload D, que incrementa el tamaño de la referencia.

Figura 6.8: Rendimiento de MUMmer, MUMmerGPU y MUMmerGPU++ para el workload D, que incrementa el tamaño mínimo del match.

Figura 6.9: Porcentaje de ejecución década una de las fases en MUMmerGPU++ para el workload A.

Figura 6.10: Tiempo de ejecución de cada fase normalizado, para la ejecución del workload A con MUMmerGPU++.

Figura 6.11: Porcentaje de ejecución de cada una de las fases de MUMmerGPU++ para el workload B.

Figura 6.12: Tiempo de ejecución de cada fase normalizado, para la ejecución del workload B con MUMmerGPU++.

Figura 6.13: Porcentaje de ejecución de cada una de las fases de MUMmerGPU++ para el workload C0.

Figura 6.14: Tiempo de ejecución de cada fase normalizado, para la ejecución del workload C0 con MUMmerGPU++.

Figura 6.15: Porcentaje de ejecución de cada una de las fases de MUMmerGPU++ para el workload D.

Figura 6.16: Tiempo de ejecución de cada fase normalizado, para la ejecución del workload D con MUMmerGPU++.

Figura 6.17: Porcentaje de ejecución de cada una de las fases de MUMmerGPU++ para el workload E.

Figura 6.18: Tiempo de ejecución de cada fase normalizado, para la ejecución del workload E con MUMmerGPU++.

Figura 6.19: Ancho de banda en MB/s con el workload A en la fase de lectura de queries.

Figura 6.20: Ancho de banda en MB/s con el workload A en la fase de lectura de queries, previamente siendo cacheada.

Figura 6.21: Rendimiento de MUMmerGPU++ con el workload C0 y C1 con datos cacheados y sin cachear.

Figura 6.22: Porcentaje de ejecución de cada una de las fases de MUMmerGPU++ para el workload A.

Figura 6.23: Porcentaje de ejecución de cada una de las fases de MUMmerGPU++ para el workload A.

Figura 6.24: Rendimiento de la aplicación MUMmerGPU++ y sus distintas optimizaciones específicas de la arquitectura GPU Fermi con el workload A.

Figura 6.25: Speedup de la aplicación MUMmerGPU++ y sus distintas optimizaciones específicas de la arquitectura GPU Fermi.

GLOSARIO DE ACRÓNIMOS

GPGPU: General-Purpose Computing on Graphics Processing Units

WGA: Whole Genome Alignment

GPUs: Graphics Processing Units

APU: Accelerated Processing Unit

CUDA: Compute Unified Device Architecture

DNA : DeoxyriboNucleic Acid

RNA: RiboNucleic Acid

HPC: High performance Computing

CMP: Chip-level Multi-Processors

SIMT: Single-Instruction Multiple-Thread

HPC: High-performance computing

SM: Stream Multiprocessors

TPCs: Thread Processing Clusters

SPs: Stream Processors

SFUs: Special Function Units

LD/ST: Loads/Stores

LIS: Longest Increase Subsequence

MUM: Maximal Unique Match

MEM: Maximal Exact Match

MAM: Maximal Almost-Unique Match

MIM: Maximal Inverse Match

ESA: Enhanced Suffix Array

LCP: Longest Common Prefix

CAPÍTULO 1 - INTRODUCCIÓN

1.1 Resumen

Este proyecto lleva a cabo una caracterización y análisis del rendimiento de algoritmos utilizados en la comparación de secuencias genómicas completas, y ejecutadas en arquitecturas *MultiCore* y *ManyCore*. A partir del análisis se evalúa la idoneidad de este tipo de arquitecturas para resolver el problema de comparar secuencias genómicas. Finalmente se propone una serie de modificaciones en las implementaciones de estos algoritmos con el objetivo de mejorar el rendimiento.

En los siguientes apartados se introduce el actual trasfondo de la bioinformática en el marco del cómputo basado en *GPGPU* (*General-Purpose Computing on Graphics Processing Units*) utilizando GPUs (*Graphics Processing Units*) con arquitectura ManyCore, los objetivos del proyecto, las motivaciones personales, un breve resumen de la organización de la memoria y la planificación y costes del proyecto.

1.2 Estado del Arte de la Bioinformática

La bioinformática se basa en aplicar técnicas de computación para la gestión, análisis y distribución de datos biológicos con la finalidad de resolver problemáticas en el ámbito de la biología. En la últimas décadas la comunidad científica ha mostrado un gran interés por el ámbito bioinformático. Los métodos de secuenciación permiten extraer la información genética de los organismos. En 3 décadas ha habido un enorme salto cuantitativo en la obtención de pares de bases genómicas, a consecuencia del avance en las nuevas tecnologías de secuenciación[1] [3]. Se ha pasando de obtener 10^4 bases diarias a obtener 10^{13} bases diarias en poco más de 30 años, además de lograr una reducción drástica del coste del proceso de obtención. (ver Fig. 1.1)

Al aumentar este número de secuenciaciones completas y la cantidad de datos biológicos disponibles, los proyectos involucrados en el análisis comparativo de 2 genomas completos han ido tomando cada vez más relevancia (ver Fig. 1.2). Este auge en la adquisición de datos genómicos permite a la comunidad bioinformática disponer de datos suficientes para plantearse problemáticas de gran envergadura, y atrae a gran cantidad de profesionales de la comunidad científica.

Identificar genes es una tarea importante en la investigación bioinformática. A partir de los genomas (*DNA*) de 2 especies relacionadas, el problema denominado WGA (*Whole Genome Alignment*) identifica regiones de los genomas que posiblemente contienen genes conservados

[1] Desde la primera secuenciación exitosa de un genoma completo mediante el método *shotgun* en 1995 (*Hemophilus influenzae*), el número de organismos cuyos genomas han sido secuenciados completamente ha tenido un aumento exponencial hasta la fecha. Actualmente *GenBank* es la mayor base de datos genómica, estando considerada como una base de datos genómica de referencia con más de 8000 genomas completos secuenciados.

entre ellos. Dos genes son conservados si los dos producen unas proteínas con funciones similares. [1]

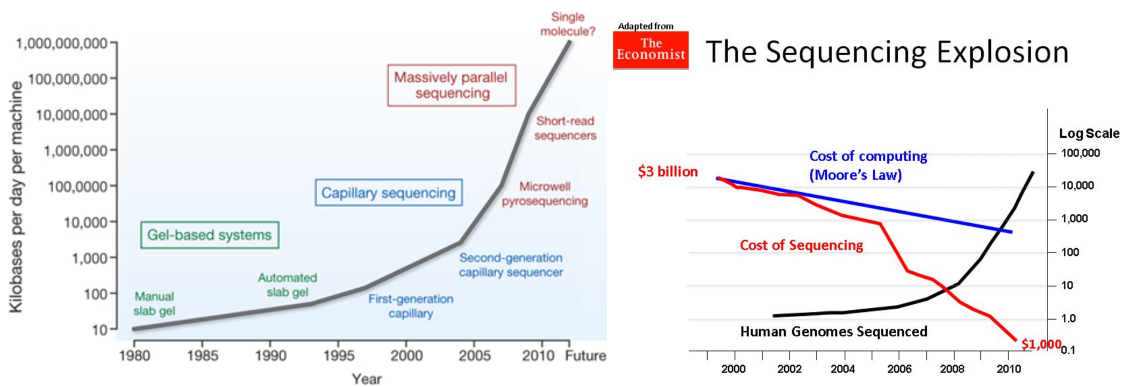


Figura 1.1: La imagen de la izquierda muestra el incremento de la obtención de bases genómicas con el avance de las tecnologías de secuenciación. La figura de la derecha indica la relación entre el coste de la secuenciación y el coste del cómputo lo largo de la última década.

WGA se ofrece como una forma muy eficaz y prometedora para investigar los mecanismos globales de evolución entre 2 especímenes estrechamente relacionados genéticamente. Además de facilitar el análisis de comparaciones cepa-a-cepa, comparaciones evolutivas y duplicaciones genómicas. [2]

Actualmente *WGA* es una problemática que supone un reto muy exigente en el ámbito de la computación. Este reto viene propiciado por los grandes requisitos en cómputo, espacio y ancho de memoria que entraña la naturaleza del problema al comparar 2 grandes secuencias de genomas completos, que ocupan en un orden de magnitud que oscila entre varios MB y cientos de GB.

En la última década se han presentado diversas propuestas para solucionar la problemática *WGA* que comprenden soluciones exactas y soluciones aproximadas mediante heurísticas.

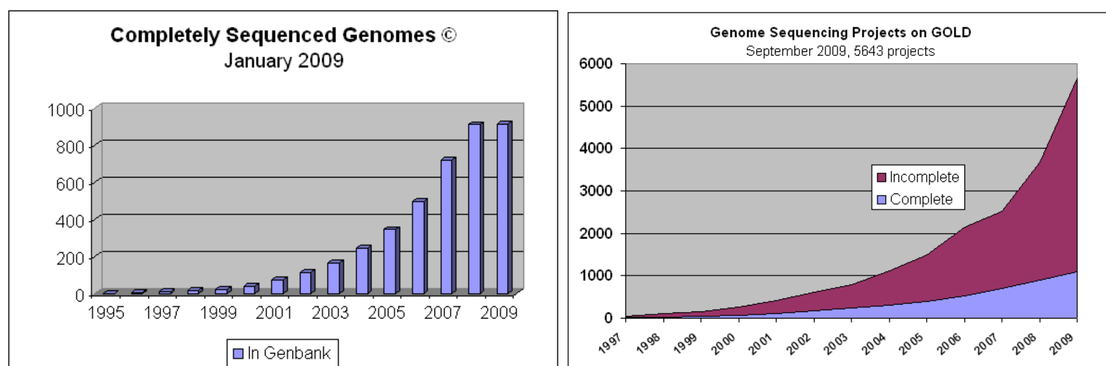


Figura 1.2: En la imagen de la izquierda podemos apreciar el incremento del número de secuenciaciones de genomas completos de la base de datos online *GenBank*. La figura de la derecha nos muestra un recuento de los proyectos activos y finalizados de la base de datos online GOLD.

Este proyecto se centrará en evaluar 3 implementaciones para arquitecturas *MultiCore* y *ManyCore* que tratan de resolver la problemática del *WGA* de forma aproximada.

1.3 Arquitecturas de procesadores actuales y cómputo GPGPU

Las técnicas de programación *GPGPU* (*General-Purpose Computing on Graphics Processing Units*) permiten hacer uso de procesadores hasta ahora destinados a procesado de gráficos por computador, para tareas de propósito más general. Las *GPUs* (*Graphics Processing Units*) son arquitecturas *ManyCores*, es decir, que cuentan con cientos de unidades de procesamiento orientadas a ejecutar aplicaciones masivamente paralelas. En este tipo de aplicación (o etapa de la aplicación) masivamente paralela es donde podemos obtener un rendimiento de un orden de magnitud superior al ofrecido por arquitecturas *MultiCores* actuales.

Las arquitecturas *ManyCore* gozan de mayor capacidad de cómputo, ancho de banda en memoria y menor consumo por FLOP que las arquitecturas *MultiCore* (ver Fig. 1.3). Por el contrario, las arquitecturas ManyCore albergan un tamaño de memoria un orden de magnitud inferior a los MultiCore y requieren un mayor esfuerzo por parte del programador para obtener una alta eficiencia. [15]

La comunidad científica comienza a interesarse en adaptar sus aplicaciones de cómputo científico para las arquitecturas *ManyCores* debido al incremento de rendimiento o la reducción de consumo que les puede suponer en sus aplicaciones paralelas.

En los últimos 5 años (2007-2011) Nvidia ha implantado e impulsado un paradigma de programación denominado *CUDA* (*Compute Unified Device Architecture*) que permite desarrollar aplicaciones de cómputo científico específicamente para las *GPUs* (*Graphics Processing Units*) que ella misma comercializa.

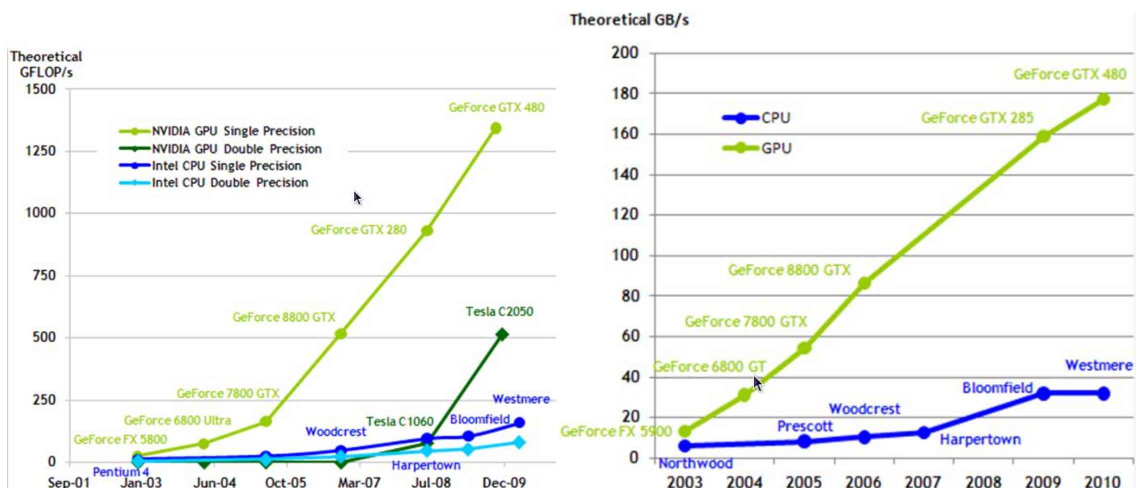


Figura 1.3: Las dos gráficas muestran comparativas cualitativas entre procesadores con arquitectura *MultiCore* y *ManyCore*. A la izquierda se compara la potencia de cómputo pico en GLOPS de simple precisión. A la derecha se compara el de ancho de banda pico en GB/s.

Nvidia proporciona un amplio soporte y abundante documentación para sus arquitecturas *ManyCores* en relación al resto de compañías. *CUDA* actualmente ha alcanzado su versión 4.0 con actualizaciones y mejoras constantes cada pocos meses. El entorno de CUDA proporciona librerías de cálculo, herramientas de depuración, herramientas de profiling, etc. [19] En este preciso momento es de los paradigmas con mayor aceptación destinados a realizar cómputo en arquitecturas *ManyCore* dentro de la comunidad científica.

En este proyecto se realizará el análisis de aplicaciones WGA ejecutadas sobre la arquitectura *Multicore* y la arquitectura *ManyCore* de Nvidia denominada Fermi, haciendo uso de su entorno de programación *CUDA*.

1.4 Objetivos

- Definir el modelo de ejecución de los procesadores *ManyCore*.
- Analizar el rendimiento de 3 implementaciones una sobre plataformas *Multicore* y otras dos para *ManyCore*.
- Analizar el código con el fin de identificar problemas de rendimiento.
- Identificar posibles cuellos de botella en el rendimiento.
- Medir y evaluar la eficiencia de la utilización del hardware
- Proponer posibles mejoras en la implementación con el fin de obtener un mayor rendimiento.

1.5 Motivaciones

Los siguientes puntos han sido clave para la motivación personal a la hora de desarrollar el proyecto:

- Estudio y análisis de una novedosa arquitectura *ManyCore* con amplio interés por parte de la comunidad científica.
- Familiarización y utilización de un emergente paradigma de programación masivamente paralela (*CUDA*) y con herramientas utilizadas en ámbito de producción.
- Posibilidad de análisis a bajo nivel de una arquitectura cerrada.
- Análisis de aplicaciones reales y ampliamente utilizadas en el ámbito de la bioinformática.
- Experimentación con datos reales, gracias al acceso a bases de datos genómicas públicas.
- Posibilidad de aplicar los resultados de este proyecto en el ámbito bioinformático y en disciplinas análogas (como por ejemplo la minería de datos).

1.6 Resumen de la memoria

A continuación se hace una pequeña descripción de cómo está organizado cada capítulo de la memoria:

· CAPÍTULO 1 – Se realiza un resumen del estado actual de la bioinformática en el ámbito de comparación genómica y de los problemas de cómputo y memoria asociados actualmente.

· CAPÍTULO 2 – Se introducen conceptos teóricos relacionados con la bioinformática para la comprensión de la problemática WGA y su resolución.

· CAPÍTULO 3 – Se detallan conceptos teóricos necesarios sobre arquitecturas MultiCore y ManyCore y el modelo de programación para arquitecturas *ManyCore* de *Nvidia*.

· CAPÍTULO 4 – Se explican conceptos teóricos sobre indexación de textos mediante los algoritmos *Suffix-Tree*, *Suffix-Array* y *Enhanced Suffix-Array*.

· CAPÍTULO 5 – Explicación de 3 aplicaciones bioinformáticas de WGA: *MUMmer*, *MUMmerGPU* y *MUMmerGPUpp* analizadas para arquitecturas *MultiCore* y *ManyCore*.

· CAPÍTULO 6 – Evaluación experimental de las aplicaciones anteriores con el objetivo de encontrar cuellos de botella e ineficiencias. Para finalizar se propone un listado de posibles optimizaciones.

· CAPÍTULO 7 – Conclusiones finales del análisis de las aplicaciones en los *ManyCores* de *Nvidia* y posibles líneas abiertas.

1.7 Planificación y Coste

A continuación, en la figura 1.4 se muestra la planificación del desarrollo del proyecto inicial:

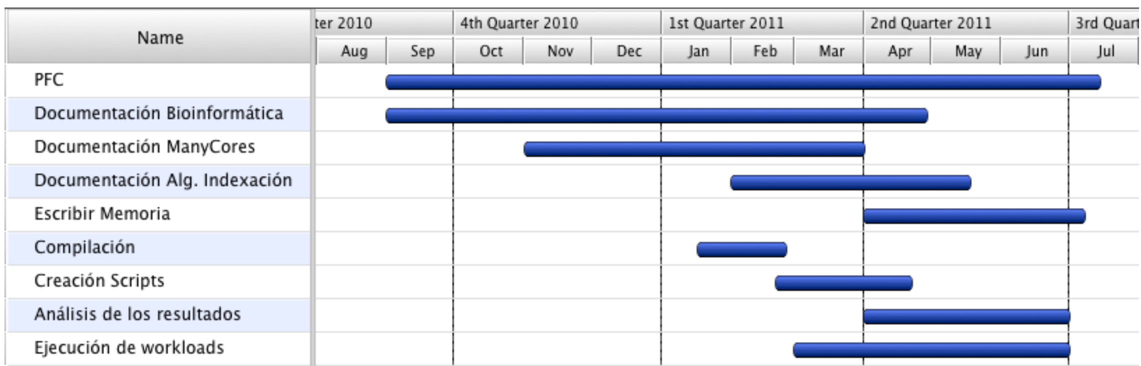


Figura 1.4: Diagrama de Gantt de la planificación del desarrollo del proyecto.

El coste viene asociado al tipo de desarrollo del proyecto, debido a que el proyecto es un trabajo de investigación introductorio no existe un análisis de requerimientos ni diseña de la aplicación, pero existen otras tareas enfocadas a la documentación y recopilación de información además de la generación de scripts y análisis de resultados.

En cuanto al coste si se realiza balance y se define que el coste de la mano de obra de un ingeniero esta en 9€, el número de horas de trabajo planificadas esta entorno a las 335h.

Se añade al coste el precio de adquisición del material como es un ordenador de sobremesa (795€) donde ejecutar las experimentaciones y un dispositivo CUDA donde poder lanzar las ejecuciones GPGPU (290€) el coste asciende a 4100€

Hay que tomar en cuenta que todas las herramientas utilizadas han sido gratuitas y en la mayoría de los casos hasta de código libre, por lo que no suponen un sobrecoste.

CAPÍTULO 2 – BIOINFORMÁTICA

2.1 Resumen

Las herramientas de análisis de secuencias genómicas permiten a los biólogos identificar y entender regiones fundamentales que tienen implicación en enfermedades genéticas. Actualmente existe una necesidad de dotar al ámbito científico de herramientas de análisis eficientes.

En los siguientes puntos del capítulo se tratará de introducir conceptos biológicos que definen las representaciones de datos genómicos utilizados en la bioinformática. También se tratan las motivaciones que propiciaron la búsqueda de métodos computaciones eficientes en el campo del análisis comparativo de secuencias genéticas completas. Finalmente se presentan las diferentes alternativas computacionales para abordar la problemática del *Whole Genome Alignment*.

2.2 Conceptos Biológicos

Las substancias químicas con mayor relevancia en nuestras células son moléculas llamadas DNA (*DeoxyriboNucleic Acid*), RNA (*RiboNucleic Acid*)[1] y proteínas[2]. Una molécula DNA contiene dos hélices enlazadas, cada hélice se encadena con cuatro tipos de nucleótidos: adenina, citosina, guanina, timina (abreviados como A, C, G, T). Cada nucleótido se une con un nucleótido opuesto de la segunda hélice. Los nucleótidos solo tienen una correspondencia uno a uno, existe un enlazado donde el nucleótido A enlaza únicamente con T y C siempre enlaza con G. [2]

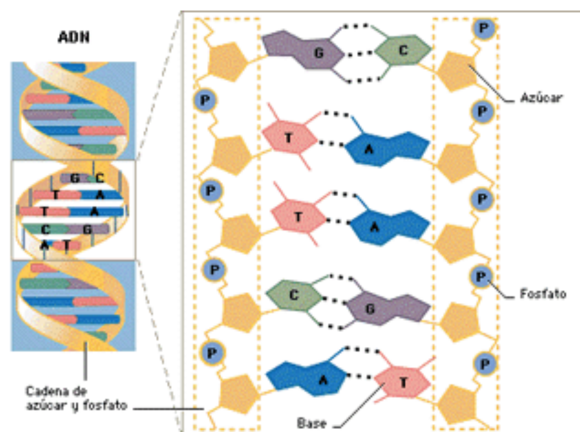


Figura 2.1- Esquema del segmento de una molécula de DNA donde se observan las hélices encadenadas.

[1] RNA traduce la información del DNA para sintetizar las proteínas. RNA utiliza un alfabeto similar al DNA, excepto que el nucleótido T es reemplazado por U.

[2] Una proteína es una secuencia sobre un alfabeto de 20 aminoácidos. Las proteínas forman estructuralmente nuestras células y cooperan en la mayoría de nuestras reacciones químicas necesarias para la vida.

Por lo tanto, una hélice de DNA puede ser considerada como una cadena sobre un alfabeto {A, C, G, T}. Podemos ver un ejemplo de secuencia en la Fig. 2.2, donde sabiendo la correspondencia entre nucleótidos, la cadena complementaria puede ser deducida.

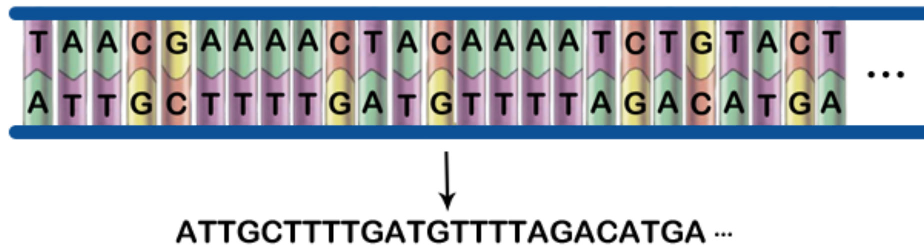


Figura 2.2- Representación de los 25 primeros pbs (pares de bases) del cromosoma 22 del *Homo sapiens*

El genoma es la secuencia completa del *DNA* de un organismo. El genoma completo está distribuido en distintos bloques denominados cromosomas. Los cromosomas albergan parte de la secuencia genética del organismo, que se divide en regiones en relación a su funcionalidad. (ver fig. 2.3).

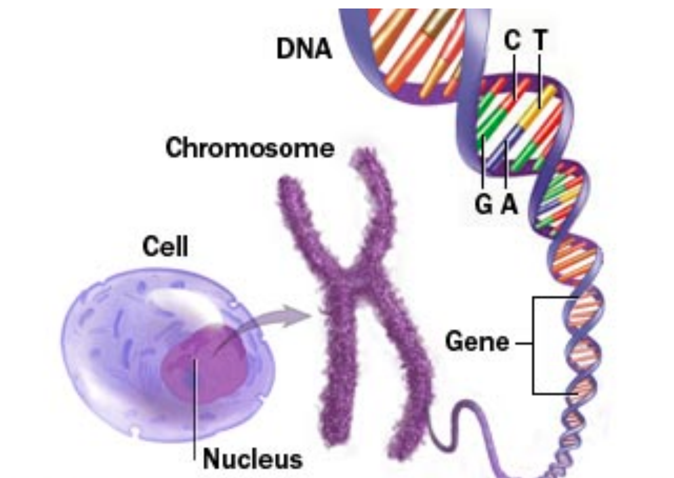


Figura 2.3- Esquema conceptual del DNA de un espécimen.

Algunas de estas regiones se denominan genes y desempeñan la síntesis del *RNA* y las proteínas. Las regiones adyacentes entre los genes se denominan regiones intergénicas. En la figura 2.4 se pueden observar diferentes tipos de regiones en una secuencia.

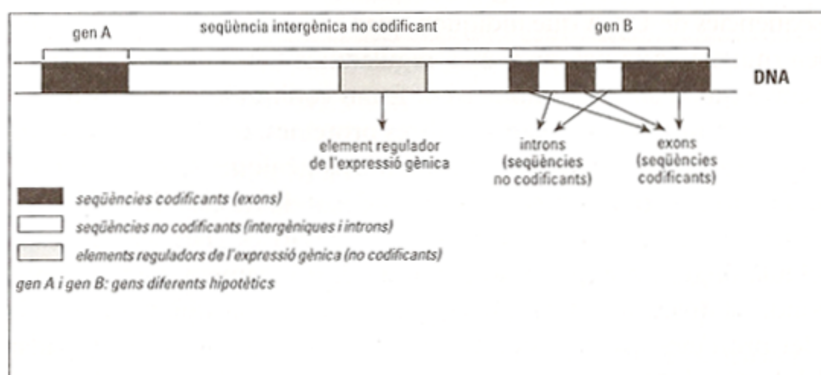


Figura 2.4 – Esquema conceptual de las regiones que puede contener una secuencia.

Para hacernos una idea del tamaño de un genoma, el humano contiene ~$3 \cdot 10^9$ pbs (*pares de bases*), que corresponden a ~715MB de información (utilizando 2bits por base), separados en 23 cromosomas. Si se analiza completamente se pueden contabilizar más de 30.000 genes. Habitualmente los genes tienen una longitud entre 2000 – 3000 pbs.

Es importante resaltar que el genoma humano no es uno de los más extensos. El más extenso conocido hasta hoy corresponde a la planta *Amoeba dubia* con ~$2,9 \cdot 10^{11}$ pbs (~70GB de información).

2.3 Motivaciones del estudio de los genomas

El genoma es el *DNA* completo de un organismo incluyendo genes y zonas no codificantes. Los genes permiten sintetizar proteínas, y cada una de éstas, conforman las células y facilitan las reacciones químicas necesarias para nuestro cuerpo. Es por ello que uno de los primeros objetivos del estudio de un genoma es identificar todos sus genes.

Las mutaciones en los genes nos pueden hacer más susceptibles a determinadas dolencias. Además, la información genética almacenada en los genes pasa de una generación a sus descendientes. Esto explica porqué muchas dolencias muestran un patrón inherente. Si se lograse identificar las localizaciones y funciones de todos los genes del organismo, se podría habilitar o deshabilitar genes que regulan procesos de gran interés, como los genes responsables de este tipo de dolencias.

Pero no únicamente es importante la identificación de genes: sino que el estudio de los genomas puede ayudarnos a identificar y entender otras regiones de los genes, como por ejemplo, las regiones reguladoras. Estas regiones tienen un papel importante en la aparición de algunas enfermedades hereditarias.

El análisis de genomas con este tipo de metodologías también permite realizar estudios filogenéticos con el fin de identificar ancestros comunes entre especies. [3]

2.4 Métodos existentes en la identificación de los genes.

Una problemática que impulsó fuertemente el ámbito del análisis de secuencias fue la fuerte necesidad de identificar nuevos genes en los genomas. Esta problemática despertó el interés de diversos ámbitos interdisciplinarios de la comunidad científica muy vinculados entre sí, que requieren el uso o el desarrollo de diferentes técnicas que incluyen informática, matemática aplicada, estadística, ciencias de la computación, inteligencia artificial, química y bioquímica.

Hasta la fecha, los métodos de identificación y análisis de secuencias de forma experimental (mediante procesos biológicos y químicos) proveen los mejores resultados, pero aplicarlos a tamaños del orden de un genoma es irrealizable por su alto tiempo de procesado y coste. Este fue uno de los motivos primordiales por el cual hubo un gran interés en buscar métodos alternativos de análisis.

Los grandes avances en el ámbito de la miniaturización de componentes semiconductores, la denominada ley de Moore[3] (ver Fig. 2.5), junto con el avance en los sistemas HPC (High performance Computing)[4] y la reducción de costes en la obtención de secuencias han permitido aplicar métodos computacionales de forma eficiente y precisa para la identificación de genes en nuevos genomas con costes más reducidos.

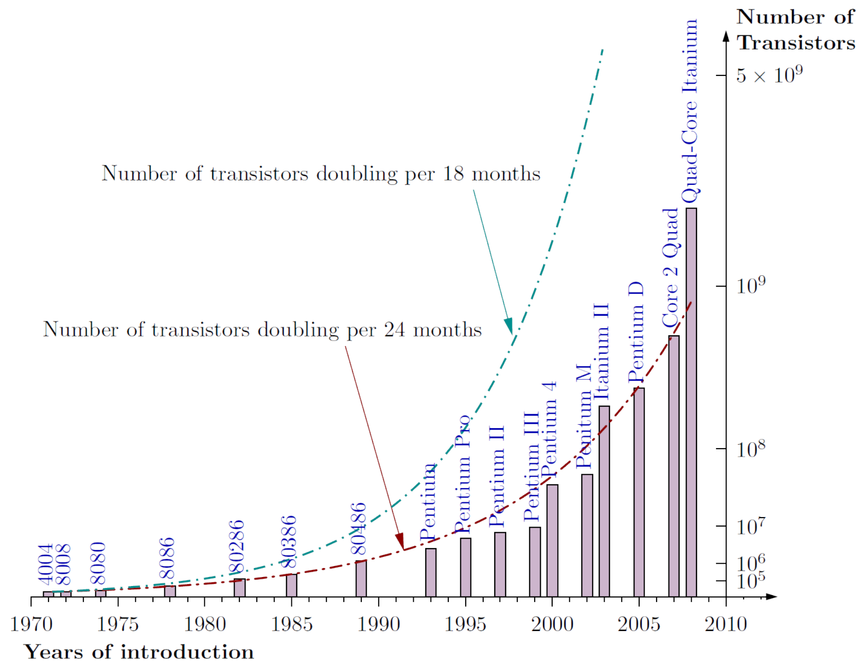


Figura 2.5: Se muestra la predicción de la Ley de Moore en el tiempo, y cómo se va cumpliendo con diferentes arquitecturas de computadores lanzadas al mercado.

Actualmente existen 3 importantes metodologías de tipo computacional en el análisis de secuencias: Método *Intrinsic*, Método *Extrinsic* y Método *Cross-species*.

- Los métodos intrínsecos predicen genes utilizando las características de los mismos (marcas genéticas y tendencias estadísticas) como entrada. Los genes a menudo tienen características específicas que permiten distinguir entre ellos a partir de regiones intergénicas. Este método se limita a secuencias con un único y completo gen.

- Los métodos extrínsecos comparan la secuencia de entrada con una base de datos de secuencias ya caracterizadas con el objetivo de verificar su similitud. Los resultados del método tienen una fuerte dependencia de los genes ya caracterizados en la base de datos. Las secuencias tienen una longitud muy limitada.

- El método *cross-species* realiza una comparación entre dos secuencias de especímenes estrechamente relacionados en la cadena evolutiva. El hecho de compartir una alta similitud entre ellas ayudará a identificar regiones en la nueva secuencia. Estos métodos permiten estudiar secuencias a una gran escala.

[3] Expresa que aproximadamente cada 24 meses se duplica el número de transistores en un circuito integrado. Se trata de una ley empírica, formulada por el co-fundador de Intel, Gordon E. Moore, en 1965.
[4] Sistemas de computación de alto rendimiento destinados a cómputo científico.

Los métodos intrínsecos y extrínsecos no son válidos para análisis de secuencias genómicas debido a su tamaño, por ello el proyecto centrará su interés en el método *Cross-Species.*

2.4.1 Método de comparación *Cross-species*

Debido a que el número de nuevos genomas secuenciados está creciendo exponencialmente en los últimos años, surgió un novedoso método para la identificación de genes. Los investigadores comenzaban a comparar los genomas de especies muy relacionadas. Dados dos genomas estrechamente relacionados, el problema del *WGA* trata de identificar las regiones compartidas entre los genomas de los dos especímenes. Además este método no sólo se utiliza para la identificación de genes sino que su aplicación se ha extendido a diferentes análisis de secuencias genómicas.

Existen diferentes investigaciones que utilizan estas técnicas, hecho que refleja que se trata de un problema con un gran impacto e interés.

El método se basa en la siguiente premisa. Dos especies estrechamente relacionadas, como por ejemplo la de los humanos y los ratones, proceden de un mismo ancestro común, dado que comparten muchos pares de genes conservados. Se dice que dos genes son conservados entre ellos si cada uno produce proteínas con funciones similares. [9]

Las regiones génicas están sometidas a mecanismos biológicos como la presión selectiva[5], las regiones reguladoras o la compactación del ADN. Estos mecanismos reducen la probabilidad de que aparezcan mutaciones genéticas o de que estas persistan en las regiones genómicas. Por lo que es probable que un par de genes conservados compartan a menudo un alto grado de similitud en sus secuencias.

Por el contrario, las regiones intergénicas[6] están sujetas a cambios. Estas mutaciones pueden ser fácilmente acumulativas, por lo que las regiones intergénicas de especies estrechamente relacionadas a menudo divergen de la región original del ancestro común, siendo diferentes entre los genomas.

Por lo tanto, el problema del *WGA* puede ser abstraído como un problema basado en localizar pares de regiones dentro de 2 grandes cadenas (ver figura 6) de tal forma que las regiones con una fuerte similitud se relacionen como partes conservadas.

Diversos estudios de comparación de especies estrechamente relacionadas verifican este hecho. [12]

---

[5] La presión selectiva hace referencia a que si una mutación repercute negativamente al fenotipo del individuo disminuye la probabilidad de que tenga descendencia y por tanto comparta esa modificación.

[6] Son regiones con una posible menor implicación dentro del genoma, con lo cual están sometidas a un menor número de medidas de control. Además, son regiones que suelen estar compactadas en menor medida o no protegidas por otras estructuras, por lo que están más expuestas a agentes externos que las puedan modificar.

Figura 2.6- Correspondencia de similitud entre regiones génicas en 2 genomas de diferentes especies.

2.5 Alineamiento entre secuencias

Un alineamiento entre 2 secuencias es un modo de representación que facilita la visualización y estimación de la homología entre secuencias.

Dos secuencias tienen múltiples alineamientos entre ellas. Un alineamiento simula los cambios sufridos en una secuencia referencia A para acabar produciéndose la secuencia de consulta B. Los cambios que pueden sufrir las secuencias pueden ser Inserciones (I), Substituciones (S) y Eliminación (D). Estos cambios quedan representados en el alineamiento por la denominada secuencia de edición (L). Los cambios indicados como S corresponden a pares de bases que se conservan entre secuencias (representadas con el carácter |) o como bases que han mutado, y se denominan Miss Match (en este caso no se incluye el carácter |).

Tomando como ejemplos las secuencias A (AGTGTTCCAG) como referencia y B (AATCGTTACAG) como secuencia consulta por alinear. Se muestran 3 posibles alineamientos (ver fig 2.7):

Azul

```
A: -AGT-GTTCCAG
    | | ||| |||
B: AA-TCGTTACAG
L: ISDSISSSSSS
```

Negro

```
A: AG-TGTTCC--AG-
   |
B: AAT--CGT-TACAG
L: SSIDDSSSDIISSI
```

Rojo

```
A: AGT-GTTCCA----G
        |
B: ---AATCGTTACAG-
L: DDDISSSSSSIIIID
```

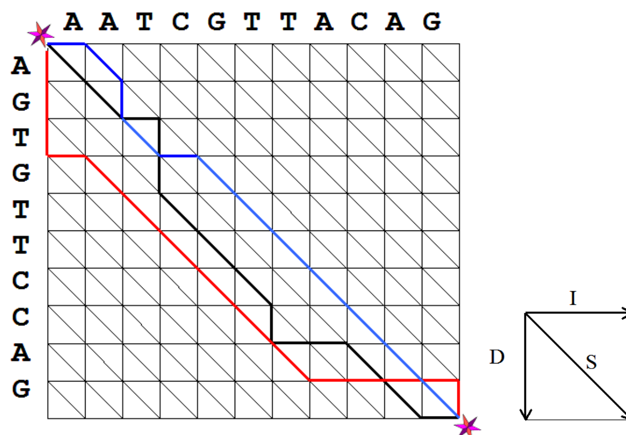


Figura 2.7- Representación matricial de 3 alineamientos entre las secuencias A y B.

Para escoger el alineamiento más verosímil desde un punto de vista biológico, se pondera la secuencia de edición L. El esquema de ponderación más sencillo es utilizar costes fijos[7] a cada cambio sufrido. [11]

Un esquema ampliamente utilizado es el siguiente:

- Mantener una base (S con el carácter |): +1
- Substitución de una base (MissMatch): -1
- Eliminación o Inserción de una base (D, I): -2

El alineamiento que obtiene la mínima ponderación entre todos los posibles se considera el alineamiento óptimo. El alineamiento Azul del ejemplo tiene una ponderación de 1.

El número de alineamientos posibles dados por dos secuencias A y B de longitudes *m* y *n* respectivamente es $\binom{m+n}{m}$, que puede ser representado como todos los caminos posibles entre las de la matriz de alineamiento (la matriz que representa todos los posibles alineamientos). En el ejemplo anterior obtenemos $\binom{11+10}{11}$, es decir, 352716 alineaciones posibles. Esto da una idea del volumen del problema cuando los valores de *n* y *m* son grandes (miles o incluso millones).

2.6 Algoritmos para el problema Whole Genome Alignment

Dados 2 genomas A y B, el alineamiento entre los genomas es un conjunto de pares de regiones de A y B. Para cada par, la región del genoma A comparte una alta similitud con la región del genoma B. Se espera que para dichas regiones exista una correspondencia biológica (ver figura 2.8).

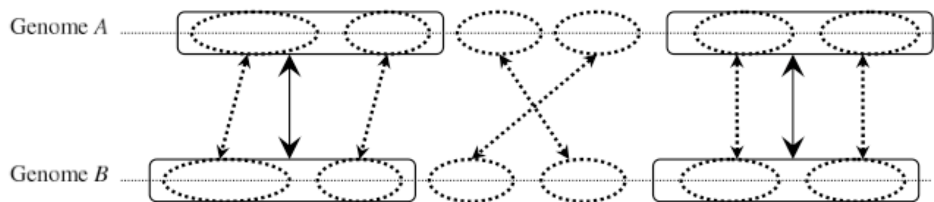


Figura 2.8 - Los genomas de entrada A y B comparten 2 pares de genes conservados. El alineamiento entre A y B consiste en 6 pares de regiones (mostrados como círculos), las 2 partes del centro son regiones intergénicas.

El objetivo de análisis del WGA es buscar cambios a gran escala en las comparaciones de genomas completos, cambios tales como repeticiones en tándem[8], inversiones a gran escala[9] o translocaciones[10].

---

[7] La responsabilidad de utilizar una política de costes u otra recae en el biólogo, que escogerá la más representativa para su problemática, la cual no tiene porque ser de costes fijos.

[8] Las repeticiones en tándem son una larga sucesión de bases que se repiten secuencialmente una tras otra en una misma secuencia.

[9] Las inversiones son una región de la secuencia que se encuentra conservada en el otro genoma, pero en sentido de lectura inverso.

En la figura 2.9, se exponen algunos de los cambios a gran escala que se comentan.

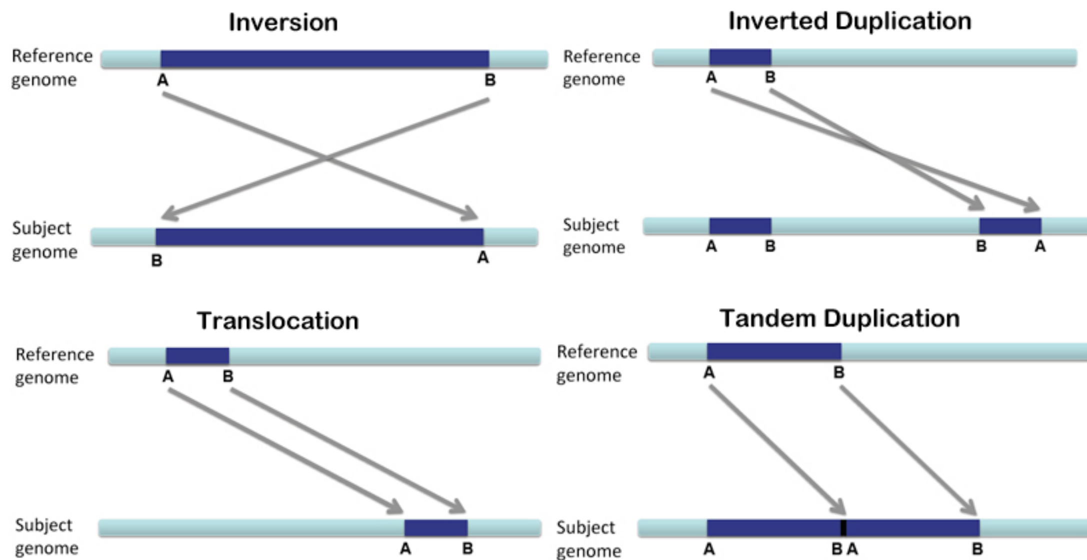


Figura 2.9 – Cambios en los genomas que analiza el WGA

2.6.1 Métodos de alineamiento exactos

El problema de alinear 2 secuencias ha sido estudiado durante décadas. Anteriores estudios como el alineamiento global propuesto por *Needleman&Wunsch* y el alineamiento local propuesto por *Smith&Waterman*, están ideados para evaluar todas las posibles alineaciones y otorgar el alineamiento óptimo.

Estos algoritmos para calcular los alineamientos globales y locales anteriores toman una complejidad cuadrática tanto en tiempo como memoria, concretamente O(m·n), donde m y n son las longitudes de las secuencias de entrada.

Estos métodos de alineamiento fueron ideados para alinear 2 secuencias genéticas de un tamaño relativamente pequeño. Su aplicación es inviable para alinear genomas completos fundamentalmente por dos motivos:

- La complejidad en tiempo y en memoria de los algoritmos, ya que los genomas son estructuras que normalmente contienen millones de nucleótidos. Esto implica que tienen tiempos de procesado y una ocupación en memoria inaceptable o inasumible.

- Este tipo de algoritmos están enfocados en descubrir inserciones, eliminaciones y mutaciones puntuales, pero no en buscar cambios a gran escala como demanda el *WGA*.

[10] Las translocaciones son desplazamientos de una parte del genoma a otra región distinta.

2.6.2 Métodos de alineamiento aproximados

Debido a la inviabilidad de aplicar métodos exactos en el análisis de genomas, actualmente existe un especial interés en desarrollar aplicaciones y algoritmos específicos para tratar el problema del *WGA.*

Estos algoritmos hacen uso de métodos de alineamiento aproximados, con el objetivo de reducir el tiempo y consumo de memoria, e intentado ofrecer un alineamiento de la mayor calidad posible[11].

Todos ellos hacen uso de una particularidad comentada anteriormente, analizar genomas que tengan una relación evolutiva muy estrecha. Esto permite utilizar heurísticas con el fin de acelerar el proceso de análisis.

Estas heurísticas estarán basadas en que una pareja de genes conservados normalmente contendrán muchas subcadenas en común y a menudo únicas en ambos genomas.

Los algoritmos de análisis de genomas (MUMmer, PIPMaker, DIALIGN, LAGAN …) utilizan una metodología muy similar [12], dividiendo el proceso en 3 etapas (fig. 10).

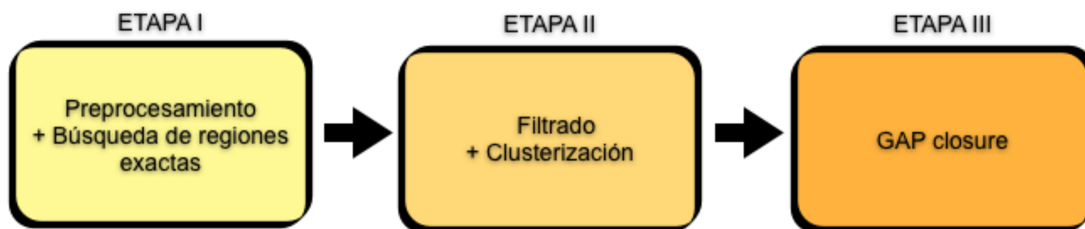


Figura 10 – Modelo de las 3 principales etapas presentas en la mayoría de algoritmos para el WGA

A continuación, se describen las diferentes etapas que tienen implicación en los algoritmos de alineamiento aproximado:

- Primera etapa: *preprocesamiento y búsqueda de regiones exactas*. Es la etapa con mayor requerimientos computacionales. El objetivo es encontrar todas las subcadenas, con un tamaño mínimo, comunes entre los 2 genomas. La parte de preprocesado suele corresponder a adecuar las secuencias para la búsqueda de las subcadenas, como por ejemplo, aplicar métodos de indexación de textos.
  Esta etapa en diferentes algoritmos se denomina búsqueda de MUMs o K-mers (coincidencias exactas de 2 cadenas).

[11] Al no ser métodos exactos, la comunidad científica puede comparar la precisión de su método con la del resto evaluando la calidad de los alineamientos que producen. Para ello se basan en 2 aspectos, el porcentaje de *coverage* y *sensitivy* de cada alineamiento.

- Segunda etapa: *filtrado y clusterización*. El objetivo de ésta etapa es escoger del conjunto de subcadenas ya encontradas, un subconjunto de éstas que represente lo más fielmente posible el futuro alineamiento. Normalmente se utilizan métodos de ordenación de secuencias respecto al genoma de referencia y heurísticas para escoger el conjunto ordenado más indicado. También se utilizan heurísticas a fin de agrupar en clústeres las anteriores subcadenas filtradas. Estos clústeres son considerados regiones conservadas entre genomas.

- Tercera etapa: *GAP closure*. Finalmente se realiza un alineamiento local entre las secuencias consecutivas de cada clúster a fin de refinar el alineamiento final. A este procedimiento se le suele denominar *GAP closure*. (fig. 2.11)

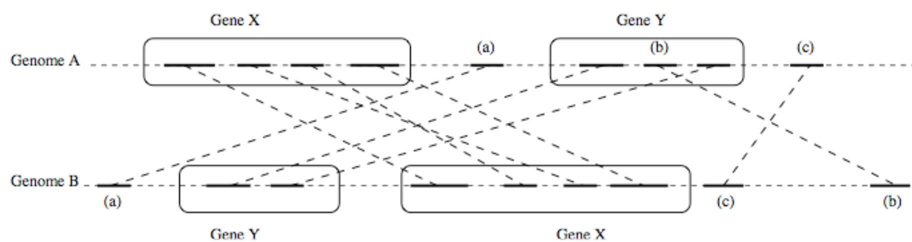


Figura 2.11 – Muestra como la tercera etapa agrupa las subcadenas encontradas en clústeres.

En próximos capítulos veremos que la primera etapa conlleva un gran cómputo y consume gran parte del tiempo de análisis. Además tiene un objetivo muy claro, encontrar subcadenas comunes en ambos genomas. De este modo, no tiene una implicación biológica que demande una amplia base biológica por parte de los informáticos que diseñan el algoritmo y su implementación.

En cambio la segunda y tercera fase hacen uso de heurísticas con base biológica, las cuales demandan un conocimiento de la biología más amplio por parte del informático, lo que las convierte en menos adecuadas para ser abordadas en un proyecto final de carrera. La optimización de estas fases puede considerarse un proceso crítico ya que el resultado final del alineamiento se puede ver muy comprometido con los cambios. Además, las diferentes implementaciones existentes (MUMmer, PIPMaker, DIALIGN, LAGAN …) hacen uso de heurísticas muy dispares. [12]

Así pues, la primera etapa de búsqueda de subcadenas es una fase que se presta a ser estudiada por la comunidad informática, ya que es utilizada en la mayoría de algoritmos y tiene una motivación computacional debido a que consume parte del tiempo de procesado. El hecho de no depender de grandes conocimientos biológicos permite adecuarlo al tiempo de duración de un proyecto final de carrera como el que se presenta.

De todas las implementaciones, *MUMmer3* es probablemente el software más popular aplicado al problema del *WGA*. Utiliza una solución aproximada aplicando un algoritmo

greedy[12]. Con las soluciones locales genera unos clústeres de pares para finalmente ordenarlos en función de su orden de aparición en los genomas.

Este proyecto analizará la implementación de *MUMmer3* y otras dos implementaciones para arquitecturas *ManyCore* basadas en él. Esta elección ha sido realizada debido a la popularidad de *MUMmer3* en el ámbito de la bioinformática, a que su código es abierto (*opensouce*) y a la amplia documentación que existe sobre él.

2.6 Puntos clave del capítulo

- El estudio de los genomas puede ayudarnos a identificar y entender otras regiones de los genes. Estas regiones tienen un papel importante en la aparición de algunas enfermedades hereditarias. Las herramientas de análisis tienen un fuerte papel en estos estudios.

- Los métodos de comparación CrossSpecies comparan secuencias de 2 especímenes estrechamente relacionados evolutivamente, por lo que encontramos muchas subcadenas compartidas entre los dos genomas a comparar.

- La problemática WGA alinea 2 genomas en busca de cambios a gran escala. Los algoritmos que resuelven esta problemática se basan en la propiedad del punto anterior para implementar heurísticas para el alineamiento que ofrecen soluciones aproximadas eficientes.

- Los algoritmos aplicados a la problemática WGA por lo general comparten 3 etapas: preprocesamiento y búsqueda de regiones exactas, filtrado y postprocesado de las búsquedas y clusterización.

- MUMmer3 es una implementación del método de comparación CrossSpecies aproximado del cual se analizará la primera etapa, que tiene un alto coste computacional y no requiere unos altos conocimientos biológicos.

[12] Un algoritmo greedy permite resolver un problema utilizando una heurística que consiste en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima.

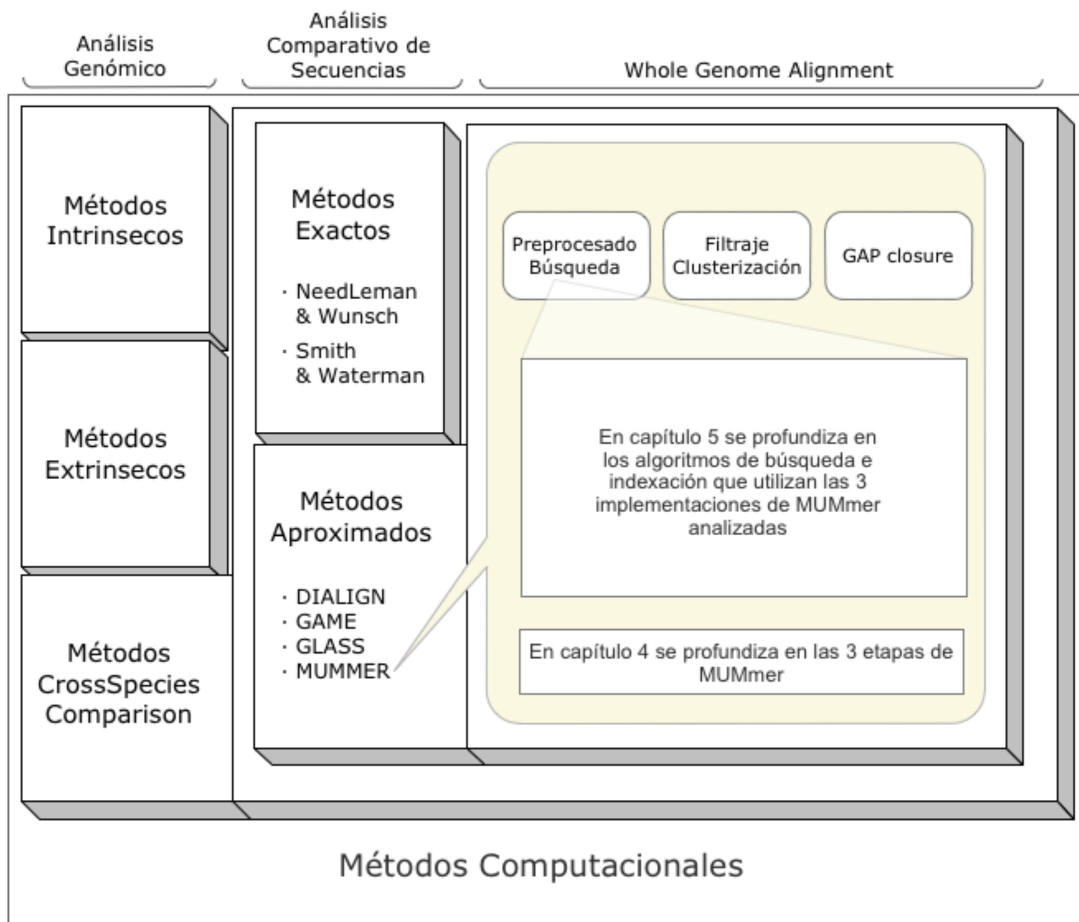


Figura 12 – Esquema y organización de los temas más importantes tratados en el capítulo.

CAPÍTULO 3 – MODELO DE PROGRAMACIÓN CUDA Y ARQUITECTURAS MANYCORE

3.1 Resumen

Actualmente la comunidad científica tiene un especial interés en estudiar y aprovechar las capacidades de los dispositivos GPU. El objetivo que se persigue es lograr ejecutar de forma eficiente algoritmos en GPU. El conjunto de técnicas para llevarlo a cabo se denomina GPGPU (*General-Purpose Computing on Graphics Processing Units*).

Entre las diferentes alternativas presentes en dispositivos GPU, *Nvidia* proporciona su propia solución para poder realizar cómputo GPGPU [15], CUDA (*Compute Unified Device Architecture*). Es un conjunto de herramientas, compiladores y soluciones para los dispositivos de *Nvidia*. Actualmente CUDA se considera la solución para realizar computo GPGPU más extendida y utilizada entre la comunidad científica. Muchos grupos de desarrollo se encuentran portando sus aplicaciones de cómputo científico a la plataforma CUDA.

Las GPUs son dispositivos que presentan una arquitectura de tipo *ManyCore* y pueden suponer una alternativa real a los tradicionales procesadores *MultiCore* para aplicaciones muy específicas. Concretamente las aplicaciones *MUMmerGPU* y *MUMmerGPU++* que se analizan en este proyecto hacen uso de arquitecturas *ManyCore* con tecnología CUDA.

En este capítulo se detalla el paradigma de programación CUDA para arquitecturas *ManyCore* de *Nvidia*, que resulta esencial para poder comprender el comportamiento de las aplicaciones bioinformáticas que se analizan.

3.2 Estado del arte en arquitecturas MultiCore y ManyCore.

El campo actual de la computación ha comenzado a evolucionar hacia nuevas direcciones, se continúan diseñando procesadores con grandes mejoras en la potencia de cómputo para los nuevos ordenadores que salen al mercado, pero ya no del mismo modo.

Hace unos años llegó el momento en el que los diseñadores de procesadores se toparon con grandes problemas que afectaban a la evolución de los actuales diseños de procesadores, problemas tales como:

- La dificultad para disipar el calor producido por los transistores de un procesador al trabajar con frecuencias de reloj muy altas, la denominada *Power Wall*.
- La creciente diferencia entre la velocidad de cómputo de una CPU y tiempo de acceso a las memorias fuera del chip, denominado *Memory Wall*.
- La incapacidad de mejorar el número de instrucciones ejecutadas por ciclo del procesador.

Todas estas problemáticas impedían alcanzar un aprovechamiento eficiente de la creciente inclusión de transistores en un solo chip que la *ley de Moore* seguía ofreciendo. Es por ello que la evolución en el diseño de los procesadores ha tomado un nuevo rumbo para dirigirse hacia

procesadores CMP (*Chip-level Multi-Processors*), lo que actualmente conocemos como procesadores *MultiCore*, que incluyen dos o más procesadores en un único chip.

Paralelamente a la evolución de las CPUs, las GPUs han tenido una arquitectura fuertemente influenciada por el ámbito en el que estaban destinadas. Siendo dispositivos de propósito específico, los cambios en el diseño de la arquitectura GPU estaban motivados por los requerimientos de elevado cómputo y ancho de banda en memoria que la computación gráfica demandaba. Los diseños de las arquitecturas GPUs cada vez incluían un mayor número de unidades funcionales en un mismo chip. Esto permitió ir obteniendo dispositivos con un creciente desempeño computacional al mismo tiempo que la *ley de Moore* marcaba un avance en la miniaturización de los recursos.

En los últimos años se eliminaron ciertas restricciones en las unidades funcionales de las GPUs. Esto supuso un punto de inflexión de cara al desarrollador de aplicaciones, ya que en ese preciso instante las GPUs comenzaban a presentarse como arquitecturas *ManyCore* destinadas a cómputo general, con centenares de unidades de procesamiento programables. [16]

Estos nuevos cambios en la evolución de los procesadores de propósito general han supuesto un cambio *en las reglas del juego* a los diseñadores de software donde anteriormente sólo debían esperar el lanzamiento del siguiente procesador para ver un notable incremento en el rendimiento de su aplicación secuencial. Ahora deben ser los propios programadores quienes medien en el proceso y realicen un esfuerzo adicional, diseñando sus aplicaciones para aprovechar estos recursos adicionales que las nuevas arquitecturas ofrecen.

Así pues, ahora el propio programador se ve condicionado a adquirir un mayor conocimiento de la arquitectura en que está desarrollando para poder sacar partido a estos recursos adicionales. Para ello debe realizar un esfuerzo extra en identificar las partes potencialmente paralelas de su código, con el objetivo de distribuir el trabajo entre los distintos recursos de cómputo que ofrece el procesador y obtener un mayor rendimiento en su aplicación.

3.3 Diferencias entre arquitecturas MultiCore y ManyCore.

El diseño de las arquitecturas *MultiCore* y *ManyCore* ha tenido una evolución muy dispar debido a las necesidades que debían cubrir. [19]

El objetivo de las arquitecturas *MultiCore* es el de reducir en la medida de lo posible las latencias de las operaciones que realizan. Para poder lograr estas bajas latencias en la ejecución implementan técnicas como la reordenación dinámica de instrucciones o la ejecución especulativa.

Por otro lado, las arquitecturas *ManyCore* están orientadas a aumentar en lo posible la productividad medida como operaciones por segundo. Para poder lograrlo implementan otro tipo de técnicas, como los cambios de contexto entre threads sin coste añadido para poder tolerar grandes latencias en memoria. [24]

Así pues, cada una ha tomado enfoques distintos en el diseño de la arquitectura y presentan distintos beneficios para las aplicaciones.

Las arquitecturas *MultiCore* ofrecen unas frecuencias de reloj más altas, un mejor tratamiento de los saltos condicionales y unas latencias de ejecución de instrucciones y de acceso a memoria muy reducidas. Esto favorece al desempeño de aplicaciones de tipo secuencial.

Las arquitecturas *ManyCore* se caracterizan por ofrecer un gran ancho de banda en memoria, gran potencia de cómputo y un alto grado de paralelismo a nivel de thread. Esto favorece a aplicaciones masivamente paralelas y con una alta carga de trabajo.

Considerando el área de ocupación del chip, las arquitecturas *MultiCore* actualmente presentan del orden de una decena de unidades de cómputo (cores), una compleja y avanzada lógica de control y grandes cachés. Por otro lado, las arquitecturas *ManyCore* destinan gran parte de su área de chip a introducir centenares de recursos de cómputo (cores), simplifican la lógica de control y reducen la complejidad de las cachés. En la figura 3.1 se detallan las diferencias de distribución de los recursos en el chip que suelen presentar las arquitecturas *MultiCore* y las *ManyCore*. [20]

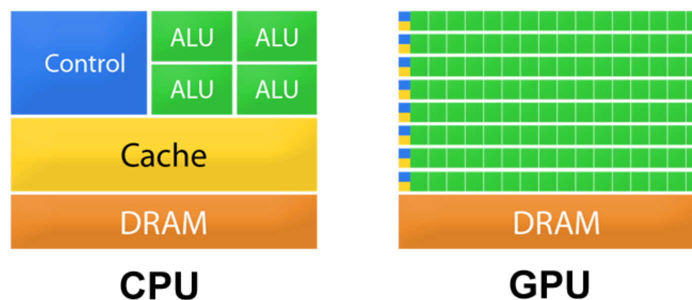


Figura 3.1: Diagrama de ocupación de los recursos en el área del chip CPU y GPU. Las áreas en color naranja indican la ocupación de la memoria principal, en amarillo las memorias caché, en azul la lógica de control y en verde los recursos de cómputo.

3.4 CUDA: Compute Unified Device Architecture

CUDA (*Compute Unified Device Architecture*) es la propuesta que presenta *Nvidia* para poder realizar cómputo GPGPU en sus dispositivos, a partir de la serie *GeForce 8*. Se presenta como una arquitectura *ManyCore* con un juego de instrucciones específico asociado, llamado PTX. Además *Nvidia* facilita un nuevo modelo de programación paralelo, un lenguaje basado en una extensión de C y C++, una serie de compiladores y herramientas para el desarrollo de las aplicaciones.

El objetivo de CUDA es permitir al programador aprovechar al máximo la capacidad de cómputo de los cientos de núcleos que tienen las GPUs. Esto se consigue lanzando decenas de miles de threads de forma concurrente que ejecutan el mismo código sobre diferentes conjuntos de datos. A este tipo de paralelismo *Nvidia* lo ha denominado SIMT (*Single-Instruction Multiple-Thread*). [20]

Actualmente CUDA se considera la solución para realizar cómputo GPGPU más extendida y utilizada entre la comunidad científica. *Nvidia* se encuentra dando un amplio soporte a la tecnología CUDA, poniendo a disposición de los usuarios amplia documentación, eventos, soporte en sus foros de opinión y actualizaciones regulares de su API y herramientas. Algo muy destacado es que está garantizando una compatibilidad hacia delante en sus futuros dispositivos, es decir, compatibilidad de aplicaciones actuales en sus futuras plataformas. Además comercializa versiones de GPUs de alto rendimiento para sistemas HPC (*High-performance computing*).

3.5 Visión general de la arquitectura GPU Fermi

Los procesadores GPU *Fermi* son la última revisión de la arquitectura con soporte CUDA de *Nvidia*. Presentan una arquitectura de tipo vectorial, esto es que diferentes recursos realizan la misma operación simultáneamente sobre conjuntos de datos distintos.

La arquitectura *Fermi* consiste en varios multiprocesadores SM (*Stream Multiprocessors*) interconectados entre ellos por un bus compartido, compartiendo además una caché de datos L2 y una memoria global GDDR5. (figura 3.2)

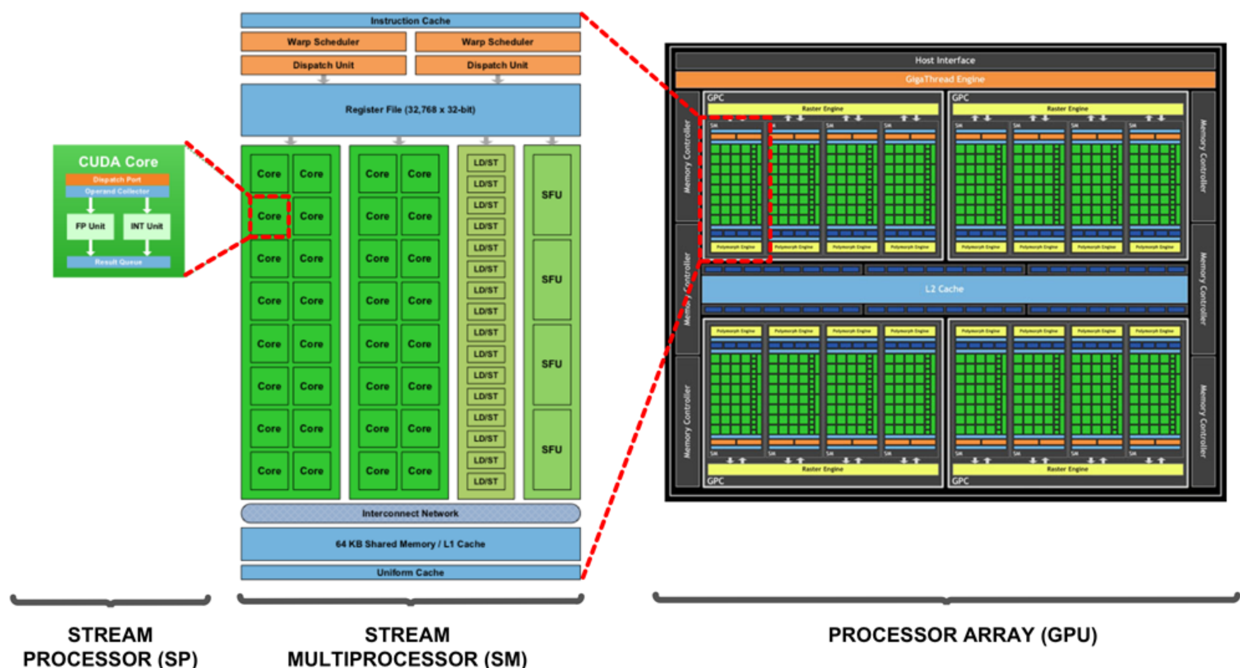


Figura 3.2: Diagrama de la arquitectura CUDA Fermi en la que se aprecia la jerarquía de los diferentes recursos de cómputo y memoria.

Estos Multiprocesadores están agrupados en TPCs (*Thread Processing Clusters*) en grupos de 4 SM. Entre los SM de un TPC se comparte una caché L2 para memoria constante y una L1 para memoria caché de texturas.

Cada uno de los 16 SM cuenta con diversas unidades de cómputo y memoria:

- 32 SPs (Stream Processors) divididos en 2 grupos de 16. Este grupo se denomina *line*. Cada *line* de 16 SP debe ejecutar simultáneamente la misma instrucción, trabajando todas al unísono sobre datos distintos. Los SPs pueden realizar operaciones enteras y en punto flotante.
- 8 SFUs (*Special Function Units*). Estas unidades están reservadas para realizar cálculos aritméticos, denominados transcendentales, como por ejemplo raíces cuadradas, sinus, etc. Este recurso también tiene estructura vectorial y deben ejecutar los 8 simultáneamente la misma operación.
- 16 LD/ST (*Loads/Stores*) estos recursos realizan las peticiones a memoria. Del mismo modo que los recursos anteriores del SM, éstos también deben ejecutar la misma instrucción de forma simultánea.
- Una memoria compartida dentro del SM de 64KB configurable como 16KB/48KB o 48KB/16KB para ejercer como caché L1 de datos y como una memoria de tipo *scratch*-pad, denominada *Shared Memory*.
- Además dentro del SM todos los recursos comparten una memoria L1 para constantes y otra L1 para texturas. [27]

La cantidad de SMs en la GPU varia según la gama del dispositivo, por ejemplo, el dispositivo *GeForce GTX 580* utiliza arquitectura Fermi GPU y presenta un total de 512 CUDA cores (16SM x 32 SP) y un ancho de banda con la memoria global de unos 190GB/s. El tamaño de la memoria global en este modelo es de 1536MB pero podemos encontrar dispositivos de alto rendimiento en la gama profesional de hasta 6GB.

3.6 Modelo de ejecución en GPU

Los algoritmos presentan etapas de su ejecución con diferentes comportamientos. Estas etapas pueden tener una naturaleza serie o por el contrario paralela. Una implementación ideal de un algoritmo para un sistema de cómputo GPGPU, es identificar estas etapas y ejecutar cada una de ellas en el ámbito de la CPU o GPU con el fin de mejorar la eficiencia. En la figura 3.3 se muestra un ejemplo de este modelo: un programa que presenta 2 etapas serie y 2 paralelas, cada una de ellas ejecutada en un ámbito distinto CPU o CPU.

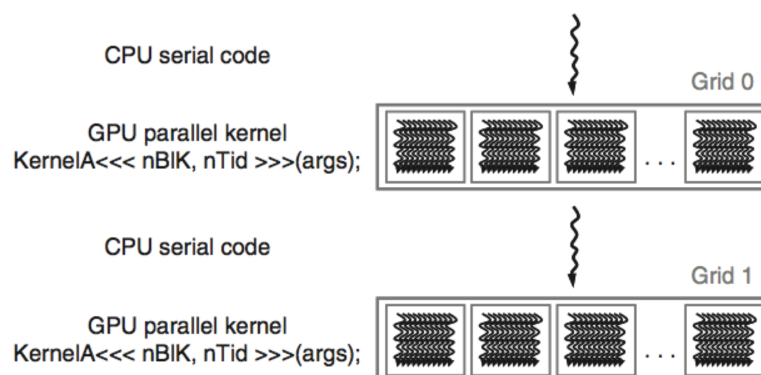


Figura 3.3: Ejemplo del modelo de ejecución de un programa con 4 etapas que es paralelizado en CUDA.

Actualmente, una GPU de *Nvidia* necesita de una CPU de apoyo, también denominada *host*, para poder realizar todas las tareas relacionadas con la ejecución de un algoritmo. Esto es debido a que la GPU, denominada *device*, no tiene un soporte directo para poder acceder a la memoria principal del sistema, los dispositivos o las interfaces agregadas al sistema, como podrían ser discos duros o la propia red. Una comunicación entre la CPU y la GPU se realiza mediante el puerto *PCI Express*.

La etapa del programa que se desea ejecutar en una GPU es denominada *kernel.* Para lanzar un *kernel* en la GPU debe realizarse una comunicación entre la CPU y GPU [16]:

1. El código comienza su ejecución en la CPU e inicializa todas las estructuras necesarias para la ejecución.
2. Se reserva el espacio de memoria necesario en la CPU para los datos de salida y el resultado. Finalmente se inicializan los datos necesarios para el cómputo.
3. La CPU reserva el espacio de memoria en la GPU para los datos de entrada y el resultado.
4. La CPU transfiere los datos de entrada de la memoria principal de la CPU a la memoria de la GPU.
5. La CPU lanza la ejecución del *kernel* en la GPU.
6. Una vez finalizada la ejecución la CPU se copia los datos del resultado generado por la GPU a su memoria principal (CPU).
7. La CPU libera los datos reservados en la memoria de la GPU.

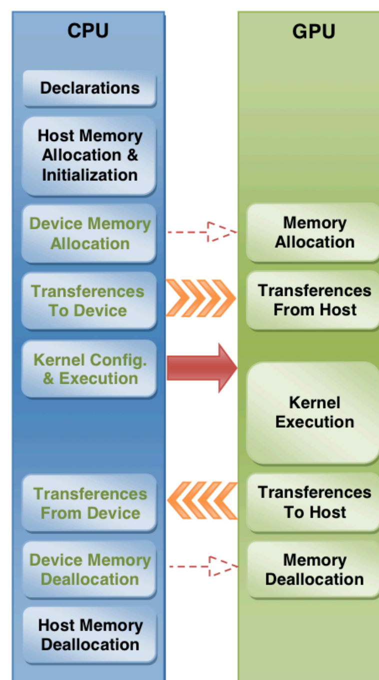


Figura 3.4: Ejemplo del modelo de ejecución de un CUDA *kernel* y la sincronización que realiza la CPU y GPU.

Las GPUs tienen una cantidad de memoria principal un orden de magnitud menor que las CPUs. Es posible que esto suponga una limitación si el número de datos a computar es mayor que el tamaño de la memoria. Deberá adaptarse la aplicación y desarrollar algoritmos paralelos que realicen el cómputo de los datos en diferentes iteraciones. Cada una de las

iteraciones lanzará un *kernel* para partir el cómputo y los datos en diferentes *kernels* que serán lanzados en la GPU.

3.7 Modelo de paralelismo a nivel de threads

Las GPUs de *Nvidia* tienen un modelo de paralelismo de tipo jerárquico que permite escalar y utilizar en los algoritmos millones de threads sin que suponga un sobrecoste aparente a la ejecución. Además, este tipo de modelo de paralelismo permite una mayor abstracción del paralelismo de la aplicación sobre el paralelismo del hardware, facilitando compatibilidad del código entre diferentes dispositivos GPU.

El modelo de paralelismo que utiliza CUDA está basado en una estructura jerárquica donde los threads se agrupan en diferentes estructuras. Estos agrupamientos definen el alcance de las definiciones (ver capítulo 3.8) y las cooperaciones entre threads.

La figura 3.5 muestra la jerarquía de threads existente en CUDA. Todos los threads que componen el *kernel* están agrupados en un conjunto denominado *grid*. Los threads de un grid son agrupados en conjuntos denominados *bloques.* Cada *bloque* contiene un número de threads. [19]

Los *bloques* deben ser totalmente independientes entre ellos, ya que la GPU no asegura una ejecución de los bloques en un orden concreto ni permiten realizar sincronizaciones de tipo *barrier* entre *bloques*. Todos los bloques deben ser del mismo tamaño (número de threads).

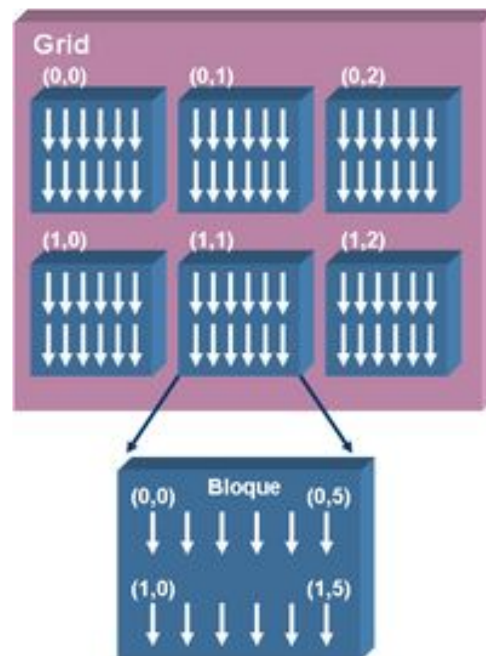


Figura 3.5: Modelo jerárquico del paralelismo a nivel de thread.

Cada thread del *bloque* está identificado por un id y cada *bloque* del *grid* está identificado por otro id. De esta forma podemos identificar de forma inequívoca cada uno de los threads. Esta

identificación permite al programador dividir el cómputo entre los diferentes threads asociando a cada thread un conjunto de datos sobre los que se realizará el cómputo.

3.8 Jerarquía de memoria

En las GPUs, donde existen decenas de miles de threads ejecutándose simultáneamente, es crítico un uso adecuado de la memoria para poder obtener alto rendimiento. Aunque las GPUs están diseñadas para ocultar las latencias a memoria, el ancho de banda cuello de botella potencial, ya que es muy probable encontrarse con miles de threads realizando peticiones simultáneas a memoria. [22]

Es por ello que las GPUs implementan un amplio conjunto de diferentes memorias que deben ser utilizadas correctamente. Cada una de ellas tiene unas características, por lo que se debe tener un mínimo de conocimiento de donde se almacenan los datos para llegar a ejecutar programas de forma eficiente.

En CUDA existen 6 tipos de recursos para almacenar la información: registros, memoria local, memoria constante, memoria compartida, memoria global y memoria de texturas.

En la siguiente tabla se especifica que alcance, la visibilidad, localización y tiempo de vida que tienen las variables que son almacenadas en cada una de las memorias.

Memoria	Localización	Acceso GPU	Visibilidad	Tiempo de vida
Registros	Dentro del chip	R/W	Thread	Kernel (thread)
Local	Fuera del chip	R/W	Thread	Kernel (thread)
Compartida	Dentro del chip	R/W	Bloque	Kernel (bloque)
Global	Fuera del chip	R/W	CPU y GPU	Programa
Constante	Caché dentro del chip	R	CPU y GPU	Programa
Texturas	Caché dentro del chip	R	CPU y GPU	Programa

Los registros son la memoria más rápida de la GPU. Se utilizan para almacenar datos intermedios en las operaciones y que van a ser utilizados con una alta frecuencia.

La memoria local permite almacenar datos privados a cada thread, compartiendo la misma visibilidad y tiempo de vida que los registros. Es útil cuando no quedan registros disponibles en el SM, entonces los datos privados de cada thread son almacenados en la memoria global. A esta situación se la denomina *spilling* de registros.

La memoria global es una memoria que se encuentra fuera del chip y conceptualmente es equiparable a la memoria RAM principal de la CPU. Tiene un tamaño abundante, hasta 6GB, pero el acceso y ancho de banda es más reducido que en el resto de memorias. En la última generación *GPU Fermi*, la memoria global incluye una jerarquía de 2 niveles de caché que beneficia a los accesos de tipo más aleatorio.

La memoria compartida, que se encuentra dentro del chip, es un recurso muy preciado en la GPU. Esta memoria permite una compartición de datos entre los threads de un mismo bloque, por lo que posibilita realizar técnicas de colaboración entre threads de forma eficiente. Por el contrario, esta memoria no tiene una lógica añadida como las cachés por lo que el programador debe gestionar los datos de forma explícita, indicando las transacciones entre la memoria global y la memoria compartida.

La memoria de texturas históricamente se ha utilizado para leer texturas de imágenes en aplicaciones graficas. Es un tipo de memoria que intenta suplir la funcionalidad de la memoria caché constante. Esta memoria de texturas está diseñada para optimizar accesos con localidad de datos en 2 dimensiones. Si el acceso a los datos a estas cachés falla, entonces se debe acceder a buscar el dato a la memoria global. En la última generación *GPU Fermi*, la memoria global incluye una jerarquía de 2 niveles de caché. Debido a este cambio, los desarrolladores comienzan a evitar utilizar la memoria de texturas debido a la complejidad que entraña a la hora de ser programada.

La memoria constante se puede ver como una pequeña memoria dentro del chip funcionando como caché, la cual tiene un acceso tan rápido como los registros. Del mismo modo que la memoria de texturas, la memoria constante es una memoria donde los datos son únicamente de lectura. Los datos de ambas memorias son alojados por la CPU antes del inicio del *kernel* y son visibles durante toda la ejecución del programa.

En la figura 3.6 se puede ver un diagrama que relaciona la jerarquía a nivel de threads con la visibilidad de los datos en memoria. Un registro sólo puede ser accedido por su thread, la memoria compartida sólo puede ser accedida por los threads de un mismo bloque y la memoria global, de texturas y de constantes puede ser accedida por cualquier thread del *grid*.

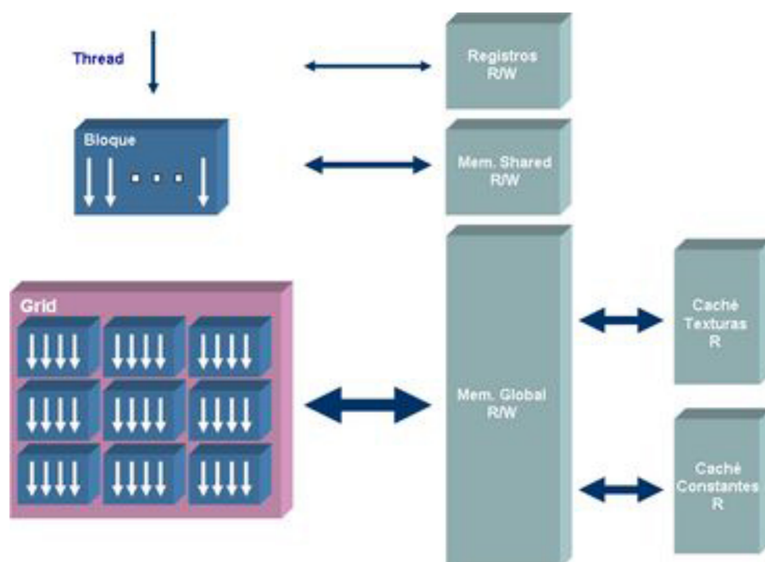


Figura 3.6: Jerarquía de memoria de CUDA

3.9 Principales factores críticos en el rendimiento

En este subcapítulo se detallan muy brevemente los factores que tienen un fuerte impacto en la eficiencia de las aplicaciones para GPU. Se dan unas pinceladas que muestran la gran diferencia entre las problemáticas de las arquitecturas *MultiCore* frente a las *ManyCore*.

3.9.1 Acceso a memoria de forma *coalesced*

Como se ha comentado en el capítulo 3.7 es importante realizar una correcta utilización de las memorias. En hecho de tener una cantidad tan elevada de threads en activo supone que el número de peticiones a memoria puede llegar a ser muy elevado.

Así pues, aunque los dispositivos GPUs permiten ocultar las latencias a memoria y disponen de un gran ancho de banda respecto los *MultiCores*, el acceso puede llegar fácilmente a ser un cuello de botella que esté limitando la potencia de cálculo de la aplicación.

Con el fin de mitigar este elevado número de peticiones, las GPUs realizan unificaciones de las peticiones a memoria que cumplan unas determinadas características. A este tipo de accesos se les denomina *coalesced*. Para permitir esta unificación, los datos de las peticiones de los threads que se han lanzado simultáneamente deben ser colindantes, como muestra la figura 3.7, donde el acceso coalesced unifica 16 peticiones en una sola. [23]

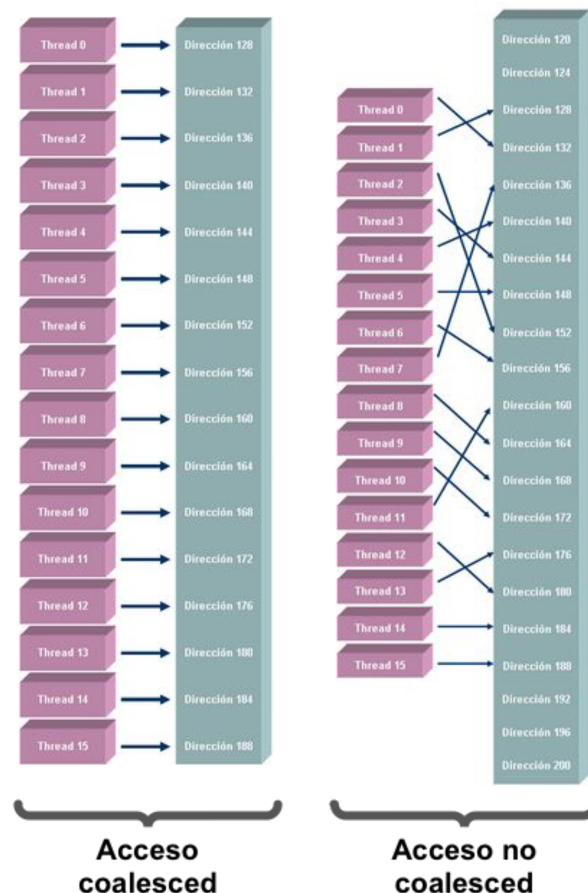


Figura 3.7: Acceso *coalesced*, unificación de las peticiones a memoria consecutivas de un conjunto de threads.

3.9.2 Shared memory

Tal y como se comenta en el capítulo 3.7, las memorias compartidas tienen una gran relevancia a la hora buscar la eficiencia en la GPU. Permiten explotar la localidad temporal y espacial de los datos y reducir las peticiones a memoria global.

Este es un tipo de memoria de tipo *scratchpad*, lo que significa que no tiene una lógica detrás como las cachés, y los datos deben ser transferidos por el programador de forma explícita. La gran virtud de estas memorias es la baja latencia y alto ancho de banda de que disponen. Además permiten realizar accesos totalmente aleatorios sin un coste agregado.

El proceso es el siguiente: el programador trae los datos a la memoria compartida, realiza el cómputo sobre ella y finalmente transfiere el resultado a la memoria global. Todos los accesos que se han realizado de la memoria shared durante este cómputo han sido evitados en la memoria global, reservando el ancho de banda para otros usos.

3.9.3 Warps y colaboración entre threads

En un *bloque*, los threads consecutivos se agrupan en conjuntos de 32, este conjunto se denomina *warp*, que se trata de una agrupación de threads que viene impuesta por el hardware [23]. Se lanzan de forma atómica 32 instrucciones iguales para cada thread que serán ejecutadas en los recursos. En la figura 3.8 se muestra las sucesivas instrucciones del *warp* que se han lanzado en los recursos.

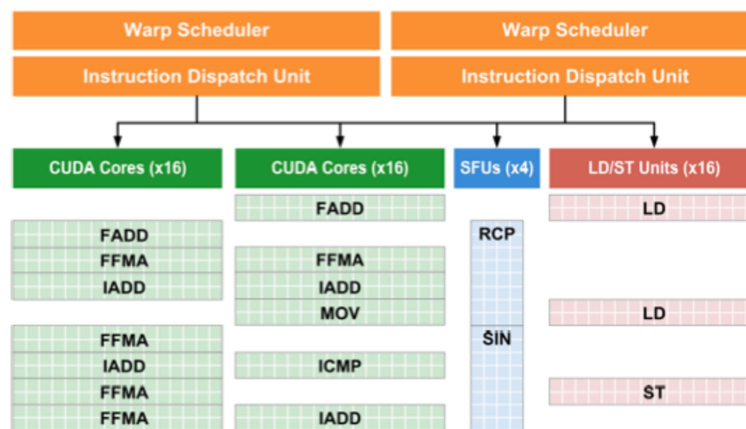


Figura 3.8: Etapa de planificación y lanzamiento de las instrucciones de los warps en los recursos del SM.

Tener en consideración este hecho puede ayudar a plantear de forma distinta el paralelismo de la aplicación. Puede ayudar a poner en practica técnicas de colaboración entre threads, como pueden ser los accesos colaborativos en memoria.

3.9.4 Divergencia

Dado que un *warp* debe ejecutar la misma instrucción para todos sus threads, en el caso de tener que ejecutar una estructura condicional es posible que cada thread tome caminos distintos en la ejecución. Esto produce una reducción de eficiencia en la ejecución, dado que la ejecución de los caminos se serializa. [25]

En la figura 3.9 se presenta un ejemplo de esta situación, donde la estructura condicional del *if* en el *warp* con threads del 0 al 31 toma 2 caminos distintos en la ejecución, por lo que se ven serializados. En el caso del *warp* con threads del 32 al 63 no existe esta divergencia debido a que al evaluar la condición todos los threads toman el mismo camino.

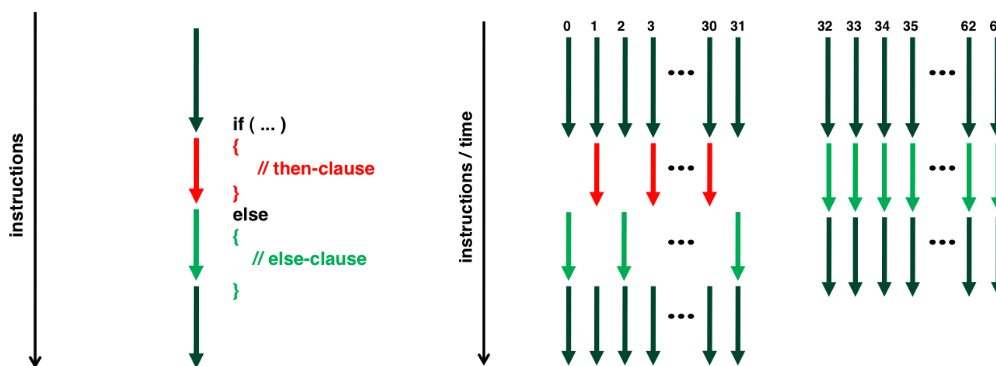


Figura 3.9: Efecto de serialización debido a la divergencia en la ejecución de una estructura condicional

3.10 Puntos clave del capítulo

En este apartado se enumeran las ideas más importantes que se han presentado en este capítulo:

- Debido a limitaciones físicas en los diseños de los procesadores se ha llegado a un punto en que las frecuencias de reloj de los computadores no pueden continuar aumentando. Debido a esto, los procesadores comienzan a evolucionar hacia otras direcciones, añadiendo un mayor número de recursos de cómputo, que deben ser explotados modificando los algoritmos para una ejecución paralela.

- Las arquitecturas paralelas *ManyCore* que conocemos hoy en día han venido propiciadas por una evolución en los dispositivos GPU que ha permitido ejecutar código de propósito general.

- La comunidad científica muestra un gran interés en beneficiarse de estas arquitecturas paralelas. Un ejemplo son las aplicaciones de alineamiento de genomas para GPUs que se analizan en este proyecto.

- Las actuales GPUs presentan cientos de cores y un ancho de banda mayor que el de las arquitecturas *MultiCore*. CUDA permite implementar y ejecutar aplicaciones en las GPUs de *Nvidia* y que las aplicaciones se beneficien de estas características de los *ManyCores*.

- La arquitectura *GPU Fermi* de *Nvidia* tiene diferentes recursos de cómputo agrupados en diferentes niveles. La GPU contiene *Streams Multiprocessors* que a su vez contienen *Stream Processors*.

- Las GPUs precisan de una unidad CPU de apoyo para la ejecución de los programas, debido a que no pueden acceder a la memoria principal de la CPU ni a dispositivos externos. Por lo tanto la CPU se encarga de realizar la inicialización, reserva de memoria, transferencia de datos y lanza la ejecución del código en la GPU.

- La arquitectura CUDA presenta un modelo de paralelismo a nivel de threads con una jerarquía. Todos los threads que componen el *kernel* están agrupados en un conjunto denominado grid. Los threads de un grid son agrupados en conjuntos denominados bloques. Cada bloque contiene un número de threads.

- Las GPUs están provistas de 6 tipos diferentes de memoria que deben conocerse debido a la importancia que tienen para poder ejecutar eficientemente las aplicaciones.

- Por contra, existen multitud de aspectos que inciden directamente en la eficiencia de estas arquitecturas, como son los accesos no coalesced, el uso de memorias compartidas o la divergencia en la ejecución.

CAPÍTULO 4 – APLICACIÓN MUMMER EN ARQUITECTURAS MULTICORE Y MANYCORE

4.1 Resumen

Debido a la inviabilidad de aplicar métodos exactos en el análisis de genomas, actualmente existe un especial interés en desarrollar aplicaciones y algoritmos específicos con soluciones aproximadas para tratar el problema del *WGA.*

Tal como se comenta en el capítulo 2.6.2, las diferentes herramientas de análisis de genomas disponibles actualmente utilizan una metodología muy similar, dividiendo el proceso de análisis en 3 etapas (fig. 2.10).

De entre todas estas herramientas de análisis existe una ampliamente utilizada por la comunidad científica, llamada *MUMmer*.

En este capítulo se detallan los procesos que realiza en cada etapa el algoritmo *MUMmer* para lograr el análisis comparativo entre genomas. Finalmente se detallan en alto nivel 3 implementaciones de MUMmer para arquitecturas Multicore y arquitecturas ManyCore. Estas implementaciones son *MUMmer 3*, *MUMmerGPU 2* y *MUMmerGPU++.*

Así pues, este capítulo presenta el estudio de los procesos que llevan acabo diferentes implementaciones de *MUMmer*. Este estudio es necesario para el posterior análisis de los datos experimentales que se extraen en el capítulo 6.

4.2 Definición y objetivos de la aplicación MUMmer

MUMmer es una aplicación de código abierto con gran aceptación entre la comunidad científica para realizar análisis genómico de secuencias. Permite abordar la problemática del WGA con un tiempo de proceso y precisión razonables. [12]

La aplicación MUMmer hace uso de una particularidad genética comentada anteriormente. Parte del supuesto de que los dos genomas a analizar tienen una relación evolutiva muy estrecha. Este hecho permite utilizar heurísticas de forma eficiente con el fin de acelerar el proceso de análisis.

El objetivo de MUMmer es proporcionar un alineamiento entre dos genomas que permita identificar cambios a gran escala entre dos genomas completos, cambios tales como repeticiones en tándem, inversiones a gran escala o translocaciones. (fig. 2.9)

MUMmer utiliza un enfoque en el cual, a partir de la localización de coincidencias exactas entre genomas y un posterior filtraje y ordenación, puede formar la base de un alineamiento que posteriormente será refinado. Para cada una de las fases el sistema utiliza una combinación de 3 ideas: un algoritmo de indexación, un algoritmo de ordenación denominado LIS junto con un proceso de clusterización y el alineamiento *Smith&Waterman*. Estas ideas son combinadas en el sistema para permitir alineaciones a gran escala.

4.3 Procedimiento del algoritmo MUMmer para la resolución de la problemática WGA

Tal como se comenta en el capítulo anterior, los algoritmos de alineamiento que resuelven el WGA suelen dividir el proceso en 3 etapas, cada una de las cuales realiza un procesado distinto a los datos de entrada hasta finalmente formar el alineamiento final (fig 2.10).

MUMmer, siendo un algoritmo de resolución aproximado, también comparte estas 3 fases diferenciadas [14]. A continuación, se describen las diferentes etapas que tienen implicación en *MUMmer*:

La primera etapa, denominada *String Matching*, realiza un preprocesamiento de la cadena de referencia mediante métodos de indexación de textos. Posteriormente la etapa realiza una búsqueda de las regiones exactas entre los 2 genomas de entrada haciendo uso del texto indexado. Esta fase de *String Matching* es la que más tiempo ocupa con diferencia en la ejecución, el proyecto centrara su análisis en esta etapa. En el capítulo 4.4 se centra en los detalles.

La segunda etapa, denominada *clusterización*, realiza un filtrado de todas las coincidencias que retorna la primera etapa. Para ello utiliza un método de ordenación denominado LIS (*Longest Increase Subsequence*). Este método agrupa las coincidencias de forma creciente en el orden de aparición del genoma original. Permite un filtrado de las coincidencias seleccionando una de las ordenaciones y descartando el resto. La selección del subconjunto se realiza en función de la longitud de las coincidencias que lo componen. Así pues, el subconjunto que representa el futuro alineamiento es aquél en el que sumadas las longitudes de sus coincidencias sea el máximo entre el resto. (fig. 4.1)

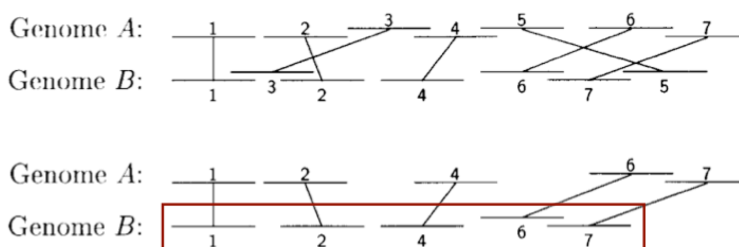


Figura 4.1: Se muestra la secuencia de referencia acaaacatat$ y su Suffix-Tree.

El conjunto resultante que representa el futuro alineamiento es sometido a un proceso de clusterización, el cual agrupa las coincidencias de la lista en función de la distancia en que son separadas. Todas las coincidencias se agrupan si presentan una separación menor que un índice de separación máximo indicado por parámetro. La separación se calcula básicamente como el número de bases insertadas o eliminadas en la secuencia de referencia entre una coincidencia de la lista y su coincidencia siguiente. (fig. 2.11)

Así pues, al finalizar la segunda etapa de filtrado y clusterización se obtiene una lista de agrupaciones de coincidencias. Cada agrupación de coincidencias es afín a corresponder a una región conservada de la referencia. Esta etapa está desarrollada en *MUMmer* como un conjunto de scripts desarrollados en PERL que se denominan como NUCmer y PROmer.

La tercera etapa, denominada *GAP closure*, realiza un refinamiento del alineamiento que se obtiene en la etapa anterior. Un *GAP* se define como una interrupción en la coincidencia debido a una inserción o eliminación de un pequeño conjunto de bases[1]. Para ello realiza un alineamiento local dentro de cada agrupación de coincidencias a fin de extender el alineamiento en las separaciones que se presentan entre las coincidencias. *MUMmer* utiliza una versión modificada del algoritmo Smith&Waterman para procesar estos alineamientos locales.

El alineamiento final se representa en ficheros de texto plano de gran tamaño. Visualizadores tales como Java Viewer o DisplayMUMs, proporcionados por los propios desarrolladores de MUMmer, permiten al biólogo interpretar los datos procesados por *MUMmer*. [14]

4.4 Etapa de *Matching*: la búsqueda de coincidencias exactas

En el momento de realizar el proceso de comparación entre los dos genomas A y B, uno de ellos es tomado como *referencia* (denominada S) y el otro genoma como un *conjunto de queries* (denominado Q). De este modo, una query del conjunto Q se denomina T.

Un *match* se define como una coincidencia de un sufijo T_i con un sufijo S_j. Un sufijo T_i es un sufijo de una query T que comienza en la posición i de T. Por otro lado, un sufijo S_j es un sufijo de la referencia S que comienza en la posición j de S.

A diferencia del *match*, un *maximal match* debe cumplir que los caracteres colindantes a los extremos no coincidan entre las dos secuencias.

En la figura 4.2 se muestran los *match* y *maximal match* resultantes de comparar la secuencia *referencia* S=ctatatc con la *query* T=gtatatta. Las coincidencias indicadas como 1, 2 y 3 son *match*, por el contrario, únicamente la coincidencia 3 se considera *maximal match.*

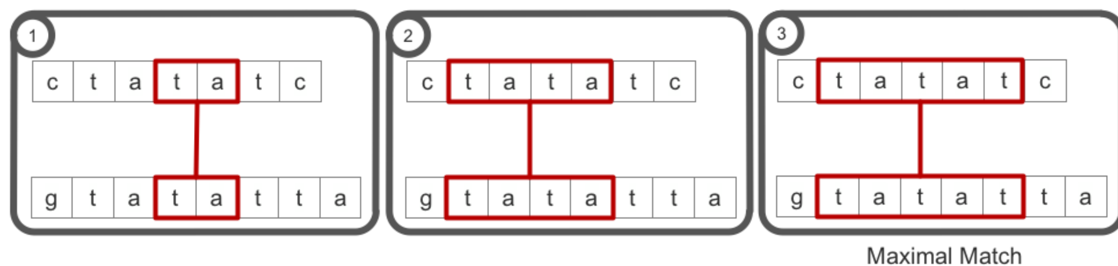


Figura 4.2: Ejemplo de *match* y *maximal match* entre las secuencia *referencia* S=ctatatc con la *query* T=gtatatta

Así pues, la etapa de *String Matching* queda definida de forma que dado un *conjunto de queries* Q se deba encontrar para cada *query* T todos los *maximal match* con una longitud mínima (denominada *minmatch*) en una cadena de *referencia* S.

La etapa de *String Matching* es la etapa que consume más tiempo de ejecución. Para poder abordar este tiempo de forma eficiente se hace uso de técnicas de indexación de textos.

[1] Biológicamente estos GAPs suelen ser dados por 4 mutaciones: un SNP, una inserción de una o múltiples bases, una región altamente polimórfica o una repetición de bases.

4.4.1 Representación de los datos

Los datos de entrada y salida que utiliza MUMmer para generar el alineamiento entre los genomas corresponden a 3 ficheros de gran tamaño en texto plano. Dos de ellos corresponden a la entrada, uno para el genoma de *referencia* S y el otro para el segundo genoma, que corresponde al *conjunto de queries* Q. La figura 4.3 muestra un ejemplo de los ficheros.

Los dos ficheros de entrada utilizan una representación denominada *MultiFASTA* [multifasta], para representar cada secuencia utilizan un identificador precedido del caracter “>” y la secuencia correspondiente. [13]

El fichero de salida contiene las coincidencias entre las secuencias de entrada. Cada coincidencia está representada por el identificador de la query T, la posición en la referencia S, la posición en la query T actual y el número de bases que comparten.

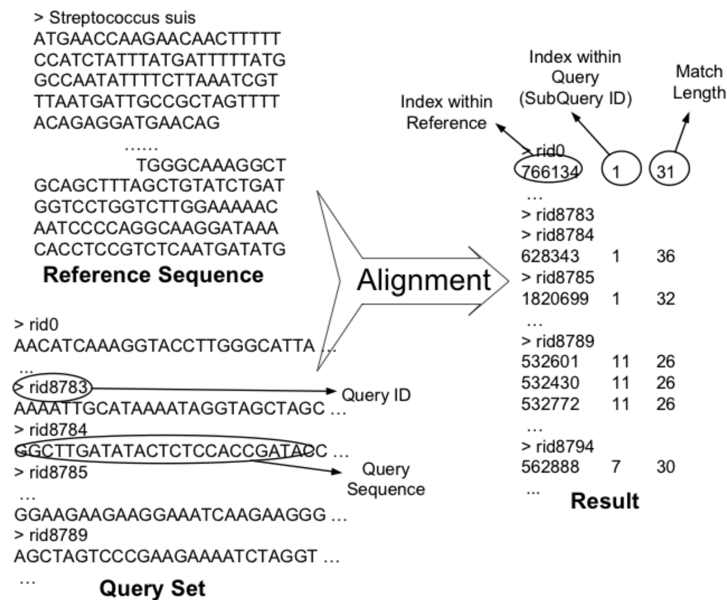


Figura 4.3: Ejemplo de los 2 ficheros de entrada (izquierda) y el salida (derecha) que utiliza MUMmer.

La división del genoma B en un *conjunto de queries* Q beneficia a las implementaciones paralelas de MUMmer, permitiendo las búsquedas de los *maximal match* de cada *query* T de forma concurrente [13]. Las *queries* T están divididas de forma solapada, con un tamaño de solapamiento de *MinMatch*-1: este tamaño asegura poder encontrar los mismos *maximal match* que una versión serie. La figura 4.4 muestra un ejemplo de este solapamiento para un valor 9 de *minmatch*.

Minmatch es parámetro de entrada en *MUMmer* que debe ser introducido por el usuario e indica el tamaño mínimo de los *maximal match* que se deben encontrar.

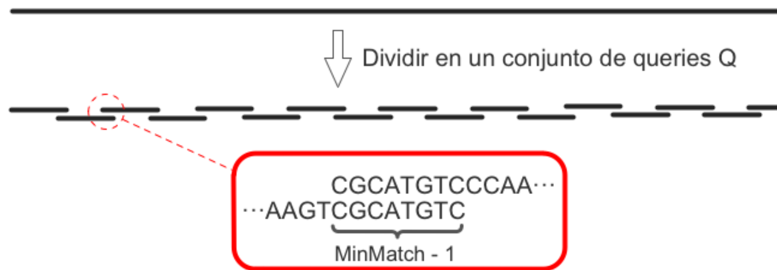


Figura 4.4: Ejemplo de división del genoma B en *un conjunto de queries* Q.

4.4.2 Concepto de Maximal Match en MUMmer

MUMmer diferencia entre las coincidencias *maximal match* que encuentra para realizar un tratamiento diferente para cada una de las coincidencias. Los *maximal match* que encuentra son 4 tipos distintos:

- MUM: Maximal Unique Match, es un *maximal match* que aparece una única vez en ambas secuencias.
- MEM: Maximal Exact Match, es un *maximal match* que aparece una o más veces en ambas secuencias.
- MAM: Maximal Almost-Unique Match, es un *maximal match* que aparece una sola vez en la secuencia *referencia* y más de una vez en la secuencia *query*.
- MIM: Maximal Inverse Match, es un *maximal match* que aparece de forma inversa en la otra secuencia.

La versión 3.X de *MUMmer* utiliza únicamente MEMs en el proceso de *string matching*, ya que engloban el resto de *maximal match* (MUM y MAM) que presenta MUMmer, excepto los MIMs que precisan una búsqueda inversa del texto. Buscar los MEMs permite obtener una mayor precisión en el alineamiento final. La figura 4.5 muestra un ejemplo de búsqueda de los diferentes *maximal match*.

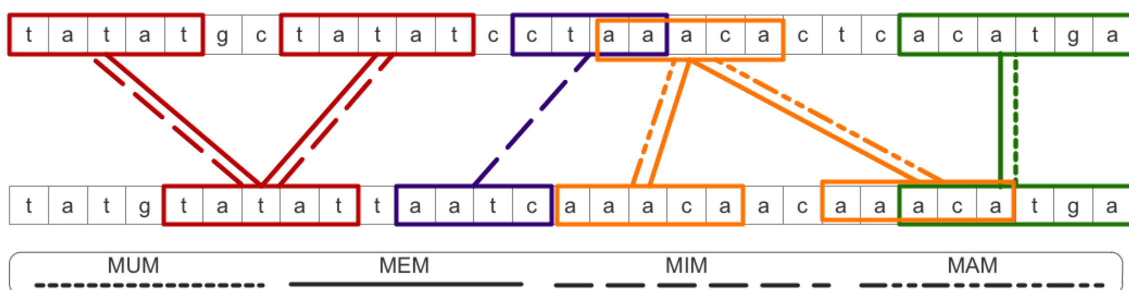


Figura 4.5: Ejemplo de los diferentes *maximal match* división del genoma B en *un conjunto de queries* Q.

En la documentación es frecuentemente encontrar que un MEM también es llamado como non Unique MEM.

A continuación en la figura 4.6, se muestra un ejemplo de búsqueda de todos los MEMs entre la secuencia de *referencia* S=acaaacatat y la *query* T=acatata. La *query* T genera 5 búsquedas distintas, una para cada uno de los sufijos mayores o igual a un de *minmatch*=3. Este número sufijos (o subqueries) corresponde a n-*minmatch*+1, siendo n el tamaño del sufijo de T. Es interesante fijarse en que la primera búsqueda con T_0 genera 2 MEMs. En caso de buscar MUMs esta búsqueda con T_0 no retornaría ningún resultado, ya que no cumple el requisito de *unique*. [11]

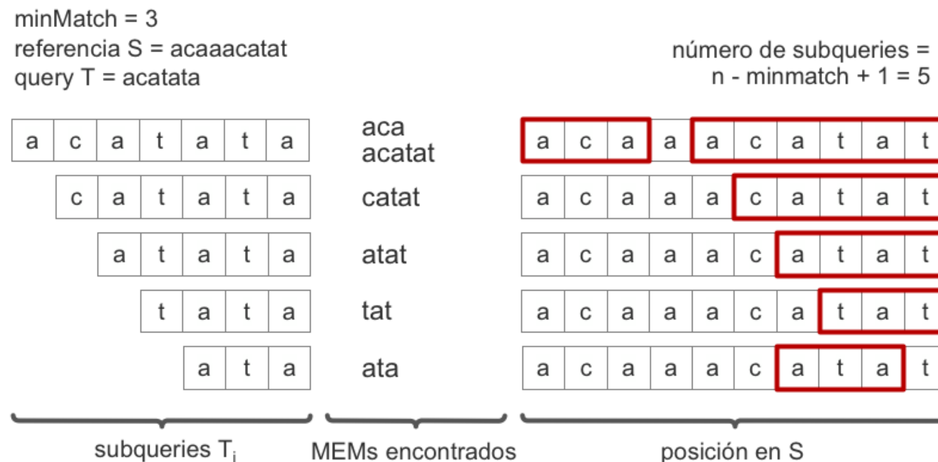


Figura 4.6: Ejemplo de búsqueda de todos los MEMs en una *referencia* S=acaaacatat de una *query* T=acatata con minmatch=3.

Así pues, un valor de *minmatch* reducido implica relajar la asunción de qué es considerado un *maximal match*, y la probabilidad de hallar un mayor número de resultados se incrementa, y viceversa. Por otro lado, al reducir el valor del minmatch incrementamos el número de *subqueries* de T a alinear y por tanto aumenta el requisito de cómputo.

4.4.3 Proceso de String Matching en arquitectura MultiCore

La primera etapa de *MUMmer* realiza el proceso de *String Matching* comentado en el punto anterior (4.3). Este proceso realiza una búsqueda de todos los MEMs que se encuentran en la *referencia* S a partir de las *queries* T.

Para realizar de forma eficiente las búsquedas de los MEMs, *MUMmer* hace uso de un algoritmo y estructura de datos de indexación para textos denominado *Suffix-tree*, detallado en el capítulo 5.3.1 y 5.3.2. *MUMmer* utiliza la librería para la generación del Suffix-Tree creada por *Stefan Kurtz* de la estructura de árbol de sufijos y para la realización de las búsquedas en el Suffix-Tree. [11]

El análisis del algoritmo sobre la arquitectura Multicore se realizará sobre la fase de *String Matching* con la versión 3.22 de *MUMmer*, una implementación serie en lenguaje C y C++, por lo que no se beneficia del TLP (paralelismo a nivel de thread) ni de la tecnología *HyperThreading* que ofrecen las arquitecturas Multicore.

Para la búsqueda de MEMs el proceso que se realiza, visto desde un alto nivel, es el siguiente:

1. Lectura de disco de la secuencia de *referencia* S almacenada en formato *multiFASTA*.
2. Generación del índice de la estructura en forma de árbol *Suffix-Tree.*
3. Lectura de una *query* T de disco.
4. Búsqueda de los MEMs de los sufijos de la T actual y la referencia S utilizando el algoritmo de búsqueda del *Suffix-Tree* y los *Suffix-Links* que se describen en el capítulo 5.3.1 y 5.3.2
5. Guardado en disco de los resultados de los MEMs encontrados en la búsqueda anterior con el formato de representación indicado en el subcapítulo 4.4.2.
6. Lectura de disco de la siguiente *query* T y vuelta al paso 4, o fin del algoritmo.

4.4.4 Diferencias entre MUMmer1 , MUMmer2 y MUMmer3

MUMmer ha tenido diferentes revisiones importantes a lo largo de su desarrollo, entre ellas las más destacadas son las versiones 1, 2 y 3 por los cambios introducidos. Los cambios que se han ido introduciendo han sido en beneficio del tiempo de ejecución, el consumo en memoria en ejecución y la calidad del alineamiento. [9]

Todas las versiones del *MUMmer* analizadas en este proyecto están basadas en *MUMmer 3.X*, por lo que los cambios entre versiones podrían inducir a confusiones en la lectura de documentación más antigua.

Los cambios más importantes han sido los siguientes:

- *MUMmer 1.X* (1999): A la hora de indexar los textos realiza una concatenación de los 2 genomas y genera un *Suffix-Tree*. Únicamente busca de MUMs.
- *MUMmer 2.X* (2002): Indexa únicamente la secuencia de referencia como un *Suffix-Tree,* por lo que reduce considerablemente el tamaño de la estructura. La búsqueda se realiza mediante el método de *Streaming String Matching* (ver capítulo 5.3.1) y dota a la estructura *Suffix-Tree* de *Suffix-Links* (ver capítulo 5.3.2).
- *MUMmer 3.X* (2004): Realiza la búsqueda de todos los MEMs en lugar de solo los MUMs. Este cambio permite mejorar la calidad del alineamiento final. Reducción del espacio ocupado por la estructura *Suffix-Tree*.

4.4.5 Proceso de String Matching en arquitecturas ManyCore

Las implementaciones de *MUMmer* paralelas para GPUs son completamente compatibles con todas las herramientas que utiliza *MUMmer*, ya que utilizan la misma representación de datos en la entrada y salida, además de contar con la misma funcionalidad.

El objetivo de una implementación de *MUMmer* paralela sobre GPUs es beneficiarse de los siguientes aspectos de estas arquitecturas *ManyCore* con el fin de reducir tiempo de ejecución:

1) El gran ancho de banda que soportan las GPUs a su memoria global. (capítulo 3)
2) El alto paralelismo que ofrecen a nivel de thread. (capítulo 3)

Por el contrario, diversos aspectos de las GPUs suponen un obstáculo de diseño a la hora de desarrollar una implementación de *MUMmer*. A continuación los aspectos más destacables:

- Los actuales modelos de GPU tienen un orden de magnitud menor de cantidad de memoria respecto a las CPUs. Esta limitación obliga a las aplicaciones a particionar el problema y realizar la ejecución en diversos *rounds*. [19]
- Las GPUs solo tienen acceso a su propia memoria, por lo que no pueden acceder directamente a la memoria del procesador ni al disco. Esto supone que la aplicación debe seguir un modelo de distintas fases: transferencia de datos de entrada a la memoria de la GPU, ejecución del código en GPU y transferencia de los resultados de GPU a CPU. [25]
- La reserva de los datos en la memoria de la GPU se debe realizar desde la CPU, por lo tanto el tamaño de los datos del resultado debe ser previsible antes de la ejecución.

Las implementaciones que se analizan de *MUMmer* para GPU son *MUMmerGPU* y *MUMmerGPU++*, ambas están desarrolladas bajo C y C++ para la parte secuencial del algoritmo y con CUDA para la parte paralela. La parte paralela corresponde a la búsqueda de los *maximal match* para la fase de *String Matching*.

La fase de *String Matching* de *MUMmer* es una operación, a priori, sencilla de paralelizar dado que las búsquedas pueden ser procesadas independientemente para cada *query* T. De este modo las *queries* T son buscadas en paralelo por cada thread, y todas las *subqueries* de una *query* son procesadas secuencialmente por un thread GPU.

Dado que el resultado del *String Matching* no es previsible y está sujeto al número de coincidencias que existan entre las dos secuencias, en la ejecución GPU se utiliza una representación comprimida del resultado de todas las coincidencias finales. Esta representación siempre toma un tamaño constante y el tamaño pasa a ser previsible a costa de realizar posteriormente un post procesado de descompresión a fin de obtener el resultado final.

Las implementaciones de MUMmerGPU y MUMmerGPU++ comparten una serie de fases en su secuencia de ejecución, en la figura 4.7 se detallan los pasos que se realizan en la ejecución a fin de ejecutar el proceso de búsqueda *String Matching* por rounds en la GPU.

La fase (A1) realiza una lectura de disco de una parte de la *referencia* S y se almacena en la memoria principal de la CPU. En (A2) se indexa la *subreferencia* y genera una estructura de indexación que representa la *subreferencia*. La fase (A3) transfiere la parte de la referencia S y sus estructuras indexadas a la GPU.

La fase (B1) realiza una lectura de disco de un *subconjunto* del *conjunto de queries* Q y se almacena en la memoria principal de la CPU. En (B2) se transfiere el subconjunto de queries a la memoria de la GPU. En la fase (B3) se tiene en memoria principal de la GPU el subconjunto de queries, la *subsecuencia* de la *referencia* S y sus estructuras indexadas. En este momento se

ejecuta el proceso de *string matching* en la GPU y genera el resultado comprimido en la memoria principal de la GPU. La fase B4 transfiere los resultados comprimidos a la memoria principal de la CPU. Finalmente en la fase B5 se realiza el postprocesado en la CPU y descomprime todos los resultados que serán almacenados en disco.

En este instante, si quedan *subconjuntos* de queries por procesar se vuelve a ejecutar la fase B1. En cambio, si todas las queries han sido procesadas se vuelve a ejecutar la fase para generar la siguiente estructura A1 a fin de procesar las siguientes *subreferencias*.

El algoritmo finalizará una vez sean procesadas todas las *subreferencias* de S.

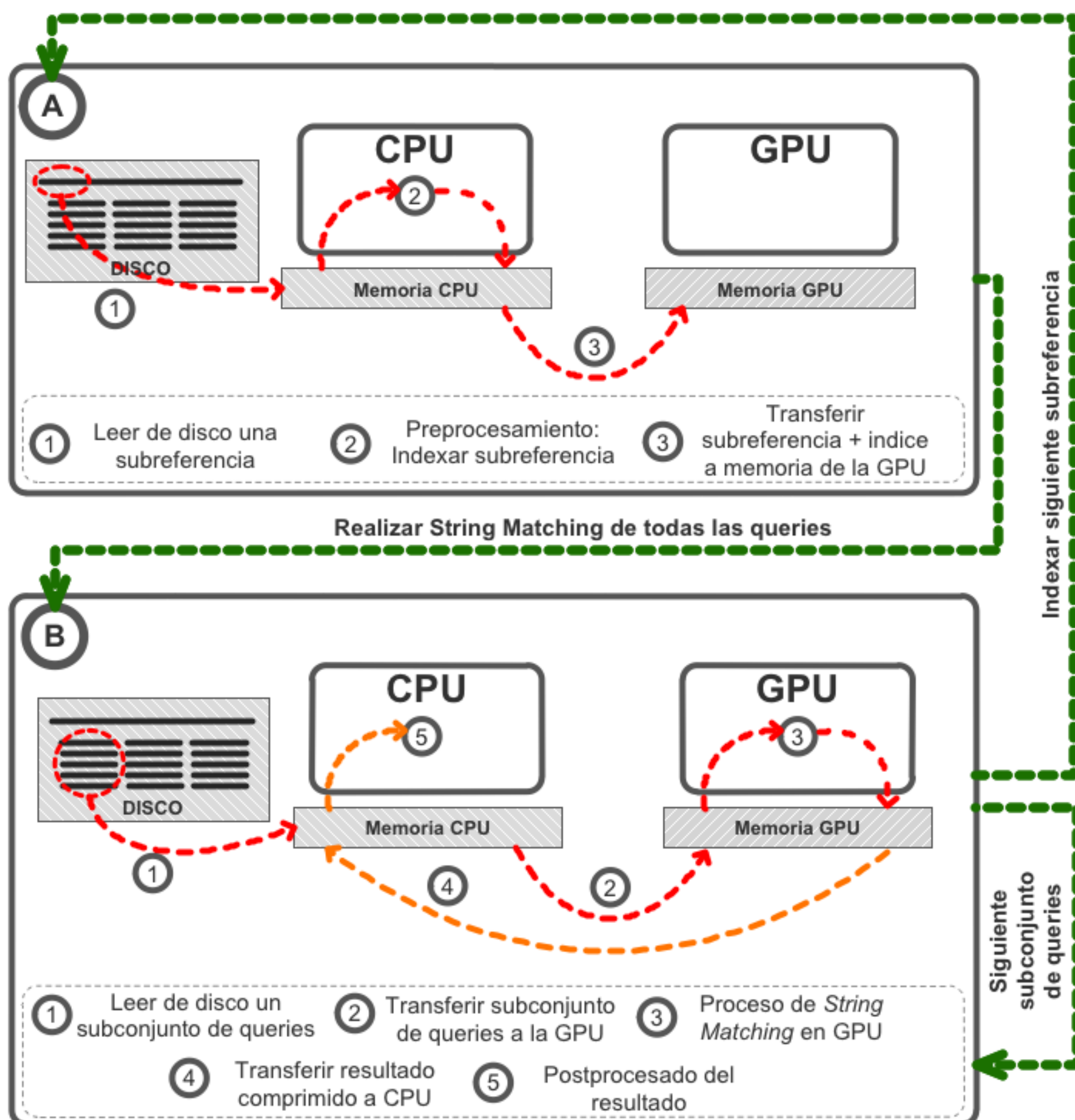


Figura 4.7: Diferentes etapas de procesado que presentan las implementaciones MUMmerGPU y MUMmerGPU++

En la figura 4.8 se refleja el pseudocódigo de este proceso de ejecución de MUMmerGPU y MUMmerGPU++ en forma de *rounds*.

```
00  subReferencias = dividirReferencia(referencia)
01  subConjuntos = dividirQueries(queries)
02
03  foreach subReferencia in subReferencias do {
04     indiceRef = indexar(subReferencia)
05     tranferirCPUaGPU(subreferencia, indiceRef)
06
07     foreach subConjunto in subConjuntos do {
08       tranferirCPUaGPU(subConjunto)
09       resCom = matchKernel(subconjunto, indiceRef, subreferencia)
10       tranferirGPUaCPU(resCom)
11       resultado = postProcesado(resCom)
12       guardarResultado(resultado)
13     }
14  }
```

Figura 4.8: Pseudocódigo del proceso de *string matching* que se ejecuta en GPU en *rounds.*

4.4.6 Diferencias entre la implementación de *MUMmerGPU* y *MUMmerGPU++*

Los cambios más significativos entre *MUMmerGPU* y *MUMmerGPU++* son los relacionados con el núcleo del cómputo, donde cada una de las implementaciones utiliza una estructura de datos de indexación diferente. Este cambio supone utilizar algoritmos de indexación, matching y postprocesado completamente diferentes.

MUMmerGPU utiliza algoritmos y estructuras de indexación de tipo *Suffix-tree* (capítulo 5.3), del mismo modo que la versión para CPU *MUMmer 3.X*, mientras que *MUMmerGPU++* utiliza *Enhanced Suffix-Arrays* (capítulo 5.4). Estos algoritmos de indexación y búsqueda se explican en detalle en el capítulo 5. [13]

Así pues, en *MUMmerGPU* las fases que difieren son:

- A2 Fase de preprocesado: En esta fase se indexa la referencia 5, en este caso se utilizan algoritmos de generación de *Suffix-Tree*, este procesado se realiza en CPU.
- B3 Fase de *matching*: Esta fase se ejecuta en GPU, aplicando el algoritmo de *String Matching* para *Suffix-Tree* y *Suffix-Link* que se detalla en el capítulo 5.3.2.
- B5 Fase de postprocesado: *MUMmerGPU* acelera esta fase realizando la ejecución de un segundo *kernel* en la GPU que realiza la descompresión. Para ello previamente precalcula en la CPU el tamaño que tendrá la salida del resultado final en GPU y realiza diversas rondas hasta que procesa toda la salida.

MUMmerGPU ha substituido la búsqueda recursiva en el árbol por una búsqueda iterativa, esto permite una búsqueda más eficiente en GPU a costa de incrementar el tamaño de las estructuras del *Suffix-Tree* añadiendo unos punteros adicionales.

En MUMmerGPU++ difieren las siguientes fases:

- A2 Fase de preprocesado: Este procesado se realiza en CPU donde se indexa la *referencia* S utilizando algoritmos de generación de estructuras *Enhanced Suffix-Array*. (capítulo 5.4)
- B3 Fase de *matching*: Esta fase se ejecuta en GPU, aplicando el algoritmo de *String Matching* para *Enhanced Suffix-Array* que se detalla en el capítulo 5.3.2.
- B5 Fase de postprocesado: Se realiza una descompresión por parte de la CPU (ver capítulo 5.5) debido a que el resultado generado por la GPU esta comprimido.

5.5 Puntos clave del capítulo

- MUMmer es una aplicación bioinformática que permite alinear genomas completos, tiene 3 fases muy diferenciadas:
  - Búsqueda de coincidencias exactas: busca todas las coincidencias máximas entre un genoma y otro que sean mayores a un determinado tamaño.
  - Filtrado y clusterización de las coincidencias: filtra y agrupa las coincidencias de la fase anterior a fin de proponer un prealineamiento entre los genomas.
  - GAP closure: refina el alineamiento anterior realizando alineamientos locales aplicando *Smith&Waterman* en la separación que se genera entre las coincidencias.

- La primera fase de búsqueda de coincidencias exactas es la que más tiempo de procesado toma en la ejecución. En su implementación se utilizan algoritmos y estructuras de indexación de textos para reducir el tiempo de búsqueda de los maximal match exactos.

- Se detallan 3 implementaciones de *MUMmer* de la primera etapa de búsquedas exactas:
  - *MUMmer 3.X* es una implementación *opensource* con ejecución serie para arquitecturas Multicore que hace uso de algoritmos de indexación de textos denominados *Suffix-Tree* y *Suffix-Links* (detallados en el tema 5)
  - *MUMmerGPU* y *MUMmerGPU++* son implementaciones paralelas para arquitecturas ManyCore, implementadas en CUDA. Realizan diferentes iteraciones de la ejecución para acabar de procesar todos los datos. [10] [13]

- Las dos implementaciones paralelas en CUDA pretenden beneficiarse del gran ancho de banda y el gran paralelismo a nivel de thread que presentan las GPUs.

- Por el contrario, el espacio de memoria reducido del que sufren las GPUs y la imposibilidad de acceder a disco o memoria principal de la CPU desde la GPU, requieren una implementación por *rounds* (o fases) del proceso de cómputo.

- Existen leves diferencias entre *MUMmerGPU* y *MUMmerGPU++* en el proceso de computar los *string match* entre los dos genomas mediante el método iterativo por rounds. Estas diferencias hacen referencia al algoritmo y estructura de datos utilizadas en cada una de las implementaciones. *MUMmerGPU* utiliza *Suffix-Tree* y *MUMmerGPU* hace uso de *Enhanced Suffix-Array* (ambos explicados detalladamente en el tema 5).

CAPÍTULO 5 – ALGORITMOS Y ESTRUCTURAS DE DATOS PARA INDEXACIÓN DE TEXTOS

5.1 Resumen

Como se comenta en el capítulo 3, la etapa 1 en la resolución de la problemática *WGA* es la parte más representativa del problema en cuanto a cantidad de tiempo y cómputo. Esta etapa se basa principalmente en una búsqueda de subcadenas dentro de una cadena de referencia genómica muy extensa.

En este capítulo se introducen 3 estructuras de indexación de textos ampliamente utilizadas en este tipo de aplicaciones bioinformáticas con el fin de acelerar el proceso de búsqueda y así lograr que toda la aplicación se vea beneficiada.

Así pues, en la primera parte se explican las motivaciones que entrañan el uso de estructuras de indexación en este tipo de problemas bioinformáticos. Después se entra en detalle en la realización de búsquedas de texto con las estructuras *Suffix Tree*, *Suffix Array* y *ESA* (*Enhanced Suffix Array*). Finalmente se realiza una comparación entre las diferentes estructuras y se detallan los puntos importantes del capítulo.

5.2 Motivaciones de las estructuras de datos para la indexación de secuencias genómicas.

La cantidad de datos genómicos disponibles ha tenido un crecimiento exponencial en los últimos años. La gestión y consulta de esta gran cantidad de datos ha supuesto la utilización de técnicas de indexación para hacer uso de los datos de forma eficiente.

En multitud de métodos de análisis para el tratamiento de datos genómicos es habitual que aparezca un subproceso de búsqueda de subcadenas en textos. A este subproceso de búsqueda se le denomina *string matching* y trata de encontrar las coincidencias de un pequeño texto, llamado *query* T_i, dentro de una cadena normalmente mucho mayor, denominada *referencia* S. Los avances aplicados en los algoritmos de *string matching* tienen una repercusión muy grande en las aplicaciones de análisis de datos.

Uno de los métodos más básicos de búsqueda de textos es a través de un algoritmo *por fuerza bruta*, que realiza una búsqueda exhaustiva en la cadena de *referencia* S. La búsqueda exhaustiva conlleva una comparación para cada subsecuencia del texto S de tamaño *m* con todas las subsecuencias del texto T_i de tamaño *n*.

Por lo tanto, *Naïve approach* tiene una complejidad espacial en memoria de $\varnothing(m+n)$, de modo que si se representa cada nucleótido con un byte éste ocupa m+n Bytes. Sin embargo, la complejidad temporal es de $\varnothing(m \cdot n)$ y se torna inasumible para búsquedas en genomas debido a su gran tamaño. [4]

Es por ello que la comunidad científica ha sentido la necesidad de generar nuevos algoritmos de indexación de textos que permitan realizar un preprocesado a la cadena de *referencia* S para realizar este tipo de búsquedas en tiempos más razonables.

Existen multitud de tipos de algoritmos de indexación de textos: *classical index*, *succint index*, *compressed index*, *self index*. Este proyecto de centra en 3 estructuras de tipo *classical index*: *Suffix Tree*, *Suffix Array* y *ESA.*

5.3 Suffix-Tree

El *Suffix-Tree* es una estructura de indexación de datos de tipo *classical index* [4]. Se basa en una estructura de datos de tipo árbol que almacena todos los sufijos de una cadena de *referencia* S. Cada sufijo comprende un único camino desde la raíz del árbol hasta una hoja del árbol (fig 5.1). [6]

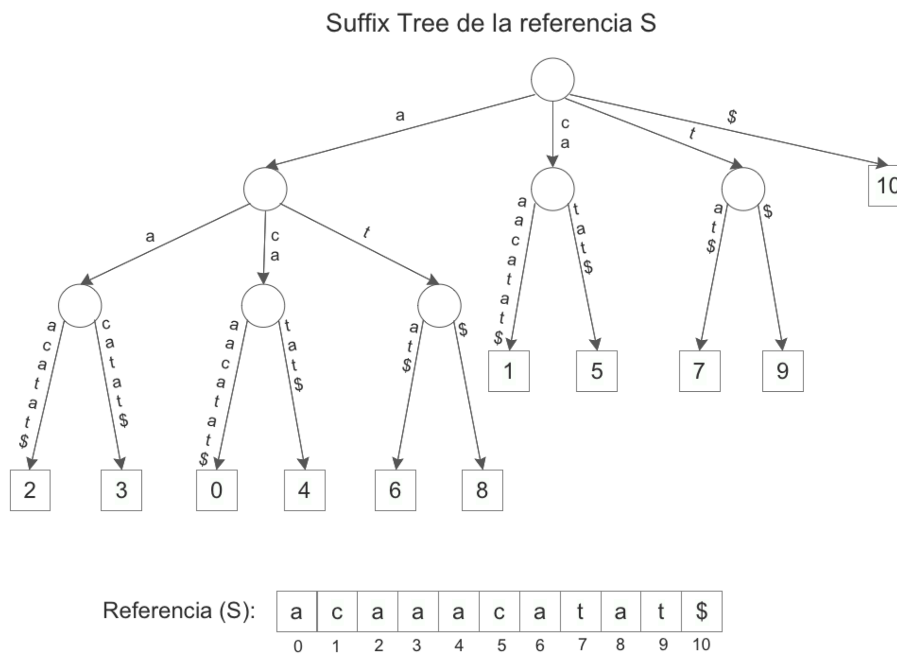


Figura 5.1: Se muestra la secuencia de referencia acaaacatat$[1] y su Suffix-Tree.

El árbol esta compuesto de nodos hoja, nodos internos y aristas etiquetadas.

- Los nodos hoja contienen el índice donde comienza el sufijo en la secuencia *referencia* S. El árbol contiene n nodos hojas, correspondientes a los n sufijos de S.
- Las aristas etiquetadas indican el contenido del sufijo[2], la concatenación de todas las aristas de un camino desde la raíz hasta la hoja proporciona el texto del sufijo (S_i). El número de aristas etiquetadas es el total de nodos menos 1.
- Los nodos internos representan el *LCP* (*Longest Common Prefix*)[3] entre todos los sufijos que comparten ese nodo. La concatenación de todas las aristas superiores al nodo intermedio proporcionan el texto del *LCP*.

---

[1] Se introduce al final de la referencia un carácter especial para facilitar el proceso de generación árbol.

[2] No es necesario que las aristas contengan el texto del sufijo, puede ser reemplazado por 2 enteros. Un índice de la secuencia de referencia que indica donde comienza la etiqueta y otro para su longitud en caracteres.

[3] Longest Common prefix es el prefijo máximo común que comparten 2 o más secuencias.

5.3.1 String matching con Suffix-Tree

Existen 2 algoritmos para realizar el proceso de string matching en el *Suffix-Tree*: uno de ellos es generando un Suffix-Tree con las 2 cadenas a comparar y la segunda forma es mediante un método denominado *streaming*.

El proceso de *streaming string matching* permite una búsqueda de una secuencia *query* T_i en la secuencia *referencia* S mediante *un Suffix-Tree* de la *referencia* S. Así pues el proceso requiere una estructura de árbol (*Suffix-Tree*) y 2 secuencias (*referencia* S y *query* T_i).

Streaming string matching realiza un recorrido desde la raíz del árbol comparando secuencialmente los caracteres de la *query* T_i con las etiquetas de las aristas. Esto indica el camino a recorrer en el árbol, pasando por el camino de mayor coincidencia entre ellos. Llegado un punto de no coincidencia, entre etiquetas y *query* T_i, cada uno de los nodos hoja por debajo de esta posición en el árbol son un resultado de matching en el proceso de búsqueda. Así, nodos hoja indican la posición de cada una de las coincidencias resultantes en la *referencia* S de la *query* T_i.

En la figura 6.2 se muestra un ejemplo de este proceso de *streaming string matching* entre la secuencia *referencia* S (acaaacatat$) y la *query* T_0 (acatata$). Se realiza el proceso de búsqueda de T_0, en Suffix-Tree previamente generado a partir de los sufijos de S.

En los pasos ①②③ se compara la etiqueta de la arista actual con la posición de T_0, en cada uno de ellos se evalúa qué camino ha de escoger. Concretamente los prefijos compartidos en cada paso son: ① acatata$, ② acatata$, ③ acatata$. En este último paso se encuentra un punto de no coincidencia, acatat**a**$, con la etiqueta de la arista. Los caracteres coincidentes son la secuencia resultante compartida entre S y T_0. Los nodos hijos que se encuentran por debajo de esta arista nos indican cuántas y en qué lugar aparecen las coincidencias en S. En este caso existe un solo nodo e indica que la coincidencia se encuentra en la posición 4 de la secuencia S (flecha naranja discontinua).

La coincidencia resultante de la búsqueda es, para S=acaaacatat$ y para $T_{0=}$acatata$. El resultado puede ser anotado como 3 enteros: la posición de inicio de la secuencia S, la de inicio de T_0 y el número de caracteres que comparten. Pro lo tanto, en este ejemplo se anota como 4 – 0 – 6.

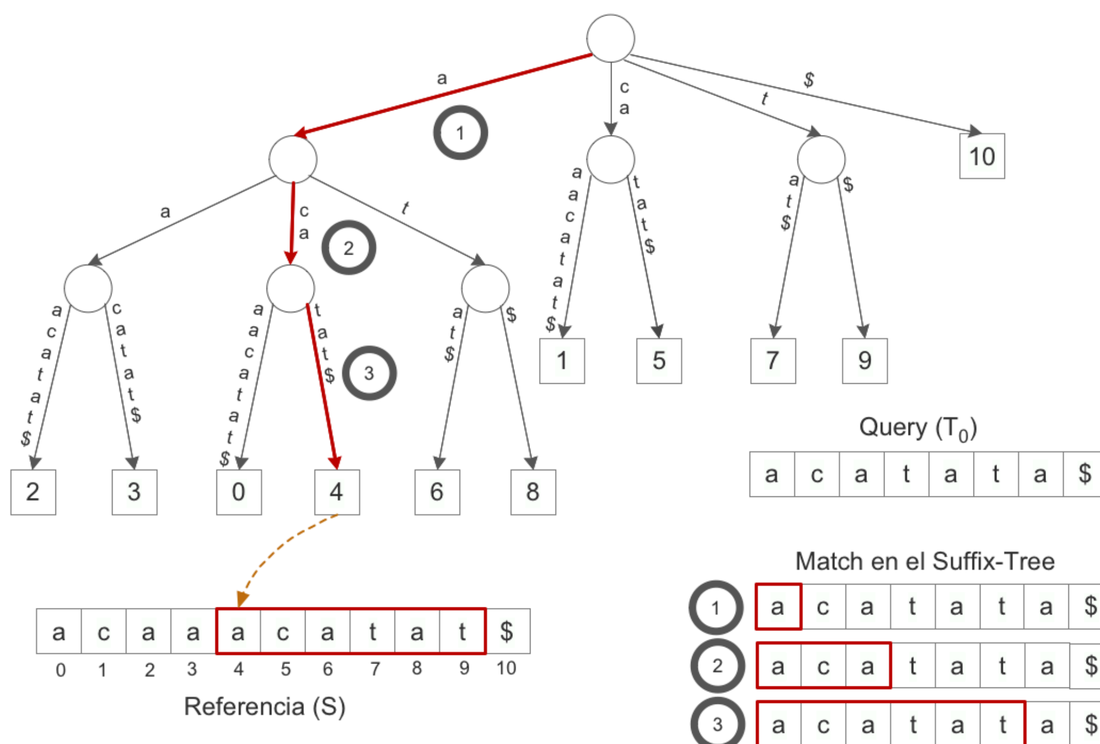


Figura 5.2: Proceso de búsqueda de la query T_0=acatata$ en el *Suffix-Tree* de la *referencia* S=acaaaacatat$

5.3.3 Suffix-Links

El algoritmo de *string matching* del *Suffix-Tree* puede ser mejorado añadiendo unos punteros adicionales entre nodos intermedios, llamados *Suffix-Links*. Una vez realizada la primera búsqueda los *Suffix-Links* permiten realizar las siguientes de forma más eficiente. [4]

Conceptualmente, un *Suffix-Link* es un puntero interno desde un nodo con un camino αw[4] hasta otro nodo con un camino *w*, donde α es un único carácter y *w* es una subcadena, concretamente un sufijo de S (fig. 5.3). Un ejemplo es el camino del nodo *aca* que apuntará al nodo *ca*.

Así pues, una vez procesada la búsqueda de la *query* actual T_i, la siguiente *query* T_{i+1} puede ser procesada de una forma más rápida utilizando *Suffix-Links*. Para que esto suceda las *queries* T_i y T_{i+1} deben compartir parte de un prefijo común. Ese prefijo común ya computado en la secuencia T_i no hace falta volver a computarlo en T_{i+1}, debido a que el *Sufix-Link* apunta al siguiente nodo del árbol que contiene la siguiente parte a computar de la cadena T_{i+1}.

En la figura 5.3 se muestra el *Suffix-Tree* de la *referencia* S anterior acaaacatat$ con los punteros entre nodos *Suffix-Links*.

[4] Este camino es la concatenación de todas las etiquetas en las aristas desde la raíz hasta el nodo.

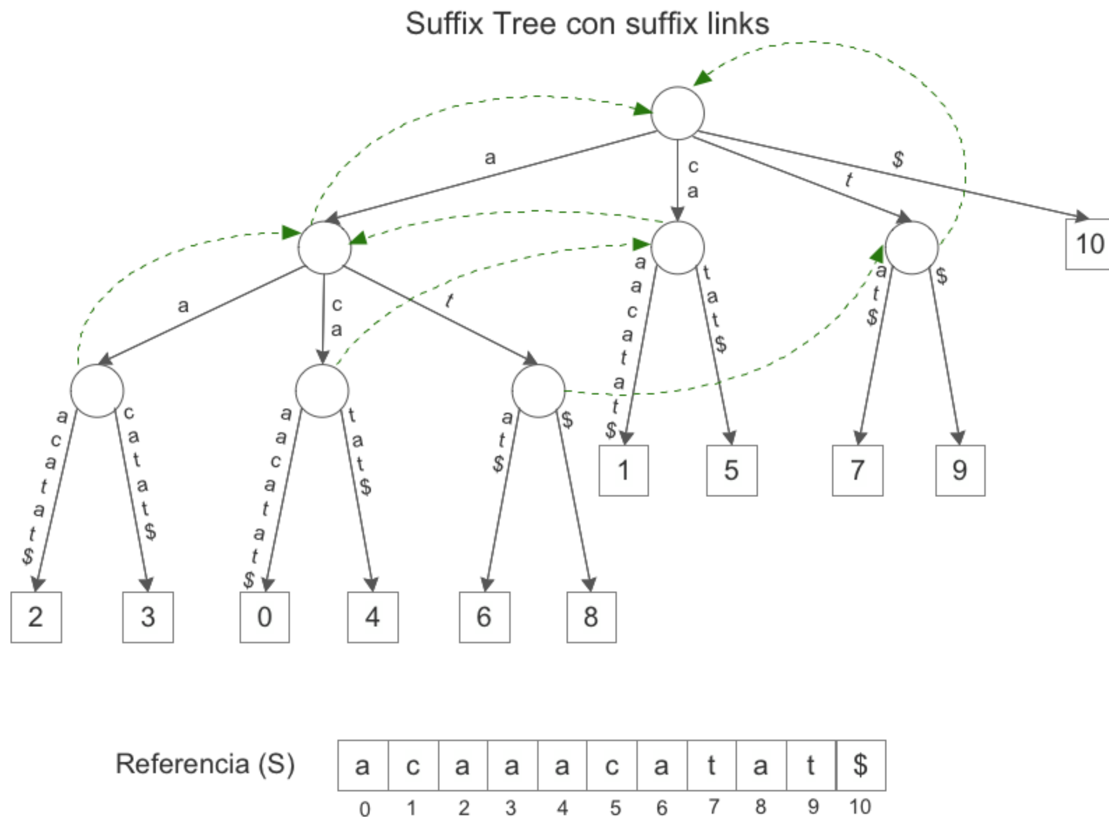


Figura 6.3: Suffix-Tree de la *referencia* S=acaaaacatat$ con los Suffix-Links añadidos.

El aumento de la eficiencia en las posteriores búsquedas utilizando Suffix-Links se debe a que las comparaciones ya realizadas en la anterior *query* que coinciden con la nueva query no son vueltas a procesar, por lo que cuanto mayor prefijo tengan en común también mayor es el ahorro de comparaciones. [6]

En la figura 5.4 se muestra un proceso de *string matching* utilizando las propiedades de los *Suffix-Links*. Partiendo del proceso realizado en la figura 5.2, donde se había realizado la búsqueda de la query T_0=*acatata$* en S=*acaaaacatat$,* ahora se realiza la búsqueda de T_1 *catata$*. Si eliminamos el primer carácter a T_0 entonces comparte con T_1 el prefijo *ca.*

Los pasos ①②③ son los ya vistos en la figura 5.2, para realizar la búsqueda de la *query* T_0. En este punto ④ el último nodo intermedio visitado contiene el *Suffix-Link* (flecha verde discontinua) que nos apunta al siguiente nodo que debemos evaluar, eliminando la comparación del prefijo *ca.* Finalmente en ⑤ evaluamos el resto del camino del mismo modo que en los puntos ①②③. El nodo hijo indica que la coincidencia está en la posición 4 de la secuencia S (flecha naranja discontinua).

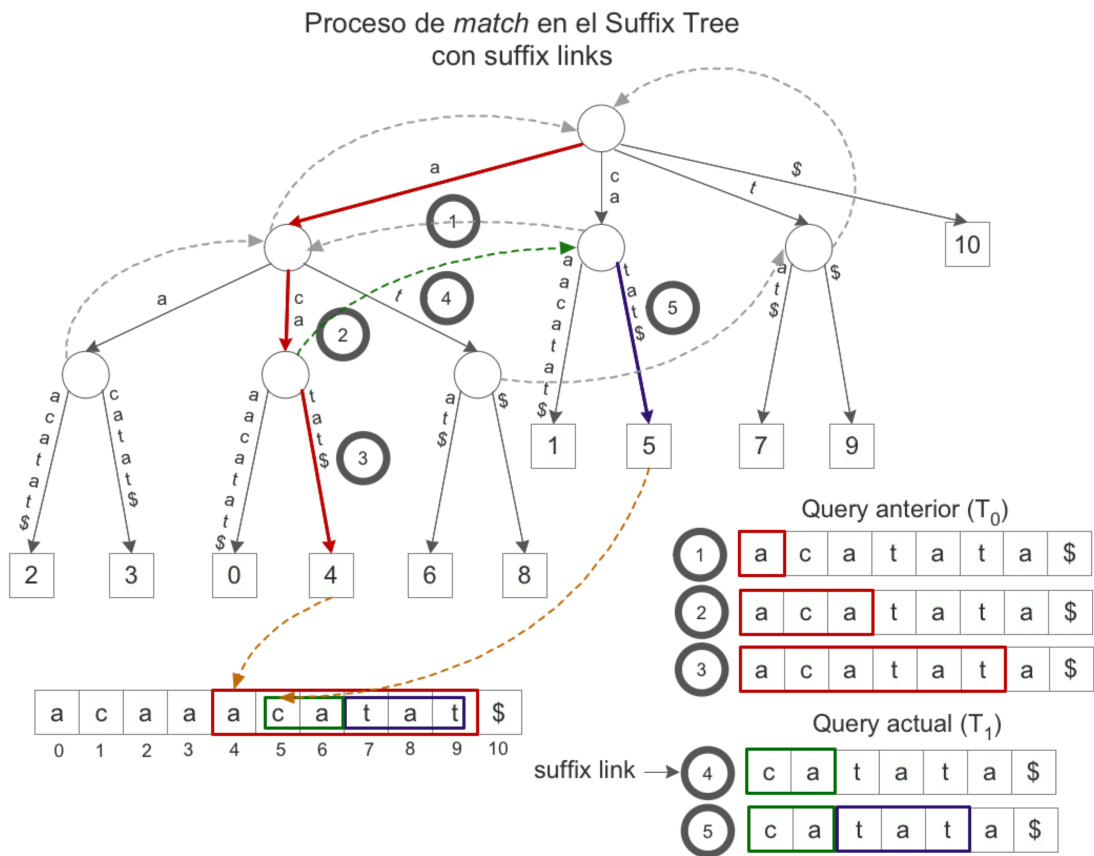


Figura 5.4: Uso de los Suffix-Links en el proceso de búsqueda de la query T_1=catata$ en el *Suffix-Tree* de la *referencia* S=acaaaacatat$ después de realizar la búsqueda de T_0=acatata$

5.3.4 Complejidad temporal y espacial

Los *Suffix-Trees* permiten realizar una búsqueda con un tiempo lineal, obteniendo una complejidad más que razonable que solo depende del tamaño de la *query* T. Por lo tanto la búsqueda tiene una complejidad de $\varnothing(m_i)$, donde m_i es la longitud de la query (T_i). [6]

Haciendo uso de Suffix-Links eliminamos las comparaciones ya realizadas, por lo que al buscar una segunda query T_{i+1} la complejidad total se reduce a $\varnothing(m_i+m_{i+1}-c)$, donde c es el número de caracteres compartidos entre T_i y T_{i+1}.

Al hacer uso del método *streaming string matching,* el requisito en memoria y la complejidad espacial queda reducida aproximadamente a la mitad, ya que sólo hemos de almacenar un Suffix-Tree.

El tipo de construcción de la estructura del árbol implementada es un factor importante en el tamaño final a ocupar. A la práctica, con secuencias genómicas, un *Suffix-Tree* con *Suffix-Links* tiene una complejidad espacial entre 22,4*n* Bytes y 32,7*n* Bytes, siendo *n* el tamaño de la referencia S y utilizando índices de 4Bytes.

La construcción del árbol *Suffix-Tree* puede ser realizada en una complejidad temporal de $\varnothing$(n) [6] siendo *n* el tamaño de la referencia S. En la práctica puede llegar a considerarse despreciable si el conjunto de secuencias a comparar es suficientemente grande. Por otra parte, los *Suffix-Links* son generados a raíz de la construcción del árbol, por lo que no suponen un tiempo extra de preprocesado para generarlos, aunque sí que consumen un espacio adicional al ser almacenados.

5.4 Suffix-Array

Al igual que el *Suffix-Tree*, el *Suffix-Array* es una estructura de indexación de datos considerada *classical index*, que permite operaciones de matching similares a las del Suffix-Tree. Esta estructura surge con el objetivo de reducir la complejidad en memoria que requiere el *Suffix-Tree*. [4]

Los *Suffix-Array* se basan en una estructura de tipo array que almacena una permutación de los índices de todos los sufijos de la secuencia de *referencia* S. La permutación de los sufijos no es una cualquiera: es el resultado de la ordenación lexicográfica del conjunto de sufijos.

En la figura 5.5 se muestra el proceso de generación de un *Suffix-Array*. A partir de la secuencia *referencia* S se generan todos sus sufijos identificados por un índice. Una vez ordenados los sufijos el índice resultante es considerado el *Suffix-Array*.

El array resultante contiene n elementos, siendo n el número de sufijos de la secuencia *referencia* S. Además del Suffix-Array es necesaria la *referencia* S para acceder a sus sufijos. [7]

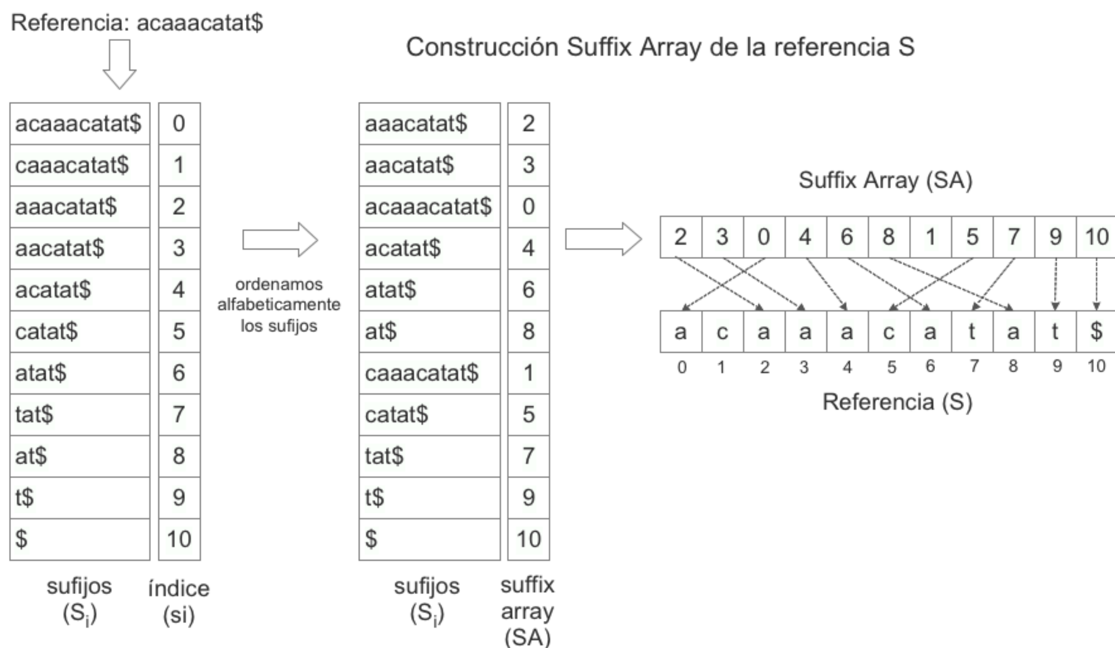


Figura 5.5: Generación de la estructura del *Suffix-Array* a partir de la secuencia de *referencia* S=acaaacatat$.

Un *Suffix-Array* puede ser representado como una simplificación del *Suffix-Tree.* Únicamente hay que coger los nodos hijo del *Suffix-Tree*, considerando que tienen una ordenación lexicográfica, y elimina las estructuras superiores a los nodos hijos. Esto conforma un array de índices ordenados lexicográficamente por los sufijos, tal y como demanda el *Suffix-Array*. La figura 5.6 muestra esta correlación entre el *Suffix-Tree* y el *Suffix-Array*.

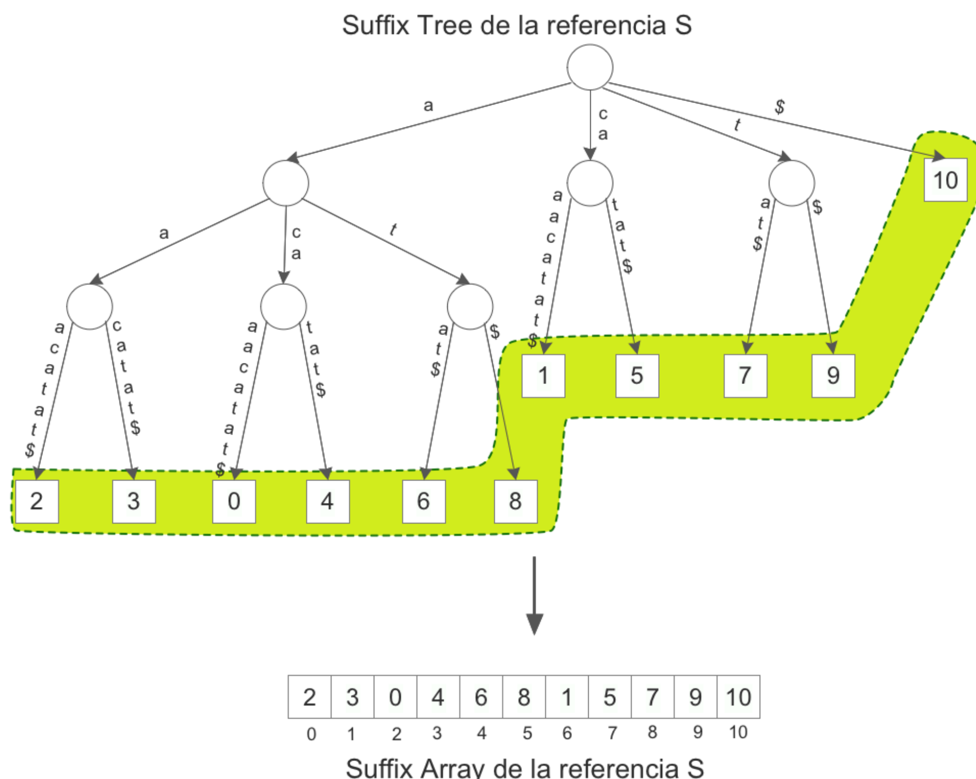


Figura 5.6: La imagen muestra la correlación entre los nodos hijos del *Suffix-Tree* y los elementos del array del *Suffix-Array*

5.4.1 String matching con Suffix-Array

Suffix-Array permite la operación de *string matching*, realizando una búsqueda binaria dentro del *Suffix-Array*.

Para realizar búsqueda binaria se requieren 3 índices a elementos del array. Esos índices acotan el rango de búsqueda dentro del array. Una vez los índices se encuentran se obtiene la coincidencia máxima de la *query* T_i en la *referencia* S.

El índice R indica el último sufijo por la derecha con correspondencia en la búsqueda, mientras que L indica el primer sufijo que tiene correspondencia en la búsqueda. Todos los sufijos comprendidos entre estos índices L y R son posibles candidatos a la coincidencia máxima que se busca, en un *Suffix-Tree* quedan representados como un subárbol. El índice M representa el próximo sufijo a evaluar y la comparación de este índice acota el espacio a la mitad en cada iteración. La figura 5.7 muestra el pseudocódigo de este proceso.

```
00  Var L = 0, R = n-1
01  (r0, ml) = COMP (S[SA[L]], T[0])
02  (r1, ml) = COMP (S[SA[R]], T[0])
03  if r0 <= 0 then
04     si = 0
05  else if r1 >= 0 then
06          si = n-1
07  else
08  {
09       while R-L > 1 do
10       {
11            M = (L + R)/2
12            (r, ml) = COMP (S[SA[M]], T[0])
13            if r <= 0 then
14                R = M
15            else
16                L = M
17       }
18  si = R
19  }
20  guardarResultado(SA[si], T, ml)
```
Figura 5.7: Pseudocodigo del proceso de *string matching* utilizando una búsqueda binaria del Suffix-Array. **si** es el índice actual del *Suffix-Array*. **COMP** realiza la comparación de 2 secuencias (**S** y **T**) retornando **r** y **ml**. **r** es un entero con signo indicando si es mayor o menor. **ml** es el número de caracteres que comparten las secuencias. **GuardarResultado** almacena la posición de la secuencia **S**, la posición de **T** y el número de caracteres que comparten (**ml**).

En la figura 5.8 se muestra un ejemplo de este proceso de *string matching* entre la secuencia *referencia* S (acaaacatat$) y la *query* T_0 (acatata$). Se realiza el proceso de búsqueda de T_0 en el *Suffix-Array* previamente generado a partir de los sufijos de S.

El número máximo de comparaciones entre secuencias máximas que tiene el algoritmo, al ser una búsqueda binaria, es de ∅($\log_2$ n). Al ser la *referencia* de 11 elementos, el resultado son las 4 comparaciones que se mostrarán ahora.

En ① se realizan 2 comparaciones de secuencias entre T_0 y las secuencias de los índices L y R. L toma el extremo izquierdo y R el derecho del array. Al comparar L_0=aaacatat$ con T_0=acatata$, T_0 es mayor que L. Al comparar R $ con T_0=acatata$, es T_0 es menor que R. Es por ello que la secuencia se encuentra dentro del intervalo del *Suffix-Array*. Finalmente, M toma el índice (L+R)/ 2, en este caso 5 que corresponde al sufijo at$.

En ② se comparan las secuencias T_0=acatata$ y M=at$, al ser T_0 menor que M, ahora R apuntará a M, posición 5. Finalmente M toma el índice (L+R)/ 2, en este caso 2 que corresponde al sufijo acatat$.

En ③ se comparan las secuencias T_0=acatata$ y M=acatat$, al ser T_0 mayor que M, ahora L apuntará a M, posición 2. Finalmente M toma el índice (L+R)/ 2, en este caso 3 que corresponde al sufijo acatat$.

Finalmente en ④ se comparan las secuencias T_0=acatata$ y M=acatat$, al ser T_0 menor que M, ahora R apuntará a M, posición 3. De este modo L y R están contiguas indicando que R es el sufijo con mayor coincidencia, acatat$ en la posición 4 de la secuencia de referencia S.

Así pues, del mismo modo que en el ejemplo Suffix-Tree, la coincidencia resultante de la búsqueda es en S, acaaacatat$, y en T_0, acatata$. El resultado puede ser anotado como 3 enteros: la posición de inicio de la secuencia S, la de inicio de T_0 y el número de caracteres que comparten, en este ejemplo 4 – 0 – 6.

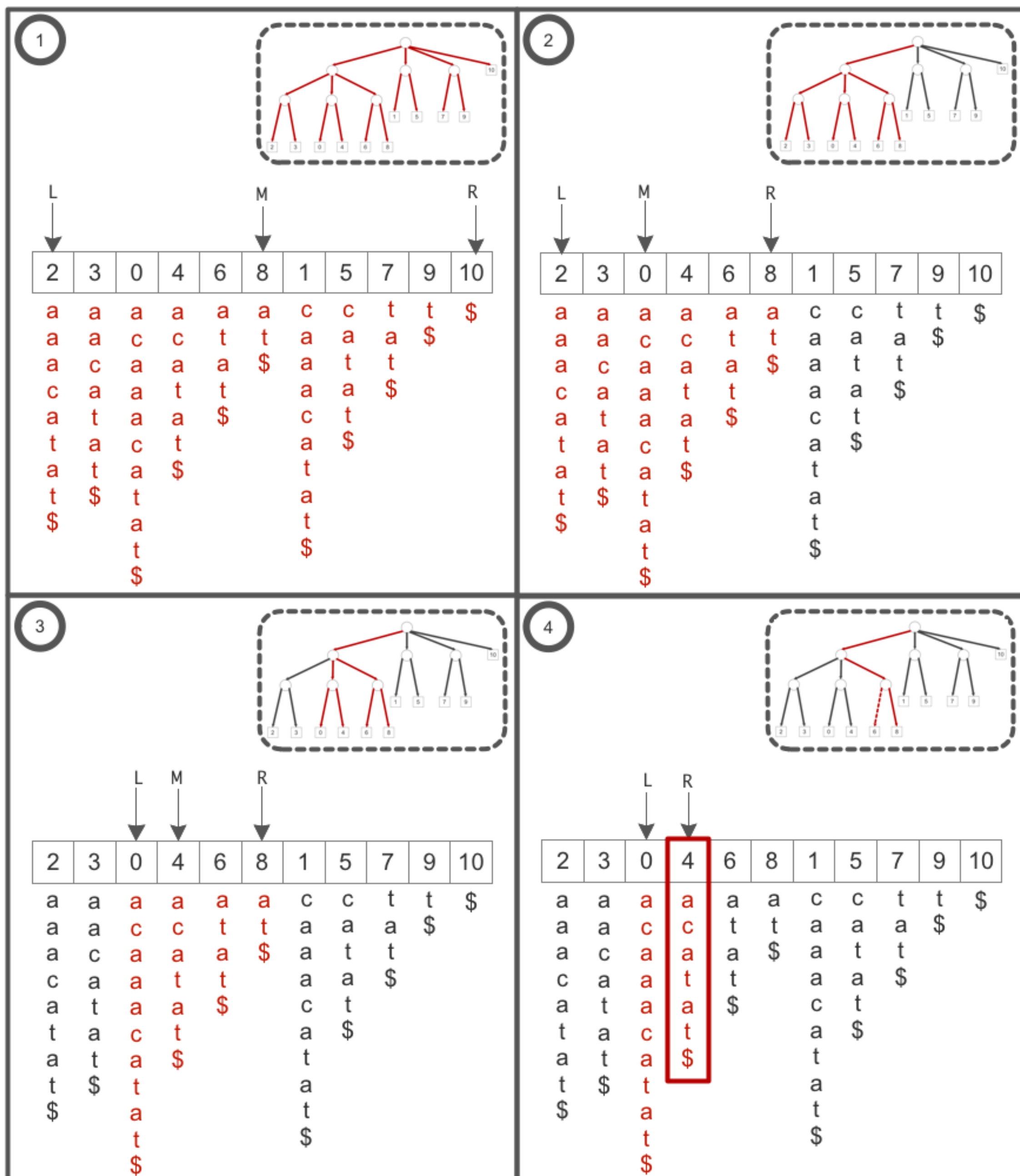


Figura 5.8: Muestra el proceso de *string matching* utilizando una estructura de indexación Suffix-Array mediante una búsqueda binaria de los sufijos. La secuencia referencia S es acaaaacatat$ y la *query* T_0 a buscar acatata$

5.4.2 String matching con Suffix-Array utilizando el LCP

Una implementación de las búsquedas binarias en los *Suffix-Arrays* propuesta por *Udi Manber* permite aprovechar de que se realizan comparaciones entre sufijos relacionados para minimizar significativamente el tiempo de búsqueda. De este modo el algoritmo aprovecha las anteriores comparaciones de la búsqueda binaria en el *Suffix-Array,* se pueden eludir comparaciones carácter a carácter en las búsquedas siguientes. [4]

Estas comparaciones corresponden a los primeros caracteres que comparten la *query* T_0 con las secuencias de los índices L y R, usualmente denominados RLCP y LLCP.

La figura 5.9 presenta el pseudocódigo de esta mejora en las búsquedas.

```
00  Var L = 0, R = n-1, skip = 0
01  (r0, ml0) = COMP (T[0], S[SA[L]])
02  (r1, ml1) = COMP (T[0], S[SA[R]])
03  Rlcp = ml0
04  Llcp = ml1
05
06  if r0 <= 0 then
07     si = 0
08  else if r1 <= 0 then
09          si = n-1
10  else
11  {
12       while R-L > 1 do
13       {
14            skip = min(Rlcp, Llcp)
15            M = (L + R)/2
16            (r, ml) = COMP (T[skip], S[SA[M]+skip])
17            if r <= 0 then
18                R = M
19                Llcp = skip + ml
20            else
21                L = M
22                Rlcp = skip + ml
23       }
24  si = R
25  }
26  guardarResultado(SA[si], T, ml)
```

Figura 5.9: Pseudocódigo del proceso de *string matching* utilizando una búsqueda binaria en el Suffix-Array aprovechando las propiedades del *LCP internal tree*. **si** es el índice actual del *Suffix-Array.* **COMP** realiza la comparación de 2 secuencias retornando **r** y **ml**. **r** es un entero con signo indicando si es mayor o menor y **ml** el número de caracteres que comparten las secuencias. **Rlcp** indica los caracteres que comparte R con M, **Llcp** los que comparte L con M. **Skip** marcará cuántos caracteres puede saltarse la actual comparación en M. **GuardarResultado** almacena la posición de la secuencia **S**, la posición de **T** y el número de caracteres que comparten (**ml**).

Aplicando este algoritmo mejorado de *string matching* en el ejemplo de la figura 6.8 se observa que: en (1) Llcp es 1 y Rlcp es 0, por lo que skip toma el mínimo entre los dos que es 0 y no ahorra ninguna comparación de caracteres entre M y T_0. En (2) Llcp es 1 y Rlcp es 1, skip toma el valor 1, por tanto ahorramos la comparación del primer carácter, *a*, entre T_0 y M. En (3) Llcp toma valor 2 y Rlcp 1, al ser el mínimo entre los dos 1, evitamos realizar la comparación del primer carácter de nuevo. En (4) la búsqueda encuentra la coincidencia máxima. En este ejemplo se ha evitado realizar 2 comparaciones menos que con la implementación usual de *string matching* Suffix-Array. En situaciones donde las secuencias tienen un tamaño considerable y con prefijos muy similares se aprecian considerables mejoras.

5.4.3 Complejidades del Suffix-Array y su construcción

Los *Suffix-Array* permiten reducir la complejidad espacial que requieren los *Suffix-Trees*. La complejidad espacial requerida por un *Suffix-Array* es 4n Bytes, siendo n el número de sufijos de la *referencia* S y utilizando enteros de 4Bytes por índice.

La búsqueda se realiza en un tiempo lineal, con una complejidad temporal de $\emptyset(m \cdot \log_2 n)$; donde $\emptyset(\log_2 n)$ es el número de comparaciones de secuencias que corresponden al proceso de acotar el espacio mediante la búsqueda binaria y $\emptyset(m)$ corresponde la cantidad de comparaciones carácter a carácter que se realizan (concretamente la longitud de la *query* T).

Esta complejidad temporal puede ser reducida con la mejora del algoritmo de búsqueda *Suffix-Array*, vista en el capítulo 5.4.2, que aprovecha la relación entre los sufijos (LCP) para minimizar el tiempo de búsqueda aprovechando comparaciones pasadas. La complejidad se reduce a $\emptyset(m+\log_2 n)$.

El *Suffix-Array* puede ser construido en un tiempo lineal $\emptyset(n \cdot \log_2 n)$. Al igual que en el *Suffix-Tree*, este tiempo puede llegar a considerarse despreciable si el conjunto de secuencias a comparar es suficientemente grande. [7]

5.5 Enhanced Suffix-Array

La estructura de datos *ESA* (*Enhanced Suffix-Array*) es de tipo *classical index*, como el *Suffix-Tree* y *Suffix-Array*. Realmente los *ESA* son una extensión de los *Suffix-Array*, debido a que se añaden estructuras de datos complementarias para poder recuperar funcionalidades que ofrecían los Suffix-Tree. [8]

Las estructuras que conforman el *ESA* son 3: el Suffix-Array (descrito en el capítulo 5.4), el *LCP-Array* y el *Rank Array*.

El *LCP-Array* es una estructura basada en un array de enteros, usualmente de tamaño 4Bytes. Estos enteros indican el LCP que comparten el sufijo actual S_i y el sufijo anterior S_{i-1} del *Suffix-Array.* Esto es el LCP(S[SA[i]], S[SA[i-1]]) descrito previamente en el capítulo 4.3.1. El *LCP-Array* nos permite simular la funcionalidad ofrecida por la información del *LCP-interval tree,* que esta contenida en el *Suffix-Tree* de forma implícita (fig. 5.10).

En la figura 5.10, se presenta un ejemplo de una estructura ESA donde se aprecia el *LCP Array* ya calculado. En la posición 3 del *LCP-Array* vemos que contiene un 3, indicando los caracteres del prefijo máximos que comparten el sufijo actual acatat$ y el anterior acaaacatat$. Además se muestra la posición que representa en el *LCP-interval tree* que simula.

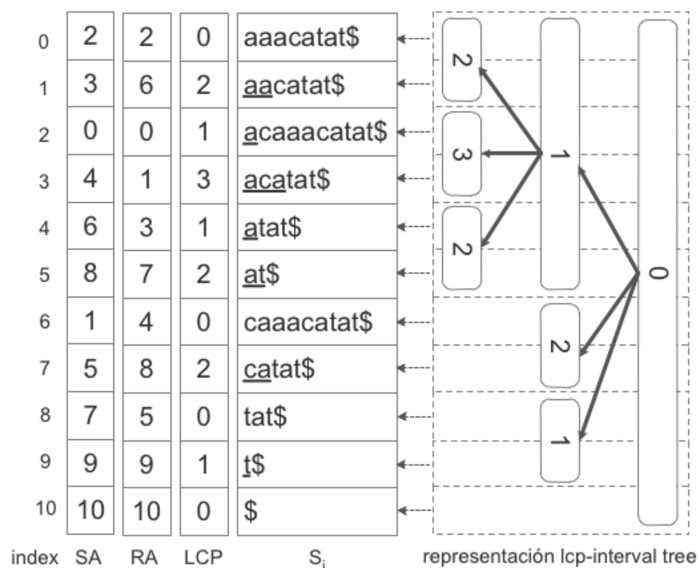


Figura 5.10: Ejemplo de *ESA* a partir de la referencia S acaaacatat$, a la derecha se representa la estructura que simula el LCP-Array

El *Rank-Array*, también llamado Suffix-Array^{-1} debido a su funcionalidad, es una estructura de tipo array de tamaño n, donde cada elemento del array es un entero de 4Bytes que contiene la inversa del *Suffix-Array*. La inversa del *Suffix-Array* cumple que (SA[k]=j) && (RA[j]=k). Una forma sencilla de construirlo es recorrer el array SA e ir asignando a RA tal que RA[SA[i]]=i, donde i es la posición actual de SA (fig. 5.11).

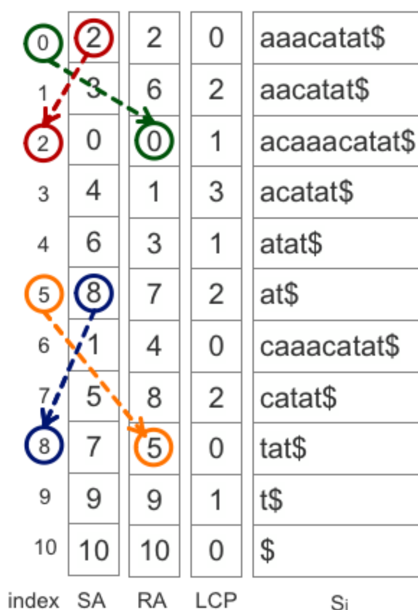


Figura 5.11: *Rank-Array* de la estructura ESA. Varios ejemplos que muestran la implementación de RA, el primero SA[0]=2, RA[0]=2 y el segundo SA[5]=8, RA[8]=5.

El *Rank-Array* permite simular la funcionalidad que proporcionan los *Suffix-Links* en el *Suffix-Tree*, por lo que aprovecha comparaciones de la anterior *query* T_{i-1} en la búsqueda de la *query* T_i actual con el propósito de reducir la complejidad temporal.

Así pues, dado un índice del sufijo S_{i-1} con contenido αw, donde α es un único carácter y *w* es una subcadena, el *Rank-Array* permitirá obtener el índice del sufijo con contenido w. Para obtener la posición del sufijo S_i, se realiza el siguiente acceso a RA, S_{i+1} = SA[RA[SA[i]+1]] siendo i el índice de la secuencia actual.

5.5.1 String matching con Enhanced Suffix-Array

Enhanced Suffix-Array permite la operación de *string matching* a fin de obtener el sufijo de máxima coincidencia, al igual que en el *Suffix-Tree* y *Suffix-Array*. Para ello realizan 3 fases:

La primera fase realiza la búsqueda binaria mejorada en el Suffix-Array, descrita en el capítulo 5.4.2. Esta fase retorna la coincidencia máxima de una primera *query* T_i en la *referencia* S.

La segunda fase comienza el procesado de la siguiente *query* T_{i+1} utilizando la estructura *Rank-Array* para simular el *Suffix-Link* y así poder saltar el procesado de diferentes comparaciones entre cadenas de T_{i+1} y S. Estas comparaciones de cadenas que evitamos corresponden a las que se realizarían en la búsqueda binaria. En lugar de realizar la búsqueda binaria de nuevo, el *Rank-Array* nos indica la posición del siguiente sufijo S_i que comparte mayor prefijo con T_i, cumpliendo lo expuesto en la figura 5.11. El cálculo de la siguiente posición se realiza con el siguiente acceso S_{i+1}=SA[RA[SA[i]+1]].

La posición dada por el *Rank-Array* ha permitido acotar el espacio de búsqueda. En la tercera fase se realiza un refinamiento de la búsqueda a partir de esta posición. Para ello se realiza una nueva búsqueda, esta vez secuencial, sobre los sufijos consecutivos a $S_{i+1.}$ Dado que el Suffix-Array contiene los sufijos ordenados lexicográficamente, todos los que comparten el mismo prefijo a S_{i+1} son consecutivos a él. Se evalúa cada uno de ellos comparándolo con T_{i+1} con el fin de encontrar una coincidencia mayor. La búsqueda termina una vez se encuentra un sufijo con menor coincidencia que la original de S_{i+1}.

La figura 5.12 muestra un ejemplo donde se realiza el proceso de *String Matching* de las *queries* T_0=acatata$ *y* T_1=cataata$ en la referencia S=acaaacatat$. A continuación se explican cada una de las 3 fases que lo conforman:

- La fase 1 corresponde a los pasos ①②③④ ya mostrados en los capítulos 5.4.1 y 5.4.2. De este modo se obtiene que la coincidencia máxima entre T_0 y S corresponde a la 4 – 0 – 6 (posición de inicio de la secuencia S, la de inicio de T_0 y el número de caracteres que comparten).

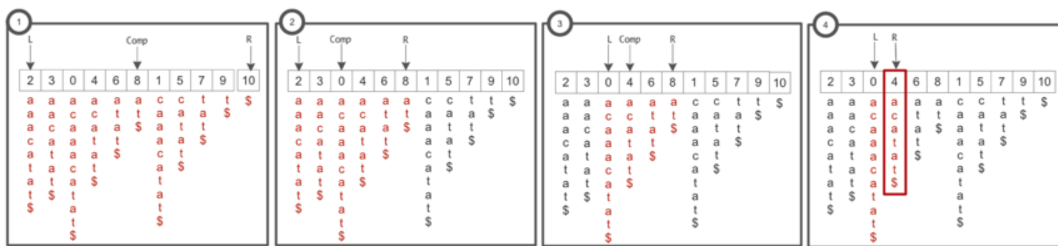
- La fase 2 acota la búsqueda de T_1 simulando un Suffix-Link con el acceso S_{i+1}=SA[RA[SA[i]+1]], que corresponde en la imagen a ⑤. SA[3] es 4 (el 4 representa la posición de inicio de secuencia S del anterior resultado T_0). RA[4+1] corresponde a 7

(se utiliza la inversa del SuffixArray para buscar la posición en SA del siguiente sufijo a S_i). El índice 7 de SA nos indica la posición del sufijo S_{i+1} que se buscaba, SA[7] = 5. Por lo tanto, con este acceso saltamos del sufijo S_i acatat$ al S_{i+1} catat$.

- La fase 3 refina la coincidencia encontrada en la fase 2. Para ello se comparan los sufijos adyacentes buscando el que contenga mayor coincidencia con T_1. En este caso como T_1 es menor que S_{i+1} se realiza la llamada a *ScanIzquierda*, que analiza los sufijos que se encuentran a su izquierda. Este recorrido se realiza hasta que el valor del LCP-Array del actual elemento sea menor que el número de caracteres que comparten la secuencia S_i actual con T_1. En el ejemplo LCP[7] es menor que el número de caracteres que comparten T_1=catata$ y S_{i+1}=catat$ (indicado en verde como s), por lo que el sufijo S_{i+1} actual es la coincidencia máxima.

El resultado del *String Matching* en este ejemplo es igual que en los anteriores: 4 – 0 – 6 (posición de inicio de la secuencia S, la de inicio de T_0 y el número de caracteres que comparten).

FASE 1: Búsqueda binaria en el Suffix Array



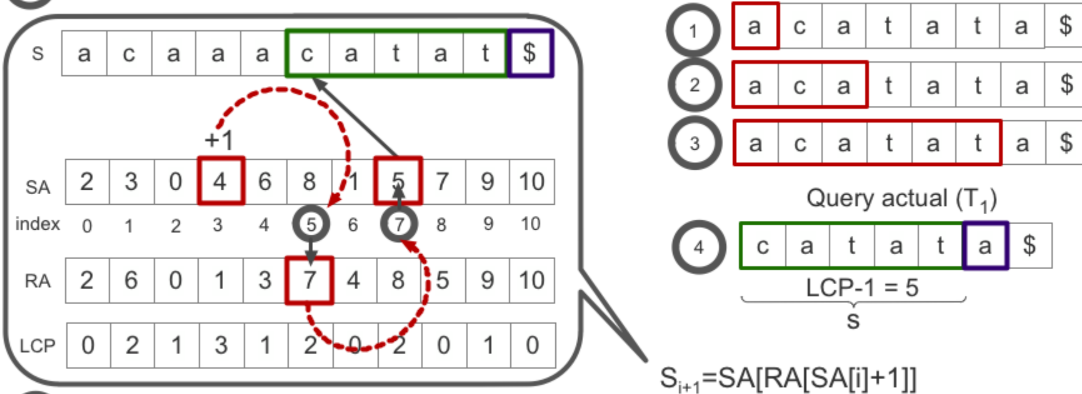


Figura 5.12: Un ejemplo de *String Matching* en *ESA* mostrando las 3 fases que realiza para encontrar la coincidencia máxima de las *queries* T_0=acatata$ *y* T_1=cataata$ en la *referencia* S acaaacatat$.

Las figuras 5.13 y 5.14 presentan el pseudocódigo del proceso String Matching mostrado en el ejemplo anterior, figura 5.12.

```
01  /*Fase1:Búsqueda binaria de la 1ª query T0*/
02  Var i = 0
03  (si, ml) = busquedaBinaria(T[i])
04  guardarResultado(si, T[i], ml)
05  i = i + 1  /*avanzamos al siguiente sufijo de T*/
06
07  while i ≤ Tlen-minmatch do
08  {
09    /*Fase 2: suffix-link simulado*/
10    s = ml – 1
12    si = Rank[SA[si]-1]
13    j = SA[si] + s
14    (r, ml) = COMP (T[i+s], S[j])
15
16    /*Fase 3: busca el más extenso*/
17    if r > 0 then
18       (si, ml) = ScanIzquierda(s+ml, si, T[i])
19    else
20       (si, ml) = ScanDerecha(s+ml, si, T[i])
21
22    guardarResultado(si, T[i], ml)
23    i = i + 1
24  }
```

Figura 5.13: Pseudocódigo del proceso de *string matching* utilizando Enhanced *Suffix-Arrays*. **si** es el índice actual del *Suffix-Array***.** **COMP** realiza la comparación de 2 secuencias retornando **r** y **ml**, **r** es un entero con signo indicando si es mayor o menor y **ml** el número de caracteres que comparten las secuencias. **s+ml** indica los caracteres que comparte en ese momento T y el sufijo de S. **Tlen** corresponde a la longitud del sufijo. **minmatch** es la longitud mínima que deben tener las coincidencias encontradas. **GuardarResultado** almacena la posición de la secuencia **S**, la posición de **T** y el número de caracteres que comparten (**ml**). **si** es el índice actual del *Suffix-Array***.**

```
00  procedure ScanIzquierda(s, si, T) {
01    r = 1
02    while LCP[si] ≥ s and r > 0 and si > 0 do
03    {
04      si = si - 1
05      j = SA[si] + s
06      (r, ml) = COMP (T[i+s], S[j])
07      s = s + ml
08    }
09    if r < 0 then si = si + 1
10    return (si, s)
11  }
```

Figura 5.14: Pseudocódigo de una parte del proceso de *string matching* en *ESA* utilizado en la fase 3 del proceso. Este pseudocódigo se presenta para el procedimiento **ScanIzquierda** pero **ScanDerecha** es muy similar, sólo que recorriendo en sentido opuesto el *LCP-Array*. **COMP** realiza la comparación de 2 secuencias retornando **r** y **ml. r** es un entero con signo indicando si es mayor o menor y **ml** el número de caracteres que comparten las secuencias. **si** es el índice actual del *Suffix-Array***.**

5.5.2 Complejidades del Enhanced Suffix-Array.

La complejidad espacial para cada estructura del *ESA* es 4n Bytes, utilizando índices de 4Bytes. Al tener 3 estructuras de datos: *Suffix-Array*, *LCP-Array* y *Rank-Array*, el espacio requerido es de 12n Bytes siendo n el tamaño de la *referencia* S. [8]

La complejidad temporal en este caso es menor que en el del *Suffix-Array*, $\varnothing(m+\log_2 n)$, para las sucesivas *queries* de T debido a que se reaprovechan comparaciones utilizando Suffix-Links

simulados. En el mejor de los casos la comparación siguiente toma ∅(1) y en al peor de los casos (aunque es poco probable) ∅(n).

5.5 Indexación de textos en MUMmer: Búsqueda de nonUnique MEMs

Las estructuras de indexación de datos mostradas previamente en los capítulos 5.1, 5.2 y 5.3 son ampliamente utilizadas en aplicaciones bioinformáticas, tal y como se comenta en el capítulo 2. Concretamente, las aplicaciones MUMmer3, MUMmerGPU2 y MUMmerGPUpp, detalladas en el capítulo 5, hacen uso de eso de estas estructuras. [9]

- MUMmer3 y MUMmerGPU utilizan *Suffix-Trees* junto a *Suffix-Links*.
- MUMmerGPUpp utiliza *Enhanced Suffix-Arrays.*

Todas estas aplicaciones, como se comenta en el capítulo 4, tienen una etapa de búsqueda de regiones exactas con 3 fases: Preprocesado de la *referencia,* String Matching y Postprocesado.

- El preprocesado de la referencia es la generación del árbol o índice de la referencia.
- El String Matching es la búsqueda de cadenas máximas exactas comunes visto en los capítulos 5.3.1, 5.4.1, 5.5.1.
- El Postprocesado es la búsqueda de los nonUnique MEMs, que son todas las posibles coincidencias con S de una query T.

Así pues en este último paso de postprocesado se precisa encontrar todas las regiones de la secuencia de referencia S que comparten un mínimo de caracteres con las queries T_i. Los algoritmos anteriores de *String Matching* vistos solo proporcionan un match máximo, llamado *MEM candidato*, por cada búsqueda de *query* T_i en S. A partir de este *MEM candidato* se pueden encontrar los *nonUnique MEMs* aplicando los siguientes algoritmos de postprocesado.

Figura 5.15: Ejemplo en el post procesado, realizando una búsqueda Bottom-Up de MEMs en el Suffix-Tree.

La búsqueda de los *nonUnique MEMs* se realiza tanto en la estructura *Suffix-Tree* como *Enhanced Suffix-Array*.

En la imagen 5.15 se muestra un ejemplo del proceso de postprocesado a realizar en un *Suffix-Tree* con la *referencia* S acaaaacatat$. A partir del *MEM candidato* catat$ se realiza una búsqueda bottom-up en el árbol hasta que las aristas que comparten tienen una longitud igual al match mínimo, en este caso 2. Todos los nodos que se encuentran por debajo de esta posición son sufijos de S que representan todos los MEMs encontrados. [6]

Un ejemplo de la operación de postprocesado en estructuras ESA se puede observar en la imagen 5.16: con una *referencia* S acaaaacatat$ y a partir del *MEM candidato* catat$ (igual que en el ejemplo anterior) realizamos dos búsquedas secuenciales en el LCP-Array, una en cada sentido del array, para encontrar todos los sufijos que tengan un LCP mayor a la longitud mínima del match que nos indican por parámetro.

El pseudocódigo de esta operación de postprocesado puede observarse en la figura 6.17.

Figura 6.16: Postprocesado en el proceso de búsqueda de coincidencias en estructuras ESA. Proceso de búsqueda de MEMs

```
00  Procedure BuscarMEMs(i, si, ml){
01    /* Mostramos el MEM máximo previamente localizado */
02    Print(SA[si], i, ml)
03
04    /* Evaluamos hacia la izquierda buscando MEMs */
05    v = si
06    m = ml
07    while v > 0 and m ≥ minmatch do
08    {
09       m = min(m, LCP[v])
10       v = v - 1
11       Print(SA[v], i, m)
12    }
13
14    /* Evaluamos hacia la derecha buscando MEMs */
15    v = si + 1
16    m = min(SA[v], LCP[v])
17    while v < reflen and m ≥ minmatch do
18    {
19       Print(SA[si], i, m)
20       v = v + 1
21       m = min(m, LCP[v])
22    }
23  }
```

Figura 6.17: Pseudocódigo del proceso de búsqueda de MEMs en las estructuras de datos ESA. El código realiza un postprocesado de las coincidencias únicas que encuentra el *String Matching*, capítulo 5.5.1. **Print** muestra la posición de la secuencia **S**, la posición de **T** y el número de caracteres que comparten (**ml** o **m**). **si** y **v** es el índice actual del *Suffix-Array***.** **minmatch** es la longitud mínima que deben tener las coincidencias encontradas. **reflen** indica la longitud de la secuencia de referencia S.

6.6 Puntos clave del capítulo

Existen multitud de estructuras de indexación: Suffix-Trees, Suffix-Arrays, Enhanced Suffix-Arrays, Compressed Suffix-Arrays, Burrows-Wheeler Transformation, Wavelet Trees, Lempel-Ziv Indexes… Estas estructuras de indexación muestran entre ellas numerosos cambios en la implementación con el propósito de mejorar el índice de $\frac{espacio\ de\ estructuras}{tiempo\ de\ búsqueda}$ en escenarios específicos.

De entre todos ellos el proyecto se centra a estudiar las estructuras de indexación Suffix-Tree, Suffix-Array y Enhanced Suffix Array por los siguientes 3 motivos:

- Son las 3 más utilizadas y que entrañan de una menor base de conocimientos previos en el ámbito de la indexación de textos.
- Son las utilizadas en las 3 implementaciones analizadas de MUMmer.
- Tienen una complejidad en el tiempo de aprendizaje adecuada para la duración de un proyecto final de carrera.

Los conceptos más destacables del capítulo son los siguientes:

- Como se ha comentado anteriormente, la parte más representativa del WGA en cantidad de tiempo y cómputo es la fase 1: preprocesamiento y búsqueda de regiones exactas.

- El proceso de búsqueda de regiones exactas se realiza con algoritmos de indexación de textos, una búsqueda de forma exhaustiva, de forma Naïve approach, es inasumible por tiempo.
- La etapa de búsqueda de regiones exactas son 3 fases: Preprocesado de la *referencia,* String Matching y Postprocesado.
    - El *Preprocesado de la referencia* es la generación del árbol o índice de la referencia.
    - El *String Matching* es la búsqueda de cadenas máximas exactas comunes.
    - El *Postprocesado* es la búsqueda de los nonUnique MEMs, que son todas las posibles coincidencias en la *referencia* de una *query*.
- La estructura del *Suffix-Tree* permite una búsqueda con una complejidad temporal de $\varnothing$(m) por cada query. Los *Suffix-Links* permiten reducir la complejidad temporal reutilizando las comparaciones realizadas en las anteriores *queries*. La complejidad espacial a la práctica es entre 22,4*n* Bytes y 32,7*n* Bytes para secuencias genómicas, siendo *n* el tamaño de la referencia S. [10]
- La estructura *Suffix-Array* permite reducir la complejidad espacial de los *Suffix-Trees* siendo de 4n Bytes. Una búsqueda tiene una complejidad de $\varnothing(m \cdot \log_2 n)$ y en su versión mejorada utilizando el LCP de $\varnothing(m+\log_2 n)$.
- La estructura *Enhanced Suffix-Array* permite recuperar funcionalidades perdidas en el *Suffix-Array* añadiendo 2 estructuras complementarias, el *LCP-Array* y el *Rank-Array*. Estas estructuras, permiten reducir la complejidad temporal de los *Suffix-Array* simulando los *Suffix-Links* con una complejidad espacial de 12n Bytes.
- Cada uno de los algoritmos a analizar apuesta por una estructura distinta:
    - *MUMmer3* y *MUMmerGPU* utilizan *Suffix-Trees* junto a *Suffix-Links*.
    - *MUMmerGPUpp* utiliza *Enhanced Suffix-Arrays.*

CAPÍTULO 6 – EVALUACIÓN EXPERIMENTAL DE LAS APLICACIONES

6.1 Resumen

En este capítulo se realiza una evaluación de forma experimental de las 3 implementaciones de MUMmer que se detallan en el capítulo 4. Se realiza un análisis de las implementaciones MUMmer3.22, MUMmerGPU2.0 y MUMmerGPU++0.1.

Para poder realizar el análisis se deben tomar mediciones de las características de las aplicaciones, como puede ser el tiempo de ejecución, el consumo de espacio en memoria, el tiempo de lectura a disco, el consumo de ancho de banda a memoria, las transacciones de memoria realizadas, etc

Estas características tienen que reflejar el comportamiento de las aplicaciones con el fin de detectar cuellos de botella y comportamientos inesperados.

Para ello, en este capítulo se muestra la metodología utilizada en estas mediciones y en la adquisición de datos, además de reflejar los datos necesarios para poder reproducir las experimentaciones a posteriori. Finalmente se realiza un análisis de las aplicaciones con los datos obtenidos de las mediciones.

6.2 Metodología utilizada en la adquisición de datos

En la fase del proyecto de obtención de datos y experimentación con las aplicaciones se toma especial atención en la metodología utilizada a fin de poder validar una correcta adquisición de datos en las ejecuciones.

Un primer paso en esta metodología ha sido verificar que las ejecuciones de cada una de las implementaciones generaban el mismo resultado. El objetivo de esta verificación es poder asegurar que se analiza la misma funcionalidad en cada implementación. El método utilizado de validación de resultados entre las diferentes aplicaciones con un mismo *workload* ha sido el siguiente:

1. Estudio de los parámetros de entrada de cada uno de los ejecutables. La funcionalidad de los parámetros de entrada se puede obtener con la ayuda de las implementaciones.
2. Dado que algunos parámetros no se encuentran documentados, se realiza una serie de ejecuciones de las 3 implementaciones con el fin de observar cuál de ellos es clave en la generación de los resultados.
3. Una vez realizadas las pruebas empíricas con diferentes entradas de parámetros, se compara el resultado generado de cada uno de ellos. Para ello se ha hecho uso de la herramienta diff.
4. Se seleccionan los parámetros de las ejecuciones en las que el resultado tiene coincidencia y se vuelve a testear para un mayor número de conjuntos de datos.

Un segundo paso realizado en la metodología fue verificar los parámetros de optimización en la etapa de compilación (capítulo 6.4). Se comprueba que se utilizan los mismos parámetros para

cada uno de los códigos fuente, tanto CPU como GPU. De no ser así, se ponen en común los que utilizan una optimización más agresiva del código y se realizan varias ejecuciones. Finalmente se seleccionan los parámetros que nos reportan a priori un mayor rendimiento en las aplicaciones.

Un tercer paso se basa en seleccionar el conjunto de workloads de entrada más representativos para el problema de alineamiento de genomas. Es importante diseñar un workload que cubra un espectro de casos que muestre el comportamiento general de los programas. Para ello se han seguido los siguientes pasos:

1. Se escogen varios workloads de entrada de forma cuasi aleatoria. En este caso se ha partido de la experiencia adquirida en artículos sobre el mismo tema utilizando workloads de similares características.
2. Se miden tiempos totales de las ejecuciones con los anteriores workloads y se contrastan los resultados de tiempo instrumentados.
3. A los tiempos que se alejan de la tendencia y sugieren ser anomalías en el comportamiento, se les aumenta la resolución. Esto significa volver a tomar medidas de nuevas ejecuciones de workloads de similares características al que tiene un valor anómalo. Esto se realiza con el fin obtener mayor información y poder discernir si los workloads similares también sufren esta anomalía en el comportamiento. El objetivo es agrupar esta serie de anomalías similares y detectar si están siendo inducidas por algún cuello de botella.

Un cuarto paso es generar un script de lanzamiento con toda la batería de workloads que se desean ejecutar. El propósito de este paso es automatizar y facilitar la adquisición de datos. Un script automatizado permite reducir los tiempos de recogida de datos y los errores en las mediciones inducidos por el factor humano.

Un quinto paso es generar un *parser* para procesar los datos obtenidos y representarlos con diagramas o gráficos que faciliten su interpretación para localizar estos cuellos de botella en la aplicación. Es importante tratar de representar los mismos resultados con diferentes métricas o diagramas a fin de escoger el que transmita la mayor información posible.

Una vez con los datos y gráficos de las mediciones disponibles es el momento de comenzar el análisis de las aplicaciones (capítulo 6.10). Llegado a este punto se ha tenido que iterar varias veces esta metodología, debido a que los datos no aportaban demasiada información, o bien para descartar posibles análisis erróneos.

6.3 Entorno y sistema de experimentación

El sistema utilizado para realizar la experimentación ha sido un computador *comodity* de escritorio que utiliza como sistema operativo *GNU/Linux Ubuntu 11.04* x86-64 con la versión del *kernel Linux* 2.6.38-8, el driver propietario de *Nvidia* 270.41.19 y sistema de ficheros *ext4*.

	Procesador CPU	Procesador GPU
Arquitectura	Multicore Intel Nehalem	ManyCore Nvidia Fermi
Modelo	Intel Core i7 920	Nvidia GTX 480
Frecuencia reloj	2,66 Ghz	1,4 Ghz
Controlador de Memoria	3 x DDR3 1066 MHz	6 x GDDR5 3696 Mhz
Cores	4	480 (15SM x 32SP)
Threads hardware	8	23040
Repertorio de instrucciones	X86-64 y SSE4.2	PTX 2.0
Cache datos L1	32KB	16KB/48KB (configurable)
Caché datos L2	256KB	768KB
Caché datos L3	8192KB	-
Memoria ScratchPad	-	16KB/48KB (configurable)
Soporte PCI	PCI Express 2.0 16x	PCI Express 2.0 16x
Memoria	6GB	1536MB

	Disco
Modelo	MAXTOR STM3320820A 320GB
Bus de transferencia	Ultra ATA 100MB/s
Caché	8192KB
Velocidad del disco	7200rpm

6.4 Herramientas

Para realizar la obtención de datos de las aplicaciones se han utilizado diversas herramientas de *profiling* y *debugging*:

Herramienta	Versión	Breve descripción de la funcionalidad que presentan
CUDA SDK & Toolkit	v4.0.17	Runtime de CUDA y herramientas de desarrollo para GPUs.
Ioping	v0.6	Mide latencias de disco simulando acceso a ficheros.
Fsck	v2.17.2	Verificar la fragmentación de un disco.
Genreads	Script	Genera *queries* a partir de la *referencia.*
Hdparm	v9.32	Mide ancho de banda en disco.
DDD	v3.3	Debuga código para CUDA y C.
Filefrag	v1.41.14	Verifica la fragmentación de ficheros.
Diff	v3.0	Busca homología entre ficheros.
Time	v1.7	Calcula el tiempo de ejecución de una tarea.
Gettimeofday	Rutina	Calcula el tiempo de ejecución de una zona del código.
Scripts bash	Script	Automatiza procesos rutinarios de la experimentación.
Scripts phyton	Script	Automatiza procesos rutinarios de la experimentación.

6.5 Carga de trabajo utilizada

Los datos utilizados en la entrada son datos biológicos reales, que están extraídos de la base de datos *genbank*. Los datos utilizados son secuencias genómicas altamente estudiadas, muy utilizadas en diferentes análisis de aplicaciones bioinformáticas, ya que permiten verificar los resultados obtenidos de forma más fidedigna.

La generación de los *workloads* se realiza de forma automatizada. Esto se logra con un script denominado *genreads,* escrito en *python* por *Cole Trapnell* y *Michel Schatz*. Es un script ampliamente utilizado en la generación de datos de entrada para el análisis de aplicaciones

bioinformáticas: a modo de ejemplo tenemos los artículos sobre MUMmer que hacen uso de él. El Script genera *queries* a partir de la secuencia de *referencia* introduciendo mutaciones pseudoaleatorias. El algoritmo beneficia a la automatización de las pruebas, permitiendo tener *workloads* muy flexibles y a la vez garantizando que se cumple un mínimo de similitud entre secuencias.

A continuación se definen los datos que componen cada uno de los 6 conjuntos de *workloads* que se han utilizado:

En los siguientes *workloads* A, B, C y E se utiliza como secuencia de *referencia* el genoma de la bacteria intracelular *Listeria monocytogenes,* causante de la enfermedad con una alta tasa de mortalidad llamada *Listeriosis*. En los Workloads D_0 y D_1 se utiliza como *referencia* el segundo cromosoma humano.

	referencia	**Tamaño referencia**	**Tamaño queries**	**Match Mínimo**
WorkLoad A	MONO – Listeria monocytogenes	2.344.528	100	20
	Número queries: 10K - 50K - 150K - 300K - 600K - 1M - 2,5M - 5M - 7,5M - 10M - 15M - 20M - 40M - 60M			

En el conjunto de *workloads* A se varia el número de queries entre 10 mil y 60 millones, mientras los otros parámetros se mantienen fijos.

	referencia	**Tamaño referencia**	**Número de queries**	**Match Mínimo**
WorkLoad B	MONO – Listeria monocytogenes	2.344.528	1M	20
	Tamaño de las queries: 25 - 50 - 100 - 200 - 400 - 800 - 1600 - 3600			

En el conjunto de *workloads* B se varia el tamaño de las queries entre 25 y 60 bases nucleótidas, mientras los otros parámetros se mantienen fijos.

	referencia	**Tamaño referencia**	**Número de queries**	**Match Mínimo**
WorkLoad C	MONO – Listeria monocytogenes	2.344.528	1M / tamaño queries	20
	Tamaño de las queries: 25 - 50 - 100 - 200 - 400 - 800 - 1600 - 3600 - 7200 - 14400			

En el caso del conjunto de *workloads* C el parámetro que varia es el tamaño de las queries y el número de queries pero varían de modo proporcional siempre con un tamaño tal que *tamaño_queries*numero_de_queries*=1M de nucleótidos.

	referencia	**Número de queries**	**Tamaño queries**	**Match Mínimo**
WorkLoad D_0	HS1 – Cromosoma 2 del Homo sapiens	1M	100	20
	Tamaño referencia: 10K - 150K - 500K - 2M - 5M - 7M - 10M - 14M - 50M			

En el conjunto de *workloads* D_0 se va incrementando el tamaño de la propia referencia entre 20 mil y 50 millones, mientras los otros parámetros se mantienen fijos.

	referencia	**Número de queries**	**Tamaño queries**	**Match Mínimo**
WorkLoadD_1	HS1 – Cromosoma 2 del Homo sapiens	10M	100	20
	Tamaño referencia: 10K - 150K - 500K - 2M - 5M - 7M - 10M - 14M - 50M			

En el conjunto de *workloads* D_1 se realiza el mismo incremento de la referencia que en D_0, pero variando el número de *queries* a 10 millones.

	referencia	**Tamaño referencia**	**Número de queries**	**Tamaño queries**
WorkLoad E	MONO – Listeria monocytogenes	2.344.528	1M	100
	Match Mínimo: 10 - 15 - 20 - 30 - 40 - 50 - 60 - 70 - 80 - 90 - 100			

En el conjunto de *workloads* E se varia el tamaño mínimo del *maximal match* de las queries entre 10 (una coincidencia muy poco restrictiva) y 100 bases (una coincidencia el máximo de restrictiva).

6.6 Parámetros de ejecución

Los parámetros de ejecución tomados han sido escogidos a raíz de realizar una experimentación. En esta experimentación se realizan ejecuciones con diferentes combinaciones de parámetros de entrada con fin de obtener una salida con los mismos resultados para todas las implementaciones.

A continuación se detallan los parámetros utilizados en las ejecuciones con el fin de facilitar el proceso de verificación o replicación de los experimentos a posteriori.

	Parámetros	**Secuencias de entrada**
MUMmer	-maxmatch -l <min_match>	<fichero_referencia> <fichero_queries>
MUMmerGPU	-l <min_match>	<fichero_referencia> <fichero_queries>
MUMmerGPUpp	-l <min_match> -t 1	<fichero_referencia> <fichero_queries>

MUMmer tiene multitud de formas de realizar la búsqueda de *maximals matchs*. Con el parámetro –maxmatch se indica que se deben buscar todas las coincidencias, tanto en la referencia como en las queries. *MUMmerGPU* y *MUMmerGPUpp* sólo tienen este tipo de búsqueda, por lo que no existe un parámetro para especificarlo. El parámetro –l permite indicar el tamaño mínimo de los *maximal matchs.* El parámetro –t 1 no se encuentra documentado. Concretamente, no sabemos que funcionalidad atañe pero, mediante pruebas empíricas probando distintos valores, hemos verificado que utilizando el valor 1 en este parámetro MUMmerGPU++ genera la misma salida que *MUMmer* y *MUMmerGPU*.

6.7 Proceso de compilación

El proceso de compilación es una etapa de la experimentación de suma importancia, ya que el código generado para cada uno de los ejecutables puede variar considerablemente entre diferentes compiladores o con diferentes parámetros de compilación. Estas variaciones entre diferentes compilaciones pueden variar el comportamiento del programa y/o desembocar en una errónea interpretación en los resultados de la instrumentación.

La documentación de este proceso facilita la posterior validez del análisis y de los datos obtenidos, dado que teniendo todos estos datos los experimentos pueden ser replicables a posteriori.

Con el propósito de poder realizar una comparación fidedigna de las implementaciones, se ha utilizado la misma versión de compilador entre ellas. Las versiones de los compiladores utilizados son las siguientes:

- Compilador para C: GNU gcc v4.5.2
- Compilador de C++: GNU g++ v4.5.2

- Compilador de CUDA: Nvidia nvcc v0.2.1221 (*release* 4.0)

Todas las compilaciones se han realizado haciendo un uso compartido de los parámetros de compilación. A no ser que se exprese lo contrario, los parámetros utilizados son los siguientes:

- -03: nivel de optimización del compilador, el nivel 3 es el más alto.
- -use_fast_math: utiliza instrucciones específicas de la arquitectura para realizar operaciones aritméticas concretas de forma más eficiente pero sacrificando precisión.
- -sm_20: realiza una generación de código con optimizaciones e instrucciones específicas de la arquitectura Fermi.
- -Xptxas -dlcm=ca: habilita la caché de datos L1 en modo 48KB en la arquitectura ManyCore.
- -Xptxas -dlcm=cg: deshabilita la caché de datos L1 en la arquitectura ManyCore.

		COMPILADORES		
		gcc	g++	nvcc
C	MUMmer3.22	-O3	-O3	
Ó	MUMmerGPU2.0		-O3	-O3 –use_fast_math
D	MUMmerGPU++0.1		-O3	-O3 –use_fast_math
I	MUMmerGPU++0.1 (fermi)		-O3	-O3 –use_fast_math –arch=sm_20
G	MUMmerGPU++0.1 (fermi & con caché)		-O3	-O3 –use_fast_math –arch=sm_20 -Xptxas -dlcm=ca
O	MUMmerGPU++0.1 (fermi & sin caché)		-O3	-O3 –use_fast_math –arch=sm_20 -Xptxas -dlcm=cg

Los parámetros anteriormente citados son los que tienen un impacto directo en la generación del código. El resto de parámetros del *makefile* no tienen una implicación directa en la generación del código y son obviados. Se han utilizado los que presentaban de origen los *makefiles* de las implementaciones.

A la implementación MUMmerGPU++ se le ha modificado la definición en tiempo de compilación de un *define* para obtener resultados equivalentes entre los resultados de los algoritmos. Este *define* corresponde a FEATURE_FINGER_TABLE.

6.8 Proceso de ejecución

El proceso de ejecución de los experimentos ha sido automatizado mediante scripts en *bash*. Una vez realizados los scripts, se economiza tiempo en tareas rutinarias como el lanzamiento de las experimentaciones o la recogida de los datos. Al estar automatizados, se reduce la probabilidad de error humano en los sucesivos lanzamientos.

Este proceso de automatización de los diferentes scripts en la obtención de las mediciones se realiza de la siguiente forma:

Se ejecuta cada uno de los conjuntos de *workloads* (A,B,C,D_0,D_1,E) con las aplicaciones MUMmer, MUMmerGPU y MUMmerGPU++. Antes de cada una de las ejecuciones se realiza la generación de las queries con el script genReads y la lectura de un fichero Dummy. Finalmente se realiza un postprocesado de los datos de las mediciones obtenidas.

La generación de las queries se realiza a partir de la referencia con el script *genReads* ya descrito en el subcapítulo 6.4.

La lectura del fichero *Dummy* consiste en leer de disco un fichero con datos no útiles, en este caso de 7GB, con el fin de eliminar la localidad temporal entre los datos de los *workloads*. Forzando a leer este fichero de datos forzamos a que no exista una copia en memoria del workload entre ejecuciones, por lo que se garantiza que los workloads serán leídos de disco en cada ejecución.

El postprocesado de las mediciones realiza un formateado de texto a fin de facilitar la importación a programas de calculo como *Excel*.

A continuación se detalla el pseudocódigo de los scripts que realizan este procesado:

```
00 conjWorkLoads = [A, B, C, D0, D1, E]
01 foreach conjWorkLoad in conjWorkLoads do {
02   foreach WorkLoad in conjWorkLoad do {
03
04     (minMatch, referencia) = leerDisco(WorkLoad)
05     queries=genReads(referencia)
06     warning(testFragmentacion(referencia, queries))
07
08     leerFicheroDummy()
09     mediciones = ejecutarMUMmer(referencia, queries, minMatch)
10     leerFicheroDummy()
11     mediciones = ejecutarMUMmerGPU(referencia, queries, minMatch)
12     leerFicheroDummy()
13     mediciones = ejecutarMUMmerGPUpp(referencia, queries, minMatch)
14
15     datosFormateados = postProcesado(mediciones)
16     guardarResultados(datosFormateados)
17   }
18 }
```

Figura 7.1: Pseudocódigo de los scripts de lanzamiento de los workloads y mediciones

Finalmente, es importante destacar que en el proceso de ejecución y medición la característica de *TurboBoost* del procesador CPU *Intel Core i7 920* ha sido desactivada para que no introdujese variabilidad a los tiempos de las ejecuciones: de este modo se obtiene una frecuencia constante en la ejecución. Por otro lado, el procesador GPU *Nvidia GTX 480* está instalado de forma que queda dedicado por completo a realizar cómputo. A fin de lograrlo, se utiliza una segunda GPU dedicada a mostrar la salida del entorno grafico, con el objetivo de no introducir artefactos en los datos de las mediciones que puedan inducir a error.

6.9 Métricas utilizadas

En el proceso de medición de cada una de las ejecuciones se han tomado unas métricas muy específicas a fin de poder comparar los resultados obtenidos entre ellas.

Las métricas y su uso son los siguientes:

Kpbs/s: Métrica de rendimiento utilizada para cuantificar la cantidad de bases nucleótidas que se procesan en un instante determinado. En este caso, miles de bases por segundo. Esta métrica permite comparar rendimientos entre ejecuciones que tienen un conjunto de *queries* distinto, es decir, un tamaño de problema diferente.

GB/s: Métrica de rendimiento utilizada para evaluar el número de datos que pueden ser transferidos en un instante determinado. En este caso, miles de millones de Bytes en un segundo. Esta métrica permite comparar el rendimiento de lectura o escritura entre los diferentes conjuntos de datos de las ejecuciones.

ns/base: Métrica de latencia para medir el tiempo de ejecución de un proceso. En este caso, nanosegundos en procesar una base del conjunto de *queries*. Esta métrica se utiliza para realizar una normalización de los resultados y poder comparar tiempos de ejecuciones con distinta carga de trabajo.

msec: Es una métrica de tiempo absoluto que permite reflejar el tiempo que percibe el usuario en la ejecución del *workload*. Es ideal para comparar las ganancias que existen entre las diferentes implementaciones de la aplicación con un mismo *workload*.

6.10 Resultados de la experimentación

En este apartado se presentan de forma grafica los datos obtenidos en las mediciones realizadas con el fin de facilitar el análisis. Así pues, se realiza un análisis e interpretación a partir de los datos graficados con el fin de identificar posibles ineficiencias y cuellos de botella en la aplicación. Además permitirá caracterizar el comportamiento de la aplicación en función del tipo de carga de entrada que tenga. Y poder identificar en qué parte del algoritmo hay que focalizar los esfuerzos con tal de optimizarlo.

6.10.1 Pruebas generales

En este subcapítulo se grafica y analizan los datos obtenidos de 3 aplicaciones: *MUMmer*, *MUMmerGPU* y *MUMmerGPU++*. Se realiza una comparativa entre las 3 aplicaciones mediante métricas de tiempo y rendimiento. Para llevar acabo la experimentación se ha utilizado el conjunto de *workloads* que se indican en el capítulo 6.5, cada una de las gráficas tiene indicado el *workload* de entrada con el que se realiza la experimentación.

En este caso se ha utilizado la herramienta *time* (ver capítulo 6.4) para medir el tiempo que toma la ejecución al procesar cada uno de los *workloads.* Los resultados se muestran en tiempo absoluto de ejecución y en *Kpbs/s*, que indican el número de bases que se procesan del conjunto de *queries* de entrada en un segundo. Esto permite comparar el rendimiento entre los algoritmos aunque utilicen una carga de trabajo distinta.

La figura 6.2 muestra el tiempo absoluto que los tres programas han tardado en ejecutar el conjunto de *workloads* A. Este *workload* incrementa el tamaño de las *queries* y fija el resto de parámetros.

Esta grafica compara de forma rápida la diferencia de tiempos que tardan los algoritmos en resolver un mismo problema utilizando la misma carga de trabajo y bajo unas mismas condiciones. Así pues, la primera impresión es que *MUMmerGPU++* tiene un tiempo de ejecución menor que el resto de programas para todos los casos analizados. Además se percibe que el tiempo de ejecución crece de forma constante para cualquier tamaño de problema en las 3 aplicaciones. La siguiente gráfica permitirá ver que este hecho no es del todo cierto.

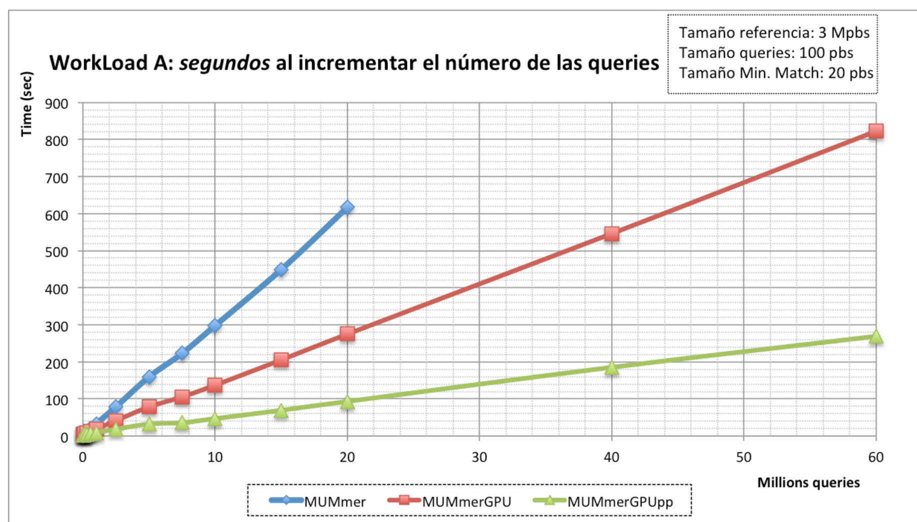


Figura 6.2: Gráfica que muestra el tiempo de procesado de *MUMmer*, *MUMmerGPU* y *MUMmerGPU++* para el *workload* A

El tipo de gráfica que se muestra en la figura 6.3 permite obtener mayor información que la anterior. Con ella se puede realizar una comparación entre el rendimiento de las tres aplicaciones en *Kpbs/s* aún disponiendo de cargas de trabajo de distinto tamaño.

En la gráfica se puede observar la mejora de rendimiento de *MUMmerGPU* de hasta 2 veces frente a la ejecución serie de *MUMmer*. *MUMmerGPU++* ofrece un rendimiento hasta 6,2 veces mejor que la implementación serie.

También se observa que el rendimiento de los programas va creciendo hasta que el conjunto de entrada es de 7,5 millones de queries, y después se estabiliza. Esto es debido a que el problema, en este rango de entrada, no es lo suficientemente grande y existe un mayor *overhead* respecto al cómputo que se realiza. Con el fin de extraer conclusiones más específicas de este comportamiento, en el subcapítulo 6.10.2 se realizan unas gráficas que nos aportan información adicional.

Así pues, podemos apreciar que aumentando el número de *queries* de entrada, el rendimiento de los programas aumenta. Incluso parece que *MUMmerGPU++* con un tamaño mayor de problema *MUMmerGPU++* podría conseguir mayores rendimientos.

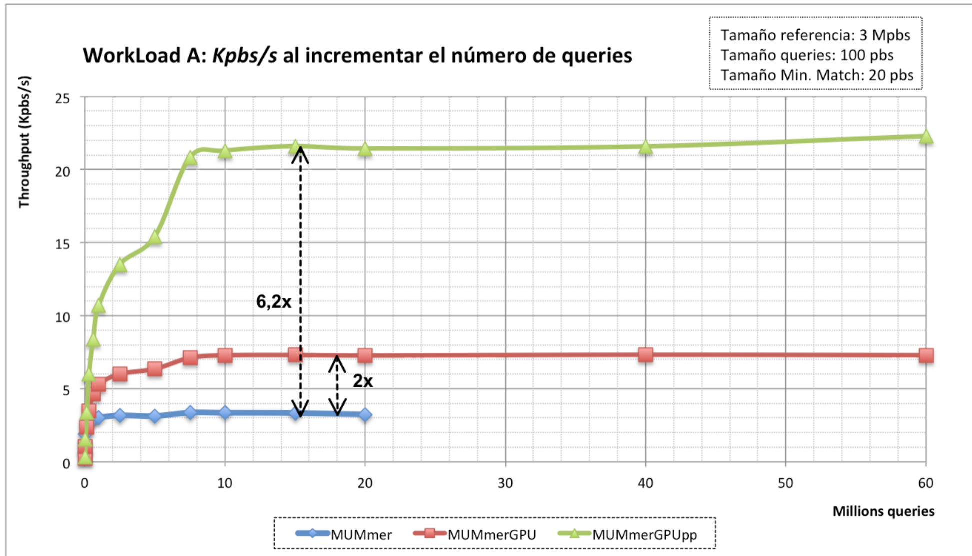


Figura 6.3: Rendimiento de *MUMmer*, *MUMmerGPU* y *MUMmerGPU++* para el *workload* A , que incrementa el tamaño de las *queries*.

En la figura 6.4 se vuelve a realizar el mismo proceso de análisis pero para el conjunto de *workloads* B. Se observa que el rendimiento de *MUMmerGPU++* es mucho mayor, obteniendo una mejora de hasta 8 veces respecto a la que obtiene la ejecución serie de *MUMmer*.

Este aumento del rendimiento frente al anterior análisis de la figura 6.3 puede ser debido a un aumento del tamaño del conjunto de trabajo (> 6Bpbs) o al hecho de trabajar con *queries* de mayor tamaño. Para poder sacar conclusiones más específicas se realizan las gráficas de la figuras 6.6 y 6.5 que nos aportan información adicional que ayuda a saber la implicación de estos factores en el rendimiento.

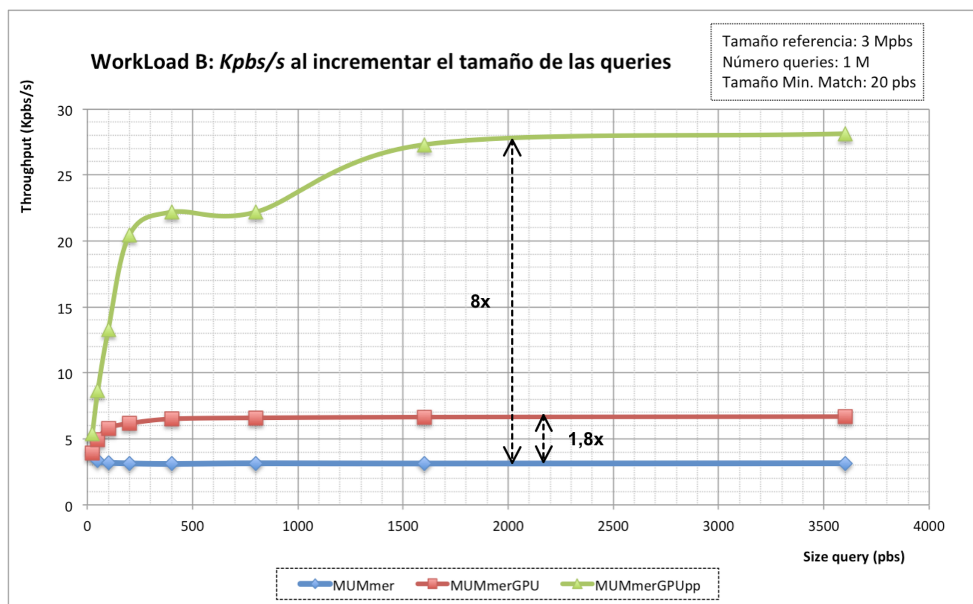


Figura 6.4: Rendimiento de MUMmer, MUMmerGPU y MUMmerGPU++ para el *workload* B, que incrementa el tamaño de las *queries*.

En la gráfica de la figura 6.5 se muestra de nuevo el rendimiento de las tres aplicaciones, pero ejecutando el *workload* C_0. El objetivo de esta experimentación es poder asegurar si el incremento en el rendimiento apreciado en el anterior análisis es a causa del aumento del tamaño de las *queries*.

En este experimento se ejecuta el *workload* C_0, se ha fijado *tamaño_queries*numero_queries* a 1Gbases. En las ejecuciones se incrementa el tamaño de la *query* por lo que el total de *queries* entonces disminuye. Esto supone que cada ejecución procesa el mismo tamaño de carga de trabajo como entrada. Se observa una relación directa entre el incremento del tamaño de las queries y el aumento de rendimiento de la aplicación *MUMmerGPU++,* pasando de un aumento del rendimiento de 6,7x a 7,4x respecto la versión serie. En un análisis posterior de la gráfica 7.13, que nos aporta más información, se evalúa el origen de esta mejora en el rendimiento en las partes más altas del gráfico.

Considerando un tamaño de *query* hasta de 750 pbs encontramos que *MUMmerGPU++* tiene ineficiencias donde *MUMmerGPU* si obtiene mayor rendimiento y el speedup se reduce a 2x.

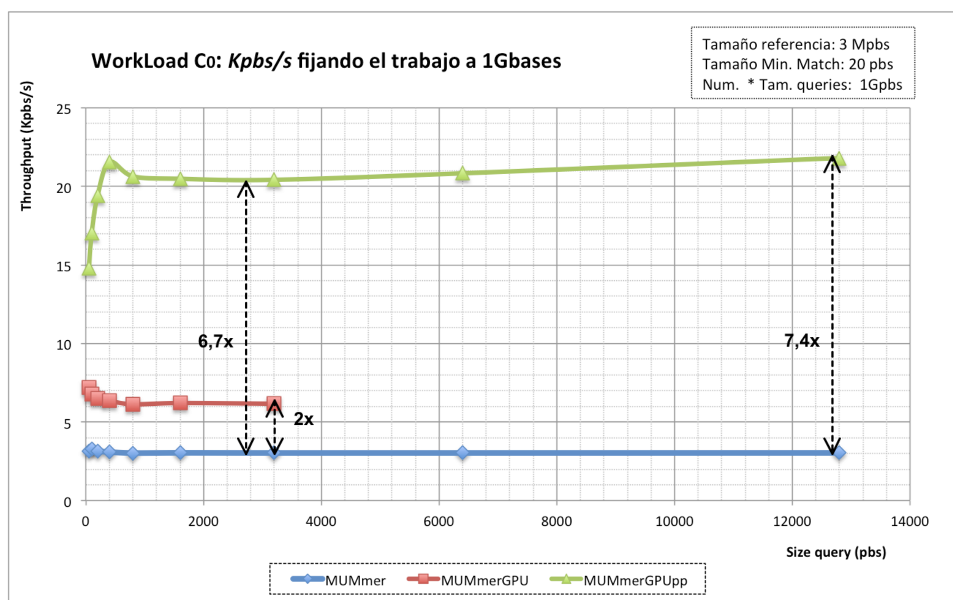


Figura 6.5: Rendimiento de MUMmer, MUMmerGPU y MUMmerGPU++ para el *workload* C_0, que incrementa el tamaño de las *queries* y mantiene una carga de trabajo fija.

En la siguiente experimentación se han tomado medidas del *workload* C_0 y C_1. Esto permite deducir si el aumento del tamaño del problema beneficia al rendimiento del *MUMmerGPU++*, de igual forma que beneficia aumentar el tamaño de la *query* visto en el análisis de la gráfica 6.4.

La figura 6.6 muestra que aumentar, en este caso 10 veces, el tamaño del problema supone un gran incremento en el rendimiento. Para un tamaño de *query* superior a ~6500 pbs se aprecia un decremento del rendimiento, así pues a estos tamaños del problema el tamaño de la *query* produce un impacto negativo en el rendimiento.

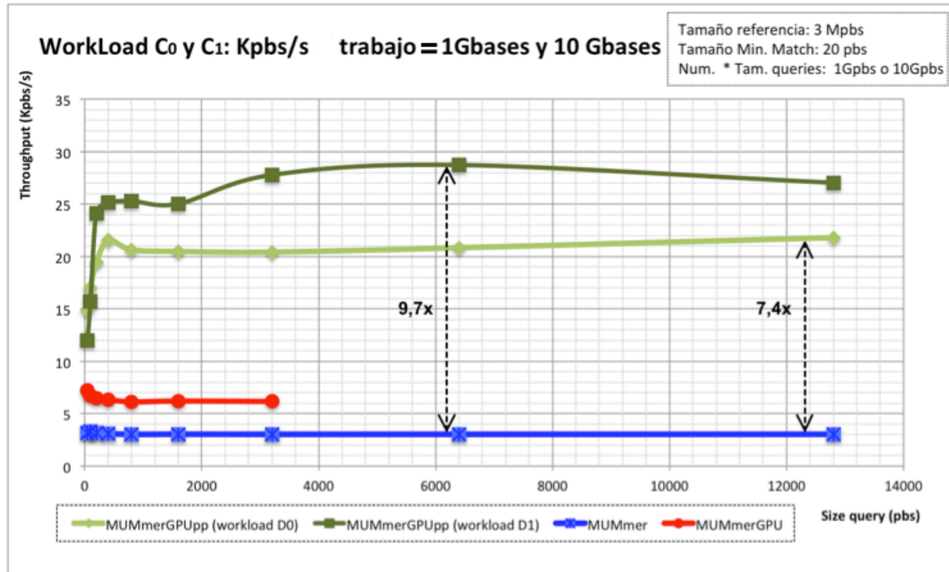


Figura 6.6: Rendimiento de MUMmer, MUMmerGPU y MUMmerGPU++ para el *workload* C_0 y C1, que incrementa el tamaño de las *queries* y mantiene una carga de trabajo fija en 1Gpbs y 10Gpbs.

En la siguiente experimentación que se muestra en la figura 6.7, se ejecuta el *workload* D. Este *workload* no varía la entrada de *queries,* como las experimentaciones anteriores*,* sino que aumenta la secuencia de *referencia* contra la que se realizan las búsquedas.

Se puede apreciar que un aumento de la referencia conlleva un impacto negativo en el rendimiento. Realmente al incrementar la referencia estamos realizando muchísimo más trabajo ya que ahora se deben buscar las *queries* en una secuencia de referencia mucho mayor. Pero es precipitado asegurar que el *cuello de botella* se encuentra en el código de la GPU. En el análisis de la figura 6.16 se identifica el origen de esta ineficiencia y también el motivo de que para tamaños grandes los tiempos entre CPU y GPU se acerquen cada vez más.

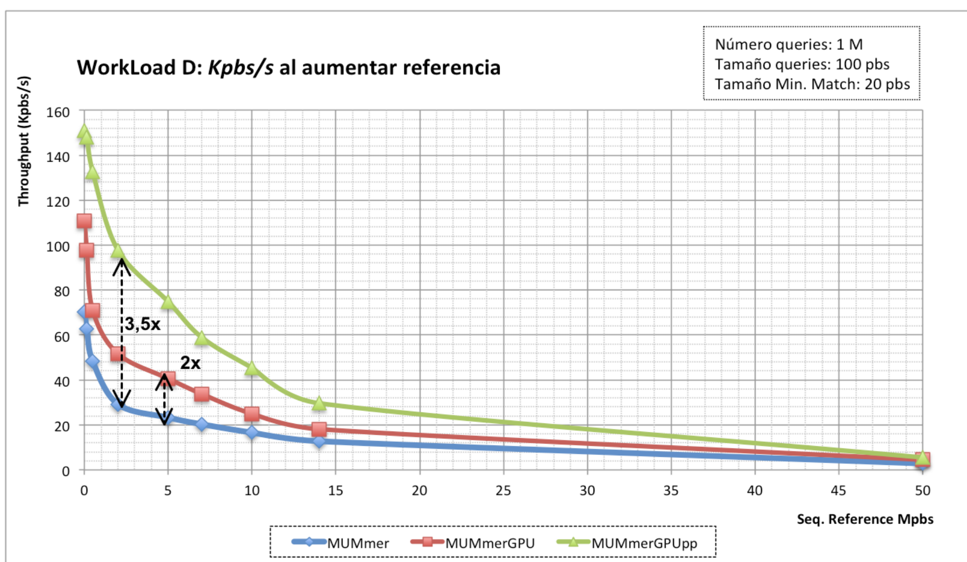


Figura 6.7: Rendimiento de MUMmer, MUMmerGPU y MUMmerGPU++ para el *workload* D, que incrementa el tamaño de la *referencia*.

En la siguiente gráfica 6.8 se muestra una experimentación realizada con el *workload* E, en el que se incrementa el tamaño mínimo de lo que se considera match. Se aprecia que aumentando el *minmatch* el rendimiento de la versión serie de *MUMmer* y la versión GPU *MUMmerGPU* alcanzan el rendimiento ofrecido por *MUMmerGPU++.*

Esto es debido a que el beneficio que se obtiene por parte de la GPU desaparece, debido a que aumentar el *minmatch* realiza un número mucho menor de búsquedas, hasta llegado el punto que cuando *minmatch* es igual al tamaño de la *query* únicamente se realiza una búsqueda.

Además, se debe añadir el *overhead* de preparar y enviar los datos a la GPU. El grafico 6.18 permite ver con más detalles este hecho.

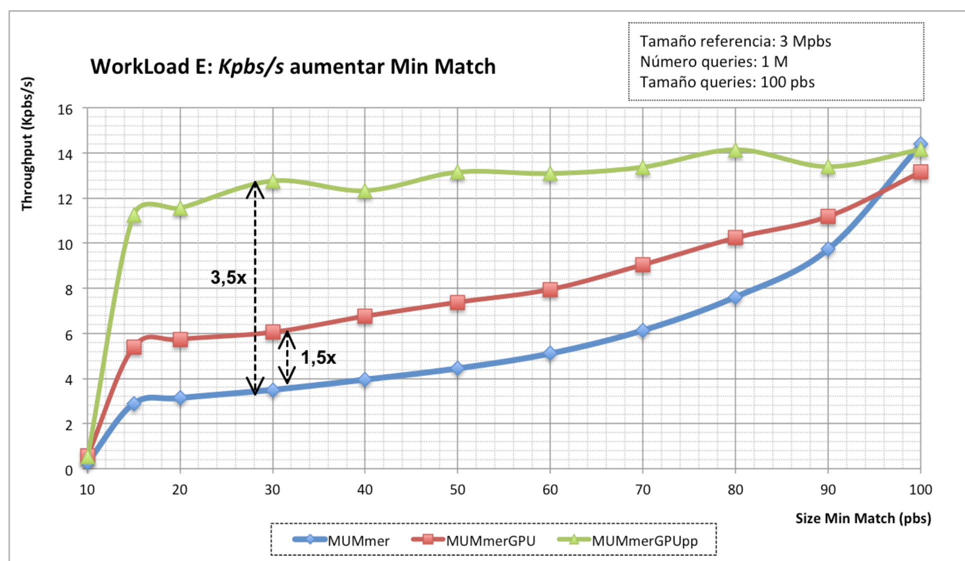


Figura 6.8: Rendimiento de MUMmer, MUMmerGPU y MUMmerGPU++ para el *workload* D, que incrementa el tamaño mínimo del match.

Una vez analizadas todas estas ejecuciones, de los workloads A, B, C, D y E. Se observa que el algoritmo *MUMmerGPUpp* es el que mayor rendimiento está obteniendo de los 3, es por ello que las siguiente pruebas que se realizan están enfocadas a esta implementación y son descritas en el capítulo 6.10.2.

7.10.2 Pruebas detalladas sobre *MUMmerGPUpp*

En este capítulo solo se ha incluido la evaluación en profundidad de la implementación *MUMmerGPU++* debido a que en la sección anterior 6.10.1 se ha observado que es la implementación que mayor rendimiento ofrece en todos los casos. Así pues es la implementación que mayor interés puede suscitar.

Este subcapítulo analiza el mismo conjunto de experimentos que los ya realizados en el capítulo anterior 6.10.1, pero en este caso recogiendo otros datos, que corresponden a segmentos de ejecución del código. Las mediciones se realizan mediante la primitiva de C *gettimeofday* con precisión en milisegundos. Esta primitiva engloba segmento de código que se desea instrumentar.

Esta segunda ejecución de los *workloads* se realiza para poder explicar las ineficiencias anteriormente comentadas, de las cuales era necesaria una información más detallada de las ejecuciones. Para poder obtener más información relevante, se miden partes del código que corresponden a distintas fases de la ejecución. Estas fases corresponden a etapas del algoritmo vistas en todos los capítulos anteriores de la memoria y corresponden a las siguientes:

- *refpreprocesing*: lectura y procesado en la CPU de la secuencia de referencia S utilizando el algoritmo de indexación *Enhanced Suffix-Array*.
- *referencetoGPU*: transferencia de la referencia S y su estructura de indexación Enhanced Suffix-Array desde la CPU hacia la GPU.
- *queriesfromdisk*: lectura de disco del conjunto Q de *queries.*
- *querytoGPU: transferencia del conjunto Q de queries.*
- *Kernel*: ejecución en la GPU de la etapa de búsqueda con el algoritmo del *Enhanced Suffix-Array* de las queries en la *referencia* S.
- *outputfromGPU:* transferencia del resultado comprimido desde la GPU hacia la CPU.
- *Printmatches*: descompresión del resultado ejecutada en la CPU, denominado postprocesado.

Las gráficas que se presentan en esta sección básicamente son dos: el porcentaje de ejecución de cada una de las fases en MUMmerGPU++ sobre toda la aplicación y el tiempo normalizado en función de la carga de trabajo en cada una de las fases.

La siguientes figuras 6.8 y 6.9 muestran datos de ejecución de las diferentes fases del algoritmo ejecutando el conjunto de *workloads* A. Esta visión de las graficas permite comparar tiempo de ejecución de cada fase en la implementación MUMmerGPU++.

Se observa que el periodo de ineficiencia, que se comentaba en el capítulo anterior, para tamaños de entrada menores a 7,5 millones de *queries* se puede observar que es debida a la generación *Enhanced Suffix-Array* de la estructura de la referencia. En estos intervalos se pierde más tiempo en realizar el preprocesado de la referencia que no en computar las búsquedas.

La grafica 6.10, nos muestra que en las etapas a las que se dedica más tiempo, para un número de *queries* superior a 7,5 millones, son la lectura de queries del disco y el post procesado del resultado. Ambas etapas se ejecutan en la CPU.

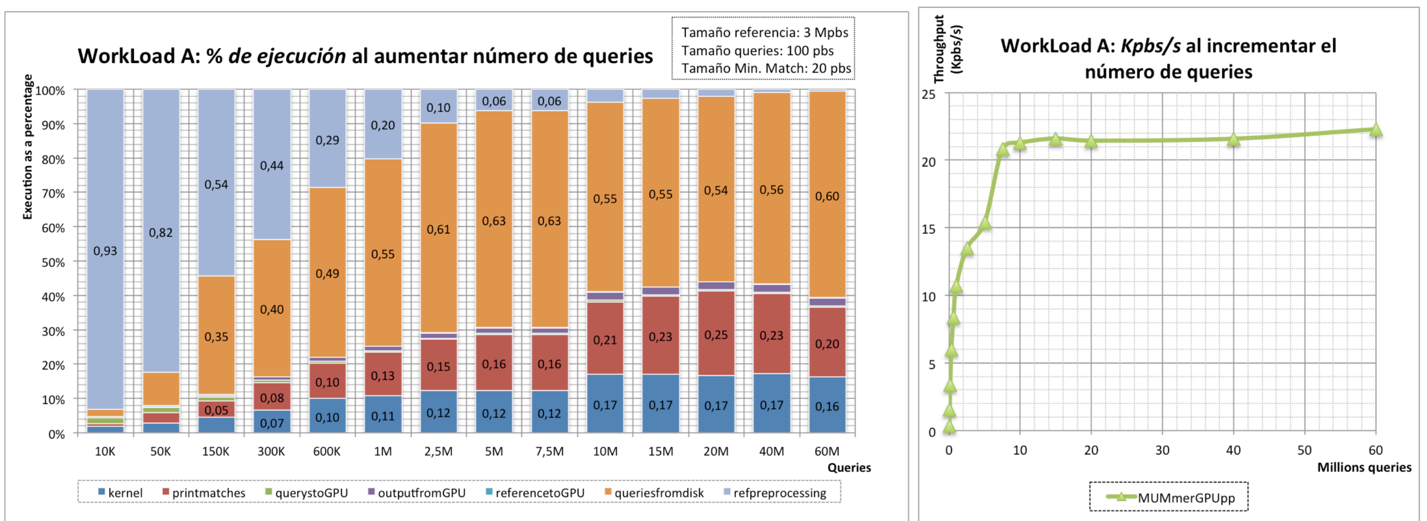


Figura 6.9: Porcentaje de ejecución de cada una de las fases en *MUMmerGPU++* para el *workload* A.

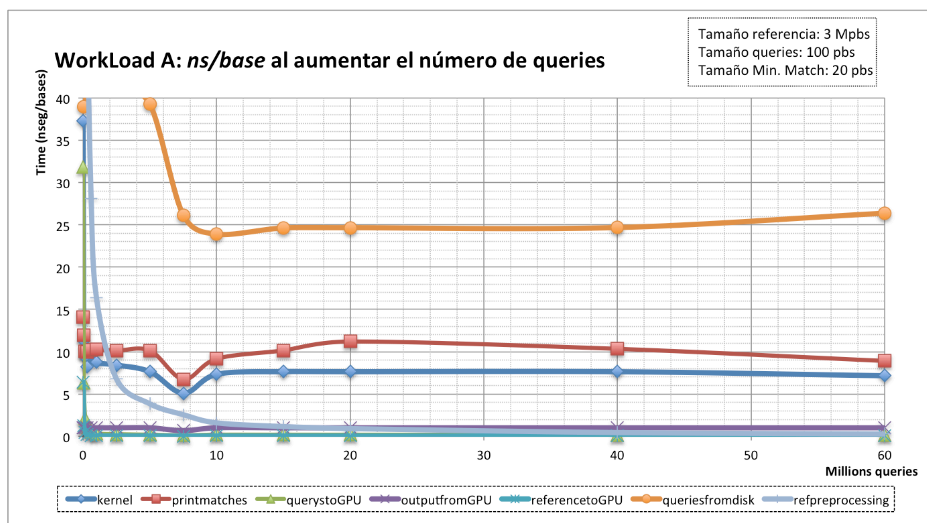


Figura 6.10: Tiempo de ejecución de cada fase normalizado, para la ejecución del *workload* A con *MUMmerGPU++*.

La gráfica 6.11 muestra que el mayor consumo del tiempo del programa está focalizado en la lectura de *queries* de disco y en la ejecución del *kernel*.

Aumentar el tamaño de las *queries* afecta positivamente a la lectura de disco donde aumenta la eficiencia y reduce los tiempos.

Se detecta una anomalía en la lectura de las *queries* del disco para el tamaño de *query* 800pbs que no podemos indicar a que es debida, pero se han realizado diversas ejecuciones para comprobar la fiabilidad del resultado y no varía.

Las ineficiencias detectadas anteriormente para tamaños con cargas pequeñas de trabajo (<750 de tamaño) vienen dados por el poco cómputo a ejecutar, existe un alto *overhead*, mayor que el tiempo dedicado al cómputo, y se ve reflejado en que la indexación toma un tiempo considerable, entre un 10 y 47% del total.

Finalmente, el tiempo de preprocesado de la referencia, al ser para todo el workload del mismo tamaño siempre tarda el mismo tiempo. Pero en relación a los tiempos del resto se reduce.

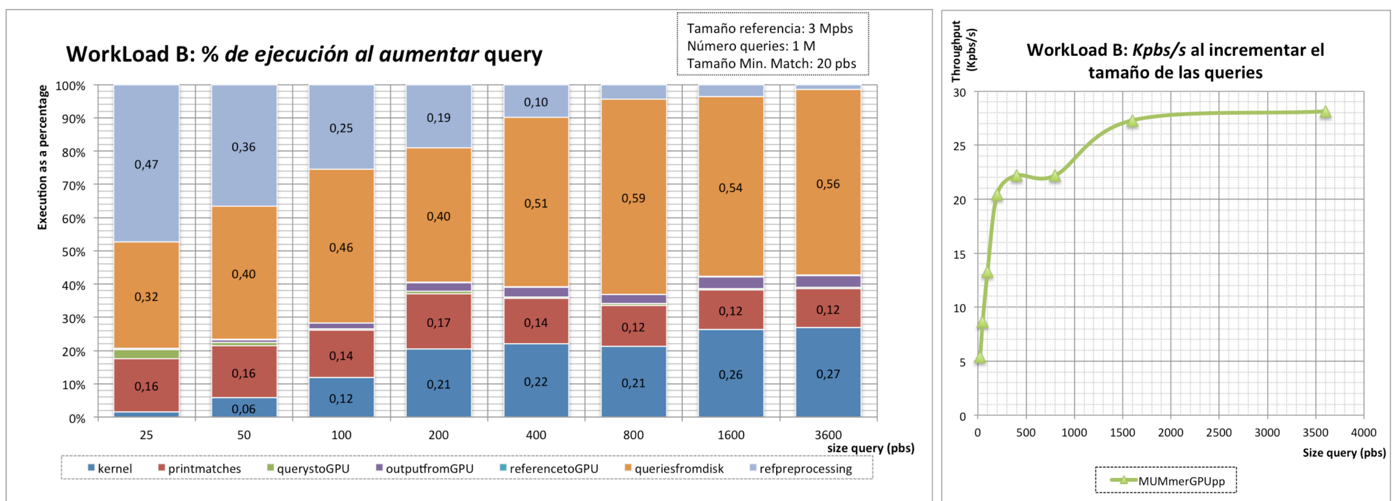


Figura 6.11: Porcentaje de ejecución de cada una de las fases en *MUMmerGPU++* para el *workload* B.

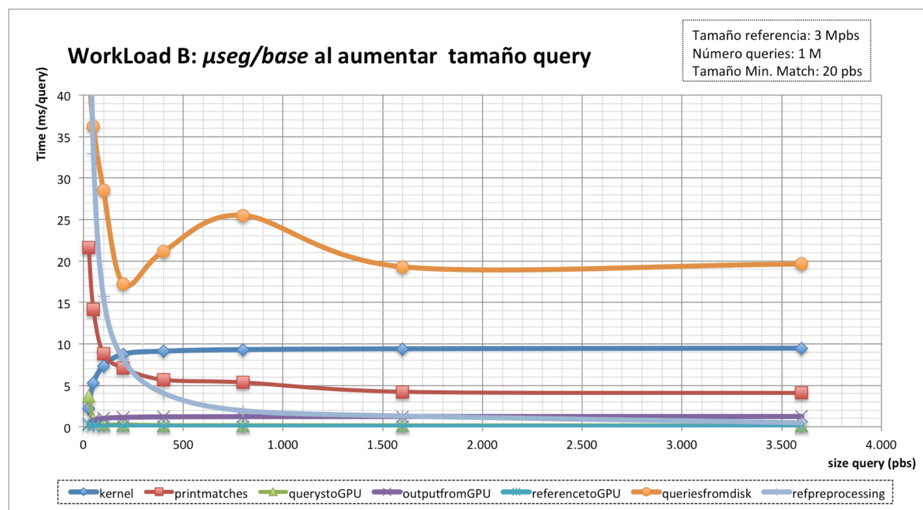


Figura 6.12: Tiempo de ejecución de cada fase normalizado, para la ejecución del *workload* B con *MUMmerGPU++*.

Como se ha comentado en el análisis de la gráfica anterior (figura 6.11), aquí (figura 6.13) se comprueba que al aumentar el tamaño de la query aumenta el rendimiento de la aplicación. La mejora es debida a la fase de las lecturas de las *queries* del disco. Este proceso se torna más eficiente con el incremento del tamaño de las *queries*.

En un primer momento en el análisis anterior sin conocer el tiempo de ejecución del *kernel*, se podía llegar a una conclusión errónea, que el incremento del rendimiento es debido a la implementación del algoritmo de búsquedas *Enhanced Suffix-Array.* El algoritmo permite reaprovechar cómputo y se ve beneficiado de *queries* de mayor tamaño. Pero no es el caso, el *kernel* mantiene su eficiencia constante y la etapa que verdaderamente contribuye es el acceso a disco como ya se ha comentado. Cerca de un 65% del tiempo es destinado al acceso a disco.

Como línea abierta del proyecto sería muy interesante medir el impacto de la funcionalidad denominada *Suffix-Links en las estructuras Enhanced Suffix-Array* en la ejecución del *kernel*. Aquí se puede ver que aparentemente no tiene un gran impacto en la ejecución. Únicamente en *queries* muy pequeñas se aprecia este impacto pero de forma muy moderada y en este caso el acceso a disco y el post procesado ocultan esta mejora debido a los altos tiempos que toman.

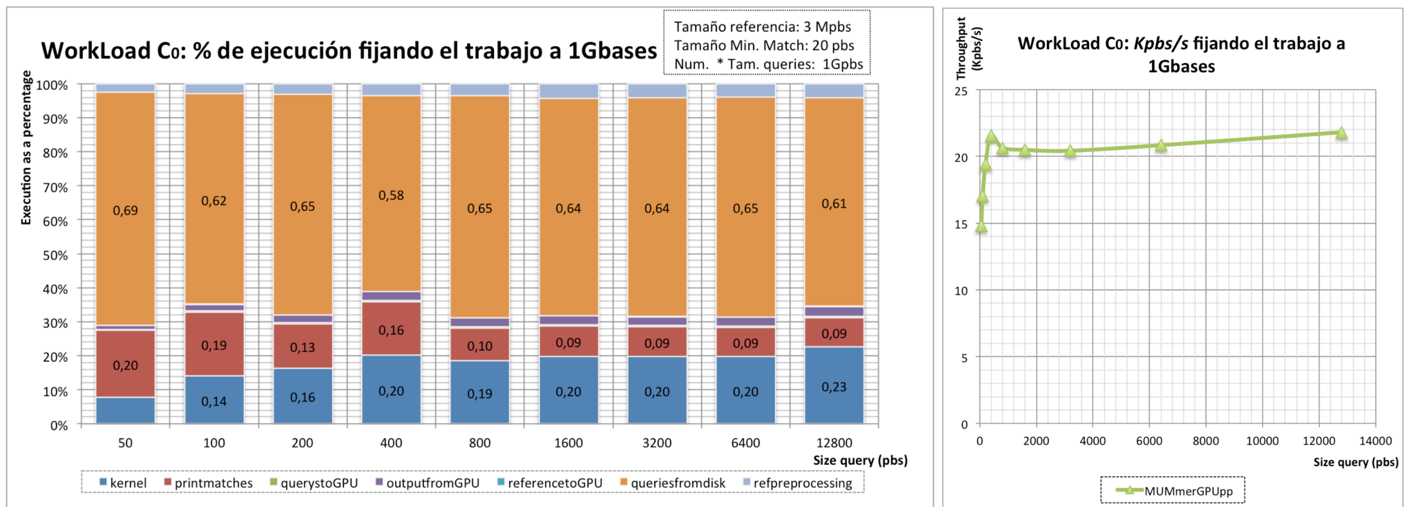


Figura 6.13: Porcentaje de ejecución de cada una de las fases en *MUMmerGPU++* para el *workload* C_0.

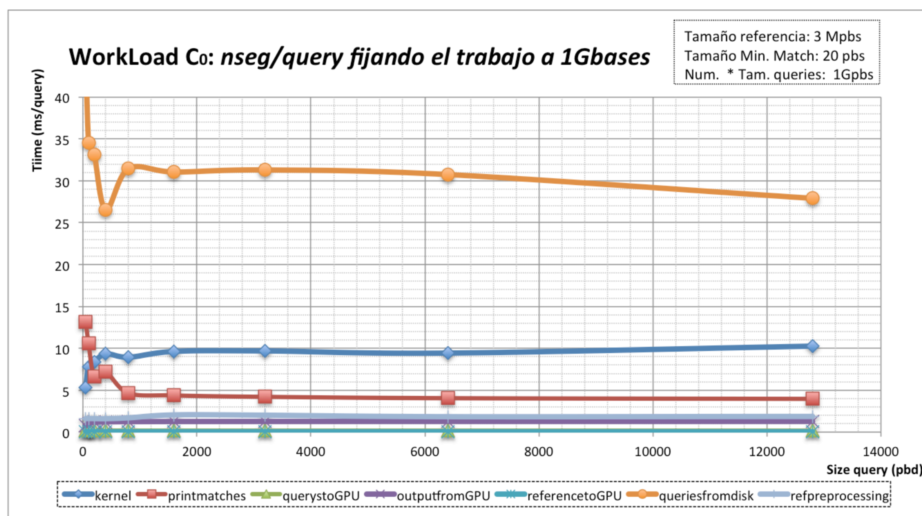


Figura 6.14: Tiempo de ejecución de cada fase normalizado, para la ejecución del *workload* C_0 con *MUMmerGPU++*.

En el posterior análisis del workload D, se dejó entrever que el problema de rendimiento podía estar ubicado en el cómputo de la GPU, ya que debido a que en un primer momento el incremento de la referencia podría indicar que se deben realizar más búsquedas ya que la secuencia es mayor.

En este análisis, gráficas 6.15 y 6.16, se observa que este planteamiento es erróneo y para tamaños de referencia mayores a 5 millones, las fases que toman más tiempo son la generación del índice de la referencia y el postprocesado. El postprocesado puede llegar a tomar hasta un 66% del tiempo total de la ejecución. Ambas fases únicamente se ejecutan en CPU.

Como optimización en líneas futuras abiertas puede plantearse hacer un estudio del algoritmo de postprocesado e intentar portarlo a GPU, del mismo modo que lo realiza *MUMmerGPU*. Este proceso ha sido estudiado por encima en este proyecto y es un proceso altamente paralelizable. Incluso plantearse realizar el preprocesado (la generación del *Suffix-Array*) en la propia GPU, esto incluso permitiría eliminar gran parte de las transferencias que se deben realizar entre CPU-GPU.

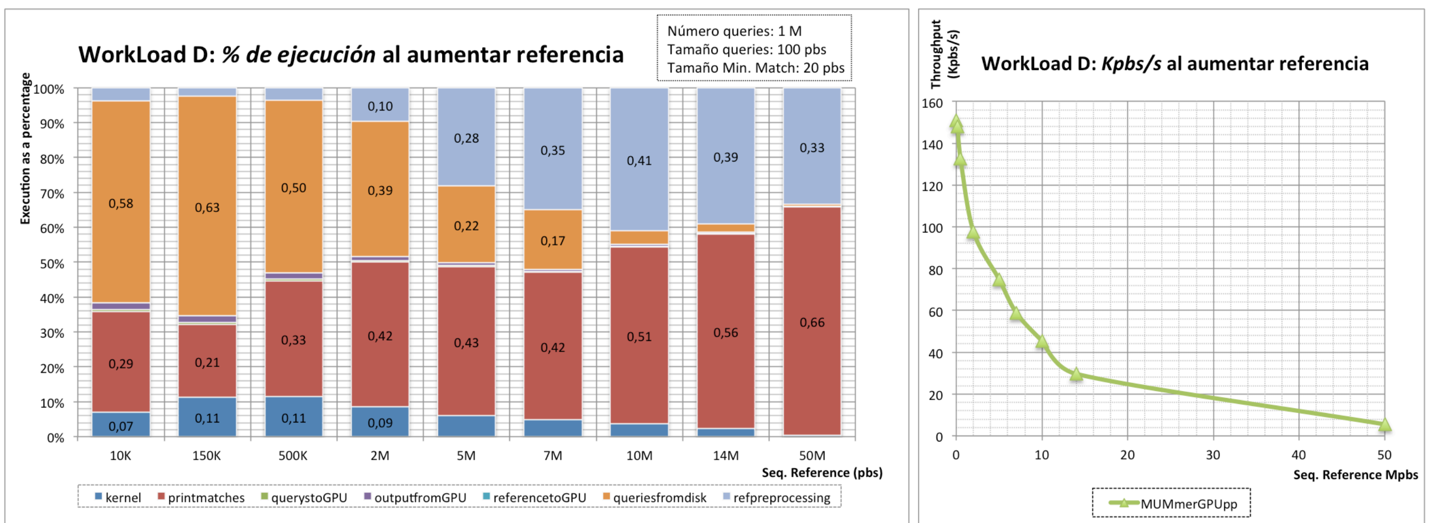


Figura 6.15: Porcentaje de ejecución de cada una de las fases en *MUMmerGPU++* para el *workload* D

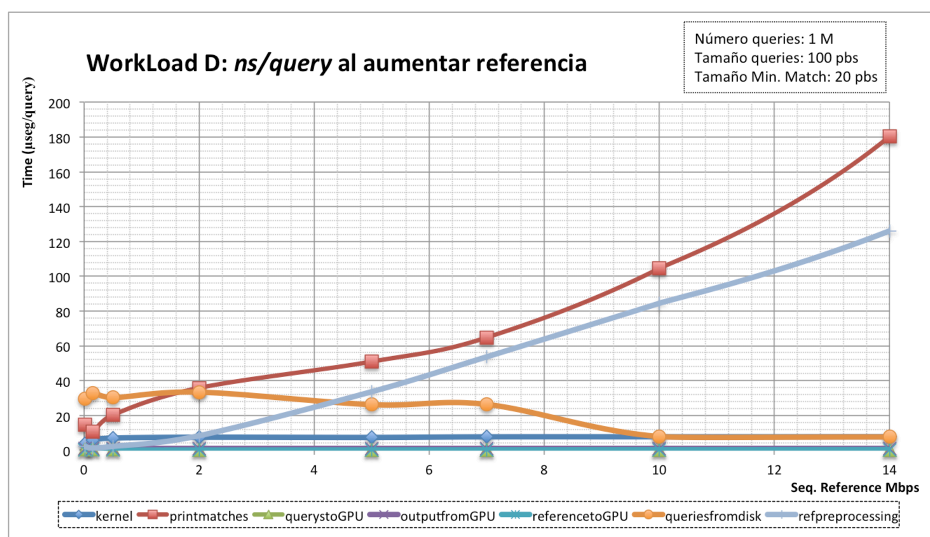


Figura 6.16: Tiempo de ejecución de cada fase normalizado, para la ejecución del *workload* D con *MUMmerGPU++*.

Tal como se comenta en el análisis de la gráfica 6.7, las ineficiencias para grandes tamaños de *minmatch* vienen dadas por el reducido uso que se le esta dando al kernel. El resto de fases que son ejecutadas son consideradas en relación como un *overhead* muy elevado. Incluso llega un punto que el *minmatch* toma el mismo valor que el tamaño de la *query,* cuando se da este caso es desastroso se tienen tiempos de transferencia a la GPU más elevados que el propio computo que se realiza. Por lo que *MUMmerGPU++* acaba teniendo hasta un rendimiento más pobre que la implementación serie *MUMmer*.

La ineficiencia para un tamaño de minmatch de 10 es debida a que con un tamaño tan pequeño el número de MEMs que se encuentran es elevadísimo y el proceso de postprocesado tiene un coste muy alto, tan alto que toma el 97% del tiempo de la ejecución.

Los puntos con mayor impacto en este análisis son la infrautilización de la GPU para valores muy alto de *minmatch* y para valores bajos el enorme cómputo que se realiza en el postprocesado de la CPU.

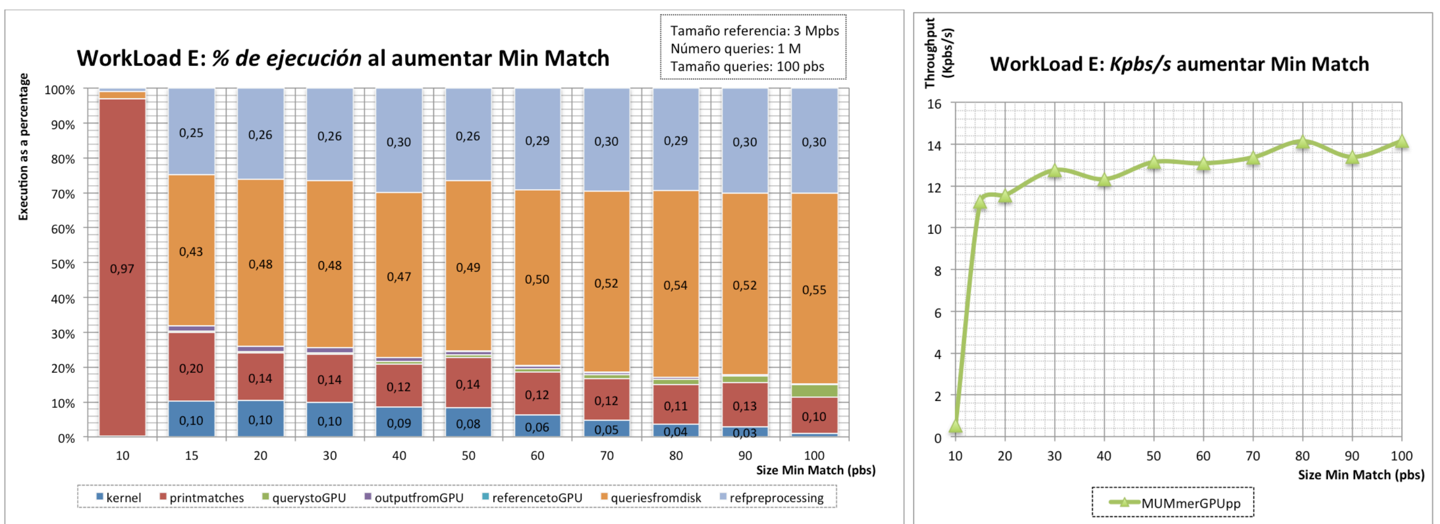


Figura 6.17: Porcentaje de ejecución de cada una de las fases en *MUMmerGPU++* para el *workload* E

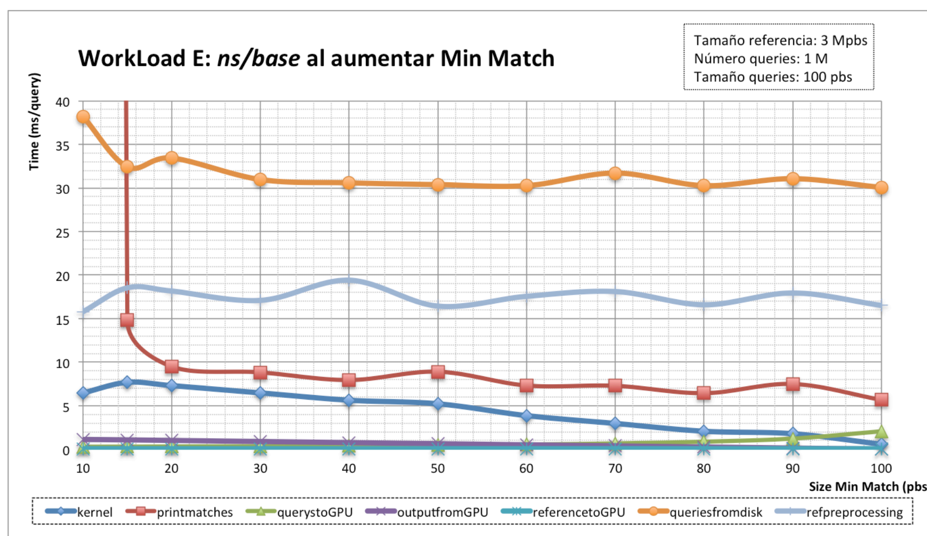


Figura 6.18: Tiempo de ejecución de cada fase normalizado, para la ejecución del *workload* E con *MUMmerGPU++*.

6.10.3 Pruebas detalladas sobre el ancho de banda

Estas experimentaciones han sido realizadas para comprobar la eficiencia de lectura del disco en el momento de leer las *queries*. Para ello se ha tomado los valores de lectura del *workload* A y se han comparado con diversas medidas de lectura tomadas con herramientas para saber el ancho de banda del disco.

Para las mediciones se ha utilizado la herramienta *hdparm* e *ioping*, permiten calcular el ancho de banda en *bruto* del disco, que en este caso ha sido 59MB/s. Esto permite hacernos una idea de si estamos alcanzando una buena utilización del ancho de banda del disco o no. Tener valores por encima de este significaría que las mediciones no se están realizando correctamente. El valor teórico del ancho de banda del disco duro se indica en las graficas como *theoric* de color negro.

También un segundo valor para poder hacer comparaciones mas precisas. Se calcula el tiempo de leer el mismo fichero que lee *MUMmerGPU++* en el workload. El fichero se lee con el comando *cat* y se anota el tiempo de lectura de todo el fichero. De este modo podemos tomar un valor de lectura del fichero sin tener el programa ejecutándose simultáneamente. Estos valores se representan en la gráfica con la etiqueta *raw* de color verde pistacho.

Finalmente, se representan los datos de lectura a disco de la fase de lectura de *queries* de *MUMmerGPU++* de color naranja con la etiqueta *queriesfromdisk*.

Todos los datos están representados como ancho de banda en *MB/s* esto permite realizar una comparación entre todas las ejecuciones del *workload*.

En la figura 6.19, se observa que para tamaños de workload superiores a 7,5 millones de queries (que representan ~715MB de datos) se obtiene una eficiencia cercana a la lectura del fichero con la herramienta *cat.* Por lo que se considera que la lectura de disco es suficientemente eficiente.

En la figura 6.20 se ha realizado previamente una lectura de disco del mismo fichero, por lo que existe una copia en la caché del sistema antes de tomar las medidas de *queriesfromdisk*. En esta gráfica se puede apreciar que para valores por debajo de los 20 millones de queries (~1,9GB) el acceso a disco es tan eficiente que supera al límite medido en bruto. Esto indica que estas lecturas por debajo de 20 millones de queries es seguro que están utilizando la caché del sistema, pudiendo llegar a alcanzar los 140MB/s leyendo de memoria principal.

Igualmente este efecto con los datos cacheados no es demasiado realista considerarlo en el entorno en que se están realizando las pruebas y se ciñe siempre a conjuntos de datos menores de ~2GB cuando los conjuntos de datos de entrada pueden llegar a ser más elevados.

Es interesante observar que para valores mayores de 40 millones de *queries* las lecturas están dando peores resultados que los datos no cacheados.

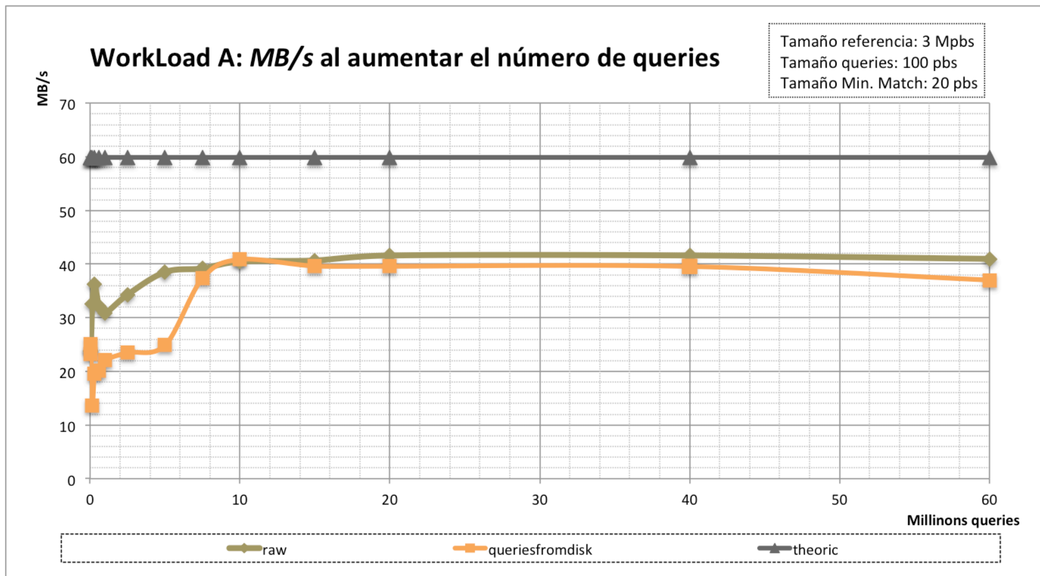


Figura 6.19: Ancho de banda en *MB/s* con el *workload* A en la fase de lectura de *queries*.

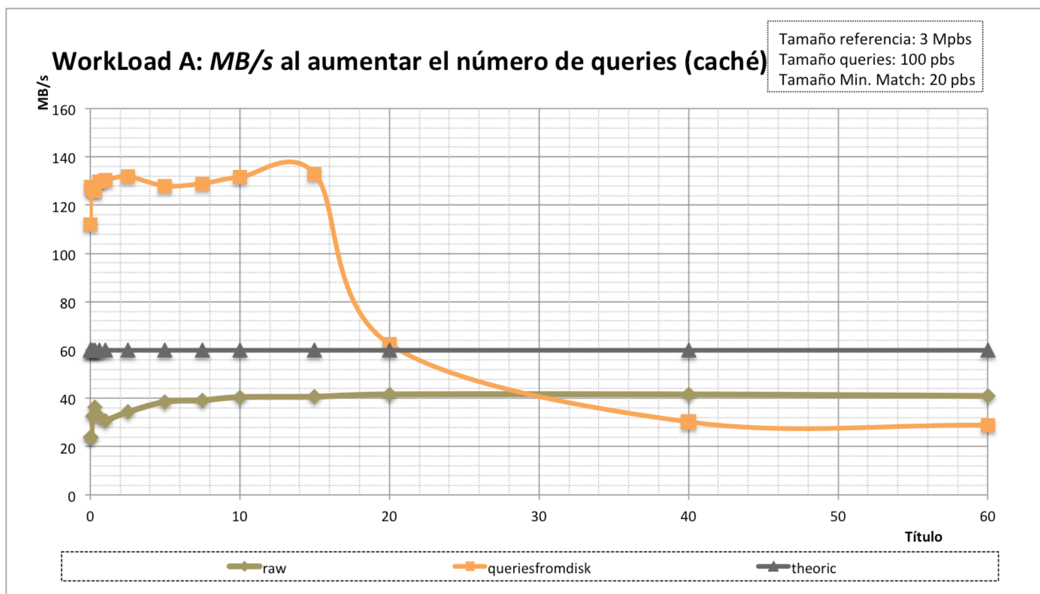


Figura 6.20: Ancho de banda en *MB/s* con el *workload* A en la fase de lectura de *queries*, previamente siendo cacheada.

6.10.4 Pruebas detalladas sobre el impacto del acceso a disco en la aplicación.

En la gráfica 6.21 se desea demostrar el impacto de la lectura del disco en el algoritmo. Se sospecha que la GPU está ociosa constantemente esperando datos y el disco le supone un cuello de botella muy grande.

Para ello se ejecutan los *worksloads* C_0 y C_1 tanto haciendo uso de la memoria caché del sistema y teniendo todos los datos en ella desde un inicio (indicados como *cached* en la leyenda) se recogen los datos y se grafican.

Se encuentra una anomalía en el comportamiento para queries de tamaño 1800 donde la ejecución con datos en caché para tamaño del problema 1Gbases y 10 Gbases no mantienen el rendimiento esperado.

Así pues, teniendo los datos en memoria *MUMmerGPU++* puede conseguir un rendimiento de hasta 14,5 veces mayor que la versión serie, y si la lectura se realiza directamente de disto se consigue un rendimiento de hasta 9 veces superior al serie.

Pro lo tanto, el disco tiene un impacto muy grande en el rendimiento del algoritmo. Para solucionar estos temas bien es posible realizar una sintonización en el sistema o quizás simplemente el entorno en el que se están realizando las experimentaciones no sea seguramente el más adecuado.

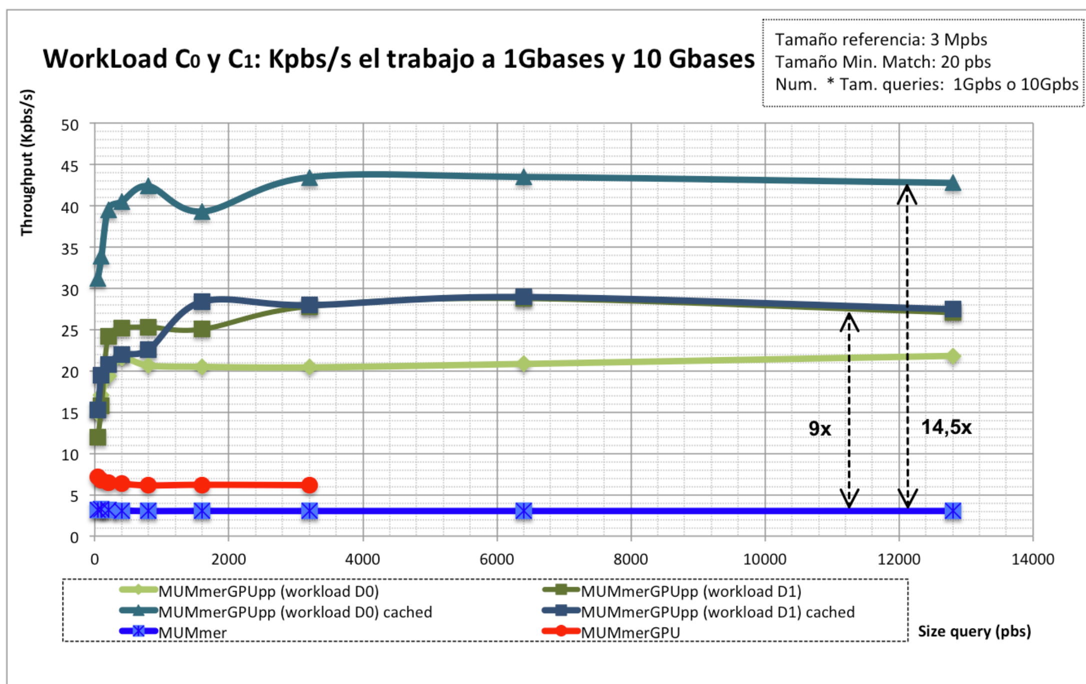


Figura 6.21: Rendimiento de *MUMmerGPU++* con el *workload* C_0 y C_1con datos cacheados y sin cachear.

6.10.5 Pruebas comparativas entre *MUMmerGPU++* y *MUMmerGPU*.

En esta subsección se hace una breve comparación entre la ejecución de *MUMmerGPU* y *MUMmerGPU++*, mediante un ejemplo. Este análisis, se realiza a muy alto nivel, ejecutando el conjunto de *workloads* A en los dos programas.

La gráfica 6.22 muestra el tanto porciento de ejecución de cada fase respecto todo el programa en el conjunto de *worksloads* A ejecutado con *MUMmerGPU++*, mientras que la gráfica 6.21 muestra la ejecución sobre *MUMmerGPU*.

El realizar la comparación de estos programas puede ayudar a dar una visión global de las diferencias en la implementación y comportamiento que presentan.

Por ejemplo, se puede comprobar que una implementación basada en *Suffix-Tree* frente a una en *Enhanced Suffix-Array* presenta un tiempo de transferencias entre la GPU y CPU mayor.

Debido a que el tamaño de la estructura ocupa un mayor espacio y por tanto las transferencias son más costosas.

Por otro lado, se observa que el tiempo de generación de la estructura de indexación en *MUMmerGPU* también es mayor. Este tiempo implicará que el tamaño del problema a ejecutar debe ser mucho mayor para poder amortizar ese tiempo y que el tiempo de las transferencias con la GPU se verán incrementadas.

También se observa que el postprocesado en *MUMmerGPU++* representa un tiempo menor respecto a todo el programa que no el postprocesado de *MUMmerGPU*. El postprocesado de *MUMmerGPU* se realiza en la GPU mediante iteraciones, mientras que el de *MUMmerGPU++* se realiza en la CPU. Otra línea futura podría ser realizar otro estudio en ese aspecto y analizar la ejecución CPU y GPU de esta fase.

También se observa que en *MUMmerGPU* el *kernel* está computando un menor tiempo respecto a toda la aplicación que en la implementación *MUMmerGPU++*. Esto supone que existe un mayor overhead en la ejecución.

Es necesario un estudio en profundidad para sacar conclusiones precisas, pero esta vía puede resultar interesante para aprender que beneficios e inconvenientes tienen las dos estructuras de indexación que hacen uso cada un de los dos programas y poder aplicar este estudio a una tercera implementación de otro programa, pudiendo mejorar su eficiencia.

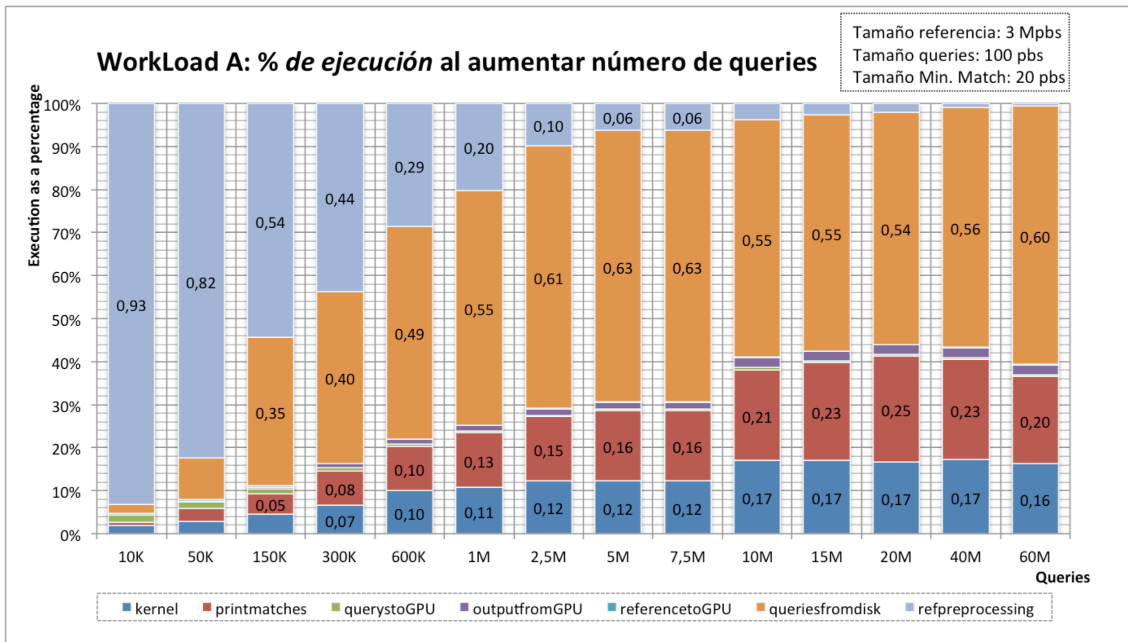


Figura 6.22: Porcentaje de ejecución de cada una de las fases en MUMmerGPU++ para el workload A.

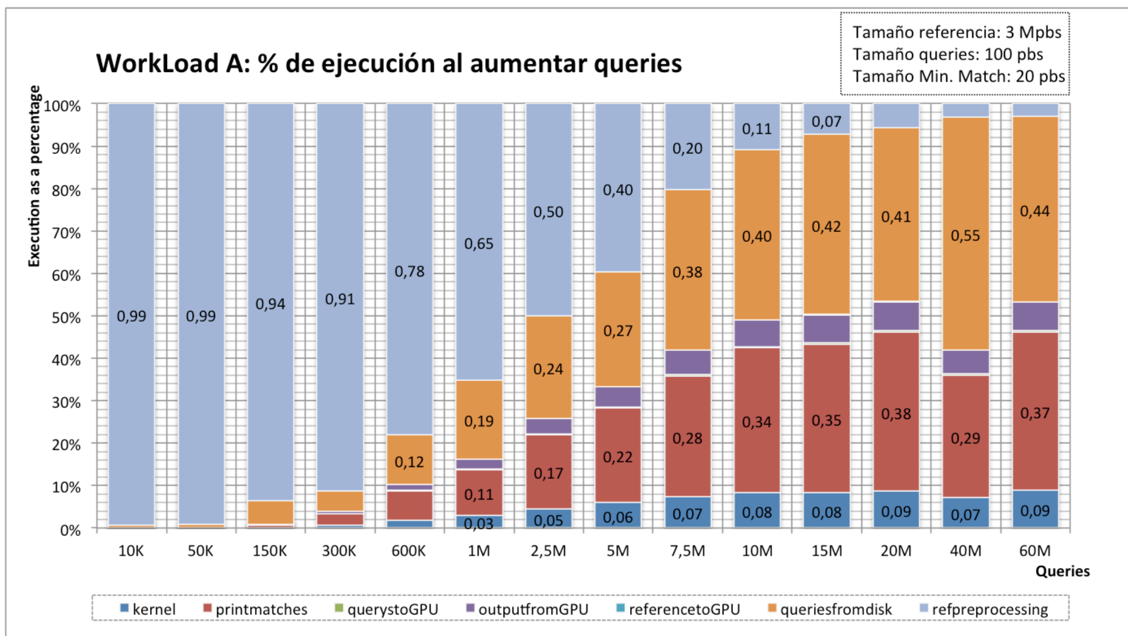


Figura 6.23: Porcentaje de ejecución de cada una de las fases en MUMmerGPU para el workload A.

6.10.6 Pruebas específicas de la arquitectura.

En esta sección de la memoria se ha realizado un estudio del impacto que tiene un código compilado específicamente para la arquitectura de la GPU.

Para ello, se ha realizado la misma ejecución de experimentos descrita en el anterior capítulo 6.10.1, pudiendo así medir el rendimiento en *Kpb/s* que obtiene cada una de las ejecuciones.

Los programas a analizar son todos distintas compilaciones de la implementación *MUMmerGPU++*, estas compilaciones han sido generadas con diferentes parámetros de compilación que son específicos de la arquitectura de la *GPU Fermi*. Los parámetros pueden verse con más detalle en el capítulo 6.7.

En la memoria sólo se mostrará el resultado de la ejecución de uno de los workloads, concretamente el A, ya que el resto de experimentaciones con los otros workloads han mostrado el mismo comportamiento.

En la gráfica 6.24 se observa que las ejecuciones de las compilaciones de MUMmerGPU++ obtienen todas aproximadamente los mismos resultados. A simple vista se podría decir que compilar con un parámetro u otro no tiene un impacto directo en el comportamiento de este programa. Pero para poder dar respuesta a esta pregunta se debe realizar un análisis más exhaustivo.

Por ello, en la figura 6.25 se ha procedido a medir las diferentes fases del algoritmo del mismo modo que en el capítulo 6.10.2.

De este modo, se puede observar que la compilación que deshabilita la caché de primer nivel en la GPU, obtiene un speedup de alrededor del 15% (esto sucede con todas las ejecuciones). Esto

es debido a que la GPU está realizando un uso del bus de datos más eficiente, dado que, al desactivar la caché L1 del dispositivo, los accesos a memoria principal se realizan de tamaño 32Bytes en lugar de 128Byes.

Debido al tipo de accesos pseudoaleatorios que tiene este programa, los accesos de 128Bytes son realizados sin localidad espacial y por tanto no suponen un beneficio en la ejecución.

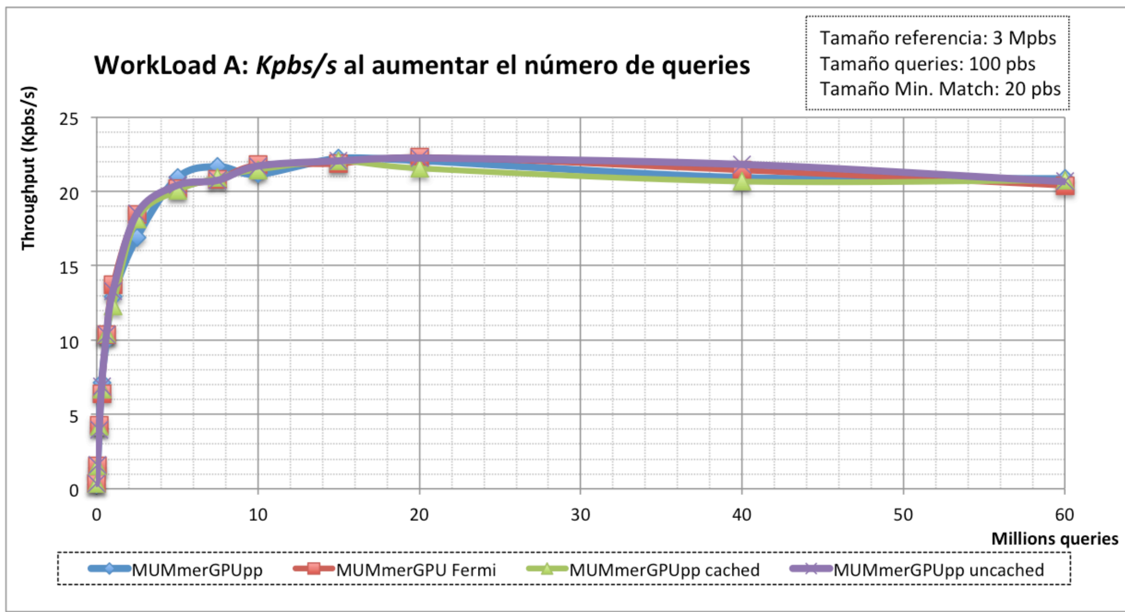


Figura 6.24: Rendimiento de la aplicación MUMmerGPU++ y sus distintas optimizaciones específicas de la arquitectura *GPU Fermi* con el *workload* A

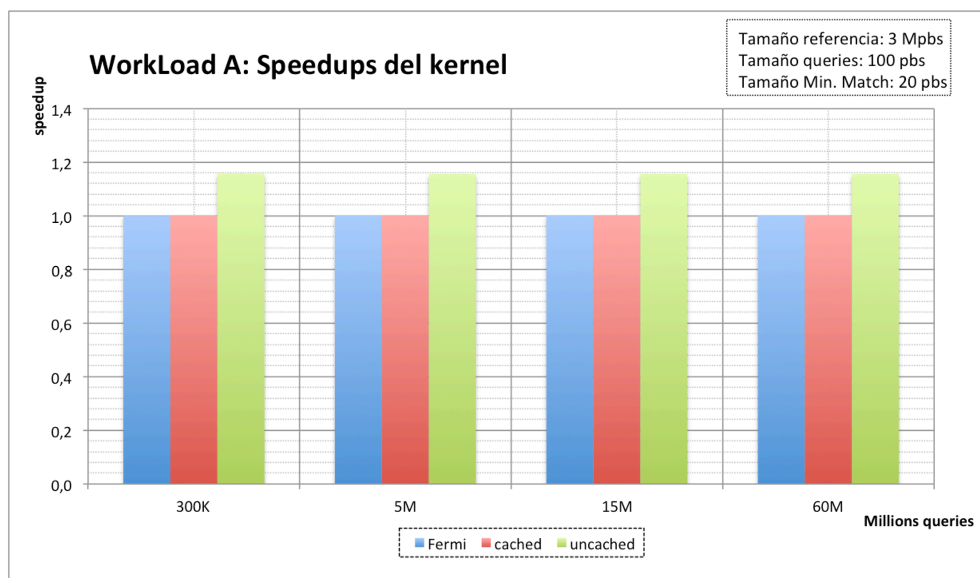


Figura 6.25: Speedup de la aplicación MUMmerGPU++ y sus distintas optimizaciones específicas de la arquitectura *GPU Fermi*.

6.11 Puntos clave del capítulo

- Un punto importante en la evaluación experimental de las aplicaciones ha sido parte de la metodología a seguir, donde se ha comprobado que todas las aplicaciones retornasen el mismo resultado y así asegurarnos que se evalúa la misma funcionalidad.
- También se ha comprobado que parámetros de compilación compartían las aplicaciones. Y se han utilizado los parámetros que maximizaban el rendimiento del conjunto. Es importante que todos compartan los mismos parámetros de compilación para que se apliquen las mismas optimizaciones al conjunto de aplicaciones.
- En la elección del workload se ha buscado el más representativo de la aplicación, para ello se ha iterado carias veces el proceso de selección del workload.
- Los datos utilizados para la carga de trabajo han sido datos genéticos reales, obtenidos de la base de datos *genbank*. Los datos para las queries son generados a partir de un script desarrollado por los autores de MUMmer. Es un script ampliamente utilizado en los análisis genéticos de la comunidad científica.
- En el proceso de ejecución de las experimentaciones se ha generado un script con el objetivo de automatizar las pruebas. Esto permite reducir tiempo empleado en lanzarlas y posibles errores humanos al realizar los lanzamientos.
- En las mediciones realizadas se ha utilizado métricas acordes al análisis que se desea realizar. Se han utilizado las métricas siguientes:
    - Kpbs/s: permite comparar rendimientos entre ejecuciones con un tamaño de problema diferente.
    - GB/s: permite comparar el rendimiento de lectura o escritura entre diferentes conjuntos de datos.
    - ns/base: con esta métrica se consigue una normalización de los resultados para comparar tiempos de ejecuciones con distinta carga de trabajo.
    - msec: es una métrica de tiempo absoluto ideal para comparar las ganancias entre ejecuciones.
- En el análisis de las aplicaciones se ha podido comprobar que *MUMmerGPU++* puede llegar a tener ganancias de 10 veces mayores que la ejecución en serie *MUMmer*.
- También se ha identificado que en la mayoría de las experimentaciones existe un cuello de botella a causa de la lectura de queries del disco.
- Otra ineficiencia identificada es por parte del proceso de postprocesado que descomprime el resultado de la GPU. En casos con tamaño de minmatch reducido o un tamaño de secuencia considerable, el porcentaje del tiempo de ejecución que toma el post procesado es muy elevado. Pudiendo llegar en muchos casos al 60%.
- Se ha comprobado la eficiencia de lectura en disco, y los resultado son razonables. Cuasi se alcanza el ancho de banda de lectura conseguido leyendo un fichero de iguales condiciones de forma secuencial
- Por otro lado se ha podido verificar que el disco genera una ineficiencia en el cómputo de la GPU donde leyendo los datos de memoria principal se lograría un speedup frente la versión serie de 15x.

CAPÍTULO 7 – CONCLUSIONES Y LÍNEAS ABIERTAS

La caracterización y el análisis realizado en el proyecto de las tres aplicaciones de *MUMmer*, ha supuesto realizar un estudio sobre diversos ámbitos: los algoritmos de comparación de genomas, los algoritmos de indexación de datos, las arquitecturas *ManyCores* y conceptos bioinformáticos generales. Todo ello a fin de comprender los detalles de las tres aplicaciones. Esto ha permitido constatar que la implementación *MUMmerGPU++* diseñada para arquitecturas *ManyCore* puede llegar a tener hasta un rendimiento 10 veces superior a la implementación serie *MUMmer* para arquitecturas *MultiCore*.

El apreciado aumento de rendimiento obtenido en todos los workloads ejecutados por parte de la implementación *MUMmerGPU++*, sugiere que la ejecución de este tipo de aplicaciones en arquitecturas *ManyCore* es adecuada y puede suponer un camino a seguir en futuras implementaciones. Incluso pueden llegar a constituirse como una prometedora alternativa para algoritmos con el mismo tipo de perfil.

Debido a las diferencias de diseño que presentan las arquitecturas *ManyCore* y *MultiCore*, se ha comprobado que no es factible predecir que si una implementación se ejecuta eficientemente en CPU, entonces su equivalente en GPU se ejecutará también de tal forma. En este proyecto se ha podido constatar que los algoritmos y estructuras de datos utilizados en las aplicaciones *MUMmerGPU y MUMmerGPU++* para la indexación de textos, constituyen factores determinantes en el rendimiento de la aplicación, comprobando que una estructura de indexación de tipo *Suffix-Array* se adecua mejor a arquitecturas *ManyCore* que no un algoritmo basado en estructuras *Suffix-Tree*.

También se ha podido contrastar que bajo diferentes *workloads* característicos, la GPU tiene un rendimiento más eficiente. Por ejemplo, el uso de *queries* de gran tamaño beneficia la ejecución y permite una mayor eficiencia en el rendimiento frente al resto de implementaciones.

Los cuellos de botella encontrados no eran del tipo esperado. En base a la documentación disponible de *MUMmer*, se esperaba tener un programa limitado por cómputo y se ha podido demostrar que está limitado por el acceso a disco. Esto produjo un giro en la planificación del proyecto para el cual ya se había comenzado a estudiar una amplia documentación sobre optimizaciones en aplicaciones limitadas por cómputo en GPU.

7.1 Líneas abiertas

El proyecto prueba que el acceso a disco supone un gran problema y limita el alto cómputo que puede realizar la GPU. Por lo que un línea abierta que deja el proyecto es estudiar y optimizar este acceso a disco, a partir de técnicas de sintonización del sistema operativo o modificando el entorno de ejecución para hacerlo más adecuado.

Por otro lado, se ha comprobado que la parte de la implementación de *MUMmerGPU++* que realiza el cómputo no tiene aplicadas las optimizaciones más usuales en GPU. Por lo que otra línea abierta que deja el proyecto es estudiar el impacto que ofrecen las siguientes

optimizaciones: uso de memorias compartidas, accesos coalesced, reducción de la divergencia en código y técnicas de colaboración a nivel de threads. Esto requiere un análisis más profundo del código y alto conocimiento de las arquitecturas GPU.

Para ese análisis más profundo del código, sería interesante indagar más en la utilización de herramientas de *profiling*, que permiten realizar caracterizaciones de la aplicación con un gran nivel de detalle.

7.2 Problemas encontrados

En el desarrollo del proyecto han surgido diferentes problemas. Uno de ellos ya comentado ha sido la predisposición en el análisis de que el cuello de botella estuviese en la ejecución GPU. Esto provocó que se centrase la atención en la parte de GPU y se perdiese tiempo en el análisis de partes no tan decisivas en el rendimiento.

Otro problema encontrado fue a causa de la propia metodología utilizada en la medición de las aplicaciones, donde una vez generado el conjunto de datos de entrada se procedía a realizar la ejecución. Esto originaba que las lecturas de disco de las ejecuciones se realizasen desde la memoria principal del sistema aprovechando la localidad temporal de los datos. Esto provocaba la generación de resultados en tiempo menores a los reales. Así pues, se solucionó realizando una lectura de un fichero *dummy* entre ejecuciones, a fin de eliminar esta localidad temporal. Hasta que este hecho fue detectado se perdió mucho tiempo de análisis con datos de mayor fiabilidad.

Otros problemas ya encontrados están relacionados con la adaptación del código de los algoritmos a las últimas versiones de los SDKs o funciones *deprecated*, que generaban comportamientos inesperados en las aplicaciones compiladas. Debido a la poca documentación del código, esta etapa tuvo un gran impacto en el tiempo destinado al proyecto. Además, al encontrarse el código en repositorios abiertos al público, nos encontrábamos con revisiones en el código que no tenían la funcionalidad esperada o documentada en los artículos de investigación.

Para agilizar el proceso de lanzamiento de los experimentos se realizaron unos scripts para automatizar esta tarea. En un inicio el desarrollo de los scripts tomó un tiempo considerable del proyecto pero al final fue beneficioso en el sentido que permitió lanzar muchas ejecuciones de forma que se ahorraba tiempo y aumentaba la fiabilidad del proceso de toma de datos en la experimentación.

Debido a que el proyecto ha sido minuciosamente documentado, puede servir como base para futuros proyectos en el ámbito de la bioinformática comparativa de secuencias, proyectos utilizando GPGPU o proyectos basados en algoritmos de indexación de textos como los *Suffix-Tree*, *Suffix-Array* o *Enhanced Suffix-Array*. Además, el hecho de haber realizado un detallado análisis del algoritmo que utiliza la aplicación *MUMmer*, puede servir como punto de partida para futuras implementaciones o mejoras de éste, sin tener que dedicar una gran parte del tiempo a la documentación y estudio del algoritmo.

La metodología aplicada en el proyecto a este tipo de entornos heterogéneos con GPU y CPU puede servir como base y ayudar a otros futuros proyectos con un entorno de desarrollo similar.

7.3 Valoración personal

Personalmente considero que el cómputo GPGPU tiene un futuro prometedor y diversas aplicaciones muy especificas pueden verse muy beneficiado de él. El esfuerzo extra que debe realizar actualmente el diseñador de la aplicación, puede verse recompensado con aumentos en el speedup muy notorios, del orden de decenas.

En base a los grandes cambios de diseño que esta realizando Nvidia a sus dispositivos, considero que existe la posibilidad de que el futuro diseño de las arquitecturas ManyCores y su metodología de trabajo no sea exactamente como la estamos percibiendo en este momento.

Los actuales modelos de GPUs, aún permitiendo realizar cómputo de propósito general, incluyen recursos y funcionalidades en su diseño propias de los procesadores gráficos que poco o nada aportan al sector GPGPU. El trabajar actualmente con procesadores destinados a gráficos es posible que simplemente sea transición hacia la verdadera evolución de las arquitecturas ManyCore. Aún así, en este posible proceso de transición que estamos viviendo, es seguro que las actuales GPU realizan una aportación considerable en el diseño de los futuros procesadores ManyCore.

7.3 Desarrollo del proyecto

Debido a los problemas ya comentados en el desarrollo del proyecto, este se dilató y existe un desfase sobre la planificación original.

La entrega del proyecto se pospuso a septiembre con un incremento de unas 85h adicionales en el desarrollo. Esto supone un incremento en el coste final de alrededor 720€ si se parte de la base que la mano de obra de un ingeniero es de 9€/h.

Por lo que el coste final del proyecto asciende a 4865 aprox. Contando la desviación de los 720€.

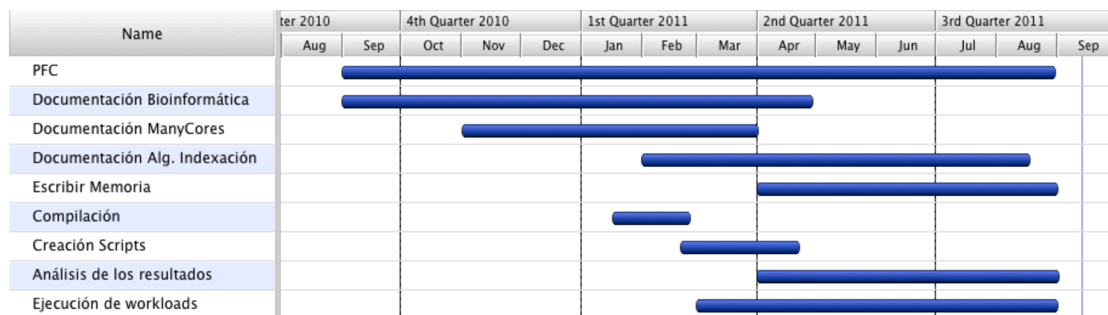


Figura 7.1: Diagrama de Gantt del desarrollo del proyecto, una vez concluido.

BIBLIOGRAFÍA

[1] - **Whole Genome Alignments and Synteny Maps**
Chong Shou

[2] - **Genomes**
T. A Brown

[3] - **Whole Genome Alignment**
Soleiman Oscar Markus Pello

[4] - **Compressed Full-Text Indexes**
Gonzalo Navarro, Veli Mäkinen

[5] - **The Enhanced Suffix Array and Its Applications to Genoma Analysis**
Mohamed Ibram Albouelda, Stefan Kurts, Enno Ohlebush

[6] - **Introduction to Computational Molecular Biology**
Athicha Muthitacharoen

[7] - **Enhanced Suffix Arrays**
Thierry Lecroq

[8] - **Replacing Suffix trees with Enhanced Suffix arrays**
Mohamed Ibram Albouelda, Stefan Kurts, Enno Ohlebush

[9] - **Fast algorithms for large-scale genome Alignment and comparison**
Artur L. Delcher, Adam Phillippy, Jane Carlton and Stevem L. Salzberg

[10] - **Optimizing data intensive GPGPU computations for DNA sequence Alignment**
Cole Trapnell, Michael C. Schatz

[11] - **Alignment of whole genomes**
Arthur L Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson

[12] - **Versatile and open software for comparing large genomes**
Stefan Kurtz, Adam Phillippy, Arthhur L Delcher, Michael Smoot

[13] - **Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance**
Abdullah Gharaibeh, Matei Ripeanu

[14] - **High-thoghput sequence Alignment using Graphics Processing Units**
Michael Schatz, Cole Trapnell, Arthur L Delcher, Amitabh Varshney

[15] - **Introduction to Massively Parallel Computing**
Nvidia

[16] - **CUDA Technical Training I**
Nvidia

[17] - **CUDA Technical Training II**
Nvidia

[18] - **Build environment Debugging/Profiling Fermi**
Nvidia

[19] - **Programming Massively Parallel Processors**
David B. Kirk, Wen Mei W. Hwu

[20] - **CUDA by Example**
Jason Sanders, Edward kandrot

[21] - **Parallel Performance Measurement of Heterogeneus Parallel Systems with GPU**
Alen D. Malony, Scott Biersdorff, Sameer Shende

[22] - **Debunking the 100x GPU vs CPU Myth: An evaluation of Throughput Computing on CPU and GPU**
Victor W Lee, Changkyu Kim, Jatin, Chhugani

[23] - **Analysis-Driven Optimization**
Pulius Micikevicius

[24] - **CUDA: Nvidia's GT200: Inside a parallel processor**
Jason Sanders, Edward kandrot

[25] - **Better Performance at Lower Occupancy**
Vasily Volkov

[26] - **CampProf: A visual Performance Analysis Tool for Memory Bound GPU kernels**
Ash M. Aji, Mayank Daga, Wu-chun Feng

[27] - **Demystifying GPU Microarchitecture through Microbenchmarking**
Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-alvandi, Andreas Moshovos

RESUMEN

Las herramientas de análisis de secuencias genómicas permiten a los biólogos identificar y entender regiones fundamentales que tienen implicación en enfermedades genéticas. Actualmente existe una necesidad de dotar al ámbito científico de herramientas de análisis eficientes.

Este proyecto lleva a cabo una caracterización y análisis del rendimiento de algoritmos utilizados en la comparación de secuencias genómicas completas, y ejecutadas en arquitecturas *MultiCore* y *ManyCore*. A partir del análisis se evalúa la idoneidad de este tipo de arquitecturas para resolver el problema de comparar secuencias genómicas. Finalmente se propone una serie de modificaciones en las implementaciones de estos algoritmos con el objetivo de mejorar el rendimiento.

RESUM

Les eines d'anàlisi de seqüències genòmiques permeten als biòlegs identificar i entendre regions fonamentals que tenen implicació en malalties genètiques. Actualment hi ha una necessitat d'aportar a l'àmbit científic eines d'anàlisi eficients.

Aquest projecte desenvolupa una caracterització i anàlisi del rendiment d'algoritmes utilitzats en la comparació de seqüències genòmiques completes executades en arquitectures *MultiCore* i *ManyCore*. A partir de l'anàlisi s'evalua la idoneïtat d'aquest tipus d'arquitectures per resoldre el problema de la comparació de seqüències genòmiques. Finalment es proposen una sèrie de modificacions en les implementacions d'aquests algoritmes amb l'objectiu de millorar el rendiment.

ABSTRACT

The analysis tools of the genomic sequence allow biologists to identify and understand the basic regions that are involved in genetic diseases. Nowadays there is the necessity to give the science efficiency analyse tools.

This project makes a characterisation and analysis of the output in the algorithms used on the complete sequence comparison, performed on MultiCore and ManyCore architectures. From this analysis the suitability of this kind of architectures on the solution of the comparison gene sequence is evaluated. Finally a series of modifications for the implementations of these algorithms are proposed, to allow the output improvement.